HTTPD for MVS 3.8j Reference

Document Number ML07-0002-0

October 2026

Mike Großmann

First Edition (October 2026)

This edition applies to Version 4 Release 5 of HTTPD, the HTTP server for MVS 3.8j (httpd 4.5.1), and to all subsequent releases and modifications until otherwise indicated in new editions.

Draft. This is a draft edition, as the last digit of the number says. The console sessions in it that show the version of the server were captured on an MVS/CE system, a throwaway container, with a package build of HTTPD 4.5.1 that was made from the source of the release branch, with the version in the project file changed from `4.5.1-dev` to `4.5.1`. The other console and client sessions may come from earlier development builds of HTTPD 4.5.0 and 4.5.1, which show the version `4.5.0-DEV` or `4.5.1-DEV`.

Comments on this book may be addressed to the issue tracker of the mvslovers/httpd repository on GitHub.

© Copyright Mike Großmann 2026. All rights reserved.

Contents

Figures . . . . . **v**

About This Book . . . . . **vii**
The Examples in This Book vii
How This Book Is Organized vii
Related Publications vii

Chapter 1. The HTTPPRM Keywords **1**
The Rules of the File 1
Listener and Connections 2
Worker Tasks and Time Limits 2
Files and Character Set 3
Authentication Values 4
Routes and Modules 5
Diagnostics and Recording 6
Keywords That Are No Longer Valid 6
Parameters That Are Not in the Member 7
What Stops the Start 7
Reading Back the Result 8

Chapter 2. Routes and Authentication Policy **9**
The Syntax of MOD and LOC 9
The Pattern Language 10
The First Match Wins 10
The Fixed Paths 11
Routes with an Extension 11
The Login Mode of a Route 11
What Each Mode Answers 12
The Resource Check 13
Limits 13
Messages While HTTPD Reads the Routes 14
Static Routes 15
Resident Modules 15
When a Module Does Not Run 16

Chapter 3. Console Commands **17**
Syntax 17
The Commands 17
DISPLAY CONFIG 18
DISPLAY LOGIN 19
DISPLAY STATS 19
DISPLAY THREADS 19
DISPLAY MEMORY 21
DISPLAY TIME 22
DISPLAY VERSION 22
SET MAXTASK and SET MINTASK 22
SET STATS 23
HELP and unknown commands 23

Stopping the server 24

Chapter 4. The Started Task and the Package 27
The Procedure 27
The symbolic parameters 27
The EXEC statement 28
The region 28
The DD Statements 29
The Parmlib member 30
The standard stream DDs 30
The JES2 data sets 31
The STEPLIB 32
Authorization 32
The Identity of the Task 33
What the Server Asks of the Security Product 34
The Start in Order 35
The listener 37
Return Codes and Abends 37
The Package 39
The FMID 39
The files in the archive 39
The data sets 40
The members 41
The document root disk 41

Chapter 5. HTTP Behaviour and Limits 43
How a Request Is Handled 43
Methods 43
Limits on the Request 45
The request line and the header lines 45
The headers 46
Parameters 46
The request body 46
URL Decoding and Escaping 47
Status Codes 47
Other methods on a route with form login 48
The phrase table 48
Response Headers 49
Keep-Alive and Time-outs 50
Keep-alive 50
Time-outs 51
A stalled send 51
Chunked Transfer 52
Static Files 52
The path 52
MIME types 53
Server-side includes 55
Character Translation 57
Workers and Threads 57
Other Hard Limits 58
The read of the request 58

Chapter 6. The Server Module Interface 59
How a Module Is Run 59
The Parameter List 59

What the Startup Code Does . . . 60
The Return Code . . . 60
A Minimal Module . . . 61
The Structures . . . 61
HTTPC, the Client . . . 61
HTTPV, a Variable . . . 63
HTTPM, a MIME Type . . . 63
The Request Variables . . . 63
The Function Vector . . . 65
The Rules . . . 65
The Entries . . . 65
Output . . . 68
Standard Error Responses . . . 69
Request Variables . . . 69
Translation . . . 70
Time, MIME Types and Addresses . . . 70
Files . . . 71
Dynamic Arrays . . . 71
Comparison . . . 72
Debug Output . . . 72
Authentication . . . 72
Sessions . . . 73
The Context Block . . . 74
Routes . . . 74
Entries for the Server Only . . . 74
The Direct Accessors . . . 75
Result Codes of a Link . . . 75
Storage Rules . . . 76
Long-Running Handlers and Shutdown . . . 77
Abends . . . 77
The Started Task Procedure . . . 78
Resident Modules . . . 79
Context Pointers . . . 79

Chapter 7. The Display Modules . . . 81
Setting Up the Modules . . . 81
Reading Storage . . . 82
HTTPDSRV, the Server Display . . . 82
Parameters . . . 82
Targets . . . 82
The Help Page . . . 89
HTTPDM, the Memory Display . . . 90
Parameters . . . 90
Output . . . 90
Limits . . . 91
HTTPDMTT, the Master Trace Table . . . 91
Parameters . . . 91
Output . . . 91
Limits . . . 91
ABEND0C1, the Abend Test . . . 92

Chapter 8. SMF Records . . . 93
Record Types and Subtypes . . . 93
The Header . . . 93

The Request Record 94
The Session Record 95
Levels and Parmlib Keywords 96
Reading the Records 96

Appendix A. Messages 99
A Healthy Start and Stop 99
A Start That Ends 100
Lifecycle and Identity (000 to 019) 101
Configuration and Start (020 to 048) 103
Listener, Workers and Stop (050 to 099) 107
Console Commands (100 to 199) 109
Routes, SMF and Resident Modules (410 to 444) 113
Diagnostics (901 to 999) 116
Lines Without a Number 118
Messages That Are Gone 118

Index 119

Figures

Tables

1. Keywords of the listener . . . 2
2. Keywords of the worker tasks and of the time limits . . . 2
3. Keywords of the files and of the character set . . . 3
4. Keywords of the authentication values . . . 4
5. Keywords of the routes and of the modules . . . 5
6. Keywords for diagnostics and for recording . . . 6
7. Retired keywords . . . 7
8. Causes of a refused start . . . 7
9. The route lines . . . 9
10. The parts of a route line . . . 9
11. The two kinds of pattern . . . 10
12. Examples of patterns . . . 10
13. The values of AUTH= . . . 11
14. What HTTPD answers to a request without a login . . . 12
15. The value of RES= . . . 13
16. Limits of the route lines . . . 13
17. Messages of the route lines . . . 14
18. Messages of the resident modules . . . 15
19. The reasons for a failure . . . 16
20. The MODIFY commands . . . 17
21. The messages of DISPLAY CONFIG . . . 18
22. The messages of DISPLAY THREADS . . . 20
23. The messages of DISPLAY MEMORY . . . 21
24. The messages of SET STATS . . . 23
25. The symbolic parameters of the procedure . . . 27
26. The PARM keywords . . . 28
27. What a module request costs . . . 28
28. A measurement of parallel module requests . . . 29
29. The DD statements of the procedure . . . 29
30. The messages for the standard stream DDs . . . 30
31. A request when the JES2 data sets cannot be opened . . . 31
32. The two ways to be authorized . . . 32
33. The messages for the identity . . . 33
34. The calls to the security product . . . 34
35. The return codes of the job step . . . 37
36. Messages with no effect on the return code . . . 38
37. The files in the archive . . . 39
38. The data sets of the package . . . 40
39. The steps of the install job . . . 40
40. The members of HTTPD.LINKLIB . . . 41
41. The members of HTTPD.SAMPLIB . . . 41
42. Dispatch by method . . . 43
43. Requests that the server rejects while it reads the headers . . . 45
44. Status codes that the server sends . . . 47

45. The status codes that the server can write . . . 48
46. The lines the server writes with every status line . . . 49
47. Defaults and ranges of the time-out keywords . . . 51
48. The MIME types of the server . . . 53
49. Server-side include directives . . . 56
50. Events in the socket thread and the workers . . . 57
51. Fixed limits of the server . . . 58
52. The parameter list of a module . . . 59
53. The fields of HTTPC . . . 61
54. The values of CSTATE . . . 62
55. The fields of HTTPV . . . 63
56. The fields of HTTPM . . . 63
57. The request variables . . . 64
58. The entries of the function vector . . . 65
59. The standard error responses . . . 69
60. The results of http_new_session . . . 73
61. The entries that belong to the server . . . 74
62. The result of http_link . . . 75
63. The texts of a failed module . . . 77
64. The DDs of the HTTPD procedure . . . 78
65. The messages of the resident load . . . 79
66. The query parameters of HTTPDSRV . . . 82
67. The targets of HTTPDSRV . . . 82
68. The fields shown for target HTTPD . . . 83
69. The fields shown for a route . . . 85
70. The words of the first line . . . 86
71. The words of a trace line . . . 86
72. The phases of a request . . . 87
73. The query parameters of HTTPDM . . . 90
74. The query parameter of HTTPDMTT . . . 91
75. The query parameter of ABEND0C1 . . . 92
76. The subtypes of the HTTPD record . . . 93
77. The 18-byte header, the same in both subtypes . . . 93
78. The request record, subtype 1 . . . 94
79. The session record, subtype 2 . . . 95
80. The recording levels . . . 96
81. Messages that end the start . . . 100
82. Messages 000 to 019 . . . 101
83. Messages 020 to 048 . . . 103
84. Messages 050 to 099 . . . 107
85. Messages 100 to 119 . . . 109
86. Messages 120 to 149 . . . 110
87. Messages 150 to 199 . . . 112
88. Messages 410 to 444 . . . 113
89. Messages 901 to 999 . . . 116

About This Book

This book describes HTTPD (httpd), the HTTP server for MVS 3.8j, in full: the keywords of the configuration member, the syntax of a route, the operator commands, the started task and the files of the package, the HTTP behaviour and the limits of the server, the interface that server modules call, the display modules, the SMF records and the messages. How to use them together is the subject of the companion volume, the *HTTPD User's Guide*.

The Examples in This Book

The console screens and the client sessions in this book are output that was recorded from a running HTTPD. None of them was typed by hand. Before they were printed, some values were replaced: passwords by asterisks, session tokens by **`<token>`**, the digits of job and task numbers by **`n`**, the names and addresses of the test systems by **`localhost`**, **`mvs.example.org`** and the address **`192.0.2.17`** of the documentation range, and the home directory of the user by **`$HOME`**. The addresses in the examples are therefore not real addresses.

How This Book Is Organized

Chapter 1	"The HTTPPRM Keywords".
Chapter 2	"Routes and Authentication Policy".
Chapter 3	"Console Commands".
Chapter 4	"The Started Task and the Package".
Chapter 5	"HTTP Behaviour and Limits".
Chapter 6	"The Server Module Interface".
Chapter 7	"The Display Modules".
Chapter 8	"SMF Records".
Appendix A	"Messages".

Related Publications

ML07-0001	*HTTPD User's Guide*: installing, configuring, running and removing HTTPD, routes, authentication, serving files, and writing a server module.

Chapter 1. The HTTPPRM Keywords

HTTPD reads its configuration once, at start, from the DD name **HTTPPRM**. This chapter lists every keyword that HTTPD accepts, with its default, its range and what HTTPD does with a value that is wrong. The route keywords **MOD** and **LOC** have a chapter of their own (Chapter 2, "Routes and Authentication Policy" on page 9). The *HTTPD User's Guide* explains how to write the member, in "Configuring HTTPD".

HTTPD reads the file only at start. It has no command to read it again, so a change takes effect at the next start. The command **F HTTPD,D CONFIG** shows what HTTPD made of the file (Chapter 3, "Console Commands" on page 17).

The Rules of the File

- The DD name is **HTTPPRM**. In the sample procedure it names one member of a PDS, and the procedure symbol **M** chooses the member (Chapter 4, "The Started Task and the Package" on page 27).
- A line has the form **KEY=value**. HTTPD splits the line at the first equal sign. Everything after it, including further equal signs, is the value.
- Blanks and tabs at both ends of the key and of the value are removed, so **PORT = 8080** is valid. A blank inside a value stays.
- The key may be in any case. HTTPD changes it to upper case. HTTPD does not change the case of a value. The tables below say where HTTPD compares a value without regard to case.
- An empty line is ignored. A line is a comment if its first character after the leading blanks is **\#** or *****. A comment takes the whole line. HTTPD knows no comment at the end of a line: the text after a value belongs to the value.
- HTTPD reads at most 255 characters of a line at a time. It takes the rest of a longer line as the next line. There is no continuation line.
- A line without an equal sign gives the message **HTTPD026W** with the first 40 characters of the line. HTTPD goes on.
- A key that HTTPD does not know gives **HTTPD029W** with the key in upper case. HTTPD goes on.
- HTTPD writes these messages to the console.
- If a key is given twice, the last line counts. There are two exceptions. Each **MOD** line and each **LOC** line adds a route, and HTTPD does not look for a duplicate (Chapter 2, "Routes and Authentication Policy" on page 9). A second **SMF** line without **TYPE=** keeps the type of the first.
- If the DD name is missing, HTTPD writes **HTTPD020W** and starts with the defaults. The defaults contain no route.
- If a read error ends the read before the end of the member, HTTPD writes **HTTPD038E** and stops with CC 0008. HTTPD does not start on half a member.

A value that is not right gets one of four treatments, and the tables below say which one applies. HTTPD *keeps* the value that was valid before, either the default or an earlier line. HTTPD *limits* the value to the nearest value that is allowed. HTTPD *reads* a text that is not a number as 0, because it converts a number with the C function **atoi**: it takes the digits at the start of the value and ignores the rest. HTTPD *refuses* the value and the start. Only some of these cases write a message.

Listener and Connections

Table 1. Keywords of the listener

Keyword	Default	Rules
PORT	8080	The TCP port on which HTTPD listens. HTTPD has no keyword for an address: it listens on every address of the host. A number below 1 or above 65535, and a text that is not a number, give **HTTPD023E** with the number that HTTPD read (0 for a text). The earlier value stays.
BIND_TRIES	10	How many times HTTPD tries again to bind the port when the bind fails because the port is in use. HTTPD limits the value to 1 to 100.
BIND_SLEEP	10	The seconds that HTTPD waits before each of these tries. HTTPD limits the value to 1 to 100.
LISTEN_QUEUE	5	The length of the queue of connections that wait to be accepted. HTTPD limits the value to 1 to 100.

Before HTTPD binds the port, it takes an ENQ with the major name **HTTPD** and the minor name **HTTPD.PORT.nnnnn**, where **nnnnn** is the port with five digits. If another address space holds the ENQ, or the ENQ fails, HTTPD writes **HTTPD037E** and ends with CC 0008. A second HTTPD on another port starts.

If the bind fails with “address in use”, HTTPD closes every socket of the stack that has this port as its local port (**HTTPD027I** for each one). If it closed at least one, it waits 2 seconds and binds again. If the bind still fails, HTTPD writes **HTTPD030E**. Then it writes **HTTPD030I** and waits **BIND_SLEEP** seconds before each of at most **BIND_TRIES** further tries. If all tries fail, HTTPD writes **HTTPD030E** again and ends with CC 0008. Any other bind error ends the start at once. The same holds for an error of **socket** (**HTTPD028E**) and of **listen** (**HTTPD031E**). When the listener is ready, HTTPD writes **HTTPD054I**.

Worker Tasks and Time Limits

Table 2. Keywords of the worker tasks and of the time limits

Keyword	Default	Rules
MINTASK	3	The least number of worker tasks. A value of 0 or less, and a text that is not a number, are ignored without a message: the earlier value stays. HTTPD limits a larger value to 255. If the value is above **MAXTASK**, HTTPD sets **MAXTASK** to the same value.
MAXTASK	9	The most worker tasks. The rules for a wrong value are those of **MINTASK**. At the end of the start HTTPD raises **MAXTASK** to the value of **MINTASK** if it is lower, whatever the order of the two lines.
CLIENT_TIMEOUT	10	The seconds that HTTPD waits for the client while it reads one line of a request: the request line or a header line. The time starts again for each line. A line of a request that comes on a connection that has served a request before uses **KEEPALIVE_TIMEOUT** instead. A value of 0 or less, and a text

		that is not a number, are ignored without a message. HTTPD limits a larger value to 255. When the time is up, HTTPD closes the connection.
KEEPALIVE_TIMEOUT	5	The seconds that HTTPD waits for the lines of the next request on a connection that stays open. HTTPD limits the value to 1 to 255.
KEEPALIVE_MAX	100	The number of requests a connection carries before HTTPD closes it. HTTPD answers the request after that number with **Connection: close**, so a connection carries the value plus one request. HTTPD limits the value to 1 to 255.
CLIENT_TIMEOUT_MSG	1	A number. A value above 0 means yes, any other value and a text that is not a number mean no. If yes, HTTPD writes one line to the console when a time limit ends while the client has sent at least one character of the line. The line has no message ID. It reads **client xxxxxxxx on socket n timed out after s seconds** and gives the address of the client block in hex, the socket and the seconds.
CLIENT_TIMEOUT_DUMP	1	The same rules. If yes, HTTPD also writes a dump of the characters received so far, with the title **Receive Buffer**, at such a time-out.
CLIENT_STATS	1	The same rules. HTTPD stores the flag and the display module **HTTPDSRV** shows it as **STATS**. No other code of HTTPD reads it, so it changes nothing that HTTPD does or writes.

The time limits count in whole seconds. **D CONFIG** shows the values of **MINTASK**, **MAXTASK**, **CLIENT_TIMEOUT**, **KEEPALIVE_TIMEOUT** and **KEEPALIVE_MAX** (Chapter 3, “Console Commands” on page 17).

Files and Character Set

Table 3. Keywords of the files and of the character set

Keyword	Default	Rules
DOCROOT	/www	The UFS directory that HTTPD puts in front of the path of a request when it serves a static file. The value must begin with a slash. Another value is ignored without a message, and the earlier value stays. HTTPD removes one slash at the end of a value that is longer than one character. HTTPD keeps at most 127 characters.
UFS	1	A number. A value above 0 means yes, any other value and a text that is not a number mean no. If yes, HTTPD creates its UFS system at start. If that fails, HTTPD writes **HTTPD044W** and goes on without it. If no, HTTPD does not create the UFS system and serves no file from it. Leave it at the default.
CODEPAGE	CP037	The table that translates between the ASCII on the wire and the EBCDIC in HTTPD. The names are **CP037**, **IBM1047** and **LEGACY**, in any case. **LEGACY** is the table of HTTPD 3.3.x. An

	empty value means **CP037**. Another name gives **HTTPD070E**, and HTTPD uses **CP037**. HTTPD keeps at most 15 characters of the value as written; **D CONFIG** shows them.

Authentication Values

Which route needs a login is a property of the route, set with **AUTH=** (Chapter 2, "Routes and Authentication Policy" on page 9). There is no keyword that sets a login policy for all routes. Three of the four keywords here set the name of the server and the life of a login. The fourth, **RES_NOPROFILE**, sets what a route with **RES=** does when the security product has no profile for its resource.

Table 4. Keywords of the authentication values

Keyword	Default	Rules
REALM	SMF ID	The name of the server. It is the realm in the **WWW-Authenticate: Basic** header and part of the login form. The default is the SMF ID of the system: up to four characters, ending at the first blank. If the system has none, the default is **MVS**. A value has 1 to 64 characters. It must not contain a control character or one of the characters double quote, backslash, **<**, **>** and **&**. Another value gives **HTTPD424W** with the first 40 characters, and the default stays (or the value of an earlier line).
SESSION_TIMEOUT	30	The minutes after which HTTPD drops the login of a client that is idle. HTTPD reads the value as a number. A negative number and a text that is not a number give 0, a number above 65535 gives 65535. A value of 0 switches this limit off.
SESSION_MAXAGE	480	The minutes after which a login ends whatever the client does, counted from the login. Activity does not renew it. The rules for a wrong value are those of **SESSION_TIMEOUT**. A value of 0 switches this limit off.
RES_NOPROFILE	ALLOW	**ALLOW** or **DENY**, in any case. The value ends at the first blank. It decides what a route with **RES=** does when no profile covers its resource. With **ALLOW** the request passes, and the start writes **HTTPD425W**. With **DENY** the request gets status 403, and the start writes **HTTPD426W**. Any other value is fatal: HTTPD writes **HTTPD427E** with the first 16 characters of the value and then **HTTPD420E**, and the start ends with CC 0008.

A login that a request is using must not be dropped. HTTPD therefore compares the two limits with **CLIENT_TIMEOUT** at the end of the start. If **SESSION_TIMEOUT** in seconds is not larger than **CLIENT_TIMEOUT**, HTTPD writes **HTTPD026E**. If **SESSION_MAXAGE** in seconds is not larger, HTTPD writes **HTTPD032E**. Both messages are warnings: HTTPD starts. A limit of 0 is not compared. The server drops logins about once a minute, if at least one of the two limits is above 0.

Note: The messages **HTTPD026W** and **HTTPD026E** have the same number and different meanings. The first is a line that is not a keyword line. The second is the check of **SESSION_TIMEOUT**.

The rules of **RES_NOPROFILE**:

- **One value for the whole server.** The keyword is not an option of a route. It applies to every route with **RES=**, except the routes with **AUTH=NONE**, which HTTPD does not check (“The Login Mode of a Route” on page 11).
- **The check is made for each request.** With **DENY** a request to a route whose resource has no profile gets status 403 and the body **403 Forbidden: no RACF profile protects this resource**. The class and the resource name are not sent. A request that a profile refuses gets the plain **403 Forbidden**. A request without a login gets the 401 of the first stage, as before.
- **A profile that is defined later needs no restart.** The check looks at the profile as the security product holds it at the time of the request. Under RAKF the product holds a new profile after the started task **RAKFPROF** has run (“RES=: Checking a Profile” in the *HTTPD User’s Guide*). Until then the route answers 403 with the body above. **HTTPD426W** is written at the start only, so it stays in the log.
- **HTTPD sees only the return code 4.** The security product answers 4 when no profile covers the resource. HTTPD cannot tell why the product had none to apply, and with **DENY** every return code 4 refuses. This book does not state in which conditions, other than a missing profile, a product answers 4.
- **The default does not change anything.** Without the keyword, or with **ALLOW**, the check passes the return codes 0 and 4 and refuses every other one, as in the releases before 4.5.0.
- **A misspelled value is fatal on purpose.** HTTPD does not fall back to **ALLOW**. A keyword that exists to close a gate must not leave the gate open when it is misspelled.

Routes and Modules

Table 5. Keywords of the routes and of the modules

Keyword	Default	Rules
MOD	none	A route to a program: **MOD=program pattern AUTH=mode RES=class:resource** . Each line adds one route. The whole syntax is in Chapter 2, “Routes and Authentication Policy” on page 9.
LOC	none	A route without a program, for a path that HTTPD serves as a static file but with a login: **LOC=pattern AUTH=mode RES=class:resource** . See Chapter 2, “Routes and Authentication Policy” on page 9.
RESIDENT	YES	**YES** or **NO**, in any case. Text after the first blank is ignored. With **YES** HTTPD holds each module of a **MOD** route from the start to the end, so a request does not fetch it from the library. With **NO** each request fetches the module, so a new level of a module is used at once. Another value gives **HTTPD443W** and the earlier value stays. Chapter 2, “Routes and Authentication Policy” on page 9 describes what HTTPD does at start.
CGI_CONTEXT_POINTERS	255	The size of the table in which modules register their context blocks: the table has this value plus one entries. HTTPD reads the value as a number. A negative number is ignored without a message. A number above 255 gives 255. A text that is not a number gives 0. A module gets a block through the function **http_cgictx_get** (Chapter 6, “The Server Module Interface” on page 59).
CGI	none	The old name of **MOD**. HTTPD writes **HTTPD410W** for each line and then reads the line as a **MOD** line. Use **MOD**.

Diagnostics and Recording

Table 6. Keywords for diagnostics and for recording

Keyword	Default	Rules
DEBUG	0	A number. A value above 0 switches the trace on, any other value and a text that is not a number switch it off. There is only one level: 5 means the same as 1. With the trace on, HTTPD opens the DD name **HTTPDBG** for output and writes its trace lines to it. If the DD name cannot be opened, HTTPD writes **HTTPD020E** with the **errno** and **HTTPD021I**, and the trace goes to the DD name **SYSTERM**. If a write to the trace fails, HTTPD writes **HTTPD039W** once and switches the trace off until the next start. The server goes on.
TRACE	0	The number of records of the request trace: 0 or a number from 16 to 1024. 0 is off. The request trace is not the trace of **DEBUG**. It keeps, for each request, the worker that serves it, the time of each phase of the request and a few counters, in a table of the server. It does not change what the server does. The target **TRACE** of **/.dsrv** shows the table (“Target TRACE” on page 86). HTTPD reads the value at the start, so a change takes a restart. Any other value, and a text that is not a number, give **HTTPD444W** and the trace stays off. With a valid number HTTPD takes one block of storage from the region of the server, 32 bytes plus 136 bytes for each record (34,848 bytes for 256 records, 139,296 bytes for 1024), and writes **HTTPD066I**. If the region has no room for the block, HTTPD writes **HTTPD067W** and the server runs without the trace. The server frees the block at the stop. The table holds the most recent requests: when it is full, a new request takes the record of a finished one. Switch the trace on to find a problem and off after it. The table holds the client address and the path of every request.
SMF	NONE	**SMF=level** or **SMF=level TYPE=nnn**. The level is **NONE**, **ERROR**, **AUTH** or **ALL**, in any case. HTTPD compares only as many characters as the word has, so **ERR** is read as **ERROR**. Another word, and an empty value, give **HTTPD431W** and the earlier level stays. **TYPE=** is the SMF record type, 128 to 255, default 243. HTTPD looks for it anywhere in the value, in upper case only. Another number gives **HTTPD432W** and the earlier type stays. Chapter 8, “SMF Records” on page 93 describes the records.

Keywords That Are No Longer Valid

These keywords are known to HTTPD, and each gives a message. A member from an earlier release starts, except in one case.

Table 7. Retired keywords

Keyword	What HTTPD does
LOGIN	A global rule for a login has no place any more: the login is set per route with **AUTH=**. An empty value, and a value that consists only of **NONE** (several words are separated by commas), give **HTTPD048W**. HTTPD ignores the line and the start goes on. Every other value (**ALL**, **CGI**, **GET**, **HEAD**, **POST**, and a word that HTTPD does not know) gives **HTTPD048E** with the value, and the start ends with **HTTPD420E** and CC 0008. Such a value required a login for every route without a policy of its own, and these routes would be open without the refusal. HTTPD does not turn the value into route policies: change the routes.
TZOFFSET	HTTPD ignores the value and writes **HTTPD025W** with the offset that applies, and **HTTPD025I**. There is no keyword for the time zone. The offset comes from the variable **TZ**, or from the time zone of the system if there is none. To choose another offset, set **TZ** in the DD name **SYSENV** or **ENVIRON** of the procedure.
RECLAIM=	This is an option of a **MOD** route, not a keyword. HTTPD ignores the value and writes **HTTPD422W** for each use. The storage of a module is always released after the request.

Note: A line **LOGIN ALL** without an equal sign is not a **LOGIN** line for HTTPD. It is a line without an equal sign, and it gives **HTTPD026W** only. The start goes on, and the routes keep the policy that they have.

Parameters That Are Not in the Member

Three names look like keywords but are not. HTTPD reads **STCUSER=** and **STCGROUP=** from the **PARM** of the procedure, and not from the member. Chapter 4, “The Started Task and the Package” on page 27 describes them. A line **STCUSER=HTTPD** in the member gives **HTTPD029W**, as for any unknown key.

The third name is **CONFIG=**. It has no effect. HTTPD writes **HTTPD024W** and **HTTPD024I**, which says to choose the member with **S HTTPD,M=member**.

What Stops the Start

HTTPD ends with CC 0008 and without a listener in these cases. Every other problem in the member gives a message, and the start goes on.

Table 8. Causes of a refused start

Message	Cause
HTTPD038E	A read error before the end of the member.
HTTPD420E	A route lost its policy: **HTTPD419E** (a route that could not be registered), **HTTPD411E** (an unknown **AUTH=** value) or **HTTPD048E** (a retired **LOGIN** that required a login). Also **HTTPD427E**, a value of **RES_NOPROFILE** that is neither **ALLOW** nor **DENY**. HTTPD reads the whole member first, so it writes all of these messages before it stops. It stops before it opens the port. Chapter 2, “Routes and Authentication Policy” on page 9 has the details.
HTTPD037E	Another address space already serves the port.

HTTPD028E, **HTTPD030E**, **HTTPD031E**	The stack refused the socket, the bind or the listen.

Reading Back the Result

The command **F HTTPD,D CONFIG** writes the values that HTTPD settled on, and the name of the member that it read (Chapter 3, “Console Commands” on page 17). It shows the port, the tasks, the time limits, the two login limits, the document root, the code page, the realm, the SMF type and level, the state of **RESIDENT** and the value of **RES_NOPROFILE**. It does not show the other keywords. The display module **HTTPDSRV** shows the whole control block of the server, including those (Chapter 7, “The Display Modules” on page 81).

Chapter 2. Routes and Authentication Policy

A route ties a path of a request to what HTTPD does with it. A route is written in the HTTPPRM member (Chapter 1, "The HTTPPRM Keywords" on page 1) as a **MOD** line or a **LOC** line. A **MOD** route names a program, a server module that HTTPD links to. A **LOC** route has no program: HTTPD serves the path as a static file. Both can carry a policy, which says who may use the route. This chapter is the full description of the syntax, of the matching and of the policy. The *HTTPD User's Guide* shows how to build routes for common cases, in "Routes and Server Modules" and in "Authentication and Security".

The Syntax of MOD and LOC

Table 9. The route lines

Line	Form
MOD	**MOD=program [pattern] [AUTH=mode] [RES=class:resource]**
LOC	**LOC=pattern [AUTH=mode] [RES=class:resource]**

Table 10. The parts of a route line

Part	Rules
program	The name of the load module. The first token after the equal sign. HTTPD changes it to upper case and keeps at most 8 characters. A longer name is cut without a message. A module name is at most 8 characters on MVS, so the cut name is the one that HTTPD links to. Required for **MOD**.
pattern	The path that the route claims. See "The Pattern Language" on page 10. For **MOD** it is the second token, and it is optional. If it is left out, HTTPD derives it from the program name ("Routes with an Extension" on page 11). For **LOC** it is the first token and it is required. A token that starts with **AUTH=**, **RES=** or **RECLAIM=**, in any case, is an option and never a pattern.
AUTH=*mode*	The login that the route needs: **NONE**, **FORM**, **BASIC** or **TOKEN**, in any case. See "The Login Mode of a Route" on page 11.
RES= *class:resource*	A RACF resource that the user must be allowed to read. See "The Resource Check" on page 13.
RECLAIM=*x*	Retired. HTTPD ignores it and writes **HTTPD422W**. The storage of a module is always released after the request.

The rules for the line as a whole:

- The tokens are separated by blanks or tabs. The options may come in any order, but after the pattern.
- HTTPD reads at most 8 tokens. It ignores the rest of the line without a message.
- A token that is not a pattern and not one of the options above gives **HTTPD413W** with the token, and HTTPD ignores it. The route is still registered. See the warning in "Messages While HTTPD Reads the Routes" on page 14.

- An option may stand twice. For **AUTH=** the last valid mode counts, but a mode that is not valid refuses the route whatever follows. For **RES=** the last well-formed value counts.
- Each line adds one route. HTTPD does not check for a route that is there already, and it sets no limit on the number of routes.
- **CGI=** is the old name of **MOD=**. HTTPD writes **HTTPD410W** and reads the line as a **MOD** line.

The member may contain these lines:

```
MOD=MVSMF /zosmf/restconsoles/*   AUTH=TOKEN RES=FACILITY:MVSMF.CONSOLE
MOD=MVSMF /zosmf/*                AUTH=TOKEN
MOD=HTTPDSRV /.dsrv               AUTH=FORM
LOC=/admin/*                      AUTH=BASIC RES=FACILITY:HTTPD.ADMIN
LOC=/*                            AUTH=NONE
```

The Pattern Language

HTTPD compares the path of the request with the pattern of each route. The path is the part of the URI before the first question mark. HTTPD decodes it first: each **%xx** becomes the character. A plus sign stays a plus sign (“URL Decoding and Escaping” on page 47). The query string never takes part in the match.

A pattern is a wildcard pattern if it contains ***** or **?**. Any other pattern is a plain path.

Table 11. The two kinds of pattern

Kind	Rules
Plain path	The path of the request and the pattern must be equal. HTTPD ignores the case of the letters. **/status** matches **/STATUS**. It does not match **/status/**.
Wildcard pattern	HTTPD matches the whole path against the pattern with the library function **__patmat**. The case of the letters counts. ***** stands for any characters, also a slash. At the end of the pattern it also stands for no character. Anywhere else it stands for at least one character. **?** stands for one character. Any other character stands for itself.

Some examples:

Table 12. Examples of patterns

Pattern	Matches	Does not match
/zosmf/*	**/zosmf/info**, **/zosmf/a/b**	**/zosmf**, **/ZOSMF/info**
/.dsrv	**/.dsrv**, **/.DSRV**	**/.dsrv/x**
***.rexx**	**/a/b.rexx**	**/a/b.REXX**, **.rexx**
/a*b	**/axb**, **/a/b**	**/ab**
/doc?.html	**/doc1.html**	**/doc12.html**

A prefix therefore needs a trailing *****. The pattern **/admin** claims only that one path.

The First Match Wins

HTTPD keeps all routes, **MOD** and **LOC** together, in one table in the order of the lines. For a request it walks the table from the top and uses the first route that matches. It never looks for the best match,

and it does not test prefixes before extensions. The order of the lines is the only thing that sets the priority.

Two consequences follow.

- A route with a general pattern hides every route below it that it also matches. The line **LOC=/*** before **MOD=MVSMF /zosmf/*** sends every request to the **LOC** route. HTTPD gives no message.
- The policy of the first route is the whole policy of the request. The policy of a later route that would match too does not apply.

Write the specific routes first and the general ones last.

A request that no route matches is public. HTTPD serves it as a static file ("Static Routes" on page 15).

The Fixed Paths

The paths **/login** and **/logout** belong to HTTPD. A request for one of them goes to the login code of HTTPD whatever the routes say and whatever the method is, and no policy applies. The match ignores the case of the letters and needs the whole path. Do not give a route to these two paths.

The paths that match **/login.*** and **/favicon.*** are exempt from the policy for the methods GET and HEAD, so that the login page can load its own parts. The case of the letters counts here. The route still applies: HTTPD only skips the check of the login. A **MOD** route that claims such a path runs its module without a login.

Routes with an Extension

If a **MOD** line has no pattern, HTTPD builds one from the program name: ***.** followed by the program name in lower case. The line **MOD=REXX** becomes the pattern ***.rexx**.

A pattern that starts with ***.** makes an extension route, whether HTTPD built it or the line states it. For a request on an extension route HTTPD sets the variable **SCRIPT_FILENAME** to the document root (**DOCROOT**, Chapter 1, "The HTTPPRM Keywords" on page 1) followed by the path of the request. If HTTPD cannot set the variable, it answers with an internal error (status 500) and does not link to the module. No other route gets this variable from HTTPD.

The pattern is a wildcard pattern, so the match is case sensitive: ***.rexx** does not match **/a/b.REXX**. The position in the table counts as for any other route.

The Login Mode of a Route

Each route has one of four modes. The mode says whether the route needs a user, and how HTTPD asks a client that has none.

Table 13. The values of AUTH=

Mode	Rules
NONE	The route is public. HTTPD does not check a login. This is also the mode of a route with no **AUTH=** and no **RES=**.
FORM	The route needs a login. A client without one gets the HTML login form.
BASIC	The route needs a login. A client without one gets status 401 with a Basic challenge.
TOKEN	The route needs a login. A client without one gets status 401 and no challenge.

The rules for a missing or a wrong mode:

- A line with **RES=** and without **AUTH=** gets the mode **BASIC**, because a resource check needs a user.
- A line with **AUTH=NONE** and with **RES=** gives **HTTPD414W**. The mode **NONE** wins, and HTTPD does not check the resource.
- A mode that is not one of the four gives **HTTPD411E** with the first 24 characters of the value. HTTPD refuses the route, and the start ends ("Messages While HTTPD Reads the Routes" on page 14).

The mode does not choose the way the client proves who it is. Before HTTPD looks for the route, it tries these sources in this order and uses the first one that gives a user.

1. The cookie **Sec-Token**, if it holds a valid session token.
2. The cookie **LtpaToken2**, if it holds a valid session token.
3. The header **Authorization: Bearer** with a session token.
4. The header **Authorization: Basic** with user and password.

If neither cookie gives a user, HTTPD reads the **Authorization** header. A Bearer header ends the search, also when its token is not valid. A header that is neither Bearer nor Basic gives no user.

The login is bound to the address of the client. A user that logged in from another address does not count. Every route accepts every source, whatever its mode. The mode only says what HTTPD sends to a client that has no login.

What Each Mode Answers

HTTPD runs the check in two stages, before it serves a file or links to a module. The first stage asks for a user. The second stage applies only if the route has **RES=**. It asks RACF whether the user may read the resource. The two stages apply to **MOD** routes and to **LOC** routes in the same way, for the methods GET, HEAD, PUT, POST and DELETE.

Table 14. What HTTPD answers to a request without a login

Mode	Answer
NONE	None. HTTPD serves the request.
FORM	A GET gets status 302 with **Location: /login**. The cookie **Sec-Uri** holds the URI of the request, encoded in base64, and the form sends the user back to it after the login. A POST is read as a login attempt: the fields **userid** and **password** of the body are the user. A request with another method gets status 200 and a text that says **Invalid REQUEST_METHOD**.
BASIC	Status 401 with the headers **WWW-Authenticate: Basic**, **Cache-Control: no-store** and **Content-Type: text/plain**. The body is the line **401 Unauthorized**. The name **realm="name"** is the value of **REALM** (Chapter 1, "The HTTPPRM Keywords" on page 1). If the request has the header **X-CSRF-ZOSMF-HEADER**, whatever its value, HTTPD leaves out the **WWW-Authenticate** header.
TOKEN	The same as **BASIC**, but HTTPD never sends the **WWW-Authenticate** header, whatever the request holds.

A request with a login, to a route with **RES=**, goes to the second stage. If the user may not read the resource, HTTPD answers with status 403, the headers **Cache-Control: no-store** and **Content-Type: text/plain**, and the body **403 Forbidden**. Otherwise HTTPD goes on with the request.

The Resource Check

Table 15. The value of RES=

Rule	Detail
Form	**RES=class:resource**. HTTPD splits the value at the first colon. The class is the text before it and the resource is the rest, so the resource may contain colons. HTTPD keeps the case as written.
Access asked	READ.
A wrong value	A value without a colon, and a value with nothing after the colon, give **HTTPD412W** with the value. HTTPD ignores the option and registers the route. Without an **AUTH=** the route is then public.
Storage	If HTTPD cannot get storage for the value, it writes **HTTPD418E**. The route is refused, and the start ends.
Two **RES=**, one line	The last well-formed value counts.
With **AUTH=NONE**	HTTPD ignores the resource. See “The Login Mode of a Route” on page 11.
Resource without a profile	HTTPD counts the return code 4 of the check, “no profile covers the resource”, as permitted, and the return code 0 as well. Every code of 8 and above is a refusal. If the security product answers a resource without a profile with 4, nothing is checked in fact. RAKF answers 4. Whether another product does depends on its settings, not on HTTPD; “What the Server Asks of the Security Product” on page 34 has the detail. If it answers with 8 or above, the user gets the 403, and HTTPD writes no **HTTPD425W**. RAKF 2.0.0 (MVS/CE) answers a class that it does not define with 8. A mistake in the resource name, and for RAKF 1.2.x (measured on TK5) a class that is not defined, looks like a missing profile, so HTTPD checks every route with a resource once at the start, and writes **HTTPD425W** for each one that has no profile. The message names the class, the resource and the pattern of the route. HTTPD does not check routes with the mode **NONE**. The keyword **RES_NOPROFILE=DENY** changes the first part: a request to a route whose resource has no profile then gets status 403, and the start writes **HTTPD426W** instead of **HTTPD425W** (“Authentication Values” on page 4).
Denied	Status 403.

Limits

Table 16. Limits of the route lines

Limit	Value
Characters read from one line	255 (“The Rules of the File” on page 1).
Tokens read from one line	8. HTTPD ignores the rest without a message.
Characters of a program name	8. HTTPD cuts a longer name without a message.

Number of routes	No limit from HTTPD. A route that cannot be added for lack of storage gives **HTTPD035W** or **HTTPD423W**.
Duplicates	HTTPD does not look for them. The first of two equal routes hides the second.
Reading the member	Once, at the start. A change of a route needs a new start.

Messages While HTTPD Reads the Routes

HTTPD reads the whole member before it decides whether to start. The messages below come while it reads. They go to the console.

Table 17. Messages of the route lines

Message	Start	Meaning
HTTPD036I	goes on	A **MOD** route is registered. The message gives the program and the pattern.
HTTPD417I	goes on	A **LOC** route is registered.
HTTPD410W	goes on	A line used the old name **CGI=**.
HTTPD421W	see below	A **MOD** line without a program name. An option stands where the name belongs, or the value is empty.
HTTPD415W	see below	A **LOC** line without a path. An option stands where the path belongs, or the value is empty.
HTTPD411E	ends	An **AUTH=** value that is not valid. HTTPD refuses the route, and writes **HTTPD419E**.
HTTPD412W	goes on	A **RES=** value without class and resource. HTTPD ignores it.
HTTPD413W	goes on	A token that is not an option. HTTPD ignores it.
HTTPD414W	goes on	**AUTH=NONE** with **RES=**. The resource is not checked.
HTTPD418E	ends	No storage for **RES=**. HTTPD writes **HTTPD419E** too.
HTTPD422W	goes on	A line used **RECLAIM=**. HTTPD ignores it.
HTTPD035W	see below	HTTPD could not register a **MOD** route. The message gives the program and the pattern.
HTTPD423W	see below	HTTPD could not register a **LOC** route.
HTTPD419E	ends	A route that asked for a login or a resource is lost. The message gives the kind and the pattern.
HTTPD420E	ends	The end of the check, see below.

A route that asked for a login or for a resource, and that HTTPD cannot build, is never dropped quietly. Without the route the path would be public. HTTPD therefore writes **HTTPD419E** for each such line. When it has read the whole member, it writes **HTTPD420E**, ends with CC 0008 and does not open the port. These lines belong to this case:

- A line with an **AUTH=** value that is not valid.
- A line with **RES=** for which HTTPD gets no storage.
- A line in which an option stands where the program name or the path belongs. An example is **LOC=AUTH=BASIC RES=FACILITY:HTTPD.ADMIN**. For such a line **HTTPD419E** gives the whole line.

HTTPD checks the options to decide: a line that asks for a login or a resource ends the start. A line with only **AUTH=NONE** or without options gives **HTTPD421W** or **HTTPD415W** and the start goes on.
- A line for which HTTPD gets no storage to split it into tokens.
- A route with a login or a resource that HTTPD cannot add to the table.

A route with the mode **NONE**, and a route without any policy, that HTTPD cannot register give a warning (**HTTPD035W**, **HTTPD423W**) and the start goes on. Losing them cannot make the path less protected than the line asked.

Note: HTTPD checks only the words it knows. A line like **MOD=ADMIN /admin/* AUTH BASIC** has the tokens **AUTH** and **BASIC**. HTTPD writes **HTTPD413W** twice and registers a public route. Look for **HTTPD413W** in the log after each change of the member. The same holds for **RES=** with a wrong value and without **AUTH=**.

Static Routes

A **LOC** route has no program. The two stages of the check apply as described above. If the request passes, HTTPD serves the path as a static file from the document root (**DOCROOT**, Chapter 1, "The HTTPPRM Keywords" on page 1), for the methods GET, HEAD, PUT, POST and DELETE. A request that no route matches is served the same way, without a check.

A **LOC** route is the way to protect a part of the document tree. The route does not change what HTTPD serves. It decides who may get it.

Three rules help:

- A route with the mode **NONE** and a general pattern, as in **LOC=/* AUTH=NONE**, claims every path that is not claimed above it. It is a good last line. As a first line it hides all that follows.
- A pattern that ends in **/*** does not claim the directory itself. Add a plain-path route for it.
- The login page lives in the document tree. Its paths **/login.*** are exempt ("The Pattern Language" on page 10), so a **LOC=/* AUTH=FORM** route still lets the page load.

Resident Modules

By default (**RESIDENT=YES**, Chapter 1, "The HTTPPRM Keywords" on page 1) HTTPD loads each module of a **MOD** route once, after it has read the member and before it serves any request. It holds the copy until the end. A request then finds the module in storage and does not fetch it from the library. HTTPD does this once for each program name, however many routes use the name. The comparison of the names ignores the case.

A module that HTTPD does not hold still works. Each request fetches it, as with **RESIDENT=NO**. The start shows why in a message.

Table 18. Messages of the resident modules

Message	Meaning
HTTPD440W	The module is not found at the start. HTTPD does not hold it. If it is there later, the route works.
HTTPD441W	The module is not reentrant and not serially reusable. A copy that is held could run twice, so HTTPD does not hold the module.
HTTPD442W	The load failed. The message gives the abend code and the reason code.

A module that HTTPD holds is loaded in the main task of the server. The end of the server releases it. The command **D CONFIG** shows the number of modules held (Chapter 3, “Console Commands” on page 17).

When a Module Does Not Run

If HTTPD cannot run the module of a **MOD** route, or the module ends badly, HTTPD first releases the storage of the module (a failure of that gives **HTTPD906W** with the program name and the return code). Then it tells the client and the operator.

For the client, HTTPD sends status 503 with the header **Content-Type: text/plain** if the module has not sent a status line yet. Then it sends the line **External program name text** to the body, where *text* is one of the reasons in the table. For the operator, HTTPD writes **HTTPD908E** with the program name and the same text.

Table 19. The reasons for a failure

Text	Cause
was not found	The module is in neither the STEPLIB nor the link list (completion code S806).
could not be loaded	The LINK failed with another completion code, for example a fetch error or a storage shortage. A module that returns –1 itself gives the same text.
was dispatched without a program name	The route has no program name.
could not be started, ESTAE create failed	HTTPD could not set up the error exit for the LINK, so the module did not run.
returned rc n	The module ran and ended with the negative return code n. A module that returns a code of zero or more is not a failure.
failed with Sxxx ABEND	The module ended with the system abend xxx.
failed with Unnnn ABEND	The module ended with the user abend nnnn, in decimal. **U0801** with the console line **@@CRT0 - No storage for C stack** means that the region had no block for the stack of the module (“The region” on page 28).

Only the abends leave a dump to look for. The other reasons mean that the module did not run, or ended by itself.

Chapter 3. Console Commands

The operator controls HTTPD from the console with the MODIFY command and the STOP command. The *HTTPD User's Guide* shows how to use them, in "Operating HTTPD". This chapter lists the syntax of every command and every message that a command writes.

Syntax

```
/F HTTPD,command
/P HTTPD
```

The name **HTTPD** is the name of the started task. HTTPD splits the text of a MODIFY command into words. A blank and a comma both separate words, so **/F HTTPD,D,C** and **/F HTTPD,D C** are the same command.

- **The echo.** HTTPD first writes the text it received, as **HTTPD100I CONS(n) "text"**. The number is the id of the console. The line is written for every MODIFY command, also for one that HTTPD does not know.
- **Case.** Commands are not case sensitive.
- **Abbreviations.** A word matches a keyword when it is the beginning of the keyword. **D**, **DIS** and **DISPLAY** are the same word. HTTPD tries the keywords in the order of the tables below, and the first match wins. This makes some short forms mean something else than they seem to. **D T** is **DISPLAY THREADS**, not **DISPLAY TIME**. Use **D TI** for the time. Likewise **D C** is **CONFIG**, **D L** is **LOGIN**, **D M** is **MEMORY**, **D P** is **PORTS**, **D S** is **STATS** and **D V** is **VERSION**. For **SET**, **S MA** is **MAXTASK**, **S MI** is **MINTASK** and **S S** is **STATS**. A plain **S M** is **MAXTASK**. The help text shows the shortest form in capitals, for example **DISPLAY TIme**.
- **No operand.** **DISPLAY** or **SET** alone, with nothing after it, gets no answer except the echo.
- **The answers.** All answers are messages to the console. None of them goes to a client or to a DD name. Each message is one line.
- **Queue.** HTTPD accepts at most 5 MODIFY commands that wait for it at one time.
- **Case of the output.** HTTPD writes its fixed text in upper case. Values keep their case, for example the document root or the name of the codepage.
- The command **/P HTTPD** (STOP) ends the server ("Stopping the server" on page 24). There is no MODIFY command that stops it.

The Commands

Table 20. The MODIFY commands

Command	Action
DISPLAY CONFIG **D C**	Shows the configuration that the server runs with ("DISPLAY CONFIG" on page 18).
DISPLAY LOGIN **D L**	Shows the users who are logged in ("DISPLAY LOGIN" on page 19).

DISPLAY MEMORY xxxxxx[,nnn] **D M xxxxxx**	Dumps storage (“DISPLAY MEMORY” on page 21).
DISPLAY PORTS **D P**	Shows the port the server listens on: **HTTPD102I HTTPD IS LISTENING ON PORT n**.
DISPLAY STATS **D S**	Shows the SMF level and the counters (“DISPLAY STATS” on page 19).
DISPLAY THREADS **D T**	Shows every thread of the server (“DISPLAY THREADS” on page 19).
DISPLAY TIME [-|+] [minutes] **D TI**	Shows the time (“DISPLAY TIME” on page 22).
DISPLAY VERSION **D V**	Shows the version of the server (“DISPLAY VERSION” on page 22).
SET MAXTASK n **S MA n**	Sets the largest number of worker threads (“SET MAXTASK and SET MINTASK” on page 22).
SET MINTASK n **S MI n**	Sets the smallest number of worker threads (“SET MAXTASK and SET MINTASK” on page 22).
SET STATS level [RESET] **S S**	Sets the SMF level, and resets the counters (“SET STATS” on page 23).
HELP **?**	Lists the commands (“HELP and unknown commands” on page 23).

No console command lists the routes of the server. The display module **HTTPDSRV** shows them. Chapter 7, “The Display Modules” on page 81 describes it.

DISPLAY CONFIG

The command shows the values that the server holds now, after it has read the Parmlib member. Use it to find out what HTTPD made of the member. The messages come in this order.

Table 21. The messages of DISPLAY CONFIG

Message	Content
HTTPD128I	**PORT n MINTASK n MAXTASK n**. The two task counts are the values from the Parmlib member. **SET MINTASK** and **SET MAXTASK** do not change them.
HTTPD129I	**CLIENT_TIMEOUT n KEEPALIVE t/m**. *n* is the client timeout in seconds, *t* the keep-alive timeout in seconds and *m* the number of requests allowed on one connection.
HTTPD130I	**SESSION_TIMEOUT n MIN**. The text **(REAPER DISABLED)** follows when the value is 0.
HTTPD135I	**SESSION_MAXAGE n MIN**. The text **(NO MAX-AGE)** follows when the value is 0.
HTTPD131I	**DOCROOT path**, or **DOCROOT (NONE)** when no document root is set.
HTTPD134I	**CODEPAGE name**. The name is **CP037** when the Parmlib member does not set one.
HTTPD136I	**REALM text**, the realm for Basic authentication.
HTTPD132I	**SMF TYPE n LEVEL level**. The level is **NONE**, **ERROR**, **AUTH** or **ALL**. It is the level in effect now, so a **SET STATS** shows here.

HTTPD133I	**CONFIG FROM name**, the Parmlib member that HTTPD read, for example **SYS2.PARMLIB(HTTPPRM0)**.
HTTPD137I	**RESIDENT YES MODULES HELD n** or **RESIDENT NO MODULES HELD n**. The count is the number of modules that the server holds in storage.
HTTPD138I	**RES_NOPROFILE ALLOW** or **RES_NOPROFILE DENY**, the value of the keyword ("Authentication Values" on page 4).

```
F HTTPD,D C
HTTPD100I CONS(0) "D C"
HTTPD128I PORT 8080  MINTASK 3  MAXTASK 9
HTTPD129I CLIENT_TIMEOUT 10  KEEPALIVE 5/100
HTTPD130I SESSION_TIMEOUT 30 MIN
HTTPD135I SESSION_MAXAGE 480 MIN
HTTPD131I DOCROOT /www
HTTPD134I CODEPAGE CP037
HTTPD136I REALM MVS.EXAMPLE
HTTPD132I SMF TYPE 243 LEVEL NONE
HTTPD133I CONFIG FROM DD:HTTPPRM
HTTPD137I RESIDENT YES  MODULES HELD 4
HTTPD138I RES_NOPROFILE ALLOW
```

The command does not show the routes, **BIND_TRIES**, **BIND_SLEEP**, **LISTEN_QUEUE** or the debug switch. Chapter 1, "The HTTPPRM Keywords" on page 1 lists every keyword.

DISPLAY LOGIN

The first message is **HTTPD121I USERS LOGGED IN: n**. One message follows for each login that is held:

```
F HTTPD,D L
HTTPD100I CONS(0) "D L"
HTTPD121I USERS LOGGED IN: 1
HTTPD122I USER IBMUSER  IP 192.0.2.17 ACEE(9BA1E0)
```

The user id is shown in up to 8 characters. The address is the address of the client. The number in brackets is the address of the security environment (ACEE) that belongs to the login, in hex.

DISPLAY STATS

Two messages. **HTTPD123I SMF: level (TYPE n)** shows the SMF level and the SMF record type. **HTTPD124I REQUESTS: n ERRORS: n BYTES: n ACTIVE: n** shows the number of requests, the number of errors, the bytes sent and the number of connections that are open now. The counters start at zero when the server starts, and **SET STATS ... RESET** sets the first three to zero. **HTTPD416I** shows the same three counters when the server stops. The levels are described in Chapter 8, "SMF Records" on page 93.

```
F HTTPD,D S
HTTPD100I CONS(0) "D S"
HTTPD123I SMF: NONE (TYPE 243)
HTTPD124I REQUESTS: 18  ERRORS: 3  BYTES: 186629  ACTIVE: 0
```

DISPLAY THREADS

The command writes one block for each thread. A line of dashes (**HTTPD199I**) ends each block. The order of the blocks is: the main thread, the socket thread, the dispatcher (the manager of the worker pool) and then the worker threads. A block exists only when the thread exists, with one exception: if there is no socket thread, the command still writes the **HTTPD103I** line of the socket thread, with the

address 00000000 and no other line. Without a worker pool there are no dispatcher and no worker blocks.

Table 22. The messages of DISPLAY THREADS

Message	Content
HTTPD103I	Starts a block: **..THREAD aaaaaaaa role**. The address is the control block of the thread. The role is **MAIN SERVER THREAD**, **SOCKET THREAD**, **DISPATCHER THREAD** or **WORKER THREAD**.
HTTPD104I	**EYE**, **TCB** and **OWNTCB** of the thread.
HTTPD105I	**TERMECB**, **RC** and **STACKSZE**. The stack size is in hex.
HTTPD106I	**FUNC** (the entry point of the thread), **ARG1** and **ARG2**.
HTTPD120I	Main thread only: **...RNAME name**. The name is **HTTPD.** and the address space id as 4 hex digits.
HTTPD119I	Socket thread only: **..SOCKET n ....PORT n (HTTP PROTOCOL)**, the socket number of the listener and the port.
HTTPD107I to **HTTPD111I**	Dispatcher only. 107: eye, task, wait ECB. 108: function, user data, stack size. 109: worker, queue and the state of the dispatcher as a number and a name (**INIT**, **RUNNING**, **QUIESCE**, **STOPPED** or **WAITING**). 110: the minimum and the maximum number of workers now in effect. 111: the number of workers and **.DISPCNT**, the number of dispatches.
HTTPD112I to **HTTPD117I**	One set for each worker. 112: worker, eye, wait ECB. 113: manager, task, queue. 114: the state of the worker as a number and a name (**INIT**, **RUNNING**, **WAITING**, **DISPATCHED**, **SHUTDOWN** or **STOPPED**). 115, 116 and 117: **START**, **WAIT** and **DISP**, each a 64-bit time value in hex followed by the date and time as text.
HTTPD141I	**.DISPCNT n**, how often this worker was dispatched.
HTTPD118I	A worker in state **RUNNING** that serves a client: **PROTOCOL HTTP ....USER name ...GROUP name**. The user is **ANONYMOUS** when the client has not logged in.
HTTPD119I	The same worker: **..REMOTE CLIENT ....PORT n ......IP a.b.c.d**, the port and the address of the client.
HTTPD120I	The same worker: **..SOCKET n ....PORT n ......IP a.b.c.d**, the socket of the connection and the port and the address of the server side.
HTTPD199I	A line of dashes after each block.

The next screen shows a server with three idle workers.

```
F HTTPD,D T
HTTPD100I CONS(0) "D T"
HTTPD103I ..THREAD 000FBFC8 MAIN SERVER THREAD
HTTPD104I .....EYE CTHDTASK  .....TCB 009DC328  ..OWNTCB 00000000
HTTPD105I .TERMECB 00000000  ......RC 00000000  STACKSZE 00010030
HTTPD106I ....FUNC 0009987C  ....ARG1 00000003  ....ARG2 000DB0F0
HTTPD120I ...RNAME HTTPD.0007
HTTPD199I ------------------------------------------------------
HTTPD103I ..THREAD 001A0FC8 SOCKET THREAD
HTTPD104I .....EYE CTHDTASK  .....TCB 009CD608  ..OWNTCB 009DC328
HTTPD105I .TERMECB 00000000  ......RC 00000000  STACKSZE 00008000
HTTPD106I ....FUNC 00098430  ....ARG1 000DB390  ....ARG2 0009E2E8
HTTPD119I ..SOCKET        3  ....PORT     8080  (HTTP PROTOCOL)
HTTPD199I ------------------------------------------------------
HTTPD103I ..THREAD 001A9FC8 DISPATCHER THREAD
HTTPD104I .....EYE CTHDTASK  .....TCB 009BAC18  ..OWNTCB 009DC328
HTTPD105I .TERMECB 00000000  ......RC 00000000  STACKSZE 00008000
HTTPD106I ....FUNC 000A15DC  ....ARG1 000DAE18  ....ARG2 00000000
HTTPD107I .....EYE CTHDMGR   ....TASK 001A9FC8  .WAITECB 809CD330
```

```
HTTPD108I ....FUNC 00098F1C  ...UDATA 000DB390  STACKSZE 00010000
HTTPD109I ..WORKER 000FB6D4  ...QUEUE 000FB07C  ...STATE 00000004 WAITING
HTTPD110I .MINTASK        3  .MAXTASK        9
HTTPD111I .WORKERS        3  .DISPCNT 18
HTTPD199I ------------------------------------------------------
HTTPD103I ..THREAD 001B4FC8 WORKER THREAD
HTTPD104I .....EYE CTHDTASK  .....TCB 009BAAA8  ..OWNTCB 009BAC18
HTTPD105I .TERMECB 00000000  ......RC 00000000  STACKSZE 00010000
HTTPD106I ....FUNC 00098F1C  ....ARG1 000DB390  ....ARG2 000FB748
HTTPD112I ..WORKER 000FB748  .....EYE CTHDWORK  .WAITECB 809B9F98
HTTPD113I .....MGR 000DAE18  ....TASK 001B4FC8  ...QUEUE 00000000
HTTPD114I ...STATE 00000002 WAITING
HTTPD115I ...START 000000006AE039DF  Tue Oct 27 00:42:55 2026
HTTPD116I ....WAIT 000000006AE03A28  Tue Oct 27 00:44:08 2026
HTTPD117I ....DISP 000000006AE03A27  Tue Oct 27 00:44:07 2026
HTTPD141I .DISPCNT 6
HTTPD199I ------------------------------------------------------
HTTPD103I ..THREAD 001C5FC8 WORKER THREAD
HTTPD104I .....EYE CTHDTASK  .....TCB 009BA378  ..OWNTCB 009BAC18
HTTPD105I .TERMECB 00000000  ......RC 00000000  STACKSZE 00010000
HTTPD106I ....FUNC 00098F1C  ....ARG1 000DB390  ....ARG2 000FB648
HTTPD112I ..WORKER 000FB648  .....EYE CTHDWORK  .WAITECB 809CD498
HTTPD113I .....MGR 000DAE18  ....TASK 001C5FC8  ...QUEUE 00000000
HTTPD114I ...STATE 00000002 WAITING
HTTPD115I ...START 000000006AE039DF  Tue Oct 27 00:42:55 2026
HTTPD116I ....WAIT 000000006AE03A27  Tue Oct 27 00:44:07 2026
HTTPD117I ....DISP 000000006AE03A27  Tue Oct 27 00:44:07 2026
HTTPD141I .DISPCNT 6
HTTPD199I ------------------------------------------------------
HTTPD103I ..THREAD 001D6FC8 WORKER THREAD
HTTPD104I .....EYE CTHDTASK  .....TCB 009BA048  ..OWNTCB 009BAC18
HTTPD105I .TERMECB 00000000  ......RC 00000000  STACKSZE 00010000
HTTPD106I ....FUNC 00098F1C  ....ARG1 000DB390  ....ARG2 000FB5C8
HTTPD112I ..WORKER 000FB5C8  .....EYE CTHDWORK  .WAITECB 809BA538
HTTPD113I .....MGR 000DAE18  ....TASK 001D6FC8  ...QUEUE 00000000
HTTPD114I ...STATE 00000002 WAITING
HTTPD115I ...START 000000006AE039DF  Tue Oct 27 00:42:55 2026
HTTPD116I ....WAIT 000000006AE03A28  Tue Oct 27 00:44:08 2026
HTTPD117I ....DISP 000000006AE03A27  Tue Oct 27 00:44:07 2026
HTTPD141I .DISPCNT 6
HTTPD199I ------------------------------------------------------
```

DISPLAY MEMORY

```
/F HTTPD,D M xxxxxx[,nnn]
```

xxxxxx is the address in hex. *nnn* is the length in bytes. A length with the prefix **0x** is hex, and a length with a leading 0 is octal. Any other length is decimal. One separator character stands between the address and the length. The comma is the one to use. The default is 256 bytes. A length that is 0 or below becomes
256. A length above 4096 becomes 4096.

Table 23. The messages of DISPLAY MEMORY

Message	Content
HTTPD148I	**USAGE: DISPLAY MEMORY XXXXXX[,NNN] (D M XXXXXX)**. There is no address.
HTTPD144E	**INVALID MEMORY ADDRESS 0x.....**. The address is above the 16 MB line (above hex 00FFFFFF). Nothing is read.
(dump)	The storage as a dump, written with the console message service.
HTTPD145I	**END OF MEMORY DUMP FOR 0x.....**. The dump is complete.
HTTPD146E	**ESTAE CREATE FAILURE, RC=0x.....**. The command could not set up its recovery and did not read the storage.
HTTPD147E	**ABEND Sxxx FOR DISPLAY MEMORY 0x.....** or **ABEND Uxxxx FOR DISPLAY MEMORY 0x.....**. Reading the storage abended, for example because the address is not valid. The recovery catches the abend.

```
F HTTPD,D M 10,16
HTTPD100I CONS(0) "D M 10,16"
Dump of 00000010 "DISPLAY MEMORY" (16 bytes)
+00000 0001D048 00000000 070E0000 00000000 :..}.............:
HTTPD145I END OF MEMORY DUMP FOR 0x000010
```

DISPLAY TIME

```
/F HTTPD,D TI [-|+][minutes]
```

The command writes two messages. The first is **HTTPD142I TIME yyyy/mm/dd hh:mm:ss GMT**. The second is **HTTPD143I TIME yyyy/mm/dd hh:mm:ss LOCAL OFFSET=+m** with the offset from GMT in minutes. Without an argument, or with 0, the offset is the one of the system. With another number, the command uses that number of minutes as the offset for this display only. The number can have a sign. HTTPD keeps it within 720 minutes (12 hours) in both directions.

```
F HTTPD,D TI
HTTPD100I CONS(0) "D TI"
HTTPD142I TIME 2026/10/27 05:44:39 GMT
HTTPD143I TIME 2026/10/27 00:44:39 LOCAL OFFSET=-300
```

DISPLAY VERSION

One message: **HTTPD140I HTTPD version (commit)**. The text is in upper case. The version is the version of the product. The commit identifies the source that was built.

```
F HTTPD,D V
HTTPD100I CONS(2) "D V"
HTTPD140I HTTPD 4.5.1 (nnnnnnn)
```

SET MAXTASK and SET MINTASK

```
/F HTTPD,S MA n
/F HTTPD,S MI n
```

The number is decimal and is required. Without a number, and with an operand that begins with a minus sign, the command answers **HTTPD149I USAGE: SET MAXTASK|MINTASK N (S MA N, S MI N), N = 1 TO 100** and changes nothing. The dispatcher is not woken. Otherwise a value of 0 is 1, and a value above 100 is 100. A word that is not a number is 1. The command sets the new limit and wakes the dispatcher, which creates or ends workers to fit. The answer is the dispatcher block of **DISPLAY THREADS** (messages **HTTPD107I** to **HTTPD111I**), followed by a line of dashes. The block shows the limits that are in effect now.

- **SET MAXTASK** with a value below the current minimum lowers the minimum to that value.
- **SET MINTASK** with a value above the current maximum raises the maximum to that value.
- The new values hold until the server stops. The Parmlib member is not changed, and **DISPLAY CONFIG** still shows the values it read.
- When the server runs without a worker pool, the commands write nothing.

A value that is too large is cut to 100. The dispatcher block shows the cut value:

```
F HTTPD,SET MA 99999
HTTPD100I CONS(0) "SET MA 99999"
HTTPD107I .....EYE CTHDMGR   ....TASK 001A9FC8  .WAITECB 40000000
HTTPD108I ....FUNC 00098F1C  ...UDATA 000DB390  STACKSZE 00010000
HTTPD109I ..WORKER 000FB6D4  ...QUEUE 000FB07C  ...STATE 00000004 WAITING
HTTPD110I .MINTASK        3  .MAXTASK      100
```

```
HTTPD111I .WORKERS       3  .DISPCNT 18
HTTPD199I ------------------------------------------------------
```

When **SET MINTASK** raises the minimum, the dispatcher starts the missing workers at once and writes **HTTPD061I** for each one:

```
F HTTPD,SET MI 4
HTTPD100I CONS(0) "SET MI 4"
HTTPD107I .....EYE CTHDMGR   ....TASK 001A9FC8  .WAITECB 40000000
HTTPD108I ....FUNC 00098F1C  ...UDATA 000DB390  STACKSZE 00010000
HTTPD109I ..WORKER 000FB6D4  ...QUEUE 000FB07C  ...STATE 00000004 WAITING
HTTPD110I .MINTASK       4  .MAXTASK      100
HTTPD111I .WORKERS       3  .DISPCNT 18
HTTPD199I ------------------------------------------------------
HTTPD061I STARTING WORKER(0FB0F0)   TCB(9B9C70) TASK(1E7FC8) STACKSIZE(65536)
```

SET STATS

```
/F HTTPD,S S NONE|ERROR|AUTH|ALL [RESET]
/F HTTPD,S S RESET
```

The level is one of **NONE**, **ERROR**, **AUTH** and **ALL**, spelled in full. Chapter 8, “SMF Records” on page 93 describes the levels.

Table 24. The messages of SET STATS

Message	Content
HTTPD126I	**SMF LEVEL SET TO level**. The level changes at once.
HTTPD125I	**STATISTICS COUNTERS RESET**. The word **RESET** follows the level, where the beginning of the word is enough, or it is the only operand, where it must be spelled in full. The three counters requests, errors and bytes are set to zero. The count of open connections is not changed. With only **RESET**, the level is not changed.
HTTPD126E	**INVALID SET STATS VALUE "text"**. The first operand is not a level and not **RESET**.
HTTPD127W	**MISSING STATS NONE|ERROR|AUTH|ALL [RESET]**. There is no operand.

```
F HTTPD,SET STATS ALL RESET
HTTPD100I CONS(0) "SET STATS ALL RESET"
HTTPD126I SMF LEVEL SET TO ALL
HTTPD125I STATISTICS COUNTERS RESET
```

```
F HTTPD,SET STATS BOGUS
HTTPD100I CONS(0) "SET STATS BOGUS"
HTTPD126E INVALID SET STATS VALUE "BOGUS"
```

HELP and unknown commands

HELP and **?** write **HTTPD150I MODIFY COMMANDS ARE:** and then one **HTTPD151I** line for each line of the help text. The text lists each command with a line of explanation below it, and ends with the example **/F HTTPD,D C**. An unknown first word gets the error and then the same text. An unknown second word gets only the error:

- **First word not known.** **HTTPD101E UNKNOWN CONSOLE COMMAND "word"** with the word as typed, followed by the help text.

- **Second word not known.** **`HTTPD101E UNKNOWN CONSOLE COMMAND`** or **`"SET word"`**, without the help **`"DISPLAY word"`** text.

This is the help text:

```
F HTTPD,HELP
HTTPD100I CONS(0) "HELP"
HTTPD150I MODIFY COMMANDS ARE:
HTTPD151I DISPLAY Config (D C)
HTTPD151I     DISPLAYS THE CONFIGURATION THIS SERVER IS RUNNING
HTTPD151I DISPLAY Login (D L)
HTTPD151I     DISPLAYS USERS THAT ARE LOGGED INTO THE HTTPD SERVER
HTTPD151I DISPLAY Memory xxxxxx~,nnn~ (D M xxxxxx)
HTTPD151I     DISPLAYS MEMORY AT THE GIVEN ADDRESS xxxxxx
HTTPD151I DISPLAY Ports (D P)
HTTPD151I     DISPLAYS THE PORT NUMBER THIS SERVER IS LISTENING ON
HTTPD151I DISPLAY Stats (D S)
HTTPD151I     DISPLAYS HTTPD CLIENT STATISTICS
HTTPD151I DISPLAY Threads (D T)
HTTPD151I     DISPLAYS INFORMATION ABOUT THE SERVER THREADS
HTTPD151I DISPLAY TIme ~-|+~~minutes~ (D TI)
HTTPD151I     DISPLAYS THE CURRENT TIME IN GMT AND LOCAL TIME
HTTPD151I DISPLAY Version (D V)
HTTPD151I     DISPLAYS THE SERVER VERSION AND BUILD
HTTPD151I
HTTPD151I SET MAxtask n (S MA n)
HTTPD151I     SETS THE WORKER MAXIMUM THREAD COUNT
HTTPD151I SET MIntask n (S MI n)
HTTPD151I     SETS THE WORKER MINIMUM THREAD COUNT
HTTPD151I SET Stats NONE|ERROR|AUTH|ALL ~RESET~
HTTPD151I     SETS THE SMF RECORDING LEVEL
HTTPD151I
HTTPD151I EXAMPLE: /F HTTPD,D C
```

An unknown second word gets only the error:

```
F HTTPD,D BOGUS
HTTPD100I CONS(0) "D BOGUS"
HTTPD101E UNKNOWN CONSOLE COMMAND "DISPLAY BOGUS"
```

Stopping the server

`/P HTTPD` ends the server. HTTPD writes **`HTTPD098I HTTPD SHUTTING DOWN`** and then works through these steps.

1. The server stops accepting connections.
2. HTTPD waits for the socket thread, up to 10 seconds. From the second second on it writes **`HTTPD040I WAITING FOR SOCKET THREAD TO TERMINATE (n)`**. When the thread has not ended after 10 seconds, **`HTTPD041I FORCE DETACHING SOCKET THREAD`** follows.
3. The listener is closed, the worker pool is stopped and the modules that the server held are released.
4. The clients and the routes are freed.
5. **`HTTPD416I STATS: n REQUESTS, n ERRORS, n BYTES`** shows the counters.
6. **`HTTPD099I HTTPD SHUTDOWN COMPLETE`** closes the sequence. The security identity of the task and the authorization are given back after it.

```
P HTTPD
HTTPD098I HTTPD SHUTTING DOWN
HTTPD060I SHUTDOWN SOCKET THREAD    TCB(9CD7C8) TASK(14FFC8) STACKSIZE(32768)
HTTPD060I SHUTDOWN WORKER(161B80)   TCB(9BA858) TASK(185FC8) STACKSIZE(65536)
HTTPD060I SHUTDOWN WORKER(161C00)   TCB(9BAB88) TASK(174FC8) STACKSIZE(65536)
HTTPD060I SHUTDOWN WORKER(161D00)   TCB(9BAEB8) TASK(163FC8) STACKSIZE(65536)
HTTPD416I STATS: 30 REQUESTS, 12 ERRORS, 21694 BYTES
HTTPD099I HTTPD SHUTDOWN COMPLETE
$HASP395 HTTPD    ENDED
IEF404I HTTPD - ENDED - TIME=00.42.25
```

The job step ends with return code 0. If the shutdown itself fails, the server writes **HTTPD912E TERMINATE FAILED WITH RC=.....**. Appendix A, “Messages” on page 99 lists every message. The *HTTPD User’s Guide* shows a stop from the console in “Operating HTTPD”.

Note: The procedure that ships with HTTPD lists the line **/F HTTPD,SHUTDOWN** in its comments. The server has no such command. A command with this word gets **HTTPD101E** and the help text. Use **/P HTTPD**.

/C HTTPD ends the address space with the system abend S222. The shutdown does not run: the console shows **IEE301I**, **IEF450I** and **$HASP395**, and none of **HTTPD098I**, **HTTPD060I**, **HTTPD416I** and **HTTPD099I**. The server itself writes no **libc370 recovery:** line and no **libc370 @@@try.c failed()** line. A module that was linked with libc370 older than 2.26.0 can still write the second one. The *HTTPD User’s Guide* shows a cancel in “Operating HTTPD”.

Chapter 4. The Started Task and the Package

HTTPD runs as a started task. This chapter lists the procedure that ships with HTTPD and every DD name in it. It then shows how the server gets its authorization and its identity, which calls it makes to the security product, the order of the start, and the return codes of the job step. The last part is a map of the package: the FMID, the data sets and the members.

The *HTTPD User's Guide* shows how to install the package and how to run the task, in "Installing HTTPD" and "Operating HTTPD".

The Procedure

The member **HTTPD** of the sample library is the procedure. The operator copies it to the procedure library, for example to **SYS2.PROCLIB(HTTPD)**. This is the member without its comment lines.

```
//HTTPD    PROC M=HTTPPRM0,
//            D='SYS2.PARMLIB',
//            STCUSER=HTTPD,
//            STCGRP=USER
//HTTPD    EXEC PGM=HTTPD,REGION=8M,TIME=1440,
//            PARM='STCUSER=&STCUSER STCGROUP=&STCGRP'
//STEPLIB  DD  DISP=SHR,DSN=HTTPD.LINKLIB
//SYSUDUMP DD  SYSOUT=*
//HTTPDERR DD  SYSOUT=*
//HTTPDOUT DD  SYSOUT=*
//HTTPDIN  DD  DUMMY
//SNAP     DD  SYSOUT=*
//HTTPDBG  DD  SYSOUT=*
//HTTPSTAT DD  SYSOUT=*
//HTTPPRM  DD  DSN=&D(&M),DISP=SHR,FREE=CLOSE
```

The member also has the statements **HASPCKPT** and **HASPACE1**, as comment lines ("The JES2 data sets" on page 31).

The package build writes the name of the library that SMP installs into, here **HTTPD.LINKLIB**, into the **STEPLIB** statement of the member.

The symbolic parameters

Table 25. The symbolic parameters of the procedure

Symbolic	Default	Meaning
M	**HTTPPRM0**	The member of the Parmlib. It is the member name in the **HTTPPRM** DD.
D	**SYS2.PARMLIB**	The data set that holds the member.
STCUSER	**HTTPD**	The user id that the server takes as its identity ("The Identity of the Task" on page 33).
STCGRP	**USER**	The group of that user id. The name is **STCGRP** and not **STCGROUP**, because the name of a JCL symbolic parameter has at most 7 characters. The PARM keyword that the program reads is **STCGROUP=**.

The START command overrides a symbolic:

```
S HTTPD
S HTTPD,M=HTTPPRM1
S HTTPD,D='MY.PARMLIB'
S HTTPD,STCUSER=WEBSRV,STCGRP=WEBGRP
```

The EXEC statement

The program is **HTTPD**. The procedure sets **REGION=8M** and **TIME=1440**. The server reads its arguments from the PARM string. It splits the string at blanks and not at commas. It knows these keywords.

Table 26. The PARM keywords

Keyword	Meaning
STCUSER=name	The user id for the identity of the task. The default is **HTTPD**. A keyword without a value gives the default.
STCGROUP=name	The group for the identity. The default is **USER**. A keyword without a value gives the default.
CONFIG=name	Has no effect. HTTPD writes **HTTPD024W CONFIG=name IS IGNORED, CONFIGURATION COMES FROM THE** and **HTTPPRM DD** then **HTTPD024I USE S HTTPD,M=MEMBER TO SELECT A DIFFERENT MEMBER**. The symbolic **M** selects the member.

HTTPD ignores any other word in the PARM string, without a message. The configuration of the server comes from the Parmlib member and not from the PARM string. Chapter 1, “The HTTPPRM Keywords” on page 1 lists the keywords of the member.

The region

The **REGION** of the **EXEC** statement is the one limit on all the storage that the server holds. It is not a Parmlib keyword. A server that runs many module requests at the same time may need more than the **8M** of the shipped procedure.

Every request that runs a module takes one contiguous block of storage for the C stack of the module, from the region, for as long as the module runs.

Table 27. What a module request costs

What	Size
C stack of a module that sets nothing	262,328 bytes, about 256 KB. It is the default of the C runtime (**@@CRT0** of LIBC/370). A module that sets **__stklen** gets that size (“What the Startup Code Does” on page 60).
C stack of a module with its own **__stklen**	The value plus 48 bytes. mvsMF sets 64 KB, so a request of mvsMF takes 65,584 bytes.
Stack of a worker thread	64 KB, once per worker thread.
Stack of the socket thread	32 KB, once.
Resident copy of a module	The size of the load module, once per module and not per request (**RESIDENT**, “Routes and Modules” on page 5).

A request runs on a worker thread, so at most **MAXTASK** module requests run at once, and their stacks add up to **MAXTASK** times the stack of the module. At the default **MAXTASK=9** and modules with the default stack that is 9 times 262,328 = 2,360,952 bytes, plus 9 times 64 KB for the worker threads. The stack

is one contiguous block, so the free storage of the region has to be whole enough to hold it and not only large enough in sum. Each step of **MAXTASK** costs at least one more worker stack and one more module stack, before the working storage of the module itself.

If the request cannot get its stack, the module does not start. The console shows

```
@@CRT0 - No storage for C stack
HTTPD908E EXTERNAL PROGRAM REXX failed with U0801 ABEND
```

and the client gets status 503 with the text **External program REXX failed with U0801 ABEND**. The other requests of the server go on. A module that is linked against a LIBC/370 older than 1.0.3 does not ask for the stack with a condition and abends **S80A** instead, from inside the system. **HTTPD904E** and **HTTPD999E** are the same shortage in the storage of the server itself.

Only a measurement shows the size that a site needs. This one was made against a release candidate of 4.4.0: 60 requests to REXX pages, 9 at a time. The table gives the failed requests of the 60, one entry per run.

Table 28. A measurement of parallel module requests

Set-up	Failed requests of 60
REGION=8M, **RESIDENT=YES**	6, 32, 0
REGION=8M, HTTPD 4.3.0 with the same module libraries	0, 4, 0
REGION=8M, **RESIDENT=NO**	0 (one run)
REGION=16M	0 (one run)

A mixed load of 6 requests at a time across all module types was clean in every set-up. The sample is small and the runs are not independent. It shows an order of magnitude and not a threshold. It was not run with Lua requests. Three rules follow from it.

- For many parallel REXX or Lua requests, set **REGION=16M** in the procedure before you raise **MAXTASK**. 16M is the largest value that was tried. If it is not enough, lower **MAXTASK** and do not raise it.
- Raise **MAXTASK** only together with the region.
- Do not use **RESIDENT=NO** as the cure. It brings back the fetch of the module on every request.

The DD Statements

The next table shows what each DD of the procedure is for.

Table 29. The DD statements of the procedure

DD name	In the procedure	Use
STEPLIB	**DISP=SHR, DSN=HTTPD.LINKLIB**	The load library of the server and of the modules of the routes (“The STEPLIB” on page 32).
SYSUDUMP	**SYSOUT=***	Not opened by HTTPD. The system writes an abend dump to it.
HTTPDERR	**SYSOUT=***	Standard error of the server. The server opens it at the start. The start code of a server module opens it as well.

HTTPDOUT	**SYSOUT=***	Standard output of the server, opened in the same way.
HTTPDIN	**DUMMY**	Standard input of the server. When the DD is missing, HTTPD opens the data set **NULLFILE** in its place.
SNAP	**SYSOUT=***	Not opened by HTTPD.
HTTPDBG	**SYSOUT=***	The debug output. HTTPD opens it only when the Parmlib member sets **DEBUG** to a value above 0 (Chapter 1, "The HTTPPRM Keywords" on page 1). When it cannot open the DD, it writes **HTTPD020E FOPEN FOR DD:HTTPDBG FAILED, ERRNO=n** and **HTTPD021I DEBUG/TRACE OUTPUT WILL BE TO, DD:SYSTERM** and it sends the debug output to the standard error stream, which is the DD **HTTPDERR**. When a write fails, HTTPD writes **HTTPD039W** once and stops the debug output until the next start.
HTTPSTAT	**SYSOUT=***	Not opened by HTTPD.
HTTPPRM	**DSN=&D(&M), DISP=SHR,FREE=CLOSE**	The Parmlib member ("The Parmlib member").
HASPCKPT	A comment line: **DISP=SHR,DSN=SYS1.HASPCKPT, UNIT=3350,VOL=SER=MVS000**	The checkpoint data set of JES2, for server modules. Optional ("The JES2 data sets" on page 31).
HASPACE1	A comment line: **DISP=SHR,DSN=SYS1.HASPACE, UNIT=3350,VOL=SER=SPOOL1**	The spool data set of JES2, for server modules. Optional ("The JES2 data sets" on page 31).

The Parmlib member

The server reads the DD **HTTPPRM** once, at the start, and to the end. These cases are possible.

- The DD is missing. HTTPD writes **HTTPD020W CANNOT OPEN DD:HTTPPRM -- USING DEFAULTS** and starts with its built-in defaults and no routes.
- The DD is there and the read fails before the end of the member. HTTPD writes **HTTPD038E READ ERROR ON DD:HTTPPRM -- CONFIGURATION INCOMPLETE** and does not start ("Return Codes and Abends" on page 37).
- Otherwise the server uses the member (Chapter 1, "The HTTPPRM Keywords" on page 1).

The standard stream DDs

The procedure must not contain the DD names **SYSPRINT**, **SYSTERM** and **SYSIN**. HTTPD tests for them before anything else. It uses the DD names **HTTPDOUT**, **HTTPDERR** and **HTTPDIN** for its standard streams instead. The routine in front of the server (**__premain**, in the file **httpstrt.c**) writes these messages.

Table 30. The messages for the standard stream DDs

Message	Text and cause
HTTPD014E	**SYSPRINT DD NOT ALLOWED, HTTPD USES HTTPDOUT FOR STDOUT**
HTTPD015E	**SYSTERM DD NOT ALLOWED, HTTPD USES HTTPDERR FOR STDERR**
HTTPD016E	**SYSIN DD NOT ALLOWED, HTTPD USES HTTPDIN FOR STDIN**
HTTPD017W	**HTTPDOUT DD NOT DEFINED, STDOUT IS NOT AVAILABLE**. The DD cannot be opened.

HTTPD018W	**HTTPDERR DD NOT DEFINED, STDERR IS NOT AVAILABLE**. The DD cannot be opened.
HTTPD019W	**HTTPDIN DD NOT DEFINED, STDIN IS NOT AVAILABLE**. The DD cannot be opened, and neither can the data set **NULLFILE**.

The routine first tests the three forbidden DDs and writes a message for each one that it finds. When it found one, it stops there. Otherwise it opens the three DDs of the server and writes a message for each that fails. Each of the six conditions ends the routine with **EXIT_FAILURE**, and the server does not start. The messages with the letter W stop the start as well.

The JES2 data sets

HTTPD does not open **HASPCKPT** or **HASPACE1** itself. A server module runs in the task of the server, because HTTPD starts it with the LINK macro. The module has no allocations of its own and opens a DD name in the allocations of the started task. The jobs API of mvsMF reads the checkpoint and the spool of JES2 through the function **jesopen()** of the C library.

The two DDs are optional. The sample member has both as comment lines. What **jesopen()** does depends on the level of the C library that the module is built with:

- With LIBC/370 2.24.0 and later, a DD in the procedure wins. When the procedure has no **HASPCKPT**, the function allocates **SYS1.HASPCKPT** itself, through the catalog or, when the data set is not cataloged, on the online volume that holds it. When the procedure has no **HASPACE1**, it allocates **SYS1.HASPACE** on the spool volume that the checkpoint names. mvsMF 1.4.0 and later are built with such a level.
- With an older level, the function opens only the DD names. The jobs API of mvsMF 1.3.0 and earlier is built this way, and it needs both DDs.

Uncomment both lines of the sample member in these two cases:

- You run mvsMF 1.3.0 or earlier.
- The JES2 of the system builds its data set names with a prefix other than **SYS1** (its **$DSNPRFX** value). The allocation looks only for the **SYS1** names. When it finds no checkpoint, the C library writes the console lines **SYS1.HASPCKPT is not cataloged and is on no online volume** and **JES2 uses another dataset name prefix than SYS1, or there is no JES2 checkpoint**.

Leaving the DDs active in any other system does no harm, because a DD wins. When you uncomment them, **UNIT** and **VOL=SER** must match the JES2 procedure of the system. Check them there before the first start. The DDs are **DISP=SHR**, because JES2 owns the data sets.

When the data sets cannot be reached, the server still starts, and everything except the jobs API works. This holds in both cases above: mvsMF 1.3.0 or earlier without the DDs, and any mvsMF level on a system with another prefix than **SYS1** without the DDs. The effect differs by request:

Table 31. A request when the JES2 data sets cannot be opened

Request	Answer
The job list	Status 500, with the message “Unable to open JES2 checkpoint and spool datasets”.
Any request that names a job by name and id, a spool read included	Status 404, “job not found”, although the job exists. The function that looks the job up cannot tell an unreachable JES2 from an unknown job.

Each of these requests writes the console line **MVSMF201E UNABLE TO OPEN THE JES2 CHECKPOINT AND SPOOL DATA SETS**, and the C library writes the

line that names the data set that it could not open. Read the console, not the status code. Submitting a job is not affected, because the submit allocates its own internal reader.

The STEPLIB

HTTPD finds the modules of its routes by the standard search of the LINK macro. It names the module from the **`MOD=`** keyword and passes no DCB. The system looks in the STEPLIB of the task first, and then in the link list. The server itself is the program of the EXEC statement. So the STEPLIB holds the server and the modules of the routes, or the link list holds the modules.

The system takes the first library in the concatenation that has the module. Two libraries are in use.

- **HTTPD.LINKLIB.** The library that SMP installs into. The sample procedure names it. Only the install job writes to it.
- **HTTPD.DEV.LINKLIB.** The target of **`mbt deploy`** in the project file. A deploy deletes the target before it receives into it, because RECEIVE does not merge. For this reason the deploy target is a library of its own and never the production library. Deploying does not activate anything. To run a development build, name **`HTTPD.DEV.LINKLIB`** in a STEPLIB deliberately. The usual way is a second HTTPD on another port. HTTPD refuses a second listener on a port that another HTTPD serves (HTTPD037E, “Return Codes and Abends” on page 37), so the production task keeps running. Put the DEV library before **`HTTPD.LINKLIB`** in the concatenation, so that its modules win.

Stop the task before you replace the load library. The task loads from it.

Authorization

The server needs APF authorization. It tests at entry whether the job step is authorized, and it then makes the task authorized in either case.

1. **Test.** The server issues one TESTAUTH. The test runs before anything else, because after the next step every task is authorized.
2. **Setup.** The server authorizes its task with SVC 244. When the C runtime has recorded that, the server authorizes the STEPLIB in the same way and marks its own entry in the contents directory of the task as an authorized library entry. It then identifies the entry **`CTHREAD`** to the system and marks that entry the same way. **`CTHREAD`** is an entry point inside the load module **`HTTPD`** and not a member of a library. The server creates its threads through it. The server does this setup also when the step was authorized at entry.
3. **Message.** After the version banner the server tells how the step was authorized.

Table 32. The two ways to be authorized

Message	Meaning
`HTTPD002I AUTHORIZED BY LIBRARY (MODULE KEY 0)`	The job step was authorized at entry. The libraries of the STEPLIB are in the list of APF-authorized libraries, or the module is in an authorized library of the link list. The system fetches the module with the protection key 0.
`HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)`	The job step was not authorized at entry. The server authorized the task itself. The system fetched the module with the key 8.

The key in the message is not measured. The server infers it from the route: a step that is authorized at entry has its module fetched with the key 0, and an unauthorized step has the key 8, and SVC 244 comes too late to change either. The same module behaves differently under the two keys. Chapter 6, “The Server Module Interface” on page 59 tells what a server module may keep in storage.

The load module **HTTPD** is linked with the authorization code 1 (**ac = 1** in the module definition). SMP copies the module into the library and does not link it again, so the code stays.

The next screen shows a start of the server in a library that is not in the APF list. The server authorized itself by SVC.

```
S HTTPD
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.44.51
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/info
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/services/authenticate
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/restconsoles/*
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/*
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD061I STARTING SOCKET THREAD    TCB(9CD7C8) TASK(152FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(164D00)   TCB(9BBEB8) TASK(166FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(164C00)   TCB(9BBB88) TASK(177FC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(164B80)   TCB(9BB858) TASK(188FC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

When the setup fails, HTTPD writes **HTTPD012E name UNABLE TO DYNAMICALLY OBTAIN APF AUTHORIZATION**, where **name** is the name of the program as the system passed it to the server. It does not write **HTTPD002I**, and the step ends with the return code of the failed call (“Return Codes and Abends” on page 37).

At the end the server gives up what it took. It first puts back the security environment that the task had at the start (“The Identity of the Task”), then it releases the authorization of the STEPLIB and of the task with SVC 244.

The Identity of the Task

RAKF has no table of started procedures. It uses the name of the procedure only to find out that the caller is a started task, and it gives every started task the same account, STC/STCGROUP. On a standard RAKF profile set this account holds ALTER authority on every data set of the system. HTTPD does not keep it. At the start the server logs on as the user id of **STCUSER** and **STCGROUP** (“The EXEC statement” on page 28).

The server reads these two keywords from the PARM string and not from the Parmlib member. The logon must happen before the server reads the member and opens the file system, and these accesses should already run under the new identity.

The logon works in this order:

1. The server enters supervisor state with the key 0. When it cannot, it writes **HTTPD003W** and skips the logon.
2. It remembers the security environment (ACEE) that the task has.
3. It calls RACINIT with **ENVIR=CREATE** for the user id and the group, without a password.
4. When the call works, the new ACEE becomes the resting identity of the address space, and the server writes **HTTPD004I**. When it fails, the server writes **HTTPD004W** and keeps the identity that the task had.

Table 33. The messages for the identity

Message	Text and cause
HTTPD004I	**STC IDENTITY SET TO user/group VIA RACINIT**. The logon worked.

HTTPD004W	**RACINIT ENVIR=CREATE FAILED FOR user/group RC=n**. The logon failed, for example because the user id is not defined. The server goes on and keeps the identity that it inherited.
HTTPD003W	**RACINIT SKIPPED, CANNOT ENTER SUPERVISOR STATE**. The server could not get the key 0 that RACINIT needs. It goes on with the inherited identity.

A failed logon is a warning and does not stop the start. The inherited identity is the more powerful one, so the warning needs action: define the user id.

The new ACEE is not freed while the server runs. The server also keeps the old ACEE and does not release it. When the logon fails, the task would otherwise be left without any identity. At the stop the server puts the old ACEE back before it releases the authorization ("Authorization" on page 32). SVC 244 is gated by the profile SVC244 in the class FACILITY, and the task took the authorization out under the old account. The server therefore gives it back under the same account, and the user id of **STCUSER** needs no access to that profile.

The user id of **STCUSER** needs READ authority on the Parmlib member and on the document root. A server module that sets no ACEE of its own uses the identity of the task for its accesses.

What the Server Asks of the Security Product

The profiles belong to the site. HTTPD names none of them. The route decides which resource a request is checked against, with the keyword **RES=** (Chapter 2, "Routes and Authentication Policy" on page 9). The server makes these calls.

Table 34. The calls to the security product

Call	Use
Logon of the task, RACINIT without a password	The user id of **STCUSER** must be defined. Without it the server writes **HTTPD004W** ("The Identity of the Task" on page 33).
Test logon at the start, RACINIT with the user id **HTTPPRB** and a wrong password	From HTTPD 4.5.1. The server makes this logon once, after the logon of the task and in the same supervisor state. A product that checks refuses it, and RAKF writes **RAKF0004**. If the product returns an ACEE, it accepts any user id. The server then writes **HTTPD042W SECURITY SYSTEM ACCEPTS ANY USERID, ALL LOGINS** and refuses every logon of a client (below). There is no test when **HTTPD003W** skipped the logon of the task, and none later. The user id **HTTPPRB** is reserved.
Logon of a client, RACINIT with user id and password	A route with **AUTH=BASIC**, **AUTH=FORM** or **AUTH=TOKEN** checks the user id and the password of the client. A form login that works gives **HTTPD007I**, and the release of a credential gives **HTTPD008I**. From HTTPD 4.5 a logon that fails for a reason other than the password gives **HTTPD009W LOGIN SESSION NOT CREATED, RACINIT FAILED RC=n**. A wrong password is the client's doing and is not written to the console. From HTTPD 4.5.1 the server refuses the logon, without a message, when the test logon at the start showed that the product accepts any user id, when the user id is **STC**, and when the ACEE that comes back names another user than the client sent. Chapter 2, "Routes and Authentication Policy" on page 9 has the policies.

Check of a resource for a route with **RES=class:resource**	The server asks for the access READ. The return codes 0 and 4 allow the request. Return code 4 means that no profile covers the resource. With **RES_NOPROFILE=DENY** the return code 4 refuses the request (“Authentication Values” on page 4). Chapter 2, “Routes and Authentication Policy” on page 9 describes what happens on a refusal.
Probe at the start	For each route with **RES=** and an **AUTH=** other than **NONE**, the server asks once whether a profile covers the resource. It asks as its own identity. Only the return code 4 gives **HTTPD425W NO PROFILE FOR class:resource -- path NOT GATED**. The route still works, and every request that reaches the check passes. With **RES_NOPROFILE=DENY** the message is **HTTPD426W NO PROFILE FOR class:resource -- path REFUSES ALL**, and **REQUESTS** every request that reaches the check gets status 403. A typing error in the resource name looks like this. A class that RAKF does not define looks like this with RAKF 1.2.x and does not with RAKF 2.0.0, see below.
Authorization by SVC 244	The security product allows or refuses the SVC for the account of the task. The server does not check the profile itself.

A refusal is not a missing profile. A resource that is defined and has nobody permitted gives the return code 8, and the probe stays silent. The return code 4 for an unprotected resource is a property of RAKF. Under RACF with PROTECTALL, an unprotected resource gives 8 for an ordinary user and 0 for a SPECIAL user, and a system with no security product gives 0. On those systems the probe stays silent without a profile as well, so a silent probe does not prove that a profile exists. **RES_NOPROFILE=DENY** acts on the return code 4 only. It does not change what a system answers with 0 or with 8.

The level of RAKF decides what a class that it does not define gives. RAKF 2.0.0 (MVS/CE) gives the return code 8 for a class that is not in the class descriptor table of the system. It writes **RAKF0012 UNDEFINED CLASS class** to the console. The route refuses every request, the probe stays silent and **RES_NOPROFILE** changes nothing. A class with a typing error gives this answer. RAKF 1.2.x (measured on TK5) gives the return code 4 for such a class, with no message: the probe writes **HTTPD425W** or, with **RES_NOPROFILE=DENY**, **HTTPD426W**, and without the keyword the route is open. A resource name with a typing error gives the return code 4 on both levels when no profile covers the misspelled name. A measurement with the RAKF of MVS/CE (RAKF 2.0.0) and with that of TK5 (RAKF 1.2.x) gave the return code 0 on both for a batch user, the operator **IBMUSER**, for the classes **TAPEVOL**, **DASDVOL** and **TERMINAL** without a profile. The start probe, which runs with the identity of the server, stays silent for these classes, so its answer is not 4. HTTPD allows a request on the return code 0, so **RES_NOPROFILE=DENY** does not close such a route. The measurement did not use the ACEE of a client.

The *HTTPD User’s Guide* shows how to define the profiles with RAKF, in “Authentication and Security”.

The Start in Order

The server does these steps in this order.

1. The routine in front of the server tests the standard stream DDs (“The standard stream DDs” on page 30).
2. The server tests the authorization and sets it up (“Authorization” on page 32). When the setup fails, it writes the banner and **HTTPD012E** and ends.
3. The banner: **HTTPD000I HTTPD version (commit) STARTING**, then **HTTPD005I** with the version of the C runtime, then **HTTPD006W BUILT FROM A MODIFIED WORKING TREE** only for a build that contains changes that were not committed, then **HTTPD002I**.

4. The server gets the console interface. Without it, it writes **HTTPD090E UNABLE TO INITIALIZE CONSOLE INTERFACE** and ends.
5. The server reads the PARM string and sets the identity ("The Identity of the Task" on page 33).
6. The server reads the Parmlib member ("The Parmlib member" on page 30). Each route is registered as it is read, with **HTTPD036I** for a module and **HTTPD417I** for a location. When a route lost its policy, the server writes **HTTPD420E ROUTE AUTHORIZATION POLICY INCOMPLETE -- HTTPD WILL NOT START** and ends before it opens the port.
7. The server sets up the codepage tables, initializes the file system and opens the debug DD when the member asks for it. A file system that does not start gives **HTTPD044W UNABLE TO INITIALIZE FILE SYSTEM**. The server goes on without static file serving.
8. The server binds the listener ("The listener" on page 37).
9. The server tests the SMF recording. When a level is set and SMF is not active, it writes **HTTPD430W SMF TYPE n CONFIGURED BUT SMF INACTIVE** (Chapter 8, "SMF Records" on page 93).
10. The server probes the **RES=** profiles ("What the Server Asks of the Security Product" on page 34).
11. The server holds the modules of the routes, unless the Parmlib member switches **RESIDENT** off (Chapter 1, "The HTTPPRM Keywords" on page 1). For each module it looks up the entry in the library and then loads it once. Three messages are possible, and the route works in each case. The module is fetched for each request.
    - **HTTPD440W MODULE name NOT FOUND AT STARTUP, NOT HELD** — the module is not in the STEPLIB or the link list.
    - **HTTPD441W MODULE name NOT REUSABLE, FETCHED PER REQUEST** — the module is linked neither reentrant nor serially reusable.
    - **HTTPD442W MODULE name LOAD FAILED, ABEND CODE Sxxx REASON rr, FETCHED PER REQUEST**.
12. The server creates the socket thread with a stack of 32 KB. When it cannot, it writes **HTTPD033E UNABLE TO CREATE SOCKET THREAD** and ends.
13. The server creates the pool of worker threads. A worker has a stack of 64 KB. A **MAXTASK** of 0 gives 9, and a **MINTASK** of 0 gives 3. When **MINTASK** is larger than **MAXTASK**, the server raises **MAXTASK** to the value of **MINTASK**. When it cannot create the pool, it writes **HTTPD034W UNABLE TO CREATE WORKER THREADS, DYNAMIC DOCUMENTS DISABLED** and goes on.
14. Each thread writes **HTTPD061I STARTING ...** when it starts.
15. The server sets up the key for the credentials. When that fails, it writes **HTTPD047E UNABLE TO INITIALIZE SERVER CREDENTIAL KEY, RC=n** and goes on. The key is new at every start.
16. The server waits until the socket thread is in its accept loop. It tests up to 50 times at an interval of 100 ms, which is 5 seconds. A healthy start does not wait. When the thread is not running, the server writes **HTTPD064E SOCKET THREAD IS NOT RUNNING, START ABANDONED** and stops.
17. The server allows 5 queued console commands (Chapter 3, "Console Commands" on page 17) and writes **HTTPD001I HTTPD version READY - SERVING docroot**. Without a document root the text is **HTTPD001I HTTPD version READY - NO DOCUMENT ROOT**. The version in the message is in upper case.

The main task then waits for console commands.

The next screen shows a start with a second member of the Parmlib, which defines more routes. The screen is the real console output of a test system.

```
S HTTPD,M=HTTPPRM1
$HASP100 HTTPD    ON STCINRDR
$HASP373 HTTPD    STARTED
IEF403I HTTPD - STARTED - TIME=05.38.23
HTTPD000I HTTPD 4.5.1 (nnnnnnn) STARTING
HTTPD005I LIBC370 2.26.0 (3AA3B28)
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)
HTTPD004I STC IDENTITY SET TO HTTPD/USER VIA RACINIT
RAKF0004 INVALID ATTEMPT TO ACCESS SYSTEM:  USER:HTTPPRB
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/info
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/services/authenticate
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/restconsoles/*
```

```
HTTPD036I MODULE MVSMF REGISTERED FOR /zosmf/*
HTTPD036I MODULE HTTPDSRV REGISTERED FOR /.dsrv
HTTPD036I MODULE HTTPDM REGISTERED FOR /.dm
HTTPD036I MODULE HTTPDMTT REGISTERED FOR /.dmtt
HTTPD417I LOCATION /pub/* REGISTERED
HTTPD417I LOCATION /private/* REGISTERED
HTTPD417I LOCATION /form/* REGISTERED
HTTPD417I LOCATION /token/* REGISTERED
HTTPD417I LOCATION /admin/* REGISTERED
HTTPD417I LOCATION /denied/* REGISTERED
HTTPD054I LISTENING ON ANY PORT 8080
HTTPD061I STARTING SOCKET THREAD    TCB(9CD6E8) TASK(1A6FC8) STACKSIZE(32768)
HTTPD061I STARTING WORKER(0FE748)   TCB(9BBA58) TASK(1BAFC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0FE648)   TCB(9BB328) TASK(1CBFC8) STACKSIZE(65536)
HTTPD061I STARTING WORKER(0FE5C8)   TCB(9BAEB8) TASK(1DCFC8) STACKSIZE(65536)
HTTPD001I HTTPD 4.5.1 READY - SERVING /www
```

The listener

The server binds to all interfaces of the host (INADDR_ANY). The steps are:

1. The server takes an ENQ of the scope SYSTEM with the major name **HTTPD** and the minor name **HTTPD.PORT.nnnnn**, where **nnnnn** is the port with 5 digits. The server asks for the resource and does not wait. When another address space holds the resource, or the ENQ fails for another reason, HTTPD writes **HTTPD037E HTTPD IS ALREADY ACTIVE ON PORT n, THIS INSTANCE ENDS** and ends. A second server on another port starts. The system releases the ENQ when the task ends, also after an abend.
2. The server creates the socket. When that fails, it writes **HTTPD028E SOCKET() FAILED, RC=n ERRNO=n**.
3. The server binds the socket. When the bind fails with EADDRINUSE, the server closes every socket that is bound to the port in the socket table. Sockets of an earlier instance that ended without closing them are among these. When it closed any, it waits and tries the bind once more.
4. When the bind still fails, the server writes **HTTPD030E BIND() FAILED FOR HTTP PORT, RC=n ERRNO=n**. With EADDRINUSE it tries again up to **BIND_TRIES** times. Before each try it writes **HTTPD030I EADDRINUSE, WAITING FOR TCPIP TO RELEASE HTTP PORT n** and waits **BIND_SLEEP** seconds. When no try works, it writes **HTTPD030E** a second time and ends. A bind that fails with another error gets the second **HTTPD030E** without a retry.
5. The server calls listen with the queue length of **LISTEN_QUEUE**. When that fails, it writes **HTTPD031E LISTEN() FAILED, RC=n ERRNO=n** and ends.
6. On success the server writes **HTTPD054I LISTENING ON ANY PORT n**.

Chapter 1, “The HTTPPRM Keywords” on page 1 lists the keywords **PORT**, **BIND_TRIES**, **BIND_SLEEP** and **LISTEN_QUEUE** with their ranges.

Return Codes and Abends

The step ends with return code 0 after a normal stop with **P HTTPD** (Chapter 3, “Console Commands” on page 17). The server issues no user abend. The next table lists every other way that the step ends by the decision of the server.

Table 35. The return codes of the job step

Return code	Cause and message
8	The server refused the start while it read the Parmlib or opened the port. The causes are a member that could not be read to the end (**HTTPD038E**), a route whose policy is incomplete (**HTTPD420E**), a port that another HTTPD serves (**HTTPD037E**), a failed socket call (**HTTPD028E**), a failed bind (**HTTPD030E**) and a failed listen (**HTTPD031E**).

8	The socket thread could not be created (**HTTPD033E**), or it is not running after 5 seconds (**HTTPD064E**).
8	The task has no COM area, so the commands MODIFY and STOP cannot work (**HTTPD090E**).
not 0	The authorization failed. When SVC 244 fails, HTTPD writes **HTTPD012E** and the step ends with the return code of that call. When a later part of the setup fails, the step ends with its return code, for example 4 when the module entry is not found, and the server writes no message and no **HTTPD002I**.

In the first two rows, the server has started its stop sequence when it ends. It writes **HTTPD098I HTTPD SHUTTING DOWN** and **HTTPD099I HTTPD SHUTDOWN COMPLETE**, and the message that names the cause comes before them. The code 8 is what tells a refused start from a normal stop.

The next messages do not change the return code.

Table 36. Messages with no effect on the return code

Message	Effect
HTTPD003W, **HTTPD004W**	The server runs with the inherited identity (“The Identity of the Task” on page 33).
HTTPD020W	The server runs on the built-in defaults.
HTTPD034W	The server runs without worker threads.
HTTPD039W	The server runs with the debug output off.
HTTPD044W	The server runs without a file system.
HTTPD047E	The server runs, and logins do not work.
HTTPD440W, **HTTPD441W**, **HTTPD442W**	The route works. The module is fetched for each request.
HTTPD912E	**TERMINATE FAILED WITH RC=.....**. The stop sequence itself failed. The code of the step stays as it was.

The routine in front of the server (“The standard stream DDs” on page 30) ends with **EXIT_FAILURE** for the messages **HTTPD014E** to **HTTPD019W**.

A worker thread is protected by a recovery routine. These two messages concern an abend while a worker serves a request:

- **HTTPD062E ABEND xxxxxxxx IN WORKER(aaaaaa) CLIENT(aaaaaaaa) SOCKET(n)** names the abend code in hexadecimal. The worker goes on with the next request. The server does not stop.
- **HTTPD063E NO RECOVERY IN WORKER(aaaaaa), CLIENT(aaaaaaaa) SOCKET(n) DROPPED** is written when the recovery routine of the worker could not be set up. The client is dropped without an answer.

The module ABEND0C1 abends on purpose to test this recovery (Chapter 7, “The Display Modules” on page 81). It writes **HTTPD921W ABEND0C1 ALLOCATED n KB IN n BLOCKS, ABENDING WITHOUT FREEING** first. Appendix A, “Messages” on page 99 lists every message with its meaning.

The Package

HTTPD comes as an SMP4 function. The command **`mbt package`** writes the archive. This section is the map of what the archive contains and what the install job does. The *HTTPD User's Guide* explains how to install it, in "Installing HTTPD".

The FMID

The FMID of a release is **`THTP`** and three digits: the major version, the minor version and the patch version. HTTPD 4.4.0 is **`THTP440`**, and HTTPD 4.5.0 is **`THTP450`**. A release spends its FMID once, and the FMID is never used again. No component of the version may exceed 9, because a FMID has 7 characters and no room for a second digit. A patch is a new function level and not a PTF on the old one. The package build makes no PTF.

The build derives the FMID and the predecessor from the version in the project file. Version 4.5.0 gives **`THTP450`** and deletes **`THTP440`**. A patch release needs an explicit FMID.

Each release deletes its predecessor. The SYSMOD carries a **`++VER`** statement with the operand **`DELETE`**, for example **`++VER(Z038) DELETE(THTP430)`** in the package of 4.4.0. SMP keys the ownership of an element on the name of the module and not on the library. A SYSMOD that does not own the module **`HTTPD`** cannot install it, and the install then copies nothing while every condition code reads success. The **`DELETE`** passes the ownership on. SMP deletes the load modules of the old FMID from the target library and copies the new ones in.

The deleted FMID has no backup entry in the SMP data set. For this reason the APPLY of an upgrade ends with return code 4, and the install job lets the ACCEPT run for a code of 4 or less. A return code does not show that the work was done. List the members of **`HTTPD.LINKLIB`** after an install, and look for the message **`HMA2380`** (copy successful).

The names of the data sets carry no version. There is one installation, and an upgrade installs into the same libraries. The allocation job is run once.

The files in the archive

The archive contains these files. The word *`version`* is the version of the release, for example **`4.4.0-dev`** for a development build.

Table 37. The files in the archive

File	Contents
`README.md`	The installation guide of the product.
`httpd-version-load.xmit`	The load modules, as an XMIT file.
`httpd-version-samplib.xmit`	The sample library, as an XMIT file.
`httpd-version-alloc.jcl`	The job that allocates the data sets of SMP.
`httpd-version-inst.jcl`	The job that installs the SYSMOD.
`httpd-webroot.img`	The document root as a UFS370 image. SMP does not know it.

The XMIT files are uploaded in binary, as RECFM=FB with LRECL=80, to data set names of the operator's choice. The install job names them in the operands **`INDSN`** of its two RECEIVE steps, and these are the only lines that the operator changes in it.

The data sets

Table 38. The data sets of the package

Name	DSORG	RECFM	Space	Use
HTTPD.LINKLIB	PO	U	CYL (5,2,20)	The target library of SMP, with BLKSIZE 15040. The STEPLIB of the procedure names it. Only the install job writes to it. The allocation job creates it.
HTTPD.AHTTPLOD	PO	U	CYL (5,2,20)	The distribution library of SMP, with BLKSIZE 15040. The ACCEPT fills it. A RESTORE returns to it. The allocation job creates it.
HTTPD.HTTPLOAD	PO	by RECEIVE	by RECEIVE	The staging library. The install job receives the load XMIT into it and deletes it at the end.
HTTPD.SAMPLIB	PO	by RECEIVE	by RECEIVE	The sample library, created by TSO RECEIVE. SMP does not own it.
HTTPD.WEBROOT.UFS	PS	U	4096-byte blocks (256)	The disk image of the document root, with BLKSIZE 4096. The job **HTTPWEBR** allocates it, and the operator uploads the image into it. SMP does not own it.

The allocation job sets no LRECL for the two load libraries. The two data sets that TSO RECEIVE creates get their attributes from the XMIT files.

The allocation job has no DELETE step, on purpose. Once the FMID is accepted, the distribution library holds the accepted copy of every element. A job that scratched it again would leave the SMP inventory reporting “accepted” with an empty library behind it. The space is in cylinders, because a space in tracks reaches the limit of 16 extents long before the volume is full. The job uses **UNIT=SYSDA**. Change the unit and the volume in the job if you want others.

The install job runs these steps in this order.

Table 39. The steps of the install job

Step	Action
DELOLD	IDCAMS deletes **HTTPD.HTTPLOAD** and **HTTPD.SAMPLIB**. **SET MAXCC=0** lets a first install pass, where neither exists. TSO RECEIVE does not merge into an existing data set, so the step makes the job run more than once.
RECV1	TSO RECEIVE of the load XMIT into **HTTPD.HTTPLOAD**.
RECV2	TSO RECEIVE of the sample XMIT into **HTTPD.SAMPLIB**.
RECV	SMP RECEIVE of the SYSMOD. The SYSMOD is part of the job as in-stream data and contains the JCLIN.
APPLYCHK	SMP APPLY with the operand **CHECK**. It changes nothing. The next step runs only if this step ends with code 0.
APPLY	SMP APPLY. It copies the modules into **HTTPD.LINKLIB**.
ACCEPT	SMP ACCEPT. It runs when the APPLY ended with a code of 4 or less. It makes this level the base of a later RESTORE.

CLEANUP	IDCAMS deletes **HTTPD.HTTPLOAD**. It runs only when the ACCEPT ended with code 0. Comment the step out to keep the library.

The JCLIN of the SYSMOD copies the members with IEBCOPY from **HTTPD.AHTTPLOD** to **HTTPD.LINKLIB** and lists five modules. SMP does not link them again, so every attribute stays as the linkage editor of the build set it, including the authorization code of the server.

The sample library is not part of SMP. The configuration pattern has comment lines that reach column 80, and an MCS card ends at column 72. The product owns the patterns in the library, and the site owns the copies that it makes.

The members

The load library **HTTPD.LINKLIB** has five members. All five are linked reentrant and serially reusable.

Table 40. The members of HTTPD.LINKLIB

Member	Contents
HTTPD	The server. It is the only member that a started task names. It is linked with the authorization code 1 and contains the entry point **CTHREAD**.
HTTPDSRV	The display of the control blocks of the server (Chapter 7, “The Display Modules” on page 81).
HTTPDM	The display of storage (Chapter 7, “The Display Modules” on page 81).
HTTPDMTT	The display of the master trace table (Chapter 7, “The Display Modules” on page 81).
ABEND0C1	The abend test (Chapter 7, “The Display Modules” on page 81).

All five modules ship, because the package build packages every module and has no way to exclude one. A module is inert unless a route of the Parmlib member names it. The line for ABEND0C1 in the sample member is a comment, and so are the lines for the three display modules.

The sample library **HTTPD.SAMPLIB** has three members.

Table 41. The members of HTTPD.SAMPLIB

Member	Contents
HTTPD	The procedure of this chapter. The operator copies it to the procedure library.
HTTPPRM0	The pattern for the Parmlib member (Chapter 1, “The HTTPPRM Keywords” on page 1).
HTTPWEBR	The job that allocates **HTTPD.WEBROOT.UFS** (“The document root disk”).

The document root disk

The file **httpd-webroot.img** is a formatted UFS370 file system with the pages that HTTPD serves from its document root. The operator uploads it with IND$FILE into the data set that the job **HTTPWEBR** allocated, and mounts it with UFSD.

The job allocates the data set with the attributes that the image needs.

- The data set exists before the upload, because IND$FILE takes the DCB from the label of an existing data set and invents one only for a new data set. The job fixes **BLKSIZE=4096**, the block size that the image was formatted with. UFSD compares the two at MOUNT and refuses a mismatch with **UFSD062E SUPERBLOCK VALIDATION FAILED FOR dsn**.

- The space is in the primary extent only. The file system is addressed by block number inside the primary extent, so a block in a secondary extent would never be reached.
- The space is 256 blocks of 4096 bytes, which is the 1 MB of the shipped image. MVS rounds the request up to a track. The superblock and not the data set says how large the file system is.
- The job card has no NOTIFY operand. The value **`NOTIFY=&SYSUID`** is replaced only by TSO SUBMIT and mvsMF, and any other way to submit gets IEF627I and a JCL error.

An image that is larger than the shipped one needs a larger space. Count it in blocks of 4096 bytes. The *HTTPD User's Guide* shows the whole procedure, in "Installing HTTPD".

Chapter 5. HTTP Behaviour and Limits

This chapter describes what HTTPD does with a request, which limits it enforces, and what it sends back. It covers the methods, the limits on the request, the decoding of the URL, the status codes and headers that the server itself produces, keep-alive and time-outs, chunked transfer, static files and their MIME types, character translation, and the limits of the worker model.

The chapter describes the server itself. A module that sends its own status line and headers decides its own response. The server only adds what this chapter names. The keywords that set the time-outs and task counts are in "The HTTPPRM Keywords" (Chapter 1, "The HTTPPRM Keywords" on page 1). The limits of a route line are in "Routes and Modules" (Chapter 2, "Routes and Authentication Policy" on page 9).

How a Request Is Handled

Every request passes the same stages, in this order.

1. The server reads the request line and the header lines from the socket. This stage rejects a request that is too long or malformed ("Limits on the Request" on page 45).
2. It parses the method, the path, the query string and, for a POST, the body. This stage rejects an unknown method and a body that is too large.
3. It resolves the credential of the request and looks for the route that matches the path.
4. The requests for **`/login`** and **`/logout`** are answered by the server before the authentication check. A GET or HEAD for a path that starts with **`/login.`** or **`/favicon.`** skips the check, too.
5. The authentication check of the route gives a 401 or a 403, or lets the request go on.
6. If the route names a program, the server runs that module. If there is no route, or the route is a **`LOC=`** route, the static file handler answers.

Methods

The server knows five methods: DELETE, GET, HEAD, POST and PUT. The comparison is case sensitive, as the method of an HTTP request is, so **`get`** is not the method **`GET`**. Any other method gets a 405. The 405 is sent before the authentication check and before the route lookup, so it is the same for every path. It carries no **`Allow`** header, the connection closes, and the body is **`405 Method Not Allowed`**.

Table 42. Dispatch by method

Method	Route with a program	Static file or `LOC=` route
GET	The module runs.	The server sends the file.
HEAD	The module runs.	The server sends the status and the headers of a GET, and no body.
POST	The module runs.	501 Not Implemented.
PUT	The module runs.	501 Not Implemented.
DELETE	The module runs.	501 Not Implemented.

Any other	405 Method Not Allowed	405 Method Not Allowed.

A route with a program has no restriction by method. The module decides which methods it accepts and sends its own response to the others.

A HEAD for a static file takes the path of a GET as far as the empty line that ends the headers, and stops there. The status and the headers are the ones that a GET sends. **`Content-Length`** is the length of the file, not 0. The response has no body. The client reads the headers and nothing else, so the connection stays usable for the next request.

```
$ curl -I http://localhost:8080/hello.txt
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 17:11:59 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
X-Content-Type-Options: nosniff
Content-Length: 61
```

These rules hold for a HEAD.

- The server opens the file as for a GET. A path with no file gets the 404 of a GET, and a directory is resolved to **`index.html`** or **`default.html`** as for a GET (“The path” on page 52). The 404 has the headers of the GET, and no body.

  ```
  $ curl -I http://localhost:8080/nothere.html
  HTTP/1.1 404 Not Found
  Date: Tue, 27 Oct 2026 17:11:59 GMT
  Server: HTTPD Server
  X-Powered-By: MVSLOVERS
  Connection: keep-alive
  Cache-Control: no-store
  Content-Type: text/plain
  ```

- On a route with **`AUTH=BASIC`** or **`AUTH=TOKEN`**, a HEAD without a valid credential gets the 401 that a GET gets, and a HEAD that the resource check refuses gets the 403. Both have the headers of the GET and no body. A route with **`AUTH=FORM`** answers a HEAD in another way (“Other methods on a route with form login” on page 48).
- The server sends **`Content-Length`** only where a GET sends it (“Response Headers” on page 49). For a file with a server-side include type, which includes every **`htm`** and **`html`** file (“Server-side includes” on page 55), and for a file of 0 bytes there is no length. The HEAD then has neither **`Content-Length`** nor **`Transfer-Encoding`**: a response to a HEAD is never chunked (“Chunked Transfer” on page 52).

  ```
  $ curl -I http://localhost:8080/docs/
  HTTP/1.1 200 OK
  Date: Tue, 27 Oct 2026 17:11:59 GMT
  Server: HTTPD Server
  X-Powered-By: MVSLOVERS
  Connection: keep-alive
  Cache-Control: no-store
  Content-Type: text/html
  X-Content-Type-Options: nosniff
  ```

- On a route with a program the module runs and decides. “Output” on page 68 says what the server does with the output of a module for a HEAD.

Note: HTTPD 4.4.0 and earlier do not answer a HEAD for a static file with the headers of a GET. They answer **`501 Not Implemented`** with no **`Content-Length`**, chunked transfer and the text **`501 Not Implemented`** as the body. A HEAD response must not have a body, so the connection could not be used for the next request. The behaviour in this section is that of HTTPD 4.5.0.

An unknown method gets a 405 and the connection closes.

```
$ curl -i -X OPTIONS http://localhost:8080/hello.txt
HTTP/1.1 405 Method Not Allowed
Date: Tue, 27 Oct 2026 05:41:53 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: close
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

405 Method Not Allowed
```

Limits on the Request

The server reads a request into a buffer of 4000 bytes. Several limits come from that size.

The request line and the header lines

A line can hold at most 3998 characters. The CR in front of the LF is skipped and is not counted. A line that is longer than that is rejected. For the request line, the answer is 414 URI Too Long. For a header line, the answer is 400 Bad Request.

The server rejects a request in these cases, too. All of these responses close the connection, because keep-alive is not yet decided when they are sent.

Table 43. Requests that the server rejects while it reads the headers

Condition	Code	Body
The request line is longer than 3998 characters.	414	**`414 URI Too Long`**
A line has a CR that is not followed by an LF.	400	**`400 Bad Request`**
The request line has no version.	400	**`400 Bad Request`**
The version is not HTTP/1.0 or HTTP/1.1.	505	**`505 HTTP Version Not Supported`**
A header name has a character other than a letter, a digit or one of **`!#$%&'*+-.^_`|~`**.	400	**`400 Bad Request`**
A header value has a control character below the blank, other than the tab.	400	**`400 Bad Request`**
The request has two **`Host`** headers.	400	**`400 Bad Request`**
The **`Content-Length`** is empty or has a character that is not a digit.	400	**`400 Bad Request`**
An HTTP/1.1 request has no **`Host`** header.	400	**`400 Bad Request: Missing Host header`**

All these bodies are **`text/plain`**. A header line without a colon is skipped without a message.

If the request line is empty, or has a method but no URI, the server closes the connection without sending a response.

The headers

There is no limit on the number of header lines. If a header appears twice, the later one replaces the earlier one. The **Host** header is the exception, because two of them are an error.

The server turns the headers and some parts of the request into environment variables for the request. A module reads them with the API (Chapter 6, "The Server Module Interface" on page 59). The server builds the names this way.

- A header becomes **HTTP_** followed by the header name, for example **HTTP_Host**.
- Each cookie of a **Cookie** header becomes **HTTP_Cookie-** followed by the cookie name. The header is split at the semicolons and the blanks.
- A query parameter becomes **QUERY_** followed by the name.
- A field of a form body becomes **POST_** followed by the name.

The server also sets **SERVER_ADDR**, **SERVER_PORT**, **SERVER_PROTOCOL** and **SERVER_SOFTWARE**. The protocol is always **HTTP/1.1**. The software is **HTTPD/** followed by the version.

A name and its value together can have at most 8192 bytes. If a module sets a larger variable, the call fails and the server writes message HTTPD905E (see Appendix A, "Messages" on page 99). A request cannot reach this size by itself, because a line is limited to 3998 characters. If setting a variable fails while the server parses a request, the request fails. On the first request of a connection, the server closes the connection without a response. On a later request, it closes the connection, too.

Parameters

A request can have at most 256 parameters. The query parameters and the form fields of the body count together. A request with more gets a 400 with the body **400 Bad Request**.

The request body

The server reads a body only for a POST. It handles the body this way.

- If the request has a **Transfer-Encoding** header, the server reads nothing, and the body is ignored.
- If there is no **Content-Length**, or the length is 0, there is no body.
- If the length is more than 3999, the answer is 413 Payload Too Large with the body **413 Payload Too Large**. The body is not read, and the connection closes. This does not apply to a route with a program: the server does not read the body there either, and does not answer 413, but runs the module after the authentication check. The module reads the body from the socket itself, as it does for a PUT. The connection closes after the response.
- Otherwise the server reads the body with one non-blocking read of at most **Content-Length** bytes. If this read fails, the server closes the connection without a response.

The server translates the body from ASCII to EBCDIC. If the content type is **application/x-www-form-urlencoded**, it sets **POST_STRING** to the body, and it sets one **POST_** variable for each field. The values are decoded as described in "URL Decoding and Escaping" on page 47. For any other content type, it sets only **POST_STRING**. The comparison covers the whole value, in any case. A value with a parameter, such as **application/x-www-form-urlencoded; charset=UTF-8**, is therefore "any other content type": the server sets **POST_STRING** and splits no fields. If the request has a body but no **Content-Type** header, the server does not read the body and sets nothing.

A PUT body is not read by the server. A module that handles PUT reads the body from the socket itself.

The 400 and the 413 that the parser sends carry **Cache-Control: no-store** and the type **text/plain**.

The server turns off keep-alive for every request that has a **Content-Length** or a **Transfer-Encoding** header. The response to a POST or a PUT therefore says **Connection: close**.

If the request is HTTP/1.1, has the header **`Expect: 100-continue`** and has a **`Content-Length`** or a **`Transfer-Encoding`** header, the server sends the interim response **`HTTP/1.1 100 Continue`** with an empty line once it has read all the headers. This happens before the authentication check.

URL Decoding and Escaping

The server splits the request target at the first question mark. The text in front is the path. The text behind is the query string. The server stores the raw query string in **`QUERY_STRING`**.

The path. The server decodes the path and stores the result in **`REQUEST_PATH`**. The decoding turns each **`%xx`** into the character for that byte. A plus sign stays a plus sign in the path. In the value of a query field it becomes a blank.

The query. The server splits the query string at each ampersand. It splits each field at the first equal sign. The name is not decoded. The value is decoded. A field without an equal sign gets an empty value.

Rules of the decoding.

- The hexadecimal value of an escape goes through the code page table of the server (“Character Translation” on page 57).
- A percent sign that is not followed by two hexadecimal digits stays as it is. So **`%zz`** and a percent sign at the end of the text are kept in the result.
- An escape that gives the byte 0 stays as it is, too: **`%00`** does not cut the path or the value, and no NUL character reaches **`REQUEST_PATH`** or a **`POST_`** variable.

Escaping in output. The server has one function that escapes text for HTML. It replaces the characters **`&`**, **`<`**, **`>`**, **`"`** and **`'`** with **`&`**, **`<`**, **`>`**, **`"`** and **`'`**. It always ends the result with a NUL character, and it cuts the result if the buffer is too small. The server uses it for the value of the **`Sec-Uri`** cookie when it writes the login form. The text of a 404 is **`text/plain`** and is not escaped.

Status Codes

The server itself sends these status codes. The body of every error response is one line of **`text/plain`**.

Table 44. Status codes that the server sends

Code	Phrase	When the server sends it
100	Continue	The interim response to **`Expect: 100-continue`** (“The request body” on page 46).
200	OK	A static file, the login form, the logout, and the pages of the display modules.
302	Moved Temporarily	A GET on a route with form login and no session. The response has **`Location: /login`** and a **`Set-Cookie`** header for the cookie **`Sec-Uri`**. Only GET gets a 302 (“Other methods on a route with form login” on page 48).
303	See Other	A successful login by POST. The response has a **`Location`** header and a **`Set-Cookie`** header for the cookie **`Sec-Token`**.
400	Bad Request	A malformed request (“The request line and the header lines” on page 45, “Parameters” on page 46).
401	Unauthorized	The route needs a credential, and the request has none or a wrong one. The body is **`401 Unauthorized`**.

403	Forbidden	The user is known, but the resource check of the route denies access. The body is **`403 Forbidden`**.
404	Not Found	No file for the path. The body is **`404 document "path" not found`**, with the decoded path.
405	Method Not Allowed	A method that the server does not know (“Methods” on page 43).
413	Payload Too Large	A body of more than 3999 bytes, for a route without a program (“The request body” on page 46).
414	URI Too Long	A request line of more than 3998 characters.
500	Internal Server Error	The server cannot build the response, for example because it cannot set the file name of a script, or a module asks for a status code that the server does not know. The body is **`500 Internal Server Error`**.
501	Not Implemented	POST, PUT or DELETE for a static file.
503	Service Unavailable	A module does not run. The body is **`External program`** followed by the module name and the reason. If the module has already sent a status line, the text is appended to its output. See “When a Module Does Not Run” in Chapter 2, “Routes and Authentication Policy” on page 9.
505	HTTP Version Not Supported	A version other than HTTP/1.0 and HTTP/1.1.

The 401 adds a **`WWW-Authenticate`** header for a route with Basic authentication. It does not add it for a route with token or form authentication. It does not add it if the request has the header **`X-CSRF-ZOSMF-HEADER`**, because the browser would open its own credential dialog.

Other methods on a route with form login

On a route with form login and no session, every method that the server knows goes to the login step, not only GET. A GET gets the 302. A POST is treated as a login attempt, whatever its path. A PUT or DELETE gets the status 200 and a **`text/plain`** body **`Invalid REQUEST_METHOD "PUT"`**, with the method that the client sent. A HEAD gets the same status 200 and the header **`Content-Type: text/plain`**, and no body. This holds for a HEAD for **`/login`** and **`/logout`** too.

The phrase table

A module calls the server to write the status line. The server knows the phrases of these 26 codes only.

Table 45. The status codes that the server can write

Code	Phrase	Code	Phrase
200	OK	404	Not Found
201	Created	405	Method Not Allowed
202	Accepted	409	Conflict
204	No Content	410	Gone
206	Partial Content	412	Precondition Failed
301	Moved Permanently	413	Payload Too Large
302	Moved Temporarily	414	URI Too Long
303	See Other	429	Too Many Requests

304	Not Modified	500	Internal Server Error
400	Bad Request	501	Not Implemented
401	Unauthorized	502	Bad Gateway
403	Forbidden	503	Service Unavailable
		505	HTTP Version Not Supported
		507	Insufficient Storage

For any other code, the server writes message HTTPD909E, **`UNSUPPORTED HTTP STATUS`** followed by the code and **`REQUESTED, SENT 500`**, and it sends a 500 instead.

Response Headers

When a module or the server asks for a status line, the server writes these lines first, in this order.

Table 46. The lines the server writes with every status line

Line	Content
Status line	**`HTTP/1.1`** or **`HTTP/1.0`**, the same as in the request. A request without a version gets **`HTTP/1.0`**.
`Date`	The current time in GMT, in the form **`Tue, 27 Oct 2026 05:41:42 GMT`**.
`Server`	**`HTTPD Server`**. The text is a constant and has no version.
`X-Powered-By`	**`MVSLOVERS`**.
`Connection`	**`keep-alive`** or **`close`** (“Keep-Alive and Time-outs” on page 50).

The server adds more headers in these cases.

- **A static file.** The server adds **`Cache-Control: no-store`** if the MIME type contains the text **`html`** or the path ends in **`.html`**. Then it writes **`Content-Type`**, **`X-Content-Type-Options: nosniff`** and **`Content-Length`**. The length is written only if the server can read the size of the file, the size is not 0, and the file is not an SSI file (“Server-side includes” on page 55).
- **An error that the server sends** (404, 405, 501, the 400 and the 413 of the parser, 401 and 403). The server adds **`Cache-Control: no-store`** and **`Content-Type: text/plain`**. The 500 of the server has a content type but no **`Cache-Control`**. The 414, the 505 and the 400 of the header reader have a content type but no **`Cache-Control`**, too.
- **A response with no length.** See “Chunked Transfer” on page 52.

The server supports no range requests and no conditional requests. It ignores the headers **`Range`**, **`If-Modified-Since`** and similar headers, and it sends the whole file with a 200.

```
$ curl -i -H "Range: bytes=0-9" http://localhost:8080/hello.txt
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:41:53 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
X-Content-Type-Options: nosniff
Content-Length: 61

Hello from the HTTPD document root.
This file is plain text.
```

```
$ curl -i -H "If-Modified-Since: Tue, 27 Oct 2026 00:00:00 GMT" http://localhost:8080/hello.txt
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:41:53 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
X-Content-Type-Options: nosniff
Content-Length: 61

Hello from the HTTPD document root.
This file is plain text.
```

Keep-Alive and Time-outs

Keep-alive

The server decides after it has read the request headers.

- An HTTP/1.1 request is kept alive unless it has **`Connection: close`**.
- An HTTP/1.0 request is closed unless it has **`Connection: keep-alive`**.
- A request with a **`Content-Length`** or a **`Transfer-Encoding`** header is closed (“The request body” on page 46).
- If the connection has already served **`KEEPALIVE_MAX`** requests, the response to the next request says **`Connection: close`**, and the server closes the connection after it.

After a response, the server frees the request and waits for the next one on the same connection. The response header **`Connection`** shows the decision.

The next screens are a run with **`KEEPALIVE_MAX=2`**. The first two responses say **`keep-alive`**. The third says **`close`**.

```
PORT=8080
DOCROOT=/www
KEEPALIVE_MAX=2
KEEPALIVE_TIMEOUT=5
```

```
$ curl -sv -o /dev/null -o /dev/null -o /dev/null http://localhost:8080/hello.txt http://localhost:8080/hello.txt http://localhost:8080/
hello.txt 2>&1 | grep -E "Connected|Re-using|Connection|^< HTTP|Closing|keep-alive|Keep-Alive|left intact"
* Connected to localhost (::1) port 8080
< HTTP/1.1 200 OK
< Connection: keep-alive
* Connection #0 to host localhost left intact
* Re-using existing http: connection with host localhost
< HTTP/1.1 200 OK
< Connection: keep-alive
* Connection #0 to host localhost left intact
* Re-using existing http: connection with host localhost
< HTTP/1.1 200 OK
< Connection: close
```

An HTTP/1.0 request gets a closed connection and a **`Content-Length`**.

```
$ curl -i --http1.0 http://localhost:8080/hello.txt
HTTP/1.0 200 OK
Date: Tue, 27 Oct 2026 05:41:53 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: close
Content-Type: text/plain
X-Content-Type-Options: nosniff
Content-Length: 61

Hello from the HTTPD document root.
This file is plain text.
```

Time-outs

The server waits for each line of a request for a limited time. The timer starts again for each line. The limit is **KEEPALIVE_TIMEOUT** seconds for the second and each later request of a connection. It is **CLIENT_TIMEOUT** seconds for the first request.

When the time is up, the server closes the connection. It sends no HTTP response. If the server has received at least one character of the request and **CLIENT_TIMEOUT_MSG** is on, it writes a line to the console. If **CLIENT_TIMEOUT_DUMP** is on, a dump of the received text follows. Both keywords are on by default. The line has no message ID.

The next screens show a client that sends a part of a request line and then waits. The Parmlib sets the time-out to 3 seconds.

```
PORT=8080
DOCROOT=/www
CLIENT_TIMEOUT=3
CLIENT_TIMEOUT_MSG=1
CLIENT_TIMEOUT_DUMP=1
```

```
$ exec 3<>/dev/tcp/localhost/8080
$ printf 'GET /hello.tx' >&3
$ sleep 6
$ exec 3>&-
```

```
client 00144FC8 on socket 1 timed out after 3 seconds
Dump of 00145028 "Receive Buffer" (13 bytes)
+00000 C7C5E340 61888593 93964BA3 A7       :GET /hello.tx   :
```

The defaults and the allowed values are these. “The HTTPPRM Keywords” (Chapter 1, “The HTTPPRM Keywords” on page 1) has the keywords in full.

Table 47. Defaults and ranges of the time-out keywords

Keyword	Default	Range
CLIENT_TIMEOUT	10	1 to 255. Zero, a negative value and text that is not a number are ignored, and the default stays.
KEEPALIVE_TIMEOUT	5	1 to 255.
KEEPALIVE_MAX	100	1 to 255.

A stalled send

When the client does not take data, the server tries again every 0.1 second. It gives up after 100 tries without progress, that is, after 10 seconds. It then abandons the client and writes message HTTPD065W. The server writes this message at most once in 60 seconds. It counts the messages that it suppresses and shows the count in the next one. A server that is stopping does not wait for the 10 seconds.

The limit holds for a chunked response and for the body of a static file that the server sends itself. After the server has given up, it does not keep the connection for another request, and the worker is free for the next client.

Note: HTTPD 4.4.0 and earlier have this limit for a chunked response only. A client that stops reading a static file keeps its worker (httpd440). The behaviour in this section is that of HTTPD 4.5.0.

If the server cannot read a file from the UFS in the middle of a response, it writes message HTTPD045E, cuts the response, and closes the connection. The server writes this message at most once in 60 seconds, too, and the next one shows the count of the errors that it held back. For an error

in a file that a server-side include pulled in (“Server-side includes” on page 55), the message names that file.

Chunked Transfer

The server uses chunked transfer when it does not know the length of a response. When the server writes the empty line that ends the headers, it adds the header **`Transfer-Encoding: chunked`** if all of these are true.

- The request is HTTP/1.1.
- The response has no **`Content-Length`** header. The server finds out by looking for a header line that starts with **`Content-Length:`**.
- The status is not 1xx, not 204 and not 304.
- The request is not a HEAD. A response to a HEAD has no body, so it is not chunked and has no final chunk.

Each chunk is the length in hexadecimal digits, a CR LF, the data, and a CR LF. The server ends the response with a chunk of length 0. If the server cannot send a chunk, it turns off the chunk framing and keep-alive. An HTTP/1.0 response is never chunked.

This applies to the responses of modules, too. A module only has to send a **`Content-Length`** header if it wants to avoid chunking. The responses of the server to errors, such as the 404 in the next screen, are sent chunked.

```
$ curl -i http://localhost:8080/nothere.html
HTTP/1.1 404 Not Found
Date: Tue, 27 Oct 2026 05:41:42 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

404 document "/nothere.html" not found
```

Static Files

The server reads static files from the UFS only. It does not read them from a data set. If the server has no UFS, it finds no file, and every request for a static path gets a 404. A static file is served for a GET, and a HEAD gets the headers of the GET (“Methods” on page 43). POST, PUT and DELETE get a 501.

The path

The server joins the document root and the decoded path. It applies these rules.

- A path with two dots in a row, anywhere in it, is refused with a 404.
- The server cuts the path to 255 characters, and the joined name to 255 characters. It does not tell the client.
- An empty path gets a 404.
- If the path ends in a slash, the server tries **`index.html`** and then **`default.html`** in that directory. If neither exists, or the name does not fit into the buffer of 300 bytes, the answer is a 404.
- If the path names a directory and does not end in a slash, the answer is a 404.

```
$ curl -i http://localhost:8080/docs/
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:41:42 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
```

```
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/html
X-Content-Type-Options: nosniff
Transfer-Encoding: chunked

<!DOCTYPE html>
<html><head><title>Documents</title></head>
<body><h1>Documents</h1>
<p>This is docs/index.html.</p></body></html>
```

```
$ curl -i http://localhost:8080/docs
HTTP/1.1 404 Not Found
Date: Tue, 27 Oct 2026 05:41:42 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

404 document "/docs" not found
```

```
$ curl -i http://localhost:8080/empty/
HTTP/1.1 404 Not Found
Date: Tue, 27 Oct 2026 05:41:42 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Cache-Control: no-store
Content-Type: text/plain
Transfer-Encoding: chunked

404 document "/empty/" not found
```

MIME types

The server takes the type from the extension of the file name. The extension is the text after the last dot in the last part of the path. The comparison ignores case. A file name with no extension, or with an extension that is not in the table, gets **`text/plain`** and is sent as text. A file **`data.json`** is an example.

```
$ curl -i http://localhost:8080/data.json
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:41:42 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/plain
X-Content-Type-Options: nosniff
Content-Length: 34

{"name": "httpd", "static": true}
```

The table has a column “Body”. **Text** means that the server opens the file as text and translates it from EBCDIC to ASCII. **Binary** means that the server opens the file in binary mode and sends the bytes as they are. The table lists the types as they are in the server, also the ones where the setting is not what the name of the type suggests (for example **`sh`** and **`tex`** are binary).

Table 48. The MIME types of the server

Extensions	Content-Type	Body
`ai`	**`application/postscript`**	Binary
`aif`, **`aifc`**, **`aiff`**	**`audio/x-aiff`**	Binary
`au`, **`snd`**	**`audio/basic`**	Binary
`avi`	**`video/x-msvideo`**	Binary
`bcpio`	**`application/x-bcpio`**	Binary
`bin`	**`application/octet-stream`**	Binary

c, **h**, **lua**, **rex**, **rexx**, **txt**	**text/plain**	Text
cdf, **nc**	**application/x-netcdf**	Binary
cpio	**application/x-cpio**	Binary
csh	**application/x-csh**	Binary
css	**text/css**	Text
dvi	**application/x-dvi**	Binary
eps, **ps**	**application/postscript**	Text
etx	**text/x-setext**	Text
gif	**image/gif**	Binary
gtar	**application/x-gtar**	Binary
hdf	**application/x-hdf**	Binary
htm, **html**	**text/html**	Text
ico, **icon**	**image/x-icon**	Binary
ief	**image/ief**	Binary
jpe, **jpeg**, **jpg**	**image/jpeg**	Binary
js	**application/x-javascript**	Text
latex	**application/x-latex**	Binary
lod	**application/x-load**	Binary
man	**application/x-troff-man**	Text
me	**application/x-troff-me**	Text
mif	**application/x-mif**	Binary
mov, **qt**	**video/quicktime**	Binary
movie	**video/x-sgi-movie**	Binary
mpe, **mpeg**, **mpg**	**video/mpeg**	Binary
ms	**application/x-troff-ms**	Binary
oda	**application/oda**	Binary
pbm	**image/x-portable-bitmap**	Binary
pdf	**application/pdf**	Binary
pgm	**image/x-portable-graymap**	Binary
png	**image/png**	Binary
pnm	**image/x-portable-anymap**	Binary
ppm	**image/x-portable-pixmap**	Binary
ras	**image/x-cmu-raster**	Binary
rgb	**image/x-rgb**	Binary
rng, **xml**	**text/xml**	Text
roff, **t**, **tr**	**application/x-troff**	Binary
rtf	**application/rtf**	Binary

rtx	**text/richtext**	Binary
sh	**application/x-sh**	Binary
shar	**application/x-shar**	Binary
shtm, **shtml**	**text/x-server-parsed-html**	Text
src	**application/x-wais-source**	Text
sv4cpio	**application/x-sv4cpio**	Binary
sv4src	**application/x-sv4crc**	Binary
svg	**image/svg+xml**	Text
tar	**application/x-tar**	Binary
tcl	**application/x-tcl**	Binary
tex	**application/x-tex**	Binary
texi, **texinfo**	**application/x-texinfo**	Binary
tif, **tiff**	**image/tiff**	Binary
tsv	**text/tab-separated-values**	Text
ustar	**application/x-ustar**	Binary
wav	**audio/x-wav**	Binary
xbm	**image/x-xbitmap**	Binary
xhtml	**application/xhtml+xml**	Text
xpm	**image/x-xpixmap**	Binary
xsl, **xslt**	**text/xsl**	Text
xwd	**image/x-xwindowdump**	Binary
zip	**application/zip**	Binary

Two screens show the headers of a CSS file and of a PNG file.

```
$ curl -i http://localhost:8080/css/site.css
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:41:42 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: text/css
X-Content-Type-Options: nosniff
Content-Length: 67

body { font-family: sans-serif; margin: 2em; }
h1 { color: #036; }
```

```
$ curl -s -D - -o /dev/null http://localhost:8080/logo.png
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2026 05:41:43 GMT
Server: HTTPD Server
X-Powered-By: MVSLOVERS
Connection: keep-alive
Content-Type: image/png
X-Content-Type-Options: nosniff
Content-Length: 70
```

Server-side includes

The server processes server-side includes in a file with the MIME type **text/html** or **text/x-server-parsed-html**. These are the extensions **htm**, **html**, **shtm** and **shtml**. The response is

chunked on HTTP/1.1, because it has no **Content-Length**. The directory index in "The path" on page 52 shows this.

The server knows three directives. A directive starts with **<!--#** and ends at the first blank-and-marker, **-->**, so a blank must stand in front of the closing marker. A line can hold several directives.

Table 49. Server-side include directives

Directive	Effect
<!--#echo var="X" -->	Inserts the value of the request variable X. If there is none, the server looks for an environment variable of its own, of that name (see below). If there is neither, it inserts the comment **<!-- Oops: we didn't find var="X" -->**.
<!--#include virtual="path" -->	Inserts the file. The form with **file=** is the same.
<!--#printenv -->	Inserts one HTML table, headed **Server Variables**, with a row for each variable of the request ("The Request Variables" on page 63). It lists nothing of the environment of the server.

The name in an **echo** directive is looked up in two places, in this order. First the server looks at the request variables ("The Request Variables" on page 63). If the request has no variable of that name, the server looks at its own environment, the one that the server task runs with, and inserts the variable of that name from it. So a page can print a single variable of the server environment by name. The name comes from the page, never from the request: a client cannot choose it and cannot make the server read any other variable. The directive can therefore be used only by someone who can put a page into the document root. A client cannot do it. The directive **printenv** never reads the server environment.

Note: The server drops the second look in HTTPD 4.6.0 (httpd421). From that version **echo** reads the request variables only, and a variable that is not among them gives the **Oops** comment. A page that prints a variable of the server environment must not rely on this.

The server writes every value that **echo** inserts, and every name and value that **printenv** lists, as HTML text. The five characters **&**, **<**, **>**, **"** and **'** become the entities **&**, **<**, **>**, **"** and **'**. A header value from the client therefore reaches the page as text and not as markup. The server never cuts a value to do this. Text that comes from the page itself, from the file or from a file that it includes, is sent as it is.

Text that the server does not recognize as a directive is sent as it is. That covers an unknown directive name and a missing blank in front of the closing marker. A directive that has a known name but a bad argument is not sent as it is. The server writes a comment in its place: **<!-- Unable to parse 'echo' request -->** or **<!-- Unable to parse 'include' request -->**.

Includes can be nested up to 10 levels. Below that level the server writes **<!-- Oops: Maximum Server Side Include levels exceeded -->** in place of the include. An include of a file that does not exist gets **<!-- Oops: we didn't find "path" -->**, and an include of a file with a binary MIME type gets a comment that starts with **<!-- Oops: we can't include a binary file**. The comment that the server writes in place of a directive is limited to 4095 characters. A value that **echo** or **printenv** writes is not cut at that limit. A read error in an included file ends the response as it does in the page itself ("A stalled send" on page 51). The includes that follow on the same line are not opened.

The server reads the file line by line and looks for directives in each line. A line can be of any length. The server reads a long line in pieces and sends it as one line. A directive that a read cuts in two is joined again, and the server recognizes it, if it is at most 384 characters long. A longer directive that a read cuts in two is not recognized in every case. The server then sends it as text, like any other text

that is not a directive. The same happens when the output buffer of the response is still full after the server has tried to send it, at the moment the rest of a long line is due: the part of the line that was held back goes out as text, and a directive that spans that cut is not recognized.

Character Translation

The network carries ASCII. The server works in EBCDIC. The Parmlib keyword **CODEPAGE** selects the table: **CP037** (the default), **IBM1047** or **LEGACY**. The name can be in any case. An unknown name gives message HTTPD070E, and the server uses **CP037**. See "Files and Character Set" in Chapter 1, "The HTTPPRM Keywords" on page 1.

The server translates from ASCII to EBCDIC with that table in these places.

- The request line and the header lines.
- The escapes **%xx** of the path and of the query values.
- The body of a POST.
- The Basic credentials.

It translates from EBCDIC to ASCII with that table in these places.

- Everything that is written with the printf function of the server. This is the status line, the headers, the chunk lengths, the texts of the error responses, the login pages and the pages of the display modules.

Two kinds of data are not translated with the table of the server.

- **A static file.** A file of the type "Text" in the MIME table is translated with the code page **IBM-1047**, whatever the keyword says. A file of the type "Binary" is not translated.
- **Data a module sends with the send function.** The server sends these bytes as they are. The module has to translate them.

Workers and Threads

The server has one socket thread and a pool of worker threads. The socket thread has a stack of 32 KB. Each worker has a stack of 64 KB.

The socket thread waits for a connection. It accepts the connection, sets the socket to non-blocking, allocates the control block of the client, and puts it in the queue of the pool. A worker takes one connection from the queue and serves it until the connection is closed. A connection with keep-alive therefore holds one worker until the connection closes, also while it is idle. It holds the worker for at most **KEEPALIVE_TIMEOUT** seconds between two requests.

The pool has at least **MINTASK** and at most **MAXTASK** workers. The defaults are 3 and 9. Both values can be 1 to 255 in the Parmlib. A value of 0 or below is ignored. The console commands **SET MAXTASK** and **SET MINTASK** change the values while the server runs, in the range 1 to 100 (see "Console Commands" in Chapter 3, "Console Commands" on page 17). The Parmlib keywords are in Chapter 1, "The HTTPPRM Keywords" on page 1.

There is no limit on the queue of connections that wait for a worker, except the storage of the region. The keyword **LISTEN_QUEUE** sets the length of the queue of the TCP/IP stack for the listener. If the pool is stopping, the server closes the new connection without a response.

Table 50. Events in the socket thread and the workers

Message	Cause
HTTPD052E	The accept call fails.

HTTPD053E	The server cannot set the socket to non-blocking.
HTTPD999E	The server cannot allocate the control block of the client. It closes the socket.
HTTPD901E	A worker has gone through its loop 1000 times for one connection without progress.
HTTPD062E	A worker abends. The recovery closes the connection, and the worker carries on.
HTTPD063E	The recovery for a worker could not be set up.

The texts of the messages are in Appendix A, “Messages” on page 99.

Other Hard Limits

Table 51. Fixed limits of the server

Item	Limit
A request line or header line	3998 characters (“The request line and the header lines” on page 45)
Header lines in a request	No limit.
Parameters of a request	256 (“Parameters” on page 46)
Body of a request that the server reads	3999 bytes (“The request body” on page 46)
Name and value of an environment variable	8192 bytes together
Output of one call of the printf function of the server	5119 characters. The server cuts the rest.
Path of a static file	255 characters, and 255 for the joined name
Nesting of server-side includes	10 levels
Routes	See Chapter 2, “Routes and Authentication Policy” on page 9.
Statistics counters	The counters for requests, errors, bytes sent and active connections are 32-bit unsigned numbers. They wrap at 4294967295.

The counter of the errors counts each response with a status of 400 or more. The counter of the bytes counts what the server hands to the send function. That includes the headers and the chunk framing. A response that does not reach its end, for example after a time-out, is not counted. The console command **`D STATS`** shows the counters (see “Console Commands” in Chapter 3, “Console Commands” on page 17).

The read of the request

The server reads the request line and the headers one byte at a time, because the TCP/IP stack of MVS 3.8j corrupts data when a read of several bytes crosses the wrap of its ring buffer.

Chapter 6. The Server Module Interface

A server module is a load module that HTTPD runs for a route. The route names the module with the keyword **MOD=**, as described in Chapter 2, "Routes and Authentication Policy" on page 9. The module receives the request, writes the response and returns. HTTPD itself needs no more from it.

This chapter is the contract between a module and the server. It describes how HTTPD starts a module and what the module finds when it runs, the control blocks that it may read, every entry of the function vector (HTTPX), the variables of a request, the storage rules, what happens when a module abends, and what the started task needs for a module to work. The header **httpcgi.h** declares everything in this chapter. A module includes it and nothing else of HTTPD.

The *HTTPD User's Guide* shows how to build and install a module, in "Writing a Server Module". The four modules that come with HTTPD are in Chapter 7, "The Display Modules" on page 81.

How a Module Is Run

HTTPD runs a module with the LINK SVC, in the task of the worker that serves the request. It runs the module once per request, whatever the method is: GET, HEAD, PUT, POST and DELETE all arrive at the module in the same way. The module has to complete the response before it returns. When it returns, HTTPD ends the request, whatever the return code is.

The LINK is made through the C library routine **__linkdsx**. The routine sets an ESTAE first, so that an abend of the module comes back to the server as a result and does not end the worker (see "Abends" on page 77). The module name is the name in the route. If the server holds the module in storage, the LINK finds that copy (see "Resident Modules" on page 79).

Before the module runs, HTTPD has read the request, parsed it into variables (see "The Request Variables" on page 63), looked for a credential and found the route. It then applies the authentication rule of the route. A request that fails the rule is answered with 401 or 403 by the server and never reaches the module. HTTPD itself answers the paths **/login** and **/logout**. Only after these steps does the server run the module of a route that has one (**MOD=**). A route without a module (**LOC=**) serves static files.

The Parameter List

HTTPD builds a parameter list of three words for the module.

Table 52. The parameter list of a module

Word	Content	Meaning
0	address of a parameter	A halfword length and the text of the parameter: the request path in double quotes, for example **"/zosmf/info"**. The text has at most 254 characters, the quotes included; a longer path is cut.
1	address of HTTPD	The server control block, with the eye catcher ***HTTPD***.
2	address of HTTPC	The control block of the client, with the eye catcher ***HTTPC***. The high bit is set to mark the end of the list.

A module does not read this list. The startup code that every module is linked with, **cgistart**, reads it.

What the Startup Code Does

cgistart supplies the entry point **__start** of the module. The module is linked with it; it comes from the CGI library of HTTPD. In this order it does the following.

1. It scans the list for the two eye catchers and stores the address of HTTPD and the address of HTTPC in the global runtime table (GRT) of the task, in the fields **grtapp1** and **grtapp2**.
2. If the module is run outside HTTPD, it stops here. It writes "is an HTTPD server module and cannot run on its own" and ends with the exit status **EXIT_FAILURE**. The display modules of Chapter 7, "The Display Modules" on page 81 write the message **HTTPD920E** and return 12 instead.
3. It opens **stdout** on the DD **HTTPDOUT**, **stderr** on the DD **HTTPDERR** and **stdin** on the DD **HTTPDIN**. If **HTTPDIN** is missing, it opens **'NULLFILE'** for **stdin**. The started task procedure has the three DDs (see "The Started Task Procedure" on page 78).
4. It loads environment variables from the DD **SYSENV**. If there is no such DD, it tries the DD **ENVIRON**.
5. It calls **tzset()**.
6. It builds **argv** from the parameter of word 0 and calls **main(argc, argv)**. Around the call it switches the heap of the module to the request subpool (see "Storage Rules" on page 76).

The module finds in **main**:

argc	2.
argv[0]	The name of the module, up to 8 characters.
argv[1]	The decoded **REQUEST_PATH**, without the quotes.
HTTPD, HTTPC	Read **grt->grtapp1** and **grt->grtapp2**. The server has the functions **cgihttpd()** and **cgihttpc()** for this lookup, but they are not part of the client library, and a module cannot link them.
HTTPX	Call **http_get_httpx(httpd)**.

printf needs a note. The startup code contains its own **printf**, and it takes the place of the library function in a module. When the module runs under HTTPD, **printf** sends its output as part of the response, like **http_printv**. It does not write to **stdout**. A call of **fprintf(stdout, ...)** writes to **HTTPDOUT**.

The module must have a C stack that is smaller than the default. The default of the runtime asks for 256 KB of contiguous storage in subpool 0 for every request. The modules of HTTPD set a smaller one in their own source file:

```
unsigned __stklen = 64*1024;
```

Put this variable in the source file of the module and not in a library.

The Return Code

When **main** returns, the startup code clamps a negative value to 0 and exits with the value. HTTPD takes the return code in this way:

- A value of 0 or more is not an error. The server does not look at it.
- An abend, a module that cannot be found or loaded, and a few other failures are errors. The server answers them as described in "Abends" on page 77.

After the module has returned, the server sets the state of the client to **CSTATE_DONE**. It closes the files, writes the statistics (see Chapter 8, "SMF Records" on page 93) and then resets the client or closes the connection. The module does not call any of these steps.

A Minimal Module

This module answers every request with a line of text.

```
#include <stdio.h>
#include <mvs/crt.h>
#include "httpcgi.h"

unsigned __stklen = 64*1024;

int main(int argc, char **argv)
{
    CLIBGRT *grt   = __grtget();
    HTTPD   *httpd = grt->grtapp1;
    HTTPC   *httpc = grt->grtapp2;
    HTTPX   *httpx = http_get_httpx(httpd);

    http_resp(httpc, 200);
    http_printf(httpc, "Content-Type: text/plain\r\n");
    http_printf(httpc, "\r\n");
    http_printf(httpc, "Hello from %s\n", argv[1]);
    return 0;
}
```

The macros of **`httpcgi.h`** expand into a call through the local variable **`httpx`**. The variable must be in scope where a macro is used. The modules in the HTTPD source that include **`httpd.h`** instead define **`httpx`** as **`(httpd->httpx)`**.

The Structures

The header **`httpcgi.h`** defines the structures that a module may look into: HTTPC, HTTPV and HTTPM, and the enumerations CSTATE and RDW. It leaves the others opaque: HTTPD, CRED, UFS, UFSFILE, CIB and HTTPROUTE. A module holds pointers to them and passes them to the vector functions. It does not look inside them. The two accessors that read HTTPD are in “The Direct Accessors” on page 75.

HTTPC, the Client

HTTPC is 4096 bytes long. HTTPD allocates it when it accepts a connection and frees it when the connection ends. The structure belongs to the server. A module reads the fields and lets the vector functions change them.

Table 53. The fields of HTTPC

Offset	Name	Type	Meaning
00	**`eye`**	char[8]	**`*HTTPC*`**.
08	**`httpd`**	**`HTTPD *`**	The server.
0C	**`env`**	**`HTTPV **`**	The array of request variables. Use the variable functions of “Request Variables” on page 69 and do not walk the array.
10	**`state`**	short	The state of the request (see below).
12	**`subtype`**	UCHAR	A state of the server’s own file transfer. Do not use it.
13	**`substate`**	UCHAR	As **`subtype`**.
14	**`socket`**	int	The socket of the client.
18	**`addr`**	unsigned	The IP address of the client. **`http_ntoa`** converts it.
1C	**`port`**	int	The port of the client.

20	**fp**	**FILE ***	The file of the response, if the server opened one. **http_done** closes it.
24	**len**	unsigned	The length of data in **buf**.
28	**pos**	unsigned	The position of the next unsent byte in **buf**.
2C	**sent**	unsigned	The bytes sent to the client for this request. It counts every raw byte, the status line, the headers and the chunk framing included.
30	**start**	double	The start time of the request in seconds.
38	**end**	double	The end time of the request in seconds.
40	**rdw**	UCHAR	A value of RDW. **http_printf** also uses this byte as the flag "the headers have ended" (see "Output" on page 68).
41	**chunked**	UCHAR	1 if the response is sent in chunks.
42	**resp**	short	The status code. It is 0 until **http_resp** has been called.
44	**cred**	**CRED ***	The credential of the client, or NULL. A module does not use it directly. Use the functions of "Authentication" on page 72.
48	**ufs**	**UFS ***	The UFS session of the client, or NULL. **http_get_ufs** creates it.
4C	**ufp**	**UFSFILE ***	The UFS file that **http_open** opened, or NULL.
50	**ssi**	UCHAR	1 if the file is processed for server side includes.
51	**ssilevel**	UCHAR	The nesting level of the includes. The maximum is 10.
52	**content_length_set**	UCHAR	1 if a **Content-Length** header was sent.
53	**keepalive**	UCHAR	1 if the connection stays open after this response.
54	**connect_time**	unsigned	The time of the connect in 1/100 s (SMF time).
58	**total_bytes_sent**	unsigned	The bytes sent on this connection, over all requests.
5C	**request_count**	unsigned	The requests answered on this connection.
60	**buf**	UCHAR[4000]	The work buffer of the server for input and file transfer. A module does not keep data in it.

The header comments for **subtype** and **substate** give the offsets 11 and 12. The structure puts them at 12 and 13, because **state** is two bytes long.

The field **state** holds a value of CSTATE:

Table 54. The values of CSTATE

Name	Value	Meaning
CSTATE_IN	0	HTTPD reads the request.
CSTATE_PARSE	1	HTTPD parses the request.

CSTATE_GET	2	The method is GET.
CSTATE_HEAD	3	The method is HEAD.
CSTATE_PUT	4	The method is PUT.
CSTATE_POST	5	The method is POST.
CSTATE_DELETE	6	The method is DELETE.
CSTATE_DONE	10	The response is complete. No more output is accepted.
CSTATE_REPORT	11	HTTPD writes the statistics.
CSTATE_RESET	12	HTTPD resets the client.
CSTATE_CLOSE	13	HTTPD closes the connection.

A module runs in one of the states 2 to 6. It reads the state to see the method. The state becomes **CSTATE_DONE** when the module returns, and also when a send fails because the client has gone. After that **http_printf**, **http_printv** and **http_send** return –1. A module that loops over a long response tests the state or the return codes and stops.

The values of RDW are **RDW_NONE** (0), **RDW_B2**, **RDW_L2**, **RDW_B4** and **RDW_L4**. The server sets **rdw** to 1 itself when the headers end (see "Output" on page 68).

HTTPV, a Variable

A request variable is a block of this form.

Table 55. The fields of HTTPV

Offset	Name	Type	Meaning
00	**eye**	char[8]	***HTTPV***.
08	**value**	**char ***	The value. It points into the same block.
0C	**name**	char[2]	The name, a zero byte, the value and a zero byte, one after the other.

HTTPM, a MIME Type

http_mime returns a pointer to this structure of 12 bytes.

Table 56. The fields of HTTPM

Offset	Name	Type	Meaning
00	**ext**	**UCHAR ***	The file extension.
04	**type**	**UCHAR ***	The MIME type, for example **text/html**.
08	**binary**	int	1 if the file is sent as binary, 0 if it is text.

The Request Variables

Before the module runs, HTTPD has read the request line and the headers and has stored them as variables of the client. A module reads them with **http_get_env**. The names are not case sensitive. All values are in EBCDIC.

Table 57. The request variables

Name	Value
HTTP_REQUEST	The request line as it was sent.
REQUEST_METHOD	The method.
REQUEST_URI	The URI as it was sent, with the query.
REQUEST_VERSION	The version of the request: **HTTP/1.0** or **HTTP/1.1**. HTTPD rejects any other with the status 505.
REQUEST_PATH	The path of the URI, decoded, without the query.
QUERY_STRING	The text after the question mark of the URI, not decoded. The variable exists only if the URI has a query.
QUERY_name	One variable for each parameter of the query. The value is decoded, the name is not. A parameter without an equals sign has the value “”.
HTTP_name	One variable for each header of the request, for example **HTTP_Host** or **HTTP_Content-Type**. The name is as the client sent it. The value follows the colon, without leading blanks. The header **Cookie** makes no **HTTP_Cookie** see the next row.
HTTP_Cookie-name	One variable for each cookie. A cookie without an equals sign has the value “”.
POST_STRING	The body of a POST request (see below).
POST_name	One variable for each field of a form body (see below).
SERVER_ADDR	The address of the server end of the connection.
SERVER_PORT	The port of the server end of the connection.
SERVER_PROTOCOL	Always **HTTP/1.1**, also for an HTTP/1.0 request.
SERVER_SOFTWARE	**HTTPD/** and the version of the server.
SCRIPT_FILENAME	The document root and the request path. HTTPD sets it only for a route whose path begins with ***.**, for example ***.lua**. If it cannot set the variable, the request fails with the status 500 and the module does not run.

A request with more than 256 parameters in the query or in a form body gets the status 400 and the connection is closed. The request line must fit in the buffer, else the status is 414. A header with a name that has a character other than a token character, or a value with a control character, gets 400. So does a request with two **Host** headers. A request with a malformed **Content-Length** gets 400. HTTP/1.1 requests without a **Host** header get 400 too. A method other than GET, HEAD, PUT, POST and DELETE gets
405. A request with **Expect: 100-continue** is answered by the server

(HTTP/1.1 only). None of these requests reaches the module.

The body of a request is treated by the method.

POST — HTTPD reads the body with one **recv** if the header **Content-Length** is present and the length is between 1 and 3999. It converts the body to EBCDIC with **http_atoe** and stores it in **POST_STRING**. If the header **Content-Type** is exactly **application/x-www-form-urlencoded**, in any case, it also splits the body into the fields **POST_name** and decodes the values. A body of 4000 bytes or more is

	not read. On a route without a program the status is 413 and the connection is closed. On a route with a program, the module runs after the authentication check, the body is still on the socket, and the connection closes after the response. A request without a body, or with a chunked body, has no **`POST_STRING`**. A request with a body and no **`Content-Type`** is not read either.
PUT	HTTPD does not read the body. The module reads it from the socket in **`httpc->socket`**.
DELETE, GET, HEAD	HTTPD does not read a body.

Every request that has a **`Content-Length`** or a **`Transfer-Encoding`** header ends the connection after the response. HTTPD turns keep-alive off because it cannot be sure that it has read the whole body.

The Function Vector

A module calls the server through the function vector HTTPX. The vector is a structure of function pointers. HTTPD fills it once, when it starts. The address of the structure is in HTTPD at offset 8, and **`http_get_httpx(httpd)`** returns it. The vector is 328 bytes (X'148') long.

The Rules

The vector is append-only. An entry never moves and never changes its meaning. A new function is added at the end. Two consequences follow for a module.

- A module that was built against an older header runs on a newer server.
- A module that was built against a newer header can run on an older server, but an entry that the older server does not have holds 0 there, or lies beyond the end of the vector. There is no function that tells a module which entries exist. The only entry that a module can test before it calls is **`http_cgi_subpool`**: it is 0 on a server that is too old to have the request subpool. Name the lowest HTTPD level of the module in its own documentation.

The tables below give the level in which each entry first appears. The level "4.0" means that the entry is in HTTPD 4.0.0 and every later release. The layouts of earlier releases, HTTPD 3.3 included, are not covered here.

The header declares a macro for every entry. The macro has the name of the entry and expands into a call through the variable **`httpx`**. The names **`http_find_cgi`**, **`http_add_cgi`** and **`http_process_cgi`** are the earlier names of **`http_find_route`**, **`http_add_route`** and **`http_process_route`**. The header keeps them. The entries did not move.

The entries 0x10C and **`http_read`** (0xD4) hold 0 in every release. Do not call them.

The Entries

Table 58. The entries of the function vector

Offset	Entry	Level	Described in
00	**`eye`**	4.0	"The Direct Accessors" on page 75
08	**`httpd`**	4.0	"The Direct Accessors" on page 75
0C	**`http_cgi_subpool`**	4.0	"Storage Rules" on page 76
10	**`http_in`**	4.0	"Entries for the Server Only" on page 74
14	**`http_parse`**	4.0	"Entries for the Server Only" on page 74
18	**`http_get`**	4.0	"Entries for the Server Only" on page 74

1C	**http_head**	4.0	“Entries for the Server Only” on page 74
20	**http_put**	4.0	“Entries for the Server Only” on page 74
24	**http_post**	4.0	“Entries for the Server Only” on page 74
28	**http_done**	4.0	“Entries for the Server Only” on page 74
2C	**http_report**	4.0	“Entries for the Server Only” on page 74
30	**http_reset**	4.0	“Entries for the Server Only” on page 74
34	**http_close**	4.0	“Entries for the Server Only” on page 74
38	**http_send**	4.0	“Output” on page 68
3C	**http_decode**	4.0	“Request Variables” on page 69
40	**http_del_env**	4.0	“Request Variables” on page 69
44	**http_find_env**	4.0	“Request Variables” on page 69
48	**http_set_env**	4.0	“Request Variables” on page 69
4C	**http_new_env**	4.0	“Request Variables” on page 69
50	**http_set_http_env**	4.0	“Request Variables” on page 69
54	**http_set_query_env**	4.0	“Request Variables” on page 69
58	**http_get_env**	4.0	“Request Variables” on page 69
5C	**http_date_rfc1123**	4.0	“Time, MIME Types and Addresses” on page 70
60	**http_printv**	4.0	“Output” on page 68
64	**http_printf**	4.0	“Output” on page 68
68	**http_resp**	4.0	“Output” on page 68
6C	**http_server_name**	4.0	“Output” on page 68
70	**http_resp_not_implemented**	4.0	“Standard Error Responses” on page 69
74	**http_etoa**	4.0	“Translation” on page 70
78	**http_atoe**	4.0	“Translation” on page 70
7C	**array_new**	4.0	“Dynamic Arrays” on page 71
80	**array_add**	4.0	“Dynamic Arrays” on page 71
84	**array_addf**	4.0	“Dynamic Arrays” on page 71
88	**array_count**	4.0	“Dynamic Arrays” on page 71
8C	**array_free**	4.0	“Dynamic Arrays” on page 71
90	**array_get**	4.0	“Dynamic Arrays” on page 71
94	**array_size**	4.0	“Dynamic Arrays” on page 71
98	**array_del**	4.0	“Dynamic Arrays” on page 71
9C	**http_cmp**	4.0	“Comparison” on page 72
A0	**http_cmpn**	4.0	“Comparison” on page 72
A4	**http_dbgw**	4.0	“Debug Output” on page 72
A8	**http_dbgs**	4.0	“Debug Output” on page 72
AC	**http_dbgf**	4.0	“Debug Output” on page 72

B0	**http_dump**	4.0	“Debug Output” on page 72
B4	**http_enter**	4.0	“Debug Output” on page 72
B8	**http_exit**	4.0	“Debug Output” on page 72
BC	**http_secs**	4.0	“Time, MIME Types and Addresses” on page 70
C0	**http_mime**	4.0	“Time, MIME Types and Addresses” on page 70
C4	**http_resp_internal_error**	4.0	“Standard Error Responses” on page 69
C8	**http_resp_not_found**	4.0	“Standard Error Responses” on page 69
CC	**http_open**	4.0	“Files” on page 71
D0	**http_send_binary**	4.0	“Files” on page 71
D4	**http_read** (always 0)	-	“The Rules” on page 65
D8	**http_send_file**	4.0	“Files” on page 71
DC	**http_send_text**	4.0	“Files” on page 71
E0	**http_is_busy**	4.0	“Entries for the Server Only” on page 74
E4	**http_set_busy**	4.0	“Entries for the Server Only” on page 74
E8	**http_reset_busy**	4.0	“Entries for the Server Only” on page 74
EC	**http_process_clients**	4.0	“Entries for the Server Only” on page 74
F0	**http_ntoa**	4.0	“Time, MIME Types and Addresses” on page 70
F4	**http_console**	4.0	“Entries for the Server Only” on page 74
F8	**http_process_client**	4.0	“Entries for the Server Only” on page 74
FC	**http_link**	4.0	“Entries for the Server Only” on page 74
100	**http_find_route**	4.0	“Routes” on page 74
104	**http_add_route**	4.0	“Routes” on page 74
108	**http_process_route**	4.0	“Entries for the Server Only” on page 74
10C	unused (always 0)	-	“The Rules” on page 65
110	**http_xlate**	4.0	“Translation” on page 70
114	**xlate_cp037**	4.0	“Translation” on page 70
118	**xlate_1047**	4.0	“Translation” on page 70
11C	**xlate_legacy**	4.0	“Translation” on page 70
120	**http_get_ufs**	4.0	“Files” on page 71
124	**http_cgictx_get**	4.0	“The Context Block” on page 74
128	**http_get_userid**	4.0	“Authentication” on page 72
12C	**http_get_acee**	4.0	“Authentication” on page 72
130	**http_get_token**	4.0	“Authentication” on page 72
134	**http_check_auth**	4.0	“Authentication” on page 72
138	**http_logout**	4.0	“Authentication” on page 72
13C	**http_get_password**	4.0	“Authentication” on page 72

140	**http_realm**	4.0	“Authentication” on page 72
144	**http_new_session**	4.2	“Sessions” on page 73

The following sections describe the entries by group. A group begins with the declarations of its functions as the macros take them. The sections list return values and constraints for each function.

Output

```
int http_resp(HTTPC *httpc, int code);
int http_printf(HTTPC *httpc, const char *fmt, ...);
int http_printv(HTTPC *httpc, const char *fmt, va_list args);
int http_send(HTTPC *httpc, const UCHAR *buf, int len);
UCHAR *http_server_name(HTTPD *httpd);
```

A response has four parts, in this order:

1. **http_resp(httpc, code)** for the status line and the standard headers.
2. Zero or more header lines, each with **http_printf**.
3. One call with the text **"\r\n"** and nothing else. It ends the headers.
4. The body, with **http_printf** or **http_send**.

http_resp — Sets **httpc->resp** to the code and sends the status line. The line has the version of the request, HTTP/1.1 or HTTP/1.0. The function also sends the headers **Date**, **Server: HTTPD Server**, **X-Powered-By: MVSLOVERS** and **Connection**, which is **keep-alive** or **close**. It does not end the headers. It returns 0, or a value other than 0 if the send fails. The codes that HTTPD knows are 200, 201, 202, 204, 206, 301, 302, 303, 304, 400, 401, 403, 404, 405, 409, 410, 412, 413, 414, 429, 500, 501, 502, 503, 505 and 507. For any other code it writes the console message **HTTPD909E** and sends 500.

http_printf, **http_printv** — Format the text in a buffer of 5120 bytes, convert it to ASCII and send it. Text that is longer than 5119 bytes is cut. They return 0 on success and −1 on failure. They do not return a count of bytes. They return −1 at once if the state of the client is **CSTATE_DONE** or later. The two functions are the same: **http_printv** takes a **va_list**.

http_send, raw bytes — Sends the bytes of the buffer as they are. The caller provides ASCII or binary data. The function does not look at the data. If the response is not chunked, it returns the number of bytes sent, which can be less than **len**. It returns 0 if the socket is not ready and −1 if the connection is dead. In the second case it sets the state to **CSTATE_DONE**. If the response is chunked, it sends the buffer as one chunk and returns **len**, or −1 after a failure. A failure turns chunking and keep-alive off. A call with **len** 0 or less returns 0.

http_server_name — Returns the string **HTTPD Server**. The argument is not used.

The end of the headers. **http_printf** watches for the call that ends the headers. This call has the text **"\r\n"** alone, exactly two characters. When it sees the call, it sets **httpc->rdw** to 1. If the request is HTTP/1.1, the status is not 1xx, 204 or 304, and no **Content-Length** header has been sent, it sends **Transfer-Encoding: chunked** first and turns chunking on. From then on every send is a chunk, and the server sends the final empty chunk when the request is done.

A HEAD request. While a module runs for a HEAD, **httpc->state** is **CSTATE_HEAD**. A response to a HEAD ends at the empty line after the headers. Once the call that ends the headers has been made, **http_printf** and **http_printv** send nothing and return 0, so a body that the module writes with them does not reach the client. The server does not send **Transfer-Encoding: chunked** for a HEAD, whatever the module sent. The status line and the header lines before the empty line go out as for any method, a **Content-Length** among them. **http_send** is not changed: bytes that a module sends with it

reach the client. A module that answers a HEAD therefore sends the status and the headers, and no body. Releases before HTTPD 4.5.0 chunk the response of a module to a HEAD and send its body.

Two consequences follow.

- Send the end of the headers as a call of its own. Do not write **`"...\r\n\r\n"`**. HTTPD does not recognize the end of the headers in such a call. The response is then not chunked, and the client does not know where the body ends. A later call with exactly **`"\r\n"`** in the body would be taken for the end of the headers and would add a header line to the body.
- A header line that goes out with **`http_send`** is not examined. HTTPD does not see a **`Content-Length`** in it.

A line that begins with **`Content-Length:`** in a call of **`http_printf`** sets **`content_length_set`**. Chunking is for HTTP/1.1 only. For an HTTP/1.0 request send a **`Content-Length`**.

If a send makes no progress, the server retries it, with a short pause between the tries. It gives up when the server is quiescing or shutting down, or when the retries have used their budget. It then writes a message to the console (throttled) and sets the state to **`CSTATE_DONE`**.

Standard Error Responses

```
int http_resp_not_found(HTTPC *httpc, const UCHAR *path);
int http_resp_internal_error(HTTPC *httpc);
int http_resp_not_implemented(HTTPC *httpc);
```

Each of the three functions sends a complete response, with the end of the headers, and returns 0, or a value other than 0 if the send fails. They do not set the state to **`CSTATE_DONE`**. Return from the module afterwards.

Table 59. The standard error responses

Function	Status	Body
`http_resp_not_found`	404	**`404 document "path" not found`**. The headers are **`Cache-Control: no-store`** and **`Content-Type: text/plain`**. If **`path`** is NULL, the body has no text.
`http_resp_internal_error`	500	**`500 Internal Server Error`**, as **`text/plain`**.
`http_resp_not_implemented`	501	**`501 Not Implemented`**, as **`text/plain`**, with **`Cache-Control: no-store`**.

Request Variables

```
UCHAR    *http_get_env(HTTPC *httpc, const UCHAR *name);
int       http_set_env(HTTPC *httpc, const UCHAR *name, const UCHAR *value);
unsigned  http_find_env(HTTPC *httpc, const UCHAR *name);
int       http_del_env(HTTPC *httpc, const UCHAR *name);
HTTPV    *http_new_env(const UCHAR *name, const UCHAR *value);
int       http_set_http_env(HTTPC *httpc, const UCHAR *name, const UCHAR *value);
int       http_set_query_env(HTTPC *httpc, const UCHAR *name, const UCHAR *value);
UCHAR    *http_decode(UCHAR *str);
```

The names are compared without regard to case.

`http_get_env` Returns the value of the variable, or NULL if there is none. The pointer is valid until the variable is changed or deleted, or until the request ends.

`http_set_env` Creates the variable, or replaces the value of an existing one. Returns 0, or –1 on failure. A failure also writes the message **`HTTPD904E`** or **`HTTPD905E`**.

`http_find_env` Returns the index of the variable plus one, or 0 if there is none.

http_del_env Deletes the variable and always returns 0.

http_new_env Builds a variable block and returns it, or NULL. The sum of name and value must not exceed 8192 bytes, else the function fails with **errno** E2BIG. It also returns NULL when it has no storage. The block comes from subpool 0. The function does not add it to the client. The other functions do that.

http_set_http_env Stores the value as **HTTP_name**. The server uses it for the headers. A header called **Cookie** is split into the variables **HTTP_Cookie-name**. The name must fit in 256 bytes.

http_set_query_env Stores the value as **QUERY_name**. The name must fit in 256 bytes.

http_decode Decodes a URL-encoded string in place and returns it. A sequence **%xx** becomes the character of the server code page (see "Translation"). A **+** becomes a blank. A **%** that is not followed by two hexadecimal digits stays as it is, and so does the escape **%00**. The server decodes the path of a request with a function of its own, in which a **+** stays a plus sign ("URL Decoding and Escaping" on page 47).

A module that creates its own variables chooses names that do not collide with the names of "The Request Variables" on page 63. The variables live until the request is reset.

Translation

```
UCHAR *http_etoa(UCHAR *buf, int len);
UCHAR *http_atoe(UCHAR *buf, int len);
unsigned char *http_xlate(unsigned char *buf, int len, const unsigned char *tbl);

typedef struct { const unsigned char *atoe; const unsigned char *etoa; } HTTPCP;
/* in the vector: HTTPCP *xlate_cp037, *xlate_1047, *xlate_legacy */
```

HTTPD works in EBCDIC and talks ASCII. The conversion tables depend on the code page of the server (see Chapter 1, "The HTTPPRM Keywords" on page 1 for the keyword).

http_etoa, **http_atoe** Translate the buffer in place with the table of the server code page and return the buffer. A **len** of 0 means the length of the string. If the server has no context, the code page is CP037. Never pass a string literal: the literal may sit in storage that the module cannot write, and the call abends with S0C4. Copy the text to an array first.

http_xlate Translates in place with a table of 256 bytes that the caller gives. A table of NULL means the table EBCDIC to ASCII of the server. Returns the buffer.

xlate_cp037, **xlate_1047**, **xlate_legacy** Entries of the vector that point to the pairs of tables for the code pages CP037, IBM-1047 and LEGACY. Use the member **etoa** or **atoe** as the table of **http_xlate**.

The names of the code pages that the server accepts are CP037, IBM1047 and LEGACY. A UFS file is stored in IBM-1047. **http_send_file** uses that table for it.

Time, MIME Types and Addresses

```
UCHAR        *http_date_rfc1123(time64_t t, UCHAR *buf, size_t size);
double       *http_secs(double *psecs);
const HTTPM  *http_mime(const UCHAR *path);
char         *http_ntoa(struct in_addr addr);
```

http_date_rfc1123 Writes the time as a date of the form **Thu, 01 Jan 2026 12:00:00 GMT** (the format of RFC 1123) into the buffer and returns the buffer.

http_secs Writes the time since the epoch, with microseconds, into the double that the argument points to. Read the double that you gave and ignore the return value.

The declaration says that the function returns a pointer, but the function returns a double by value. Only the lower 32 bits of the seconds are used.

http_mime Returns the MIME type of the path. The type comes from the extension of the last part of the path. The search does not depend on case. For a name of the form **dd:name(member)** the part in the parentheses is dropped. An unknown extension gives the type **text/plain** as text. The function never returns NULL. The extensions **html** and **htm** give **text/html**. The extensions **shtm** and **shtml** give **text/x-server-parsed-html**.

http_ntoa Writes the address as **a.b.c.d** into a buffer of the C runtime of the task and returns the buffer. The next call in the same task overwrites it.

Files

```
FILE *http_open(HTTPC *httpc, const UCHAR *path, const HTTPM *mime);
int   http_send_text(HTTPC *httpc);
int   http_send_binary(HTTPC *httpc);
int   http_send_file(HTTPC *httpc, int binary);
UFS  *http_get_ufs(HTTPC *httpc);
```

http_open Opens a file of the document root. **It always returns NULL.** Look at **httpc->ufp** to see whether the open worked: it is not NULL if it did. The function refuses a path that contains **..** and cuts a path to 255 bytes. If **mime** is NULL, it calls **http_mime** itself. It opens the file as binary if the type is binary and as text if not. It sets **httpc->ssi** to 1 for the types **text/html** and **text/x-server-parsed-html** and to 0 for the others. The file name is the document root with the path appended.

http_send_text, **http_send_binary**, **http_send_file** Send the open file in steps. One call reads a piece of the file into **httpc->buf**, sends what it can and returns 0. Call it again and again until **httpc->state** is **CSTATE_DONE**. The state becomes **CSTATE_DONE** at the end of the file, and also when the send fails or the UFS read fails. The text form translates the data from IBM-1047 to ASCII, the binary form sends it as it is. **http_send_text** is **http_send_file(httpc, 0)** and **http_send_binary** is **http_send_file(httpc, 1)**. The function processes server side includes if **httpc->ssi** is 1: **<!--#echo var="X" -->**, **<!--#include file="x" -->**, the form with **virtual** and **<!--#printenv -->**. Send the headers before the first call. Do not close the file. **http_done** closes it. If a UFS read fails, the response ends: chunking and keep-alive go off and the server writes a console message.

http_get_ufs Returns the UFS session of the client. It creates the session the first time. It returns NULL if the server has no UFS.

The server uses these functions for its own static GET. Its steps are a model for a module that serves files: get the type with **http_mime**, open the file with **http_open**, send the status, **Content-Type** and the end of the headers, then send the file.

Dynamic Arrays

```
void     *array_new(unsigned count);
int       array_add(void *varray, void *item);
int       array_addf(void *varray, const char *fmt, ...);
unsigned  array_count(void *varray);
unsigned  array_size(void *varray);
void     *array_get(void *varray, unsigned index);
void     *array_del(void *varray, unsigned index);
int       array_free(void *varray);
```

The array entries give a module the dynamic arrays of the C library, so that the module does not need to be linked with them. An array is a pointer to the items. The first argument of the functions is the address of that pointer, not the pointer itself. The indexes run from 1 to the count. **array_free** frees

the array and does not free the items. HTTPD itself tests the result of **array_add** against 0 for success. The C library documents the rest.

Comparison

```
int http_cmp(const UCHAR *s1, const UCHAR *s2);
int http_cmpn(const UCHAR *s1, const UCHAR *s2, int n);
```

http_cmp Compares two strings without regard to case. It returns the difference of the first characters that differ, and 0 if the strings are the same.

http_cmpn The same, for at most **n** characters.

Debug Output

```
int http_dbgw(const char *buf, int len);
int http_dbgs(const char *str);
int http_dbgf(const char *fmt, ...);
int http_dump(void *addr, int len, const char *title, ...);
int http_enter(const char *fmt, ...);
int http_exit(const char *fmt, ...);
```

The six functions write to the debug file of the server, the DD **HTTPDBG**. The server opens the file only if the keyword **DEBUG** is on (see Chapter 1, “The HTTPPRM Keywords” on page 1). If it is off, the functions write nothing. If the file cannot be opened, the server writes **HTTPD020E** and the output goes to **stderr**. If a write fails, the server writes **HTTPD039W** once and switches the debug output off until the next start. From then on the functions write nothing.

http_dbgw Writes **len** bytes.

http_dbgs Writes a string.

http_dbgf Writes formatted text.

http_dump Writes a dump of **len** bytes in lines of 16 bytes with a title. The title is formatted into a buffer of 80 bytes. Keep it short.

http_enter, **http_exit** Write the text with the prefix “Enter “ or “Exit “ to mark the entry to a function and the exit from it.

Authentication

```
UCHAR        *http_get_userid(HTTPC *httpc, UCHAR *out, unsigned outlen);
struct acee  *http_get_acee(HTTPC *httpc);
int           http_get_token(HTTPC *httpc, UCHAR *out, unsigned outlen);
int           http_check_auth(HTTPC *httpc, const char *class,
                              const char *resource, int attr);
int           http_logout(HTTPC *httpc);
UCHAR        *http_get_password(HTTPC *httpc, UCHAR *out, unsigned outlen);
const char   *http_realm(HTTPD *httpd);
```

The server has already identified the client when the module runs. It reads the credential from these sources in this order: the cookie **Sec-Token**, the cookie **LtpaToken2**, a Bearer token and then Basic authentication. The mode of the route (**AUTH=**, see Chapter 2, “Routes and Authentication Policy” on page 9) changes only the response to a client that has no credential. A route with **AUTH=NONE** passes the request without a credential. All functions in this group accept a client without a credential.

http_get_userid Copies the userid of the client into **out** and returns **out**. A userid has at most 8 characters, so a buffer of 9 bytes holds it; in a smaller buffer the userid is cut. The function returns NULL, with an empty string in **out**, if the client has no credential. It returns NULL if **out** is NULL or **outlen** is 0.

http_get_acee Returns the ACEE of the client, or NULL. A module that calls the RACF routines itself with this ACEE sees the raw SAF code. An unprotected resource then answers 4, not 0. Use **http_check_auth** unless you need the difference.

http_get_token Copies the token of the session of the client, 32 bytes, into **out** and returns 32. It returns 0 if there is no credential or **outlen** is less than 32. The token is binary. The caller encodes it with base64 to send it.

http_check_auth Asks RACF whether the client may use the resource under the class, with the attribute **attr**: **RACF_ATTR_READ**, **RACF_ATTR_UPDATE**, **RACF_ATTR_CONTROL** or **RACF_ATTR_ALTER**. 0 means READ. It returns 0 if the access is permitted, 8 or more (the return code of SAF) if it is refused, and –1 if the client is not authenticated. The case "no profile" is also returned as 0. Test for 0. Never test for a value of 4 or less: that would let an unauthenticated client (-1) through.

http_logout Removes the session of the client and clears the credential of the request. It returns 0, or –1 if there was no session.

http_get_password Decrypts the password of the client and writes it into **out**. The password is in upper case. It returns **out**, or NULL if there is no credential or if the credential has no password (a session that is known only by token). In a buffer that is too small the password is cut. Overwrite the buffer when you are done with it.

http_realm Returns the realm of the server: the value of the keyword **REALM**, or the SMF identifier of the system if there is none (see Chapter 1, "The HTTPPRM Keywords" on page 1). It is never NULL. The string stays valid for the life of the address space. A module that sends its own **WWW-Authenticate** challenge uses it as the realm, so that the challenge of the module and that of the server agree.

Sessions

```
int http_new_session(HTTPC *httpc);
```

http_new_session creates a new login session for the client of the request. The session has the same userid, password and address as the credential of the request. It has a token of its own and an ACEE of its own. The function is for a login endpoint of a module. Each call costs a RACF logon and a place in the credential store. The entry exists from HTTPD 4.2.

The function returns 0 on success. Then **http_get_token** returns the token of the new session. If the module cannot send the token to the client, it calls **http_logout** so that the session does not stay.

From HTTPD 4.5 the function reports why it failed:

Table 60. The results of http_new_session

Constant	Value	Meaning
HTTP_SESSION_NONE	–1	The request is not authenticated, or it carries no password to log on again with. A credential that came from a token alone has no password.
HTTP_SESSION_REFUSED	–2	RACF refused the logon (return code 4, 8, 12 or 28 of RACINIT). The server marks the credential as stale. It does not answer by password or token from it any more and frees it 600 seconds later. The current request still runs on it. The module answers 401.

HTTP_SESSION_BUSY	−3	The client has 16 sessions, all in use, or the credential store is full. A session is replaced only if it has been idle for 60 seconds or more.
HTTP_SESSION_NOMEM	−4	The server is out of storage, or RACF could not be asked: RACF is not active, an installation exit or a terminal or time restriction stopped the logon. The server then writes the message **HTTPD009W** a refusal of the password is not written.

Before HTTPD 4.5 every failure returns −1. Test the result for **!= 0** or **< 0**, not for **== -1**. Treat any other negative value as an unknown failure.

The Context Block

```
void *http_cgictx_get(HTTPD *httpd, const char *eye, unsigned size);
```

The function returns a block of storage that lives as long as the server and is the same for every request. See “Context Pointers” on page 79 for the details.

Routes

```
HTTPROUTE *http_find_route(HTTPD *httpd, const char *path);
HTTPROUTE *http_add_route(HTTPD *httpd, const char *pgm, const char *path,
                          int login);
```

http_find_route Searches the route table in the order of the Parmlib and returns the first route that matches the path, or NULL. A route with a wildcard (***** or **?**) matches by pattern, any other route by equality without case.

http_add_route Adds a route. It copies **pgm** and **path**. A **pgm** of NULL makes a route for static files (as the keyword **LOC=**). The argument **login** is not used. The new route is public, as with **AUTH=NONE**. It returns NULL on failure.

Entries for the Server Only

The following entries are the steps of the request cycle and the control of the worker pool. The server calls them. A module does not call them. The table says what each one does.

Table 61. The entries that belong to the server

Entry	What it does
http_in, **http_parse**	Read the request and parse it into variables.
http_get	Serves a static file for GET. See Chapter 5, “HTTP Behaviour and Limits” on page 43.
http_head	Serves a static file for HEAD. It calls **http_get**, which sends the status and the headers of a GET and stops at the empty line when the state is **CSTATE_HEAD**. The state then becomes **CSTATE_DONE**. See “Methods” on page 43. Before HTTPD 4.5.0 it was a stub that answered 501.
http_put, **http_post**	Stubs. Each answers 501 and sets the state to **CSTATE_DONE**. POST is routed to **http_put**.
http_done	Closes **fp** and **ufp**, sends the final chunk of a chunked response, sets the end time and the state **CSTATE_REPORT**.

http_report	Adds to the counters and writes the SMF record (see Chapter 8, “SMF Records” on page 93). The state becomes **CSTATE_RESET**.
http_reset	Frees the variables and clears the request fields. With keep-alive the state becomes **CSTATE_IN**, else **CSTATE_CLOSE**.
http_close	Removes the client from the table, closes the files, the UFS session and the socket, and frees HTTPC.
http_is_busy, **http_set_busy**, **http_reset_busy**	Manage the table of the clients that a worker is serving.
http_process_clients, **http_process_client**	The loop of the worker.
http_console	Handles an operator command.
http_link, **http_process_route**	Run a module (see “How a Module Is Run” on page 59). **http_link** returns the result codes of “Result Codes of a Link”.

The Direct Accessors

HTTPD is opaque. Two macros read a field of it. They are not entries of the vector. They read the storage at a fixed offset, and the server checks the offsets when it is built.

```
#define http_get_httpx(h)  (*(HTTPX **)((char *)(h) + 8))
#define http_get_flag(h)   (*(volatile unsigned char *)((char *)(h) + 0x2C))

#define HTTPD_FLAG_QUIESCE  0x40
#define HTTPD_FLAG_SHUTDOWN 0x80
```

http_get_httpx Returns the function vector.

http_get_flag Returns the flag byte of the server. The bit **HTTPD_FLAG_QUIESCE** means that the server accepts no new requests. The bit **HTTPD_FLAG_SHUTDOWN** means that the server is shutting down.

Both macros exist from HTTPD 4.0. See “Long-Running Handlers and Shutdown” on page 77 for the use of the flags.

Result Codes of a Link

http_link returns the outcome of the LINK as one integer. The server uses it to build the answer that is described in “Abends” on page 77. The values:

Table 62. The result of http_link

Value	Number	Meaning
0 or more		The module ran and returned this code.
HTTP_LINK_ENOLOAD	–0x01000001	The LINK failed and the cause is not S806.
HTTP_LINK_ENOPGM	–0x01000002	The route has no program name.
HTTP_LINK_EESTAE	–0x01000003	The ESTAE could not be set. The module did not run.

HTTP_LINK_ENOMOD	–0x01000004	The module is not in the STEPLIB or the link list (S806).
HTTP_LINK_EPGMRC(rc)	–0x02000000 + rc	The module ran and returned a negative code **rc**. The macros **HTTP_LINK_IS_PGMRC(v)** and **HTTP_LINK_PGMRC(v)** test the value and take the code out. The startup code clamps a negative code to 0, so only a module with its own start code returns one.
any other negative value		The module abended. The value is the negated completion code in the form X'00sssuuu' (system code sss, user code uuu).

Every negative value means that the module did not finish its work. For the values that say that the module never ran, there is no dump to look for.

Storage Rules

A module runs in the task of a worker of HTTPD. Its storage follows from that.

Heap
: The startup code switches the heap of the module to the request subpool, subpool 5, for the time of **main**. Everything that the module gets with **malloc** belongs to the request. A module frees what it allocates, on every path. If the module abends, the server frees the whole subpool with one FREEMAIN. When the worker task ends, MVS frees what is left.

http_cgi_subpool
: **UCHAR http_cgi_subpool(HTTPC *httpc)** returns the number of the request subpool, 5, also available as the constant **HTTP_CGI_SUBPOOL**. It returns 0 for a client that the server does not own, and the entry holds 0 in a server that is too old to have the subpool. In that case the startup code sets no subpool and the heap is as it was. A module does not need to call it.

Storage that must live longer
: Get it with **__getmsp(size, 0)** (declared in **mvs/storage.h**), or ask for a block with **http_cgictx_get** (see "Context Pointers" on page 79). Storage in any other subpool does not survive the request.

No writable static data
: A module must not write to a file-scope variable or to a static variable of a function. A module that is fetched from an APF-authorized library or from the link list is in storage of key 0, while its code runs in key 8. The first write to the module storage then abends with S0C4. The library decides this, not the attribute AC(1) and not RENT. A switch to key 0 with **__super()** does not help, because a module can be in key 0 without the right to switch.

Where to keep state
: Keep the state of a request in local variables, on the heap, or in the memory that **__wsaget()** returns. Keep state of the server in the block of **http_cgictx_get**. Variables that the module only reads are fine in the static data: tables, and constant strings.

Strings that are translated
: Declare **UCHAR text[] = "..."** and not **UCHAR *text = "..."**, if you translate the text in place. The array is a copy on the stack. The pointer points to the constant itself.

Server allocations
: The functions **http_new_env**, **http_open**, **http_get_ufs** and **http_add_route** get their storage from subpool 0 inside the server. A reclaim of subpool 5 does not touch it.

Stack
: See "What the Startup Code Does" on page 60 for **__stklen**.

Long-Running Handlers and Shutdown

A request that takes long holds a worker. When the server stops, it sets the flags **`HTTPD_FLAG_QUIESCE`** and **`HTTPD_FLAG_SHUTDOWN`** in the flag byte, and it waits up to 10 seconds, in steps of 1 second, for the socket thread to end (it writes the messages **`HTTPD040I`** and **`HTTPD041I`** while it waits). The sends of the server honor the flags: they stop and set the state to **`CSTATE_DONE`**.

A module that loops, waits or sends in pieces looks at the flags in each round and returns when a flag is set:

```
while (more_to_send) {
    if (http_get_flag(httpd) & (HTTPD_FLAG_QUIESCE | HTTPD_FLAG_SHUTDOWN))
        break;
    if (httpc->state == CSTATE_DONE)
        break;
    /* ... send the next piece ... */
}
```

Abends

Two layers of recovery protect a worker.

The module. The LINK of the module is made through **`__linkdsx`**, which sets an ESTAE. If the module abends, the ESTAE retries the call of the server. The worker lives and serves its next request. The server then does this in order:

1. It frees the subpool 5. If the FREEMAIN fails, it writes **`HTTPD906W STORAGE RECLAIM FOR pgm FAILED, RC=n`**.
2. It builds the text of the answer from the result of “Result Codes of a Link” on page 75.
3. If no response was started (**`httpc->resp`** is 0), it sends the status 503 with **`Content-Type: text/plain`** and the end of the headers. If the module had begun the response, the text goes after what the module sent.
4. It writes the body **`External program pgm text`**, and the console message **`HTTPD908E EXTERNAL PROGRAM pgm text`**.
5. It sets the state to **`CSTATE_DONE`**.

The text is one of these:

Table 63. The texts of a failed module

Text	Cause
was not found	The module is in no library of the STEPLIB or link list (S806).
could not be loaded	The LINK failed for another reason, for example a fetch error or too little storage.
was dispatched without a program name	The route has no program.
could not be started, ESTAE create failed	The ESTAE could not be set.
returned rc n	The module returned the negative code n.
failed with Snnn ABEND	A system abend with the code Snnn.
failed with Unnnn ABEND	A user abend with the code Unnnn, in decimal.

An abend of a module does not end the worker. A reclaim of subpool 5 releases what the module allocated with **`malloc`**. It does not release what the module holds outside subpool 5: files that the

module opened itself, a UFS session that it created, or storage from other subpools. A module that needs to clean these up after an abend must do so with its own recovery.

The worker. Each worker runs under an ESTAE of its own, with a retry. If the abend happens in the code of the server and not in a module, the server writes **HTTPD062E ABEND code IN WORKER(id) CLIENT(addr) SOCKET(n)**, clears the busy mark of the client, closes the client and the worker serves on. If the worker cannot recover, it writes the message **HTTPD063E**, which says that there is no recovery in the worker and that the client is dropped. The messages are in Appendix A, “Messages” on page 99.

The Started Task Procedure

A module runs in the address space of the server. It has no allocations of its own. When it opens a ddname, the system looks for it in the procedure of the server. A module is therefore a part of the contract of the procedure: if the procedure lacks a DD that a module needs, the module fails.

The procedure that comes with HTTPD has these DDs. It also has the DDs **SYSUDUMP**, **SNAP** and **HTTPSTAT**, all with **SYSOUT=***. The DDs **HASPCKPT** and **HASPACE1** are in it as comment lines.

Table 64. The DDs of the HTTPD procedure

DD	Use
STEPLIB	The load library of the server and of the modules.
HTTPDERR	**stderr** of a module.
HTTPDOUT	**stdout** of a module.
HTTPDIN	**stdin** of a module. It is DUMMY.
HTTPDBG	The debug file (see “Debug Output” on page 72).
HTTPPRM	The Parmlib member (**FREE=CLOSE**).
HASPCKPT, **HASPACE1**	Optional, comment lines in the sample. The JES2 checkpoint data set and the spool volume. HTTPD never opens them. See below.

The two JES2 DDs are for the modules that read JES2. A module that calls the C library routine **jesopen()** opens the checkpoint and the spool. The jobs interface of mvsMF is such a module. A DD in the procedure wins. When the procedure has no DD, the routine of LIBC/370 2.24.0 and later allocates **SYS1.HASPCKPT** and **SYS1.HASPACE** itself. The DDs are therefore optional for a module built with such a level, which includes mvsMF 1.4.0 and later. They stay mandatory in two cases:

- The module is built with an older level of the C library. The jobs interface of mvsMF 1.3.0 and earlier is such a module. Without the DDs, every job list answers 500 and every lookup of a job by name and id answers 404, a spool read included, and two lines go to the console.
- The JES2 of the system builds its data set names with a prefix other than **SYS1**. The allocation looks only for the **SYS1** names.

A job that is submitted does not need the DDs, because the submit goes through the internal reader, which the module allocates itself. The messages and the answers are in “The JES2 data sets” on page 31.

To use the DDs, uncomment them in the sample member:

```
//HASPCKPT DD DSN=SYS1.HASPCKPT,DISP=SHR,UNIT=3350,VOL=SER=MVS000
//HASPACE1 DD DSN=SYS1.HASPACE,DISP=SHR,UNIT=3350,VOL=SER=SPOOL1
```

The unit and the volume serial must be those of the JES2 procedure of the site. The values above are those of the sample procedure. Before you remove a DD that is active in a procedure, find out which modules of other products use it.

Resident Modules

With the keyword **RESIDENT=YES**, which is the default, the server loads each module of its routes once, when it starts. It does a BLDL for every different name in a **MOD=** keyword and then a LOAD in the main task. A LINK of the worker then finds the module in storage and does not fetch it from disk again. The keyword is described in Chapter 1, “The HTTPPRM Keywords” on page 1.

The server holds a module only if it is reentrant or reusable (RENT or REUS). For any other case it writes a message and leaves the module to be fetched at each request:

Table 65. The messages of the resident load

Message	Meaning
HTTPD440W	The module was not found when the server started. It is not held.
HTTPD441W	The module is not reusable. It is fetched for each request.
HTTPD442W	The load failed with the abend code shown. The module is fetched for each request.
HTTPD443W	The value of **RESIDENT** is not YES or NO.

The server holds every module that it could load. Its count of the held modules, which **D C** and the server block show, stops at 255. The server deletes the modules when it ends. Two consequences follow for the module author:

- A new level of a resident module does not take effect until the server is restarted. With **RESIDENT=NO** the next request fetches the new level.
- A module that is not RENT or REUS works, but it is fetched anew for every request.

The display module HTTPDSRV (see Chapter 7, “The Display Modules” on page 81) shows the setting and the number of held modules.

Context Pointers

Storage that a module allocates in a request does not outlive it. A module that needs data across requests, such as a cache or a table of open resources, asks for a block of the server:

```
void *http_cgictx_get(HTTPD *httpd, const char *eye, unsigned size);
```

The server keeps a table of blocks. The block is identified by an eye catcher of 8 bytes. The first call with an eye catcher creates the block. It gets the requested size from storage that lives as long as the address space, clears it and writes the eye catcher into its first 8 bytes. Every later call with the same eye catcher returns the same block. The size counts only when the block is created, so a module always gives the same size. The server never frees the block.

The function returns NULL if the table is full or if there is no storage. The table is locked for the duration of the call, but the block itself is not. A module that shares a block between requests, which can run at the same time in different workers, serializes its own access.

The size of the table is set by the Parmlib keyword **CGI_CONTEXT_POINTERS**. The default is 255 and the maximum is 255. The server cuts a larger value to 255 and ignores a negative value. The table has

one more place than the value. The keyword is in Chapter 1, “The HTTPPRM Keywords” on page 1. HTTPDSRV shows the value (see Chapter 7, “The Display Modules” on page 81).

The function exists from HTTPD 4.0.

Chapter 7. The Display Modules

HTTPD comes with four small server modules for diagnosis. Three of them show live data of the running server: HTTPDSRV shows the control blocks of the server, HTTPDM shows any piece of storage as a dump, and HTTPDMTT shows the master trace table of the system. The fourth, ABEND0C1, abends on purpose to test the recovery of the server. All four are load modules that are built with the server. They are server modules in the sense of Chapter 6, "The Server Module Interface" on page 59: HTTPD runs them for the route that you give them in the Parmlib.

This chapter lists, for each module, the URL, the query parameters, what it shows, the Parmlib line, the authentication it belongs behind, and its limits. The *HTTPD User's Guide* shows how an operator uses the modules, in "Operating HTTPD".

Setting Up the Modules

HTTPD registers no route for these modules by itself. The sample Parmlib member has the four lines, commented out. They are the recommended routes:

```
MOD=HTTPDSRV /.dsrv     AUTH=FORM
MOD=HTTPDM   /.dm       AUTH=BASIC
MOD=HTTPDMTT /.dmtt     AUTH=BASIC
MOD=ABEND0C1 /.abend    AUTH=BASIC
```

The keywords of a route line are in Chapter 1, "The HTTPPRM Keywords" on page 1 and Chapter 2, "Routes and Authentication Policy" on page 9. Observe these rules.

- A display module reads storage of the server address space and, for HTTPDMTT, the console log. A route without an **`AUTH=`** is public in HTTPD, so never register one of these modules without it. The sample member says the same: none of the lines may be uncommented without an **`AUTH=`** of its own.
- The modules do not look at the user. Every user who passes the route gate sees everything. To restrict the route to some users, add a **`RES=`** with a RACF profile, for example **`RES=FACILITY:HTTPD.ADMIN`**. The keyword is described in Chapter 2, "Routes and Authentication Policy" on page 9.
- HTTPDSRV writes links to the path **`/.dm`**. The route for HTTPDM must therefore be exactly **`/.dm`** if the links in the pages of HTTPDSRV are to work. HTTPDSRV is a browser tool, so **`AUTH=FORM`** (a login page) suits it. HTTPDM and HTTPDMTT also work with **`curl`**, so **`AUTH=BASIC`** suits them.
- Use the modules on a test system or for a short time. They are diagnostic tools, not production endpoints.

HTTPDSRV, HTTPDM and HTTPDMTT send the response header **`Cache-Control: no-store`**. None of them sends **`Access-Control-Allow-Origin`**, so a page of another origin cannot read an answer in a browser that holds a session with the server. The **`text/plain`** answers of HTTPDM and HTTPDMTT (**`d`** set to a number other than 0) and the pages of HTTPDSRV also send **`X-Content-Type-Options: nosniff`**, so a browser does not take the text for HTML.

A display module that is started in another way than by HTTPD (for example from JCL) does nothing. It writes the message **`HTTPD920E`** to the console and a line "This program ... must be called by the HTTPD web server" to its standard output, and it ends with the return code 12.

Reading Storage

HTTPDSRV and HTTPDM read storage at addresses that come from the URL. They can read what the server task may read. They do not check an address before they read it. Four things follow from this.

- The memory dump of both modules runs under an abend handler (ESTAE). An address that the task cannot read ends the dump with a line that starts with **`ABEND S0C4`**. The server keeps working.
- The tables that HTTPDSRV prints after a dump are not under the handler. If an address in the URL of the targets **`FILE`**, **`TASK`**, **`MOD`** or **`WORKERS`** cannot be read, the module abends. HTTPD answers with the status 503 and the text "External program HTTPDSRV failed with S0C4 ABEND", and writes the message **`HTTPD908E`** to the console. The server keeps working.
- The modules show storage as it is at the moment of the request. The values can change while the page is built.
- An address is only good while the server runs. After a restart all addresses are new.

HTTPDSRV, the Server Display

HTTPDSRV shows the control blocks of the server as HTML tables. The path in the recommended route is **`/.dsrv`**. The module answers every request method in the same way. It ignores the request body.

Parameters

Table 66. The query parameters of HTTPDSRV

Parameter	Default	Meaning
`target`, **`t`**	**`HTTPD`**	What to display. The value is matched against the names in "Targets". If both names are given, **`target`** counts.
`m`, **`mem`**, **`memory`**	none	The address of the block to display, as hexadecimal digits. The first of the three names that is present counts. The targets that need an address stop with a message when it is missing.

The comparison of the target is not case sensitive, and the value only has to be the start of a name. HTTPDSRV tests the names in a fixed order and takes the first one that fits. The order is **`FILE`**, **`FS`**, **`MGR`**, **`MOD`**, **`HTTPD`**, **`TASK`**, **`TRACE`**, **`WORKERS`**. So **`target=M`** is **`MGR`**, **`target=MO`** is **`MOD`**, **`target=F`** is **`FILE`**, and **`target=T`** is **`TASK`**. The target **`TRACE`** needs at least the two letters **`TR`**. A value that fits none of the names gives the help page ("The Help Page" on page 89). The help page lists only the targets of this table.

Targets

Table 67. The targets of HTTPDSRV

Target	Needs an address	Shows
`HTTPD`	no	The HTTPD control block of the server. This is the default.
`MOD`	yes	The route array and every route in it.
`MGR`	no (ignored)	The manager of the worker pool.
`WORKERS`	yes	The array of workers, with the request that each one is processing.
`TASK`	yes	A thread (task) handle.
`TRACE`	no	The request trace, as plain text ("Target TRACE" on page 86).

FILE	yes	A C file handle (the debug file is an example).
FS	no	The UFS system block of the server.

Every page except the help page has a heading with the address of the block, a dump of the block with links, and a table of the fields. The table has four columns: the offset of the field in the block (in hexadecimal, with a plus sign), the name of the field, a description, and the contents.

The pages use the status 200 and the type **text/html**. A problem such as a missing address is reported in the text of the page, still with the status
200.

Target HTTPD

The page shows the heading "HTTPD Handle" with the address of the block. The table that follows has one row for every field of the control block, in order of offset. The fields are these.

Table 68. The fields shown for target HTTPD

Field	Contents
eye	The eye catcher of the block.
httpx	The address of the function vector ("The Function Vector" on page 65). A link opens the vector in HTTPDM.
httpc	The address of the client array, with the number of clients.
addr, **port**, **listen**	The address and the port that the server listens on, and the number of the listen socket.
cfg_realm	The Basic realm (the server name).
dbg	The debug file handle. When the debug file is open, a link opens it with the target **FILE**.
tzoffset	The time zone offset in seconds.
busy	The array of busy clients, with its count.
flag	The processing flag in hexadecimal, followed by the names of the bits that are set: **INIT**, **LISTENER**, **READY**, **CFGERR**, **QUIESCE**, **SHUTDOWN**.
client	The client options in hexadecimal, followed by the names of the bits that are set: **TIMEOUT**, **DUMP**, **STATS**.
sockstate	The state of the socket thread: 0 starting, 1 running, 2 gone.
socket_thread	The socket thread handle, with a link to the target **TASK**.
mgr	The thread manager handle, with a link to the target **MGR**.
rname	The resource name of the server.
route	The address of the route array, with a link to the target **MOD**.
uptime	The start time of the server, as text.
ufssys	The UFS system handle, with a link to the target **FS**.
readerr_last	The time of the last console message about a read error of the file system (**HTTPD045E**), 0 if there was none.
version	The version string of the server.
readerr_held	The number of such read errors not yet reported.

cfg_maxtask, **cfg_mintask**	The configured maximum and minimum number of workers.
cfg_client_timeout	The client timeout in seconds.
smf_level, **smf_type**	The SMF level (number and name) and the SMF record type (Chapter 8, “SMF Records” on page 93).
cfg_resident	Whether the server holds the modules of its routes in storage (“Resident Modules” on page 79).
resident_held	The number of modules held.
cfg_cgictx	The configured number of context pointers (“Context Pointers” on page 79).
ufs_enabled, **dbg_enabled**	Whether UFS is enabled and whether debug output is on.
bind_tries, **bind_sleep**	The retry count and the delay in seconds for the bind of the listen port.
total_requests, **total_errors**, **total_bytes_sent**	The counters of the server. An error is a response with a status of 400 or more.
active_connections	The number of open client connections.
xlate	The address of the code page pair in effect.
giveup_last	The time of the last console message about a client that was given up.
stc_prev_acee	The address of the ACEE that the server restores at shutdown.
giveup_held	The number of such cases not yet reported.
ufs	The UFS session handle of the server.
cfg_session_maxage	The hard maximum age of a credential in minutes, 0 for off.
self	The main thread handle, with a link to the target **TASK**.
cgictx	The address of the context pointer array. A link opens it in HTTPDM.
docroot	The UFS document root.
listen_queue	The listen backlog.
cfg_res_noprof	The value of **RES_NOPROFILE**: the page shows **ALLOW** or **DENY**.
sec_open	From HTTPD 4.5.1. Whether the security product accepted the test logon at the start (“What the Server Asks of the Security Product” on page 34): the page shows **NO**, or **YES, LOGIN REFUSED** when the server wrote **HTTPD042W** and refuses all logins. The byte after it is padding.
codepage	The code page name from the **CODEPAGE** keyword. When the keyword is not set, the page shows “CP037 (default, no CODEPAGE keyword)”.
cfg_keepalive_timeout, **cfg_keepalive_max**	The keep-alive idle timeout in seconds, and the maximum number of requests per connection.
cfg_session_timeout	The idle time of a credential in minutes, 0 for off.
credkey, **credarr**	The addresses of the credential key and of the credential array. The page shows the addresses only.
cfg_realm_val	The text of the realm.

trace	The address of the request trace, 0 when it is off (“Target TRACE” on page 86).

The page shows fields that the server no longer uses as reserved, so that the table accounts for every byte of the block. It also has a small form that posts to **/.dm** and shows an arbitrary memory area (“HTTPDM, the Memory Display” on page 90).

Target MOD

The target needs the address of the route array in **m**. The link on the field **route** of the HTTPD page supplies it. Without **m** the page says **Missing memory address (&m=nnnnnnnn)**.

The page shows a dump of the array. For every route it shows a heading “Route #n” with **MOD** for a route that has a program and **LOC** for a static prefix, a dump of the route block, and a table:

Table 69. The fields shown for a route

Field	Contents
eye	The eye catcher of the route block.
wild	Whether the path contains a wildcard.
len	The length of the path.
path	The path of the route, as written in the Parmlib.
pgm	The name of the program. A **LOC=** route has none.
auth	The policy: **NONE** (public), **FORM** (an HTML login form), **BASIC** (a 401 with a **WWW-Authenticate** header), or **TOKEN** (a bare 401, for an API route).
resattr	The access level that **RES=** asks for: READ, UPDATE, CONTROL or ALTER. The value 0 means that no level was set, and READ is assumed.
resclass, **resname**	The RACF class and the resource name of the **RES=** keyword.

This is the page to use to find out which policy a running server applies to a path (Chapter 2, “Routes and Authentication Policy” on page 9).

Target MGR

The target shows the thread manager of the worker pool. It does not take an address: it ignores **m** and uses the manager of the server. The page prints **Invalid Eye Catcher** if the block is not a manager. The help page says the same: the target takes no address.

The fields are the task of the manager (with a link to the target **TASK**), the wait flag, the function that the workers run, the user data pointer, the stack size, the worker array (with a link to the target **WORKERS**), the queue, the state (**INIT**, **RUNNING**, **QUIESCE**, **STOPPED** or **WAITING**), the minimum and maximum number of tasks, the number of requests dispatched since the start, the start time and the resource name.

Target WORKERS

The target needs the address of the worker array in **m**. The link on the field **worker** of the MGR page supplies it. The page skips an entry whose eye catcher is not a worker.

For every worker the page shows a dump and a table with the eye catcher, the wait flag, the manager, the task (with a link to the target **TASK**), the queue, the state (**INIT**, **RUNNING**, **WAITING**, **DISPATCH**, **SHUTDOWN** or **STOPPED**), the options (**TIMER**, **NOWORK**), the start time, the wait time and the dispatch time, and the number of requests that the worker dispatched.

When a worker is in the state **RUNNING** and has a queue entry, the page adds a block that tells what the worker is doing: the protocol (HTTP), the user (the user ID of the credential, or "ANONYMOUS" if there is none), the group, the socket number, and the local and the remote address with the port.

Target TRACE

The target shows the request trace that the Parmlib keyword **TRACE** switches on ("Diagnostics and Recording" on page 6). It needs no address. The answer is not HTML. It has the type **text/plain** and the header **Cache-Control: no-store**. If the keyword is not set, or its value was refused, the answer is the single line

```
#HTTPTRC off
```

The query parameter **min** is a number of milliseconds. With it, the answer leaves out each finished record whose time from the start to the last mark is shorter. A record of a request that is still running is always shown. A value that is not a number counts as 0.

The first line of the answer starts with **#HTTPTRC v1** and has these words.

Table 70. The words of the first line

Word	Meaning
now	The clock at the readout, in ticks of 1.024 ms (see below).
wall	The time of the readout in seconds since 1 January 1970.
unit	The length of a tick, always **1.024ms**.
slots	The number of records, the value of **TRACE**.
seq	The number of requests that got a record since the start. It is also the number of the newest record.
lost	The number of requests that got no record. In practice all records belonged to requests that were still running.
miss	How often a phase mark found no record for its request. A request that was lost causes a miss at each of its marks.
workers	The number of workers now.

One line for each record follows. The lines are in the order of the table, not of time: sort by **seq**. A line is a row of **name=value** words without a space inside a value. The line of the request that reads the trace is among them, and the request is still running.

Table 71. The words of a trace line

Word	Meaning
seq	The number of the request since the start.
c	The address of the client block (HTTPC), in hex.
w, **tcb**	The address of the worker (the same as **WORKER(xxxxxx)** on the WORKERS page and in **HTTPD061I**) and of its task block, in hex. Both are 0 until a worker has taken the connection.
ip, **port**	The address and the port of the client.
n	The number of the request on its connection, from 0. A connection that is kept alive has one record for each request.

resp	The status code at the last mark. It is 0 as long as no response line was written.
ago	The milliseconds from the start of the record to the readout.
live	1 while the request is running, 0 when it is finished.
last, **st**	The last phase that the request reached, and the state of the client at that mark (**IN**, **PARSE**, **GET**, **HEAD**, **PUT**, **POST**, **DELETE**, **DONE**, **REPORT**, **RESET** or **CLOSE**). **last** is **q** if no phase was reached.
age	The milliseconds from the start to the last mark, or to the readout for a request that is still running.
run to **close**	The milliseconds from the start to each phase, or **-** if the request never got there (the next table).
polls	How often the server waited half a second for the next data of the request line or of a header line.
txn	The number of calls of **send** for the request, including calls that sent nothing.
txst	How often the server paused for a tenth of a second because TCP/IP took no data. The send of a chunked response and the send of a static file count these pauses.
txms	The milliseconds spent inside **send**.
sent	The bytes sent for the request. The server fills the word when the request ends, so it is 0 for a request that is still running.
path	The first 23 characters of the decoded path of the request, without the query. A byte at or below hex 40 (a blank or a control character) shows as **?**. The word is empty until the request is parsed. It is always the last word.

The start of a record is the accept of the connection for the first request of a connection. For a later request on the same connection it is the end of the request before, so the time that the client needs to send its next request is part of the times of that record. These are the phases, in the order a request normally reaches them.

Table 72. The phases of a request

Word	The mark is set when ...
run	a worker took the connection. For a later request of a connection it is 0, because the worker still holds the connection.
line1	the request line was read.
req	the server stopped reading the request line and the header lines. The mark is also set when the read failed: the client closed the connection, the read timed out, or the request line has no second word. Then **line1** can be **-**. The request goes no further, so **done** stays **-** and **resp** is 0. The first request of a connection ends with the mark **end**. A later request of a connection that is kept alive ends with the mark **close** and has no **end**.
parse	the request was parsed. For a POST the server has read the body by then.
cred	the credential of the request was resolved, with or without a result.
route	the request passed the check of the policy of its route. A request that the check refuses, and the paths **/login** and **/logout**, do not get the mark.

lnk0, **lnk1**	just before the module of a **MOD=** route is linked, and when it returns. A static file has neither. The time between the two is the run of the module, including its load when the server does not hold it in storage (“Resident Modules” on page 79).
snd1, **sndl**	the first and the last call of **send** that handed data to TCP/IP. The headers count as data.
done	the response is complete.
end	the request was reported (SMF and the counters) and the server is about to reset the connection.
close	the connection was closed while the record was still open. It is the mark of a connection that ends without another request.

This is the record of one request, taken from the lab server. The command puts the words three to a row; the order of the words in the line is from left to right and from top to bottom:

```
$ curl -s -u IBMUSER:****** "http://localhost:8080/.dsrv?target=TRACE" | grep "^seq=3 " | tr " " "\n" | xargs -n 3 printf "%-16s %-16s
%s\n"
seq=3            c=1C0FC8         w=17F9C0
tcb=9BB378       ip=192.0.2.17    port=36640
n=0              resp=401         ago=1142
live=0           last=end         st=RESET
age=28           run=2            line1=3
req=6            parse=7          cred=7
route=7          lnk0=8           lnk1=27
snd1=18          sndl=21          done=27
end=28           close=-          polls=0
txn=12           txst=0           txms=0
sent=292         path=/zosmf/info
```

The request was the first on its connection (**n=0**). A module answered it with the status 401. The server read the request line 3 ms after the accept and had all header lines at 6 ms (**req**). The check of the policy of the route passed at 7 ms. The module was linked at 8 ms and returned at 27 ms (**lnk0**, **lnk1**). It handed its first data to TCP/IP at 18 ms and its last at 21 ms (**snd1**, **sndl**), in 12 calls of **send** that took together less than a millisecond. The response was complete at 27 ms, and the request was over at 28 ms.

The next example shows a connection that stays open. The command shows 11 of the words of each record. Records 3 and 4 are the two requests of one connection, and record 5 is the end of that connection. Records 1 and 2 are another connection, and record 6 is the request that reads the trace:

```
$ curl -s -u IBMUSER:****** "http://localhost:8080/.dsrv?target=TRACE" | tail -n +2 | cut -d" " -f1,3,7,8,10,11,13,14,16,26,27
seq=1 w=17FA40 n=0 resp=401 live=0 last=end age=60 run=6 req=16 close=- polls=0
seq=2 w=17FA40 n=1 resp=0 live=0 last=close age=3 run=0 req=3 close=3 polls=0
seq=3 w=17F9C0 n=0 resp=401 live=0 last=end age=83 run=5 req=16 close=- polls=0
seq=4 w=17F9C0 n=1 resp=401 live=0 last=end age=48 run=0 req=5 close=- polls=0
seq=5 w=17F9C0 n=2 resp=0 live=0 last=close age=508 run=0 req=507 close=508 polls=1
seq=6 w=17FB40 n=0 resp=0 live=1 last=sndl age=24 run=2 req=7 close=- polls=0
```

Records 3 and 4 have the same worker (**w**), and **n** counts 0 and 1. Record 4 has **run=0**: the worker already held the connection, so the mark is the start of the record. Record 5 has no request line (**last=close**, **resp=0**). It is the time from the end of record 4 to the close of the connection, 508 ms, and in it the server waited once for the next data (**polls=1**). Record 6 is still running: **live=1**, and its last mark so far is **sndl**, because the server is sending the page that shows it.

The last example shows requests whose read failed. The command shows 14 of the words of each record:

```
$ curl -s -u IBMUSER:****** "http://localhost:8080/.dsrv?target=TRACE" | tail -n +2 | cut -d" " -f1,3,7,8,10,11,13,14,15,16,24,25,26,27
seq=1 w=17FA40 n=0 resp=0 live=0 last=end age=5 run=3 line1=- req=5 done=- end=5 close=- polls=0
seq=2 w=17F9C0 n=0 resp=401 live=0 last=end age=25 run=2 line1=3 req=4 done=24 end=25 close=- polls=0
seq=3 w=17F9C0 n=1 resp=0 live=0 last=close age=1004 run=0 line1=- req=1003 done=- end=- close=1004 polls=2
seq=4 w=17FB40 n=0 resp=0 live=0 last=end age=9526 run=2 line1=- req=9525 done=- end=9526 close=- polls=19
seq=5 w=17FA40 n=0 resp=0 live=1 last=sndl age=18 run=1 line1=2 req=4 done=- end=- close=- polls=0
```

Record 1 is a connection that the client closed before it sent anything. The server stopped reading at 5 ms (**req**), there is no request line (**line1=-**), and the record ends with **end**. Record 2 is a request that was answered with 401, and record 3 is the end of its connection after 1,004 ms, as in the example before: the connection was kept alive and the client closed it. That record has **close** and no **end**. Record 4 is a connection that sent nothing until the server gave up after about ten seconds, the default of **CLIENT_TIMEOUT**. The server waited 19 times for half a second (**polls=19**), and the record ends with **end**. None of the three has **done**, and all have **resp=0**.

The times come from the TOD clock. The server shifts the clock value right by 22 bits and keeps 32 bits, so one tick is 1.024 milliseconds. Only **now** is in ticks. The server converts every other time of a line to real milliseconds (1000 ticks are 1024 milliseconds) and drops the fraction. The tick count wraps after about 51 days. The trace only uses the difference of two ticks, so the wrap does not matter, but the **now** of one readout cannot be compared with the **now** of a much later one.

The table does not keep the records in time order. A new record takes the next free place, counted from a position that moves by one for each request. A record of a running request is never taken. A record of a finished request that took 977 ticks (about one second) or more is taken only when no other record is left, so a slow request stays in the table longer than a fast one. The time of a request counts from the accept if it is the first request of its connection, and from its request line if it is a later one, so the time that a client takes to send its next request does not make a request slow. When all records belong to running requests, the next request gets none and **lost** grows.

Note: The record holds the address and the port of the client and the first 23 characters of the path of each request. Anyone who may call the HTTPDSRV route can read them. Give the route an **AUTH=** (see the start of this chapter), and switch **TRACE** on only while you look for a problem.

Target TASK

The target needs the address of a thread handle in **m**. The fields are the eye catcher, the task control block address (**tcb**), **owntcb**, the termination ECB (**termecb**), the return code, the stack size, the function, its two arguments, and the address of the stack. A link on the stack opens it in HTTPDM with the stack size as the length.

Target FILE

The target needs the address of a C file handle in **m**. The fields are the eye catcher; the DCB address (a link opens it in HTTPDM); the assembler buffer; the logical record length and the block size; the file position; the buffer address (a link opens it) with the pointers into it; the flags, which are decoded as **DYNAMIC**, **OPEN**, **READ**, **WRITE**, **APPEND**, **BINARY**, **RECORD**, **BSAM**, **TERM**, **ERROR** and **EOF** the record format, which is decoded (fixed, variable or undefined, and the attributes blocked, spanned, ASA, machine control, key length); the DD name, the member, the data set name and the mode.

Target FS

Without **m** the page shows the UFS system block of the server: a dump of the block and the eye catcher. A line tells that the state of the disks and inodes belongs to the UFSD started task. HTTPD holds only the client side of UFS, so the page cannot show more.

Do not use **m** with this target. With **m**, the module does not display the block at that address. If the storage at the address does not start with one of the two block names that the module looks for, the module sends no response.

The Help Page

A value of **target** that matches no name gives the page “HTTPDSRV Help”. It has the status 200, shows the usage line with the host and the path of the request, and lists the targets. The host and the path are escaped for HTML.

HTTPDM, the Memory Display

HTTPDM shows a block of storage as a hexadecimal dump with a character column. It is the module behind the links in the pages of HTTPDSRV. It answers every request method in the same way.

Parameters

HTTPDM takes the first name that is present from each row of the table.

Table 73. The query parameters of HTTPDM

Parameter	Default	Meaning
title, **t**	none	A title that is printed in the first line of the dump. In the HTML form the module escapes **<**, **>**, **&**, **"** and **'**, and the escaped title has at most 255 characters, a replacement is never cut in two. With **d** other than 0 it is written as it is.
memory, **mem**, **m**	0	The address to dump, in hexadecimal. HTTPDM keeps the low 24 bits of the value.
length, **len**, **l**	256	The number of bytes to dump. The module converts the number as decimal, or as hexadecimal with a leading 0x. A value of 0 or less gives 256, a value above 4096 gives 4096.
chunk, **size**, **c**, **s**	16	The number of bytes in a line. See the rule below.
data, **d**	0	The output form. 0 gives an HTML page, any other number gives plain text. The module converts the number as decimal, or as hexadecimal with a leading 0x.

The rule for **chunk**: a number of 0 gives 16. A number above 64 gives 64. Any other number is rounded up to a multiple of 4. So 1 to 4 give 4, 5 to 8 give 8, and 61 to 64 give 64.

Output

With **d=0** (the default) the response is **text/html**. It has a form with the fields Memory Address, Data Length and Chunk Size, and a preformatted block with the dump. With **d** set to another number, the response is **text/plain** and holds the dump only, without the form and without any report of an abend.

The dump starts with the line

```
Dump of [title ]AAAAAAAA (n byte chunks) (m bytes total)
```

in which **AAAAAAAA** is the address in hexadecimal, and **n** and **m** are the chunk size and the length. One line follows for every chunk. A line has the offset from the start (a plus sign and five hexadecimal digits), then the bytes in hexadecimal in groups of four, and then the characters of the bytes between colons. The character column shows the bytes as EBCDIC characters. A byte that is not a printable character shows as a period.

In the HTML form every group of four bytes is a link. The link dumps the storage to which the four bytes point, with the same length, chunk size and data parameters. The link has the form **?m=address&l=length&c=chunk&d=data**. So you follow a chain of pointers with a click. In the character column the characters **<**, **>** and **&** are written as HTML entities.

An address that the task cannot read gives, in the HTML form, the line **ABEND S0C4 occured for DISPLAY MEMORY 0x...** in place of the rest of the dump. A user abend shows as **ABEND Unnnn occured...**. If the abend handler cannot be created, the line is

ESTAE CREATE failure, RC=0x... with the code. With **d** other than 0 these lines are not written and the output simply stops.

Limits

- The largest dump is 4096 bytes per request.
- HTTPDM keeps the low 24 bits of the address, because the system uses 24-bit addresses. The form always shows the address that the module used.
- The route belongs behind a login (“Setting Up the Modules” on page 81), because the dump shows storage of the server.
- The dump can reach any storage of the server address space that the task may read. That may include data that the server holds for its users, such as cached credentials.

HTTPDMTT, the Master Trace Table

HTTPDMTT shows the master trace table of the system. The table holds the recent messages of the system console. The module makes a copy of the table and prints the entries in order.

Parameters

Table 74. The query parameter of HTTPDMTT

Parameter	Default	Meaning
data, **d**	0	The output form. The module converts the number as decimal. 0 gives an HTML page, any other number gives plain text.

Output

With **d=0** the response is **text/html**. The page has the heading “Master Trace Table”, then one preformatted block with one line for every entry of the table. With **d** set to another number, the response is **text/plain** and has the lines only. The module prints the text of an entry as text. It does not read the text as a format, so a percent sign in a message does no harm.

In the HTML page the module escapes **<**, **>**, **&**, **"** and **'** in every entry, so a message such as **<b>** shows as that text and is not read as markup. The table holds the messages of every job, not only those of HTTPD. In the plain form the entries are as the table holds them, without escaping. The module reads an entry up to its length in the table, or up to a null character if that comes first.

If the module cannot get storage for the copy, it prints “Unable to allocate Master Trace Table.” and ends. The module frees the copy at the end.

The module sends an entry in pieces of at most 255 characters, after any escaping. It checks the sends of the response header, of the start of the HTML page up to the body tag, and of the entries. At the first of these that fails, for example because the client has closed the connection, it stops and sends nothing more: not the rest of the entry, not the next entries and not the closing lines of the HTML page. An entry longer than one piece can therefore end in the middle of the line. The module frees the copy in this case too. It does not check the sends of the heading, of the opening line of the preformatted block, of the closing lines of the page and of the allocation message; a failure there does not stop it.

Limits

- The table is the master trace table of the system, so it shows every message that went to the console. It exposes the console log to everyone who may use the route. It needs strong authentication.
- The module does not select, filter or search. Take the plain form (**d=1**) and use a tool of your own for that.

ABEND0C1, the Abend Test

ABEND0C1 takes storage and abends with a program check (S0C1). It exists to prove that HTTPD gives back the storage of a failed module and keeps its workers (“Abends” on page 77). Call it many times and check that the other modules still load and run as fast as before.

The module looks for the text **kb=** anywhere in the query string, not case sensitive, and reads the digits after it. An example is **/.abend?kb=512**.

Table 75. The query parameter of ABEND0C1

Parameter	Default	Meaning
kb	128	The amount of storage to take, in kilobytes. A value above 1024 gives 1024. A value of 0, a missing value and a value that is not a number give 128.

What the module does:

1. It takes the storage in blocks of 16 KB with **malloc** and writes the byte hex C1 into every block. It stops when no storage is left.
2. It writes the console message **HTTPD921W**. The message names the amount taken in KB and the number of blocks.
3. It abends with S0C1, without freeing the blocks. It sends no response itself.

The client then gets the answer of HTTPD for a failed module: the status 503 with the text “External program ABEND0C1 failed with S0C1 ABEND”. HTTPD releases the storage that the module took (“Abends” on page 77).

The default amount does not endanger the server. The module is a probe that you may point at a running system.

Note: Register this module only for a test, and with **AUTH=BASIC** or stronger. A route to it makes a failure on request.

Chapter 8. SMF Records

HTTPD writes two kinds of SMF record. The *request record* tells what one HTTP request did. The *session record* tells what one client connection did. Both start with the same 18-byte header and have a fixed body. This chapter lists the record types, the header, both bodies field by field, the levels that choose which records HTTPD writes, and how to read the fields.

HTTPD writes the records through the SMF routines of the C library. It writes nothing while SMF is not active in the system (“Levels and Parmlib Keywords” on page 96).

Record Types and Subtypes

All records of HTTPD have one SMF record type. The default is 243. The Parmlib keyword **SMF** sets another type in the range 128 to 255 (“Levels and Parmlib Keywords” on page 96). The subtype is a halfword in the body of the record, at offset 26 (hex 1A).

Table 76. The subtypes of the HTTPD record

Subtype	Name	Length	Meaning
1	Request	124 bytes	One record for each HTTP request that the level selects. HTTPD writes it when it has finished the request.
2	Session	56 bytes	One record for each client connection. HTTPD writes it when it closes the connection, and only at the level **ALL**.

The eight-byte field **subsys** at offset 18 is always the characters **HTTPD** followed by three blanks, in both subtypes. The characters are in EBCDIC.

The Header

Every record starts with the standard SMF header of 18 bytes. HTTPD sets all its fields. The body begins at offset 18.

Table 77. The 18-byte header, the same in both subtypes

Offset	Hex	Length	Type	Meaning
0	00	2	binary	The length of the whole record: 124 for the request record, 56 for the session record.
2	02	2	binary	The segmentation descriptor. HTTPD sets it to 0.
4	04	1	flags	The system indicator. HTTPD sets it to 2.
5	05	1	binary	The record type: the value of **TYPE=**, 243 by default.
6	06	4	binary	The time of day in hundredths of a second since midnight, local time.
10	0A	1	binary	The century indicator of the date: 0 for the years 19xx, 1 for 20xx.

11	0B	3	packed	The date as the digits **YYDDD** and the sign C, which is positive: the two digits of the year within the century and the three digits of the day of the year.
14	0E	4	characters	The system ID (SID) of the system that writes the record.

HTTPD takes the time and the date from the local clock when it writes the record. For a request record this is the time at which the request ended. For a session record it is the time at which the connection closed.

The Request Record

The request record has 124 bytes (hex 7C). HTTPD sets the whole record to binary zeros first and then fills the fields. A field that the text calls “padded with zeros” therefore ends in binary zeros, not in blanks.

Table 78. The request record, subtype 1

Offset	Hex	Length	Type	Meaning
0	00	18	header	The header (“The Header” on page 93).
18	12	8	characters	**subsys**: **HTTPD** and three blanks.
26	1A	2	binary	**subtype**: 1.
28	1C	8	characters	**userid**: the user of the request. Eight blanks when the request has no credential.
36	24	4	binary	**client_addr**: the IPv4 address of the client. The four bytes are in the order a.b.c.d (the network order), so the first byte is the first number of the address.
40	28	4	unsigned binary	**resp_code**: the HTTP status code of the response, for example 200 or 404.
44	2C	4	unsigned binary	**bytes_sent**: the number of bytes that HTTPD handed to the network for this response.
48	30	4	unsigned binary	**duration_us**: the duration of the request in microseconds.
52	34	8	characters	**method**: the request method, for example **GET**, cut to 8 characters and padded with zeros.
60	3C	64	characters	**uri**: the path of the request, decoded and without the query string, cut to 64 characters and padded with zeros. A path of 64 or more characters has no terminating zero.

userid
: The user ID of the credential that the request carried. A user ID of fewer than 8 characters is not padded with blanks. The credential store pads it with binary zeros, so read the field up to the first zero or blank. A request without a credential, for example a request to a public route, has 8 blanks.

bytes_sent
: The counter belongs to one request. It counts every byte that HTTPD sends for the response, which are the status line, the headers and the body, and the framing of a chunked response. The counter starts again at zero for the next

request of a kept-alive connection. When the client breaks the connection, the field holds the bytes that went out before the break.

duration_us
: The time from the start mark to the end mark of the request. HTTPD sets the start mark when it accepts the connection, and again each time it resets a kept-alive connection for the next request. For a request on a kept-alive connection the duration therefore includes the time that HTTPD waited for the request to arrive. HTTPD sets the end mark when the response is complete. A negative difference is stored as 0. The field is unsigned and holds about 4 294 seconds at most.

method, uri
: The URI is the path only. The query string is not recorded.

The Session Record

The session record has 56 bytes (hex 38). HTTPD writes it once for a client connection, when it closes the connection. It exists only at the level **`ALL`**.

Table 79. The session record, subtype 2

Offset	Hex	Length	Type	Meaning
0	00	18	header	The header (“The Header” on page 93).
18	12	8	characters	**`subsys`**: **`HTTPD`** and three blanks.
26	1A	2	binary	**`subtype`**: 2.
28	1C	8	characters	**`userid`**: the user of the connection when it closed. See below.
36	24	4	binary	**`client_addr`**: the IPv4 address of the client, in network order as in the request record.
40	28	4	unsigned binary	**`connect_time`**: the local time at which HTTPD accepted the connection, in hundredths of a second since midnight.
44	2C	4	unsigned binary	**`duration_us`**: the duration of the connection in microseconds. See below.
48	30	4	unsigned binary	**`request_count`**: the number of requests after which the connection stayed open. See below.
52	34	4	unsigned binary	**`total_bytes`**: the sum of the **`bytes_sent`** values of the requests of the connection.

userid
: HTTPD takes the credential away from the connection when it resets the connection at the end of each request, and it writes the session record after that. The field is therefore normally 8 blanks. It holds a user ID only when the worker closes the connection without a reset, which is the case after a failure in the worker.

duration_us
: HTTPD takes the local time of the close, in hundredths of a second since midnight, subtracts **`connect_time`** and multiplies the result by 10 000. If the close time is smaller than the connect time, which happens when the connection spans midnight, the field is 0. The resolution is a hundredth of a second.

request_count
: HTTPD adds 1 each time it resets the connection for the next request. A connection that carried one request and was then closed by the server has the

value 0. A connection that carried three requests and was left idle until the timeout has the value 3.

Levels and Parmlib Keywords

The level chooses which records HTTPD writes. The Parmlib keyword is **SMF**, with an optional **TYPE=**. The list of all keywords is in Chapter 1, "The HTTPPRM Keywords" on page 1.

```
SMF=ALL TYPE=244
```

Table 80. The recording levels

Level	HTTPD writes
NONE	No records. This is the default.
ERROR	A request record for each request whose status code is 400 or more.
AUTH	A request record for each request with the status 401 or 403, and for each other status of 400 or more. The set of requests is the same as at **ERROR**.
ALL	A request record for every request, and a session record for every connection.

The rules for the keyword:

- The level is the first word after the equal sign. HTTPD compares the word with the names **NONE**, **ERROR**, **AUTH** and **ALL**, in this order, for the length of the word that you write. The comparison is not case sensitive, and a word that is the start of a name is taken for that name. **ERR** is **ERROR**. **A** is **AUTH**, because **AUTH** comes before **ALL**. Write the whole name.
- A word that is not a level gives the message **HTTPD431W** with the value of the keyword. The level stays as it was, which is **NONE** at start.
- HTTPD searches the rest of the value for **TYPE=**. Capital letters are required. It converts the number after it as a decimal number. The number must be in the range 128 to 255. A number outside the range gives the message **HTTPD432W** with the number, and the type stays as it was, which is 243 at start.
- The keyword may be used without **TYPE=**. The type then stays.

The counters of the server (the total of requests, of errors and of bytes sent) are updated for every request at every level, also at **NONE**.

When the level at start is above **NONE** and SMF is not active in the system, HTTPD writes the message **HTTPD430W** to the console. The server runs and writes no records. HTTPD checks for every record whether SMF is active, so records appear when SMF becomes active later. The SMF parameters of the installation decide which record types the system keeps. This book does not describe them.

The operator can change the level while the server runs, with the command **F HTTPD,SET STATS** and one of the four names. The word **RESET** after the level, or alone after **STATS**, sets the counters to zero. The messages **HTTPD126I** (level set) and **HTTPD125I** (counters reset) confirm the two actions. The commands are in Chapter 3, "Console Commands" on page 17, the messages in Appendix A, "Messages" on page 99. The display commands show the type and the level (**HTTPD132I** for the configuration, **HTTPD123I** for the statistics). A change by command lasts until the server stops. It does not change the Parmlib member.

Reading the Records

A program that reads the records follows these rules.

- All binary numbers are big endian, as the System/370 stores them. A program on a little-endian machine swaps the bytes of the halfwords and of the 4-byte numbers.
- The fields marked “characters” are EBCDIC, also the user ID, the method and the URI. The IPv4 address is not text. It is four bytes, each one a number from 0 to 255.
- The time in the header and **`connect_time`** are hundredths of a second since midnight in local time. Divide by 100 for seconds. One hour is 360 000, one minute is 6 000.
- The date in the header is split in two fields. The century indicator at offset 10 is 0 for 19xx and 1 for 20xx. The three bytes at offset 11 are packed decimal **`YYDDDC`**. The 15th of March 2027 is the 74th day of the year. The century indicator is 1 and the three bytes are hex **`27 07 4C`**.
- The two **`duration_us`** fields are unsigned 4-byte numbers. They wrap after about 4 294 seconds (71 minutes).
- Read **`uri`** and **`method`** up to the first binary zero, or up to the length of the field.

Note: HTTPD fills the time, the date, the system indicator and the system ID through the C library of the platform. The layout of these header fields in this chapter is the layout that the library documents for them.

Appendix A. Messages

This appendix lists every numbered message of HTTPD in the order of the numbers. The header file of the product has one text for each message, and this appendix has one entry for each of those texts: 178 texts, which carry 170 distinct combinations of number and severity letter. Where two texts share a number, both are listed.

A message has the form **HTTPDnnnx**. The letter at the end is the severity: **I** information, **W** warning, **E** error. There is no other letter. The letter does not tell what the server does next. Some messages with **E** only warn, and some with **W** end the start. Every entry names the action, and it names the return code where the server ends.

In the texts, the parts in *italic* vary: a number, a name, a path. The rest is fixed. HTTPD writes the fixed text in upper case. The values that it shows as they are, such as paths and names that a user typed, keep their case. A run of blanks in a text is shown as one blank. The messages have no trailing new line.

Every numbered message is written with a single-line WTO. It goes to the console, to the master trace table (MTT) and to the hardcopy log. A line longer than about 70 to 90 characters is cut by the system. HTTPD writes no numbered message to a data set. The display module for the trace table (**HTTPDMTT**, Chapter 7, “The Display Modules” on page 81) shows the table from inside the server.

A few console lines have no number. They are listed at the end of this appendix (“Lines Without a Number” on page 118). What a client causes (a bad request, an unknown route, a file that is not there) is answered in the HTTP response and does not appear here.

The ranges are:

000 to **019**	Identity, login and the standard files (“Lifecycle and Identity (000 to 019)” on page 101).
020 to **048**	Configuration and the start of the listener (“Configuration and Start (020 to 048)” on page 103).
050 to **099**	Listener, workers and the stop (“Listener, Workers and Stop (050 to 099)” on page 107).
100 to **199**	Console commands (“Console Commands (100 to 199)” on page 109).
410 to **444**	Routes, SMF and resident modules (“Routes, SMF and Resident Modules (410 to 444)” on page 113).
901 to **999**	Diagnostics (“Diagnostics (901 to 999)” on page 116).

The numbers 020 to 048 belong to the configuration although they come from the time before the 4xx block. To find a message, search by the number and do not rely on the range.

A Healthy Start and Stop

A server that starts without a problem writes these messages in this order. A message with a condition appears only under that condition.

1. **HTTPD000I**, the banner with the version and the commit.
2. **HTTPD005I**, the version of the C library.
3. **HTTPD006W**, only for a build from a modified working tree.

4. **HTTPD002I**, how the server got its authorization. The message is written only if the authorization worked.
5. **HTTPD004I**, the identity of the started task (or **HTTPD003W** or **HTTPD004W** if it could not be set).
6. **HTTPD042W**, only if the security product accepts any user ID. In a healthy start RAKF writes **RAKF0004** for the user ID **HTTPPRB** at this place.
7. The messages of the configuration ("Configuration and Start (020 to 048)" on page 103 and "Routes, SMF and Resident Modules (410 to 444)" on page 113): the warnings about the parameters, and one **HTTPD036I** or **HTTPD417I** for each route.
8. **HTTPD054I**, the listener is bound to the port. A bind that has to wait for the port adds **HTTPD030E** and **HTTPD030I** first.
9. The warnings that follow the bind, such as **HTTPD430W**, **HTTPD425W** or **HTTPD426W**, and **HTTPD440W** to **HTTPD442W**.
10. **HTTPD066I**, only if **TRACE=** is set: the request trace is on. (**HTTPD067W** instead if the storage for it was not available.)
11. **HTTPD061I** for the socket thread and for each worker thread. The threads write these lines themselves, so their order is not fixed.
12. **HTTPD001I**, the last message of the start. After it the server answers the clients and the console commands.

The stop by **P HTTPD** writes:

1. **HTTPD098I**, the stop begins.
2. **HTTPD040I**, while the socket thread still runs. The message is repeated up to nine times, one second apart. If the thread does not end, **HTTPD041I** follows.
3. **HTTPD060I** for the socket thread and for each worker thread, written by the threads.
4. **HTTPD416I**, the statistics of the run.
5. **HTTPD099I**, the last message of the stop. The return code is 0.

A Start That Ends

Some messages end the start. The return code of the job tells which kind.

Table 81. Messages that end the start

Message	Return code	What happens afterwards
HTTPD037E, **HTTPD028E**, **HTTPD030E**, **HTTPD031E**, **HTTPD420E**, **HTTPD038E**, **HTTPD033E**, **HTTPD064E**	8	The server cleans up and stops. **HTTPD098I**, **HTTPD416I** and **HTTPD099I** follow. The listener number in the statistics is 0.
HTTPD090E	8	The server stops at once. No other stop message follows.
HTTPD014E, **HTTPD015E**, **HTTPD016E**, **HTTPD017W**, **HTTPD018W**, **HTTPD019W**	12	The program ends before the banner.
HTTPD012E	the return code of the authorization call	The banner follows, without **HTTPD002I**, and the program ends.

Note: The messages **HTTPD048E** and **HTTPD427E** set an error flag in the configuration. **HTTPD420E** follows at the end of the configuration and the start ends with the return code 8.

Lifecycle and Identity (000 to 019)

Table 82. Messages 000 to 019

Message	Meaning and action
HTTPD000I HTTPD *version* **(**_commit_**) STARTING**	The first message of the start. It names the version and the commit of the build, in upper case. HTTPD writes it also when the authorization failed (**HTTPD012E**).
HTTPD001I HTTPD *version* **READY - SERVING** *docroot*	The last message of a healthy start. The server accepts clients. The document root keeps its case.
HTTPD001I HTTPD *version* **READY - NO DOCUMENT ROOT**	The same message for a server whose document root is empty.
HTTPD002I AUTHORIZED BY LIBRARY (MODULE KEY 0)	The program was authorized when it started: the load library is APF authorized. HTTPD infers this from the state of the program before its own authorization call. The message is written only if the setup worked.
HTTPD002I AUTHORIZED BY SVC (MODULE KEY 8)	The program was not authorized when it started. The authorization call of the C runtime made it so. The message is written only if the setup worked.
HTTPD003W RACINIT SKIPPED, CANNOT ENTER SUPERVISOR STATE	The server cannot enter supervisor state, so it does not set the identity of the started task. **System action.** The identity is not changed. The start goes on with the identity that the task inherited.
HTTPD004I STC IDENTITY SET TO *user*/*group* **VIA RACINIT**	The started task has the identity that is shown. The default is **HTTPD**/**USER**. The environment variables **HTTPD_STC_USER** and **HTTPD_STC_GROUP** change it. The **PARM** of the JCL changes it with **STCUSER=** and **STCGROUP=**.
HTTPD004W RACINIT ENVIR=CREATE FAILED FOR *user*/*group* **RC=**_rc_	The security product refused the identity. *rc* is the return code of the call. **System action.** The start goes on with the inherited identity.
HTTPD005I *text*	The version of the C library that is linked into the server, in upper case.
HTTPD006W BUILT FROM A MODIFIED WORKING TREE	The program was built from sources with changes that are not committed. The commit in **HTTPD000I** does not describe the code.
HTTPD007I USER *userid* **IP** *a.b.c.d* **LOGIN SUCCESSFUL ACEE(**_acee_**)**	A client logged in with the login form. *userid* is the user (up to 8 characters), then the address of the client and the address of the security environment (6 hex digits).

HTTPD008I USER *userid* **IP** *a.b.c.d* **LOGOUT SUCCESSFUL**	A request to **/logout** released the credential of the client.
HTTPD008W USER *userid* **IP** *a.b.c.d* **LOGOUT FAILED**	The client has a credential, but the release of it failed. The client gets the answer "Logout failed.".
HTTPD009W LOGIN SESSION NOT CREATED, RACINIT FAILED RC=*rc*	From HTTPD 4.5. The security product failed the logon with a return code that is not a plain refusal. A refusal (the return codes 4, 8, 12 and 28) is silent. The source names three other cases as examples: the product is not active (32), an installation exit refuses the user (24), and the terminal or the time of the user is restricted (hex 30). The client sees "Login failed, no session available". **System action.** The login creates no session. The server goes on.
HTTPD010I *name* **IS APF AUTHORIZED**	Not issued in this release. The text is defined, and no code that is built writes it.
HTTPD011I *name* **WAS APF AUTHORIZED VIA SVC 244**	Not issued in this release. The text is defined, and no code that is built writes it.
HTTPD012E *name* **UNABLE TO DYNAMICALLY OBTAIN APF AUTHORIZATION**	The authorization call failed. *name* is the name that the program was started with. **System action.** The banner (**HTTPD000I**) and **HTTPD005I** follow, without **HTTPD002I**. The program ends with the return code of the call.
HTTPD013I STEPLIB IS NOW APF AUTHORIZED	Not issued in this release. The text is defined, and no code that is built writes it.
HTTPD014E SYSPRINT DD NOT ALLOWED, HTTPD USES HTTPDOUT FOR STDOUT	The JCL has a DD statement **SYSPRINT**. HTTPD refuses the name. **System action.** The program ends with the return code 12. **Response.** Remove the DD statement. HTTPD uses the private DD name **HTTPDOUT**.
HTTPD015E SYSTERM DD NOT ALLOWED, HTTPD USES HTTPDERR FOR STDERR	The JCL has a DD statement **SYSTERM**. HTTPD refuses the name. **System action.** The program ends with the return code 12. **Response.** Remove the DD statement. HTTPD uses the private DD name **HTTPDERR**.
HTTPD016E SYSIN DD NOT ALLOWED, HTTPD USES HTTPDIN FOR STDIN	The JCL has a DD statement **SYSIN**. HTTPD refuses the name. **System action.** The program ends with the return code 12. **Response.** Remove the DD statement. HTTPD uses the private DD name **HTTPDIN**.
HTTPD017W HTTPDOUT DD NOT DEFINED, STDOUT IS NOT AVAILABLE	HTTPD could not open the DD name **HTTPDOUT** for output.

	System action. Although the letter is W, the start ends. The program closes the files that it opened and ends with the return code 12. **Response.** Add the DD statement to the JCL of the started task.
HTTPD018W HTTPDERR DD NOT DEFINED, STDERR IS NOT AVAILABLE	HTTPD could not open the DD name **HTTPDERR** for output. **System action.** Although the letter is W, the start ends with the return code 12. **Response.** Add the DD statement to the JCL of the started task.
HTTPD019W HTTPDIN DD NOT DEFINED, STDIN IS NOT AVAILABLE	HTTPD could not open the DD name **HTTPDIN** for input, and it could not open the data set **NULLFILE** as a replacement. **System action.** Although the letter is W, the start ends with the return code 12. **Response.** Add the DD statement to the JCL of the started task.

Configuration and Start (020 to 048)

The configuration is read from the DD name **HTTPPRM** (Chapter 1, "The HTTPPRM Keywords" on page 1). The rest of this range reports the start of the listener.

Table 83. Messages 020 to 048

Message	Meaning and action
HTTPD020W CANNOT OPEN DD:*ddname* **-- USING DEFAULTS**	The DD name of the configuration (**HTTPPRM**) cannot be opened. **System action.** The server starts with its defaults: port 8080, 3 to 9 workers, the document root **/www**, no routes and no SMF records (Chapter 1, "The HTTPPRM Keywords" on page 1 lists the defaults). **Response.** Add the DD statement if the defaults are not what you want.
HTTPD020E FOPEN FOR DD:HTTPDBG FAILED, ERRNO=*errno*	The keyword **DEBUG=** is greater than 0, and the DD name **HTTPDBG** cannot be opened. **HTTPD021I** follows. **System action.** The server goes on. The debug output goes to the standard error file.
HTTPD021I DEBUG/TRACE OUTPUT WILL BE TO DD:SYSTERM	The debug output goes to the standard error file. The text names **SYSTERM**, but HTTPD allocates the standard error file as **HTTPDERR**.
HTTPD023E INVALID PORT VALUE (*port***)**	The value of **PORT=** is not a number from 1 to 65535. **System action.** The server keeps the earlier value or the default and goes on.

HTTPD024W CONFIG=*value* **IS IGNORED, CONFIGURATION COMES FROM THE** *ddname* **DD**	The **PARM** of the JCL has **CONFIG=**. HTTPD ignores it and reads the DD name. *value* is cut after 32 characters. **System action.** The server goes on. **HTTPD024I** follows.
HTTPD024I USE S HTTPD,M=MEMBER TO SELECT A DIFFERENT MEMBER	The hint that follows **HTTPD024W**.
HTTPD025W TZOFFSET IS NO LONGER USED, THE SYSTEM OFFSET GMT *signhh*:*mm* **APPLIES**	The configuration has the retired keyword **TZOFFSET=**. The server uses the time zone offset of the system. **System action.** The server goes on. **HTTPD025I** follows.
HTTPD025I SET TZ IN THE SYSENV OR ENVIRON DD TO OVERRIDE IT FOR ALL TASKS	The hint that follows **HTTPD025W**.
HTTPD026W UNRECOGNIZED CONFIGURATION LINE: *line*	A line of the configuration is not a comment and has no equal sign. The form is **KEYWORD=VALUE**. A comment starts with **\#** or *****. The text shows the first 40 characters of the line. **System action.** The line is skipped. The server goes on.
HTTPD026E SESSION_TIMEOUT (*minutes* **MIN) <= CLIENT_TIMEOUT (***seconds* **SEC), RAISE IT**	The session timeout (in minutes) is not longer than the client timeout (in seconds). The message is written after the listener is bound, and only if the session timeout is not 0. **System action.** Although the letter is E, this is a warning. The server goes on with the values as they are. **Response.** Raise **SESSION_TIMEOUT=** in the configuration.
HTTPD027I CLOSING STALE SOCKET *socket* **ON PORT** *port*	The bind failed because the address is in use. The server closes each other socket of the address space that is bound to the port, then waits 2 seconds and binds once more.
HTTPD028E SOCKET() FAILED, RC=*rc* **ERRNO=** *errno*	The server could not create the listener socket. **System action.** The start ends with the return code 8.
HTTPD029W UNKNOWN CONFIGURATION KEYWORD: *keyword*	A line of the configuration has a keyword that HTTPD does not know. The keyword is shown in upper case. **System action.** The line is skipped. The server goes on.
HTTPD030E BIND() FAILED FOR HTTP PORT, RC=*rc* **ERRNO=***errno*	The bind of the listener to the port failed. If the cause is an address in use, the server waits and tries again (**HTTPD030I**), up to **BIND_TRIES=** times, **BIND_SLEEP=** seconds apart. The message is written

	for the first failure. It is written a second time if the last try also fails. For any other cause, it is written twice in a row. **System action.** A try that works within the limit lets the start go on. Otherwise the start ends with the return code 8.
HTTPD030I EADDRINUSE, WAITING FOR TCPIP TO RELEASE HTTP PORT *port*	The port is still in use, for example after a quick restart. The server waits **BIND_SLEEP=** seconds (default 10) and binds again. The default number of tries is 10 (**BIND_TRIES=**).
HTTPD031E LISTEN() FAILED, RC=*rc* **ERRNO=** *errno*	The server could not put the socket into the listen state. **System action.** The start ends with the return code 8.
HTTPD032E SESSION_MAXAGE (*minutes* **MIN) <= CLIENT_TIMEOUT (***seconds* **SEC), RAISE IT**	The maximum age of a session (in minutes) is not longer than the client timeout (in seconds). The message is written after the listener is bound, and only if the maximum age is not 0. **System action.** Although the letter is E, this is a warning. The server goes on. **Response.** Raise **SESSION_MAXAGE=** in the configuration.
HTTPD033E UNABLE TO CREATE SOCKET THREAD	The task for the listener could not be created. **System action.** The server closes the listener. The start ends with the return code 8.
HTTPD034W UNABLE TO CREATE WORKER THREADS, DYNAMIC DOCUMENTS DISABLED	The pool of workers could not be created. Dynamic documents (the routes that run programs) are disabled. **System action.** The server comes up anyway.
HTTPD035W UNABLE TO REGISTER MODULE *module* **FOR** *path*	The server could not store a **MOD=** route. The path is served from the document root, with nothing that gates it. **System action.** If the route asked for a gate, **HTTPD419E** and **HTTPD420E** follow and the server does not start.
HTTPD036I MODULE *module* **REGISTERED FOR** *path*	A **MOD=** route is registered. The module name is in upper case and has up to 8 characters. The path is as written in the configuration. If the line has no path, the path is ***.** and the program name in lower case.
HTTPD037E HTTPD IS ALREADY ACTIVE ON PORT *port***, THIS INSTANCE ENDS**	Another HTTPD holds the lock for the port. The lock is the name **HTTPD.PORT.nnnnn** (the port with 5 digits). A second server on another port is allowed. **System action.** This instance ends with the return code 8.

	Response. Stop the other server, or use another port.
HTTPD038E READ ERROR ON DD:*ddname* **-- CONFIGURATION INCOMPLETE**	The server opened the configuration and could not read it to the end. **System action.** The start ends with the return code 8. **Response.** Check the data set of the DD name **HTTPPRM** and start again.
HTTPD039W WRITE ERROR ON DD:HTTPDBG -- DEBUG OUTPUT SWITCHED OFF	A write of the trace failed, for example because the data set of the DD name **HTTPDBG** is full. The server writes the message once in a run, for the first failed write, from whichever thread meets it. The text names **HTTPDBG** also when the trace goes to the standard error file (**HTTPD021I**). The keyword **DEBUG=** must be greater than 0 for the message to appear. **System action.** The server switches the trace off and goes on. It writes no more trace lines until the next start. The trace ends at the point of the failure. **Response.** Give the DD name **HTTPDBG** a data set with more room, and restart the server to trace again.
HTTPD040I WAITING FOR SOCKET THREAD TO TERMINATE (*n***)**	Written during the stop while the socket thread still runs. *n* counts from 1 to 9. The server waits one second between the messages.
HTTPD041I FORCE DETACHING SOCKET THREAD	The socket thread did not end in time. The server detaches the task.
HTTPD042W SECURITY SYSTEM ACCEPTS ANY USERID, ALL LOGINS REFUSED	From HTTPD 4.5.1. At the start, after the identity of the started task is set, the server logs on once with the user ID **HTTPPRB** and a password that cannot be right. The security product returned an ACEE for it, so it accepts any user ID. RAKF 1.2.x does this when its user table is missing or invalid, and the console then also shows **RAKF0001** and **RAKF0002**, and **RAKF0003** and **RAKF0009** if the profile table is missing too. A product that checks gives no ACEE, and RAKF writes **RAKF0004** for the user ID **HTTPPRB** instead. **System action.** The start goes on. Every Basic, form and token login is refused, with the same answer as for a wrong password, and the server writes no message for it. Routes that need no login are served as before. The server does not test again. **Response.** Correct the tables of the security product and restart HTTPD. See 'A Security System That Accepts Any User ID' in the *HTTPD User's Guide*. The server does not run the test if **HTTPD003W** skipped the logon of the task.

HTTPD044W UNABLE TO INITIALIZE FILE SYSTEM	The file system is enabled and the server could not create its handle to it. The server has no handle for UFSD. **System action.** The server goes on. The sessions of the clients to the file system are created only if the handle exists.
HTTPD045E UFS READ ERROR RC=*rc***, RESPONSE ABORTED:** *path*, *count* **SUPPRESSED**	A read of a file failed while the server sent it. *path* is the path of the request, or the path as written in the directive if the error was in a file that a server-side include pulled in. It is cut after 40 characters. *count* is the number of read errors that the server did not report since the last message of this kind, 0 if none. **System action.** The server drops the data that is not sent and closes the connection. It writes this message at most once in 60 seconds. The first error after a quiet minute is written at once, the others in the next 60 seconds are only counted, and the count stays unreported until the next error. The server goes on.
HTTPD047E UNABLE TO INITIALIZE SERVER CREDENTIAL KEY, RC=*rc*	The server could not initialize its credential key. Login does not work. **System action.** The server comes up anyway.
HTTPD048W LOGIN IS RETIRED -- USE AUTH= ON EACH MOD=/LOC= ROUTE	The configuration has the keyword **LOGIN=** with an empty value or with only **NONE** operands. The keyword has no effect. **System action.** The server goes on. **Response.** Remove the keyword and write **AUTH=** on each route (Chapter 2, “Routes and Authentication Policy” on page 9).
HTTPD048E LOGIN *operand* **IS RETIRED -- ROUTES WITHOUT AUTH= WOULD BECOME PUBLIC**	The configuration has the keyword **LOGIN=** with an operand that asked for a login. *operand* is shown as written. A route without **AUTH=** is public. **System action.** The server sets an error flag. **HTTPD420E** follows and the start ends with the return code 8. **Response.** Remove the keyword and write **AUTH=** on each route (Chapter 2, “Routes and Authentication Policy” on page 9).

Listener, Workers and Stop (050 to 099)

Table 84. Messages 050 to 099

Message	Meaning and action
HTTPD050E MAXSOCK IS ZERO, THE LISTENER IS IN AN INVALID STATE	The loop of the socket thread found no socket to wait for.

	System action. The socket thread ends. The server does not start it again. The main task still answers the console commands.
HTTPD051E SELECTEX() FAILED, RC=*rc* **ERRNO=***errno*	The wait for work on the sockets failed. The message is not written while the server stops. **System action.** The socket thread ends. The server does not accept clients any more.
HTTPD052E ACCEPT() FAILED, RC=*rc* **ERRNO=** *errno*	The server could not accept a connection. The message is not written while the server stops. **System action.** The listener goes on.
HTTPD053E UNABLE TO SET NON-BLOCKING I/O FOR SOCKET *socket*	The server could not change the mode of an accepted socket. **System action.** The server writes the message and goes on with the connection.
HTTPD054I LISTENING ON *address* **PORT** *port*	The listener is bound. The address in the text is always **ANY**: HTTPD listens on every address of the host.
HTTPD060I SHUTDOWN *thread* **TCB(***tcb***) TASK(***task***) STACKSIZE(***size***)**	A thread ends. The thread writes the message itself. *thread* is **SOCKET THREAD** or **WORKER(xxxxxx)**, where the number is the address of the work block. *tcb* and *task* are 6 hex digits each. *size* is the stack size.
HTTPD061I STARTING *thread* **TCB(***tcb***) TASK(***task***) STACKSIZE(***size***)**	A thread starts. The thread writes the message itself. The layout is the same as in **HTTPD060I**.
HTTPD062E ABEND *code* **IN WORKER(***work***) CLIENT(***client***) SOCKET(***socket***)**	A worker ended a request with an abend. *code* is the abend word (8 hex digits). The other values are the work block, the control block of the client and the socket. **System action.** The server closes the client. The worker goes on with the next work item.
HTTPD063E NO RECOVERY IN WORKER(*work***), CLIENT(***client***) SOCKET(***socket***) DROPPED**	The worker could not set up its recovery routine. It neither read nor answered the request. **System action.** The server closes the connection. The worker goes on.
HTTPD064E SOCKET THREAD IS NOT RUNNING, START ABANDONED	The socket thread is not up after its creation. **System action.** The start ends with the return code 8, after the clean up.
HTTPD065W CLIENT *a.b.c.d* **ABANDONED, NO SEND PROGRESS FOR** *seconds* **SEC,** *bytes* **BYTES SENT,** *n* **SUPPRESSED**	A client did not receive any data for the whole time of the limit (10 seconds), so the server gave up. This happens on a chunked response and on the body of a static file. *bytes* is the number of bytes that were sent. The server writes the message at most once in 60 seconds. *n* is the number of messages that it held back in the meantime. A server that stops writes nothing.

	System action. The server cuts the response and closes the connection.
HTTPD066I REQUEST TRACE ON, *records* **RECORDS,** *bytes* **BYTES**	The Parmlib keyword **TRACE=** asked for the request trace and the server allocated its table. *records* is the number of requests the table holds and *bytes* is the storage it uses. The table is read with the display module HTTPDSRV, target **TRACE**.
HTTPD067W NO STORAGE FOR REQUEST TRACE, TRACE OFF	**TRACE=** asked for the request trace, but the server could not get the storage for the table. **System action.** The server runs without the trace. Use a smaller value or a larger region.
HTTPD070E UNKNOWN CODEPAGE "*name***", USING CP037**	The value of **CODEPAGE=** is not known. The known values are **CP037**, **IBM1047** and **LEGACY**. **System action.** The server uses **CP037** and goes on.
HTTPD090E UNABLE TO INITIALIZE CONSOLE INTERFACE	The server could not set up the interface for the console commands. **System action.** The start ends with the return code 8. No stop message follows.
HTTPD098I HTTPD SHUTTING DOWN	The stop begins. It is the answer to **P HTTPD** and also the first step after a refused start.
HTTPD099I HTTPD SHUTDOWN COMPLETE	The last message of the stop.

Console Commands (100 to 199)

The messages of this range are the output of the console commands (Chapter 3, “Console Commands” on page 17). Most of them are the answer of a command and have no response.

Table 85. Messages 100 to 119

Message	Meaning and action
HTTPD100I CONS(*console***) "***command***"**	The server echoes each command that it gets with **MODIFY**. *console* is the identification of the console. *command* is the text as the operator typed it.
HTTPD101E UNKNOWN CONSOLE COMMAND "*word* **"**	The first word of the command is not known. The help (**HTTPD150I**, then **HTTPD151I**) follows. A word may be shortened to any leading part. The first match in the list of the help wins. **Response.** Use one of the commands in the help.
HTTPD101E UNKNOWN CONSOLE COMMAND "DISPLAY *word***"**	The word after **DISPLAY** is not known. There is no help.
HTTPD101E UNKNOWN CONSOLE COMMAND "SET *word***"**	The word after **SET** is not known. There is no help.
HTTPD102I HTTPD IS LISTENING ON PORT *port*	The answer of **DISPLAY PORTS**.

HTTPD103I ..THREAD *address role*	The first line of a thread in the answer of **DISPLAY THREADS**. *role* is **MAIN SERVER THREAD**, **SOCKET THREAD**, **DISPATCHER THREAD** or **WORKER THREAD**.
HTTPD104I .....EYE *eye* **.....TCB** *tcb* **..OWNTCB** *owntcb*	The task block of a thread: the eye catcher, *tcb* and *owntcb*, in hex.
HTTPD105I .TERMECB *ecb* **......RC** *rc* **STACKSZE** *size*	The task block, second line: *ecb*, the return code and the stack size, in hex.
HTTPD106I ....FUNC *func* **....ARG1** *arg1* **....ARG2** *arg2*	The task block, third line: the function that the thread runs and its two arguments, in hex.
HTTPD107I .....EYE *eye* **....TASK** *task* **.WAITECB** *ecb*	The manager of the pool of workers: eye catcher, task and wait ECB. It is shown if the manager exists.
HTTPD108I ....FUNC *func* **...UDATA** *udata* **STACKSZE** *size*	The manager, second line: the function, the user data and the stack size, in hex.
HTTPD109I ..WORKER *worker* **...QUEUE** *queue* **...STATE** *state name*	The manager, third line: the worker array and the queue, the state in hex, then the name of the state: **INIT**, **RUNNING**, **QUIESCE**, **STOPPED**, **WAITING** or **unknown**.
HTTPD110I .MINTASK *mintask* **.MAXTASK** *maxtask*	The limits of the pool of workers, in decimal. **SET MINTASK** and **SET MAXTASK** answer with this line.
HTTPD111I .WORKERS *workers* **.DISPCNT** *count*	The number of workers now and the number of dispatches, in decimal.
HTTPD112I ..WORKER *worker* **.....EYE** *eye* **.WAITECB** *ecb*	A worker: the block, the eye catcher and the wait ECB.
HTTPD113I .....MGR *mgr* **....TASK** *task* **...QUEUE** *queue*	A worker, second line: its manager, its task and its queue.
HTTPD114I ...STATE *state name*	A worker, state: in hex and by name. The names are **INIT**, **RUNNING**, **WAITING**, **DISPATCHED**, **SHUTDOWN**, **STOPPED** and **unknown**.
HTTPD115I ...START *time text*	The start time of the worker: 64 bits in hex, then the date and time as text. The text is empty if the time is 0.
HTTPD116I ....WAIT *time text*	The time of the last wait of the worker. The layout is the same as in **HTTPD115I**.
HTTPD117I ....DISP *time text*	The time of the last dispatch of the worker. The layout is the same as in **HTTPD115I**.
HTTPD118I PROTOCOL HTTP ....USER *user* **...GROUP** *group*	The client of a running worker. *user* is **ANONYMOUS** and the group is empty if the client has no credential.
HTTPD119I ..REMOTE CLIENT ....PORT *port* **......IP** *address*	The remote side of the client of a worker: port and address of the client.
HTTPD119I ..SOCKET *socket* **....PORT** *port* **(HTTP PROTOCOL)**	The listener in the answer of **DISPLAY THREADS**: the socket and the port.

Table 86. Messages 120 to 149

Message	Meaning and action

HTTPD120I ..SOCKET *socket* **....PORT** *port* **......IP** *address*	The socket of the client of a running worker, with the port and the address of the peer.
HTTPD120I ...RNAME *rname*	The resource name of the server: **HTTPD.** and the 4 hex digits of the address space number. It is shown for the main thread.
HTTPD121I USERS LOGGED IN: *n*	The first line of the answer of **DISPLAY LOGIN**. One **HTTPD122I** for each credential follows.
HTTPD122I USER *userid* **IP** *a.b.c.d* **ACEE(** *acee***)**	A logged in user: the user, the address of the client and the address of the security environment.
HTTPD123I SMF: *level* **(TYPE** *type***)**	The first line of the answer of **DISPLAY STATS**: the level and the record type of SMF (Chapter 8, "SMF Records" on page 93).
HTTPD124I REQUESTS: *requests* **ERRORS:** *errors* **BYTES:** *bytes* **ACTIVE:** *active*	The counters of the server: requests, errors, bytes and active clients.
HTTPD125I STATISTICS COUNTERS RESET	The counters of requests, errors and bytes are zero. It is the answer of **SET STATS RESET**, and also follows a level that has the word **RESET** after it.
HTTPD126I SMF LEVEL SET TO *level*	The answer of **SET STATS** with a level.
HTTPD126E INVALID SET STATS VALUE " *operand***"**	The operand of **SET STATS** is not **NONE**, **ERROR**, **AUTH**, **ALL** or **RESET**. The level does not change. **Response.** Give a valid operand.
HTTPD127W MISSING STATS NONE|ERROR|AUTH|ALL [RESET]	**SET STATS** has no operand. **Response.** Give a level.
HTTPD128I PORT *port* **MINTASK** *mintask* **MAXTASK** *maxtask*	The first line of the answer of **DISPLAY CONFIG**: port and limits of the pool of workers.
HTTPD129I CLIENT_TIMEOUT *seconds* **KEEPALIVE** *timeout***/***max*	The client timeout in seconds, the keep-alive timeout and the maximum number of requests on a connection.
HTTPD130I SESSION_TIMEOUT *minutes* **MIN** *text*	The session timeout in minutes. The text **(REAPER DISABLED)** follows if the value is 0.
HTTPD131I DOCROOT *path*	The document root, or **(NONE)**.
HTTPD132I SMF TYPE *type* **LEVEL** *level*	The record type and the level of SMF (Chapter 8, "SMF Records" on page 93). The level is **NONE**, **ERROR**, **AUTH** or **ALL**.
HTTPD133I CONFIG FROM *source*	Where the configuration came from: the data set name with the member, or **DD:HTTPPRM**.
HTTPD134I CODEPAGE *name*	The code page in use. It is **CP037** if none is set.
HTTPD135I SESSION_MAXAGE *minutes* **MIN** *text*	The maximum age of a session in minutes. The text **(NO MAX-AGE)** follows if the value is 0.
HTTPD136I REALM *realm*	The realm of Basic authentication.
HTTPD137I RESIDENT *yesno* **MODULES HELD** *n*	Whether the server keeps the modules in storage (**YES** or **NO**) and how many it holds now.
HTTPD138I RES_NOPROFILE *value*	The value of the keyword **RES_NOPROFILE**: **ALLOW** or **DENY**. It is the last line of **DISPLAY CONFIG**.

HTTPD140I HTTPD *version* **(***commit***)**	The answer of **DISPLAY VERSION**, in upper case.
HTTPD141I .DISPCNT *count*	The number of dispatches of a worker, in the answer of **DISPLAY THREADS**.
HTTPD142I TIME *yyyy/mm/dd hh:mm:ss* **GMT**	The answer of **DISPLAY TIME**, the first line: the time in GMT.
HTTPD143I TIME *yyyy/mm/dd hh:mm:ss* **LOCAL**	The second line: the local time. The server adds the text **OFFSET=** and the sign and the minutes of the offset. An argument in minutes sets the offset (from −720 to +720). Without an argument, the offset of the system applies.
HTTPD144E INVALID MEMORY ADDRESS 0x *address*	The address of **DISPLAY MEMORY** is above 16 MB. **Response.** Give an address below 16 MB.
HTTPD145I END OF MEMORY DUMP FOR 0x *address*	The dump of **DISPLAY MEMORY** is complete. The lines of the dump have no number (“Lines Without a Number” on page 118).
HTTPD146E ESTAE CREATE FAILURE, RC=0x*rc*	The server could not set up the recovery for **DISPLAY MEMORY**. The dump is not made.
HTTPD147E ABEND S*code* **FOR DISPLAY MEMORY 0x***address*	The read of the storage for **DISPLAY MEMORY** ended with a system abend. *code* has 3 hex digits.
HTTPD147E ABEND U*code* **FOR DISPLAY MEMORY 0x***address*	The read of the storage for **DISPLAY MEMORY** ended with a user abend. *code* is in decimal.
HTTPD148I USAGE: DISPLAY MEMORY XXXXXX[,NNN] (D M XXXXXX)	**DISPLAY MEMORY** has no address. The default length of a dump is 256 bytes and the maximum is 4096.
HTTPD149I USAGE: SET MAXTASK|MINTASK N (S MA N, S MI N), N = 1 TO 100	**SET MAXTASK** or **SET MINTASK** has no number, or a negative number. The limits do not change, and the dispatcher is not woken. **Response.** Give a decimal number.

Table 87. Messages 150 to 199

Message	Meaning and action
HTTPD150I MODIFY COMMANDS ARE:	The header of the help. The command **?** or **HELP** and an unknown command show it.
HTTPD151I *line*	A line of the help. The help lists **DISPLAY** with the operands **CONFIG**, **LOGIN**, **MEMORY**, **PORTS**, **STATS**, **THREADS**, **TIME** and **VERSION**, and **SET** with **MAXTASK**, **MINTASK** and **STATS**. Each command has a line with the command and its short form and a line with a description. The last line is the example **/F HTTPD,D C**.
HTTPD199I **----------**	A separator line. It closes each block of the answer of **DISPLAY THREADS**, and the answer of **SET MAXTASK** and **SET MINTASK**.

Routes, SMF and Resident Modules (410 to 444)

The routes are described in Chapter 2, “Routes and Authentication Policy” on page 9 and the keywords in Chapter 1, “The HTTPPRM Keywords” on page 1. Several messages of this range say that a route could not be registered. A route that is lost does not remove its path: the server serves the path from the document root, with nothing that gates it. Therefore a lost route that asked for a gate (**AUTH=** other than **NONE**, or **RES=**) is an error. The server sets an error flag, writes **HTTPD419E** and refuses to start with **HTTPD420E**.

Table 88. Messages 410 to 444

Message	Meaning and action
HTTPD410W CGI= IS DEPRECATED, USE MOD= INSTEAD	The configuration has the old keyword **CGI=**. The server still honors it. **Response.** Write **MOD=** instead.
HTTPD411E UNKNOWN AUTH MODE '*mode***' -- USE NONE, FORM, BASIC OR TOKEN**	A route has an **AUTH=** value that HTTPD does not know. The value is cut after 24 characters. **System action.** The server refuses the route. **HTTPD419E** and **HTTPD420E** follow, and the start ends with the return code 8. **Response.** Write **NONE**, **FORM**, **BASIC** or **TOKEN**.
HTTPD412W IGNORING MALFORMED RES= '*text* **' (NEED CLASS:RESOURCE)**	The value of **RES=** of a route is not in the form **class:resource**. **System action.** The server ignores the option and goes on.
HTTPD413W IGNORING UNKNOWN ROUTE OPTION '*token***'**	A route has a word that is not an option. The old option **RECLAIM=** gets **HTTPD422W** instead. **System action.** The server ignores the word and goes on.
HTTPD414W AUTH=NONE IGNORES RES=*class***:** *resource* **(PUBLIC ROUTE)**	A route has **AUTH=NONE** and also **RES=**. **NONE** wins and the route is public. A **RES=** without **AUTH=** means **BASIC**. **System action.** The server goes on.
HTTPD415W LOC= REQUIRES A PATH (E.G. LOC /ADMIN/* AUTH=BASIC)	A **LOC=** line is empty, or it has an option where the path belongs. **System action.** The server drops the line. If the line asked for a gate, **HTTPD419E** and **HTTPD420E** follow.
HTTPD416I STATS: *requests* **REQUESTS,** *errors* **ERRORS,** *bytes* **BYTES**	The counters of the run, written at the stop before **HTTPD099I**.
HTTPD417I LOCATION *path* **REGISTERED**	A **LOC=** route is registered.
HTTPD418E NO STORAGE FOR RES=*class***:** *resource*	The server had no storage for the **RES=** option of a route. The class is cut after 16 characters and the resource after 40.

	System action. The server refuses the route. **HTTPD419E** and **HTTPD420E** follow, and the start ends with the return code 8.
HTTPD419E *kind*=*path* **COULD NOT BE REGISTERED -- ITS AUTH POLICY IS LOST**	A route that asked for a gate could not be registered. *kind* is **MOD** or **LOC**. *path* is the path, or the line as written if the line could not be split, cut after 40 characters. The causes are no storage for the route or for **RES=**, a line that cannot be split, a missing path or program name, and an unknown **AUTH=** value. **System action.** The server sets an error flag. **HTTPD420E** follows. **Response.** Correct the line in the configuration.
HTTPD420E ROUTE AUTHORIZATION POLICY INCOMPLETE -- HTTPD WILL NOT START	At least one route lost its policy (**HTTPD419E**, or **HTTPD048E**), or the value of **RES_NOPROFILE** is not valid (**HTTPD427E**). The message is written at the end of the configuration, before the bind. Its text is the same for every cause. The cause is in the message before it. **System action.** The start ends with the return code 8. **Response.** Correct the lines that the other messages name.
HTTPD421W MOD= REQUIRES A PROGRAM NAME (E.G. MOD=MVSMF /ZOSMF/* AUTH=BASIC)	A **MOD=** line is empty, or it has an option where the program name belongs. **System action.** The server drops the line. If the line asked for a gate, **HTTPD419E** and **HTTPD420E** follow.
HTTPD422W *keyword* **IS RETIRED -- CGI STORAGE RECLAIM IS ALWAYS ON**	A route has the option **RECLAIM=**. The keyword is shown as written. The option has no effect. **System action.** The server goes on.
HTTPD423W UNABLE TO REGISTER LOCATION *path*	The server could not store a **LOC=** route. The prefix is served from the document root, with nothing that gates it. **System action.** If the route asked for a gate, **HTTPD419E** and **HTTPD420E** follow and the server does not start.
HTTPD424W INVALID REALM VALUE: *value*	The value of **REALM=** is not valid. The value is cut after 40 characters. **System action.** The server keeps the default realm (the SMF identification) and goes on.
HTTPD425W NO PROFILE FOR *class*:*resource* **--** *path* **NOT GATED**	A route with **RES=** (not **AUTH=NONE**) names a resource that the security product does not protect. The check at the start found no profile. The class is cut after 8 characters, the resource after 32 and the path after 24.

	System action. The route is served, without a gate. The server goes on. **Response.** Define the profile, or correct the name in the configuration. To refuse the requests instead, set **RES_NOPROFILE=DENY** (then **HTTPD426W** replaces this message).
HTTPD426W NO PROFILE FOR *class*:*resource* **--** *path* **REFUSES ALL REQUESTS**	As **HTTPD425W**, for a server that runs with **RES_NOPROFILE=DENY**. The check at the start found no profile for the resource of a route with **RES=** (not **AUTH=NONE**). The class is cut after 8 characters, the resource after 32 and the path after 24. **System action.** The server goes on. The route answers every request that reaches the check with status 403, until a profile exists. A profile that is defined and loaded later takes effect without a restart, but this message is written at the start only. **Response.** Define the profile, or correct the name in the configuration.
HTTPD427E INVALID RES_NOPROFILE VALUE " *value***" (ALLOW|DENY)**	The value of **RES_NOPROFILE** is neither **ALLOW** nor **DENY**. The value is cut after 16 characters. **System action.** The server sets an error flag. It does not fall back to **ALLOW**. **HTTPD420E** follows and the start ends with the return code 8. **Response.** Write **ALLOW** or **DENY**.
HTTPD430I SMF TYPE *type* **LEVEL** *level*	Not issued in this release. The text is defined, and no code that is built writes it. **DISPLAY CONFIG** shows the same values in **HTTPD132I**.
HTTPD430W SMF TYPE *type* **CONFIGURED BUT SMF INACTIVE**	The level of SMF is above **NONE**, but SMF is not active on the system. **System action.** The server goes on. **Response.** Start SMF, or set the level to **NONE**.
HTTPD431W INVALID SMF LEVEL "*value***"**	The value of **SMF=** is not a level. **System action.** The server keeps the earlier level and goes on.
HTTPD432W INVALID SMF TYPE=*type* **(128-255)**	The record type is not in the range from 128 to 255. **System action.** The server keeps the earlier type and goes on.
HTTPD440W MODULE *module* **NOT FOUND AT STARTUP, NOT HELD**	With **RESIDENT=YES**, the server tried to load a **MOD=** module at the start and did not find it. The name is in upper case, up to 8 characters. **System action.** The server does not hold the module. It fetches it for each request.
HTTPD441W MODULE *module* **NOT REUSABLE, FETCHED PER REQUEST**	The module is not marked reusable, so the server cannot keep it.

	System action. The server fetches it for each request. The route works.
HTTPD442W MODULE *module* **LOAD FAILED, ABEND CODE S**_code_ **REASON** *reason*, **FETCHED PER REQUEST**	The load of the module at the start failed with a system abend. The codes in the source comment are 806 (not found), 106 (fetch error) and 80A (no storage). **System action.** The server fetches the module for each request. The route works.
HTTPD443W INVALID RESIDENT VALUE "*value* **" (YES|NO)**	The value of **RESIDENT=** is not **YES** or **NO**. **System action.** The default **YES** stays.
HTTPD444W INVALID TRACE VALUE "*value***" (0 OR 16-1024), TRACE OFF**	The value of **TRACE=** is not a number, or it is not 0 and not in the range from 16 to 1024. **System action.** The request trace stays off.

Diagnostics (901 to 999)

The header says that a message of this range means the server met a state that its code did not expect, and that each one is worth a bug report.

Table 89. Messages 901 to 999

Message	Meaning and action
HTTPD901E SPIN IN SERVE_CLIENT: CLIENT(*client***) STATE(***state***) SOCKET(***socket***) REQ(***count***) WORKER(***work***) -- FORCING CLOSE**	The processing of a client went around 1000 times without a change of state. **System action.** The server clears the busy entry and closes the client.
HTTPD904E NO STORAGE FOR ENVIRONMENT VARIABLE *name* **CLIENT(***client***)**	The server could not store an environment variable for a request. The name is cut after 32 characters. **System action.** The set of the variable fails.
HTTPD905E ENVIRONMENT VARIABLE *name* **TOO LARGE CLIENT(***client***)**	The value of an environment variable does not fit. **System action.** The set of the variable fails.
HTTPD906W STORAGE RECLAIM FOR *module* **FAILED, RC=***rc*	After a module ran, the server could not free the storage of the module. The storage stays allocated until the server restarts. **System action.** The server goes on.
HTTPD907W STALE BUSY ENTRY FOR CLIENT(*client***) STATE(***state***) SOCKET(***socket***) REQ(***count***) -- CLEARED**	A client was in the list of busy clients when the processing found no progress. **System action.** The server clears the entry and goes on.
HTTPD908E EXTERNAL PROGRAM *module reason*	A program that a route runs failed. *reason* is one of: “was not found”, “could not be loaded”, “was dispatched without a program name”, “could not be started, ESTAE create failed”, “returned rc n”, “failed with Sxxx ABEND” or “failed with Unnnn ABEND”.

	The name is “(none)” if there is none. “failed with U0801 ABEND”, with **@@CRT0 - No storage for C stack** before it on the console, means that the region had no free block for the stack of the module (see “The region” on page 28). **System action.** The server frees the storage of the program (**HTTPD906W** if that fails). If no response header went out, the client gets the status 503 with the same text. The server goes on.
HTTPD909E UNSUPPORTED HTTP STATUS *code* **REQUESTED, SENT 500**	A module asked for a status that HTTPD does not know. **System action.** The server sends the status 500.
HTTPD910E ABEND S*code* **DETECTED**	Not issued in this release. The text is defined, and no code that is built writes it.
HTTPD910E ABEND U*code* **DETECTED**	Not issued in this release. The text is defined, and no code that is built writes it.
HTTPD911E *function* **LOCK FAILURE**	The lock of the server block failed. *function* is the name of the function, in lower case. **System action.** The function skips its work. The socket thread goes on.
HTTPD912E TERMINATE FAILED WITH RC=*rc*	The stop routine ended with an abend or an error. *rc* is the value that it returned, in 8 hex digits. **System action.** The server goes on to end. **HTTPD416I** and **HTTPD099I** may be missing.
HTTPD913E UNEXPECTED SUBTYPE *n* **IN HTTP_GET()**	The processing of a GET found a type of request that it does not handle. **System action.** The client gets the status 500.
HTTPD914E HTTP_SEND() POSITION UNDERFLOW *pos*	The send loop found a position before the start of the buffer. **System action.** The loop stops.
HTTPD915E *function* **LENGTH INCONSISTENCY, LEN=***len* **BUFLEN=***buflen*	The formatting of text for the response failed. *function* is **ssi_printv**. **System action.** The function fails.
HTTPD916E HTTP_SET_BUSY() FAILED	The server could not mark a client as busy. **System action.** The server does not serve the client in this pass.
HTTPD920E *name* **MUST BE CALLED BY THE HTTPD SERVER**	A display module (**HTTPDM**, **HTTPDMTT** or **HTTPDSRV**) was run from TSO or from a batch job (Chapter 7, “The Display Modules” on page 81). The same sentence goes to the standard output. **System action.** The module ends with the return code 12.

	Response. Call the module through its route of the server.
HTTPD921W ABEND0C1 ALLOCATED *kb* **KB IN** *blocks* **BLOCKS, ABENDING WITHOUT FREEING**	The test module **ABEND0C1** allocated storage and now ends with an S0C1 abend on purpose. **HTTPD906W** and **HTTPD908E** follow.
HTTPD922E HTTPD CONTROL BLOCK NOT FOUND BY *name*	A module could not find the control block of the server. A trace back follows.
HTTPD923E HTTPC CONTROL BLOCK NOT FOUND BY *name*	Not issued in this release. The text is defined, and no code that is built writes it. Its only call is compiled out.
HTTPD999E OUT OF MEMORY	The server had no storage for the control block of a new client. **System action.** The server closes the accepted socket. The listener goes on.

Lines Without a Number

A few console lines have no message number.

Client timeout — A client sent part of a request and then went silent. The line is **client xxxxxxxx on socket n timed out after n seconds**, where the first value is the address of the control block of the client and the last is the keep-alive timeout if the connection already served a request, or else the client timeout. A second output is a dump of the buffer with the title **Receive Buffer**. Both are on by default. The keywords **CLIENT_TIMEOUT_MSG=0** and **CLIENT_TIMEOUT_DUMP=0** switch them off (Chapter 1, "The HTTPPRM Keywords" on page 1). The connection is closed.

Memory dump — The lines of **DISPLAY MEMORY** between the command and **HTTPD145I**. The title is **DISPLAY MEMORY**.

Offset — The text **OFFSET=** at the end of **HTTPD143I**.

Messages That Are Gone

The numbers **HTTPD900D**, **HTTPD902D** and **HTTPD903D** were debug counters and are no longer defined. HTTPD uses no message letter other than I, W and E.

The texts of **HTTPD010I**, **HTTPD011I**, **HTTPD013I**, **HTTPD430I**, both texts of **HTTPD910E** and **HTTPD923E** are defined in the header and no code that is built writes them. The function that would write the first three is never called. The text of **HTTPD012E** is written, but from a different place.

Index

/

/.abend 92
/.dm 90
/.dmtt 91
/.dsrv 82
/login 11
/logout 11

1

100 Continue 47

4

400 Bad Request 47
401 12
401 Unauthorized 47
403 12
403 Forbidden 47
404 Not Found 47
405 Method Not Allowed 43
413 Payload Too Large 46
414 URI Too Long 45

5

500 Internal Server Error 47
501 Not Implemented 43
503 16
503 Service Unavailable 47
505 HTTP Version Not Supported 47

_

__getmsp 76
__wsaget 76

A

abend
  module 77
  test module 92
abend codes 37
ABEND0C1 41, 81, 92
accounting 93
ACEE
  started task 33
APF authorization 32
append-only rule 65
argv
  module 59
array_add 71
array_new 71
AUTH= 9
  modes 11
authentication
  module 72
  route policy 11
authorization
  library or SVC 32

B

BIND_SLEEP 2
BIND_TRIES 2, 37

C

C stack 28
Cache-Control 49
CANCEL command 25
CC 0008 7
CGI_CONTEXT_POINTERS 5, 79
cgistart 59
chunked transfer 52
CLIENT_STATS 2
CLIENT_TIMEOUT 2, 50
CLIENT_TIMEOUT_DUMP 2, 51
CLIENT_TIMEOUT_MSG 2, 51
code page 57, 70
CODEPAGE 3, 57
comments
  HTTPPRM 1
completion codes 37
CONFIG
  parameter 7
configuration
  display 18
Connection header 50
console commands
  reference 17
Content-Length 46
Content-Type 49
  static files 53
context pointers 79
counters
  statistics 58
CSTATE 61

D

D CONFIG 8
D STATS 58
data sets
  package 40
Date header 49
DD names 29
DEBUG 6
  module output 72
default.html 52
DELETE 43
DEV library 32
diagnostics
  display modules 81

DISPLAY CONFIG 18
DISPLAY LOGIN 19
DISPLAY MEMORY 21
display modules 81
  authentication 81
  route 81
DISPLAY STATS 19
DISPLAY THREADS 19
DISPLAY TIME 22
DISPLAY VERSION 22
DOCROOT 3, 52
document root 52
drain 77
duplicate keys 1
dynamic array 71

E

ENQ
  port 37
environment variable
  size limit 46
environment variables
  request 63
ESTAE 77
Expect: 100-continue 46
extension routing 11

F

file extension 53
first match 10
FMID 39
form login
  methods 48
function vector 65

G

GET 43

H

HASPACE1 29, 78
HASPCKPT 29, 78
HEAD 43
  static file 44
header
  length limit 45
  request 46
HELP command 23
Host header 46
HTTP
  behaviour 43
http_add_route 74
http_atoe 70
http_cgictx_get 74
  context block 79
http_check_auth 72
http_cmp 72
http_cmpn 72
http_date_rfc1123 70
http_dbgf 72
http_decode 69
http_etoa 70
http_find_env 69
http_find_route 74
http_get_env 69
http_get_flag 75
http_get_httpx 75
http_get_password 72
http_get_token 72
http_get_ufs 71
http_get_userid 72
http_link
  result codes 75
HTTP_LINK_ENOMOD 75
HTTP_LINK_EPGMRC 75
http_logout 72
http_mime 70
http_new_session 73
http_ntoa 70
http_open 71
http_printf 68
http_realm 72
http_resp 68
http_resp_internal_error 69
http_resp_not_found 69
http_resp_not_implemented 69
http_secs 70
http_send 68
http_send_binary 71
http_send_file 71
http_send_text 71
http_set_env 69
http_xlate 70
HTTPC 59
  fields 61
httpcgi.h 59
HTTPD
  control block 59
HTTPD procedure 27
httpd-webroot.img 41
HTTPD.AHTTPLOD 40
HTTPD.DEV.LINKLIB 32
HTTPD.HTTPLOAD 40
HTTPD.LINKLIB 32, 40
HTTPD.SAMPLIB 40
HTTPD.WEBROOT.UFS 40
HTTPD000I 101
HTTPD001I 35, 101
HTTPD002I 32, 101
HTTPD003W 33, 101
HTTPD004I 33, 101
HTTPD004W 33, 101
HTTPD005I 101
HTTPD006W 101
HTTPD007I 101
HTTPD008I 102
HTTPD008W 102
HTTPD009W 102
HTTPD010I 102
HTTPD011I 102
HTTPD012E 102
HTTPD013I 102
HTTPD014E 102
HTTPD015E 102
HTTPD016E 102

HTTPD017W 102
HTTPD018W 103
HTTPD019W 103
HTTPD020E 103
HTTPD020W 1, 103
HTTPD021I 103
HTTPD023E 2, 103
HTTPD024I 104
HTTPD024W 7, 104
HTTPD025I 104
HTTPD025W 6, 104
HTTPD026E 4, 104
HTTPD026W 1, 104
HTTPD027I 104
HTTPD028E 104
HTTPD029W 1, 104
HTTPD030E 104
HTTPD030I 105
HTTPD031E 105
HTTPD032E 4, 105
HTTPD033E 105
HTTPD034W 105
HTTPD035W 105
HTTPD036I 105
HTTPD037E 37, 105
HTTPD038E 106
HTTPD039W 106
HTTPD040I 106
HTTPD041I 106
HTTPD042W 106
HTTPD044W 3, 107
HTTPD045E 51, 107
HTTPD047E 107
HTTPD048E 6, 107
HTTPD048W 6, 107
HTTPD050E 107
HTTPD051E 108
HTTPD052E 108
HTTPD053E 108
HTTPD054I 37, 108
HTTPD060I 108
HTTPD061I 108
HTTPD062E 77, 108
HTTPD063E 77, 108
HTTPD064E 108
HTTPD065W 51, 108
HTTPD066I 6, 109
HTTPD067W 6, 109
HTTPD070E 3, 109
HTTPD090E 109
HTTPD098I 109
HTTPD099I 109
HTTPD100I 109
HTTPD101E 109
HTTPD102I 109
HTTPD103I 110
HTTPD104I 110
HTTPD105I 110
HTTPD106I 110
HTTPD107I 110
HTTPD108I 110
HTTPD109I 110
HTTPD110I 110
HTTPD111I 110
HTTPD112I 110
HTTPD113I 110
HTTPD114I 110
HTTPD115I 110
HTTPD116I 110
HTTPD117I 110
HTTPD118I 110
HTTPD119I 110
HTTPD120I 111
HTTPD121I 111
HTTPD122I 111
HTTPD123I 111
HTTPD124I 111
HTTPD125I 111
HTTPD126E 111
HTTPD126I 111
HTTPD127W 111
HTTPD128I 111
HTTPD129I 111
HTTPD130I 111
HTTPD131I 111
HTTPD132I 111
HTTPD133I 111
HTTPD134I 111
HTTPD135I 111
HTTPD136I 111
HTTPD137I 111
HTTPD138I 111
HTTPD140I 112
HTTPD141I 112
HTTPD142I 112
HTTPD143I 112
HTTPD144E 112
HTTPD145I 112
HTTPD146E 112
HTTPD147E 112
HTTPD148I 112
HTTPD149I 112
HTTPD150I 112
HTTPD151I 112
HTTPD199I 112
HTTPD410W 113
HTTPD411E 11, 113
HTTPD412W 13, 113
HTTPD413W 113
HTTPD414W 113
HTTPD415W 113
HTTPD416I 113
HTTPD417I 113
HTTPD418E 13, 113
HTTPD419E 14, 114
HTTPD420E 14, 114
HTTPD421W 14, 114
HTTPD422W 6, 114
HTTPD423W 114
HTTPD424W 4, 114
HTTPD425W 13, 34, 114
HTTPD426W 4, 13, 34, 115
HTTPD427E 4, 115
HTTPD430I 115

HTTPD430W 115
HTTPD431W 6, 115
HTTPD432W 6, 115
HTTPD440W 15, 79, 115
HTTPD441W 15, 79, 115
HTTPD442W 15, 79, 116
HTTPD443W 5, 79, 116
HTTPD444W 6, 116
HTTPD901E 116
HTTPD904E 116
HTTPD905E 46, 116
HTTPD906W 16, 116
HTTPD907W 116
HTTPD908E 16, 77, 116
HTTPD909E 48, 117
HTTPD910E 117
HTTPD911E 117
HTTPD912E 117
HTTPD913E 117
HTTPD914E 117
HTTPD915E 117
HTTPD916E 117
HTTPD920E 117
HTTPD921W 118
HTTPD922E 118
HTTPD923E 118
HTTPD999E 118
HTTPD_FLAG_QUIESCE 75
HTTPD_FLAG_SHUTDOWN 75
HTTPDBG 29, 72
HTTPDERR 29
HTTPDIN 29
HTTPDM 41, 81
  parameters 90
HTTPDMTT 41, 81
  parameters 91
HTTPDOUT 29
HTTPDSRV 41, 81
  targets 82
HTTPM 61
HTTPPRM
  DD name 29
  keywords 1
  line format 1
HTTPPRM0 41
HTTPSTAT
  DD name 29
HTTPV 61
HTTPWEBR 41
HTTPX 59
  append-only 65
  function vector 65
  levels 65

I

index.html 52

J

JES2 data sets 31
jesopen 31, 78

K

keep-alive 50
KEEPALIVE_MAX 2, 50
KEEPALIVE_TIMEOUT 2, 50

L

limits
  hard 58
  HTTP 43
  routes 13
LISTEN_QUEUE 2
listener 37
load library 32
load modules 41
LOC 5
  static routes 15
  syntax 9
logged in users 19
LOGIN
  retired 6

M

master trace table 91
MAXTASK 2, 57
  and REGION 29
memory dump 90
messages
  console 99
  format 99
  length 99
  list 99
  master trace table 99
  retired 118
  return code 100
  start order 99
  start refused 100
  stop order 99
  unnumbered 118
  unnumbered lines 99
methods 43
MIME types 53
MINTASK 2, 57
MOD 5
  syntax 9
MODIFY command 17
module
  interface 59
  storage 76
module load failure 16
mvsMF
  jobs API and HASPCKPT 31

O

operator commands 17

P

package 39
  SMP 27
parameters
  limit 46
PARM keywords 28
Parmlib

configuration member 1
path traversal 52
pattern
route 10
percent encoding 47
PORT 2
POST 43
POST_STRING 63
PROC
module requirements 78
procedure
started task 27
PUT 43

Q

query string 46
decoding 47
QUERY_STRING 63
quiesce
module 77

R

RACF
resource check 13
RACINIT 33
RAKF 34
RAKF0012 35
RDW 61
READY message 35
REALM 4
RECLAIM
retired 6
recv
one byte 58
REGION
size 28
started task 28
request
limits 45
processing order 43
request body
limit 46
request line
length 45
request trace 86
request variables 63
REQUEST_PATH 63
RES= 9
profile check 34
RES_NOPROFILE 4
resource check 13
RES_NOPROFILE 4
rules 4
RESIDENT 5, 79
start 15
resident modules 79
response
chunked 68
headers 68
response headers 49
retired keywords 6
return codes 37
route errors 14
route table 9
routes 9

S

SAF 34
SCRIPT_FILENAME 11
send stall 51
Server header 49
server module 59
entry 59
LINK 59
return code 60
server-side includes 55
escaping 56
server environment 56
session
login 73
SESSION_MAXAGE 4
SESSION_TIMEOUT 4
SET MAXTASK 22
SET MINTASK 22
SET STATS 23
shutdown 24
module 77
SMF
byte order 96
date 96
header 93
keyword 6, 96
level 96
record type 93
records 93
request record 94
session record 95
subtype 93
subtype 1 94
subtype 2 95
time 96
TYPE 96
type 243 93
SMF level
set 23
SMP
package 39
SNAP
DD name 29
socket thread 57
SSI 55
start refused 7
start-up sequence 35
started task 27
procedure 78
static data
module 76
static files 52
routes 15
status codes 47
phrase table 48
STC identity 33
STCGROUP 7, 33
PARM keyword 28

STCGRP 27
STCUSER 7, 33
STEPLIB 32
STOP command 24
storage dump 21
subpool 5 76
SVC 244 32
symbolic parameters 27
SYS2.PROCLIB(HTTPD) 27
SYSIN
  not allowed 30
SYSMOD 39
SYSPRINT
  not allowed 30
SYSTERM
  not allowed 30
SYSUDUMP 29

T

threads 57
  display 19
THTP 39
time zone offset 22
time-out
  client 50
  keep-alive 51
TRACE
  keyword 6
  target 86
Transfer-Encoding 52
translation
  ASCII EBCDIC 57
  EBCDIC and ASCII 70
TZOFFSET
  retired 6

U

U0801 28
UFS 3
UFS370
  webroot 41
unknown command 23
URL
  decoding 47

W

webroot image 41
wildcard 10
worker tasks 57
worker threads
  set 22

X

X-Content-Type-Options 49
X-Powered-By 49