Oracle® iPlanet Web Proxy Server 4.0.14 NSAPI Developer's Guide

Part No: 821–1886
June 2010

ORACLE®

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related software documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are “commercial computer software” or “commercial technical data” pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. UNIX is a registered trademark licensed through X/Open Company, Ltd.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

100624@24378

Contents

Preface 13

1 **Creating Custom SAFs** 19

Future Compatibility Issues 20

SAF Interface 20

SAF Parameters 20

pb (parameter block) 20

sn (session) 21

rq (request) 21

Result Codes 22

Creating and Using Custom SAFs 23

▼ To create a custom SAF 23

Writing the Source Code 24

Compiling and Linking 25

Loading and Initializing the SAF 27

Instructing the Server to Call the SAFs 28

Restarting the Server 29

Testing the SAF 29

Overview of NSAPI C Functions 30

Parameter Block Manipulation Routines 30

Protocol Utilities for Service SAFs 30

Memory Management 31

File I/O 31

Network I/O 32

Threads 32

Utilities 32

Required Behavior of SAFs for Each Directive 33

Init SAFs 34

AuthTrans SAFs ... 34
NameTrans SAFs ... 34
PathCheck SAFs ... 35
ObjectType SAFs ... 35
Input SAFs ... 35
Output SAFs ... 35
Service SAFs ... 35
Error SAFs ... 36
AddLog SAFs ... 36
Connect ... 36
DNS ... 36
Filter ... 37
Route ... 37
CGI to NSAPI Conversion ... 37

2 Creating Custom Filters ... 39
Future Compatibility Issues ... 39
NSAPI Filter Interface ... 40
Filter Methods ... 40
C Prototypes for Filter Methods ... 40
insert ... 41
remove ... 41
flush ... 42
read ... 42
write ... 42
writev ... 43
sendfile ... 43
Position of Filters in the Filter Stack ... 43
Filters That Alter Content-Length ... 45
Creating and Using Custom Filters ... 46
▼ To create a custom filter ... 46
Writing the Source Code ... 46
Compiling and Linking ... 47
Loading and Initializing the Filter ... 47
Instructing the Server to Insert the Filter ... 48

Restarting the Server 48
Testing the Filter 48
Overview of NSAPI Functions for Filter Development 49

3 Examples of Custom SAFs and Filters 51
Examples in the Build 52
AuthTrans Example 52
Installing the AuthTrans Example 53
AuthTrans Example Source Code 53
NameTrans Example 55
Installing the NameTrans Example 56
NameTrans Example Source Code 56
PathCheck Example 58
Installing the PathCheck Example 58
PathCheck Example Source Code 59
ObjectType Example 61
Installing the ObjectType Example 61
ObjectType Example Source Code 62
Output Example 63
Installing the Output Example 63
Output Example Source Code 63
Service Example 69
Installing the Service Example 69
Service Example Source Code 70
More Complex Service Example 71
AddLog Example 72
Installing the AddLog Example 72
AddLog Example Source Code 72

4 NSAPI Function Reference 75
NSAPI Functions (in Alphabetical Order) 75
C 76
cache_digest 76
cache_filename 76
cache_fn_to_dig 77

CALLOC 77
ce_free 78
ce_lookup 78
cif_write_entry 79
cinfo_find 80
condvar_init 80
condvar_notify 81
condvar_terminate 81
condvar_wait 82
crit_enter 83
crit_exit 83
crit_init 84
crit_terminate 84
D 85
daemon_atrestart 85
dns_set_hostent 85
F 86
fc_close 86
fc_open 87
filebuf_buf2sd 87
filebuf_close 88
filebuf_getc 89
filebuf_open 89
filebuf_open_nostat 90
filter_create 91
filter_find 92
filter_insert 92
filter_layer 93
filter_name 94
filter_remove 94
flush 95
FREE 95
fs_blk_size 96
fs_blks_avail 97
func_exec 97
func_find 98

func_insert 98
I 99
insert 99
L 100
log_error 100
M 101
magnus_atrestart 101
MALLOC 101
N 102
net_flush 102
net_ip2host 103
net_read 103
net_sendfile 104
net_write 105
netbuf_buf2sd 106
netbuf_close 107
netbuf_getc 107
netbuf_grab 108
netbuf_open 108
nsapi_module_init 109
NSAPI_RUNTIME_VERSION 109
NSAPI_VERSION 110
P 111
param_create 111
param_free 111
pblock_copy 112
pblock_create 112
pblock_dup 113
pblock_find 113
pblock_findlong 114
pblock_findval 115
pblock_free 115
pblock_nlinsert 116
pblock_nninsert 116
pblock_nvinsert 117
pblock_pb2env 118

pblock_pblock2str 118
pblock_pinsert 119
pblock_remove 119
pblock_replace_name 120
pblock_str2pblock 121
PERM_CALLOC 121
PERM_FREE 122
PERM_MALLOC 123
PERM_REALLOC 123
PERM_STRDUP 124
prepare_nsapi_thread 125
protocol_dump822 125
protocol_finish_request 126
protocol_handle_session 126
protocol_parse_request 127
protocol_scan_headers 127
protocol_set_finfo 128
protocol_start_response 129
protocol_status 130
protocol_uri2url 131
protocol_uri2url_dynamic 131
R 132
read 132
REALLOC 133
remove 134
request_create 134
request_free 135
request_header 135
S 136
sem_grab 136
sem_init 137
sem_release 137
sem_terminate 138
sem_tgrab 138
sendfile 139
session_create 140

session_dns ... 140
session_free ... 141
session_maxdns ... 141
shexp_casecmp ... 142
shexp_cmp ... 142
shexp_match ... 143
shexp_valid ... 144
shmem_alloc ... 145
shmem_free ... 145
STRDUP ... 146
system_errmsg ... 147
system_fclose ... 147
system_flock ... 148
system_fopenRO ... 148
system_fopenRW ... 149
system_fopenWA ... 149
system_fread ... 150
system_fwrite ... 151
system_fwrite_atomic ... 151
system_gmtime ... 152
system_localtime ... 153
system_lseek ... 153
system_rename ... 154
system_ulock ... 154
system_unix2local ... 155
systhread_attach ... 156
systhread_current ... 156
systhread_getdata ... 157
systhread_init ... 157
systhread_newkey ... 158
systhread_setdata ... 158
systhread_sleep ... 159
systhread_start ... 159
systhread_terminate ... 160
See Also ... 160
systhread_timerset ... 160

U 161
USE_NSAPI_VERSION 161
util_can_exec 162
util_chdir2path 163
util_cookie_find 163
util_does_process_exist 164
util_env_create 164
util_env_find 165
util_env_free 166
util_env_replace 166
util_env_str 167
util_get_current_gmt 167
util_get_int_from_aux_file 168
util_get_int_from_file 168
util_get_long_from_aux_file 169
util_get_long_from_file 169
util_get_string_from_aux_file 170
util_get_string_from_file 171
util_getline 171
util_hostname 172
util_is_mozilla 172
util_is_url 173
util_itoa 173
util_later_than 174
util_make_filename 174
util_make_gmt 175
util_make_local 175
util_move_dir 176
util_move_file 176
util_parse_http_time 177
util_put_int_to_file 177
util_put_long_to_file 178
util_put_string_to_aux_file 179
util_put_string_to_file 179
util_sect_id 180
util_sh_escape 180

util_snprintf 181
util_sprintf 182
util_strcasecmp 182
util_strftime 183
util_strncasecmp 184
util_uri_check 184
util_uri_escape 185
util_uri_is_evil 185
util_uri_parse 186
util_uri_unescape 186
util_url_cmp 187
util_url_fix_host name 188
util_url_has_FQDN 188
util_vsnprintf 189
util_vsprintf 189
W 190
write 190
writev 191

5 Data Structure Reference 193
Privatization of Some Data Structures 194
Session 194
pblock 195
pb_entry 195
pb_param 195
Session->client 196
Request 196
stat 197
shmem_s 197
cinfo 198
sendfiledata 198
Filter 198
FilterContext 199
FilterLayer 199
FilterMethods 199

CacheEntry Data Structure 200
CacheState Data Structure 201
ConnectMode Data Structure 202

6 Using Wildcard Patterns 203
Wildcard Patterns 203
Wildcard Examples 204

7 Time Formats 207
Time format strings 207

8 Hypertext Transfer Protocol 209
HTTP Compliance 209
HTTP Requests 210
Request Method, URI, and Protocol Version 210
Request Headers 210
Request Data 210
Server Responses 211
HTTP Protocol Version, Status Code, and Reason Phrase 211
Response Headers 212
Response Data 213
Buffered Streams 213

A Alphabetical List of NSAPI Functions and Macros 215

Index 223

Preface

This *NSAPI Developer's Guide* provides a reference of the NSAPI functions you can use to define new plugins.

This preface consists of the following sections:

- "Who Should Use This Book" on page 13
- "How This Book Is Organized" on page 13
- "The Proxy Server Documentation Set" on page 14
- "Documentation Conventions" on page 15
- "Default Paths and File Names" on page 17
- "Documentation, Support, and Training" on page 17
- "Searching Oracle Product Documentation" on page 18
- "Third-Party Web Site References" on page 18

Who Should Use This Book

The intended audience for this guide is the person who develops, assembles, and deploys NSAPI plug-ins in a corporate enterprise. This guide assumes you are familiar with the following topics:

- HTTP
- HTML
- NSAPI
- C programming
- Software development processes, including debugging and source code control

How This Book Is Organized

The following table lists the chapters in the guide and their contents.

TABLE P–1 Guide Organization

Chapter	Description
Chapter 1, "Creating Custom SAFs"	This chapter discusses how to create your own plug-ins that define new SAFs to modify or extend the way the server handles requests.

TABLE P–1 Guide Organization (Continued)

Chapter 2, "Creating Custom Filters"	This chapter discusses how to create custom filters that you can use to intercept, and potentially modify, incoming content presented to or generated by another function.
Chapter 3, "Examples of Custom SAFs and Filters"	This chapter provides examples of custom SAFs to use at each stage in the request-handling process.
Chapter 4, "NSAPI Function Reference"	This chapter presents a reference of the NSAPI functions. You use NSAPI functions to define SAFs.
Chapter 5, "Data Structure Reference"	This chapter discusses some of the commonly used NSAPI data structures.
Chapter 6, "Using Wildcard Patterns"	This chapter lists the wildcard patterns you can use when specifying values in `obj.conf` and various predefined SAFs.
Chapter 7, "Time Formats"	This chapter lists time formats.
Chapter 8, "Hypertext Transfer Protocol"	This chapter gives an overview of HTTP.
Appendix A, "Alphabetical List of NSAPI Functions and Macros"	This appendix provides an alphabetical list of NSAPI functions and macros.

The Proxy Server Documentation Set

The documentation set lists the Oracle documents that are related to Proxy Server. The URL for Proxy Server 4.0.14 documentation is `http://docs.sun.com/coll/1311.14`. For an introduction to Proxy Server, refer to the books in the order in which they are listed in the following table.

TABLE P–2 Proxy Server Documentation

Document Title	Contents
Oracle iPlanet Web Proxy Server 4.0.14 Release Notes	The Proxy Server release: ■ Late-breaking information about the software and the documentation ■ New features ■ Supported platforms and environments ■ System requirements ■ Known issues and workarounds
Oracle iPlanet Web Proxy Server 4.0.14 Installation and Migration Guide	Performing installation and migration tasks: ■ Installing Proxy Server ■ Migrating from version 3.6 to version 4

TABLE P–2 Proxy Server Documentation *(Continued)*

Document Title	Contents
Oracle iPlanet Web Proxy Server 4.0.14 Administration Guide	Performing administration and management tasks: ■ Using the administration and command-line interfaces ■ Configuring server preferences ■ Managing users and groups ■ Monitoring and logging server activity ■ Using certificates and public key cryptography to secure the server ■ Controlling server access ■ Proxying and routing URLs ■ Caching ■ Filtering content ■ Using a reverse proxy ■ Using SOCKS
Oracle iPlanet Web Proxy Server 4.0.14 Configuration File Reference	Editing configuration files
Oracle iPlanet Web Proxy Server 4.0.14 NSAPI Developer's Guide	Creating custom Netscape Server Application Programming Interface (NSAPI) plugins
Oracle iPlanet Web Proxy Server 4.0.14 Performance Tuning, Sizing, and Scaling Guide	Tuning Proxy Server to optimize performance

Documentation Conventions

This section describes the following conventions used in Proxy Server documentation:

- "Typographic Conventions" on page 15
- "Symbol Conventions" on page 16
- "Shell Prompts in Command Examples" on page 17

Typographic Conventions

The following table describes the typographic changes that are used in this book.

TABLE P–3 Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	A placeholder to be replaced with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized (note that some emphasized items appear bold online)	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file.

Symbol Conventions

The following table explains symbols that might be used in this book.

TABLE P–4 Symbol Conventions

Symbol	Description	Example	Meaning
`[ ]`	Contains optional arguments and command options.	`ls [-l]`	The `-l` option is not required.
`{ | }`	Contains a set of choices for a required command option.	`-d {y|n}`	The `-d` option requires that you use either the `y` argument or the `n` argument.
`${ }`	Indicates a variable reference.	`${com.sun.javaRoot}`	References the value of the `com.sun.javaRoot` variable.
-	Joins simultaneous multiple keystrokes.	Control-A	Press the Control key while you press the A key.
+	Joins consecutive multiple keystrokes.	Ctrl+A+N	Press the Control key, release it, and then press the subsequent keys.
→	Indicates menu item selection in a graphical user interface.	File → New → Templates	From the File menu, choose New. From the New submenu, choose Templates.

Shell Prompts in Command Examples

The following table shows default system prompts and superuser prompts.

TABLE P–5 Shell Prompts

Shell	Prompt
C shell on UNIX and Linux systems	`machine_name%`
C shell superuser on UNIX and Linux systems	`machine_name#`
Bourne shell and Korn shell on UNIX and Linux systems	`$`
Bourne shell and Korn shell superuser on UNIX and Linux systems	`#`

Default Paths and File Names

The following table describes the default paths and file names used in Proxy Server documentation.

TABLE P–6 Default Paths and File Names

Placeholder	Description	Default Value
install-dir	Represents the base installation directory for iPlanet Web Proxy Server.	Solaris and Linux installations: `$HOME/Oracle/Middleware/ProxyServer4` Windows installations: `C:\Oracle\Middleware\ProxyServer4`

Documentation, Support, and Training

The Oracle web site provides information about the following additional resources:

- Documentation (http://docs.sun.com/)
- Support (http://www.sun.com/support/)
- Training (http://education.oracle.com/pls/web_prod-plq-dad/db_pages.getpage?page_id=315)

Searching Oracle Product Documentation

Besides searching Oracle product documentation from the docs.sun.com web site, you can use a search engine by typing the following syntax in the search field:

search-term `site:docs.sun.com`

For example, to search for "proxy," type the following:

```
proxy site:docs.sun.com
```

To include other Oracle web sites in your search (for example, java.sun.com, www.sun.com, and developers.sun.com), use `sun.com` in place of `docs.sun.com` in the search field.

Third-Party Web Site References

Third-party URLs are referenced in this document and provide additional, related information.

Note – Oracle is not responsible for the availability of third-party web sites mentioned in this document. Oracle does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Oracle will not be responsible or liable for any actual or alleged damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

CHAPTER 1

Creating Custom SAFs

This chapter describes how to write your own NSAPI plug-ins that define custom Server Application Functions (SAFs). Creating plug-ins enables you to modify or extend the built-in functionality of Proxy Server. For example, you can modify the server to handle user authorization in a special way or generate dynamic HTML pages based on information in a database.

This chapter contains the following sections:

- "Future Compatibility Issues" on page 20
- "SAF Interface" on page 20
- "SAF Parameters" on page 20
- "Result Codes" on page 22
- "Creating and Using Custom SAFs" on page 23
- "Overview of NSAPI C Functions" on page 30
- "Required Behavior of SAFs for Each Directive" on page 33
- "CGI to NSAPI Conversion" on page 37

Before writing custom SAFs, you should familiarize yourself with the request-handling process, as described in *Oracle iPlanet Web Proxy Server 4.0.14 Configuration File Reference*. Also, before writing a custom SAF, see whether a built-in SAF already accomplishes the tasks you have in mind.

For information about predefined SAFs used in the `obj.conf` file, see *Oracle iPlanet Web Proxy Server 4.0.14 Configuration File Reference*.

For a complete list of the NSAPI routines for implementing custom SAFs, see Chapter 4, "NSAPI Function Reference"

Future Compatibility Issues

The NSAPI interface might change in a future version of Proxy Server. To keep your custom plug-ins upgradeable :

- Make sure plug-in users know how to edit the configuration files such as `magnus.conf` and `obj.conf` manually. The plug-in installation software should not be used to edit these configuration files.
- Keep the source code so you can recompile the plug-in.

SAF Interface

All SAFs, both custom and built-in have the same C interface regardless of the request-handling step for which they are written. These small functions are designed for a specific purpose within a specific request-response step. They receive parameters from the directive that invokes them in the `obj.conf` file, from the server, and from previous SAFs.

The C interface for a SAF:

```
int function(pblock *pb, Session *sn, Request *rq);
```

The SAF parameter section discusses the parameters in detail.

The SAF returns a result code that indicates whether and how it succeeded. The server uses the result code from each function to determine how to proceed with processing the request. See , for details of the result codes.

SAF Parameters

This section discusses the SAF parameters in detail. The parameters are:

- “pb (parameter block)” on page 20 — Contains the parameters from the directive that invokes the SAF in the `obj.conf` file.
- “sn (session)” on page 21 — Contains information relating to a single TCP/IP session.
- “rq (request)” on page 21 — Contains information relating to the current request.

pb (parameter block)

The `pb` parameter is a pointer to a `pblock` data structure that contains values specified by the directive that invokes the SAF. A `pblock` data structure contains a series of name-value pairs.

The following example shows a directive that invokes the `basic-nsca` function:

```
AuthTrans fn=basic-ncsa auth-type=basic dbm=/<Install_Root>
    /<Instance_Directory>/userdb/rs
```

In this case, the `pb` parameter passed to `basic-ncsa` contains name-value pairs that correspond to `auth-type=basic` and `dbm=/<Install_Root>/<Instance_Directory>/userdb/rs`.

NSAPI provides a set of functions for working with `pblock` data structures. For example, `pblock_findval()` returns the value for a given name in a `pblock`. See "Parameter Block Manipulation Routines" on page 30, for a summary of the most commonly used functions for working with parameter blocks.

sn (session)

The `sn` parameter is a pointer to a `session` data structure. This parameter contains variables related to an entire session, that is, the time between the opening and closing of the TCP/IP connection between the client and the server. The same `sn` pointer is passed to each SAF called within each request for an entire session. The following list describes the most important fields in this data structure see Chapter 4, "NSAPI Function Reference," for information about NSAPI routines for manipulating the `session` data structure.

- `sn->client`

  Pointer to a `pblock` containing information about the client such as its IP address, DNS name, or certificate. If the client does not have a DNS name or if the name cannot be found, it will be set to `-none`.

- `sn->csd`

  Platform-independent client socket descriptor. This value is passed to the routines for reading from and writing to the client.

rq (request)

The `rq` parameter is a pointer to a `request` data structure. This parameter contains variables related to the current request, such as the request headers, URI, and local file system path. The same `request` pointer is passed to each SAF called in the request-response process for an HTTP request.

The following list describes the most important fields in this data structure. See Chapter 4, "NSAPI Function Reference," for information about NSAPI routines for manipulating the `request` data structure.

- `rq->vars`

Pointer to a `pblock` containing the server's working variables. This `pblock` includes anything not specifically found in the other three `pblocks`. The contents of this `pblock` vary depending on the specific request and the type of SAF. For example, an `AuthTrans` SAF might insert an `auth-user` parameter into `rq->vars` that can be used subsequently by a `PathCheck` SAF.

- `rq->reqpb`

  Pointer to a `pblock` containing elements of the HTTP request. This `pblock` includes the HTTP method (`GET`, `POST`, and so on), the URI, the protocol (normally HTTP/1.0), and the query string. This `pblock` does not normally change throughout the request-response process.

  Note – While obtaining the query string associated with a request, the query string is stored in `reqpb` `pblock` of the `request` structure only if the request URL is relative. If the request URL is absolute, the query string is not separated.

- `rq->headers`

  Pointer to a `pblock` containing all of the request headers, such as `User-Agent`, `If-Modified-Since`, and so on, received from the client in the HTTP request. See Chapter 8, "Hypertext Transfer Protocol," for more information about request headers. This `pblock` does not normally change throughout the request-response process.

- `rq->srvhdrs`

  Pointer to a `pblock` containing the response headers, such as `Server`, `Date`, `Content-Type`, `Content-Length`, and so on, to be sent to the client in the HTTP response. See Chapter 8, "Hypertext Transfer Protocol," for more information about response headers.

The `rq` parameter is the primary mechanism for passing information throughout the request-response process. On input to a SAF, `rq` contains the values that were inserted or modified by previously executed SAFs. On output, `rq` contains any modifications or additional information inserted by the SAF. Some SAFs depend on the existence of specific information provided at an earlier step in the process. For example, a `PathCheck` SAF retrieves values in `rq->vars` that were previously inserted by an `AuthTrans` SAF.

Result Codes

Upon completion, a SAF returns a result code. The result code indicates what the server should do next. The result codes are:

- `REQ_PROCEED`

Indicates that the SAF achieved its objective. For some request-response steps (`AuthTrans`, `NameTrans`, `Service`, and `Error`), this result code tells the server to proceed to the next request-response step, skipping any other SAFs in the current step. For the other request-response steps (`PathCheck`, `ObjectType`, and `AddLog`), the server proceeds to the next SAF in the current step.

- `REQ_NOACTION`

  Indicates that the SAF took no action. The server continues with the next SAF in the current server step.
- `REQ_ABORTED`

  Indicates that an error occurred and an HTTP response should be sent to the client to indicate the cause of the error. A SAF returning `REQ_ABORTED` should also set the HTTP response status code. If the server finds an `Error` directive matching the status code or reason phrase, it executes the SAF specified. If no directive is found, the server sends a default HTTP response with the status code and reason phrase plus a short HTML page reflecting the status code and reason phrase for the user. The server then goes to the first `AddLog` directive.
- `REQ_EXIT`

  Indicates that the connection to the client was lost. This result code should be returned when the SAF fails in reading or writing to the client. The server then goes to the first `AddLog` directive.

Creating and Using Custom SAFs

Custom SAFs are functions in shared libraries that are loaded and called by the server.

▼ To create a custom SAF

1. **“Writing the Source Code” on page 24 using the NSAPI functions. Each SAF is written for a specific directive.**
2. **“Compiling and Linking” on page 25 the source code to create a shared library (`.so`, `.sl`, or `.dll`) file.**
3. **“Loading and Initializing the SAF” on page 27 by editing the `magnus.conf` file to perform the following actions:**
   - Load the shared library file containing your custom SAFs
     - Initialize the SAFs if necessary

4. **"Instructing the Server to Call the SAFs" on page 28 by editing `obj.conf` to call your custom SAFs at the appropriate time.**

5. **"Restarting the Server" on page 29.**

6. **"Testing the SAF" on page 29 by accessing your server from a browser with a URL that triggers your function.**

   The following sections describe these steps in greater detail.

Writing the Source Code

Write custom SAFs using NSAPI functions. For a summary of some of the most commonly used NSAPI functions, see "Overview of NSAPI C Functions" on page 30. For information about available routines, see Chapter 4, "NSAPI Function Reference"

For examples of custom SAFs, see `nsapi/examples/` in the server root directory, and Chapter 3, "Examples of Custom SAFs and Filters"

The signature for all SAFs is:

```
int function(pblock *pb, Session *sn, Request *rq);
```

For more details on the parameters, see "SAF Parameters" on page 20.

Proxy Server runs as a multi-threaded single process. UNIX platforms uses two processes, a parent and a child, for historical reasons. The parent process performs some initialization and forks the child process. The child process performs further initialization and handles all of the HTTP requests.

Keep the following guidelines in mind when writing your SAF:

- Write thread-safe code
- Blocking can affect performance
- Write small functions with parameters and configure them in `obj.conf`
- Carefully check and handle all errors and log them so you can determine the source of problems and fix them

If necessary, write an initialization function that performs initialization tasks required by your new SAFs. The initialization function has the same signature as other SAFs:

```
int function(pblock *pb, Session *sn, Request *rq);
```

SAFs work by obtaining certain types of information from their parameters. In most cases, parameter block (`pblock`) data structures provide the fundamental storage mechanism for

these parameters. A `pblock` maintains its data as a collection of name-value pairs. For a summary of the most commonly used functions for working with `pblock` structures, see "Parameter Block Manipulation Routines" on page 30.

A SAF, definition does not specifically state which directive it is written for. However, each SAF must be written for a specific directive such as `AuthTrans`, `Service`, and so on. A SAF must conform behavior consistent wit the directive for which it was written. For more details, see "Required Behavior of SAFs for Each Directive" on page 33.

Compiling and Linking

Compile and link your code with the native compiler for the target platform. For UNIX, use the `gmake` command. For Windows, use the `nmake` command. For Windows, use Microsoft Visual C++ 6.0 or newer. You must have an import list that specifies all global variables and functions to access from the server binary. Use the correct compiler and linker flags for your platform. Refer to the example Makefile in the *server_root*`/plugins/nsapi/examples` directory.

This section provides following guidelines for compiling and linking.

Include Directory and nsapi.h File

Add the *server_root*`/plugins/include` (UNIX) or *server_root*`\\plugins\\include` (Windows) directory to your makefile to include the `nsapi.h` file.

Linker Libraries

Add the *server_root*`/bin/https/lib` (UNIX) or *server_root*`\\bin\\https\\bin` (Windows) library directory to your linker command.

The following table lists the relevant libraries.

TABLE 1–1 Linker Libraries

Platform	Library
Windows	`ns-httpd40.dll` (in addition to the standard Windows libraries)
HP-UX	`libns-httpd40.sl`
All other UNIX platforms	`libns-httpd40.so`

Linker Commands and Options for Generating a Shared Object

To generate a shared library, use the commands and options listed in the following table.

TABLE 1–2 Linker Commands and Options

Platform	Options
Solaris Operating System (SPARC Platform Edition)	`ld -G` or `cc -G`
Windows	`link -LD`
HP-UX	`cc +Z -b -Wl,+s -Wl,-B,symbolic`
AIX	`cc -p 0 -berok -blibpath:$(LD_RPATH)`
Compaq	`cc -shared`
Linux	`gcc -shared`
IRIX	`cc -shared`

Additional Linker Flags

Use the linker flags in the following table to specify which directories should be searched for shared objects during runtime to resolve symbols.

TABLE 1–3 Linker Flags

Platform	Flags
Solaris SPARC	`-R` *dir*`:`*dir*
Windows	no flags, but the `ns-httpd40.dll` file must be in the system PATH variable
HP-UX	`-Wl,+b,`*dir*`,`*dir*
AIX	`-blibpath:`*dir*`:`*dir*
Compaq	`-rpath` *dir*`:`*dir*
Linux	`-Wl,-rpath,`*dir*`:`*dir*
IRIX	`-Wl,-rpath,`*dir*`:`*dir*

On UNIX, you can also set the library search path using the `LD_LIBRARY_PATH` environment variable, which must be set when you start the server.

Compiler Flags

The following table lists the flags and defines you need to use for compilation of your source code.

TABLE 1–4 Compiler Flags and Defines

Parameter	Description
Solaris SPARC	`-DXP_UNIX -D_REENTRANT -KPIC -DSOLARIS`
Windows	`-DXP_WIN32 -DWIN32 /MD`
HP-UX	`-DXP_UNIX -D_REENTRANT -DHPUX`
AIX	`-DXP_UNIX -D_REENTRANT -DAIX $(DEBUG)`
Compaq	`-DXP_UNIX -KPIC`
Linux	`-DLINUX -D_REENTRANT -fPIC`
IRIX	`-o32 -exceptions -DXP_UNIX -KPIC`
All platforms	`-MCC_HTTPD -NET_SSL`

The following table lists the optional flags and defines you can use.

TABLE 1–5 Optional Flags and Defines

Flag/Define	Platforms	Description
`-DSPAPI20`	All	Needed for the proxy utilities function include file `putil.h`

Loading and Initializing the SAF

For each shared library (plug-in) containing custom SAFs to be loaded into Proxy Server, add an `Init` directive that invokes the `load-modules` SAF to `obj.conf`.

The syntax for a directive that calls `load-modules` is:

```
Init fn=load-modules shlib=[path]sharedlibname funcs="SAF1,...,SAFn"
```

- `shlib` is the local file system path to the shared library (plug-in).
- `funcs` is a comma-separated list of function names to be loaded from the shared library. Function names are case sensitive. You may use dash a (-) in place of an underscore (_) in function names. do not include spaces in the function name list.

  If the new SAFs require initialization, be sure that the initialization function is included in the `funcs` list.

For example, if you created a shared library `animations.so` that defines two SAFs `do_small_anim()` and `do_big_anim()` and also defines the initialization function `init_my_animations`, you would add the following directive to load the plug-in:

```
Init fn=load-modules shlib=animations.so funcs="do_small_anim,do_big_anim,
    init_my_animations"
```

If necessary, also add an `Init` directive that calls the initialization function for the newly loaded plug-in. For example, if you defined the function `init_my_new_SAF()` to perform an operation on the `maxAnimLoop` parameter, you would add a directive such as the following to `magnus.conf`:

```
Init fn=init_my_animations maxAnimLoop=5
```

Instructing the Server to Call the SAFs

Next, add directives to `obj.conf` to instruct the server to call each custom SAF at the appropriate time. The syntax for directives is:

Directive `fn=`*function-name* `[`*name1*`="`*value1*`"]...[`*nameN*`="`*valueN*`"]`

- *Directive* is one of the server directives, such as `AuthTrans`, `Service`, and so on.
- *function-name* is the name of the SAF to execute.
- *nameN="valueN"* are the names and values of parameters which are passed to the SAF.

Depending on what your new SAF does, you might need to add just one directive to `obj.conf`, or you might need to add more than one directive to provide complete instructions for invoking the new SAF.

For example, if you define a new `AuthTrans` or `PathCheck` SAF, you could just add an appropriate directive in the default object. However, if you define a new `Service` SAF to be invoked only when the requested resource is in a particular directory or has a new kind of file extension, you would need to take extra steps.

If your new `Service` SAF is to be invoked only when the requested resource has a new kind of file extension, you might need to add an entry to the MIME types file so that the `type` value is set properly during the `ObjectType` stage. Then you could add a `Service` directive to the default object that specifies the desired `type` value.

If your new `Service` SAF is to be invoked only when the requested resource is in a particular directory, you might need to define a `NameTrans` directive that generates a `name` or `ppath` value that matches another object. Then, in the new object, you could invoke the new `Service` function.

For example, suppose your plug-in defines two new SAFs, `do_small_anim()` and `do_big_anim()`, which both take `speed` parameters. These functions run animations. All files to be treated as small animations reside in the directory `D:/<Install_Root>/<Instance_Directory>/docs/animations/small`, while all files to be treated as full-screen animations reside in the directory `D:/<Install_Root>/<Instance_Directory>/docs/animations/fullscreen`.

To ensure that the new animation functions are invoked whenever a client sends a request for either a small or full-screen animation, you would add NameTrans directives to the default object to translate the appropriate URLs to the corresponding path names and also assign a name to the request.

```
NameTrans fn=pfx2dir from="/animations/small"
 dir="/<Install_Root>/<Instance_Directory>/docs/animations/small"
    name="small_anim"
NameTrans fn=pfx2dir from="/animations/fullscreen"
 dir="<Install_Root>/<Instance_Directory>docs/animations/fullscreen"
 name="fullscreen_anim"
```

You also need to define objects that contain the Service directives that run the animations and specify the speed parameter.

```
<Object name="small_anim">
Service fn=do_small_anim speed=40
</Object>
<Object name="fullscreen_anim">
Service fn=do_big_anim speed=20
</Object>
```

Restarting the Server

After modifying obj.conf, you need to restart the server. A restart is required for all plug-ins that implement SAFs or filters.

Testing the SAF

Test your SAF by accessing your server from a browser with a URL that triggers your function. For example, if your new SAF is triggered by requests to resources in http://*server-name*/animations/small, try requesting a valid resource that starts with that URI.

You should disable caching in your browser so that the server is sure to be accessed. In Netscape Navigator, Press the Shift key while clicking the Reload button to ensure that the cache is not used. If the images are already in the cache, this action does not always force the client to fetch images from the source.

You might also want to disable the server cache using the cache-init SAF.

Examine the access log and error log to help with debugging.

Overview of NSAPI C Functions

NSAPI provides a set of C functions that are used to implement SAFs. These functions serve several purposes; They provide platform independence across Proxy Server operating system and hardware platforms. They provide improved performance. They are thread-safe which is a requirement for SAFs. They prevent memory leaks. And they provide functionality necessary for implementing SAFs. You should always use these NSAPI routines when defining new SAFs.

This section provides an overview of the function categories available and some of the more commonly used routines. All of the public routines are described in detail in Chapter 4, "NSAPI Function Reference"

The main categories of NSAPI functions are:

- "Parameter Block Manipulation Routines" on page 30
- "Protocol Utilities for Service SAFs" on page 30
- "Memory Management" on page 31
- "File I/O" on page 31
- "Network I/O" on page 32
- "Threads" on page 32
- "Utilities" on page 32

Parameter Block Manipulation Routines

The parameter block manipulation functions provide routines for locating, adding, and removing entries in a `pblock` data structure

- "pblock_findval" on page 115 returns the value for a given name in a `pblock`.
- "pblock_nvinsert" on page 117 adds a new name-value entry to a `pblock`.
- "pblock_remove" on page 119 removes a `pblock` entry by name from a `pblock`. The entry is not disposed. Use "param_free" on page 111 to free the memory used by the entry.
- "param_free" on page 111 frees the memory for the given `pblock` entry.
- "pblock_pblock2str" on page 118 creates a new string containing all of the name-value pairs from a `pblock` in the form "*name*=*value name*=*value*." This string can be useful for debugging.

Protocol Utilities for Service SAFs

Protocol utilities provide functionality necessary to implement `Service` SAFs

- "request_header" on page 135 returns the value for a given request header name, reading the headers if necessary. This function must be used when requesting entries from the browser header `pblock` (`rq->headers`).

- "protocol_status" on page 130 sets the HTTP response status code and reason phrase.
- "protocol_start_response" on page 129 sends the HTTP response and all HTTP headers to the browser.

Memory Management

Memory management routines provide fast, platform-independent versions of the standard memory management routines. These routines also prevent memory leaks by allocating from a temporary memory, called "pooled" memory for each request, and then disposing the entire pool after each request. There are wrappers for standard memory routines for using permanent memory.

- "MALLOC" on page 101
- "FREE" on page 95
- "PERM_STRDUP" on page 124
- "REALLOC" on page 133
- "CALLOC" on page 77
- "PERM_MALLOC" on page 123
- "PERM_FREE" on page 122
- "PERM_REALLOC" on page 123
- "PERM_CALLOC" on page 121

File I/O

The file I/O functions provide platform-independent, thread-safe file I/O routines.

- "system_fopenRO" on page 148 opens a file for read-only access.
- "system_fopenRW" on page 149 opens a file for read-write access, creating the file if necessary.
- "system_fopenWA" on page 149 opens a file for write-append access, creating the file if necessary.
- "system_fclose" on page 147 closes a file.
- "system_fread" on page 150 reads from a file.
- "system_fwrite" on page 151 writes to a file.
- "system_fwrite_atomic" on page 151 locks the given file before writing to it. This avoids interference between simultaneous writes by multiple threads.

Network I/O

Network I/O functions provide platform-independent, thread-safe network I/O routines. These routines work with SSL when SSL is enabled.

- "netbuf_grab" on page 108 reads from a network buffer's socket into the network buffer.
- "netbuf_getc" on page 107 gets a character from a network buffer.
- "net_flush" on page 102 flushes buffered data.
- "net_read" on page 103 reads bytes from a specified socket into a specified buffer.
- "net_sendfile" on page 104 sends the contents of a specified file to a specified a socket.
- "net_write" on page 105 writes to the network socket.

Threads

Thread functions include functions for creating your own threads that are compatible with the server's threads. There are also routines for critical sections and condition variables.

- "systhread_start" on page 159 creates a new thread.
- "systhread_sleep" on page 159 puts a thread to sleep for a given time.
- "crit_init" on page 84 creates a new critical section variable.
- "crit_enter" on page 83 gains ownership of a critical section.
- "crit_exit" on page 83 surrenders ownership of a critical section.
- "crit_terminate" on page 84 disposes of a critical section variable.
- "condvar_init" on page 80 creates a new condition variable.
- "condvar_notify" on page 81 awakens any threads blocked on a condition variable.
- "condvar_wait" on page 82 blocks on a condition variable.
- "condvar_terminate" on page 81 disposes of a condition variable.
- "prepare_nsapi_thread" on page 125 allows threads that are not created by the server to act like server-created threads.

Utilities

Utility functions include platform-independent, thread-safe versions of many standard library functions (such as string manipulation), as well as new utilities useful for NSAPI.

- "daemon_atrestart" on page 85 (UNIX only) registers a user function to be called when the server is sent a restart signal (HUP) or at shutdown.
- "condvar_init" on page 80 gets the next line (up to a LF or CRLF) from a buffer.
- "util_hostname" on page 172 gets the local host name as a fully qualified domain name.
- "util_later_than" on page 174 compares two dates.

- "util_snprintf" on page 181 is the same as the standard library routine `sprintf()`.
- "util_strftime" on page 183 is the same as the standard library routine `strftime()`.
- "util_uri_escape" on page 185 converts the special characters in a string into URI-escaped format.
- "util_uri_unescape" on page 186 converts the URI-escaped characters in a string back into special characters.

Note – You cannot use an embedded null in a string, because NSAPI functions assume that a null is the end of the string. Therefore, passing Unicode-encoded content through an NSAPI plug-in.

Required Behavior of SAFs for Each Directive

SAF definitions , you should define it to do certain things, depending on which stage of the request-handling process will invoke the SAF. For example, SAFs to be invoked during the `Init` stage must conform to different requirements than SAFs to be invoked during the `Service` stage.

The `rq` parameter is the primary mechanism for passing information throughout the request-response process. On input to a SAF, `rq` contains whatever values were inserted or modified by previously executed SAFs. On output, `rq` contains any modifications or additional information inserted by the SAF. Some SAFs depend on the existence of specific information provided at an earlier step in the process. For example, a `PathCheck` SAF retrieves values in `rq->vars` that were previously inserted by an `AuthTrans` SAF.

This section outlines the expected behavior of SAFs used at each stage in the request-handling process.

- Init SAFs
- AuthTrans SAFs
- NameTrans SAFs
- PathCheck SAFs
- ObjectType SAFs
- Input SAFs
- Output SAFs
- Service SAFs
- AddLog SAFs
- Error SAFs
- Connect SAFs
- DNS SAFs
- Filter SAFs
- Route SAFs

For more detailed information about these SAFs, see *Oracle iPlanet Web Proxy Server 4.0.14 Configuration File Reference*.

Init SAFs

- Purpose: Initialize at startup.
- Called at server startup and restart.
- `rq` and `sn` are NULL.
- Initialize any shared resources such as files and global variables.
- Can register callback function with `daemon_atrestart()` to clean up.
- On error, insert `error` parameter into `pb` describing the error and return `REQ_ABORTED`.
- If successful, return `REQ_PROCEED`.

AuthTrans SAFs

- Verify any authorization information. Only basic authorization is currently defined in the HTTP/1.0 specification.
- Check for an `Authorization` header in `rq->headers` that contains the authorization type and uu-encoded user and password information. If a header was not sent, return `REQ_NOACTION`.
- If a header exists, check the authenticity of user and password.
- If the user name and password are authentic, create an `auth-type`, plus `auth-user` or `auth-group` parameter in `rq->vars` to be used later by `PathCheck` SAFs.
- Return `REQ_PROCEED` if the user was successfully authenticated Return `REQ_NOACTION` otherwise.

NameTrans SAFs

- Purpose: Convert a logical URI to a physical path.
- Perform operations on the logical path (`ppath` in `rq->vars`) to convert it into a full local file system path.
- Return `REQ_PROCEED` if `ppath` in `rq->vars` contains the full local file system path, or `REQ_NOACTION` if not.
- To redirect the client to another site, change `ppath` in `rq->vars` to `/URL`. Add `url` to `rq->vars` with full URL (for example, `http://home.netscape.com/`). Return `REQ_PROCEED`.

PathCheck SAFs

- Purpose: Check path validity and user's access rights.
- Check `auth-type`, `auth-user`, or `auth-group` in `rq->vars.`
- Return `REQ_PROCEED` if the user and group are authorized for this area (`ppath` in `rq->vars`).
- If not authorized, insert `WWW-Authenticate` to `rq->srvhdrs` with a value such as: `Basic; Realm=\\"Our private area\\"`. Call `protocol_status()` to set the HTTP response status to `PROTOCOL_UNAUTHORIZED`. Return `REQ_ABORTED`.

ObjectType SAFs

- Purpose: Determine `content-type` of data.
- If `content-type` in `rq->srvhdrs` already exists, return `REQ_NOACTION`.
- Determine the MIME type and create `content-type` in `rq->srvhdrs`
- Return `REQ_PROCEED` if `content-type` is created, `REQ_NOACTION` otherwise.

Input SAFs

- Purpose: Insert filters that process incoming (client-to-server) data.
- `Input` SAFs are executed when a plug-in or the server first attempts to read entity body data from the client.
- `Input` SAFs are executed at most once per request.
- Return `REQ_PROCEED` to indicate success, or `REQ_NOACTION` to indicate that the SAF performed no action.

Output SAFs

- Purpose: Insert filters that process outgoing (server-to-client) data.
- `Output` SAFs are executed when a plug-in or the server first attempts to write entity body data from the client.
- `Output` SAFs are executed at most once per request.
- Return `REQ_PROCEED` to indicate success, or `REQ_NOACTION` to indicate the SAF performed no action.

Service SAFs

- Purpose: Generate and send the response to the client.

- A Service SAF is only called if each of the optional parameters type, method, and query specified in the directive in obj.conf match the request.
- Remove existing content-type from rq->srvhdrs. Insert correct content-type in rq->srvhdrs.
- Create any other headers in rq->srvhdrs.
- Call "protocol_set_finfo" on page 128 to set the HTTP response status.
- Call "protocol_start_response" on page 129 to send the HTTP response and headers.
- Generate and send data to the client using "net_write" on page 105 .
- Return REQ_PROCEED if successful, REQ_EXIT on write error, or REQ_ABORTED on other failures.

Error SAFs

- Purpose: Respond to an HTTP status error condition.
- The Error SAF is only called if each of the optional parameters code and reason specified in the directive in obj.conf match the current error.
- Error SAFs perform the same action as Service SAFs, but only in response to an HTTP status error condition.

AddLog SAFs

- Purpose: Log the transaction to a log file.
- AddLog SAFs can use any data available in pb, sn, or rq to log this transaction.
- Return REQ_PROCEED.

Connect

- Purpose: Call the connect function you specify.
- Only the first applicable Connect function is called, starting from the most restrictive object. Occasionally you might want to call multiple functions until a connection is established. The function returns REQ_NOACTION if the next function should be called. If it fails to connect, the return value is REQ_ABORT. If it connects successfully, the connected socket descriptor will be returned.

DNS

- Purpose: Calls either the dns-config built-in function or a DNS function that you specify.

Filter

- Purpose: Run an external command and then pipes the data through the external command before processing that data in the proxy. This process is accomplished using the pre-filter function.

Route

- Purpose: Specify information about where the proxy server should route requests.

CGI to NSAPI Conversion

When converting a CGI variable into a SAF using NSAPI, Since the CGI environment variables are not available to NSAPI, you'll retrieve them from the NSAPI parameter blocks. The table below indicates how each CGI environment variable can be obtained in NSAPI.

TABLE 1–6 Parameter Blocks for CGI Variables

CGI getenv()	NSAPI
AUTH_TYPE	pblock_findval("auth-type", rq->vars);
AUTH_USER	pblock_findval("auth-user", rq->vars);
CONTENT_LENGTH	pblock_findval("content-length", rq->headers);
CONTENT_TYPE	pblock_findval("content-type", rq->headers);
GATEWAY_INTERFACE	"CGI/1.1"
HTTP_*	pblock_findval("*", rq->headers); (* is lowercase; dash replaces underscore)
PATH_INFO	pblock_findval("path-info", rq->vars);
PATH_TRANSLATED	pblock_findval("path-translated", rq->vars);
QUERY_STRING	pblock_findval("query", rq->reqpb); (GET only; POST puts query string in body data)
REMOTE_ADDR	pblock_findval("ip", sn->client);
REMOTE_HOST	session_dns(sn) ? session_dns(sn) : pblock_findval("ip", sn->client);
REMOTE_IDENT	pblock_findval("from", rq->headers);(not usually available)
REMOTE_USER	pblock_findval("auth-user", rq->vars);
REQUEST_METHOD	pblock_findval("method", req->reqpb);
SCRIPT_NAME	pblock_findval("uri", rq->reqpb);

TABLE 1–6 Parameter Blocks for CGI Variables *(Continued)*

CGI getenv()	NSAPI
SERVER_NAME	char *util_hostname();
SERVER_PORT	conf_getglobals()->Vport; (as a string)
SERVER_PROTOCOL	pblock_findval("protocol", rq->reqpb);
SERVER_SOFTWARE	MAGNUS_VERSION_STRING
Sun ONE-specific:	
CLIENT_CERT	pblock_findval("auth-cert", rq->vars)
HOST	char *session_maxdns(sn);(may be null)
HTTPS	security_active ? "ON" : "OFF";
HTTPS_KEYSIZE	pblock_findval("keysize", sn->client);
HTTPS_SECRETKEYSIZE	pblock_findval("secret-keysize", sn->client);
QUERY	pblock_findval(query", rq->reqpb); (GET only, POST puts query string in entity-body data)
SERVER_URL	http_uri2url_dynamic("","", sn, rq);

Your code must be thread-safe under NSAPI. You should use NSAPI functions that are thread-safe. Also, you should use the NSAPI memory management and other routines for speed and platform independence.

CHAPTER 2

Creating Custom Filters

This chapter describes how to create custom filters that can be used to intercept and possibly modify the content presented to or generated by another function.

This chapter contains the following sections:

- "Future Compatibility Issues" on page 39
- "NSAPI Filter Interface" on page 40
- "Filter Methods" on page 40
- "Position of Filters in the Filter Stack" on page 43
- "Filters That Alter Content-Length" on page 45
- "Creating and Using Custom Filters" on page 46
- "Overview of NSAPI Functions for Filter Development" on page 49

Future Compatibility Issues

The NSAPI interface might change in a future version of Proxy Server. To keep your custom plug-ins upgradeable:

- Make sure plug-in users know how to edit the configuration files such as `magnus.conf` and `obj.conf` manually. The plug-in installation software should not be used to edit these configuration files.
- Keep the source code so you can recompile the plug-in.

NSAPI Filter Interface

Proxy Server 4 extends NSAPI by introducing a new filter interface that complements the existing Server Application Function (SAF) interface. You use filters to intercept and possibly modify data sent to and from the server. The server communicates with a filter by calling the filter's filter methods. Each filter implements one or more filter methods. A filter method is a C function that performs a specific operation, such as processing data sent by the server.

Filter Methods

This section describes the filter methods that a filter can implement, which are:

- "insert" on page 41
- "remove" on page 41
- "flush" on page 42
- "read" on page 42
- "write" on page 42
- "writev" on page 43
- "sendfile" on page 43

For more information about these methods, see Chapter 4, "NSAPI Function Reference"

C Prototypes for Filter Methods

The C prototypes for the filter methods are:

```
int insert(FilterLayer *layer, pblock *pb);
void remove(FilterLayer *layer);
int flush(FilterLayer *layer);
int read(FilterLayer *layer, void *buf, int amount, int timeout);
int write(FilterLayer *layer, const void *buf, int amount);
int writev(FilterLayer *layer, const struct iovec *iov, int iov_size);
int sendfile(FilterLayer *layer, sendfiledata *sfd);
```

The `layer` parameter is a pointer to a `FilterLayer` data structure, which contains variables related to a particular instance of a filter. Following is a list of the most important fields in the `FilterLayer` data structure:

- `context->sn` — Contains information relating to a single TCP/IP session. This pointer is the same `sn` pointer that's passed to SAFs.
- `context->rq` — Contains information relating to the current request. This pointer is the same `rq` pointer that is passed to SAFs.
- `context->data` — Pointer to filter-specific data.

- lower — A platform-independent socket descriptor used to communicate with the next filter in the stack.

The meaning of the context->data field is defined by the filter developer. Filters that must maintain state information across filter method calls can use context->data to store that information.

For more information about FilterLayer, see "FilterLayer" on page 199.

insert

The insert filter method is called when an SAF such as insert-filter calls the filter_insert function to request that a specific filter be inserted into the filter stack. Each filter must implement the insert filter method.

When insert is called, this method can determine whether the filter should be inserted into the filter stack. For example, the filter could inspect the Content-Type header in the rq->srvhdrs pblock to determine whether the type of data that will be transmitted is relevant. If the filter should not be inserted, the insert filter method should return REQ_NOACTION.

If the filter should be inserted, the insert filter method provides an opportunity to initialize this particular instance of the filter. For example, the insert method could allocate a buffer with MALLOC and store a pointer to that buffer in layer->context->data.

The filter is not part of the filter stack until after insert returns. As a result, the insert method should not attempt to read from, write to, or otherwise interact with the filter stack.

See Also

"insert" on page 99 in Chapter 4, "NSAPI Function Reference"

remove

The remove filter method is called when a filter stack is destroyed when the corresponding socket descriptor is closed, when the server finishes processing the request the filter was associated with, or when an SAF such as remove-filter calls the filter_remove function. The remove filter method is optional.

The remove method can be used to clean up any data the filter allocated in insert and to pass any buffered data to the next filter by calling net_write(layer->lower, ...).

See Also

"remove" on page 134 in Chapter 4, "NSAPI Function Reference"

flush

The flush filter method is called when a filter or SAF calls the net_flush function. The flush method should pass any buffered data to the next filter by calling net_write(layer->lower, ...). The flush method is optional, but it should be implemented by any filter that buffers outgoing data.

See Also

"flush" on page 95 in Chapter 4, "NSAPI Function Reference"

read

The read filter method is called when a filter or SAF calls the net_read function. Filters that deal with in incoming data (data sent from a client to the server) implement the read filter method.

Typically, the read method will attempt to obtain data from the next filter by calling net_read(layer->lower, ...). The read method may then modify the received data before returning it to its caller.

See Also

"read" on page 132 in Chapter 4, "NSAPI Function Reference"

write

The write filter method is called when a filter or SAF calls the net_write function. Filters that are deal with outgoing data (data sent from the server to a client) implement the write filter method.

Typically, the write method will pass data to the next filter by calling net_write(layer->lower, ...). The write method may modify the data before calling net_write. For example, the http-compression filter compresses data before passing it on to the next filter.

If a filter implements the write filter method but does not pass the data to the next layer before returning to its caller, that is, if the filter buffers outgoing data, the filter should also implement the flush method.

See Also

"write" on page 190 in Chapter 4, "NSAPI Function Reference"

writev

The writev filter method performs the same function as the write filter method, but the format of its parameters is different. You do not have to implement the writev filter method. If a filter implements the write filter method but not the writev filter method, the server uses the write method instead of the writev method. A filter should not implement the writev method unless it also implements the write method.

Under some circumstances, the server may run slightly faster when filters that implement the write filter method also implement the writev filter method.

See Also

"writev" on page 191 in Chapter 4, "NSAPI Function Reference"

sendfile

The sendfile filter method performs a function similar to the writev filter method, but it sends a file directly instead of first copying the contents of the file into a buffer. You do not have to implement the sendfile filter method. If a filter implements the write filter method but not the sendfile filter method, the server will use the write method instead of the sendfile method. A filter should not implement the sendfile method unless it also implements the write method.

Under some circumstances, the server may run slightly faster when filters that implement the write filter method also implement the sendfile filter method.

See Also

"sendfile" on page 139 in Chapter 4, "NSAPI Function Reference"

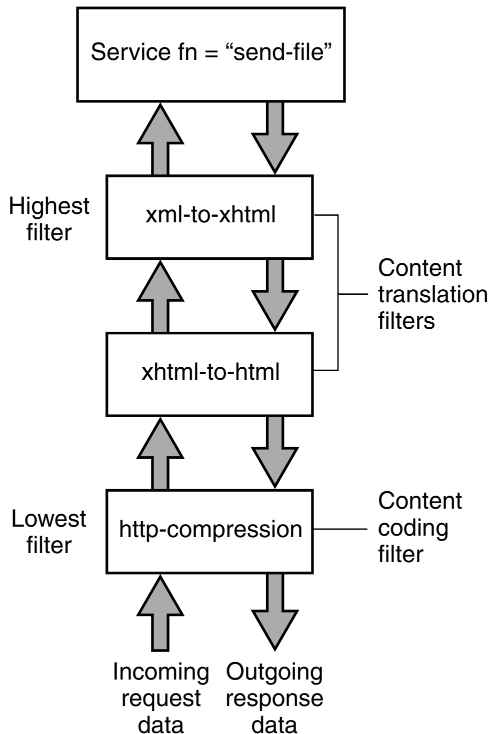
Position of Filters in the Filter Stack

All data sent to the server, such as the result of an HTML form or sent from the server, such as the output of a JSP page is passed through a set of filters known as a filter stack. The server creates a separate filter stack for each connection. While processing a request, individual filters can be inserted into and removed from the stack.

Different types of filters occupy different positions within a filter stack. Filters that deal with application-level content, such as filters that translates a page from XHTML to HTML, occupy a higher position than filters that deal with protocol-level issues, such as filters that format HTTP responses. When two or more filters are defined to occupy the same position in the filter stack, filters that were inserted later will appear higher than filters that were inserted earlier.

Filters positioned higher in the filter stack are given an earlier opportunity to process outgoing data. Filters positioned lower in the stack are given an earlier opportunity to process incoming data. For example, in the following figure, the `xml-to-xhtml` filter is given an earlier opportunity to process outgoing data than the `xhtml-to-html` filter.

FIGURE 2–1 Position of Filters in the Filter Stack



When you create a filter with the `filter_create` function, you specify what position your filter should occupy in the stack. You can also use the `init-filter-order` `Init` SAF to control the position of specific filters within filter stacks. For example, `init-filter-order` can be used to ensure that a filter that converts outgoing XML to XHTML is inserted above a filter that converts outgoing XHTML to HTML.

For more information, see “filter_create” on page 91

Filters That Alter Content-Length

Filters that can alter the length of an incoming request body or outgoing response body must take special steps to ensure interoperability with other filters and SAFs.

Filters that process incoming data are referred to as input filters. If an input filter can alter the length of the incoming request body (for example, if a filter decompresses incoming data) and the `rq->headers` pblock contains a `Content-Length` header, the filter's `insert` filter method should remove the `Content-Length` header and replace it with a `Transfer-encoding: identity` header as follows:

```
pb_param *pp;

pp = pblock_remove("content-length", layer->context->rq->headers);
if (pp != NULL) {
   param_free(pp);
   pblock_nvinsert("transfer-encoding", "identity", layer->context->
      rq->headers);
}
```

Because some SAFs expect a `Content-Length` header when a request body is present, before calling the first `Service` SAF the server will insert all relevant filters, read the entire request body, and compute the length of the request body after it has been passed through all input filters. However, by default, the server will read at most 8192 bytes of request body data. If the request body exceeds 8192 bytes after being passed through the relevant input filters, the request will be cancelled. For more information, see the description of `ChunkedRequestBufferSize` in *Oracle iPlanet Web Proxy Server 4.0.14 Configuration File Reference*.

Filters that process outgoing data are referred to as output filters. If an output filter can alter the length of the outgoing response body (for example, if the filter compresses outgoing data), the filter's `insert` filter method should remove the `Content-Length` header from `rq->srvhdrs` as follows:

```
pb_param *pp;

pp = pblock_remove("content-length", layer->context->rq->srvhdrs);
if (pp != NULL)
   param_free(pp);
```

Creating and Using Custom Filters

Custom filters are defined in shared libraries that are loaded and called by the server.

▼ To create a custom filter

1. **“Writing the Source Code” on page 46 using the NSAPI functions.**
2. **“Compiling and Linking” on page 47 the source code to create a shared library (`.so`, `.sl`, or `.dll`) file.**
3. **“Loading and Initializing the Filter” on page 47 by editing the `magnus.conf` file.**
4. **“Instructing the Server to Insert the Filter” on page 48 by editing the `obj.conf` file to insert your custom filter(s) at the appropriate time.**
5. **“Restarting the Server” on page 48.**
6. **“Testing the Filter” on page 48 by accessing your server from a browser with a URL that triggers your filter.**

   These steps are described in greater detail in the following sections.

Writing the Source Code

Write your custom filter methods using NSAPI functions. For a summary of the NSAPI functions specific to filter development, see “Overview of NSAPI Functions for Filter Development” on page 49. For a summary of general purpose NSAPI functions, see Chapter 4, “NSAPI Function Reference,” Each filter method must be implemented as a separate function. See “Filter Methods” on page 40 for the filter method prototypes.

The filter must be created by a call to `filter_create`. Typically, each plug-in defines an `nsapi_module_init` function that is used to call `filter_create` and perform any other initialization tasks. See “nsapi_module_init” on page 109 and “filter_create” on page 91 for more information.

Filter methods are invoked whenever the server or an SAF calls certain NSAPI functions such as `net_write` or `filter_insert`. As a result, filter methods can be invoked from any thread and should only block using NSAPI functions, for example, `crit_enter` and `net_read`. If a filter method blocks using other functions, for example, the Windows `WaitForMultipleObjects` and `ReadFile` functions, the server might hang. Also, shared objects that define filters should be loaded with the `NativeThread="no"` flag, as described in “Loading and Initializing the Filter” on page 47

If a `filter` method must block using a non-NSAPI function, `KernelThreads 1` should be set in `magnus.conf`. For more information about `KernelThreads`, see the description in Chapter 3, "Syntax and Use of the magnus.conf File," in *Oracle iPlanet Web Proxy Server 4.0.14 Configuration File Reference*.

Keep the following in mind when writing your filter:

- Write thread-safe code
- IO should only be performed using the NSAPI functions documented in "File I/O" on page 31 and "Network I/O" on page 32.
- Thread synchronization should only be performed using the NSAPI functions documented in "Threads" on page 32.
- Blocking can affect performance.
- Carefully check and handle all errors.

For examples of custom filters, see *server_root*`/plugins/nsapi/examples` and also Chapter 3, "Examples of Custom SAFs and Filters"

Compiling and Linking

Filters are compiled and linked in the same way as SAFs. See "Compiling and Linking" on page 25, for more information.

Loading and Initializing the Filter

For each shared library (plug-in) containing custom SAFs to be loaded into Proxy Server, add an `Init` directive that invokes the `load-modules` SAF to `obj.conf`. The syntax for a directive that loads a filter plug-in is:

```
Init fn=load-modules shlib=[path]sharedlibname NativeThread="no"
```

- `shlib` is the local file system path to the shared library (plug-in).
- `NativeThread` indicates whether the plug-in requires native threads. Filters should be written to run on any type of thread. For more information, see "Writing the Source Code" on page 24.

When the server encounters such a directive, server calls the plug-in's `nsapi_module_init` function to initialize the filter.

Instructing the Server to Insert the Filter

Add an `Input` or `Output` directive to `obj.conf` to instruct the server to insert your filter into the filter stack. The format of the directive is as follows:

Directive `fn=insert-filter filter=`"*filter-name*" `[`*name1*="*value1*"`]...[`*nameN*="*valueN*"`]`

- *Directive* is `Input` or `Output`.
- *filter-name* is the name of the filter, as passed to `filter_create`, to insert.
- *nameN*="*valueN*" are the names and values of parameters that are passed to the filter's `insert` filter method.

Filters that process incoming data should be inserted using an `Input` directive. Filters that process outgoing data should be inserted using an `Output` directive.

To ensure that your filter is inserted whenever a client sends a request, add the `Input` or `Output` directive to the default object. For example, the following portion of `obj.conf` instructs the server to insert a filter named `example-replace` and pass it two parameters, `from` and `to`:

```
<Object name="default">
Output fn=insert-filter
       filter="example-replace"
       from="Old String"
       to="New String"
...
</Object>
```

Restarting the Server

For the server to load your plug-in, you must restart the server. A restart is required for all plug-ins that implement SAFs and/or filters.

Testing the Filter

Test your SAF by accessing your server from a browser. You should disable caching in your browser so that the server is sure to be accessed. In Netscape Navigator, press the Shift key while clicking the Reload button to ensure that the cache is not used. If the images are already in the cache, this action does not always force the client to fetch images from source. Examine the access and error logs to help with debugging.

Overview of NSAPI Functions for Filter Development

NSAPI provides a set of C functions that are used to implement SAFs and filters. This section lists the functions that are specific to the development of filters. The public routines are described in detail in Chapter 4, "NSAPI Function Reference."

The NSAPI functions specific to the development of filters are:

- "filter_create" on page 91 — Creates a new filter
- "filter_insert" on page 92 — Inserts the specified filter into a filter stack
- "filter_remove" on page 94 — Removes the specified filter from a filter stack
- "filter_name" on page 94 — Returns the name of the specified filter
- "filter_find" on page 92 — Finds an existing filter given a filter name
- "filter_layer" on page 93 — Returns the layer in a filter stack that corresponds to the specified filter

CHAPTER 3

Examples of Custom SAFs and Filters

This chapter provides examples of custom Server Application Functions (SAFs) and filters for each directive in the request-response process. You can use these examples as the basis for implementing your own custom SAFs and filters. For more information about creating your own custom SAFs, see Chapter 2, “Creating Custom Filters”

Before writing custom SAFs, you should be familiar with the request-response process and the role of the `obj.conf` configuration file. This file is discussed in *Oracle iPlanet Web Proxy Server 4.0.14 Configuration File Reference*.

Before writing your own SAF, whether see if an existing SAF serves your purpose. The predefined SAFs are discussed in *Oracle iPlanet Web Proxy Server 4.0.14 Configuration File Reference*.

For a list of the NSAPI functions for creating new SAFs, see Chapter 4, “NSAPI Function Reference”

This chapter contains the following sections:

- “Examples in the Build” on page 52
- “AuthTrans Example” on page 52
- “NameTrans Example” on page 55
- “PathCheck Example” on page 58
- “ObjectType Example” on page 61
- “Output Example” on page 63
- “Service Example” on page 69
- “AddLog Example” on page 72

Examples in the Build

The `plugins/nsapi/examples` subdirectory within the server installation directory contains examples of source code for SAFs.

You can use the `example.mak` makefile in the same directory to compile the examples and create a library containing the functions in all of the example files.

To test an example, load the `examples` shared library into Proxy Server by adding the following directive in the `Init` section of `obj.conf`:

```
Init fn=load-modules shlib=examples.so/dll
funcs=
        function1,function2,function3
```

The `funcs` parameter specifies the functions to load from the shared library.

If the example uses an initialization function, specify the initialization function in the `funcs` argument to `load-modules`. Also, add an `Init` directive to call the initialization function.

For example, the `PathCheck` example implements the `restrict-by-acf` function, which is initialized by the `acf-init` function. The following directive loads both these functions:

```
Init fn=load-modules yourlibrary funcs=acf-init,restrict-by-acf
```

The following directive calls the `acf-init` function during server initialization:

```
Init fn=acf-init file=extra-arg
```

To invoke the new SAF at the appropriate step in the response handling process, add an appropriate directive in the object to which it applies, for example:

```
PathCheck fn=restrict-by-acf
```

After adding new `Init` directives to `obj.conf`, restart Proxy Server to load the changes `Init` directives are only applied during server initialization.

AuthTrans Example

This simple example of an `AuthTrans` function demonstrates how to use custom methods to verify that the user name and password that a remote client provided is accurate. This program uses a hard-coded table of user names and passwords and checks a given user's password against the one in the static data array. The *userdb* parameter is not used in this function.

AuthTrans directives work in conjunction with PathCheck directives. An AuthTrans function checks whether the user name and password associated with the request are acceptable. This directory does not allow or deny access to the request. Access is handled by a PathCheck function.

AuthTrans functions get the user name and password from the headers associated with the request. When a client initially makes a request, the user name and password are unknown. The AuthTrans function and PathCheck function therefore together to reject the request, because they can't validate the user name and password. When the client receives the rejection, the usual response is to present a dialog box asking the user for the user name and password. The client then submits the request again, this time including the user name and password in the headers.

In this example, the hardcoded-auth function, which is invoked during the AuthTrans step, checks whether the user name and password correspond to an entry in the hard-coded table of users and passwords.

Installing the AuthTrans Example

To install the function on Proxy Server, add the following Init directive to obj.conf to load the compiled function:

```
Init fn=load-modules shlib=yourlibrary funcs=hardcoded-auth
```

Inside the default object in obj.conf, add the following AuthTrans directive:

```
AuthTrans fn=basic-auth auth-type="basic" userfn=hardcoded-auth
userdb=unused
```

This function does not enforce authorization requirements. The function only takes given information and informs the server whether the information is correct or not. The PathCheck function require-auth performs the enforcement, so add the following PathCheck directive as well:

```
PathCheck fn=require-auth realm="test realm" auth-type="basic"
```

AuthTrans Example Source Code

The source code for this example is in the auth.c file in the nsapi/examples/ or plugins/nsapi/examples subdirectory of the server root directory.

```
#include "nsapi.h"
typedef struct {
    char *name;
```

```
    char *pw;
} user_s;

static user_s user_set[] = {
    {"joe", "shmoe"},
    {"suzy", "creamcheese"},
    {NULL, NULL}
};

#include "frame/log.h"


#ifdef __cplusplus
extern "C"
#endif
NSAPI_PUBLIC int hardcoded_auth(pblock *param, Session *sn, Request *rq)
{
    /* Parameters given to us by auth-basic */
    char *pwfile = pblock_findval("userdb", param);
    char *user = pblock_findval("user", param);
    char *pw = pblock_findval("pw", param);

    /* Temp variables */
    register int x;

    for(x = 0; user_set[x].name != NULL; ++x) {
        /* If this isn't the user we want, keep going */
        if(strcmp(user, user_set[x].name) != 0) continue;

        /* Verify password */
        if(strcmp(pw, user_set[x].pw)) {
             log_error(LOG_SECURITY, "hardcoded-auth", sn, rq,
                    "user %s entered wrong password", user);
            /* This will cause the enforcement function to ask */
            /* user again */
            return REQ_NOACTION;
        }
        /* If we return REQ_PROCEED, the username will be accepted */
        return REQ_PROCEED;
    }
    /* No match, have it ask them again */
    log_error(LOG_SECURITY, "hardcoded-auth", sn, rq,
        "unknown user %s", user);
    return REQ_NOACTION;
}
```

NameTrans Example

The `ntrans.c` file in the `plugins/nsapi/examples` subdirectory of the server root directory contains source code for two example `NameTrans` functions:

- `explicit_pathinfo`

  This example allows the use of explicit extra path information in a URL.
- `https_redirect`

  This example redirects the URL if the client is a particular version of Netscape Navigator.

This section discusses the first example. The source code in `ntrans.c` provides the second example.

Note – A `NameTrans` function is used primarily to convert the logical URL in `ppath` in `rq->vars` to a physical path name. However, in the example `explicit_pathinfo` does not translate the URL into a physical path name. It changes the value of the requested URL. See the second example, `https_redirect`, in `ntrans.c` for an example of a `NameTrans` function that converts the value of `ppath` in `rq->vars` from a URL to a physical path name.

The `explicit_pathinfo` example allows URLs to explicitly include extra path information for use by a CGI program. The extra path information is delimited from the main URL by a specified separator, such as a comma.

For example:

```
http://server-name/cgi/marketing,/jan/releases/hardware
```

In this case, the URL of the requested resource, which would be a CGI program, is `http://`*server-name*`/cgi/marketing`, and the extra path information to give to the CGI program is `/jan/releases/hardware`.

When choosing a separator, use a character that will never be used as part of the real URL.

The `explicit_pathinfo` function reads the URL, the text following the comma, and puts it in the `path-info` field of the `vars` field in the `request` object (`rq->vars`). CGI programs can access this information through the `PATH_INFO` environment variable.

When `explicit_pathinfo` is used the separator character is added to the end of the `SCRIPT_NAME` CGI environment variable.

`NameTrans` directives usually return `REQ_PROCEED` when they change the path, so that the server does not process any more `NameTrans` directives. However, in this case name translation should continue after the path information has been extracted , because the URL has not yet been translated to a physical path name.

Installing the NameTrans Example

To install the function on Proxy Server, add the following Init directive to obj.conf to load the compiled function:

```
Init fn=load-modules shlib=your-library funcs=explicit-pathinfo
```

Inside the default object in obj.conf, add the following NameTrans directive:

```
NameTrans fn=explicit-pathinfo separator=","
```

This NameTrans directive should appear before other NameTrans directives in the default object.

NameTrans Example Source Code

The following example is located in the ntrans.c file in the plugins/nsapi/examples subdirectory of the server root directory.

```
#include "nsapi.h"
#include <string.h>           /* strchr */
#include "frame/log.h"         /* log_error */
#ifdef __cplusplus
extern "C"
#endif
NSAPI_PUBLIC int explicit_pathinfo(pblock *pb, Session *sn, Request *rq)
{
    /* Parameter: The character to split the path by */
    char *sep = pblock_findval("separator", pb);
    /* Server variables */
    char *ppath = pblock_findval("ppath", rq->vars);
    /* Temp var */
    char *t;
    /* Verify correct usage */
    if(!sep) {
       log_error(LOG_MISCONFIG, "explicit-pathinfo", sn, rq,
           "missing parameter (need root)");
       /*     When we abort, the default status code is 500 Server
          Error */
       return REQ_ABORTED;
    }
    /* Check for separator. If not there, don't do anything */
    t = strchr(ppath, sep[0]);
   if(!t)
       return REQ_NOACTION;
    /* Truncate path at the separator */
    *t++ = '\\0';
```

```
    /* Assign path information */
    pblock_nvinsert("path-info", t, rq->vars);
    /*     Normally NameTrans functions return REQ_PROCEED when they
        change the path. However, we want name translation to
       continue after we're done. */
   return REQ_NOACTION;
}
#include "base/util.h"        /* is_mozilla */
#include "frame/protocol.h"   /* protocol_status */
#include "base/shexp.h"        /* shexp_cmp */
#ifdef __cplusplus
extern "C"
#endif
NSAPI_PUBLIC int https_redirect(pblock *pb, Session *sn, Request *rq)
{
    /* Server Variable */
    char *ppath = pblock_findval("ppath", rq->vars);
    /* Parameters */
    char *from = pblock_findval("from", pb);
    char *url = pblock_findval("url", pb);
    char *alt = pblock_findval("alt", pb);
    /* Work vars */
    char *ua;
    /* Check usage */
    if((!from) || (!url)) {
       log_error(LOG_MISCONFIG, "https-redirect", sn, rq,
          "missing parameter (need from, url)");
       return REQ_ABORTED;
    }
    /*     Use wildcard match to see if this path is one we should
       redirect */
   if(shexp_cmp(ppath, from) != 0)
       return REQ_NOACTION;   /* no match */
    /*     Sigh. The only way to check for SSL capability is to
        check UA */
    if(request_header("user-agent", &ua, sn, rq) == REQ_ABORTED)
       return REQ_ABORTED;
    /*     The is_mozilla function checks for Mozilla version 0.96
        or greater */
    if(util_is_mozilla(ua, "0", "96")) {
       /* Set the return code to 302 Redirect */
       protocol_status(sn, rq, PROTOCOL_REDIRECT, NULL);
       /*     The error handling functions use this to set the
           Location: */
       pblock_nvinsert("url", url, rq->vars);
       return REQ_ABORTED;
    }
    /* No match. Old client. */
```

```
    /* If there is an alternate document specified, use it. */
    if(alt) {
       pb_param *pp = pblock_find("ppath", rq->vars);
       /* Trash the old value */
       FREE(pp->value);
       /*     We must dup it because the library will later free
           this pblock */
       pp->value = STRDUP(alt);
       return REQ_PROCEED;
    }
    /* Else do nothing */
    return REQ_NOACTION;
}
```

PathCheck Example

The example in this section demonstrates how to implement a custom SAF for performing path checks. This example checks whether the requesting host is on a list of allowed hosts.

The `Init` function `acf-init` loads a file containing a list of allowable IP addresses with one IP address per line. The `PathCheck` function `restrict_by_acf` gets the IP address of the host that is making the request and checks whether it is on the list. If the host is on the list, it is allowed access; otherwise, access is denied.

For simplicity, the stdio library is used to scan the IP addresses from the file.

Installing the PathCheck Example

To load the shared object containing your functions, add the following line in the `Init` section of the `obj.conf` file:

`Init fn=load-modules` *yourlibrary* `funcs=acf-init,restrict-by-acf`

To call `acf-init` to read the list of allowable hosts, add the following line to the `Init` section in `obj.conf`. This line must appears after the line that loads the library containing `acf-init`).

`Init fn=acf-init file=`*fileContainingHostsList*

To execute your custom SAF during the request-response process for some object, add the following line to that object in the `obj.conf` file:

```
PathCheck fn=restrict-by-acf
```

PathCheck Example Source Code

The source code for this example is located in `pcheck.c` in the `plugins/nsapi/examples` subdirectory within the server root directory.

```
#include "nsapi.h"
/*     Set to NULL to prevent problems with people not calling
    acf-init */
static char **hosts = NULL;
#include <stdio.h>
#include "base/daemon.h"
#include "base/util.h"      /* util_sprintf */
#include "frame/log.h"      /* log_error */
#include "frame/protocol.h" /* protocol_status */
/* The longest line we'll allow in an access control file */
#define MAX_ACF_LINE 256
/* Used to free static array on restart */
#ifdef __cplusplus
extern "C"
#endif
NSAPI_PUBLIC void acf_free(void *unused)
{
    register int x;
    for(x = 0; hosts[x]; ++x)
        FREE(hosts[x]);
    FREE(hosts);
    hosts = NULL;
}
#ifdef __cplusplus
extern "C"
#endif
NSAPI_PUBLIC int acf_init(pblock *pb, Session *sn, Request *rq)
{
    /* Parameter */
    char *acf_file = pblock_findval("file", pb);
    /* Working variables */
    int num_hosts;
    FILE *f;
    char err[MAGNUS_ERROR_LEN];
    char buf[MAX_ACF_LINE];
    /*     Check usage. Note that Init functions have special
         error logging */
    if(!acf_file) {
        util_sprintf(err, "missing parameter to acf_init
             (need file)");
        pblock_nvinsert("error", err, pb);
        return REQ_ABORTED;
    }
```

```
    f = fopen(acf_file, "r");
    /* Did we open it? */
    if(!f) {
        util_sprintf(err, "can't open access control file %s (%s)",
            acf_file, system_errmsg());
        pblock_nvinsert("error", err, pb);
        return REQ_ABORTED;
    }
    /* Initialize hosts array */
    num_hosts = 0;
    hosts = (char **) MALLOC(1 * sizeof(char *));
    hosts[0] = NULL;
    while(fgets(buf, MAX_ACF_LINE, f)) {
        /* Blast linefeed that stdio helpfully leaves on there */
        buf[strlen(buf) - 1] = '\\0';
        hosts = (char **) REALLOC(hosts, (num_hosts + 2) *
             sizeof(char *));
        hosts[num_hosts++] = STRDUP(buf);
        hosts[num_hosts] = NULL;
    }
    fclose(f);
    /* At restart, free hosts array */
    daemon_atrestart(acf_free, NULL);
    return REQ_PROCEED
}
#ifdef __cplusplus
extern "C"
#endif
NSAPI_PUBLIC int restrict_by_acf(pblock *pb, Session *sn, Request *rq)
{
    /* No parameters */
    /* Working variables */
    char *remip = pblock_findval("ip", sn->client);
    register int x;
    if(!hosts) {
        log_error(LOG_MISCONFIG, "restrict-by-acf", sn, rq,
             "restrict-by-acf called without call to acf-init");
        /*     When we abort, the default status code is 500 Server
             Error */
        return REQ_ABORTED;
    }
    for(x = 0; hosts[x] != NULL; ++x) {
        /* If they're on the list, they're allowed */
        if(!strcmp(remip, hosts[x]))
        return REQ_NOACTION;
    }
    /* Set response code to forbidden and return an error. */
    protocol_status(sn, rq, PROTOCOL_FORBIDDEN, NULL);
```

```
    return REQ_ABORTED;
}
```

ObjectType Example

The example in this section demonstrates how to implement `html2shtml`, a custom SAF that instructs the server to treat a `.html` file as an `.shtml` file if an `.shtml` version of the requested file exists.

The `ObjectType` function checks whether the content type is already set. If the type is set, returns ObjectType `REQ_NOACTION`.

```
if(pblock_findval("content-type", rq->srvhdrs))
    return REQ_NOACTION;
```

The primary task an `ObjectType` directive needs to perform is to set the content type if it is not already set. This example sets the content type to `magnus-internal/parsed-html` in the following lines:

```
/* Set the content-type to magnus-internal/parsed-html */
pblock_nvinsert("content-type", "magnus-internal/parsed-html",
    rq->srvhdrs);
```

The `html2shtml` function checks at the requested file name. If the filename ends with `.html`, the function checks for a file with the same base name but with the extension `.shtml` instead. If such a file is found, the function uses that path and informs the server that the file is parsed HTML instead of regular HTML. This process requires an extra `stat` call for every HTML file accessed.

Installing the ObjectType Example

To load the shared object containing your function, add the following line in the `Init` section of the `obj.conf` file:

```
Init fn=load-modules shlib=yourlibrary funcs=html2shtml
```

To execute the custom SAF during the request-response process for some object, add the following line to that object in the `obj.conf` file:

```
ObjectType fn=html2shtml
```

ObjectType Example Source Code

The source code for this example is in `otype.c` in the `nsapi/examples/` or `plugins/nsapi/examples` subdirectory within the server root directory.

```
#include "nsapi.h"
#include <string.h>    /* strncpy */
#include "base/util.h"

#ifdef __cplusplus
extern "C"
#endif
NSAPI_PUBLIC int html2shtml(pblock *pb, Session *sn, Request *rq)
{
    /* No parameters */

    /* Work variables */
    pb_param *path = pblock_find("path", rq->vars);
    struct stat finfo;
    char *npath;
    int baselen;

    /* If the type has already been set, don't do anything */
    if(pblock_findval("content-type", rq->srvhdrs))
        return REQ_NOACTION;

    /* If path does not end in .html, let normal object types do
     * their job */
    baselen = strlen(path->value) - 5;
    if(strcasecmp(&path->value[baselen], ".html") != 0)
        return REQ_NOACTION;

    /* 1 = Room to convert html to shtml */
    npath = (char *) MALLOC((baselen + 5) + 1 + 1);
    strncpy(npath, path->value, baselen);
    strcpy(&npath[baselen], ".shtml");

    /* If it's not there, don't do anything */
    if(stat(npath, &finfo) == -1) {
        FREE(npath);
        return REQ_NOACTION;
    }
    /* Got it, do the switch */
    FREE(path->value);
    path->value = npath;

    /* The server caches the stat() of the current path. Update it. */
    (void) request_stat_path(NULL, rq);
```

```
    pblock_nvinsert("content-type", "magnus-internal/parsed-html",
                     rq->srvhdrs);
    return REQ_PROCEED;
}
```

Output Example

This section describes an example NSAPI filter named `example-replace`, which examines outgoing data and substitutes one string for another. The example shows how you can create a filter that intercepts and modifies outgoing data.

Installing the Output Example

To load the filter, add the following line in the `Init` section of the `obj.conf` file:

```
Init fn="load-modules" shlib="<path>/replace.
            ext" NativeThread="no"
```

To execute the filter during the request-response process for some object, add the following line to that object in the `obj.conf` file:

```
Output fn="insert-filter" type="text/*" filter="example-replace"
    from="iPlanet" to="Sun ONE"
```

Output Example Source Code

The source code for this example is in the `replace.c` file in the `plugins/nsapi/examples` subdirectory of the server root directory.

```
#ifdef XP_WIN32
#define NSAPI_PUBLIC __declspec(dllexport)
#else /* !XP_WIN32 */
#define NSAPI_PUBLIC
#endif /* !XP_WIN32 */

/*
  * nsapi.h declares the NSAPI interface.
 */
#include "nsapi.h"
```

```
/* -------------------ExampleReplaceData------------------------ */

/*
 * ExampleReplaceData will be used to store information between
 * filter method invocations. Each instance of the example-replace
 * filter will have its own ExampleReplaceData object.
*/

typedef struct ExampleReplaceData ExampleReplaceData;

struct ExampleReplaceData {
    char *from;   /* the string to replace */
    int fromlen;  /* length of "from" */
    char *to;     /* the string to replace "from" with */
    int tolen;    /* length of "to" */
    int matched;  /* number of "from" chars matched */
};


/* ------------- example_replace_insert ----------------------- */

/*
 * example_replace_insert implements the example-replace filter's
 * insert method. The insert filter method is called before the
 * server adds the filter to the filter stack.
*/

#ifdef __cplusplus
extern "C"
#endif
int example_replace_insert(FilterLayer *layer, pblock *pb)
{
    const char *from;
    const char *to;
    ExampleReplaceData *data;

    /*
     * Look for the string to replace, "from", and the string to
     * replace it with, "to". Both values are required.
    */
    from = pblock_findval("from", pb);
    to = pblock_findval("to", pb);
    if (from == NULL || to == NULL || strlen(from) < 1) {
        log_error(LOG_MISCONFIG, "example-replace-insert",
                  layer->context->sn, layer->context->rq,
                  "missing parameter (need from and to)");
        return REQ_ABORTED; /* error preparing for insertion */
```

```
    }

    /*
     * Allocate an ExampleReplaceData object that will store
     * configuration and state information.
     */
    data = (ExampleReplaceData *)MALLOC(sizeof(ExampleReplaceData));
    if (data == NULL)
        return REQ_ABORTED; /* error preparing for insertion */

    /* Initialize the ExampleReplaceData */
    data->from = STRDUP(from);
    data->fromlen = strlen(from);
    data->to = STRDUP(to);
    data->tolen = strlen(to);
    data->matched = 0;

    /* Check for out of memory errors */
    if (data->from == NULL || data->to == NULL) {
        FREE(data->from);
        FREE(data->to);
        FREE(data);
        return REQ_ABORTED; /* error preparing for insertion */
    }

    /*
     * Store a pointer to the ExampleReplaceData object in the
     * FilterLayer. This information can then be accessed from other
     * filter methods.
     */
    layer->context->data = data;

    /* Remove the Content-length: header if we might change the
     * body length */
    if (data->tolen != data->fromlen) {
        pb_param *pp;
        pp = pblock_remove("content-length", layer->context->rq->srvhdrs);
        if (pp)
             param_free(pp);
    }

    return REQ_PROCEED; /* insert filter */
}


/* -------------- example_replace_remove ------------------------ */

/*
```

```
 * example_replace_remove implements the example-replace filter’s
 * remove method. The remove filter method is called before the
 * server removes the filter from the filter stack.
 */

#ifdef __cplusplus
extern "C"
#endif
void example_replace_remove(FilterLayer *layer)
{
    ExampleReplaceData *data;

    /* Access the ExampleReplaceData we allocated in
        example_replace_insert */
    data = (ExampleReplaceData *)layer->context->data;

    /* Send any partial "from" match */
    if (data->matched > 0)
        net_write(layer->lower, data->from, data->matched);

    /* Destroy the ExampleReplaceData object */
    FREE(data->from);
    FREE(data->to);
    FREE(data);
}


/* ------------- example_replace_write ------------------------ */

/*
* example_replace_write implements the example-replace filter’s
* write method. The write filter method is called when there is data
* to be sent to the client.
*/

#ifdef __cplusplus
extern "C"
#endif
int example_replace_write(FilterLayer *layer, const void *buf, int amount)
{
    ExampleReplaceData *data;
    const char *buffer;
    int consumed;
    int i;
    int unsent;
    int rv;

    /* Access the ExampleReplaceData we allocated in
```

```
    example_replace_insert */
data = (ExampleReplaceData *)layer->context->data;

/* Check for "from" matches in the caller's buffer */
buffer = (const char *)buf;
consumed = 0;
for (i = 0; i < amount; i++) {
    /* Check whether this character matches */
    if (buffer[i] == data->from[data->matched]) {
         /* Matched a(nother) character */
        data->matched++;

        /* If we've now matched all of "from"... */
        if (data->matched == data->fromlen) {
            /* Send any data that preceded the match */
            unsent = i + 1 - consumed - data->matched;
             if (unsent > 0) {
                rv = net_write(layer->lower, &buffer[consumed], unsent);
                if (rv != unsent)
                     return IO_ERROR;
            }

            /* Send "to" in place of "from" */
            rv = net_write(layer->lower, data->to, data->tolen);
            if (rv != data->tolen)
                 return IO_ERROR;

            /* We've handled up to and including buffer[i] */
             consumed = i + 1;

            /* Start looking for the next "from" match from scratch */
            data->matched = 0;
        }

    } else if (data->matched > 0) {
        /* This match didn't pan out, we need to backtrack */
        int j;
        int backtrack = data->matched;
        data->matched = 0;

        /* Check for other potential "from" matches
         * preceding buffer[i] */
        for (j = 1; j < backtrack; j++) {
            /* Check whether this character matches */
            if (data->from[j] == data->from[data->matched]) {
                /* Matched a(nother) character */
                data->matched++;
```

```
                } else if (data->matched > 0) {
                   /* This match didn't pan out, we need to
                    * backtrack */
                    j -= data->matched;
                    data->matched = 0;
                }
            }

            /* If the failed (partial) match begins before the buffer... */
             unsent = backtrack - data->matched;
             if (unsent > i) {
                /* Send the failed (partial) match */
                rv = net_write(layer->lower, data->from, unsent);
                if (rv != unsent)
                    return IO_ERROR;

                /* We've handled up to, but not including,
                 * buffer[i] */
                consumed = i;
            }

            /* We're not done with buffer[i] yet */
            i--;
        }
    }

    /* Send any data we know won't be part of a future
     * "from" match */
    unsent = amount - consumed - data->matched;
    if (unsent > 0) {
        rv = net_write(layer->lower, &buffer[consumed], unsent);
        if (rv != unsent)
             return IO_ERROR;
    }

    return amount;
}


/* ---------------- nsapi_module_init --------------------------- */

/*
 * This is the module initialization entry point for this NSAPI
 * plugin. The server calls this entry point in response to the
 * Init fn="load-modules" line in magnus.conf.
 */

NSAPI_PUBLIC nsapi_module_init(pblock *pb, Session *sn, Request *rq)
```

```
{
    FilterMethods methods = FILTER_METHODS_INITIALIZER;
    const Filter *filter;

    /*
     * Create the example-replace filter. The example-replace filter
     * has order FILTER_CONTENT_TRANSLATION, meaning it transforms
     * content (entity body data) from one form to another. The
     * example-replace filter implements the write filter method,
     * meaning it is interested in outgoing data.
     */
    methods.insert = &example_replace_insert;
    methods.remove = &example_replace_remove;
    methods.write = &example_replace_write;
    filter = filter_create("example-replace",
                           FILTER_CONTENT_TRANSLATION,
                            &methods);
    if (filter == NULL) {
        pblock_nvinsert("error", system_errmsg(), pb);
        return REQ_ABORTED; /* error initializing plugin */
    }

    return REQ_PROCEED; /* success */
}
```

Service Example

This section discusses a very simple `Service` function called `simple_service`. This function sends a message in response to a client request. The message is initialized by the `init_simple_service` function during server initialization.

For a more complex example, see the file `service.c` in the `examples` directory, which is discussed in "More Complex Service Example" on page 71

Installing the Service Example

To load the shared object containing your functions, add the following line in the `Init` section of the `obj.conf` file:

```
Init fn=load-modules shlib=
            yourlibrary funcs=simple-service-init,simple-service
```

To call the `simple-service-init` function to initialize the message representing the generated output, add the following line to the `Init` section in `obj.conf`. This line must appear after the line that loads the library containing `simple-service-init`.)

```
Init fn=simple-service-init
 generated-output="<H1>
            Generated output msg</H1>"
```

To execute the custom SAF during the request-response process for an object, add the following line to that object in the `obj.conf` file:

```
Service type="text/html" fn=simple-service
```

The `type="text/html"` argument indicates that this function is invoked during the `Service` stage only if the `content-type` has been set to `text/html`.

Service Example Source Code

```
#include <nsapi.h>
static char *simple_msg = "default customized content";
/* This is the initialization function.
 * It gets the value of the generated-output parameter
 * specified in the Init directive in magnus.conf
*/
NSAPI_PUBLIC int init-simple-service(pblock *pb, Session *sn,
 Request *rq)
{
    /* Get the message from the parameter in the directive in
      * magnus.conf
     */
    simple_msg = pblock_findval("generated-output", pb);
    return REQ_PROCEED;
}
/* This is the customized Service SAF.
 * It sends the "generated-output" message to the client.
*/
NSAPI_PUBLIC int simple-service(pblock *pb, Session *sn, Request *rq)
 {
     int return_value;
    char msg_length[8];
    /* Use the protocol_status function to set the status of the
     * response before calling protocol_start_response.
    */
    protocol_status(sn, rq, PROTOCOL_OK, NULL);
```

```
    /* Although we would expect the ObjectType stage to
     * set the content-type, set it here just to be
    * completely sure that it gets set to text/html.
    */
    param_free(pblock_remove("content-type", rq->srvhdrs));
    pblock_nvinsert("content-type", "text/html", rq->srvhdrs);
    /* If you want to use keepalive, need to set content-length header.
    * The util_itoa function converts a specified integer to a
     * string, and returns the length of the string. Use this
     * function to create a textual representation of a number.
    */
    util_itoa(strlen(simple_msg), msg_length);
    pblock_nvinsert("content-length", msg_length, rq->srvhdrs);
    /* Send the headers to the client*/
    return_value = protocol_start_response(sn, rq);
     if (return_value == REQ_NOACTION) {
         /* HTTP HEAD instead of GET */
         return REQ_PROCEED;
     }
    /* Write the output using net_write*/
    return_value = net_write(sn->csd, simple_msg,
         strlen(simple_msg));
     if (return_value == IO_ERROR) {
         return REQ_EXIT;
     }
    return REQ_PROCEED;
}
```

More Complex Service Example

The `send-images` function is a custom SAF that replaces the `doit.cgi` demonstration available on the iPlanet home pages. When a file is accessed as `/dir1/dir2/something.picgroup`, the `send-images` function checks whether the file is being accessed by a Mozilla/1.1 browser. If the file is not being accessed, the function sends a short error message. The file `something.picgroup` contains a list of lines, each of which specifies a file name followed by a `content-type`, for example, `one.gif image/gif`.

To load the shared object containing your function, add the following line at the beginning of the `obj.conf` file:

```
Init fn=load-modules shlib=yourlibrary funcs=send-images
```

Also, add the following line to the `mime.types` file:

```
type=magnus-internal/picgroup exts=picgroup
```

To execute the custom SAF during the request-response process for an object, add the following line to that object in the obj.conf file. send-images takes an optional parameter, delay, which is not used for this example.

```
Service method=(GET|HEAD) type=magnus-internal/picgroup fn=send-images
```

The source code is located in service.c in the plugins/nsapi/examples subdirectory within the server root directory.

AddLog Example

The example in this section demonstrates how to implement brief-log, a custom SAF for logging only three items of information about a request: the IP address, the method, and the URI, for example, 198.93.95.99 GET /jocelyn/dogs/homesneeded.html.

Installing the AddLog Example

To load the shared object containing your functions, add the following line in the Init section of the magnus.conf file:

```
Init fn=load-modules shlib=yourlibrary funcs=brief-init,brief-log
```

To call brief-init to open the log file, add the following line to the Init section in obj.conf. This line must appear after the line that loads the library containing brief-init.)

```
Init fn=brief-init file=/tmp/brief.log
```

To execute your custom SAF during the AddLog stage for an object, add the following line to that object in the obj.conf file:

```
AddLog fn=brief-log
```

AddLog Example Source Code

The source code is in addlog.c is in the plugins/nsapi/examples subdirectory within the server root directory.

```
#include "nsapi.h"
#include "base/daemon.h" /* daemon_atrestart */
#include "base/file.h"   /* system_fopenWA, system_fclose */
```

```
#include "base/util.h"   /* sprintf */


/* File descriptor to be shared between the processes */

static SYS_FILE logfd = SYS_ERROR_FD;

#ifdef __cplusplus
extern "C"
#endif
NSAPI_PUBLIC void brief_terminate(void *parameter)
{
    system_fclose(logfd);
    logfd = SYS_ERROR_FD;
}

#ifdef __cplusplus
extern "C"
#endif
NSAPI_PUBLIC int brief_init(pblock *pb, Session *sn, Request *rq)
{
    /* Parameter */
    char *fn = pblock_findval("file", pb);

    if(!fn) {
        pblock_nvinsert("error", "brief-init: please supply a file name",
            pb); return REQ_ABORTED;
    }
    logfd = system_fopenWA(fn);
    if(logfd == SYS_ERROR_FD) {
        pblock_nvinsert("error", "brief-init: please supply a file name",
            pb);return REQ_ABORTED;
    }
    /* Close log file when server is restarted */
    daemon_atrestart(brief_terminate, NULL);
    return REQ_PROCEED;
}

#ifdef __cplusplus
extern "C"
#endif
NSAPI_PUBLIC int brief_log(pblock *pb, Session *sn, Request *rq)
{
    /* No parameters */

    /* Server data */
    char *method = pblock_findval("method", rq->reqpb);
    char *uri = pblock_findval("uri", rq->reqpb);
```

```
    char *ip = pblock_findval("ip", sn->client);

    /* Temp vars */
    char *logmsg;
    int len;

    logmsg = (char *)
        MALLOC(strlen(ip) + 1 + strlen(method) + 1 + strlen(uri) + 1 + 1);
    len = util_sprintf(logmsg, "%s %s %s\\n", ip, method, uri);
    /* The atomic version uses locking to prevent interference */
    system_fwrite_atomic(logfd, logmsg, len);
    FREE(logmsg);

    return REQ_PROCEED;
}
```

CHAPTER 4

NSAPI Function Reference

This chapter lists the public C functions and macros of the Netscape Server Applications Programming Interface (NSAPI) in alphabetic order. You use these functions when writing your own Server Application Functions (SAFs).

For information about the predefined SAFs used in `obj.conf`, see *Oracle iPlanet Web Proxy Server 4.0.14 Configuration File Reference*.

Each function provides the name, syntax, parameters, return value, a description of what the function does, and sometimes an example of its use and a list of related functions.

For more information on data structures, see Chapter 5, "Data Structure Reference." Also look in the `nsapi.h` header file in the `include` directory in the build for iPlanet Web Proxy Server 4.0.14.

NSAPI Functions (in Alphabetical Order)

For an alphabetical list of function names, see Appendix A, "Alphabetical List of NSAPI Functions and Macros."

"C" on page 76	"I" on page 99	"N" on page 102	"S" on page 136
"D" on page 85	"L" on page 100	"P" on page 111	"U" on page 161
"F" on page 86	"M" on page 101	"R" on page 132	"W" on page 190

C

cache_digest

The cache_digest function calculates the MD5 signature of a specified URL and stores the signature in a digest variable.

Syntax

```
#include <libproxy/cache.h>
void cache_digest(char *url, unsigned char digest[16]));
```

Returns

void

Parameters

char **url* is a string containing the cache file name of a URL.

name **digest* is an array to store the MD5 signature of the URL.

See Also

"cache_fn_to_dig" on page 77

cache_filename

The cache_filename function returns the cache file name for a given URL, specified by the MD5 signature.

Syntax

```
#include <libproxy/cutil.h>
char *cache_filename(unsigned char digest[16]);
```

Returns

A new string containing the cache filename.

Parameters

char **digest* is an array containing the MD5 signature of a URL.

See Also

"cache_fn_to_dig" on page 77

cache_fn_to_dig

The cache_fn_to_dig function converts a cache file name of a URL into a partial MD5 digest.

Syntax

```
#include <libproxy/cutil.h>
void *cache_fn_to_dig(char *name, unsigned char digest[16]));
```

Returns

void

Parameters

char *name is a string containing the cache file name of a URL.

name *digest is an array to receive first 8 bits of the signature of the URL.

CALLOC

The CALLOC macro is a platform-independent substitute for the C library routine calloc. It allocates num*size bytes from the request's memory pool. If pooled memory has been disabled in the configuration file with the pool-init built-in SAF, PERM_CALLOC and CALLOC both obtain their memory from the system heap.

Syntax

```
void *CALLOC(int size)
```

Returns

A void pointer to a block of memory.

Parameters

int size is the size in bytes of each element.

Example

```
char *name;name = (char *) CALLOC(100);
```

See Also

"FREE" on page 95, "REALLOC" on page 133, "STRDUP" on page 146, "PERM_MALLOC" on page 123, "PERM_FREE" on page 122, "PERM_REALLOC" on page 123, "PERM_STRDUP" on page 124

ce_free

The ce_free function releases memory allocated by the ce_lookup function.

Syntax

```
#include <libproxy/cache.h>
void cd_free(CacheEntry *ce);
```

Returns

void

Parameters

CacheEntry *ce is a cache entry structure to be destroyed.

See Also

"ce_lookup" on page 78

ce_lookup

The ce_lookup cache entry lookup function looks up a cache entry for a specified URL.

Syntax

```
#include <libproxy/cache.h>
CacheEntry *ce_lookup(Session *sn, Request *rq, char *url, time_t ims_c);
```

Returns

- NULL if caching is not enabled
- A newly allocated CacheEntry structure, whether or not a copy existed in the cache. Within that structure, the ce->state field reports about the existence:

  CACHE_NO signals that the document is not and will not be cached. Other fields in the cache structure may be NULL

  CACHE_CREATE signals that the cache file doesn't exist but may be created once the remote server is contacted. However, during the retrieval it may turn out that the document not be cacheable.

  CACHE_REFRESH signals that the cache file exists but must be refreshed before being used. The data might still be up to date but the remote server needs to be contacted to find out. If the file is not up to date, the cache file will be replaced with the new document version sent by the remote origin server.

`CACHE_RETURN_FROM_CACHE` signals that the cache file exists and is up-to-date based on the configuration and current parameters controlling what is considered fresh.

`CACHE_RETURN_ERROR` is a signal that happens only if the proxy is set to no-network mode `connect-Modenese`, and the document does not exist in the cache.

Parameters

`Session` **sn* identifies the Session structure.

`Request` **rq* identifies the Request structure.

`char` **url* contains the name of the URL for which the cache is being sought.

`time-out` *misc.* is the if-modified-since time.

See Also

"ce_free" on page 78

cif_write_entry

The `cif_write_entry` function writes a CIF entry for a specified `CacheEntry` structure. The CIF entry is stored in the cache file itself.

Syntax

```
#include <libproxy/cif.h>
int cif_write_entry(CacheEntry *ce,int new_cachefile)
```

Returns

- nonzero if the write was successful
- 0 if the write was unsuccessful

Parameters

`CacheEntry` **ce* is a cache entry structure to be written to the `.cif` file.

`int new_cachefile` The values are 1 or 0:

1 if the file is a new cache file

0 if the file exists and the CIF entry is to be modified

cinfo_find

The `cinfo_find()` function uses the MIME types information to find the type, encoding, and/or language based on the extensions of the Universal Resource Identifier (URI) or local file name. Use this information to send headers (`rq->srvhdrs`) to the client indicating the `content-type`, `content-encoding`, and `content-language` of the data that the client will be receiving from the server.

The name used consists of all of the text after the last slash (/) or the whole string if no slash is found. File name extensions are not case-sensitive. The name may contain multiple extensions separated by period (.) to indicate type, encoding, or language. For example, the URI `a/b/filename.jp.txt.zip` could represent a Japanese language, `text/plain` type, .zip-encoded file.

Syntax

```
cinfo *cinfo_find(char *uri);
```

Returns

A pointer to a newly allocated `cinfo` structure if content info was found, or NULL if no content was found.

The `cinfo` structure that is allocated and returned contains pointers to the `content-type`, `content-encoding`, and `content-language`, if found. Each is a pointer into static data in the types database, or NULL if not found. Do not free these pointers. You should free the `cinfo` structure when you are done using it.

Parameters

`char *uri` is a Universal Resource Identifier (URI) or local file name. Multiple file name extensions should be separated by periods (.).

condvar_init

The `condvar_init` function is a critical-section function that initializes and returns a new condition variable associated with a specified critical-section variable. You can use the condition variable to prevent interference between two threads of execution.

Syntax

```
CONDVAR condvar_init(CRITICAL id);
```

Returns

A newly allocated condition variable (`CONDVAR`).

Parameters

CRITICAL id is a critical-section variable.

See Also

"condvar_notify" on page 81, "condvar_terminate" on page 81, "condvar_wait" on page 82, "crit_init" on page 84, "crit_enter" on page 83, "crit_exit" on page 83, "crit_terminate" on page 84

condvar_notify

The condvar_notify function is a critical-section function that awakens any threads that are blocked on the given critical-section variable. Use this function to awaken threads of execution of a given critical section. First, use crit_enter to gain ownership of the critical section. Then use the returned critical-section variable to call condvar_notify to awaken the threads. Finally, when condvar_notify returns, call crit_exit to surrender ownership of the critical section.

Syntax

```
void condvar_notify(CONDVAR cv);
```

Returns

void

Parameters

CONDVAR cv is a condition variable.

See Also

"condvar_init" on page 80, "condvar_terminate" on page 81, "condvar_wait" on page 82, "crit_init" on page 84, "crit_enter" on page 83, "crit_exit" on page 83, "crit_terminate" on page 84

condvar_terminate

The condvar_terminate function is a critical-section function that frees a condition variable. Use this function to free a previously allocated condition variable.

Warning

Terminating a condition variable that is in use can lead to unpredictable results.

Syntax

```
void condvar_terminate(CONDVAR cv);
```

Returns

void

Parameters

CONDVAR cv is a condition variable.

See Also

"condvar_init" on page 80, "condvar_notify" on page 81, "condvar_wait" on page 82, "crit_init" on page 84, "crit_enter" on page 83, "crit_exit" on page 83, "crit_terminate" on page 84

condvar_wait

The condvar_wait function is a critical-section function that blocks on a given condition variable. Use this function to wait for a critical section specified by a condition variable argument to become available. The calling thread is blocked until another thread calls condvar_notify with the same condition variable argument. The caller must have entered the critical section associated with this condition variable before calling condvar_wait.

Syntax

```
void condvar_wait(CONDVAR cv);
```

Returns

void

Parameters

CONDVAR cv is a condition variable.

See Also

"condvar_init" on page 80, "condvar_terminate" on page 81, "condvar_notify" on page 81, "crit_init" on page 84, "crit_enter" on page 83, "crit_exit" on page 83, "crit_terminate" on page 84

crit_enter

The `crit_enter` function is a critical-section function that attempts to enter a critical section. Use this function to gain ownership of a critical section. If another thread already owns the section, the calling thread is blocked until the first thread surrenders ownership by calling `crit_exit`.

Syntax

```
void crit_enter(CRITICAL crvar);
```

Returns

`void`

Parameters

`CRITICAL crvar` is a critical-section variable.

See Also

"crit_init" on page 84, "crit_exit" on page 83, "crit_terminate" on page 84

crit_exit

The `crit_exit` function is a critical-section function that surrenders ownership of a critical section. Use this function to surrender ownership of a critical section. If another thread is blocked waiting for the section, the block will be removed and the waiting thread will be given ownership of the section.

Syntax

```
void crit_exit(CRITICAL crvar);
```

Returns

`void`

Parameters

`CRITICAL crvar` is a critical-section variable.

See Also

"crit_init" on page 84, "crit_enter" on page 83, "crit_terminate" on page 84

crit_init

The `crit_init` function is a critical-section function that creates and returns a new critical-section variable (a variable of type `CRITICAL`). Use this function to obtain a new instance of a variable of type `CRITICAL` (a critical-section variable) to be used in managing the prevention of interference between two threads of execution. At the time of its creation, no thread owns the critical section.

Warning

Threads must not own or be waiting for the critical section when `crit_terminate` is called.

Syntax

```
CRITICAL crit_init(void);
```

Returns

A newly allocated critical-section variable (`CRITICAL`).

Parameters

none

See Also

“crit_enter” on page 83, “crit_exit” on page 83, “crit_terminate” on page 84

crit_terminate

The `crit_terminate` function is a critical-section function that removes a previously allocated critical-section variable (a variable of type `CRITICAL`). Use this function to release a critical-section variable previously obtained by a call to `crit_init`.

Syntax

```
void crit_terminate(CRITICAL crvar);
```

Returns

`void`

Parameters

`CRITICAL crvar` is a critical-section variable.

See Also

"crit_init" on page 84, "crit_enter" on page 83, "crit_exit" on page 83

D

daemon_atrestart

The daemon_atrestart function enables you to register a callback function named by fn to be used when the server terminates. Use this function when you need a callback function to deallocate resources allocated by an initialization function. The daemon_atrestart function is a generalization of the magnus_atrestart function.

The magnus.conf directives TerminateTimeout and ChildRestartCallback also affect the callback of NSAPI functions.

Syntax

```
void daemon_atrestart(void (*fn)(void *), void *data);
```

Returns

void

Parameters

void (* fn) (void *) is the callback function.

void *data is the parameter passed to the callback function when the server is restarted.

Example

```
/* Register the log_close function, passing it NULL */ /* to close *a log
    file when the server is *//* restarted or shutdown.
    */daemon_atrestart(log_close, NULL);NSAPI_PUBLIC void log_close(void *parameter)
    {system_fclose(global_logfd);}
```

dns_set_hostent

The dns_set_hostent function sets the DNS host entry information in the request. If this function set, the proxy won't try to resolve host information by itself. Instead, the function will use this host information that was already resolved within custom DNS resolution SAF.

Syntax

```
int dns_set_hostent(struct hostent *hostent, Session *sn, Request *rq);
```

Returns

REQ_PROCEED on success or REQ_ABORTED on error.

Parameters

struct hostent *hostent is a pointer to the host entry structure.

Session *sn identifies the Session structure.

Request *rq identifies the Request structure.

Example

```
int my_dns_func(pblock *pb, Session *sn, Request *rq)
{
    char *host = pblock_findval("dns-host", rq->vars);
    struct hostent *hostent;
    hostent = gethostbyname(host);//replace with custom DNS implementation
    dns_set_hostent(hostent, sn, rq);
    return REQ_PROCEED;
}
```

F

fc_close

The fc_close function closes a file that was opened using fc_open. This function should only be called with files that were opened using fc_open.

Syntax

```
void fc_close(PRFileDesc *fd, FcHdl *hDl;
```

Returns

void

Parameters

PRFileDesc *fd is a valid pointer returned from a prior call to fc_open.

FcHdl *hDl is a valid pointer to a structure of type FcHdl. This pointer must have been initialized by a prior call to fc_open.

fc_open

The fc_open function returns a pointer to PRFileDesc that refers to an open file fileName. The fileName value must be the full path name of an existing file. The file is opened in read mode only. The application calling this function should not modify the currency of the file pointed to by the PRFileDesc * unless the DUP_FILE_DESC is also passed to this function. In other words, the application at minimum should not issue a read operation based on this pointer that would modify the currency for the PRFileDesc *. If such a read operation is required that might change the currency for the PRFileDesc * , then the application should call this function with the argument DUP_FILE_DESC.

On a successful call to this function, a valid pointer to PRFileDesc is returned and the handle 'FcHdl' is properly initialized. The size information for the file is stored in the 'fileSize' member of the handle.

Syntax

```
PRFileDesc *fc_open(const char *fileName, FcHdl *hDl,PRUint32 flags,
    Session *sn, Request *rq);
```

Returns

Pointer to PRFileDesc, or NULL on failure.

Parameters

const char *fileName is the full path name of the file to be opened.

FcHdl*hDl is a valid pointer to a structure of type FcHdl.

PRUint32 flags can be 0 or DUP_FILE_DESC.

Session *sn is a pointer to the session.

Request *rq is a pointer to the request.

filebuf_buf2sd

The filebuf_buf2sd function sends a file buffer to a socket (descriptor) and returns the number of bytes sent.

Use this function to send the contents of an entire file to the client.

Syntax

```
int filebuf_buf2sd(filebuf *buf, SYS_NETFD sd);
```

Returns

The number of bytes sent to the socket if successful, or the constant IO_ERROR if the file buffer could not be sent.

Parameters

filebuf *buf is the file buffer that must already have been opened.

SYS_NETFD sd is the platform-independent socket descriptor. Normally this will be obtained from the csd (client socket descriptor) field of the sn (session) structure.

Example

```
if (filebuf_buf2sd(buf, sn->csd) == IO_ERROR)    return(REQ_EXIT);
```

See Also

“filebuf_close” on page 88, “filebuf_open” on page 89, “filebuf_open_nostat” on page 90, “filebuf_getc” on page 89

filebuf_close

The filebuf_close function deallocates a file buffer and closes its associated file.

Use filebuf_open first to open a file buffer, and then filebuf_getc to access the information in the file. After you have finished using the file buffer, use filebuf_close to close it.

Syntax

```
void filebuf_close(filebuf *buf);
```

Returns

void

Parameters

filebuf *buf is the file buffer previously opened with filebuf_open.

Example

```
filebuf_close(buf);
```

See Also

“filebuf_open” on page 89, “filebuf_open_nostat” on page 90, “filebuf_buf2sd” on page 87, “filebuf_getc” on page 89

filebuf_getc

The filebuf_getc function retrieves a character from the current file position and returns it as an integer. the function then increments the current file position.

Use filebuf_getc to sequentially read characters from a buffered file.

Syntax

```
filebuf_getc(filebuf b);
```

Returns

An integer containing the character retrieved, or the constant IO_EOF or IO_ERROR upon an end of file or error.

Parameters

filebuf b is the name of the file buffer.

See Also

"filebuf_close" on page 88, "filebuf_buf2sd" on page 87, "filebuf_open" on page 89, "filter_create" on page 91

filebuf_open

The filebuf_open function opens a new file buffer for a previously opened file. The function returns a new buffer structure. Buffered files provide more efficient file access by guaranteeing the use of buffered file I/O in environments where it is not supported by the operating system.

Syntax

```
filebuf *filebuf_open(SYS_FILE fd, int sz);
```

Returns

A pointer to a new buffer structure to hold the data if successful, or NULL if no buffer could be opened.

Parameters

SYS_FILE fd is the platform-independent file descriptor of the file that has already been opened.

int sz is the size, in bytes, to be used for the buffer.

Example

```
filebuf *buf = filebuf_open(fd, FILE_BUFFERSIZE);if (!buf)
    { system_fclose(fd);}
```

See Also

"filebuf_getc" on page 89, "filebuf_buf2sd" on page 87, "filebuf_close" on page 88, "filebuf_open_nostat" on page 90

filebuf_open_nostat

The `filebuf_open_nostat` function opens a new file buffer for a previously opened file. The function returns a new buffer structure. Buffered files provide more efficient file access by guaranteeing the use of buffered file I/O in environments where it is not supported by the operating system.

This function is the same as `filebuf_open` but is more efficient because this function does not need to call the `request_stat_path` function. This function requires that the stat information be passed in.

Syntax

```
filebuf* filebuf_open_nostat(SYS_FILE fd, int sz,     struct stat *finfo);
```

Returns

A pointer to a new buffer structure to hold the data if successful, or NULL if no buffer could be opened.

Parameters

`SYS_FILE fd` is the platform-independent file descriptor of the file that has already been opened.

`int sz` is the size, in bytes, to be used for the buffer.

`struct stat *finfo` is the file information of the file. Before calling the `filebuf_open_nostat` function, you must call the `request_stat_path` function to retrieve the file information.

Example

```
filebuf *buf = filebuf_open_nostat(fd, FILE_BUFFERSIZE, &finfo);if (!buf)
    { system_fclose(fd);}
```

See Also

“filebuf_close” on page 88, “filebuf_open” on page 89, “filebuf_getc” on page 89, “filebuf_buf2sd” on page 87

filter_create

The `filter_create` function defines a new filter.

The `name` parameter specifies a unique name for the filter. If a filter with the specified name already exists, that file will be replaced.

Names beginning with `magnus-` or `server-` are reserved by the server.

The order parameter indicates the position of the filter in the filter stack by specifying that class of functionality that the filter implements.

The following table describes parameters that are allowed order constants and their associated meanings for the `filter_create` function. The left column lists the name of the constant, the middle column describes the functionality the filter implements, and the right column lists the position the filter occupies in the filter stack.

TABLE 4–1 `filter-create` Constants

Constant	Functionality Filter Implements	Position in Filter Stack
`FILTER_CONTENT_TRANSLATION`	Translates content from one form to another, for example, XSLT	Top
`FILTER_CONTENT_CODING`	Encodes content, for example, HTTP gzip compression	Middle
`FILTER_TRANSFER_CODING`	Encodes entity bodies for transmission, for example, HTTP chunking	Bottom

The `methods` parameter specifies a pointer to a `FilterMethods` structure. Before calling `filter_create`, you must first initialize the “FilterMethods” on page 199 structure using the `FILTER_METHODS_INITIALIZER` macro, and then assign function pointers to the individual `FilterMethods` members (for example, `insert`, `read`, `write`, and so on) that correspond to the filter methods the filter will support.

`filter_create` returns `const Filter *`, a pointer to an opaque representation of the filter. This value may be passed to `filter_insert` to insert the filter in a particular filter stack.

Syntax

```
const Filter *filter_create(const char *name, int order,
    const FilterMethods *methods);
```

Returns

The const Filter * that identifies the filter or NULL if an error occurred.

Parameters

const char *name is the name of the filter.

int order is one of the order constants above.

const FilterMethods *methods contains pointers to the filter methods that the filter supports.

Example

```
FilterMethods methods = FILTER_METHODS_INTIALIZER;
const Filter *filter;
/* This filter will only support the "read" filter method */
methods.read = my_input_filter_read;
/* Create the filter */
filter = filter_create("my-input-filter", FILTER_CONTENT_TRANSLATION,
&methods);
```

filter_find

The filter_find function finds the filter with the specified name.

Syntax

```
const Filter *filter_find(const char *name);
```

Returns

The const Filter * that identifies the filter, or NULL if the specified filter does not exist.

Parameters

const char *name is the name of the filter of interest.

filter_insert

The filter_insert function inserts a filter into a filter stack, creating a new filter layer and installing the filter at that layer. The filter layer's position in the stack is determined by the order value specified when "filter_create" on page 91 was called, and any explicit ordering configured by init-filter-order. If a filter layer with the same order value already exists in the stack, the new layer is inserted above that layer.

Parameters may be passed to the filter using the pb and data parameters. The semantics of the data parameter are defined by individual filters. However, all filters must be able to handle a data parameter of NULL.

When possible, plug-in developers should avoid calling filter_insert directly, and instead use the insert-filter SAF applicable in Input-class directives.

Syntax

```
int filter_insert(SYS_NETFD sd, pblock *pb, Session *sn, Request *rq,
    void *data, const Filter *filter);
```

Returns

Returns REQ_PROCEED if the specified filter was inserted successfully, or REQ_NOACTION if the specified filter was not inserted because it was not required. Any other return value indicates an error.

Parameters

SYS_NETFD sd is NULL (reserved for future use).

pblock *pb is a set of parameters to pass to the specified filter's init method.

Session *sn identifies the Session structure.

Request *rq identifies the Request structure.

void *data is filter-defined private data.

const Filter *filter is the filter to insert.

filter_layer

The filter_layer function returns the layer in a filter stack that corresponds to the specified filter.

Syntax

```
FilterLayer *filter_layer(SYS_NETFD sd, const Filter *filter);
```

Returns

The topmost FilterLayer * associated with the specified filter, or NULL if the specified filter is not part of the specified filter stack.

Parameters

`SYS_NETFD sd` is the filter stack to inspect.

`const Filter *filter` is the filter of interest.

filter_name

The `filter_name` function returns the name of the specified filter. The caller should not free the returned string.

Syntax

```
const char *filter_name(const Filter *filter);
```

Returns

The name of the specified filter, or NULL if an error occurred.

Parameters

`const Filter *filter` is the filter of interest.

filter_remove

The `filter_remove` function removes the specified filter from the specified filter stack, destroying a filter layer. If the specified filter was inserted into the filter stack multiple times, only that filter's topmost filter layer is destroyed.

When possible, plug-in developers should avoid calling `filter_remove` directly, and instead use the `remove-filter` SAF applicable in `Input`-, `Output`-, `Service`-, and `Error`-class directives.

Syntax

```
int filter_remove(SYS_NETFD sd, const Filter *filter);
```

Returns

Returns `REQ_PROCEED` if the specified filter was removed successfully or `REQ_NOACTION` if the specified filter was not part of the filter stack. Any other return value indicates an error.

Parameters

`SYS_NETFD sd` is the filter stack, `sn->csd.`

`const Filter *filter` is the filter to remove.

flush

The flush filter method is called when buffered data should be sent. Filters that buffer outgoing data should implement the flush filter method.

Upon receiving control, a flush implementation must write any buffered data to the filter layer immediately below it. Before returning success, a flush implementation must successfully call the “net_flush” on page 102 function:

```
net_flush(layer->lower).
```

Syntax

```
int flush(FilterLayer *layer);
```

Returns

0 on success or -1 if an error occurred.

Parameters

FilterLayer *layer is the filter layer the filter is installed in.

Example

```
int myfilter_flush(FilterLayer *layer)
{
    MyFilterContext context = (MyFilterContext *)layer->context->data;
    if (context->buf.count) {
       int rv;
       rv = net_write(layer->lower, context->buf.data, context->buf.count);
       if (rv != context->buf.count)
           return -1; /* failed to flush data */
       context->buf.count = 0;
    }
    return net_flush(layer->lower);
}
```

See Also

“net_flush” on page 102

FREE

The FREE macro is a platform-independent substitute for the C library routine free. This macro deallocates the space previously allocated by MALLOC, CALLOC, or STRDUP from the request’s memory pool.

Syntax

```
FREE(void *ptr);
```

Returns

```
void
```

Parameters

`void *ptr` is a `(void *)` pointer to a block of memory. If the pointer was not created by `MALLOC`, `CALLOC`, or `STRDUP`, the behavior is undefined.

Example

```
char *name;name = (char *) MALLOC(256);...FREE(name);
```

See Also

"CALLOC" on page 77, "REALLOC" on page 133, "STRDUP" on page 146, "PERM_MALLOC" on page 123, "PERM_FREE" on page 122, "PERM_REALLOC" on page 123, "PERM_STRDUP" on page 124

fs_blk_size

The `fs_blk_size` function returns the block size of the disk partition on which a specified directory resides.

Syntax

```
#include <libproxy/fs.h>
long fs_blk_size(char *root);
```

Returns

The block size, in bytes.

Parameters

`char` **root* is the name of the directory.

See Also

"fs_blks_avail" on page 97

fs_blks_avail

The fs_blks_avail function returns the number of disk blocks available on the disk partition on which a specified directory resides.

Syntax

```
#include <libproxy/fs.h>
long fs_blks_avail(char *root);
```

Returns

The number of available disk blocks.

Parameters

char **root* is the name of the directory.

See Also

“fs_blk_size” on page 96

func_exec

The func_exec function executes the function named by the fn entry in a specified pblock. If the function name is not found, the func_exec function logs the error and returns REQ_ABORTED.

You can use this function to execute a built-in Server Application Function (SAF) by identifying the SAF in the pblock.

Syntax

```
int func_exec(pblock *pb, Session *sn, Request *rq);
```

Returns

The value returned by the executed function, or the constant REQ_ABORTED if no function was executed.

Parameters

pblock pb is the pblock containing the function name (fn) and parameters.

Session *sn identifies the Session structure.

Request *rq identifies the Request structure.

The `Session` and `Request` parameters are the same as the parameters passed into your SAF.

See Also

"log_error" on page 100

func_find

The `func_find` function returns a pointer to the function specified by `name`. If that function does not exist, the `func_find` function returns NULL.

Syntax

```
FuncPtr func_find(char *name);
```

Returns

A pointer to the chosen function, suitable for dereferencing, or NULL if the function could not be found.

Parameters

`char *name` is the name of the function.

Example

```
/* this block of code does the same thing as func_exec */
    char *afunc = pblock_findval("afunction", pb);FuncPtr afnptr = func_find(afunc);if (afnptr)
    return (afnptr)(pb, sn, rq);
```

See Also

"func_exec" on page 97

func_insert

The `func_insert` function dynamically inserts a named function into the server's table of functions. This function should only be called during the `Init` stage.

Syntax

```
FuncStruct *func_insert(char *name, FuncPtr fn);
```

Returns

Returns the `FuncStruct` structure that identifies the newly inserted function. The caller should not modify the contents of the `FuncStruct` structure.

Parameters

char *name is the name of the function.

FuncPtr fn is the pointer to the function.

Example

```
func_insert("my-service-saf", &my_service_saf);
```

See Also

"func_exec" on page 97, "func_find" on page 98

I

insert

The insert filter method is called when a filter is inserted into a filter stack by the "filter_insert" on page 92 function or insert-filter SAF (applicable in Input-class directives).

Syntax

```
int insert(FilterLayer *layer, pblock *pb);
```

Returns

Returns REQ_PROCEED if the filter should be inserted into the filter stack, REQ_NOACTION if the filter should not be inserted because it is not required, or REQ_ABORTED if the filter should not be inserted because of an error.

Parameters

FilterLayer *layer is the filter layer at which the filter is being inserted.

pblock *pb is the set of parameters passed to filter_insert or specified by the fn="insert-filter" directive.

Example

```
FilterMethods myfilter_methods = FILTER_METHODS_INITIALIZER;
    const Filter *myfilter;int myfilter_insert(FilterLayer *layer, pblock *pb)
    {if (pblock_findval("dont-insert-filter", pb)) return REQ_NOACTION;
    return REQ_PROCEED;}...myfilter_methods.insert = &myfilter_insert;
    myfilter = filter_create("myfilter", &myfilter_methods);...
```

L

log_error

The `log_error` function creates an entry in an error log, recording the date and the severity of the error , and a specified message.

Syntax

```
int log_error(int degree, char *func, Session *sn, Request *rq,
    char *fmt, ...);
```

Returns

`0` if the log entry was created, or `-1` if the log entry was not created.

Parameters

`int degree` specifies the severity of the error. This value must be one of the following constants:

`LOG_WARN` — warning `LOG_MISCONFIG` — Asyntax error or permission violation `LOG_SECURITY` — An authentication failure or `403` error from a host`LOG_FAILURE` — An internal problem`LOG_CATASTROPHE` — A nonrecoverable server error`LOG_INFORM` — An informational message

`char *func` is the name of the function where the error has occurred.

`Session *sn` identifies the Session structure.

`Request *rq` identifies the Request structure.

The `Session` and `Request` parameters are the same as the ones passed into your SAF.

`char *fmt` specifies the format for the `printf` function that delivers the message.

`...` represents a sequence of parameters for the `printf` function.

Example

```
log_error(LOG_WARN, "send-file", sn, rq,
    "error opening buffer from %s (%s)"), path, system_errmsg(fd));
```

See Also

"func_exec" on page 97

M

magnus_atrestart

Note – Use the daemon-atrestart function in place of the obsolete magnus_atrestart function.

The magnus_atrestart function enables you to register a callback function named by *fn* to be used when the server receives a restart signal. Use this function when you need a callback function to deallocate resources allocated by an initialization function.

Syntax

```
#include <netsite.h>
void magnus_atrestart(void (*fn)(void *), void *data);
```

Returns

void

Parameters

void *(* fn) (void *)* is the callback function.

void **data* is the parameter passed to the callback function when the server is restarted.

Example

```
/* Close log file when server is restarted */
    magnus_atrestart(brief_terminate, NULL);return REQPROCEED;
```

MALLOC

The MALLOC macro is a platform-independent substitute for the C library routine malloc. This macro normally allocates from the request's memory pool. If pooled memory has been disabled in the configuration file with the pool-init built-in SAF, PERM_MALLOC and MALLOC both obtain their memory from the system heap.

Syntax

```
void *MALLOC(int size)
```

Returns

A void pointer to a block of memory.

Parameters

`int size` is the number of bytes to allocate.

Example

```
/* Allocate 256 bytes for a name */char *name;name = (char *) MALLOC(256);
```

See Also

"FREE" on page 95, "CALLOC" on page 77, "REALLOC" on page 133, "STRDUP" on page 146, "PERM_MALLOC" on page 123, "PERM_FREE" on page 122, "PERM_CALLOC" on page 121, "PERM_REALLOC" on page 123, "PERM_STRDUP" on page 124

N

net_flush

The `net_flush` function flushes any buffered data. If you require that data be sent immediately, call `net_flush` after calling network output functions such as `net_write` or `net_sendfile`.

Syntax

```
int net_flush(SYS_NETFD sd);
```

Returns

`0` on success, or a negative value if an error occurred.

Parameters

`SYS_NETFD sd` is the socket to flush.

Example

```
net_write(sn->csd, "Please wait... ", 15);
net_flush(sn->csd);
/* Perform some time-intensive operation */
...
net_write(sn->csd, "Thank you.\\n", 11);
```

See Also

"net_write" on page 105, "net_sendfile" on page 104

net_ip2host

The `net_ip2host` function transforms a textual IP address into a fully qualified domain name and returns that name.

Note – This function works only if the `DNS` directive is enabled in the `obj.conf` file. For more information, see *Oracle iPlanet Web Proxy Server 4.0.14 Configuration File Reference*.

Syntax

```
char *net_ip2host(char *ip, int verify);
```

Returns

A new string containing the fully qualified domain name if the transformation was accomplished, or NULL if the transformation was not accomplished.

Parameters

`char *ip` is the IP address as a character string in dotted-decimal notation: nnn.nnn.nnn.nnn

`int verify`, if nonzero, specifies that the function should verify the fully qualified domain name. This parameter requires an extra query but you should use it when checking access control.

net_read

The `net_read` function reads bytes from a specified socket into a specified buffer. The function waits to receive data from the socket until either at least one byte is available in the socket or the specified time has elapsed.

Syntax

```
int net_read (SYS_NETFD sd, char *buf, int sz, int timeout);
```

Returns

The number of bytes read, which will not exceed the maximum size, `sz`. A negative value is returned if an error has occurred, in which case `errno` is set to the constant `ETIMEDOUT` if the operation did not complete before `timeout` seconds elapsed.

Parameters

SYS_NETFD sd is the platform-independent socket descriptor.

char *buf is the buffer to receive the bytes.

int sz is the maximum number of bytes to read.

int timeout is the number of seconds to allow for the read operation before returning. Do not use timeout to return because not enough bytes were read in the given time instead, use this parameter to limit the amount of time devoted to waiting until some data arrives.

See Also

"net_write" on page 105

net_sendfile

The net_sendfile function sends the contents of a specified file to a specified socket. Either the whole file or a fraction of a file may be sent. The contents of the file may optionally be preceded or followed by caller-specified data.

Parameters are passed to net_sendfile in the sendfiledata structure. Before invoking net_sendfile, the caller must initialize every sendfiledata structure member.

Syntax

```
int net_sendfile(SYS_NETFD sd, const sendfiledata *sfd);
```

Returns

A positive number indicates the number of bytes successfully written, including the headers, file contents, and trailers. A negative value indicates an error.

Parameters

SYS_NETFD sd is the socket to write to.

const sendfiledata *sfd identifies the data to send.

Example

The following Service SAF sends a file bracketed by the strings "begin" and "end."

```
#include <string.h>
#include "nsapi.h"
```

```
NSAPI_PUBLIC int service_net_sendfile(pblock *pb, Session *sn, Request *rq)
{
    char *path;
    SYS_FILE fd;
    struct sendfiledata sfd;
    int rv;

    path = pblock_findval("path", rq->vars);
    fd = system_fopenRO(path);
    if (!fd) {
        log_error(LOG_MISCONFIG, "service-net-sendfile", sn, rq,
                  "Error opening %s (%s)", path, system_errmsg());
        return REQ_ABORTED;
    }

    sfd.fd = fd;                   /* file to send */
    sfd.offset = 0;                /* start sending from the beginning */
    sfd.len = 0;                   /* send the whole file */
    sfd.header = "begin";         /* header data to send before the file */
    sfd.hlen = strlen(sfd.header); /* length of header data */
    sfd.trailer = "end";          /* trailer data to send after the file */
    sfd.tlen = strlen(sfd.trailer);/* length of trailer data */

    /* send the headers, file, and trailers to the client */
    rv = net_sendfile(sn->csd, &sfd);

    system_fclose(fd);

    if (rv < 0) {
        log_error(LOG_INFORM, "service-net-sendfile", sn, rq,"Error sending
            %s (%s)", path, system_errmsg()); return REQ_ABORTED;
    }

    return REQ_PROCEED;
}
```

See Also

“net_flush” on page 102

net_write

The `net_write` function writes a specified number of bytes to a specified socket from a specified buffer.

Syntax

```
int net_write(SYS_NETFD sd, char *buf, int sz);
```

Returns

The number of bytes written, which may be less than the requested size if an error occurred.

Parameters

`SYS_NETFD sd` is the platform-independent socket descriptor.

`char *buf` is the buffer containing the bytes.

`int sz` is the number of bytes to write.

Example

```
if (net_write(sn->csd, FIRSTMSG, strlen(FIRSTMSG)) == IO_ERROR)
    return REQ_EXIT;
```

See Also

"net_read" on page 103

netbuf_buf2sd

The `netbuf_buf2sd` function sends a buffer to a socket. You can use this function to send data from IPC pipes to the client.

Syntax

```
int netbuf_buf2sd(netbuf *buf, SYS_NETFD sd, int len);
```

Returns

The number of bytes transferred to the socket, if successful, or the constant `IO_ERROR` if unsuccessful.

Parameters

`netbuf *buf` is the buffer to send.

`SYS_NETFD sd` is the platform-independent identifier of the socket.

`int len` is the length of the buffer.

See Also

"netbuf_close" on page 107, "netbuf_getc" on page 107, "netbuf_grab" on page 108, "netbuf_open" on page 108

netbuf_close

The `netbuf_close` function deallocates a network buffer and closes its associated files. Use this function when you need to deallocate the network buffer and close the socket.

You should never close the `netbuf` parameter in a `session` structure.

Syntax

```
void netbuf_close(netbuf *buf);
```

Returns

`void`

Parameters

`netbuf *buf` is the buffer to close.

See Also

"netbuf_buf2sd" on page 106, "netbuf_getc" on page 107, "netbuf_grab" on page 108, "netbuf_open" on page 108

netbuf_getc

The `netbuf_getc` function retrieves a character from the cursor position of the network buffer specified by `b`.

Syntax

```
netbuf_getc(netbuf b);
```

Returns

The integer representing the character if one was retrieved, or the constant `IO_EOF` or `IO_ERROR` for end of file or error.

Parameters

`netbuf b` is the buffer from which to retrieve one character.

See Also

"netbuf_buf2sd" on page 106, "netbuf_close" on page 107, "netbuf_grab" on page 108, "netbuf_open" on page 108

netbuf_grab

The `netbuf_grab` function reads `sz` number of bytes from the network buffer's (`buf`) socket into the network buffer. If the buffer is not large enough, it is resized. The data can be retrieved from `buf->inbuf` on success.

This function is used by the function `netbuf_buf2sd`.

Syntax

```
int netbuf_grab(netbuf *buf, int sz);
```

Returns

The number of bytes actually read (between `1` and `sz`) if the operation was successful, or the constant `IO_EOF` or `IO_ERROR` for end of file or error.

Parameters

`netbuf *buf` is the buffer to read into.

`int sz` is the number of bytes to read.

See Also

"netbuf_buf2sd" on page 106, "netbuf_close" on page 107, "netbuf_grab" on page 108, "netbuf_open" on page 108

netbuf_open

The `netbuf_open` function opens a new network buffer and returns it. You can use `netbuf_open` to create a `netbuf` structure and start using buffered I/O on a socket.

Syntax

```
netbuf* netbuf_open(SYS_NETFD sd, int sz);
```

Returns

A pointer to a new `netbuf` structure (network buffer).

Parameters

`SYS_NETFD sd` is the platform-independent identifier of the socket.

`int sz` is the number of characters to allocate for the network buffer.

See Also

“netbuf_buf2sd” on page 106, “netbuf_close” on page 107, “netbuf_getc” on page 107, “netbuf_grab” on page 108

nsapi_module_init

plug-in developers may define an nsapi_module_init function, which is a module initialization entry point that enables a plug-in to create filters when it is loaded. When an NSAPI module contains an nsapi_module_init function, the server will call that function immediately after loading the module. The nsapi_module_init presents the same interface as an Init SAF, and it must follow the same rules.

The nsapi_module_init function may be used to register SAFs with func_insert, and create filters with “filter_create” on page 91 .

Syntax

```
int nsapi_module_init(pblock *pb, Session *sn, Request *rq);
```

Returns

Returns REQ_PROCEED on success, or REQ_ABORTED on error.

Parameters

pblock *pb is a set of parameters specified by the fn="load-modules" directive.

Session *sn (the Session) is NULL.

Request *rq (the Request) is NULL.

NSAPI_RUNTIME_VERSION

The NSAPI_RUNTIME_VERSION macro defines the NSAPI version available at runtime. This value is the same as the highest NSAPI version supported by the server the plug-in is running in. The NSAPI version is encoded as in USE_NSAPI_VERSION.

The value returned by the NSAPI_RUNTIME_VERSION macro is valid only in Oracle iPlanet Web Server 7.0.9, Sun Java System Web Server 6.1, Sun iPlanet Web Server 6.0, Netscape Enterprise Server 6.0, and iPlanet Web Proxy Server 4.0 and higher. The server must support NSAPI 3.1 for this macro to return a valid value. Additionally, to use NSAPI_RUNTIME_VERSION, you must compile against an nsapi.h header file that supports NSAPI 3.2 or higher.

plug-in developers should not attempt to set the value of the NSAPI_RUNTIME_VERSION macro directly. Instead, see the USE_NSAPI_VERSION macro.

Syntax

```
int NSAPI_RUNTIME_VERSION
```

Example

```
NSAPI_PUBLIC int log_nsapi_runtime_version(pblock *pb, Session *sn,
    Request *rq) {log_error(LOG_INFORM, "log-nsapi-runtime-version", sn, rq,
                "Server supports NSAPI version %d.%d\\n",
                NSAPI_RUNTIME_VERSION / 100,
                NSAPI_RUNTIME_VERSION % 100);
return REQ_PROCEED;
}
```

See Also

"NSAPI_VERSION" on page 110

"USE_NSAPI_VERSION" on page 161

NSAPI_VERSION

The NSAPI_VERSION macro defines the NSAPI version used at compile time. This value is determined by the value of the USE_NSAPI_VERSION macro, or, if the plug-in developer did not define USE_NSAPI_VERSION, by the highest NSAPI version supported by the nsapi.h header the plug-in was compiled against. The NSAPI version is encoded as in USE_NSAPI_VERSION.

plug-in developers should not attempt to set the value of the NSAPI_VERSION macro directly. Instead, see the USE_NSAPI_VERSION macro..

Syntax

```
int NSAPI_VERSION
```

Example

```
NSAPI_PUBLIC int log_nsapi_compile_time_version(pblock *pb, Session *sn,
    Request *rq) {log_error(LOG_INFORM, "log-nsapi-compile-time-version", sn, rq,
            "Plugin compiled against NSAPI version %d.%d\\n",
            NSAPI_VERSION / 100,
            NSAPI_VERSION % 100);
return REQ_PROCEED;
}
```

See Also

"NSAPI_RUNTIME_VERSION" on page 109

"USE_NSAPI_VERSION" on page 161

P

param_create

The param_create function creates a pb_param structure containing a specified name and value. The name and value are copied. Use this function to prepare a pb_param structure to be used in calls to pblock routines such as pblock_pinsert.

Syntax

```
pb_param *param_create(char *name, char *value);
```

Returns

A pointer to a new pb_param structure.

Parameters

char *name is the string containing the name.

char *value is the string containing the value.

Example

```
pb_param *newpp = param_create("content-type","text/plain");
    pblock_pinsert(newpp, rq->srvhdrs);
```

See Also

“param_free” on page 111, “pblock_pinsert” on page 119, “pblock_remove” on page 119

param_free

The param_free function frees the pb_param structure specified by pp and its associated structures. Use the param_free function to dispose a pb_param after removing it from a pblock with pblock_remove.

Syntax

```
int param_free(pb_param *pp);
```

Returns

1 if the parameter was freed or 0 if the parameter was NULL.

Parameters

pb_param *pp is the name-value pair stored in a pblock.

Example

```
if (param_free(pblock_remove("content-type", rq-srvhdrs))) return;
    /* we removed it */
```

See Also

"param_create" on page 111, "pblock_pinsert" on page 119, "pblock_remove" on page 119

pblock_copy

The pblock_copy function copies the entries of the source pblock and adds them into the destination pblock. Any previous entries in the destination pblock are left intact.

Syntax

```
void pblock_copy(pblock *src, pblock *dst);
```

Returns

void

Parameters

pblock *src is the source pblock.

pblock *dst is the destination pblock.

Names and values are newly allocated so that the original pblock may be freed, or the new pblock changed without affecting the original pblock.

See Also

"pblock_create" on page 112, "pblock_dup" on page 113, "pblock_free" on page 115, "pblock_find" on page 113, "pblock_findval" on page 115, "pblock_remove" on page 119, "pblock_nvinsert" on page 117

pblock_create

The pblock_create function creates a new pblock. The pblock maintains an internal hash table for fast name-value pair lookups.

Syntax

```
pblock *pblock_create(int n);
```

Returns

A pointer to a newly allocated `pblock`.

Parameters

`int n` is the size of the hash table (number of name-value pairs) for the pblock.

See Also

"pblock_copy" on page 112, "pblock_dup" on page 113, "pblock_find" on page 113, "pblock_findval" on page 115, "pblock_free" on page 115, "pblock_nvinsert" on page 117, "pblock_remove" on page 119

pblock_dup

The `pblock_dup` function duplicates a pblock. This function is equivalent to a sequence of `pblock_create` and `pblock_copy`.

Syntax

```
pblock *pblock_dup(pblock *src);
```

Returns

A pointer to a newly allocated `pblock`.

Parameters

`pblock *src` is the source pblock.

See Also

"pblock_create" on page 112, "pblock_find" on page 113, "pblock_findval" on page 115, "pblock_free" on page 115, "pblock_nvinsert" on page 117, "pblock_remove" on page 119

pblock_find

The `pblock_find` function finds a specified name-value pair entry in a pblock, and returns the pb_param structure. If you only want the value associated with the name, use the `pblock_findval` function.

This function is implemented as a macro.

Syntax

```
pb_param *pblock_find(char *name, pblock *pb);
```

Returns

A pointer to the `pb_param` structure if one was found, or NULL if name was not found.

Parameters

`char *name` is the name of a name-value pair.

`pblock *pb` is the `pblock` to be searched.

See Also

"pblock_copy" on page 112, "pblock_dup" on page 113, "pblock_findval" on page 115, "pblock_free" on page 115, "pblock_nvinsert" on page 117, "pblock_remove" on page 119

pblock_findlong

The `pblock_findlong` function finds a specified name-value pair entry in a parameter block, and retrieves the name and structure of the parameter block. Use `pblock_findlong` if you want to retrieve the name, structure, and value of the parameter block. However, if you want only the name and structure of the parameter block, use the `pblock_find` function. Do not use these two functions in conjunction.

Syntax

```
#include <libproxy/util.h>
long pblock_findlong(char *name, pblock *pb);
```

Returns

- A `long` containing the value associated with the name
- -1 if no match was found

Parameters

`char` **name* is the name of a name-value pair.

`pblock` **pb* is the parameter block to be searched.

See Also

pblock_nlinsert

pblock_findval

The pblock_findval function finds the value of a specified name in a pblock. If you just want the pb_param structure of the pblock, use the pblock_find function.

The pointer returned is a pointer into the pblock. Do not FREE it. If you want to modify it, do a STRDUP and modify the copy.

Syntax

```
char *pblock_findval(char *name, pblock *pb);
```

Returns

A string containing the value associated with the name or NULL if no match was found.

Parameters

char *name is the name of a name-value pair.

pblock *pb is the pblock to be searched.

Example

see "pblock_nvinsert" on page 117.

See Also

"pblock_create" on page 112, "pblock_copy" on page 112, "pblock_find" on page 113, "pblock_free" on page 115, "pblock_nvinsert" on page 117, "pblock_remove" on page 119, "request_header" on page 135

pblock_free

The pblock_free function frees a specified pblock and any entries inside it. If you want to save a variable in the pblock, remove the variable using the function pblock_remove and save the resulting pointer.

Syntax

```
void pblock_free(pblock *pb);
```

Returns

void

Parameters

pblock *pb is the pblock to be freed.

See Also

“pblock_copy” on page 112, “pblock_create” on page 112, “pblock_dup” on page 113, “pblock_find” on page 113, “pblock_findval” on page 115, “pblock_nvinsert” on page 117, “pblock_remove” on page 119

pblock_nlinsert

The pblock_nlinsert function creates a new parameter structure with a given name and long numeric value and inserts the structure into a specified parameter block. The name and value parameters are also newly allocated.

Syntax

```
#include <libproxy/util.h>
pb_param *pblock_nlinsert(char *name, long value, pblock *pb);
```

Returns

The newly allocated parameter block structure

Parameters

char **name* is the name by which the name-value pair is stored.

long *value* is the long or integer value being inserted into the parameter block.

pblock **pb* is the parameter block into which the insertion occurs.

See Also

“pblock_findlong” on page 114

pblock_nninsert

The pblock_nninsert function creates a new entry with a given name and a numeric value in the specified pblock. The numeric value is first converted into a string. The name and value parameters are copied.

Syntax

```
pb_param *pblock_nninsert(char *name, int value, pblock *pb);
```

Returns

A pointer to the new pb_param structure.

Parameters

char *name is the name of the new entry.

int value is the numeric value being inserted into the pblock. This parameter must be an integer. If the value you assign is not a number, then instead use the function pblock_nvinsert to create the parameter.

pblock *pb is the pblock into which the insertion occurs.

See Also

"pblock_copy" on page 112, "pblock_create" on page 112, "pblock_find" on page 113, "pblock_free" on page 115, "pblock_nvinsert" on page 117, "pblock_remove" on page 119, "pblock_str2pblock" on page 121

pblock_nvinsert

The pblock_nvinsert function creates a new entry with a given name and character value in the specified pblock. The name and value parameters are copied.

Syntax

```
pb_param *pblock_nvinsert(char *name, char *value, pblock *pb);
```

Returns

A pointer to the newly allocated pb_param structure.

Parameters

char *name is the name of the new entry.

char *value is the string value of the new entry.

pblock *pb is the pblock into which the insertion occurs.

Example

```
pblock_nvinsert("content-type", "text/html", rq->srvhdrs);
```

See Also

"pblock_copy" on page 112, "pblock_create" on page 112, "pblock_find" on page 113, "pblock_free" on page 115, "pblock_nninsert" on page 116, "pblock_remove" on page 119, "pblock_str2pblock" on page 121

pblock_pb2env

The pblock_pb2env function copies a specified pblock into a specified environment. The function creates one new environment entry for each name-value pair in the pblock. Use this function to send pblock entries to a program that you are going to execute.

Syntax

```
char **pblock_pb2env(pblock *pb, char **env);
```

Returns

A pointer to the environment.

Parameters

pblock *pb is the pblock to be copied.

char **env is the environment into which the pblock is to be copied.

See Also

"pblock_copy" on page 112, "pblock_create" on page 112, "pblock_find" on page 113, "pblock_free" on page 115, "pblock_nvinsert" on page 117, "pblock_remove" on page 119, "pblock_str2pblock" on page 121

pblock_pblock2str

The pblock_pblock2str function copies all parameters of a specified pblock into a specified string. The function allocates additional nonheap space for the string if needed.

Use this function to stream the pblock for archival and other purposes.

Syntax

```
char *pblock_pblock2str(pblock *pb, char *str);
```

Returns

The new version of the str parameter. If str is NULL, the string is a new string otherwise, the string is a reallocated string. In either case, it is allocated from the request's memory pool.

Parameters

pblock *pb is the pblock to be copied.

char *str is the string into which the pblock is to be copied. The string must have been allocated by MALLOC or REALLOC, not by PERM_MALLOC or PERM_REALLOC, which allocate from the system heap.

Each name-value pair in the string is separated from its neighbor pair by a space, and is in the format *name*="*value*."

See Also

"pblock_copy" on page 112, "pblock_create" on page 112, "pblock_find" on page 113, "pblock_free" on page 115, "pblock_nvinsert" on page 117, "pblock_remove" on page 119, "pblock_str2pblock" on page 121

pblock_pinsert

The function pblock_pinsert inserts a pb_param structure into a pblock.

Syntax

```
void pblock_pinsert(pb_param *pp, pblock *pb);
```

Returns

void

Parameters

pb_param *pp is the pb_param structure to insert.

pblock *pb is the pblock.

See Also

"pblock_copy" on page 112, "pblock_create" on page 112, "pblock_find" on page 113, "pblock_free" on page 115, "pblock_nvinsert" on page 117, "pblock_remove" on page 119, "pblock_str2pblock" on page 121

pblock_remove

The pblock_remove function removes a specified name-value entry from a specified pblock. If you use this function, you should eventually call param_free to deallocate the memory used by the pb_param structure.

Syntax

```
pb_param *pblock_remove(char *name, pblock *pb);
```

Returns

A pointer to the named `pb_param` structure if it was found, or NULL if the named `pb_param` was not found.

Parameters

`char *name` is the name of the `pb_param` to be removed.

`pblock *pb` is the `pblock` from which the name-value entry is to be removed.

See Also

"pblock_copy" on page 112, "pblock_create" on page 112, "pblock_find" on page 113, "pblock_free" on page 115, "pblock_nvinsert" on page 117, "param_create" on page 111, "param_free" on page 111

pblock_replace_name

The `pblock_replace_name` function replaces the name of a name-value pair, retaining the value.

Syntax

```
#include <libproxy/util.h>
void pblock_replace_name(char *oname,char *nname, pblock *pb);
```

Returns

`void`

Parameters

`char *`*oname* is the old name of a name-value pair.

`char *`*nname* is the new name for the name-value pair.

`pblock *`*pb* is the parameter block to be searched.

See Also

"pblock_remove" on page 119

pblock_str2pblock

The pblock_str2pblock function scans a string for parameter pairs, adds them to a pblock, and returns the number of parameters added.

Syntax

```
int pblock_str2pblock(char *str, pblock *pb);
```

Returns

The number of parameter pairs added to the pblock, if any, or -1 if an error occurred.

Parameters

char *str is the string to be scanned.

The name-value pairs in the string can have the format *name*=*value* or *name*="*value*."

All backslashes (\\) must be followed by a literal character. If string values are found with no unescaped = signs (no name=), it assumes the names 1, 2, 3, and so on, depending on the string position. For example, if pblock_str2pblock finds "some strings together," the function treats the strings as if they appeared in name-value pairs as 1="some" 2="strings" 3="together."

pblock *pb is the pblock into which the name-value pairs are stored.

See Also

"pblock_copy" on page 112, "pblock_create" on page 112, "pblock_find" on page 113, "pblock_free" on page 115, "pblock_nvinsert" on page 117, "pblock_remove" on page 119, "pblock_pblock2str" on page 118

PERM_CALLOC

The PERM_CALLOC macro is a platform-independent substitute for the C library routine calloc. This macro allocates int size bytes of memory that persist after the request that is being processed has been completed. If pooled memory has been disabled in the configuration file with the pool-init built-in SAF, PERM_CALLOC and CALLOC both obtain their memory from the system heap.

Syntax

```
void *PERM_CALLOC(int size)
```

Returns

A void pointer to a block of memory.

Parameters

`int size` is the size in bytes of each element.

Example

```
char **name;name = (char **) PERM_CALLOC(100);
```

See Also

"PERM_FREE" on page 122, "PERM_STRDUP" on page 124, "PERM_MALLOC" on page 123, "PERM_REALLOC" on page 123, "MALLOC" on page 101, "FREE" on page 95, "CALLOC" on page 77, "STRDUP" on page 146, "REALLOC" on page 133

PERM_FREE

The `PERM_FREE` macro is a platform-independent substitute for the C library routine `free`. It deallocates the persistent space previously allocated by `PERM_MALLOC`, `PERM_CALLOC`, or `PERM_STRDUP`. If pooled memory has been disabled in the configuration file with the `pool-init` built-in SAF, `PERM_FREE` and `FREE` both deallocate memory in the system heap.

Syntax

```
PERM_FREE(void *ptr);
```

Returns

`void`

Parameters

`void *ptr` is a `(void *)` pointer to block of memory. If the pointer is not one created by `PERM_MALLOC`, `PERM_CALLOC`, or `PERM_STRDUP`, the behavior is undefined.

Example

```
char *name;name = (char *) PERM_MALLOC(256);...PERM_FREE(name);
```

See Also

"FREE" on page 95, "MALLOC" on page 101, "CALLOC" on page 77, "REALLOC" on page 133, "STRDUP" on page 146, "PERM_MALLOC" on page 123, "PERM_CALLOC" on page 121, "PERM_REALLOC" on page 123, "PERM_STRDUP" on page 124

PERM_MALLOC

The PERM_MALLOC macro is a platform-independent substitute for the C library routine malloc. This macro provides allocation of memory that persists after the request that is being processed has been completed. If pooled memory has been disabled in the configuration file with the pool-init built-in SAF, PERM_MALLOC and MALLOC both obtain their memory from the system heap.

Syntax

```
void *PERM_MALLOC(int size)
```

Returns

A void pointer to a block of memory.

Parameters

int size is the number of bytes to allocate.

Example

```
/* Allocate 256 bytes for a name */char *name;name = (char *)
    PERM_MALLOC(256);
```

See Also

"PERM_FREE" on page 122, "PERM_STRDUP" on page 124, "PERM_CALLOC" on page 121, "PERM_REALLOC" on page 123, "MALLOC" on page 101, "FREE" on page 95, "CALLOC" on page 77, "STRDUP" on page 146, "REALLOC" on page 133

PERM_REALLOC

The PERM_REALLOC macro is a platform-independent substitute for the C library routine realloc. This macro changes the size of a specified memory block that was originally created by MALLOC, CALLOC, or STRDUP. The contents of the object remains unchanged up to the lesser of the old and new sizes. If the new size is larger, the new space is uninitialized.

Warning

Calling PERM_REALLOC for a block that was allocated with MALLOC, CALLOC, or STRDUP will not work.

Syntax

```
void *PERM_REALLOC(vod *ptr, int size)
```

Returns

A void pointer to a block of memory.

Parameters

void *ptr is a void pointer to a block of memory created by PERM_MALLOC, PERM_CALLOC, or PERM_STRDUP.

int size is the number of bytes to which the memory block should be resized.

Example

```
char *name;name = (char *) PERM_MALLOC(256);if (NotBigEnough())
    name = (char *) PERM_REALLOC(512);
```

See Also

"PERM_MALLOC" on page 123,"PERM_FREE" on page 122, "PERM_CALLOC" on page 121, "PERM_STRDUP" on page 124, "MALLOC" on page 101, "FREE" on page 95, "STRDUP" on page 146, "CALLOC" on page 77, "REALLOC" on page 133

PERM_STRDUP

The PERM_STRDUP macro is a platform-independent substitute for the C library routine strdup. This macro creates a new copy of a string in memory that persists after the request that is being processed has been completed. If pooled memory has been disabled in the configuration file with the pool-init built-in SAF, PERM_STRDUP and STRDUP both obtain their memory from the system heap.

The PERM_STRDUP routine is functionally equivalent to:

```
newstr = (char *) PERM_MALLOC(strlen(str) + 1);strcpy(newstr, str);
```

A string created with PERM_STRDUP should be disposed with PERM_FREE.

Syntax

```
char *PERM_STRDUP(char *ptr);
```

Returns

A pointer to the new string.

Parameters

`char *ptr` is a pointer to a string.

See Also

"PERM_MALLOC" on page 123,"PERM_FREE" on page 122, "PERM_CALLOC" on page 121, "PERM_REALLOC" on page 123, "MALLOC" on page 101, "FREE" on page 95, "STRDUP" on page 146, "CALLOC" on page 77, "REALLOC" on page 133

prepare_nsapi_thread

The `prepare_nsapi_thread` function enables threads that are not created by the server to act like server-created threads. This function must be called before any NSAPI functions are called from a thread that is not server-created.

Syntax

```
void prepare_nsapi_thread(Request *rq, Session *sn);
```

Returns

`void`

Parameters

`Request *rq` identifies the Request structure.

`Session *sn` identifies the Session structure.

The Request and Session parameters are the same as the ones passed into your SAF.

See Also

"protocol_start_response" on page 129

protocol_dump822

The `protocol_dump822` function prints headers from a specified `pblock` into a specific buffer, with a specified size and position. Use this function to serialize the headers so that they can be sent, for example, in a mail message.

Syntax

```
char *protocol_dump822(pblock *pb, char *t, int *pos, int tsz);
```

Returns

A pointer to the buffer, which will be reallocated if necessary.

The function also adds `*pos` to the end of the headers in the buffer.

Parameters

`pblock *pb` is the `pblock` structure.

`char *t` is the buffer, allocated with `MALLOC`, `CALLOC`, or `STRDUP`.

`int *pos` is the position within the buffer at which the headers are to be dumped.

`int tsz` is the size of the buffer.

See Also

"protocol_start_response" on page 129, "protocol_status" on page 130

protocol_finish_request

The `protocol_finish_request` function finishes a specified request. For HTTP, the function closes the socket.

Syntax

```
#include <frame/protocol.h>
void protocol_finish_request(Session *sn, Request *rq);
```

Returns

`void`

Parameters

`Session` **sn* is the Session that generated the request.

`Request` **rq* is the Request to be finished.

See Also

protocol_handle_session, *protocol_scan_headers*, *protocol_start_response*, *protocol_status*

protocol_handle_session

The `protocol_handle_session` function processes each request generated by a specified session.

Syntax

```
#include <frame/protocol.h>
void protocol_handle_session(Session *sn);
```

Parameters

`Session` **sn* is the that generated the requests.

See Also

protocol_scan_headers, *protocol_start_response*, *protocol_status*

protocol_parse_request

Parses the first line of an HTTP request.

Syntax

```
#include <frame/protocol.h>
int protocol_parse_request(char *t, Request *rq, Session *sn);
```

Returns

Returns `REQ_PROCEED` if the operation succeeded, or `REQ_ABORTED` if the operation did not succeed.

Parameters

`char` **t* defines a string of length `REQ_MAX_LINE`. This is an optimization for the internal code to reduce usage of runtime stack.

`Request` **rq* is the request to be parsed.

`Session` **sn* is the session that generated the request.

See Also

"protocol_scan_headers" on page 127, "protocol_start_response" on page 129, "protocol_status" on page 130

protocol_scan_headers

Scans HTTP headers from a specified network buffer, and places them in a specified parameter block.

Folded lines are joined and the linefeeds are removed but not the whitespace. Any repeat headers are joined and the two field bodies are separated by a comma and space. For example, multiple mail headers are combined into one header and a comma is used to separate the field bodies.

Syntax

```
#include <frame/protocol.h>
int protocol_scan_headers(Session *sn, netbuf *buf, char *t, pblock *headers);
```

Returns

Returns `REQ_PROCEED` if the operation succeeded, or `REQ_ABORTED` if the operation did not succeed.

Parameters

`Session` **sn* is the session that generated the request. The structure named by *sn* contains a pointer to a netbuf called *inbuf*. If the parameter *buf* is NULL, the function automatically uses *inbuf*.

Note that *sn* is an optional parameter that is used for error logs. Use NULL if you wish.

`netbuf` **buf* is the network buffer to be scanned for HTTP headers.

`char` **t* defines a string of length `REQ_MAX_LINE`. This is an optimization for the internal code to reduce usage of runtime stack.

`pblock` **headers* is the parameter block to receive the headers.

See Also

"protocol_handle_session" on page 126, "protocol_start_response" on page 129, "protocol_status" on page 130

protocol_set_finfo

The `protocol_set_finfo` function retrieves the `content-length` and `last-modified` date from a specified `stat` structure and adds them to the response headers (`rq->srvhdrs`). Call `protocol_set_finfo` before calling `protocol_start_response`.

Syntax

```
int protocol_set_finfo(Session *sn, Request *rq, struct stat *finfo);
```

Returns

Returns REQ_PROCEED if the request can proceed normally, or REQ_ABORTED if the function should treat the request normally but not send any output to the client.

Parameters

Session *sn identifies the Session structure.

Request *rq identifies the Request structure.

The Session and Request parameters are the same as the parameter passed into your SAF.

stat *finfo is the stat structure for the file.

The stat structure contains the information about the file from the file system. You can get the stat structure info using request_stat_path.

See Also

"protocol_start_response" on page 129, "protocol_status" on page 130

protocol_start_response

The protocol_start_response function initiates the HTTP response for a specified session and request. If the protocol version is HTTP/0.9, the function does nothing, because that version has no concept of status. If the protocol version is HTTP/1.0, the function sends a status line followed by the response headers. Use this function to set up HTTP and prepare the client and server to receive the body (or data) of the response.

Syntax

```
int protocol_start_response(Session *sn, Request *rq);
```

Returns

The constant REQ_PROCEED if the operation succeeded, in which case you should send the data you were preparing to send.

The constant REQ_NOACTION if the operation succeeded but the request method was HEAD, in which case no data should be sent to the client.

The constant REQ_ABORTED if the operation did not succeed.

Parameters

Session *sn is the Session.

Request *rq is the Request.

The Session and Request parameters are the same as the ones passed into your SAF.

Example

```
/* A noaction response from this function means the request was HEAD */
    if (protocol_start_response(sn, rq) == REQ_NOACTION) { filebuf_close(groupbuf);
    /* close our file*/    return REQ_PROCEED;}
```

See Also

"protocol_status" on page 130

protocol_status

The protocol_status function sets the session status to indicate whether an error condition occurred. If the reason string is NULL, the server attempts to find a reason string for the given status code. If the server finds no string, it returns "Unknown reason." The reason string is sent to the client in the HTTP response line. Use this function to set the status of the response before calling the function protocol_start_response.

For the complete list of valid status code constants, refer to the nsapi.h file in the server distribution.

Syntax

```
void protocol_status(Session *sn, Request *rq, int n, char *r);
```

Returns

void, but the function sets values in the Session/Request designated by sn/rq for the status code and the reason string.

Parameters

Session *sn identifies the Session structure.

Request *rq identifies the Request structure.

The Session and Request parameters are the same as the parameters passed into your SAF.

int n is one of the status code constants above.

char *r is the reason string.

Example

```
/* if we find extra path-info, the URL was bad so tell the */
/* browser it was not found */if (t = pblock_findval("path-info", rq->vars))
{  protocol_status(sn, rq, PROTOCOL_NOT_FOUND, NULL);
    log_error(LOG_WARN, "function-name", sn, rq, "%s not found", path);
    return REQ_ABORTED;}
```

See Also

"protocol_start_response" on page 129

protocol_uri2url

The protocol_uri2url function takes strings containing the given URI prefix and URI suffix, and creates a newly allocated, fully qualified URL in the form http:// server : port prefix suffix. See protocol_uri2url_dynamic.

If you want to omit either the URI prefix or suffix, use "" instead of NULL as the value for either parameter.

Syntax

```
char *protocol_uri2url(char *prefix, char *suffix);
```

Returns

A new string containing the URL.

Parameters

char *prefix is the prefix.

char *suffix is the suffix.

See Also

"protocol_start_response" on page 129, "protocol_status" on page 130, "pblock_nvinsert" on page 117, "protocol_uri2url_dynamic" on page 131

protocol_uri2url_dynamic

The protocol_uri2url function takes strings containing the given URI prefix and URI suffix, and creates a newly allocated, fully qualified URL in the form http://server:port prefix suffix.

If you want to omit either the URI prefix or suffix, use "" instead of NULL as the value for either parameter.

The `protocol_uri2url_dynamic` function is similar to the `protocol_uri2url` function, but should be used whenever the `session` and `request` structures are available. This function ensures that the URL that the function constructs refers to the host that the client specified.

Syntax

```
char *protocol_uri2url(char *prefix, char *suffix, Session *sn, Request *rq);
```

Returns

A new string containing the URL.

Parameters

`char *prefix` is the prefix.

`char *suffix` is the suffix.

`Session *sn` identifies the Session structure.

`Request *rq` identifies the Request structure.

The `Session` and `Request` parameters are the same as the parameters passed into your SAF.

See Also

"protocol_start_response" on page 129, "protocol_status" on page 130, "protocol_uri2url_dynamic" on page 131

R

read

The `read` filter method is called when input data is required. Filters that modify or consume incoming data should implement the `read` filter method.

Upon receiving control, a read implementation should fill `buf` with up to `amount` bytes of input data. This data may be obtained by calling the "net_read" on page 103 function, as shown in the example below.

Syntax

```
int read(FilterLayer *layer, void *buf, int amount, int timeout);
```

Returns

The number of bytes placed in buf on success, 0 if no data is available, or a negative value if an error occurred.

Parameters

FilterLayer *layer is the filter layer in which the filter is installed.

void *buf is the buffer in which data should be placed.

int amount is the maximum number of bytes that should be placed in the buffer.

int timeout is the number of seconds to allow for the read operation before returning. Do not use timeoutto return because not enough bytes were read in the given time. Instead, use this function to limit the amount of time devoted to waiting until some data arrives.

Example

```
int myfilter_read(FilterLayer *layer, void *buf, int amount, int timeout)
    { return net_read(layer->lower, buf, amount, timeout);}
```

See Also

"net_read" on page 103

REALLOC

The REALLOC macro is a platform-independent substitute for the C library routine realloc. It changes the size of a specified memory block that was originally created by MALLOC, CALLOC, or STRDUP. The contents of the object remains unchanged up to the lesser of the old and new sizes. If the new size is larger, the new space is uninitialized.

Warning

Calling REALLOC for a block that was allocated with PERM_MALLOC, PERM_CALLOC, or PERM_STRDUP will not work.

Syntax

```
void *REALLOC(void *ptr, int size);
```

Returns

A pointer to the new space if the request could be satisfied.

Parameters

void *ptr is a (void *) pointer to a block of memory. If the pointer was not created by MALLOC, CALLOC, or STRDUP, the behavior is undefined.

int size is the number of bytes to allocate.

Example

```
char *name;name = (char *) MALLOC(256);if (NotBigEnough())
    name = (char *) REALLOC(512);
```

See Also

"MALLOC" on page 101, "FREE" on page 95, "STRDUP" on page 146, "CALLOC" on page 77, "PERM_MALLOC" on page 123, "PERM_FREE" on page 122, "PERM_REALLOC" on page 123, "PERM_CALLOC" on page 121, "PERM_STRDUP" on page 124

remove

The remove filter method is called when the filter stack is destroyed, or when a filter is removed from a filter stack by the "filter_remove" on page 94 function or remove-filter SAF (applicable in Input-, Output-, Service-, and Error-class directives).

Waiting to flush buffered data when the remove method is invoked might be too late. For this reason, filters that buffer outgoing data should implement the flush filter method.

Syntax

```
void remove(FilterLayer *layer);
```

Returns

void

Parameters

FilterLayer *layer is the filter layer the filter is installed in.

See Also

"flush" on page 95

request_create

The request_create function is a utility function that creates a new request structure.

Syntax

```
#include <frame/req.h>
Request *request_create(void);
```

Returns

A Request structure

Parameters

No parameter is required.

See Also

"request_free" on page 135, "request_header" on page 135

request_free

The request_free function frees a specified request structure.

Syntax

```
#include <frame/req.h>
void request_free(Request *req);
```

Returns

void

Parameters

Request **rq* is the Request structure to be freed.

See Also

"request_header" on page 135

request_header

The request_header function finds an entry in the pblock containing the client's HTTP request headers (rq->headers). You must use this function rather than pblock_findval when accessing the client headers because the server might begin processing the request before the headers have been completely read.

Syntax

```
int request_header(char *name, char **value, Session *sn, Request *rq);
```

Returns

Returns REQ_PROCEED if the header was found, REQ_ABORTED if the header was not found, or REQ_EXIT if an error occurred reading from the client.

Parameters

char *name is the name of the header.

char **value is the address where the function will place the value of the specified header. If none is found, the function stores NULL.

Session *sn identifies the Session structure.

Request *rq identifies the Request structure.

The Session and Request parameters are the same as the parameters passed into your SAF.

See Also

"request_create" on page 134, "request_free" on page 135

S

sem_grab

The sem_grab function requests exclusive access to a specified semaphore. If exclusive access is unavailable, the caller blocks execution until exclusive access becomes available. Use this function to ensure that only one server processor thread performs an action at a time.

Syntax

```
#include <base/sem.h>
int sem_grab(SEMAPHORE id);
```

Returns

- -1 if an error occurred
- 0 to signal success

Parameters

SEMAPHORE *id* is the unique identification number of the requested semaphore.

See Also

"sem_init" on page 137, "sem_release" on page 137, "sem_terminate" on page 138, "sem_tgrab" on page 138

sem_init

The `sem_init` function creates a semaphore with a specified name and unique identification number. Use this function to allocate a new semaphore that will be used with the functions `sem_grab` and `sem_release`. Call `sem_init` from an `init` class function to initialize a static or global variable that the other classes will later use.

Syntax

```
#include <base/sem.h>
SEMAPHORE sem_init(char *name, int number);
```

Returns

The constant SEM_ERROR if an error occurred.

Parameters

`SEMAPHORE` **name* is the name for the requested semaphore. The file name of the semaphore should be a file accessible to the process.

`int` *number* is the unique identification number for the requested semaphore.

See Also

"sem_grab" on page 136, "sem_release" on page 137, "sem_terminate" on page 138

sem_release

The `sem_release` function releases the process's exclusive control over a specified semaphore. Use this function to release exclusive control over a semaphore created with the function `sem_grab`.

Syntax

```
#include <base/sem.h>
int sem_release(SEMAPHORE id);
```

Returns

- `-1` if an error occurred
- `0` if no error occurred

Parameters

SEMAPHORE *id* is the unique identification number of the semaphore.

See Also

"sem_grab" on page 136, "sem_init" on page 137, "sem_terminate" on page 138

sem_terminate

The sem_terminate function deallocates the semaphore specified by *id*. You can use this function to deallocate a semaphore that was previously allocated with the function sem_init.

Syntax

```
#include <base/sem.h>
void sem_terminate(SEMAPHORE id);
```

Returns

void

Parameters

SEMAPHORE *id* is the unique identification number of the semaphore.

See Also

"sem_grab" on page 136, "sem_init" on page 137, "sem_release" on page 137

sem_tgrab

The sem_tgrab function tests and requests exclusive use of a semaphore. Unlike the similar sem_grab function, if exclusive access is unavailable the caller is not blocked but receives a return value of -1. Use this function to ensure that only one server processor thread performs an action at a time.

Syntax

```
#include <base/sem.h>
int sem_grab(SEMAPHORE id);
```

Returns

- -1 if an error occurred or if exclusive access was not available
- 0 exclusive access was granted

Parameters

SEMAPHORE *id* is the unique identification number of the semaphore.

See Also

"sem_grab" on page 136, "sem_init" on page 137, "sem_release" on page 137, "sem_terminate" on page 138

sendfile

The sendfile filter method is called when the contents of a file are to be sent. Filters that modify or consume outgoing data might implement the sendfile filter method.

If a filter implements the write filter method but not the sendfile filter method, the server will automatically translate "net_sendfile" on page 104 calls to "net_write" on page 105 calls. As a result, filters interested in the outgoing data stream do not need to implement the sendfile filter method. However, for performance reasons, filters that implement the write filter method should also implement the sendfile filter method.

Syntax

```
int sendfile(FilterLayer *layer, const sendfiledata *data);
```

Returns

The number of bytes consumed, which may be less than the requested amount if an error occurred.

Parameters

FilterLayer *layer is the filter layer the filter is installed in.

const sendfiledata *sfd identifies the data to send.

Example

```
int myfilter_sendfile(FilterLayer *layer, const sendfiledata *sfd)
{
    return net_sendfile(layer->lower, sfd);
}
```

See Also

"net_sendfile" on page 104

session_create

The `session_create` function creates a new Session structure for the client with a specified socket descriptor and a specified socket address. The function returns a pointer to that structure.

Syntax

```
#include <base/session.h>
Session *session_create(SYS_NETFD csd, struct sockaddr_in *sac);
```

Returns

- A pointer to the new Session structure f one was created.
- NULL if no new Session structure was created.

Parameters

`SYS_NETFD` *csd* is the platform-independent socket descriptor.

`sockaddr_in` **sac* is the socket address.

See Also

"session_maxdns" on page 141

session_dns

The `session_dns` function resolves the IP address of the client associated with a specified session into its DNS name. The function returns a newly allocated string. You can use `session_dns` to change the numeric IP address into something more readable.

The `session_maxdns` function verifies the client identification information. The `session_dns` function does not perform this verification.

Note – This function works only if the `DNS` directive is enabled in the `obj.conf` file. For more information, see *Oracle iPlanet Web Proxy Server 4.0.14 Configuration File Reference*.

Syntax

```
char *session_dns(Session *sn);
```

Returns

A string containing the host name, or NULL if the DNS name cannot be found for the IP address.

Parameters

`Session *sn` identifies the Session structure.

The `Session` is the same as the Session structure passed to your SAF.

session_free

The `session_free` function frees a specified Session structure. The `session_free` function does not close the client socket descriptor associated with the Session structure.

Syntax

```
#include <base/session.h>
void session_free(Session *sn);
```

Returns

`void`

See Also

"session_create" on page 140, "session_maxdns" on page 141

Parameters

`Session` **sn* is the Session structure to be freed.

session_maxdns

The `session_maxdns` function resolves the IP address of the client associated with a specified session into its DNS name. The function returns a newly allocated string. You can use `session_maxdns` to change the numeric IP address into more readable format.

Note – This function works only if the `DNS` directive is enabled in the `obj.conf` file. For more information, see *Oracle iPlanet Web Proxy Server 4.0.14 Configuration File Reference*.

Syntax

```
char *session_maxdns(Session *sn);
```

Returns

A string containing the host name, or NULL if the DNS name cannot be found for the IP address.

Parameters

Session *sn identifies the Session structure.

The Session is the same as the Session structure passed to your SAF.

shexp_casecmp

The shexp_casecmp function validates a specified shell expression and compares it with a specified string. The function returns one of three possible values representing match, no match, and invalid comparison. This comparison, in contrast to that of the shexp_cmp function, is not case sensitive.

Use this function if you have a shell expression like *.netscape.com and you want to make sure that a string matches it, such as foo.netscape.com.

Syntax

```
int shexp_casecmp(char *str, char *exp);
```

Returns

0 if a match was found.

1 if no match was found.

-1 if the comparison resulted in an invalid expression.

Parameters

char *str is the string to be compared.

char *exp is the shell expression (wildcard pattern) to compare against.

See Also

"shexp_cmp" on page 142, "shexp_match" on page 143, "shexp_valid" on page 144

shexp_cmp

The shexp_casecmp function validates a specified shell expression and compares the expression with a specified string. The function returns one of three possible values representing match, no match, and invalid comparison. This comparison, in contrast to the comparison made by the shexp_casecmp function, is case sensitive.

Use this function if you have a shell expression like *.netscape.com and you want to make sure that a string matches it, such as foo.netscape.com.

Syntax

```
int shexp_cmp(char *str, char *exp);
```

Returns

`0` if a match was found.

`1` if no match was found.

`-1` if the comparison resulted in an invalid expression.

Parameters

`char *str` is the string to be compared.

`char *exp` is the shell expression (wildcard pattern) to compare against.

Example

```
/* Use wildcard match to see if this path is one we want */
char *path;char *match = "/usr/netscape/*";if (shexp_cmp(path, match) != 0)
    return REQ_NOACTION;   /* no match */
```

See Also

"shexp_casecmp" on page 142, "shexp_match" on page 143, "shexp_valid" on page 144

shexp_match

The `shexp_match` function compares a specified prevalidated shell expression against a specified string. The function returns one of three possible values representing match, no match, and invalid comparison. This comparison, in contrast to the comparison made by the `shexp_casecmp` function, is case sensitive.

The `shexp_match` function doesn't perform validation of the shell expression. The function assumes `shexp_valid` have already been called .

Use this function if you have a shell expression such as `*.netscape.com` and you want to make sure that a string matches it, such as `foo.netscape.com`.

Syntax

```
int shexp_match(char *str, char *exp);
```

Returns

0 if a match was found.

1 if no match was found.

-1 if the comparison resulted in an invalid expression.

Parameters

char *str is the string to be compared.

char *exp is the prevalidated shell expression (wildcard pattern) to compare against.

See Also

"shexp_casecmp" on page 142, "shexp_cmp" on page 142, "shexp_valid" on page 144

shexp_valid

The shexp_valid function validates a specified shell expression named by exp. Use this function to validate a shell expression before using the function shexp_match to compare the expression with a string.

Syntax

```
int shexp_valid(char *exp);
```

Returns

The constant NON_SXP if exp is a standard string.

The constant INVALID_SXP if exp is a shell expression but is invalid.

The constant VALID_SXP if exp is a valid shell expression.

Parameters

char *exp is the shell expression (wildcard pattern) to be validated.

See Also

"shexp_casecmp" on page 142, "shexp_match" on page 143, "shexp_cmp" on page 142

shmem_alloc

The `shmem_alloc` function allocates a region of shared memory of the given size, using the given name to avoid conflicts between multiple regions in the program. The size of the region will not be automatically increased if its boundaries are overrun. Use the `shmem_realloc` function for that automatic increases.

This function must be called before any daemon workers are spawned in order for the handle to the shared region to be inherited by the children.

Because the region must be inherited by the children, the region cannot be reallocated with a larger size when necessary.

Syntax

```
#include <base/shmem.h>
shmem_s *shmem_alloc(char *name, int size, int expose);
```

Returns

A pointer to a new shared memory region.

Parameters

`char` **name* is the name for the region of shared memory being created. The value of *name* must be unique to the program that calls the `shmem_alloc()` function or conflicts will occur.

`int` *size* is the number of characters of memory to be allocated for the shared memory.

`int` *expose* is either zero or nonzero. If nonzero, then on systems that support it, the file that is used to create the shared memory becomes visible to other processes running on the system.

See Also

“shmem_free” on page 145

shmem_free

The `shmem_free` function deallocates (frees) the specified region of memory.

Syntax

```
#include <base/shmem.h>
void *shmem_free(shmem_s *region);
```

Returns

`void`

Parameters

shmem_s **region* is a shared memory region to be released.

See Also

"shmem_alloc" on page 145

STRDUP

The STRDUP macro is a platform-independent substitute for the C library routine strdup. It creates a new copy of a string in the request's memory pool.

The STRDUP routine is functionally equivalent to:

```
newstr = (char *) MALLOC(strlen(str) + 1);
strcpy(newstr, str);
```

A string created with STRDUP should be disposed with FREE.

Syntax

```
char *STRDUP(char *ptr);
```

Returns

A pointer to the new string.

Parameters

char *ptr is a pointer to a string.

Example

```
char *name1 = "MyName";char *name2 = STRDUP(name1);
```

See Also

"MALLOC" on page 101, "FREE" on page 95, "CALLOC" on page 77, "REALLOC" on page 133, "PERM_MALLOC" on page 123, "PERM_FREE" on page 122, "PERM_CALLOC" on page 121, "PERM_REALLOC" on page 123, "PERM_STRDUP" on page 124

system_errmsg

The system_errmsg function returns the last error that occurred from the most recent system call. This function is implemented as a macro that returns an entry from the global array sys_errlist. Use this macro to help with I/O error diagnostics.

Syntax

```
char *system_errmsg(int param1);
```

Returns

A string containing the text of the latest error message that resulted from a system call. Do not FREE this string.

Parameters

int param1 is reserved, and should always have the value 0.

See Also

"system_fopenRO" on page 148, "system_fopenRW" on page 149, "system_fopenWA" on page 149, "system_lseek" on page 153, "system_fread" on page 150, "system_fwrite" on page 151, "system_fwrite_atomic" on page 151, "system_flock" on page 148, "system_ulock" on page 154, "system_fclose" on page 147

system_fclose

The system_fclose function closes a specified file descriptor. The system_fclose function must be called for every file descriptor opened by any of the system_fopen functions.

Syntax

```
int system_fclose(SYS_FILE fd);
```

Returns

0 if the close succeeded, or IO_ERROR if the close failed.

Parameters

SYS_FILE fd is the platform-independent file descriptor.

Example

```
SYS_FILE logfd; system_fclose(logfd);
```

See Also

"system_errmsg" on page 147, "system_fopenRO" on page 148, "system_fopenRW" on page 149, "system_fopenWA" on page 149, "system_lseek" on page 153, "system_fread" on page 150, "system_fwrite" on page 151, "system_fwrite_atomic" on page 151, "system_flock" on page 148, "system_ulock" on page 154

system_flock

The `system_flock` function locks the specified file against interference from other processes. Use `system_flock` if you do not want other processes to use the file you currently have open. Overusing file locking can cause performance degradation and possibly lead to deadlocks.

Syntax

```
int system_flock(SYS_FILE fd);
```

Returns

Returns `IO_OKAY` if the lock succeeded, or `IO_ERROR` if the lock failed.

Parameters

`SYS_FILE fd` is the platform-independent file descriptor.

See Also

"system_errmsg" on page 147, "system_fopenRO" on page 148, "system_fopenRW" on page 149, "system_fopenWA" on page 149, "system_lseek" on page 153, "system_fread" on page 150, "system_fwrite" on page 151, "system_fwrite_atomic" on page 151, "system_ulock" on page 154, "system_fclose" on page 147

system_fopenRO

The `system_fopenRO` function opens the file identified by `path` in read-only mode and returns a valid file descriptor. Use this function to open files that will not be modified by your program. In addition, you can use `system_fopenRO` to open a new file buffer structure using `filebuf_open`.

Syntax

```
SYS_FILE system_fopenRO(char *path);
```

Returns

The system-independent file descriptor (`SYS_FILE`) if the open succeeded, or `0` if the open failed.

Parameters

`char *path` is the file name.

See Also

"system_errmsg" on page 147, "system_fopenRO" on page 148, "system_fopenWA" on page 149, "system_lseek" on page 153, "system_fread" on page 150, "system_fwrite" on page 151, "system_fwrite_atomic" on page 151, "system_flock" on page 148, "system_ulock" on page 154, "system_fclose" on page 147

system_fopenRW

The `system_fopenRW` function opens the file identified by `path` in read-write mode and returns a valid file descriptor. If the file already exists, `system_fopenRW` does not truncate it. Use this function to open files that will be read from and written to by your program.

Syntax

```
SYS_FILE system_fopenRW(char *path);
```

Returns

The system-independent file descriptor (`SYS_FILE`) if the open succeeded, or `0` if the open failed.

Parameters

`char *path` is the file name.

Example

```
SYS_FILE fd;fd = system_fopenRO(pathname);if (fd == SYS_ERROR_FD)    break;
```

See Also

"system_errmsg" on page 147, "system_fopenRO" on page 148, "system_fopenWA" on page 149, "system_lseek" on page 153, "system_fread" on page 150, "system_fwrite" on page 151, "system_fwrite_atomic" on page 151, "system_flock" on page 148, "system_ulock" on page 154, "system_fclose" on page 147

system_fopenWA

The `system_fopenWA` function opens the file identified by `path` in write-append mode and returns a valid file descriptor. Use this function to open those files to which your program will append data.

Syntax

```
SYS_FILE system_fopenWA(char *path);
```

Returns

The system-independent file descriptor (`SYS_FILE`) if the open succeeded, or `0` if the open failed.

Parameters

`char *path` is the file name.

See Also

"system_errmsg" on page 147, "system_fopenRO" on page 148, "system_fopenRW" on page 149, "system_lseek" on page 153, "system_fread" on page 150, "system_fwrite" on page 151, "system_fwrite_atomic" on page 151, "system_flock" on page 148, "system_ulock" on page 154, "system_fclose" on page 147

system_fread

The `system_fread` function reads a specified number of bytes from a specified file into a specified buffer. It returns the number of bytes read. Before `system_fread` can be used, you must open the file using any of the `system_fopen` functions except `system_fopenWA`.

Syntax

```
int system_fread(SYS_FILE fd, char *buf, int sz);
```

Returns

The number of bytes read, which might be less than the requested size if an error occurred or the end of the file was reached before that number of characters were obtained.

Parameters

`SYS_FILE fd` is the platform-independent file descriptor.

`char *buf` is the buffer to receive the bytes.

`int sz` is the number of bytes to read.

See Also

“system_errmsg” on page 147, “system_fopenRO” on page 148, “system_fopenRW” on page 149, “system_fopenWA” on page 149, “system_lseek” on page 153, “system_fwrite” on page 151, “system_fwrite_atomic” on page 151, “system_flock” on page 148, “system_ulock” on page 154, “system_fclose” on page 147

system_fwrite

The `system_fwrite` function writes a specified number of bytes from a specified buffer into a specified file.

Before `system_fwrite` can be used, you must open the file using any of the `system_fopen` functions except `system_fopenRO`.

Syntax

```
int system_fwrite(SYS_FILE fd, char *buf, int sz);
```

Returns

Returns `IO_OKAY` if the write succeeded, or `IO_ERROR` if the write failed.

Parameters

`SYS_FILE fd` is the platform-independent file descriptor.

`char *buf` is the buffer containing the bytes to be written.

`int sz` is the number of bytes to write to the file.

See Also

“system_errmsg” on page 147, “system_fopenRO” on page 148, “system_fopenRW” on page 149, “system_fopenWA” on page 149, “system_lseek” on page 153, “system_fread” on page 150, “system_fwrite_atomic” on page 151, “system_flock” on page 148, “system_ulock” on page 154, “system_fclose” on page 147

system_fwrite_atomic

The `system_fwrite_atomic` function writes a specified number of bytes from a specified buffer into a specified file. The function also locks the file prior to performing the write, and then unlocks it when done. This process avoids interference between simultaneous write actions. Before `system_fwrite_atomic` can be used, you must open the file using any of the `system_fopen` functions except `system_fopenRO`.

Syntax

```
int system_fwrite_atomic(SYS_FILE fd, char *buf, int sz);
```

Returns

Returns `IO_OKAY` if the write/lock succeeded, or `IO_ERROR` if the write/lock failed.

Parameters

`SYS_FILE fd` is the platform-independent file descriptor.

`char *buf` is the buffer containing the bytes to be written.

`int sz` is the number of bytes to write to the file.

Example

```
SYS_FILE logfd;char *logmsg = "An error occurred.";
    system_fwrite_atomic(logfd, logmsg, strlen(logmsg));
```

See Also

"system_errmsg" on page 147, "system_fopenRO" on page 148, "system_fopenRW" on page 149, "system_fopenWA" on page 149, "system_lseek" on page 153, "system_fread" on page 150, "system_fwrite" on page 151, "system_flock" on page 148, "system_ulock" on page 154, "system_fclose" on page 147

system_gmtime

The `system_gmtime` function is a thread-safe version of the standard `gmtime` function. It returns the current time adjusted to Greenwich Mean Time.

Syntax

```
struct tm *system_gmtime(const time_t *tp, const struct tm *res);
```

Returns

A pointer to a calendar time (`tm`) structure containing the GMT time. Depending on your system, the pointer may point to the data item represented by the second parameter, or it may point to a statically allocated item. For portability, do not assume either situation.

Parameters

`time_t *tp` is an arithmetic time.

`tm *res` is a pointer to a calendar time (`tm`) structure.

Example

```
time_t tp;struct tm res, *resp;tp = time(NULL);
    resp = system_gmtime(&tp, &res);
```

See Also

"system_localtime" on page 153, "util_strftime" on page 183

system_localtime

The `system_localtime` function is a thread-safe version of the standard `localtime` function. It returns the current time in the local time zone.

Syntax

```
struct tm *system_localtime(const time_t *tp, const struct tm *res);
```

Returns

A pointer to a calendar time (`tm`) structure containing the local time. Depending on your system, the pointer may point to the data item represented by the second parameter, or it may point to a statically allocated item. For portability, do not assume either situation.

Parameters

`time_t *tp` is an arithmetic time.

`tm *res` is a pointer to a calendar time (`tm`) structure.

See Also

"system_gmtime" on page 152, "util_strftime" on page 183

system_lseek

The `system_lseek` function sets the file position of a file. This function affects where data from `system_fread` or `system_fwrite` is read or written.

Syntax

```
int system_lseek(SYS_FILE fd, int offset, int whence);
```

Returns

The offset, in bytes, of the new position from the beginning of the file if the operation succeeded, or `-1` if the operation failed.

Parameters

SYS_FILE fd is the platform-independent file descriptor.

int offset is a number of bytes relative to whence. It may be negative.

int whence is one of the following constants:

SEEK_SET, from the beginning of the file.

SEEK_CUR, from the current file position.

SEEK_END, from the end of the file.

See Also

"system_errmsg" on page 147, "system_fopenRO" on page 148, "system_fopenRW" on page 149, "system_fopenWA" on page 149, "system_fread" on page 150, "system_fwrite" on page 151, "system_fwrite_atomic" on page 151, "system_flock" on page 148, "system_ulock" on page 154, "system_fclose" on page 147

system_rename

The system_rename function renames a file. This function might not work on directories if the old and new directories are on different file systems.

Syntax

```
int system_rename(char *old, char *new);
```

Returns

0 if the operation succeeded, or -1 if the operation failed.

Parameters

char *old is the old name of the file.

char *new is the new name for the file.

system_ulock

The system_ulock function unlocks the specified file that has been locked by the function system_lock. For more information about locking, see system_flock.

Syntax

```
int system_ulock(SYS_FILE fd);
```

Returns

Returns `IO_OKAY` if the operation succeeded, or `IO_ERROR` if the operation failed.

Parameters

`SYS_FILE fd` is the platform-independent file descriptor.

See Also

`system_errmsg`, `system_fopenRO`, `system_fopenRW`, `system_fopenWA`, `system_fread`, `system_fwrite`, `system_fwrite_atomic`, `system_flock`, `system_fclose`

"system_errmsg" on page 147, "system_fopenRO" on page 148, "system_fopenRW" on page 149, "system_fopenWA" on page 149, "system_fread" on page 150, "system_fwrite" on page 151, "system_fwrite_atomic" on page 151, "system_flock" on page 148, "system_fclose" on page 147

system_unix2local

The `system_unix2local` function converts a specified UNIX-style path name to a local file system path name. Use this function when you have a file name in the UNIX format using forward slashes, and you need to access a file on another system such as Windows. You can use `system_unix2local` to convert the UNIX file name into the format that Windows accepts. In the UNIX environment this function does nothing, but may be called for portability.

Syntax

```
char *system_unix2local(char *path, char *lp);
```

Returns

A pointer to the local file system path string.

Parameters

`char *path` is the UNIX-style path name to be converted.

`char *lp` is the local path name.

You must allocate the parameter `lp`. The parameter must contain enough space to hold the local path name.

See Also

"system_fclose" on page 147, "system_flock" on page 148, "system_fopenRO" on page 148, "system_fopenRW" on page 149, "system_fopenWA" on page 149, "system_fwrite" on page 151

systhread_attach

The `systhread_attach` function converts an existing thread into a platform-independent thread.

Syntax

```
SYS_THREAD systhread_attach(void);
```

Returns

A `SYS_THREAD` pointer to the platform-independent thread.

Parameters

none

See Also

"systhread_current" on page 156, "systhread_getdata" on page 157, "systhread_init" on page 157, "systhread_newkey" on page 158, "systhread_setdata" on page 158, "systhread_sleep" on page 159, "systhread_start" on page 159, "systhread_timerset" on page 160

systhread_current

The `systhread_current` function returns a pointer to the current thread.

Syntax

```
SYS_THREAD systhread_current(void);
```

Returns

A `SYS_THREAD` pointer to the current thread.

Parameters

none

See Also

"systhread_getdata" on page 157, "systhread_newkey" on page 158, "systhread_setdata" on page 158, "systhread_sleep" on page 159, "systhread_start" on page 159, "systhread_timerset" on page 160

systhread_getdata

The systhread_getdata function gets data that is associated with a specified key in the current thread.

Syntax

```
void *systhread_getdata(int key);
```

Returns

A pointer to the data that was earlier used with the systhread_setkey function from the current thread, using the same value of key if the call succeeds. Returns NULL if the call did not succeed; for example, if the systhread_setkey function was never called with the specified key during this session.

Parameters

int key is the value associated with the stored data by a systhread_setdata function. Keys are assigned by the systhread_newkey function.

See Also

"systhread_current" on page 156, "systhread_newkey" on page 158, "systhread_setdata" on page 158, "systhread_sleep" on page 159, "systhread_start" on page 159, "systhread_timerset" on page 160

systhread_init

The systhread_init function initializes the threading system.

Syntax

```
#include <base/systhr.h>
void systhread_init(char *name);
```

Returns

void

Parameters

char **name* is a name to be assigned to the program for debugging purposes.

See also

systhread_attach, systhread_current, systhread_getdata, systhread_newkey, systhread_setdata, systhread_sleep,systhread_start, systhread_terminate, systhread_ timerset

systhread_newkey

The `systhread_newkey` function allocates a new integer key (identifier) for thread-private data. Use this key to identify a variable that you want to localize to the current thread, then use the `systhread_setdata` function to associate a value with the key.

Syntax

```
int systhread_newkey(void);
```

Returns

An integer key.

Parameters

none

See Also

"systhread_current" on page 156, "systhread_getdata" on page 157, "systhread_setdata" on page 158, "systhread_sleep" on page 159, "systhread_start" on page 159, "systhread_timerset" on page 160

systhread_setdata

The `systhread_setdata` function associates data with a specified key number for the current thread. Keys are assigned by the `systhread_newkey` function.

Syntax

```
void systhread_setdata(int key, void *data);
```

Returns

`void`

Parameters

`int key` is the priority of the thread.

`void *data` is the pointer to the string of data to be associated with the value of `key`.

See Also

"systhread_current" on page 156, "systhread_getdata" on page 157, "systhread_newkey" on page 158, "systhread_sleep" on page 159, "systhread_start" on page 159, "systhread_timerset" on page 160

systhread_sleep

The `systhread_sleep` function puts the calling thread to sleep for a given time.

Syntax

```
void systhread_sleep(int milliseconds);
```

Returns

`void`

Parameters

`int milliseconds` is the number of milliseconds the thread is to sleep.

See Also

"systhread_current" on page 156, "systhread_getdata" on page 157, "systhread_newkey" on page 158, "systhread_setdata" on page 158, "systhread_start" on page 159, "systhread_timerset" on page 160

systhread_start

The `systhread_start` function creates a thread with the given priority, allocates a stack of a specified number of bytes, and calls a specified function with a specified argument.

Syntax

```
SYS_THREAD systhread_start(int prio, int stksz,    void (*fn)(void *),
    void *arg);
```

Returns

A new `SYS_THREAD` pointer if the call succeeded, or `SYS_THREAD_ERROR` if the call did not succeed.

Parameters

int prio is the priority of the thread. Priorities are system-dependent.

int stksz is the stack size in bytes. If stksz is zero (0), the function allocates a default size.

void (*fn)(void *) is the function to call.

void *arg is the argument for the fn function.

See Also

"systhread_current" on page 156, "systhread_getdata" on page 157, "systhread_newkey" on page 158, "systhread_setdata" on page 158, "systhread_sleep" on page 159, "systhread_timerset" on page 160

systhread_terminate

The systhread_terminate function terminates a specified thread.

Syntax

```
#include <base/systhr.h>
void systhread_terminate(SYS_THREAD thr);
```

Returns

void

Parameters

SYS_THREAD *thr* is the thread to terminate.

See Also

"systhread_current" on page 156, "systhread_getdata" on page 157, "systhread_newkey" on page 158, "systhread_setdata" on page 158, "systhread_sleep" on page 159, "systhread_start" on page 159, "systhread_timerset" on page 160

systhread_timerset

The systhread_timerset function starts or resets the interrupt timer interval for a thread system.

Because most systems don't allow the timer interval to be changed, this function should be considered a suggestion rather than a command.

Syntax

```
void systhread_timerset(int usec);
```

Returns

```
void
```

Parameters

`int usec` is the time, in microseconds

See Also

"systhread_current" on page 156, "systhread_getdata" on page 157, "systhread_newkey" on page 158, "systhread_setdata" on page 158, "systhread_sleep" on page 159, "systhread_start" on page 159

U

USE_NSAPI_VERSION

To request a particular version of NSAPI, define the `USE_NSAPI_VERSION` macro before including the `nsapi.h` header file. The requested NSAPI version is encoded by multiplying the major version number by 100 and then adding this to the minor version number. For example, the following code requests NSAPI 3.2 features:

```
#define USE_NSAPI_VERSION 302 /* We want NSAPI 3.2 (Web Server 6.1) */
#include "nsapi.h"
```

To develop a plug-in that is compatible across multiple server versions, define `USE_NSAPI_VERSION` to the highest NSAPI version supported by all of the target server versions.

The following table lists server versions and the highest NSAPI version supported by each:

TABLE 4–2 NSAPI Versions Supported by Different Servers

Server Version	NSAPI Version
Sun iPlanet Web Server 4.1	3.0

TABLE 4–2 NSAPI Versions Supported by Different Servers *(Continued)*

Server Version	NSAPI Version
Sun iPlanet Web Server 6.0	3.1
Netscape Enterprise Server 6.0	3.1
Netscape Enterprise Server 6.1	3.1
Sun ONE Application Server 7.0	3.1
Sun Java System Web Server 6.1	3.2
Oracle iPlanet Web Server 7.0.9	3.3
iPlanet Web Proxy Server 4	3.3

Do not request a version of NSAPI higher than the highest version supported by the `nsapi.h` header that the plug-in is being compiled against. Additionally, to use `USE_NSAPI_VERSION`, you must compile against an `nsapi.h` header file that supports NSAPI 3.3 or higher.

Syntax

```
int USE_NSAPI_VERSION
```

Example

The following code can be used when building a plug-in designed to work with Proxy Server 4:

```
#define USE_NSAPI_VERSION 303 /* We want NSAPI 3.3 (Proxy Server 4) */
#include "nsapi.h"
```

See Also

"NSAPI_RUNTIME_VERSION" on page 109, "NSAPI_VERSION" on page 110

util_can_exec

UNIX Only

The `util_can_exec` function checks that a specified file can be executed, returning either a `1` (executable) or a `0`. The function checks whether the file can be executed by the user with the given user and group ID.

Use this function before executing a program using the `exec` system call.

Syntax

```
int util_can_exec(struct stat *finfo, uid_t uid, gid_t gid);
```

Returns

1 if the file is executable, or 0 if the file is not executable.

Parameters

stat *finfo is the stat structure associated with a file.

uid_t uid is the UNIX user ID.

gid_t gid is the UNIX group ID. Together with uid, this value determines the permissions of the UNIX user.

See Also

"util_env_create" on page 164, "util_getline" on page 171, "util_hostname" on page 172

util_chdir2path

The util_chdir2path function changes the current directory to a specified directory where you will access a file.

When running under Windows, use a critical section to ensure that more than one thread does not call this function at the same time.

Using util_chdir2path makes file access a little quicker because this function does not require a full path.

Syntax

```
int util_chdir2path(char *path);
```

Returns

0 if the directory was changed, or -1 if the directory could not be changed.

Parameters

char *path is the name of a directory.

The parameter must be a writable string because the string is not permanently modified.

util_cookie_find

The util_cookie_find function finds a specific cookie in a cookie string and returns its value.

Syntax

```
char *util_cookie_find(char *cookie, char *name);
```

Returns

If successful, returns a pointer to the NULL-terminated value of the cookie. Otherwise, returns NULL. This function modifies the cookie string parameter by NULL-terminating the name and value.

Parameters

`char *cookie` is the value of the `Cookie:` request header.

`char *name` is the name of the cookie whose value is to be retrieved.

util_does_process_exist

The `util_does_process_exist` function verifies that a given process ID is that of an executing process.

Syntax

```
#include <libproxy/util.h>
int util_does_process_exist (int pid)
```

Returns

- nonzero if the *pid* represents an executing process
- 0 if the *pid* does not represent an executing process

Parameters

`int` *pid* is the process ID to be tested.

See Also

"util_url_fix_host name" on page 188, "util_uri_check" on page 184

util_env_create

The `util_env_create` function creates and allocates the environment specified by *env*, returning a pointer to the environment. If the parameter *env* is NULL, the function allocates a new environment. Use `util_env_create` to create an environment when executing a new program.

Syntax

```
#include <base/util.h>
char **util_env_create(char **env, int n, int *pos);
```

Returns

A pointer to an environment.

Parameters

`char **env` is the existing environment or NULL.

`int` *n* is the maximum number of environment entries that you want in the environment.

`int` **pos* is an integer that keeps track of the number of entries used in the environment.

See Also

"util_env_replace" on page 166, "util_env_str" on page 167 , "util_env_free" on page 166 , "util_env_find" on page 165

util_env_find

The `util_env_find` function locates the string denoted by a name in a specified environment and returns the associated value. Use this function to find an entry in an environment.

Syntax

```
char *util_env_find(char **env, char *name);
```

Returns

The value of the environment variable if it is found, or NULL if the string was not found.

Parameters

`char **env` is the environment.

`char *name` is the name of an environment variable in `env`.

See Also

"util_env_replace" on page 166, "util_env_str" on page 167 , "util_env_free" on page 166 , "util_env_create" on page 164

util_env_free

The `util_env_free` function frees a specified environment. Use this function to deallocate an environment that you created using the function `util_env_create`.

Syntax

```
void util_env_free(char **env);
```

Returns

`void`

Parameters

`char **env` is the environment to be freed.

See Also

"util_env_replace" on page 166, "util_env_str" on page 167 , "util_env_create" on page 164 , "util_env_find" on page 165

util_env_replace

The `util_env_replace` function replaces the occurrence of the variable denoted by a name in a specified environment with a specified value. Use this function to change the value of a setting in an environment.

Syntax

```
void util_env_replace(char **env, char *name, char *value);
```

Returns

`void`

Parameters

`char **env` is the environment.

`char *name` is the name of a name-value pair.

`char *value` is the new value to be stored.

See Also

"util_env_str" on page 167, "util_env_free" on page 166 , "util_env_find" on page 165, "util_env_create" on page 164

util_env_str

The `util_env_str` function creates an environment entry and returns it. This function does not check for nonalphanumeric symbols in the name such as the equal sign "=". You can use this function to create a new environment entry.

Syntax

```
char *util_env_str(char *name, char *value);
```

Returns

A newly allocated string containing the name-value pair.

Parameters

`char *name` is the name of a name-value pair.

`char *value` is the new value to be stored.

See Also

"util_env_replace" on page 166, "util_env_free" on page 166 , "util_env_create" on page 164 , "util_env_find" on page 165

util_get_current_gmt

The `util_get_current_gmt` function obtains the current time, represented in terms of GMT (Greenwich Mean Time).

Syntax

```
#include <libproxy/util.h>
time_t util_get_current_gmt(void);
```

Returns

the current GMT

Parameters

No parameter is required.

See Also

"util_make_local" on page 175

util_get_int_from_aux_file

The `util_get_int_from_aux_file` function is used to get a single line from a specified file and return it in the form of an integer. This ifunction enables you to store single numbers in a file.

Syntax

```
#include <libproxy/cutil.h>
int util_get_int_from_file(char *root, char *name);
```

Returns

An integer from the file.

Parameters

`char*root` is the name of the directory containing the file to be read.

`char*name` is the name of the file to be read.

See Also

"util_get_long_from_aux_file" on page 169, "util_get_string_from_aux_file" on page 170, "util_get_int_from_file" on page 168, "util_get_long_from_file" on page 169, "util_get_string_from_file" on page 171, "util_put_int_to_file" on page 177, "util_put_long_to_file" on page 178, "util_put_string_to_aux_file" on page 179, "util_put_string_to_file" on page 179

util_get_int_from_file

The `util_get_int_from_file` function is used to get a single line from a specified file and return it in the form of an integer. This function enables you to store single numbers in a file.

Syntax

```
#include <libproxy/cutil.h>
int util_get_int_from_file(char *filename);
```

Returns

- An integer from the file.
- `-1` if no value was obtained from the file.

Parameters

`char` *filename* is the name of the file to be read.

See Also

"util_get_long_from_file" on page 169, "util_get_string_from_file" on page 171, "util_put_int_to_file" on page 177, "util_put_long_to_file" on page 178, "util_put_string_to_file" on page 179

util_get_long_from_aux_file

The `util_get_long_from_file` function is used to get a single line from a specified file and return it in the form of a long number. This function enables you to store single long numbers in a file.

Syntax

```
#include <libproxy/cutil.h>
long util_get_long_from_file(char *root,char *name);
```

Returns

A long integer from the file.

Parameters

`char` **root* is the name of the directory containing the file to be read.

`char` **name* is the name of the file to be read.

See Also

"util_get_int_from_aux_file" on page 168, "util_get_string_from_aux_file" on page 170, "util_get_int_from_file" on page 168, "util_get_long_from_file" on page 169, "util_get_string_from_file" on page 171, "util_put_int_to_file" on page 177, "util_put_long_to_file" on page 178, "util_put_string_to_aux_file" on page 179, "util_put_string_to_file" on page 179

util_get_long_from_file

The `util_get_long_from_file` function is used to get a single line from a specified file and return it in the form of a long number. This function enables you to store single long numbers in a file.

Syntax

```
#include <libproxy/cutil.h>
long util_get_long_from_file(char *filename);
```

Returns

- A long integer from the file.
- `-1` if no value was obtained from the file.

Parameters

`char` **file* is the name of the file to be read.

See Also

"util_get_int_from_file" on page 168, "util_get_string_from_file" on page 171, "util_put_int_to_file" on page 177, "util_put_long_to_file" on page 178, "util_put_string_to_file" on page 179

util_get_string_from_aux_file

The `util_get_string_from_aux_file` function is used to get a single line from a specified file and return it in the form of a word. This function enables you to store single words in a file.

Syntax

```
#include <libproxy/cutil.h>
char *util_get_string_from_file(char *root, char *name, char *buf, int maxsize);
```

Returns

A string containing the next line from the file.

Parameters

`char` **root* is the name of the directory containing the file to be read.

`char` **name* is the name of the file to be read.

`char` **buf* is the string to use as the file buffer.

`int` *maxsize* is the maximum size for the file buffer.

See Also

"util_get_int_from_aux_file" on page 168, "util_get_long_from_aux_file" on page 169, "util_get_int_from_file" on page 168, "util_get_long_from_file" on page 169, "util_get_string_from_file" on page 171, "util_put_int_to_file" on page 177, "util_put_long_to_file" on page 178, "util_put_string_to_aux_file" on page 179, "util_put_string_to_file" on page 179

util_get_string_from_file

The `util_get_string_from_file` function is used to get a single line from a specified file and return it in the form of a word. This function enables you to store single words in a file.

Syntax

```
#include <libproxy/cutil.h>
char *util_get_string_from_file(char *filename, char *buf, int maxsize);
```

Returns

- A string containing the next line from the file.
- NULL if no string was obtained.

Parameters

`char` **file* is the name of the file to be read.

`char` **buf* is the string to use as the file buffer.

`int` *maxsize* is the maximum size for the file buffer.

See Also

"util_get_int_from_file" on page 168, "util_get_long_from_file" on page 169, "util_put_int_to_file" on page 177, "util_put_long_to_file" on page 178, "util_put_string_to_file" on page 179

util_getline

The `util_getline` function scans the specified file buffer to find a line feed or carriage return/line feed terminated string. The string is copied into the specified buffer, and NULL-terminates it. The function returns a value that indicates whether the operation stored a string in the buffer, encountered an error, or reached the end of the file.

Use this function to scan lines out of a text file, such as a configuration file.

Syntax

```
int util_getline(filebuf *buf, int lineno, int maxlen, char *line);
```

Returns

0 if successful line contains the string.

1 if the end of file was reached line contains the string.

-1 if an error occurred line contains a description of the error.

Parameters

filebuf *buf is the file buffer to be scanned.

int lineno is used to include the line number in the error message when an error occurs. The caller is responsible for making sure that the line number is accurate.

int maxlen is the maximum number of characters that can be written into l.

char *l is the buffer in which to store the string. The user is responsible for allocating and deallocating line.

See Also

"util_can_exec" on page 162, "util_env_create" on page 164, "util_hostname" on page 172

util_hostname

The util_hostname function retrieves the local host name and returns it as a string. If the function cannot find a fully qualified domain name, it returns NULL. You may reallocate or free this string. Use this function to determine the name of the system you are on.

Syntax

```
char *util_hostname(void);
```

Returns

If a fully qualified domain name was found, returns a string containing that name. Otherwise, the function returns NULL if the fully qualified domain name was not found.

Parameters

none

util_is_mozilla

The util_is_mozilla function checks whether a specified user-agent header string is a Netscape browser of at least a specified revision level. The function uses strings to specify the revision level to avoid ambiguities such as 1.56 > 1.5.

Syntax

```
int util_is_mozilla(char *ua, char *major, char *minor);
```

Returns

1 if the user-agent is a Netscape browser, or 0 if the user-agent is not a Netscape browser.

Parameters

char *ua is the user-agent string from the request headers.

char *major is the major release number to the left of the decimal point.

char *minor is the minor release number to the right of the decimal point.

See Also

"util_is_url" on page 173, "util_later_than" on page 174

util_is_url

The util_is_url function checks whether a string is a URL. The string is a URL if it begins with alphabetic characters followed by a colon (:).

Syntax

```
int util_is_url(char *url);
```

Returns

1 if the string specified by url is a URL, or 0 if the string specified by url is not a URL.

Parameters

char *url is the string to be examined.

See Also

"util_is_mozilla" on page 172, "util_later_than" on page 174

util_itoa

The util_itoa function converts a specified integer to a string, and returns the length of the string. Use this function to create a textual representation of a number.

Syntax

```
int util_itoa(int i, char *a);
```

Returns

The length of the string created.

Parameters

`int i` is the integer to be converted.

`char *a` is the ASCII string that represents the value. The user is responsible for the allocation and deallocation of `a`. The string should be at least 32 bytes long.

util_later_than

The `util_later_than` function compares the date specified in a time structure against a date specified in a string. If the date in the string is later than or equal to the one in the time structure, the function returns `1`. Use this function to handle RFC 822, RFC 850, and `ctime` formats.

Syntax

```
int util_later_than(struct tm *lms, char *ims);
```

Returns

`1` if the date represented by `ims` is the same as or later than that represented by the `lms`, or `0` if the date represented by `ims` is earlier than that represented by the `lms.`

Parameters

`tm *lms` is the time structure containing a date.

`char *ims` is the string containing a date.

See Also

"util_strftime" on page 183

util_make_filename

The `util_make_filename` function concatenates a directory name and a file name into a newly created string. This function is useful when you are dealing with a number of files that all go to the same directory.

Syntax

```
#include <libproxy/cutil.h>
char *util_make_filename(char *root, char *name);
```

Returns

A new string containing the directory name concatenated with the file name.

Parameters

`char` **root* is a string containing the directory name.

`char` **name* is a string containing the file name.

util_make_gmt

The `util_make_gmt` function converts a given local time to GMT (Greenwich Mean Time), or obtains the current GMT.

Syntax

```
#include <libproxy/util.h>
time_t util_make_gmt(time_t t);
```

Returns

- The GMT equivalent to the local time *t*, if t is not 0
- The current GMT if *t* is 0

Parameters

`time_t` *t* is a time.

See Also

"util_make_local" on page 175

util_make_local

The `util_make_local` function converts a given GMT to local time.

Syntax

```
#include <libproxy/util.h>
time_t util_make_local(time_t t);
```

Returns

The local equivalent to the GMT *t*.

Parameters

`time_t` *t* is a time.

See Also

"util_make_gmt" on page 175

util_move_dir

The `util_move_dir` function moves a directory, preserving permissions, creation times, and last-access times. If renaming fails, for example, if the source and destination are on two different file systems, the function copies the directory.

Syntax

```
#include <libproxy/util.h>
int util_move_dir (char *src, char *dst);
```

Returns

- `0` if the move failed.
- nonzero if the move succeeded.

Parameters

`char` **src* is the fully qualified name of the source directory.

`char` **dst* is the fully qualified name of the destination directory.

See Also

"util_move_file" on page 176

util_move_file

The `util_move_dir` function moves a file, preserving permissions, creation time, and last-access time. If renaming fails, for example, if the source and destination are on two different file systems, the function copies the file.

Syntax

```
#include <libproxy/util.h>
int util_move_file (char *src, char *dst);
```

Returns

- `0` if the move failed.
- nonzero if the move succeeded.

Parameters

`char` **src* is the fully qualified name of the source file.

`char` **dst* is the fully qualified name of the destination file.

See Also

"util_move_dir" on page 176

util_parse_http_time

The `util_parse_http_time` function converts a given HTTP time string to `time_t` format.

Syntax

```
#include <libproxy/util.h>
time_t util_parse_http_time(char *date_string);
```

Returns

The `time_t` equivalent to the GMT *t*.

Parameters

`time_t` *t* is a time.

See Also

"util_make_gmt" on page 175

util_put_int_to_file

The `util_put_int_to_file` function writes a single line containing an integer to a specified file.

Syntax

```
#include <libproxy/cutil.h>
int util_put_int_to_file(char *filename, int i);
```

Returns

- nonzero if the operation succeeded
- 0 if the operation failed

Parameters

`char` **file* is the name of the file to be written.

`int` *i* is the integer to write.

See Also

“util_get_int_from_file” on page 168, “util_get_long_from_file” on page 169,
“util_put_long_to_file” on page 178, “util_put_string_to_file” on page 179

util_put_long_to_file

The `util_put_long_to_file` function writes a single line containing a long integer to a specified file.

Syntax

```
#include <libproxy/cutil.h>
ing util_put_long_to_file(char *filename, long l);
```

Returns

- nonzero if the operation succeeded
- 0 if the operation failed

Parameters

`char` **file* is the name of the file to be written.

`long` *l* is the long integer to write.

See Also

“util_get_int_from_file” on page 168, “util_get_long_from_file” on page 169,
“util_put_int_to_file” on page 177, “util_put_string_to_file” on page 179

util_put_string_to_aux_file

The `util_put_string_to_aux_file` function writes a single line containing a string to a file specified by directory name and file name.

Syntax

```
#include <libproxy/cutil.h>
int util_put_string_to_aux_file(char *root, char *name, char *str);
```

Returns

- non-zero if the operation succeeded.
- `0` if the operation failed.

Parameters

`char` **root* is the name of the directory where the file is to be written.

`char` **name* is the name of the file is to be written.

`char` **str* is the string to write.

See Also

"util_get_int_from_file" on page 168, "util_get_long_from_file" on page 169, "util_put_int_to_file" on page 177, "util_put_long_to_file" on page 178, "util_put_string_to_file" on page 179

util_put_string_to_file

The `util_put_string_to_file` function writes a single line containing a string to a specified file.

Syntax

```
#include <libproxy/cutil.h>
int util_put_string_to_file(char *filename, char *str);
```

Returns

- nonzero if the operation succeeded.
- `0` if the operation failed.

Parameters

`char` **file* is the name of the file to be read.

`char` **str* is the string to write.

See Also

"util_get_int_from_file" on page 168, "util_get_long_from_file" on page 169, "util_put_int_to_file" on page 177, "util_put_long_to_file" on page 178

util_sect_id

The `util_sect_id` function creates a section ID from the section dim and an index.

Syntax

```
#include <libproxy/cutil.h>
void util_sect_id(int dim, int idx, char *buf);
```

Returns

- nonzero if the operation succeeded.
- `0` if the operation failed.

Parameters

`int` *dim* is the section dim.

`int` *idx* is the index.

`char` **buf* is the buffer to receive the section ID.

util_sh_escape

The `util_sh_escape` function parses a specified string and places a backslash (`\\`) in front of any shell-special characters, returning the resultant string. Use this function to ensure that strings from clients won't cause a shell to behave unexpectedly.

The shell-special characters are the space plus the following characters:

```
&;\Q’"|*?~<>^()[]{}$\\#!
```

Syntax

```
char *util_sh_escape(char *s);
```

Returns

A newly allocated string.

Parameters

`char *s` is the string to be parsed.

See Also

"util_uri_escape" on page 185

util_snprintf

The `util_snprintf` function formats a specified string, using a specified format, into a specified buffer using the `printf`-style syntax and performs bounds checking. The function returns the number of characters in the formatted buffer.

For more information, see the documentation on the `printf` function for the runtime library of your compiler.

Syntax

```
int util_snprintf(char *s, int n, char *fmt, ...);
```

Returns

The number of characters formatted into the buffer.

Parameters

`char *s` is the buffer to receive the formatted string.

`int n` is the maximum number of bytes allowed to be copied.

`char *fmt` is the format string. The function handles only `%d` and `%s` strings. It does not handle any width or precision strings.

`...` represents a sequence of parameters for the `printf` function.

See Also

"util_sprintf" on page 182, "util_vsnprintf" on page 189, "util_vsprintf" on page 189

util_sprintf

The `util_sprintf` function formats a specified string, using a specified format, into a specified buffer, using the `printf`-style syntax without bounds checking. The function returns the number of characters in the formatted buffer.

Because `util_sprintf` doesn't perform bounds checking, use this function only if you are certain that the string fits the buffer. Otherwise, use the function `util_snprintf`. For more information, see the documentation on the `printf` function for the runtime library of your compiler.

Syntax

```
int util_sprintf(char *s, char *fmt, ...);
```

Returns

The number of characters formatted into the buffer.

Parameters

`char *s` is the buffer to receive the formatted string.

`char *fmt` is the format string. The function handles only `%d` and `%s` strings. It does not handle any width or precision strings.

`...` represents a sequence of parameters for the `printf` function.

Example

```
char *logmsg;int len;logmsg = (char *) MALLOC(256);
    len = util_sprintf(logmsg, "%s %s %s\\n", ip, method, uri);
```

See Also

"util_snprintf" on page 181, "util_vsnprintf" on page 189, "util_vsprintf" on page 189

util_strcasecmp

The `util_strcasecmp` function performs a comparison of two alphanumeric strings and returns a `-1`, `0`, or `1` to signal which string is larger or that the strings are identical.

The comparison is not case sensitive.

Syntax

```
int util_strcasecmp(const char *s1, const char *s2);
```

Returns

1 if s1 is greater than s2.

0 if s1 is equal to s2.

-1 if s1 is less than s2.

Parameters

char *s1 is the first string.

char *s2 is the second string.

See Also

"util_strncasecmp" on page 184

util_strftime

The util_strftime function translates a tm structure, which is a structure describing a system time, into a textual representation. This function is a thread-safe version of the standard strftime function

Syntax

```
int util_strftime(char *s, const char *format, const struct tm *t);
```

Returns

The number of characters placed into s, not counting the terminating NULL character.

Parameters

char *s is the string buffer to put the text into. The function does not check bounds, so you must make sure that your buffer is large enough for the text of the date.

const char *format is a format string resembling a printf string in that it consists of text with certain %x substrings. You may use the constant HTTP_DATE_FMT to create date strings in the standard Internet format. For more information, see the documentation on the printf function for the runtime library of your compiler. Refer to Chapter 7, "Time Formats," for details on time formats.

const struct tm *t is a pointer to a calendar time (tm) structure, usually created by the function system_localtime or system_gmtime.

See Also

system_localtime, system_gmtime

util_strncasecmp

The `util_strncasecmp` function performs a comparison of the first `n` characters in the alphanumeric strings and returns a `-1`, `0`, or `1` to signal which string is larger or that the strings are identical.

The function's comparison is not case sensitive.

Syntax

```
int util_strncasecmp(const char *s1, const char *s2, int n);
```

Returns

`1` if `s1` is greater than `s2`.

`0` if `s1` is equal to `s2`.

`-1` if `s1` is less than `s2`.

Parameters

`char *s1` is the first string.

`char *s2` is the second string.

`int n` is the number of initial characters to compare.

See Also

"util_strcasecmp" on page 182

util_uri_check

The `util_uri_check` function checks whether a URI has a format conforming to the standard.

At present, the only URI checked for is a URL. The standard format for a URL is

protocol`://`*user*`:`*password*`@`*host*`:`*port*`/`*url-path*

where *user:password*, *:password*. *:port*, or */url-path* can be omitted.

Syntax

```
#include <libproxy/util.h>
int util_uri_check (char *uri);
```

Returns

- `0` if the URI does not have the proper form.
- nonzero if the URI has the proper form.

Parameters

`char` **uri* is the URI to be tested.

util_uri_escape

The `util_uri_escape` function converts any special characters in the URI into the URI format, `%XX`, where `XX` is the hexadecimal equivalent of the ASCII character, and returns the escaped string. The special characters are `%?#:+&*"<>`, space, carriage return, and line feed.

Use `util_uri_escape` before sending a URI back to the client.

Syntax

```
char *util_uri_escape(char *d, char *s);
```

Returns

The string, possibly newly allocated with escaped characters replaced.

Parameters

`char *d` is a string. If `d` is not NULL, the function copies the formatted string into `d` and returns it. If `d` is NULL, the function allocates a properly sized string and copies the formatted special characters into the new string, then returns it.

The `util_uri_escape` function does not check bounds for the parameter `d`. Therefore, if `d` is not NULL, it should be at least three times as large as the string `s`.

`char *s` is the string containing the original unescaped URI.

See Also

"util_uri_is_evil" on page 185, "util_uri_parse" on page 186, "util_uri_unescape" on page 186

util_uri_is_evil

The `util_uri_is_evil` function checks a specified URI for insecure path characters. Insecure path characters include `//`, `/./`, `/../` and`/.`, `/..` (and for Windows `./`) at the end of the URI. Use this function to see whether a URI requested by the client is insecure.

Syntax

```
int util_uri_is_evil(char *t);
```

Returns

`1` if the URI is insecure, or `0` if the URI is OK.

Parameters

`char *t` is the URI to be checked.

See Also

"util_uri_escape" on page 185, "util_uri_parse" on page 186

util_uri_parse

The `util_uri_parse` function converts `//`, `/./`, and `/*/../` into `/` in the specified URI where `*` is any character other than `/`. You can use this function to convert a URI's unacceptable sequences into valid ones. First use the function `util_uri_is_evil` to determine whether the function has an incorrect sequence.

Syntax

```
void util_uri_parse(char *uri);
```

Returns

`void`

Parameters

`char *uri` is the URI to be converted.

See Also

"util_uri_is_evil" on page 185, "util_uri_unescape" on page 186

util_uri_unescape

The `util_uri_unescape` function converts the encoded characters of a URI into their ASCII equivalents. Encoded characters appear as `%XX`, where `XX` is a hexadecimal equivalent of the character.

Note – You cannot use an embedded null in a string, because NSAPI functions assume that a null is the end of the string. Therefore, passing Unicode-encoded content through an NSAPI plug-in doesn't work.

Syntax

```
void util_uri_unescape(char *uri);
```

Returns

`void`

Parameters

`char *uri` is the URI to be converted.

See Also

"util_uri_escape" on page 185, "util_uri_is_evil" on page 185, "util_uri_parse" on page 186

util_url_cmp

The `util_url_cmp` function compares two URLs. This function is analogous to the `strcmp( )()` library function of C.

Syntax

```
#include <libproxy/util.h>
int util_url_cmp (char *s1, char *s2);
```

Returns

- `-1` if the first URL, *s1*, is less than the second, *s2*
- `0` if they are identical
- `1` if the first URL, *s1*, is greater than the second, *s2*

Parameters

`char` **s1* is the first URL to be tested.

`char` **s2* is the second URL to be tested.

See Also

"util_url_fix_host name" on page 188, "util_uri_check" on page 184

util_url_fix_host name

The `util_url_fix_host name` function converts the host name in a URL to lowercase and removes redundant port numbers.

Syntax

```
#include <libproxy/util.h>
void util_url_fix_host name(char *url);
```

Returns

`void` but changes the value of its parameter string

The protocol specifier and the host name in the parameter string are changed to lowercase. The function also removes redundant port numbers, such as 80 for HTTP, 70 for gopher, and 21 for FTP.

Parameters

`char` **url* is the URL to be converted.

See Also

"util_url_cmp" on page 187, "util_uri_check" on page 184

util_url_has_FQDN

The `util_url_has_FQDN` function returns a value to indicate whether a specified URL references a fully qualified domain name.

Syntax

```
#include <libproxy/util.h>
int util_url_has_FQDN(char *url);
```

Returns

- `1` if the URL has a fully qualified domain name.
- `0` if the URL does not have a fully qualified domain name.

Parameters

`char` **url* is the URL to be examined.

util_vsnprintf

The util_vsnprintf function formats a specified string, using a specified format, into a specified buffer using the vprintf-style syntax and performs bounds checking. The function returns the number of characters in the formatted buffer.

For more information, see the documentation on the printf function for the runtime library of your compiler.

Syntax

```
int util_vsnprintf(char *s, int n, register char *fmt, va_list args);
```

Returns

The number of characters formatted into the buffer.

Parameters

char *s is the buffer to receive the formatted string.

int n is the maximum number of bytes allowed to be copied.

register char *fmt is the format string. The function handles only %d and %s strings. It does not handle any width or precision strings.

va_list args is an STD argument variable obtained from a previous call to va_start.

See Also

“util_snprintf” on page 181, “util_vsprintf” on page 189

util_vsprintf

The util_vsprintf function formats a specified string, using a specified format, into a specified buffer using the vprintf-style syntax without bounds checking. The function returns the number of characters in the formatted buffer.

For more information, see the documentation on the printf function for the runtime library of your compiler.

Syntax

```
int util_vsprintf(char *s, register char *fmt, va_list args);
```

Returns

The number of characters formatted into the buffer.

Parameters

char *s is the buffer to receive the formatted string.

register char *fmt is the format string. The function handles only %d and %s strings. It does not handle any width or precision strings.

va_list args is an STD argument variable obtained from a previous call to va_start.

See Also

"util_snprintf" on page 181, "util_vsnprintf" on page 189

W

write

The write filter method is called when output data is to be sent. Filters that modify or consume outgoing data should implement the write filter method.

Upon receiving control, a write implementation should first process the data as necessary, and then pass it on to the next filter layer; for example, by calling net_write(layer->lower, ...,). If the filter buffers outgoing data, it should implement the "flush" on page 95 filter method.

Syntax

```
int write(FilterLayer *layer, const void *buf, int amount);
```

Returns

The number of bytes consumed, which might be less than the requested amount if an error occurred.

Parameters

FilterLayer *layer is the filter layer in which the filter is installed.

const void *buf is the buffer that contains the outgoing data.

int amount is the number of bytes in the buffer.

Example

```
int myfilter_write(FilterLayer *layer, const void *buf, int amount)
{
    return net_write(layer->lower, buf, amount);
}
See Also
```

“flush” on page 95, “net_write” on page 105, “writev” on page 191

writev

The `writev` filter method is called when multiple buffers of output data are to be sent. Filters that modify or consume outgoing data might implement the `writev` filter method.

If a filter implements the `write` filter method but not the `writev` filter method, the server automatically translates `net_writev` calls to “net_write” on page 105 calls. As a result, filters that deal with the outgoing data stream do not need to implement the `writev` filter method. However, for performance reasons, filters that implement the `write` filter method should also implement the `writev` filter method.

Syntax

```
int writev(FilterLayer *layer, const struct iovec *iov, int iov_size);
```

Returns

The number of bytes consumed, which might be less than the requested amount if an error occurred.

Parameters

`FilterLayer *layer` is the filter layer the filter is installed in.

`const struct iovec *iov` is an array of `iovec` structures, each of which contains outgoing data.

`int iov_size` is the number of `iovec` structures in the `iov` array.

Example

```
int myfilter_writev(FilterLayer *layer, const struct iovec *iov,
    int iov_size)
{
    return net_writev(layer->lower, iov, iov_size);
}
```

See Also

"flush" on page 95, "net_write" on page 105, "write" on page 190

CHAPTER 5

Data Structure Reference

NSAPI uses many data structures that are defined in the `nsapi.h` header file, which is in the directory *server-root*/`plugins/include`.

The NSAPI functions described in Chapter 4, "NSAPI Function Reference," provide access to most of the data structures and data fields. Before directly accessing a data structure in `naspi.h`, check whether an accessor function exists for that structure.

For information about the privatization of some data structures in Proxy Server 4, see "Privatization of Some Data Structures" on page 194

The rest of this chapter describes public data structures in `nsapi.h`. Data structures in `nsapi.h` that are not described in this chapter are considered private and might change incompatibly in future releases.

This chapter contains the following sections:

- "Privatization of Some Data Structures" on page 194
- "Session" on page 194
- "pblock" on page 195
- "pb_entry" on page 195
- "pb_param" on page 195
- "Session->client" on page 196
- "Request" on page 196
- "stat" on page 197
- "shmem_s" on page 197
- "cinfo" on page 198
- "sendfiledata" on page 198
- "Filter" on page 198
- "FilterContext" on page 199
- "FilterLayer" on page 199
- "FilterMethods" on page 199
- "CacheEntry Data Structure" on page 200

- "CacheState Data Structure" on page 201
- "ConnectMode Data Structure" on page 202

Privatization of Some Data Structures

The data structures in `nsapi_pvt.h` are now considered to be private data structures. Do not write code that accesses them directly. Instead, use accessor functions. This change should have very little impact on customer-defined plug-ins. Examine `nsapi_pvt.h` to see which data structures have been removed from the public domain. You can also see the accessor functions you can use now to access these dat structures.

Plug-ins written for Enterprise Server 3.*x* that access contents of data structures defined in `nsapi_pvt.h` will not be source compatible with Proxy Server 4 you will have to `#include "nsapi_pvt.h"` to build such plug-ins from source. These programs might not be binary compatible with Proxy Server 4, because some of the data structures in `nsapi_pvt.h` have changed size. In particular, the `directive` structure is larger, which means that a plug-in that indexes through the directives in a `dtable` will not work without being rebuilt with `nsapi_pvt.h` included.

Because the majority of plug-ins do not reference the internals of data structures in `nsapi_pvt.h`, most existing NSAPI plug-ins will be both binary and source compatible with Proxy Server 4.

Plug-ins written for Sun iPlanet Web Proxy Server 3.6 will not be binary compatible with Proxy Server 4. These plug-ins will have to be recompiled and relinked using Proxy Server 4's NSAPI header files and libraries.

Session

A session is the time between the opening and closing of the connection between the client and the server. The `session` data structure holds variables that apply session wide, regardless of the requests being sent, as shown in the following example.

```
typedef struct {
/* Information about the remote client */
    pblock *client;

    /* The socket descriptor to the remote client */
    SYS_NETFD csd;

    /* The input buffer for that socket descriptor */
    netbuf *inbuf;
```

```
    /* Raw socket information about the remote */
    /* client (for internal use) */
    struct in_addr iaddr;
} Session;
```

pblock

The parameter block is the hash table that holds `pb_entry` structures. Its contents are transparent to most code. This data structure is frequently used in NSAPI. It provides the basic mechanism for packaging up parameters and values. Many functions exist for creating and managing parameter blocks, and for extracting, adding, and deleting entries. See the functions whose names start with `pblock_` in Chapter 4, “NSAPI Function Reference.” You should not need to write code that accesses `pblock` data fields directly.

```
typedef struct {
    int hsize;
    struct pb_entry **ht;
} pblock;
```

pb_entry

The `pb_entry` is a single element in the parameter block.

```
struct pb_entry {
    pb_param *param;
    struct pb_entry *next;
};
```

pb_param

The `pb_param` represents a name-value pair, as stored in a `pb_entry`.

```
typedef struct {
    char *name,*value;
} pb_param;
```

Session->client

The `Session->client` parameter block structure contains two entries:

- The `ip` entry is the IP address of the client machine.
- The `dns` entry is the DNS name of the remote machine. This member must be accessed through the `session_dns` function call

```
/** session_dns returns the DNS host name of the client for this* session
    and inserts it into the client pblock. Returns NULL if* unavailable.
    */char *session_dns(Session *sn);
```

Request

Under HTTP protocol, only one request is made per session. The `request` structure contains the variables that apply to the request in that session, for example, the variables include the client's HTTP headers.

```
typedef struct {
    /* Server working variables */
    pblock *vars;

    /* The method, URI, and protocol revision of this request */
    block *reqpb;

    /* Protocol specific headers */
    int loadhdrs;
    pblock *headers;

    /* Server's response headers */
    int senthdrs;
    pblock *srvhdrs;

    /* The object set constructed to fulfill this request */
    httpd_objset *os;
} Request;
```

stat

When a program calls the `stat( )` function for a given file, the system returns a structure that provides information about the file. The specific details of the structure should be obtained from your platform's implementation, but the basic outline of the structure is shown in the following example.

```
struct stat {
    dev_t      st_dev;     /* device of inode */
    inot_t     st_ino;     /* inode number */
    short      st_mode;    /* mode bits */
    short      st_nlink;   /* number of links to file /*
    short      st_uid;     /* owner's user id */
    short      st_gid;     /* owner's group id */
    dev_t      st_rdev;    /* for special files */
    off_t      st_size;    /* file size in characters */
    time_t     st_atime;   /* time last accessed */
    time_t     st_mtime;   /* time last modified */
    time_t     st_ctime;   /* time inode last changed*/
}
```

The elements that are most significant for server plug-in API activities are `st_size`, `st_atime`, `st_mtime`, and `st_ctime`.

shmem_s

```
typedef struct {
    void       *data;   /* the data */
    HANDLE     fdmap;
    int        size;    /* the maximum length of the data */
    char       *name;   /* internal use: filename to unlink if exposed */
    SYS_FILE   fd;      /* internal use: file descriptor for region */
} shmem_s;
```

cinfo

The `cinfo` data structure records the content information for a file.

```
typedef struct {
    char    *type;
            /* Identifies what kind of data is in the file*/
    char    *encoding;
             /* encoding identifies any compression or other /*
            /* content-independent transformation that's been /*
            /* applied to the file, such as uuencode)*/
    char    *language;
            /* Identifies the language a text document is in. */
} cinfo;
```

sendfiledata

The `sendfiledata` data structure is used to pass parameters to the `net_sendfile` function. It is also passed to the `sendfile` method in an installed filter in response to a `net_sendfile` call.

```
typedef struct {
    SYS_FILE fd;         /* file to send */
    size_t offset;       /* offset in file to start sending from */
    size_t len;          /* number of bytes to send from file */
    const void *header;  /* data to send before file */
    int hlen;            /* number of bytes to send before file */
    const void *trailer; /* data to send after file */
    int tlen;            /* number of bytes to send after file */
} sendfiledata;
```

Filter

The `Filter` data structure is an opaque representation of a filter. A `Filter` structure is created by calling "filter_create" on page 91.

```
typedef struct Filter Filter;
```

FilterContext

The `FilterContext` data structure stores context associated with a particular filter layer. Filter layers are created by calling “filter_insert” on page 92.

Filter developers may use the data member to store filter-specific context information.

```
typedef struct {
    pool_handle_t *pool; /* pool context was allocated from */
    Session *sn;         /* session being processed */
    Request *rq;         /* request being processed */
    void *data;          /* filter-defined private data */
} FilterContext;
```

FilterLayer

The `FilterLayer` data structure represents one layer in a filter stack. The `FilterLayer` structure identifies the filter installed at that layer. It provides pointers to layer-specific context and a filter stack that represents the layer immediately below it in the filter stack.

```
typedef struct {
    Filter *filter; /* the filter at this layer in the filter stack */
    FilterContext *context; /* context for the filter */
    SYS_NETFD lower; /* access to the next filter layer in the stack */
} FilterLayer;
```

FilterMethods

The `FilterMethods` data structure is passed to “filter_create” on page 91 to define the filter methods a filter supports. Each new `FilterMethods` instance must be initialized with the `FILTER_METHODS_INITIALIZER` macro. For each filter method a filter supports, the corresponding `FilterMethods` member should point to a function that implements that filter method.

```
typedef struct {
    size_t size;
    FilterInsertFunc *insert;
    FilterRemoveFunc *remove;
    FilterFlushFunc *flush;
```

```
    FilterReadFunc *read;
    FilterWriteFunc *write;
    FilterWritevFunc *writev;
    FilterSendfileFunc *sendfile;
} FilterMethods;
```

CacheEntry Data Structure

The CacheEntry data structure holds all the information about one cache entry. The structure is created by the `ce_lookup` function and destroyed by the `ce_free` function CacheEntry is defined in the `libproxy/cache.h` file.

```
typedef struct _CacheEntry {
    CacheState  state;    /* state of the cache file; DO NOT refer to any
                 * of the other fields in this C struct if state
                 * is other than
                 *        CACHE_REFRESH or
                 *        CACHE_RETURN_FROM_CACHE
                 */
SYS_FILE    fd_in;    /* do not use: open cache file for reading */
int fd_out;    /* do not use: open (locked) cache file for writing */
struct stat finfo;    /* stat info for the cache file */
unsigned char  digest[CACHE_DIGEST_LEN];    /* MD5 for the URL */
char *         url_dig;  /* URL used to for digest; field #8 in CIF */
char *        url_cif;   /* URL read from CIF file */
char *        filname;   /* Relative cache file name */
char *        dirname;   /* Absolute cache directory name */
char *        absname;   /* Absolute cache file path */
char *        lckname;   /* Absolute locked cache file path */
char *        cifname;   /* Absolute CIF path */
int        sect_idx;     /* Cache section index */
int        part_idx;     /* Cache partition index */
CSect *        section;  /* Cache section that this file belongs to */
CPart *        partition;/* Cache partition that this file belongs to */
int         xfer_time;   /* secs *//* Field #2 in CIF */
time_t         last_modified;/* GMT *//* Field #3 in CIF */
time_t         expires;      /* GMT *//* Field #4 in CIF */
time_t         last_checked; /* GMT *//* Field #5 in CIF */
long         content_length; /* Field #6 in CIF */
char *        content_type;  /* Field #7 in CIF */
int        is_auth;          /* Authenticated data -- always do recheck */
int        auth_sent;        /* Client did send the Authorization header */
long    min_size;            /* Min size for a cache file (in KB) */
long    max_size;            /* Max size for a cache file (in KB) */
```

```
time_t         last_accessed;/* GMT for proxy, local for gc */
time_t        created;       /* localtime (only used by gc, st_mtime) */
int        removed;          /* gc only; file was removed from disk */
long        bytes;           /* from stat(), using this we get hdr len */
long        bytes_written;   /* Number of bytes written to disk */
long        bytes_in_media;  /* real fs size taken up */
long        blks;            /* size in 512 byte blocks */
int        category;         /* Value category; bigger is better */
int        cif_entry_ok;     /* CIF entry found and ok */
time_t ims_c;                /* GMT; Client -> proxy if-modified-since */
time_t        start_time;    /* Transfer start time */
int         inhibit_caching; /* Bad expires/other reason not to cache */
int corrupt_cache_file;      /* Cache file gone corrupt => remove */
int     write_aborted;       /* True if the cache file write was aborted */
int batch_update;    /* We’re doing batch update (no real user) */
char *        cache_exclude; /* Hdrs not to write to cache (RE) */
char *        cache_replace; /* Hdrs to replace with fresh ones
                                        from 304 response (RE) */
char *        cache_nomerge; /* Hdrs not to merge with the
                                        cached ones (RE) */
Session *     sn;
Request *     rq;
} CacheEntry;
```

CacheState Data Structure

The CacheState data structure is actually an enumerated list of constants. Aways use the constant names because values are subject to implementation change.

```
typedef enum {
CACHE_EXISTS_NOT = 0,        /* Internal flag -- do not use! */
CACHE_EXISTS,                /* Internal flag -- do not use! */
CACHE_NO,                    /* No caching: don’t read, don’t write cache */
CACHE_CREATE,                /* Create cache; don’t read */
CACHE_REFRESH,               /* Refresh cache; read if not modified */
CACHE_RETURN_FROM_CACHE,     /* Return directly, no check */
CACHE_RETURN_ERROR           /* With connect-mode=never when not in cache */
} CacheState;
```

ConnectMode Data Structure

The ConnectMode data structure is actually an enumerated list of constants. Always use the constant names because values are subject to implementation change.

```
typedef enum {
CM_NORMAL = 0,            /* normal -- retrieve/refresh when necessary */
CM_FAST_DEMO,            /* fast -- retrieve only if not in cache already */
CM_NEVER            /* never -- never connect to network */
} ConnectMode;
```

CHAPTER 6

Using Wildcard Patterns

This chapter describes the format of wildcard patterns used by Proxy Server. These wildcard are used in:

- Directives in the configuration file obj.conf see *Oracle iPlanet Web Proxy Server 4.0.14 Configuration File Reference* for detailed information about obj.conf.
- Various built-in SAFs see *Oracle iPlanet Web Proxy Server 4.0.14 Configuration File Reference* for more information about these predefined SAFs.
- Some NSAPI functions.

Wildcard patterns use special characters. To use one of these characters without the special meaning, precede it with a backslash (\\) character.

This chapter contains the following sections:

- "Wildcard Patterns" on page 203
- "Wildcard Examples" on page 204

Wildcard Patterns

The following table describes wildcard patterns, listing the pattern and its use.

TABLE 6–1 Wildcard Patterns

Pattern	Use
*	Match zero or more characters.
?	Match exactly one occurrence of any character.

TABLE 6–1 Wildcard Patterns *(Continued)*

<table>
<tr><th>Pattern</th><th>Use</th></tr>
<tr><td>|</td><td>An or expression. The substrings used with this operator can contain other special characters such as * or $. The substrings must be enclosed in parentheses, for example, (a|b|c), but the parentheses cannot be nested.</td></tr>
<tr><td>$</td><td>Match the end of the string. This wildcard is useful in or expressions.</td></tr>
<tr><td>[abc]</td><td>Match one occurrence of the characters a, b, or c. Within these expressions, the only character that needs to be treated as a special character is]. All other characters are not special.</td></tr>
<tr><td>[a-z]</td><td>Match one occurrence of a character between a and z.</td></tr>
<tr><td>[^az]</td><td>Match any character except a or z.</td></tr>
<tr><td>*~</td><td>This expression, followed by another expression, removes any pattern matching the second expression.</td></tr>
<tr><td>*</td><td>Match zero or more characters.</td></tr>
</table>

Wildcard Examples

The following table provides wildcard examples, listing the pattern and the result.

TABLE 6–2 Wildcard Examples

<table>
<tr><th>Pattern</th><th>Result</th></tr>
<tr><td>*.netscape.com</td><td>Matches any string ending with the characters .netscape.com.</td></tr>
<tr><td>(quark|energy).netscape.com</td><td>Matches either quark.netscape.com or energy.netscape.com.</td></tr>
<tr><td>198.93.9[23].???</td><td>Matches a numeric string starting with either 198.93.92 or 198.93.93 and ending with any 3 characters.</td></tr>
<tr><td>*.*</td><td>Matches any string with a period in it.</td></tr>
<tr><td>*~netscape-*</td><td>Matches any string except those starting with netscape-.</td></tr>
<tr><td>*.netscape.com~quark.netscape.com</td><td>Matches any host from domain netscape.com except a single host quark.netscape.com.</td></tr>
<tr><td>*.netscape.com~(quark|energy|neutrino).netscape.com</td><td>Matches any host from domain .netscape.com except hosts quark.netscape.com, energy.netscape.com, and neutrino.netscape.com.</td></tr>
</table>

TABLE 6–2 **Wildcard Examples** *(Continued)*

Pattern	Result
`*.com~*.netscape.com`	Matches any host from domain `.com` except hosts from subdomain `netscape.com`.
`type=*~magnus-internal/*`	Matches any type that does not start with `magnus-internal/`. This wildcard pattern is used in the file `obj.conf` in the catch-all `Service` directive.

CHAPTER 7

Time Formats

This chapter describes the format strings used for dates and times. These formats are used by the NSAPI function `util_strftime`, by some built-in SAFs such as `append-trailer`, and by server-parsed HTML (`parse-html`). The formats are similar to those used by the `strftime` C library routine, but not identical.

Time format strings

The following table describes the formats, listing the symbols and their meanings.

TABLE 7–1 **Time Formats**

Symbol	Meaning
%a	Abbreviated weekday name (3 characters)
%d	Day of month as decimal number (01-31)
%S	Second as decimal number (00-59)
%M	Minute as decimal number (00-59)
%H	Hour in 24-hour format (00-23)
%Y	Year with century, as decimal number, up to 2099
%b	Abbreviated month name (3 characters)
%h	Abbreviated month name (3 characters)
%T	Time "HH:MM:SS"
%X	Time "HH:MM:SS"
%A	Full weekday name
%B	Full month name

TABLE 7–1 **Time Formats** *(Continued)*

Symbol	Meaning
`%C`	"%a %b %e %H:%M:%S %Y"
`%c`	Date & time "%m/%d/%y %H:%M:%S"
`%D`	Date "%m/%d/%y"
`%e`	Day of month as decimal number (1-31) without leading zeros
`%I`	Hour in 12-hour format (01-12)
`%j`	Day of year as decimal number (001-366)
`%k`	Hour in 24-hour format (0-23) without leading zeros
`%l`	Hour in 12-hour format (1-12) without leading zeros
`%m`	Month as decimal number (01-12)
`%n`	Line feed
`%p`	a.m./p.m. indicator for 12-hour clock
`%R`	Time "%H:%M"
`%r`	Time "%I:%M:%S %p"
`%t`	tab
`%U`	Week of year as decimal number, with Sunday as first day of week (00-51)
`%w`	Weekday as decimal number (0-6; Sunday is 0)
`%W`	Week of year as decimal number, with Monday as first day of week (00-51)
`%x`	Date "%m/%d/%y"
`%y`	Year without century, as decimal number (00-99)
`%%`	Percent sign

CHAPTER 8

Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) is a protocol that enables a client such as a web browser and a web proxy server to communicate with each other.

HTTP is based on a request-response model. The browser opens a connection to the server and sends a request to the server. The server processes the request and generates a response, which it sends to the browser. The server then closes the connection.

This chapter provides a short introduction to a few HTTP basics. For more information on HTTP, see the IETF home page at: `http://www.ietf.org/home.html`.

This chapter contains the following sections:

- "HTTP Compliance" on page 209
- "HTTP Requests" on page 210
- "Server Responses" on page 211
- "Buffered Streams" on page 213

HTTP Compliance

Proxy Server 4 supports HTTP/1.1. Previous versions of the server supported HTTP/1.0. The server is conditionally compliant with the HTTP/1.1 proposed standard, as approved by the Internet Engineering Steering Group (IESG), and the Internet Engineering Task Force (IETF) HTTP working group.

For more information on the criteria for being conditionally compliant, see the Hypertext Transfer Protocol -- HTTP/1.1 specification (RFC 2068) at:
`http://www.ietf.org/rfc/rfc2068.txt?number=2068`

HTTP Requests

A request from a browser to a server includes the following information:

- "Request Method, URI, and Protocol Version" on page 210
- "Request Headers" on page 210
- "Request Data" on page 210

Request Method, URI, and Protocol Version

A browser can request information using a number of methods. The commonly used methods are:

- `GET` — Requests the specified resource such as a document or image
- `HEAD` — Requests only the header information for the document
- `POST` — Requests that the server accept some data from the browser, such as form input for a CGI program
- `PUT` — Replaces the contents of a server's document with data from the browser

Request Headers

The browser can send headers to the server. Most headers are optional.

The following table lists some of the commonly used request headers.

TABLE 8–1 **Common Request Headers**

Request Header	Description
`Accept`	File types the browser can accept.
`Authorization`	Used if the browser wants to authenticate itself with a server. Information such as the user name and password are included.
`User-Agent`	Name and version of the browser software.
`Referer`	URL of the document where the user clicked the link.
`Host`	Internet host and port number of the resource being requested.

Request Data

If the browser has made a `POST` or `PUT` request, it sends data after the blank line following the request headers. If the browser sends a `GET` or `HEAD` request, there is no data to send.

Server Responses

The server's response includes the following information:

- "HTTP Protocol Version, Status Code, and Reason Phrase" on page 211
- "Response Headers" on page 212
- "Response Data" on page 213

HTTP Protocol Version, Status Code, and Reason Phrase

The server sends back a status code in response to a request, which is a three-digit numeric code. The five categories of status codes are:

- `100-199` — A provisional response
- `200-299` — A successful transaction
- `300-399` — The requested resource should be retrieved from a different location
- `400-499` — An error was caused by the browser
- `500-599` — A serious error occurred in the server

The following table lists some common status codes.

TABLE 8–2 Common HTTP Status Codes

Status Code	Meaning
`200`	Request has succeeded for the method used (`GET`, `POST`, `HEAD`).
`201`	The request has resulted in the creation of a new resource reference by the returned URI.
`206`	The server has sent a response to byte-range requests.
`302`	Found. Redirection to a new URL. The original URL has moved. This code is not an error because most browsers will get the new page.
`304`	Use a local copy. If a browser already has a page in its cache, and the page is requested again, some browsers, such as Netscape Navigator, relay to the web server the "last-modified" timestamp on the browser's cached copy. If the copy on the server is not newer than the browser's copy, the server returns a 304 code instead of returning the page, reducing unnecessary network traffic. This code is not an error.
`400`	Sent if the request is not a valid HTTP/1.0 or HTTP/1.1 request. For example HTTP/1.1 requires a host to be specified either in the `Host` header or as part of the URI on the request line.
`401`	Unauthorized. The user requested a document but didn't provide a valid user name or password.
`403`	Forbidden. Access to this URL is forbidden.

TABLE 8–2 Common HTTP Status Codes *(Continued)*

Status Code	Meaning
404	Not found. The document requested isn't on the server. This code can also be sent if the server has been instructed to send this response to unauthorized user.
408	If the client starts a request but does not complete it within the keep-alive timeout configured in the server, then this response will be sent and the connection closed. The request can be repeated with another open connection.
411	The client submitted a `POST` request with chunked encoding, which is of variable length. However, the resource or application on the server requires a fixed length - a `Content-Length` header to be present. This code tells the client to resubmit its request with content-length.
413	Some applications, for example, certain NSAPI plug-ins, cannot handle very large amounts of data, so these applications will return this code.
414	The URI is longer than the maximum the web server is willing to serve.
416	Data was requested outside the range of a file.
500	Server error. A server-related error occurred. The server administrator should check the server's error log.
503	Sent if the quality of service mechanism was enabled and bandwidth or connection limits were attained. The server will then serve requests with that code. See the "quality of service" section.

Response Headers

The response headers contain information about the server and the response data.

The following table lists some common response headers.

TABLE 8–3 Common Response Headers

Response Header	Description
`Server`	Name and version of the web server
`Date`	Current date (in Greenwich Mean Time)
`Last-Modified`	Date when the document was last modified
`Expires`	Date when the document expires
`Content-Length`	Length of the data that follows (in bytes)
`Content-Type`	MIME type of the following data

TABLE 8–3 Common Response Headers *(Continued)*

Response Header	Description
`WWW-Authenticate`	Used during authentication and includes information that tells the browser software what is necessary for authentication such as user name and password.

Response Data

The server sends a blank line after the last header. The server then sends the response data such as an image or an HTML page.

Buffered Streams

Buffered streams improve the efficiency of network I/O, for example, the exchange of HTTP requests and responses, especially for dynamic content generation. Buffered streams are implemented as transparent NSPR I/O layers, so existing NSAPI modules can use them without any change.

The buffered streams layer adds the following features to Proxy Server:

- Enhanced keep-alive support. When the response is smaller than the buffer size, the buffering layer generates the `Content-Length` header so that the client can detect the end of the response and reuse the connection for subsequent requests.
- Response length determination. If the buffering layer cannot determine the length of the response, it uses HTTP/1.1 chunked encoding instead of the `Content-Length` header to convey the delineation information. If the client only understands HTTP/1.0, the server must close the connection to indicate the end of the response.
- Deferred header writing. Response headers are written out as late as possible to give the servlets a chance to generate their own headers, for example, the session management header `set-cookie`.
- Ability to understand request entity bodies with chunked encoding. Though popular clients do not use chunked encoding for sending `POST` request data, this feature is mandatory for HTTP/1.1 compliance.

The improved connection handling and response length header generation provided by buffered streams also addresses the HTTP/1.1 protocol compliance issues, where absence of the response length headers is regarded as a category 1 failure. In previous Enterprise Server versions, the dynamic content generation programs was expected to send the length headers. If a CGI script did not generate the `Content-Length` header, the server had to close the connection to indicate the end of the response, breaking the keep-alive mechanism. However, keeping track of response length in CGI scripts or servlets is often very inconvenient, and as an application platform provider, the web server is expected to handle such low-level protocol issues.

Output buffering has been built in to the functions that transmit data, such as "net_write" on page 105. You can specify the following `Service` SAF parameters that affect stream buffering, which are described in detail in Chapter 3, "Syntax and Use of the magnus.conf File," in *Oracle iPlanet Web Proxy Server 4.0.14 Configuration File Reference*.

- `UseOutputStreamSize`
- `ChunkedRequestBufferSize`
- `ChunkedRequestTimeout`

The `UseOutputStreamSize`, `ChunkedRequestBufferSize`, and `ChunkedRequestTimeout` parameters also have equivalent `magnus.conf` directives, as described in Chapter 3, "Syntax and Use of the magnus.conf File," in *Oracle iPlanet Web Proxy Server 4.0.14 Configuration File Reference*. The `obj.conf` parameters override the `magnus.conf` directives.

The `UseOutputStreamSize` parameter can be set to zero (`0`) in the `obj.conf` file to disable output stream buffering. For the `magnus.conf` file, setting `UseOutputStreamSize` to zero has no effect.

To override the default behavior when invoking an SAF that uses one of the functions "net_read" on page 103 or "netbuf_grab" on page 108, you can specify the value of the parameter in `obj.conf`, for example:

```
Service fn="my-service-saf" type=perf UseOutputStreamSize=8192
```

APPENDIX A

Alphabetical List of NSAPI Functions and Macros

This appendix provides an alphabetical list for the easy lookup of NSAPI functions and macros.

C “cache_digest” on page 76

“cache_filename” on page 76

“cache_fn_to_dig” on page 77

“CALLOC” on page 77

“ce_free” on page 78

“ce_lookup” on page 78

“cif_write_entry” on page 79

“cinfo_find” on page 80

“condvar_init” on page 80

“condvar_notify” on page 81

“condvar_terminate” on page 81

“condvar_wait” on page 82

“crit_enter” on page 83

“crit_exit” on page 83

“crit_init” on page 84

“crit_terminate” on page 84

D “daemon_atrestart” on page 85

"dns_set_hostent" on page 85

F "fc_close" on page 86

"fc_open" on page 87

"filebuf_buf2sd" on page 87

"filebuf_close" on page 88

"filebuf_getc" on page 89

"filebuf_open" on page 89

"filebuf_open_nostat" on page 90

"filter_create" on page 91

"filter_find" on page 92

"filter_insert" on page 92

"filter_layer" on page 93

"filter_name" on page 94

"filter_remove" on page 94

"flush" on page 95

"FREE" on page 95

"fs_blk_size" on page 96

"fs_blks_avail" on page 97

"func_exec" on page 97

"func_find" on page 98

"func_insert" on page 98

I "insert" on page 99

L "log_error" on page 100

M "magnus_atrestart" on page 101

"MALLOC" on page 101

N "net_flush" on page 102

"net_ip2host" on page 103

"net_read" on page 103

"net_sendfile" on page 104

"net_write" on page 105

"netbuf_buf2sd" on page 106

"netbuf_close" on page 107

"netbuf_getc" on page 107

"netbuf_grab" on page 108

"netbuf_open" on page 108

"nsapi_module_init" on page 109

"NSAPI_RUNTIME_VERSION" on page 109

"NSAPI_VERSION" on page 110

P "param_create" on page 111

"param_free" on page 111

"pblock_copy" on page 112

"pblock_create" on page 112

"pblock_dup" on page 113

"pblock_find" on page 113

"pblock_findlong" on page 114

"pblock_findval" on page 115

"pblock_free" on page 115

"pblock_nlinsert" on page 116

"pblock_nninsert" on page 116

"pblock_nvinsert" on page 117

"pblock_pb2env" on page 118

"pblock_pblock2str" on page 118

"pblock_pinsert" on page 119

"pblock_remove" on page 119

"pblock_replace_name" on page 120

"pblock_str2pblock" on page 121

"PERM_CALLOC" on page 121

"PERM_FREE" on page 122

"PERM_MALLOC" on page 123

"PERM_REALLOC" on page 123

"PERM_STRDUP" on page 124

"prepare_nsapi_thread" on page 125

"protocol_dump822" on page 125

"protocol_finish_request" on page 126

"protocol_handle_session" on page 126

"protocol_parse_request" on page 127

"protocol_scan_headers" on page 127

"protocol_set_finfo" on page 128

"protocol_start_response" on page 129

"protocol_status" on page 130

"protocol_uri2url" on page 131

"protocol_uri2url_dynamic" on page 131

R "read" on page 132

"REALLOC" on page 133

"remove" on page 134

"request_create" on page 134

"request_free" on page 135

"request_header" on page 135

S "sem_grab" on page 136

"sem_init" on page 137

"sem_release" on page 137

"sem_terminate" on page 138

"sem_tgrab" on page 138

"sendfile" on page 139

"session_create" on page 140

"session_dns" on page 140

"session_free" on page 141

"session_maxdns" on page 141

"shexp_casecmp" on page 142

"shexp_cmp" on page 142

"shexp_match" on page 143

"shexp_valid" on page 144

"shmem_alloc" on page 145

"shmem_free" on page 145

"STRDUP" on page 146

"system_errmsg" on page 147

"system_fclose" on page 147

"system_flock" on page 148

"system_fopenRO" on page 148

"system_fopenRW" on page 149

"system_fopenWA" on page 149

"system_fread" on page 150

"system_fwrite" on page 151

"system_fwrite_atomic" on page 151

"system_gmtime" on page 152

"system_localtime" on page 153

"system_lseek" on page 153

"system_rename" on page 154

"system_ulock" on page 154

"system_unix2local" on page 155

"systhread_attach" on page 156

"systhread_current" on page 156

"systhread_getdata" on page 157

"systhread_init" on page 157

"systhread_newkey" on page 158

"systhread_setdata" on page 158

"systhread_sleep" on page 159

"systhread_start" on page 159

"systhread_terminate" on page 160

"systhread_timerset" on page 160

U "USE_NSAPI_VERSION" on page 161

"util_can_exec" on page 162

"util_chdir2path" on page 163

"util_cookie_find" on page 163

"util_does_process_exist" on page 164

"util_env_create" on page 164

"util_env_find" on page 165

"util_env_free" on page 166

"util_env_replace" on page 166

"util_env_str" on page 167

"util_get_current_gmt" on page 167

"util_get_int_from_aux_file" on page 168

"util_get_int_from_file" on page 168

"util_get_long_from_aux_file" on page 169

"util_get_long_from_file" on page 169

"util_get_string_from_aux_file" on page 170

"util_get_string_from_file" on page 171

"util_getline" on page 171

"util_hostname" on page 172

"util_is_mozilla" on page 172

"util_is_url" on page 173

"util_itoa" on page 173

"util_later_than" on page 174

"util_make_filename" on page 174

"util_make_gmt" on page 175

"util_make_local" on page 175

"util_move_dir" on page 176

"util_move_file" on page 176

"util_parse_http_time" on page 177

"util_put_int_to_file" on page 177

"util_put_long_to_file" on page 178

"util_put_string_to_aux_file" on page 179

"util_put_string_to_file" on page 179

"util_sect_id" on page 180

"util_sh_escape" on page 180

"util_snprintf" on page 181

"util_sprintf" on page 182

"util_strcasecmp" on page 182

"util_strftime" on page 183

"util_strncasecmp" on page 184

"util_uri_check" on page 184

"util_uri_escape" on page 185

"util_uri_is_evil" on page 185

"util_uri_parse" on page 186

"util_uri_unescape" on page 186

"util_url_cmp" on page 187

"util_url_fix_host name" on page 188

"util_url_has_FQDN" on page 188

"util_vsnprintf" on page 189

"util_vsprintf" on page 189

W "write" on page 190

"writev" on page 191

Index

A

AddLog
- example of custom SAF, 72-74
- requirements for SAFs, 33-37

API funct, 142

API functions
- cache_digest, 76
- cache_filename, 76
- cache_fn_to_dig, 77
- CALLOC, 77
- ce_free, 78
- ce_lookup, 78-79
- cif_write_entry, 79
- cinfo_find, 80
- condvar_init, 80-81
- condvar_notify, 81
- condvar_terminate, 81-82
- condvar_wait, 82
- crit_enter, 83
- crit_exit, 83
- crit_init, 84
- crit_terminate, 84-85
- daemon_atrestart, 85
- fc_close, 86
- filebuf_buf2sd, 87
- filebuf_close, 88
- filebuf_getc, 89
- filebuf_open, 89-90
- filebuf_open_nostat, 90-91
- filter_create, 91-92
- filter_find, 92
- filter_insert, 92-93

API functions *(Continued)*
- filter_layer, 93-94
- filter_name, 94
- filter_remove, 94
- flush, 95
- FREE, 95-96
- fs_blk_size, 96
- fs_blks_available, 97
- func_exec, 97-98
- func_find, 98
- func_insert, 98-99
- insert, 99
- log_error, 100
- magnus_atrestart, 101
- MALLOC, 101-102
- net_ip2host, 103
- net_read, 103-104
- net_write, 105-106
- netbuf_buf2sd, 106
- netbuf_close, 107
- netbuf_getc, 107
- netbuf_grab, 108
- netbuf_open, 108-109
- param_create, 111
- param_free, 111-112
- pblock_copy, 112
- pblock_create, 112-113
- pblock_dup, 113
- pblock_find, 113-114
- pblock_findlong, 114
- pblock_findval, 115
- pblock_free, 115-116

API functions *(Continued)*
- pblock_nlinsert, 116
- pblock_nninsert, 116-117
- pblock_nvinsert, 117-118
- pblock_pb2env, 118
- pblock_pblock2str, 118-119
- pblock_pinsert, 119
- pblock_remove, 119-120
- pblock_replace_name, 120
- pblock_str2pblock, 121
- PERM_FREE, 122
- PERM_MALLOC, 121-122, 123
- PERM_STRDUP, 124-125
- prepare_nsapi_thread, 125
- protocol_dump822, 125-126
- protocol_set_finfo, 128-129
- protocol_start_response, 129-130
- protocol_status, 130-131
- protocol_uri2url, 131
- read, 132-133
- REALLOC, 133-134
- remove, 134
- request_create, 134-135
- request_free, 135
- request_header, 135-136
- sem_grab, 136-137
- sem_init, 137
- sem_release, 137-138
- sem_terminate, 138
- sem_tgrab, 138-139
- sendfile, 139
- session_create, 140
- session_dns, 140-141
- session_free, 141
- session_maxdns, 141-142
- shem_alloc, 145
- shexp_cmp, 142-143
- shexp_match, 143-144
- shexp_valid, 144
- shmem_free, 145-146
- STRDUP, 146
- system_errmsg, 147
- system_fclose, 147-148
- system_flock, 148

API functions *(Continued)*
- system_fopenRO, 148-149
- system_fopenRW, 149
- system_fopenWA, 149-150
- system_fread, 150-151
- system_fwrite, 151
- system_fwrite_atomic, 151-152
- system_gmtime, 152-153
- system_localtime, 153
- system_lseek, 153-154
- system_rename, 154
- system_ulock, 153-154, 154
- system_unix2local, 155
- systhread_attach, 156
- systhread_current, 156
- systhread_getdata, 157
- systhread_newkey, 147, 158
- systhread_setdata, 158-159
- systhread_sleep, 159
- systhread_start, 159-160
- systhread_terminate, 160
- systhread_timerset, 147, 160-161
- util_can_exec, 162-163
- util_chdir2path, 163
- util-cookie_find, 163-164
- util_cookie_find, 163-164
- util-does_process_exist, 164
- util_env_create, 164-165
- util_env_find, 165
- util_env_free, 166
- util_env_replace, 166
- util_env_str, 167
- util_get_current_gmt, 167
- util_get_int_from_file, 171
- util_get_long_from_file, 169-170
- util_get_string_from_file, 171
- util_getline, 171-172
- util_hostname, 172
- util_is_mozilla, 172-173
- util_is_url, 173
- util_itoa, 173-174
- util_later_than, 174
- util_make_filename, 174-175
- util_make_gmt, 175

API functions *(Continued)*
util_make_local, 175-176
util_move_dir, 176
util_move_file, 176-177
util_parse_http_time, 177
util_put_int_to_file, 177-178
util_put_long_to_file, 178
util_put_string_to_file, 179-180
util_sect_id, 180
util_sh_escape, 180-181
util_snprintf, 181
util-sprintf, 182
util_strcasecmp, 182-183
util_strftime, 183
util_strncasecmp, 184
util_uri_escape, 185
util_uri_is_evil, 185-186
util_uri_parse, 186
util_uri_unescape, 186-187
util_url_fix_hosthame, 188
util_vsnprintf, 189
util_vsprintf, 189-190
write, 190-191
writev, 191-192
AUTH_TYPE environment variable, 37
AUTH_USER environment variable, 37
AuthTrans
example of custom SAF, 52-54
requirements for SAFs, 33-37

B

buffered streams, 213-214

C

cache_digest, API function, 76
cache_filename, API function, 76
cache_fn_to_dig, API function, 77
CALLOC API function, 77
ce_free, API function, 78
ce_lookup, API function, 78-79
CGI
environment variables in NSAPI, 37-38
to NSAPI conversion, 37-38
chunked encoding, 213-214, 214
cif_write_entry, API function, 79
cinfo_find API function, 80
cinfo NSAPI data structure, 198
client
field in session parameter, 21
getting DNS name for, 196
getting IP address for, 196
sessions and, 194
CLIENT_CERT environment variable, 38
compatibility issues, 20, 194
compiling custom SAFs, 25-27
condvar_init API function, 80-81
condvar_notify API function, 81
condvar_terminate API function, 81-82
condvar_wait API function, 82
CONTENT_LENGTH environment variable, 37
CONTENT_TYPE environment variable, 37
context->data, 40
context->rq, 40
context->sn, 40
creating
custom filters, 46-48
custom NSAPI plugins, 15
crit_enter API function, 83
crit_exit API function, 83
crit_init API function, 84
crit_terminate API function, 84-85
csd field in session parameter, 21
custom, NSAPI plugins, 15

D

daemon_atrestart API function, 85
data structures
cinfo, 198
compatibility issues, 194
Filter, 198
FilterContext, 199
FilterLayer, 199
FilterMethods, 199-200

data structures *(Continued)*
- nsapi.h header file, 193
- nsapi_pvt.h, 194
- pb_entry, 195
- pb_param, 195
- pblock, 195
- privatization of, 194
- removed from nsapi.h, 194
- request, 196
- sendfiledata, 198
- session, 194-195
- Session->client, 196
- shmem_s, 197
- stat, 197

day of month, 207
DNS names, getting clients, 196

E

environment variables, CGI to NSAPI conversion, 37-38
Error directive
- requirements for SAFs, 33-37

errors, finding most recent system error, 147
examples
- location in the build, 52
- of custom SAFs in the build, 52
- wildcard patterns, 204-205

F

fc_close API function, 86
features, Proxy Server, 14
file descriptor
- closing, 147-148
- locking, 148
- opening read-only, 148-149
- opening read-write, 149
- opening write-append, 149-150
- reading into a buffer, 150-151
- unlocking, 153-154, 154
- writing from a buffer, 151
- writing without interruption, 151-152

file I/O routines, 31
filebuf_buf2sd API function, 87
filebuf_close API function, 88
filebuf_getc API function, 89
filebuf_open API function, 89-90
filebuf_open_nostat API function, 90-91
filter_create API function, 91-92
filter_find API function, 92
filter_insert API function, 92-93
filter_layer API function, 93-94
filter methods, 40-43
- C prototypes for, 40-41
- FilterLayer data structure, 40
- flush, 42
- insert, 41
- remove, 41
- sendfile, 43
- write, 42
- writev, 43

filter_name API function, 94
Filter NSAPI data structure, 198
filter_remove API function, 94
FilterContext NSAPI data structure, 199
FilterLayer NSAPI data structure, 40, 199
- context->data, 40
- context->rq, 40
- context->sn, 40
- lower, 41

FilterMethods NSAPI data structure, 199-200
filters
- altering Content-length, 45
- functions used to implement, 49
- input, 45
- interface, 40
- methods, 40-43
- NSAPI function overview, 49
- output, 45
- stack position, 43-44
- using, 46-48

flush API function, 42, 95
FREE API function, 95-96
fs_blk_size, API function, 96
fs_blks_available, API function, 97
func_exec API function, 97-98

func_find API function, 98
func_insert API function, 98-99
funcs parameter, 27

G

GATEWAY_INTERFACE environment variable, 37
GMT time, getting thread-safe value, 152-153

H

headers
  field in request parameter, 22
  request, 210
  response, 212-213
HOST environment variable, 38
HTTP
  buffered streams, 213-214
  compliance with HTTP/1.1, 209
  HTTP/1.1 specification, 209
  overview, 209
  requests, 210
  responses, 211-213
  status codes, 211
HTTP_* environment variable, 37
HTTPS environment variable, 38
HTTPS_KEYSIZE environment variable, 38
HTTPS_SECRETKEYSIZE environment variable, 38

I

IETF home page, 209
include directory, for SAFs, 25
Init SAFs in magnus.conf
  requirements for SAFs, 33-37
initializing
  plugins, 27-28
  SAFs, 27-28
Input
  requirements for SAFs, 33-37
input filters, 45
insert API function, 41, 99
IP address, getting client', 196

K

known issues, more information about, 14

L

layer parameter, 40
linking SAFs, 25-27
load-modules function, example, 27
loading
  custom SAFs, 27-28
  plugins, 27-28
  SAFs, 27-28
localtime, getting thread-safe value, 153
log_error API function, 100

M

magnus_atrestart, API function, 101
MALLOC API function, 101-102
matching, special characters, 203-204
memory management routines, 31
month name, 207

N

NameTrans
  example of custom SAF, 55-58
  requirements for SAFs, 33-37
net_ip2host API function, 103
net_read API function, 103-104
net_write API function, 105-106
netbuf_buf2sd API function, 106
netbuf_close API function, 107
netbuf_getc API function, 107
netbuf_grab API function, 108
netbuf_open API function, 108-109
network I/O routines, 32
new features, Proxy Server, 14

NSAPI
- CGI environment variables, 37-38
- filter interface, 40
- function overview, 30-33

NSAPI filters
- interface, 40
- methods, 40-43

nsapi.h, 193
NSAPI plugins, custom, 15
nsapi_pvt.h, 194

O

obj.conf, adding directives for new SAFs, 28-29
ObjectType
- example of custom SAF, 61-63
- requirements for SAFs, 33-37

order, of filters in filter stack, 43-44
Output
- example of custom SAF, 63-69
- requirements for SAFs, 33-37

output filters, 45

P

param_create API function, 111
param_free API function, 111-112
parameter block
- manipulation routines, 30
- SAF parameter, 20-21

parameters, for SAFs, 20-22
PATH_INFO environment variable, 37
path name, converting UNIX-style to local, 155
PATH_TRANSLATED environment variable, 37
PathCheck
- example of custom SAF, 58-61
- requirements for SAFs, 33-37

pb_entry NSAPI data structure, 195
pb_param NSAPI data structure, 195
pb SAF parameter, 20-21
pblock, NSAPI data structure, 195
pblock_copy API function, 112
pblock_create API function, 112-113
pblock_dup API function, 113
pblock_find API function, 113-114
pblock_findlong, API function, 114
pblock_findval API function, 115
pblock_free API function, 115-116
pblock_nlinsert, API function, 116
pblock_nninsert API function, 116-117
pblock_nvinsert API function, 117-118
pblock_pb2env API function, 118
pblock_pblock2str API function, 118-119
pblock_pinsert API function, 119
pblock_remove API function, 119-120
pblock_replace_name, API function, 120
pblock_str2pblock API function, 121
PERM_FREE API function, 122
PERM_MALLOC API function, 121-122, 123
PERM_STRDUP API function, 124-125
platforms, supported, 14
plugins
- compatibility issues, 20, 194
- creating, 19
- instructing the server to use, 28-29
- loading and initializing, 27-28
- private data structures, 194

prepare_nsapi_thread API function, 125
private data structures, 194
protocol_dump822 API function, 125-126
protocol_set_finfo API function, 128-129
protocol_start_response API function, 129-130
protocol_status API function, 130-131
protocol_uri2url API function, 131
protocol utility routines, 30-31
Proxy Server, features, 14

Q

QUERY environment variable, 38
QUERY_STRING environment variable, 37

R

read API function, 42, 132-133
REALLOC API function, 133-134

Release Notes, 14
REMOTE_ADDR environment variable, 37
REMOTE_HOST environment variable, 37
REMOTE_IDENT environment variable, 37
REMOTE_USER environment variable, 37
remove API function, 41, 134
replace.c, 63
REQ_ABORTED response code, 23
REQ_EXIT response code, 23
REQ_NOACTION response code, 23
REQ_PROCEED response code, 22
reqpb, field in request parameter, 22
request
    NSAPI data structure, 196
    SAF parameter, 21-22
request_create, API function, 134-135
request_free, API function, 135
request-handling process, 33-37
request_header API function, 135-136
request headers, 210
REQUEST_METHOD environment variable, 37
request-response model, 209
requests, HTTP, 210
requirements for SAFs, 33-37
    AddLog, 36
    AuthTrans, 34
    Error directive, 36
    Init, 34
    Input, 35
    NameTrans, 34
    ObjectType, 35
    Output, 35
    PathCheck, 35
    Service, 35-36
response headers, 212-213
responses, HTTP, 211-213
result codes, 22-23
rq->headers, 22
rq->reqpb, 22
rq->srvhdrs, 22
rq->vars, 21
rq SAF parameter, 21-22

S

s, 196
SAFs
    compiling and linking, 25-27
    include directory, 25
    interface, 20
    loading and initializing, 27-28
    parameters, 20-22
    result codes, 22-23
    return values, 22
    signature, 20
    testing, 29
SCRIPT_NAME environment variable, 37
sem_grab, API function, 136-137
sem_init, API function, 137
sem_release, API function, 137-138
sem_terminate, API function, 138
sem_tgrab, API function, 138-139
semaphore
    creating, 137
    deallocating, 138
    gaining exclusive access, 136-137
    releasing, 137-138
    testing for exclusive access, 138-139
sendfile API function, 43, 139
sendfiledata NSAPI data structure, 198
server, instructions for using plugins, 28-29
SERVER_NAME environment variable, 38
SERVER_PORT environment variable, 38
SERVER_PROTOCOL environment variable, 38
SERVER_SOFTWARE environment variable, 38
SERVER_URL environment variable, 38
Service
    directives for new SAFs (plugins), 29
    example of custom SAF, 69-72
    requirements for SAFs, 33-37
session
    defined, 194
    NSAPI data structure, 194-195
    resolving the IP address of, 140-141, 141-142
Session->client NSAPI data structure, 196
session_create, API function, 140
session_dns API function, 140-141
session_free, API function, 141

session_maxdns API function, 141-142
session SAF parameter, 21
session structure
creating, 140
freeing, 141
shared memory
allocating, 145
freeing, 145-146
shell expression
comparing (case-sensitive) to a string, 142-143, 143-144
validating, 144
shexp_casecmp API function, 142
shexp_cmp API function, 142-143
shexp_match API function, 143-144
shexp_valid API function, 144
shlib parameter, 27
shmem_alloc, API function, 145
shmem_free, API function, 145-146
shmem_s NSAPI data structure, 197
sn->client, 21
sn->csd, 21
sn SAF parameter, 21
socket
closing, 107
reading from, 103
sending a buffer to, 106
sending file buffer to, 87
writing to, 105
sprintf, see util_sprintf, 182
srvhdrs, field in request parameter, 22
stat NSAPI data structure, 197
status codes, 211
STRDUP API function, 146
streams, buffered, 213-214
string, creating a copy of, 146
supported platforms, 14
system_errmsg API function, 147
system_fclose API function, 147-148
system_flock API function, 148
system_fopenRO API function, 148-149
system_fopenRW API function, 149
system_fopenWA API function, 149-150
system_fread API function, 150-151
system_fwrite API function, 151
system_fwrite_atomic API function, 151-152
system_gmtime API function, 152-153
system_localtime API function, 153
system_lseek API function, 153-154
system_rename API function, 154
system requirements, 14
system_ulock API function, 153-154, 154
system_unix2local API function, 155
systhread_attach API function, 156
systhread_current API function, 156
systhread_getdata API function, 157
systhread_newkey, API function, 147
systhread_newkey API function, 158
systhread_setdata API function, 158-159
systhread_sleep API function, 159
systhread_start API function, 159-160
systhread_terminate, API function, 160
systhread_timerset, API function, 147
systhread_timerset API function, 160-161

T

testing custom SAFs, 29
thread
allocating a key for, 147, 158
creating, 159-160
getting a pointer to, 156
getting data belonging to, 157
putting to sleep, 159
setting data belonging to, 158-159
setting interrupt timer, 147, 160-161
terminating, 160
thread routines, 32

U

unicode, 33, 187
util_can_exec API function, 162-163
util_chdir2path API function, 163
util_cookie_find API function, 163-164
util_does_process_exist, API function, 164
util_env_create, API function, 164-165

util_env_find API function, 165
util_env_free API function, 166
util_env_replace API function, 166
util_env_str API function, 167
util_get_current_gmt, API function, 167
util_get_int_from_file, API function, 171
util_get_long_from_file, API function, 169-170
util_get_string_from_file, API function, 171
util_getline API function, 171-172
util_hostname API function, 172
util_is_mozilla API function, 172-173
util_is_url API function, 173
util_itoa API function, 173-174
util_later_than API function, 174
util_make_filename, API function, 174-175
util_make_gmt, API function, 175
util_make_local, API function, 175-176
util_move_dir, API function, 176
util_move_file, API function, 176-177
util_parse_http_time, API function, 177
util_put_int_to_file, API function, 177-178
util_put_long_to_file, API function, 178
util_put_string_to_file, API function, 179-180
util_sect_id, API function, 180
util_sh_escape API function, 180-181
util_snprintf API function, 181
util_sprintf API function, 182
util_strcasecmp API function, 182-183
util_strftime API function, 183, 207
util_strncasecmp API function, 184
util_uri_escape API function, 185
util_uri_is_evil API function, 185-186
util_uri_parse API function, 186
util_uri_unescape API function, 186-187
util_url_fix_hostname
 API function, 188
util_vsnprintf API function, 189
util_vsprintf API function, 189-190
utility routines, 32-33

V

vars, field in request parameter, 21
vsnprintf, see util_vsnprintf, 189
vsprintf, see util_vsprintf, 189-190

W

weekday, 207
workarounds, more information about, 14
write API function, 42, 190-191
writev API function, 43, 191-192