Sun Java System Web Server 7.0 Update 3 NSAPI Developer's Guide

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 820–4843
June 2008

080626@20490

Contents

Examples

Preface

This guide discusses how to use Netscape Server Application Programmer's Interface (NSAPI) to build plug-ins that define Server Application Functions (SAFs) to extend and modify Sun™ Java™ System Web Server 7.0. The guide also provides a reference of the NSAPI functions you can use to define new plug-ins.

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

Before You Read This Book

Web Server 7.0 can be installed as a stand-alone product or as a component of Sun Java Enterprise System (Java ES), a software infrastructure that supports enterprise applications distributed across a network or Internet environment. If you are installing Web Server 7.0 as a component of Java ES, you should be familiar with the system documentation at `http://docs.sun.com/coll/1286.3`.

Web Server 7.0 Documentation Set

The Web Server 7.0 documentation set describes how to install and administer the Web Server. You can access Web Server 7.0 Update 3 documentation at `http://docs.sun.com/coll/1653.2`. For an introduction to Web Server 7.0, refer to the books in the order in which they are listed in the following table.

TABLE P–1 Books in the Web Server 7.0 Documentation Set

Documentation Title	Contents
Sun Java System Web Server 7.0 Update 3 Documentation Center	Web Server documentation topics organized by tasks and subject
Sun Java System Web Server 7.0 Update 3 Release Notes	■ Late-breaking information about the software and documentation ■ Supported platforms and patch requirements for installing Web Server
Sun Java System Web Server 7.0 Update 3 Installation and Migration Guide	Performing installation and migration tasks: ■ Installing Web Server and its various components, ■ Migrating data from Sun ONE Web Server 6.0 or 6.1 to Sun Java System Web Server 7.0
Sun Java System Web Server 7.0 Update 3 Administrator's Guide	Performing the following administration tasks: ■ Using the Administration GUI and command-line interface ■ Configuring server preferences ■ Using server instances ■ Monitoring and logging server activity ■ Using certificates and public key cryptography to secure the server ■ Configuring access control to secure the server ■ Using Java Platform Enterprise Edition (Java EE) security features ■ Deploying applications ■ Managing virtual servers ■ Defining server workload and sizing the system to meet performance needs ■ Searching the contents and attributes of server documents, and creating a text search interface ■ Configuring the server for content compression ■ Configuring the server for web publishing and content authoring using WebDAV
Sun Java System Web Server 7.0 Update 3 Developer's Guide	Using programming technologies and APIs to do the following: ■ Extend and modify Sun Java System Web Server ■ Dynamically generate content in response to client requests and modify the content of the server
Sun Java System Web Server 7.0 Update 3 NSAPI Developer's Guide	Creating custom Netscape Server Application Programmer's Interface (NSAPI) plug-ins

TABLE P–1 Books in the Web Server 7.0 Documentation Set *(Continued)*

Documentation Title	Contents
Sun Java System Web Server 7.0 Update 3 Developer's Guide to Java Web Applications	Implementing Java Servlets and JavaServer Pages^{TM} (JSP^{TM}) technology in Sun Java System Web Server
Sun Java System Web Server 7.0 Update 3 Administrator's Configuration File Reference	Editing configuration files
Sun Java System Web Server 7.0 Update 3 Performance Tuning, Sizing, and Scaling Guide	Tuning Sun Java System Web Server to optimize performance
Sun Java System Web Server 7.0 Update 3 Troubleshooting Guide	Troubleshooting Web Server

Related Books

The URL for all documentation about Sun Java Enterprise System (Java ES) and its components is at `http://docs.sun.com/coll/1286.3`.

Default Paths and File Names

The following table describes the default paths and file names that are used in this book.

TABLE P–2 Default Paths and File Names

Placeholder	Description	Default Value
install-dir	Represents the base installation directory for Web Server 7.0	Sun Java Enterprise System (Java ES) installations on the Solaris™ platform: `/opt/SUNWwbsvr7` Java ES installations on the Linux and HP-UX platform: `/opt/sun/webserver/` Java ES installations on the Windows platform: *system-drive*`:\Program Files\Sun\JavaES5\WebServer7` Other Solaris, Linux, and HP-UX installations, non-root user: *home-directory*`/sun/webserver7` Other Solaris, Linux, and HP-UX installations, root user: `/sun/webserver7` Windows, all installations: *system-drive*`:\Program Files\Sun\WebServer7`
instance-dir	Directory that contains the instance-specific subdirectories.	For Java ES installations, the default location for instances on Solaris: `/var/opt/SUNWwbsvr7` For Java ES installations, the default location for instances on Linux and HP-UX: `/var/opt/sun/webserver7` For Java ES installations, the default location for instance on Windows: *system-drive*`:\Program Files\Sun\JavaES5\WebServer7` For stand-alone installations, the default location for instance on Solaris, Linux, and HP-UX:*install-dir* For stand-alone installations, the default location for instance on Windows: *system-drive*`:\Program Files\sun\WebServer7`

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.

TABLE P–4 Symbol Conventions *(Continued)*

Symbol	Description	Example	Meaning
→	Indicates menu item selection in a graphical user interface.	File → New → Templates	From the File menu, choose New. From the New submenu, choose Templates.

Documentation, Support, and Training

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

- Documentation (`http://www.sun.com/documentation/`)
- Support (`http://www.sun.com/support/`)
- Training (`http://www.sun.com/training/`)

Searching Sun Product Documentation

Besides searching Sun product documentation from the docs.sun.com^{SM} 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 "broker," type the following:

```
broker site:docs.sun.com
```

To include other Sun 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 – Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun 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. Sun 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.

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. To share your comments, go to `http://docs.sun.com` and click Send Comments. In the online form, provide the full document title and part number. The part number is a 7-digit or 9-digit number that can be found on the book's title page or in the document's URL. For example, the part number of this book is 820-2205.

◆ ◆ ◆ CHAPTER 1

About NSAPI

This chapter provides an overview of the Netscape Server API (NSAPI).

Overview of NSAPI

The NSAPI is an extension that enables you to extend or customize the core functionality of Sun Java System Web Server to provide a scalable, efficient mechanism for building interfaces between the HTTP server and back-end applications.

The NSAPI provides solutions to solve performance and efficiency problems common to installations that make liberal use of Common Gateway Interface (CGI) functionality. CGI is common across most HTTP server implementations for running external programs, or gateways, between the information server and external applications.

HTTP Request-Response Process

This sections provides the logical breakdown of the HTTP request-response process.

After the client sends its request to a server, it is helpful to define a set of logical steps which the server must perform before a response is sent.

The following steps are performed in the normal response process:

- Authorization translation
- Name translation
- Path checks
- Object type
- Respond to request
- Log the transaction

If at any time one of these steps fail, another step must be performed to handle the error and inform the client about what happened.

About Server Application Function

Server Application Functions (SAFs) are used to perform the steps to generate a HTTP request-response. These functions take the request, and server configuration database as input, and return a response to the client as output. The set of functions which are applied are determined by the inputs.

Server application functions have a particular *class*, where the class corresponds to the request-response step it helps implement. There is an additional class of application function, the *initialization* function, which is executed upon server startup and performs static data initialization for the various server modules.

Server application functions have a single class and are not informed by the server which class they are being used for. The server keeps an internal table of available functions, and maps these function pointers to unique character strings which identify them. By using this string in the configuration database, a *function* can be called to carry out one of the above steps.

You can create custom Server Application Functions (SAFs). Creation of SAFs allow you to modify or extend the Sun Java System Web Server's built-in functionality. 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.

For more information about how to create a custom server application function, see Chapter 2, "Creating Custom Server Application Functions."

About Parameter Block

The *parameter block* is the fundamental data structure within the server code. The parameter block, or `pblock`, is a hash table keyed on the name string, which maps these name strings onto their value character strings.

For more information about the parameter block, see "`pblock` Data Structure" on page 169.

Passing Parameters to SAF

All server application functions have the following syntax:

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

Where, `pb` is the parameter block with the parameters given by the site administrator for this function invocation. This parameter should be considered read-only, and any data modification should be performed on copies of the data. Doing otherwise is unsafe in threaded server architectures, and yields unpredictable results in multi-process server architectures.

CHAPTER 2

Creating Custom Server Application Functions

This chapter describes how to write your own NSAPI plug-ins that define custom Server Application Functions (SAFs). The ability to create custom plug-ins enables you to modify or extend the Sun Java System Web Server's built-in functionality. 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 has the following sections:

- "Future Compatibility Issues" on page 22
- "SAF Interface" on page 22
- "SAF Parameters" on page 22
- "Result Codes" on page 23
- "Creating and Using Custom SAFs" on page 24
- "Overview of NSAPI C Functions" on page 30
- "Required Behavior of SAFs for Each Directive" on page 35
- "CGI to NSAPI Conversion" on page 39

Before writing custom SAFs, you must familiarize yourself with the request-handling process, as described in detail in the *Sun Java System Web Server 7.0 Update 3 Administrator's Configuration File Reference*. Also, before writing a custom SAF, check to see whether a built-in SAF already accomplishes the tasks you have in mind.

See Appendix B, "Alphabetical List of NSAPI Functions and Macros," for a list of the predefined `Init` SAFs. For information about predefined SAFs used in the `obj.conf` file, see the *Sun Java System Web Server 7.0 Update 3 Administrator's Configuration File Reference*.

For a complete list of the NSAPI routines for implementing custom SAFs, see Chapter 6, "NSAPI Function and Macro Reference."

Future Compatibility Issues

To keep your custom plug-ins upgradable, do the following:

- 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 whether custom or built-in have the same C interface regardless of the request-handling step for which they are written. SAFs are small functions intended for a specific purpose within a specific request-response step. SAFs 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 is:

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

The next 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. For more information on the result codes, see "Result Codes" on page 23.

SAF Parameters

This section discusses the SAF parameters in detail.

`pb` (Parameter Block) Parameter

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.

For example, a directive that invokes the `basic-nsca` function might look like the following:

```
AuthTrans fn=basic-ncsa auth-type=basic dbm=users.db
```

In this case, the `pb` parameter passed to `basic-ncsa` contains name-value pairs that correspond to `auth-type=basic` and `dbm=users.db`.

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`. For information on working with parameter blocks, see "Parameter Block Manipulation Routines" on page 31.

`sn` (Session) Parameter

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. For a list of important fields, see "`Session` Data Structure" on page 168.

`rq` (Request) Parameter

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. For a list of important fields, see "`Request` Data Structure" on page 170.

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 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 (`Input`, `Output`, `Route`, `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, the server executes the SAF specified. If not, the server sends a default HTTP response with the status code and reason phrase, in addition to 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 the connection to the client was lost. This 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.

The general steps to create a custom SAF are as follows:

- Write the Source Code using the NSAPI functions. Each SAF is written for a specific directive.
- Compile and Link the source code to create a shared library (`.so` , `.sl` , or `.dll`) file.
- Load and Initialize the SAF by editing the `magnus.conf` file to do the following actions:
  - Load the shared library file containing your custom SAFs
  - Initialize the SAF if necessary

  Instruct the Server to Call the SAFs by editing `obj.conf` to call your custom SAFs at the appropriate time.
- Restart the Server.
- Test the SAF 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 for a Custom SAF

Write your 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 and for available routines, see Chapter 6, "NSAPI Function and Macro Reference."

For examples of custom SAFs, see Chapter 4, "Examples of Custom SAFs and Filters."

The signature for all SAFs is as follows:

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

For more details on the parameters, see "SAF Parameters" on page 22.

You must register your SAFs with the server. SAFs may be registered using the `funcs` parameter of the `load-modules` `Init` SAF or by a call to `func_insert`. A plug-in may define a `nspai_module_init` function that is used to call `func_insert` and perform any other initialization tasks. For more information, see "`nsapi_module_init()` Function" on page 96 and "`func_insert()` Function" on page 84.

The server runs as a multi-threaded single process. On UNIX platforms, the server runs 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 in mind when writing your SAF:

- Write thread-safe code
- Blocking can affect performance
- Write small functions with parameters and configure the parameters in `obj.conf`
- Carefully check and handle all errors and log the errors 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 must be named `nsapi_module_init` and has the same signature as other SAFs:

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

SAFs should to be able to obtain certain types of information from their parameters. In most cases, parameter block (`pblock`) data structures provide the fundamental storage mechanism for these parameters. `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 31.

When defining a SAF, you do 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. Each directive requires its SAFs to behave in particular ways, and your SAF must conform to the requirements of the directive for which it was written. For details on what each directive requires of its SAFs, see "Required Behavior of SAFs for Each Directive" on page 35.

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 *install-dir*`/samples/nsapi` directory.

Adhere to the following guidelines for compiling and linking.

Including Directory and `nsapi.h` File

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

Libraries

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

The following table lists the library that you need to link to.

TABLE 2–1 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.

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.

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.

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	`-DXP_UNIX -D_REENTRANT -fPIC`
IRIX	`-o32 -exceptions -DXP_UNIX -KPIC`

Compiling and Linking in 64–bit Mode

On the Solaris platform, the server can run in either 32–bit or 64–bit mode. Because a 32–bit shared library cannot be used in a 64–bit process and conversely, you may want to compile and link two separate shared libraries. By default, the Sun compiler and linker produce 32–bit binaries. To compile and link your plug-in for 64–bit mode on Solaris SPARC, you must use Sun Workshop 5.0 or higher with the `-xarch=v9` flag. To compile and link your plug-in for 64–bit mode on Solaris x86, you must use Sun Java Studio 11 or higher with the `-xarch=amd64` flag.

Issues With Using C++ in a NSAPI Plug-in

NSAPI plug-ins are typically written using the C programming language. Using the C++ programming language in an NSAPI plug-in raises special compatibility issues.

On Solaris, the server is built using the new C++ 5 ABI. If your shared library uses C++, it must be compiled with Sun Workshop 5.0 or higher. Sun Java Studio 11 or higher is recommended. Do not use the `-compat=4` option when compiling and linking a shared library that uses C++. When running in 32–bit mode on Solaris SPARC, the server provides some backward compatibility for the old C++ 4 ABI or Sun Workshop 4.2. This backward compatibility may be removed at some future date. For all new NSAPI plug-ins, use the new C++ 5 ABI or Sun Workshop 5.0 or later versions.

On Linux, Web Server is built using the gcc 3.2 C++ ABI. If your shared library uses C++, compile with `gcc 3.2.x`. Because of the volatility of the `gcc` C++ ABI, avoid using C++ in NSAPI plug-ins on Linux.

Load and Initialize the SAF

For each shared library (plug-in) containing custom SAFs to be loaded into the server, add an `Init` directive that invokes the `load-modules` SAF to `magnus.conf`. The `load-modules` SAF loads the shared library and calls the shared library's `nsapi_module_init` function. For more information, see "`nsapi_module_init()` Function" on page 96.

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

```
Init fn=load-modules
     [shlib=path]
     [funcs="SAF1,...,SAFn"]
     [name1="value1"]...[nameN="valueN"]
```

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

  If the new SAFs require initialization, omit the `funcs` parameter and instead define an `nsapi_module_init` function in your shared library. Any custom parameters on the `Init` directive will be passed to `nsapi_module_init` in the `pb` parameter block.
- `nameN="`*valueN*`"` are the optional names and values of parameters passed to the shared library's `nsapi_module_init` function in the `pb` parameter block.

Instruct the Server to Call the SAFs

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 that are passed to the SAF.

Depending on the purpose of the new SAF, 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 gets 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:/docs/animations/small, while all files to be treated as full-screen animations reside in the directory D:/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="D:/docs/animations/small"
          name="small_anim"
NameTrans fn=pfx2dir
          from="/animations/fullscreen"
          dir="D:/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 and/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 Mozilla Firefox, hold the shift key while clicking the Reload button to ensure that the cache is not used.

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:

- Provide platform independence across operating system and hardware platforms
- Improved performance
- Thread-safe, a requirement for SAFs
- Prevent memory leaks
- Provide functionality necessary for implementing SAFs

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 detailed in Chapter 6, "NSAPI Function and Macro Reference."

The main categories of NSAPI functions are:

- Parameter Block Manipulation Routines
- Protocol Utilities for Service SAFs

- Memory Management
- File I/O
- Network I/O
- Threads
- Utilities
- Virtual Server

Parameter Block Manipulation Routines

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

- `pblock_findval` returns the value for a given name in a `pblock`. For more information, see "`pblock_findval()` Function" on page 102.
- `pblock_nvinsert` adds a new name-value pair entry to a `pblock`. For more information, see "`pblock_nvinsert()` Function" on page 104.
- `pblock_remove` removes a `pblock` entry by name from a `pblock`. The entry is not disposed. Use "`param_free()` Function" on page 99 to free the memory used by the entry. For more information, see "`pblock_remove()` Function" on page 106.
- `param_free` frees the memory for the given `pblock` entry. For more information, see "`param_free()` Function" on page 99.
- `pblock_pblock2str` creates a new string containing all of the name-value pairs from a `pblock` in the form "*name*=*value* *name*=*value*." This function can be a useful for debugging. For more information, see "`pblock_pblock2str()` Function" on page 105.

Protocol Utilities for Service SAFs

Protocol utilities provide functionality necessary to implement `Service` SAFs:

- `protocol_status` sets the HTTP response status code and reason phrase. For more information, see "`protocol_status()` Function" on page 114.
- `protocol_start_response` sends the HTTP response and all HTTP headers to the browser. For more information, see "`protocol_start_response()` Function" on page 113.

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. Wrappers enable standard memory routines to use permanent memory.

To disable the server's pooled memory allocator for debugging,use the built-in SAF `pool-init`. For more information, see the *Sun Java System Web Server 7.0 Update 3 Administrator's Configuration File Reference*.

- "`MALLOC()` Macro" on page 87
- "`FREE()` Macro" on page 82
- "`PERM_STRDUP()` Macro" on page 111
- "`REALLOC()` Macro" on page 118
- "`CALLOC()` Macro" on page 67
- "`PERM_MALLOC()` Macro" on page 109
- "`PERM_FREE()` Macro" on page 108
- "`PERM_STRDUP()` Macro" on page 111
- "`PERM_REALLOC()` Macro" on page 110
- "`PERM_CALLOC()` Macro" on page 108

File I/O

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

- `system_fopenRO` opens a file for read-only access. For more information, see "`system_fopenRO()` Function" on page 129.
- `system_fopenRW` opens a file for read-write access, creating the file if necessary. For more information, see "`system_fopenRW()` Function" on page 130.
- `system_fopenWA` opens a file for write-append access, creating the file if necessary. For more information, see "`system_fopenWA()` Function" on page 130.
- `system_fclose` closes a file. For more information, see "`system_fclose()` Function" on page 128.
- `system_fread` reads from a file. For more information, see "`system_fread()` Function" on page 131.
- `system_fwrite` writes to a file. For more information, see "`system_fwrite()` Function" on page 132.
- `system_fwrite_atomic` locks the given file before writing to it. This locking avoids interference between simultaneous writes by multiple processes or threads. For more information, see "`system_fwrite_atomic()` Function" on page 132.

Network I/O

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

- `netbuf_grab` reads from a network buffer's socket into the network buffer. For more information, see "`netbuf_grab()` Function" on page 95.
- `netbuf_getbytes` gets a character from a network buffer. For more information, see "`netbuf_getbytes()` Function" on page 93.
- net_flush flushes buffered data. For more information, see "`net_flush()` Function" on page 88.
- `net_read` reads bytes from a specified socket into a specified buffer. For more information, see "`net_read()` Function" on page 89.
- `net_sendfile` sends the contents of a specified file to a specified a socket. For more information, see "`net_sendfile()` Function" on page 90.
- `net_write` writes to the network socket. For more information, see "`net_write()` Function" on page 91.

Threads

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

- `systhread_start` creates a new thread. For more information, see "`systhread_start()` Function" on page 140.
- `systhread_sleep` puts a thread to sleep for a given time. For more information, see "`systhread_sleep()` Function" on page 139.
- `crit_init` creates a new critical section variable. For more information, see "`crit_init()` Function" on page 72.
- `crit_enter` gains ownership of a critical section. For more information, see "`crit_enter()` Function" on page 71.
- `crit_exit` surrenders ownership of a critical section. For more information, see "`crit_exit()` Function" on page 72.
- `crit_terminate` disposes of a critical section variable. For more information, see "`crit_terminate()` Function" on page 73.
- `condvar_init` creates a new condition variable. For more information, see "`condvar_init()` Function" on page 69.
- `condvar_notify` awakens any threads blocked on a condition variable. For more information, see "`condvar_notify()` Function" on page 69.
- `condvar_wait` blocks on a condition variable. For more information, see "`condvar_wait()` Function" on page 70.
- `condvar_terminate` disposes of a condition variable. For more information, see "`condvar_terminate()` Function" on page 70.

- `prepare_nsapi_thread` enables threads that are not created by the server to act like server-created threads. For more information, see "`prepare_nsapi_thread()` Function" on page 111.

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` registers a user function to be called when the server is sent a restart signal (`HUP`) or at shutdown. For more information, see "`daemon_atrestart()` Function" on page 73.
- `daemon_atrestart` gets the local host name as a fully qualified domain name. For more information, see "`util_hostname()` Function" on page 147.
- `util_later_than` compares two dates. For more information, see "`util_later_than()` Function" on page 149.
- `util_sprintf` is the same as the standard library routine `sprintf()`. For more information, see "`util_sprintf()` Function" on page 151.
- `util_strftime` is the same as the standard library routine `strftime()`. For more information, see "`util_strftime()` Function" on page 152.
- `util_uri_escape` converts the special characters in a string into URI-escaped format. For more information, see "`util_uri_escape()` Function" on page 154.
- `util_uri_unescape` converts the URI-escaped characters in a string back into special characters. For more information, see "`util_uri_unescape()` Function" on page 155.

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 does not work.

Virtual Server

The virtual server functions provide routines for retrieving information about virtual servers.

- `request_get_vs` finds the virtual server to which a request is directed. For more information, see "`request_get_vs()` Function" on page 119.
- `vs_alloc_slot` allocates a new slot for storing a pointer to data specific to a certain virtual server. For more information, see "`vs_alloc_slot()` Function" on page 157.
- `vs_get_data` finds the value of a pointer to data for a given virtual server and slot. For more information, see "`vs_get_data()` Function" on page 158.

- `vs_get_default_httpd_object` obtains a pointer to the default or root object from the virtual server's virtual server class configuration. For more information, see "`vs_get_default_httpd_object()` Function" on page 158.
- `vs_get_doc_root` finds the document root for a virtual server. For more information, see "`vs_get_doc_root()` Function" on page 159.
- `vs_get_httpd_objset` obtains a pointer to the virtual server class configuration for a given virtual server. For more information, see
- `vs_get_id` finds the ID of a virtual server. For more information, see "`vs_get_id()` Function" on page 160.
- `vs_get_mime_type` determines the MIME type that would be returned in the `content-type:` header for the given URI. For more information, see "`vs_get_mime_type()` Function" on page 160.
- `vs_lookup_config_var` finds the value of a configuration variable for a given virtual server. For more information, see "`vs_lookup_config_var()` Function" on page 161.
- `vs_register_cb` enables a plug-in to register functions that will receive notifications of virtual server initialization and destruction events. For more information, see "`vs_register_cb()` Function" on page 161.
- `vs_set_data` sets the value of a pointer to data for a given virtual server and slot. For more information, see "`vs_set_data()` Function" on page 162.
- `vs_translate_uri` translates a URI as though it were part of a request for a specific virtual server. For more information, see "`vs_translate_uri()` Function" on page 163.

Required Behavior of SAFs for Each Directive

When writing a new SAF, you should define it to accomplish certain actions, depending on which stage of the request-handling process will invoke it. 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 along 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. The SAFs are described in the following sections:

- "`Init()` SAFs" on page 36

For more detailed information about these SAFs, see the *Sun Java System Web Server 7.0 Update 3 Administrator's 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

- Purpose: Verify any authorization information.
- Return `REQ_PROCEED` if the user was successfully and completely authenticated, `REQ_NOACTION` otherwise.

`NameTrans()` SAFs

- Purpose: Convert logical URI to physical path.
- Perform operations on 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, add `url` to `rq->vars` with full URL. For example, `http://www.sun.com/`. Call `protocol_status()` to set HTTP response status to `PROTOCOL_REDIRECT`, `NULL`. Return `REQ_ABORTED`.

`PathCheck()` SAFs

- Purpose: Check path validity and user's access rights.
- Check `auth-type`, `auth-user`, and/or `auth-group` in `rq->vars.`
- Return `REQ_PROCEED` if user and group is 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 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 it 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 it 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_status to set HTTP response status. For more information, see "protocol_status() Function" on page 114.
- Call protocol_start_response to send HTTP response and headers. For more information, see "protocol_start_response() Function" on page 113.
- Generate and send data to the client using net_write. For more information, see "net_write() Function" on page 91.
- Return REQ_PROCEED if successful, REQ_EXIT on write error, 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 do the same 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.

CGI to NSAPI Conversion

.The CGI environment variables are not available to NSAPI. Therefore, if you need to convert a CGI variable into an SAF using NSAPI, you retrieve them from the NSAPI parameter blocks. The following table indicates how each CGI environment variable can be obtained in NSAPI.

Keep in mind that 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.

TABLE 2–2 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);`
`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);`
`SERVER_NAME`	`char *util_hostname();`
`SERVER_PORT`	`conf_getglobals()->Vport;` (as a string)
`SERVER_PROTOCOL`	`pblock_findval("protocol", rq->reqpb);`
`SERVER_SOFTWARE`	`system_version()`
Sun Java System-specific:	

TABLE 2–2 Parameter Blocks for CGI Variables *(Continued)*

CGI getenv()	NSAPI
`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);`
`SERVER_URL`	`protocol_uri2url_dynamic("","", sn, rq);`

CHAPTER 3

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 has the following sections:

Future Compatibility Issues

The NSAPI interface may change in a future version of Sun Java System Web Server.

To keep your custom plug-ins upgradable, do the following:

- 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

The NSAPI filter interface complements the NSAPI Server Application Function (SAF) interface. Filters enable functions 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. To create a filter, a filter developer implements one or more of these methods.

This section describes the following filter methods:

- `insert()`
- `remove()`
- `flush()`
- `read()`
- `write()`
- `writev()`
- `sendfile()`

For more information about these methods, see Chapter 6, "NSAPI Function and Macro 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.

The most important fields in the `FilterLayer` data structure are:

- `context->sn`: Contains information relating to a single TCP/IP session (the same `sn` pointer that's passed to SAFs).
- `context->rq`: Contains information relating to the current request (the same `rq` pointer that's 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` Data Structure" on page 173.

`insert` Filter Method

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, the filter can determine whether it 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 it is interested in the type of data that will be transmitted. If the filter should not be inserted, the `insert` filter method should indicate this by returning `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.

For more information, see "`insert()` Function" on page 85 in Chapter 6, "NSAPI Function and Macro Reference."

`remove` Filter Method

The `remove` filter method is called when a filter stack is destroyed (that is, 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, ...)`.

For more information, see "`remove()` Function" on page 118 in Chapter 6, "NSAPI Function and Macro Reference."

`flush` Filter Method

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.

For more information, see "`flush()` Function" on page 82 in Chapter 6, "NSAPI Function and Macro Reference."

`read` Filter Method

The `read` filter method is called when a filter or SAF calls the `net_read` function. Filters that are interested in incoming data, that is, 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.

For more information, see "`read()` Function" on page 117 in Chapter 6, "NSAPI Function and Macro Reference."

`write` Filter Method

The `write` filter method is called when a filter or SAF calls the `net_write` function. Filters that are interested in outgoing data, that is, 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.

For more information, see "`write()` Function" on page 163 in Chapter 6, "NSAPI Function and Macro Reference."

`sendfile` Filter Method

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 might run slightly faster when filters that implement the `write` filter method also implement the `sendfile` filter method.

For more information, see "`sendfile()` Function" on page 122 in Chapter 6, "NSAPI Function and Macro Reference"

`writev` Filter Method

The `writev` filter method performs the same function as the `write` filter method, but the format of its parameters is different. It is not necessary 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.

For more information, see "`writev()` Function" on page 164 in Chapter 6, "NSAPI Function and Macro 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 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, while filters positioned lower in the stack are given an earlier opportunity to process incoming data.

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()` Function” on page 77 and `init-filter-order` in the *Sun Java System Web Server 7.0 Update 3 Administrator's Configuration File Reference*.

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, so that an input filter can alter the length of the incoming request body. For example, if a filter decompresses incoming data and a `Content-Length` header is in the `rq->headers` `pblock`, 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 the “Syntax and Use of `obj.conf`”chapter in the *Sun Java System Web Server 7.0 Update 3 Administrator's 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. Foor 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. The general steps for creating a custom filter are as follows:

- Write the source code using the NSAPI functions.
- Compile and link the source code to create a shared library (`.so`, `.sl`, or `.dll`) file.
- Load and initialize the filter by editing the `magnus.conf` file.
- Instruct the server to insert the filter by editing the `obj.conf` file to insert your custom filter(s) at the appropriate time.
- Restart the server.
- Test the filter 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 50 and “Filter Methods” on page 42 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. For more information, see “`nsapi_module_init()` Function” on page 96 and “`filter_create()` Function” on page 77.

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 could 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 48.

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 the chapter Syntax and Use of `magnus.conf` in the *Sun Java System Web Server 7.0 Update 3 Administrator's 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 32.
- Thread synchronization should only be performed using NSAPI functions documented in "Threads" on page 33.
- Blocking might affect performance.
- Carefully check and handle all errors.

For examples of custom filters, see Chapter 4, "Examples of Custom SAFs and Filters."

Compiling and Linking

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

Loading and Initializing the Filter

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

```
Init fn=load-modules shlib=path 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, as described in "Writing the Source Code" on page 47.

  When the server encounters such a directive, it 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 filter by accessing your server from a web browser. You should disable caching in your web browser so that the server is sure to be accessed. In Mozilla Firefox, you may hold the shift key while clicking the Reload button to ensure that the cache is not used. 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. All of the public routines are described in detail in Chapter 6, "NSAPI Function and Macro Reference."

The NSAPI functions specific to the development of filters are:

- `filter_create()` creates a new filter
- `filter_insert()` inserts the specified filter into a filter stack
- `filter_remove()` removes the specified filter from a filter stack
- `filter_name()` returns the name of the specified filter
- `filter_find()` finds an existing filter given a filter name
- `filter_layer()` returns the layer in a filter stack that corresponds to the specified filter

◆ ◆ ◆ CHAPTER 4

Examples of Custom SAFs and Filters

This chapter provides examples of custom Sever 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 Server Application Functions." For information about creating your own filters, see Chapter 3, "Creating Custom Filters."

Before writing your own SAF, check to see whether an existing SAF serves your purpose. The predefined SAFs are discussed in the *Sun Java System Web Server 7.0 Update 3 Administrator's Configuration File Reference*.

If you need to writer a custom SAF, you should be familiar with the request-response process and the role of the configuration file `obj.conf`. See the *Sun Java System Web Server 7.0 Update 3 Administrator's Configuration File Reference* for information on the `obj.conf` file.

For a list of the NSAPI functions for creating new SAFs, see Chapter 6, "NSAPI Function and Macro Reference."

This chapter has the following sections:

Using the NSAPI Examples

The *install-dir*/`samples/nsapi` directory contains examples of source code for SAFs.

You can use the `example.mak` (Windows) or `Makefile` (UNIX) makefile in the same directory to compile the examples and create shared libraries containing the functions in all of the example files.

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

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

The `shlib` parameter specifies the path to the shared library, for example, `../../samples/nsapi/examples.so`, and the `funcs` parameter specifies the functions to load from the shared library.

If the example uses an initialization function, be sure to 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
     shlib="path"
     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 `magnus.conf`, restart the Web 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 your own custom methods to verify that the user name and password that a remote client provides 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. Generally, an AuthTrans function checks whether the user name and password associated with the request are acceptable. However, it does not allow or deny access to the request. The PathCheck function handles access.

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 reject the request, because the user name and password have not yet been submitted. When the client receives the rejection, the usual response is to present a dialog box asking the user for their 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 the Web Server, add the following Init directive to magnus.conf to load the compiled function:

```
Init fn=load-modules
     shlib="path"
     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
```

Note that this function does not actually enforce authorization requirements. It only takes given information and tells the server whether it is correct. The PathCheck function require-auth performs the enforcement. Therefore, add the following PathCheck directive:

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

The source code for this example is in the `auth.c` file in the *install-dir*/`samples/nsapi/`directory.

NameTrans() Example

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

- `explicit_pathinfo`— Enables the use of explicit extra path information in a URL.
- `https_redirect` — Redirects the URL if the client is a particular version of Netscape Navigator.

This section discusses the first example. The second example is found in `ntrans.c`.

Note – A `NameTrans` function is used primarily to convert the logical URL in `ppath` in `rq->vars` to a physical path name. However, the example discussed here, `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 enables 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, a CGI program is `http://`*server-name*`/cgi/marketing`. The extra path information to give to the CGI program is `/jan/releases/hardware`.

When choosing a separator, be sure to pick a character that is never used as part of a real URL.

The `explicit_pathinfo` function reads the URL, strips out everything following the comma, and puts the string 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.

One side effect of `explicit_pathinfo` is that the `SCRIPT_NAME` CGI environment variable has the separator character appended to the end.

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 needs to continue after the path info is extracted, because the URL to a physical path name has not yet been translated.

Installing the NameTrans Example

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

```
Init fn=load-modules
     shlib="path"
     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.

The source code for this example is in the ntrans.c file in the *install-dir*/smaples/nsapi/directory.

PathCheck() Example

The example in this section demonstrates how to implement a custom SAF for performing path checks. This example simply 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 directive in the `Init` section of the `magnus.conf` file:

```
Init fn=load-modules
     shlib="path"
     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 `magnus.conf`. This line must come after the one 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
```

The source code for this example is in `pcheck.c` in the *install-dir*/`samples/nsapi/`directory.

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 a `.shtml` file if a `.shtml` version of the requested file exists.

A well-behaved `ObjectType` function checks whether the content type is already set. If the type is set, the function returns `REQ_NOACTION`.

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

If the content type is not set, `ObjectType` directive sets the content type. 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 the requested file name. If the name ends with .html, the function searches for a file with the same base name, with the extension .shtml. If a .shtml is found, the function uses that path and informs the server that the file is parsed HTML instead of regular HTML. Note that this check 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 directive in the Init section of the magnus.conf file:

```
Init fn=load-modules
     shlib="path"
     funcs=html2shtml
```

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

```
ObjectType fn=html2shtml
```

The source code for this example is in otype.c in the *install-dir*/samples/nsapi/ directory.

Output() Example

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

Installing the Output() Example

To load the filter, add the following directive in the Init section of the magnus.conf file:

```
Init fn="load-modules"
     shlib=yourlibrary
     NativeThread="no"
```

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

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

The source code for this example is in the `replace.c` file in the *install-dir*/`samples/nsapi/` directory.

`Service()` Example

This section discusses two `Service()` function examples: one simple and one more complex.

Simple `Service()` Example

This section describes 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.

To load the shared object containing your functions, add the following directive in the `Init` section of the `magnus.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 directive to the `Init` section in `magnus.conf`. This directive must come after the directive 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 code 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`.

The source code for this example is in the `service.c` file in the *install-dir*/`samples/nsapi` directory.

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 not, 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 directive at the beginning of the magnus.conf file:

```
Init fn=load-modules
        shlib=your-library
        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 code 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 for this example is in the service.c file in the *install-dir*/samples/nsapi directory.

AddLog() Example

The example in this section demonstrates how to implement brief-log, a custom SAF for logging 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 directive in the Init section of the magnus.conf file:

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

To call brief-init to open the log file, add the following code to the Init section in magnus.conf. This line must come after the one 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
```

The source code for this example is in addlog.c file in the *install-dir*/samples/nsapi directory.

Quality of Service() Example

The code for the qos-handler (AuthTrans) and qos-error (Error) SAFs is provided in case you want to define your own SAFs for quality of service handling.

For more information about predefined SAFs, see the *Sun Java System Web Server 7.0 Update 3 Administrator's Configuration File Reference*.

Installing the Quality of Service() Example

Inside the default object in obj.conf, add the following AuthTrans and Error directives:

```
AuthTrans fn=qos-handler
...
Error fn=qos-error code=503
```

The source code for this example is in the qos.c file in the samples/nsapi subdirectory of the server root directory.

◆ ◆ ◆ CHAPTER 5

Creating Custom Server-Parsed HTML Tags

This chapter describes the procedure to create customer server-parsed HTML tags. This chapter contains the following sections:

Defining Custom Server-parsed HTML Tags

HTML files can contain tags that are executed on the server. For general information about server-parsed HTML tags, see the *Sun Java System Web Server 7.0 Update 3 Developer's Guide*.

In Web Server 7.0, you can define your own server-side tags. For example, you could define the tag `HELLO` to invoke a function that prints `Hello World!` You could have the following code in your `hello.shtml` file:

```
<html>
    <head>
            <title>shtml custom tag example</title>
    </head>
    <body>
            <!--#HELLO-->
    </body>
</html>
```

When the browser displays this code, each occurrence of the `HELLO` tag calls the function.

The general steps for defining a customized server-parsed tag are:

- “Defining the Functions that Implement the Tag” on page 62.

You must define the tag execution function. You must also define other functions that are called on tag loading and unloading, and on page loading and unloading.

- "Writing an Initialization Function" on page 65.

  Write an initialization function that registers the tag using the `shtml_add_tag` function.
- "Loading the New Tag into the Server" on page 66.

Defining the Functions that Implement the Tag

Define the functions that implement the tags in C, using NSAPI. Include the header `shtml_public.h`, which is in the directory *install-dir*`/include/shtml`. Link against the SHTML shared library in the *install_-dir*`/lib` directory. On Windows, the SHTML shared library is named `sshtml.dll`. On UNIX platforms, the library is named `libShtml.so` or `libShtml.sl`.

`ShtmlTagExecuteFunc` is the tag handler which gets called with the NSAPI `pblock`, `Session`, and `Request` variables. In addition, this handler also gets passed the `TagUserData` created from the result of executing the tag loading and page loading functions, if any such functions are defined for that tag.

The signature for the tag execution function is:

```
typedef int (*ShtmlTagExecuteFunc)
              (pblock*, Session*, Request*, TagUserData, TagUserData);
```

Write the body of the tag execution function to generate the output to replace the tag in the `.shtml` page. Use the `net_write` NSAPI function, which writes a specified number of bytes to a specified socket from a specified buffer.

For more information about writing NSAPI plug-ins, see Chapter 2, "Creating Custom Server Application Functions."

For more information about `net_write` and other NSAPI functions, see Chapter 6, "NSAPI Function and Macro Reference."

The tag execution function must return an `int` value that indicates whether the server should proceed to the next instruction in `obj.conf`, which is one of the following:

- `REQ_PROCEED` — the execution was successful
- `REQ_NOACTION` — nothing happened
- `REQ_ABORTED` — an error occurred
- `REQ_EXIT` — the connection was lost

The other functions you must define for your tag are:

- ShtmlTagInstanceLoad — Called when a page containing the tag is parsed. This function is not called if the page is retrieved from the browser's cache. It serves as a constructor, the result of which is cached and is passed into ShtmlTagExecuteFunc whenever the execution function is called.
- ShtmlTagInstanceUnload — A destructor for cleaning up whatever was created in the ShtmlTagInstanceLoad function. This function gets passed the result that was originally returned from the ShtmlTagInstanceLoad function.
- ShtmlTagPageLoadFunc — Called when a page containing the tag is executed, regardless of whether the page is still in the browser's cache. This function provides a way to make information persistent between occurrences of the same tag on the same page.
- ShtmlTagPageUnLoadFn — Called after a page containing the tag has executed. This function provides a way to clean up any allocations done in a ShtmlTagPageLoadFunc and hence gets passed the result returned from the ShtmlTagPageLoadFunc.

The signature for these functions are:

```
#define TagUserData void*
typedef TagUserData (*ShtmlTagInstanceLoad)
                    (const char* tag, pblock*, const char*, size_t);
typedef void (*ShtmlTagInstanceUnload)(TagUserData);
typedef int (*ShtmlTagExecuteFunc)
            (pblock*, Session*, Request*, TagUserData, TagUserData);
typedef TagUserData (*ShtmlTagPageLoadFunc)
                    (block* pb, Session*, Request*);
typedef void (*ShtmlTagPageUnLoadFunc)(TagUserData);
```

The following code example implements the HELLO tag:

```
/*
 * mytag.c: NSAPI functions to implement #HELLO SSI calls
*/
#include "nsapi.h"
#include "shtml/shtml_public.h"
/* FUNCTION : mytag_con
 *
 * DESCRIPTION: ShtmlTagInstanceLoad function
 */
#ifdef __cplusplus
extern "C"
#endif
TagUserData
mytag_con(const char* tag, pblock* pb, const char* c1, size_t t1)
{
    return NULL;
```

```
}
/* FUNCTION : mytag_des
 *
 * DESCRIPTION: ShtmlTagInstanceUnload
 */
#ifdef __cplusplus
extern "C"
#endif
void
mytag_des(TagUserData v1)
{
}
/* FUNCTION : mytag_load
 * DESCRIPTION: ShtmlTagPageLoadFunc
 */
#ifdef __cplusplus
extern "C"
#endif
TagUserData
mytag_load(pblock *pb, Session *sn, Request *rq)
{
    return NULL;
}
/* FUNCTION : mytag_unload
 *
 * DESCRIPTION: ShtmlTagPageUnloadFunc
 */
#
#ifdef __cplusplus
extern "C"
#endif
void
mytag_unload(TagUserData v2)
{
}
/* FUNCTION : mytag
    * DESCRIPTION: ShtmlTagExecuteFunc
 */
#ifdef __cplusplus
extern "C"
#endif
int
mytag(pblock* pb, Session* sn, Request* rq, TagUserData t1, TagUserData t2)
{
    char* buf;
    int length;
    char* client;
    buf = (char *) MALLOC(100*sizeof(char));
    length = util_sprintf(buf, "<h1>Hello World! </h1>", client);
    if (net_write(sn->csd, buf, length) == IO_ERROR)
    {
        FREE(buf);
        return REQ_ABORTED;
    }
    FREE(buf);
    return REQ_PROCEED;
}
/* FUNCTION : mytag_init
    * DESCRIPTION: initialization function, calls shtml_add_tag() to
* load new tag
*/
#
#ifdef __cplusplus
extern "C"
#endif
int
mytag_init(pblock* pb, Session* sn, Request* rq)
{
    int retVal = 0;
// NOTE: ALL arguments are required in the shtml_add_tag() function
    retVal = shtml_add_tag("HELLO", mytag_con, mytag_des, mytag, mytag_load, mytag_unload);
return retVal;
}
/* end mytag.c */
```

Writing an Initialization Function

In the initialization function for the shared library that defines the new tag, register the tag using the function `shtml_add_tag`. The signature is:

```
NSAPI_PUBLIC int shtml_add_tag (
             const char* tag,
             ShtmlTagInstanceLoad ctor,
             ShtmlTagInstanceUnload dtor,
             ShtmlTagExecuteFunc execFn,
             ShtmlTagPageLoadFunc pageLoadFn,
             ShtmlTagPageUnLoadFunc pageUnLoadFn);
```

Any of these arguments can return NULL except for the `tag` and `execFn`.

Loading the New Tag into the Server

After creating the shared library that defines the new tag, you load the library into the Web Server. Add the following directives to the configuration file `magnus.conf`:

An `Init` directive whose `fn` parameter is `load-modules` and whose `shlib` parameter is the shared library to load. For example, if you compiled your tag into the shared object *install-dir*`/hello.so`, the `Init` directive would be:

```
Init funcs="mytag,mytag_init" shlib="install-dir/hello.so" fn="load-modules"
```

An `Init` directive whose `fn` parameter is the initialization function in the shared library that uses `shtml_add_tag` to register the tag. For example:

```
Init fn="mytag_init"
```

CHAPTER 6

NSAPI Function and Macro Reference

This chapter lists all the public C functions and macros of the Netscape Server Applications Programming Interface (NSAPI). Use these functions when writing your own Server Application Functions (SAFs) and filters.

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 7, "Data Structure Reference."

NSAPI Functions and Macros

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

C	D	F	I	L	M	N	P	R	S	U	V	W

C

`CALLOC()` Macro

The `CALLOC` macro is a platform-independent substitute for the C library routine `calloc`. It allocates `size` bytes from the request's memory pool and initializes the memory to zeros. The memory can be explicitly freed by a call to `FREE`. If the memory is not explicitly freed, it is automatically freed after processing of the current request 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. However, because memory allocated by `CALLOC` is automatically freed, the memory should not be shared with threads.

Syntax

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

Return Values

A void pointer to a block of memory.

Parameters

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

Example

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

See Also

“`FREE()` Macro” on page 82, “`MALLOC()` Macro” on page 87, “`REALLOC()` Macro” on page 118, “`STRDUP()` Macro” on page 127, “`PERM_CALLOC()` Macro” on page 108

`cinfo_find()` Function

The `cinfo_find()` function uses the MIME types information to find the type, encoding, 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 the client will be receiving from the server.

The URI name used is everything after the last slash (/) or the whole string if no slash is found. File name extensions are not case sensitive. The name can contain multiple extensions separated by a period (.) to indicate type, encoding, or language. For example, the URI `a/b/filename.jp.txt.zip` represents 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 the find succeeds, or NULL if the find fails.

The `cinfo` structure that is allocated and returned contains pointers to the `content-type`, `content-encoding`, and `content-language`, if found. Each structure points to static data in the types database, or NULL if not found. Do not free these pointers. You should free the `cinfo` structure after 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()` Function

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);
```

Return Values

A newly allocated condition variable (`CONDVAR`).

Parameters

`CRITICAL id` is a critical-section variable.

See Also

“`condvar_notify()` Function” on page 69, “`condvar_terminate()` Function” on page 70, “`condvar_wait()` Function” on page 70, “`crit_init()` Function” on page 72, “`crit_enter()` Function” on page 71, “`crit_exit()` Function” on page 72, “`crit_terminate()` Function” on page 73

`condvar_notify()` Function

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);
```

Return Values

`void`

Parameters

`CONDVAR cv` is a condition variable.

See Also

"`condvar_init()` Function" on page 69, "`condvar_terminate()` Function" on page 70, "`condvar_wait()` Function" on page 70, "`crit_init()` Function" on page 72, "`crit_enter()` Function" on page 71, "`crit_exit()` Function" on page 72, "`crit_terminate()` Function" on page 73

`condvar_terminate()` Function

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

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

Syntax

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

Return Values

`void`

Parameters

`CONDVAR cv` is a condition variable.

See Also

"`condvar_init()` Function" on page 69, "`condvar_notify()` Function" on page 69, "`condvar_wait()` Function" on page 70, "`crit_init()` Function" on page 72, "`crit_enter()` Function" on page 71, "`crit_exit()` Function" on page 72, "`crit_terminate()` Function" on page 73

`condvar_wait()` Function

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);
```

Return Values

`void`

Parameters

`CONDVAR cv` is a condition variable.

See Also

"`condvar_init()` Function" on page 69, "`condvar_terminate()` Function" on page 70, "`condvar_notify()` Function" on page 69, "`crit_init()` Function" on page 72, "`crit_enter()` Function" on page 71, "`crit_exit()` Function" on page 72, "`crit_terminate()` Function" on page 73

`crit_enter()` Function

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);
```

Return Values

`void`

Parameters

`CRITICAL crvar` is a critical-section variable.

See Also

"`crit_init()` Function" on page 72, "`crit_exit()` Function" on page 72, "`crit_terminate()` Function" on page 73

`crit_exit()` Function

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 is removed and the waiting thread is given ownership of the section.

Syntax

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

Return Values

`void`

Parameters

`CRITICAL` `crvar` is a critical-section variable.

See Also

"`crit_init()` Function" on page 72, "`crit_enter()` Function" on page 71, "`crit_terminate()` Function" on page 73

`crit_init()` Function

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. Use this variable to prevent interference between two threads of execution. At the time this variable is created, no thread owns the critical section.

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

Syntax

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

Return Values

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

Parameters

none

See Also

"`crit_enter()` Function" on page 71, "`crit_exit()` Function" on page 72, "`crit_terminate()` Function" on page 73

`crit_terminate()` Function

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);
```

Return Values

`void`

Parameters

`CRITICAL crvar` is a critical-section variable.

See Also

"`crit_init()` Function" on page 72, "`crit_enter()` Function" on page 71, "`crit_exit()` Function" on page 72

D

`daemon_atrestart()` Function

The `daemon_atrestart` function enables to you register a callback function named `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);
```

Return Values

`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);}
```

F

`filebuf_buf2sd()` Function

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);
```

Return Values

The number of bytes sent to the socket if successful, or the constant `IO_ERROR` if the file buffer cannot 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 parameter is obtained from the `csd`, client socket descriptor field of the `sn` or the `session` structure.

Example

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

See Also

"filebuf_close() Function" on page 75, "filebuf_open() Function" on page 76, "filebuf_open_nostat() Function" on page 77, "filebuf_getc() Function" on page 75

filebuf_close() Function

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

Generally, 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);
```

Return Values

```
void
```

Parameters

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

Example

```
filebuf_close(buf);
```

See Also

"filebuf_open() Function" on page 76, "filebuf_open_nostat() Function" on page 77, "filebuf_buf2sd() Function" on page 74, "filebuf_getc() Function" on page 75

filebuf_getc() Function

The filebuf_getc function retrieves a character from the current file position and returns the character 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);
```

Return Values

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

Parameters

`filebuf b` is the name of the file buffer.

See Also

“`filebuf_close()` Function” on page 75, “`filebuf_buf2sd()` Function” on page 74, “`filebuf_open()` Function” on page 76, “`filter_create()` Function” on page 77

`filebuf_open()` Function

The `filebuf_open` function opens a new file buffer for a previously opened file. It 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);
```

Return Values

A pointer to a new buffer structure to hold the data if successful, or NULL if no buffer can 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()` Function” on page 75, “`filebuf_buf2sd()` Function” on page 74, “`filebuf_close()` Function” on page 75, “`filebuf_open_nostat()` Function” on page 77

filebuf_open_nostat() Function

The filebuf_open_nostat function opens a new file buffer for a previously opened file. It 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 it 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);
```

Return Values

A pointer to a new buffer structure to hold the data if successful, or NULL if no buffer can 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() Function" on page 75, "filebuf_open() Function" on page 76, "filebuf_getc() Function" on page 75, "filebuf_buf2sd() Function" on page 74

filter_create() Function

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, it 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 what class of functionality the filter implements.

The following table describes parameters allowed 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 6–1 `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 initialize the `FilterMethods` 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 supports.

`filter_create` returns `const Filter *`, a pointer to an opaque representation of the filter. This value can 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);
```

Return Values

The `const Filter *` that identifies the filter, or NULL if an error occurs.

Parameters

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

`int order` is one of the order constants listed in Table Table 6–1.

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

Example

```
FilterMethods methods = FILTER_METHODS_INITIALIZER;
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);
```

See Also

"`filter_insert()` Function" on page 79, "`insert()` Function" on page 85, "`flush()` Function" on page 82, "`read()` Function" on page 117, "`sendfile()` Function" on page 122, "`write()` Function" on page 163, "`writev()` Function" on page 164, "`FilterMethods` Data Structure" on page 173

`filter_find()` Function

The `filter_find` function finds the filter with the specified name.

Syntax

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

Return Values

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()` Function

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 specified when `filter_create` 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 are 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.

Note – When possible, plug-in developers should avoid calling filter_insert directly, and instead use the insert-filter SAF.

Syntax

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

Return Values

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, and is reserved for future use.

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

Session *sn is the session.

Request *rq is the request.

void *data is filter-defined private data.

const Filter *filter is the filter to insert.

See Also

"filter_create() Function" on page 77

filter_layer() Function

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);
```

Return Values

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()` Function

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);
```

Return Values

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

Parameters

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

`filter_remove()` Function

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 the top filter layer of the filter is destroyed.

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

Syntax

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

Return Values

`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()` Function

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` function:

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

Syntax

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

Return Values

`0` on success or `-1` if an error occurs.

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()` Function” on page 88, “`filter_create()` Function” on page 77

`FREE()` Macro

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

Note – Calling FREE for a block that was allocated with PERM_MALLOC, PERM_CALLOC, or PERM_STRDUP will not work.

Syntax

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

Return Values

```
void
```

Parameters

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

Example

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

See Also

"CALLOC() Macro" on page 67, "MALLOC() Macro" on page 87, "REALLOC() Macro" on page 118, "STRDUP() Macro" on page 127, "PERM_FREE() Macro" on page 108

func_exec() Function

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

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

Syntax

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

Return Values

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

Parameters

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

Session *sn is the session.

Request *rq is the request.

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

See Also

"log_error() Function" on page 86

func_find() Function

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

Syntax

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

Return Values

A pointer to the chosen function, suitable for de-referencing, or NULL if the function is not 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() Function" on page 83

func_insert() Function

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);
```

Return Values

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() Function" on page 83, "func_find() Function" on page 84

I

insert() Function

The insert filter method is called when a filter is inserted into a filter stack by the filter_insert function or insert-filter SAF.

Syntax

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

Return Values

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

```
int myfilter_insert(FilterLayer *layer, pblock *pb)
{
        if (pblock_findval("dont-insert-filter", pb))
        return REQ_NOACTION;
        return REQ_PROCEED;
}
...

FilterMethods myfilter_methods = FILTER_METHODS_INITIALIZER;
const Filter *myfilter;

myfilter_methods.insert = &myfilter_insert;
myfilter = filter_create("myfilter", &myfilter_methods);
...
```

See Also

“`filter_insert()` Function” on page 79, “`filter_create()` Function” on page 77

L

`log_error()` Function

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

Syntax

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

Return Values

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

Parameters

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

- `LOG_VERBOSE` — Debug message
- `LOG_VERBOSE` — Debug message
- `LOG_INFORM` — Information message
- `LOG_WARN` — Warning

- LOG_FAILURE — Operation failed
- LOG_MISCONFIG— Misconfiguration
- LOG_SECURITY — Authentication or authorization failure
- LOG_CATASTROPHE— Nonrecoverable server error

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

Session *sn is the session.

Request *rq is the request.

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

Example

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

See Also

"func_exec() Function" on page 83

M

MALLOC() Macro

The MALLOC macro is a platform-independent substitute for the C library routine malloc. It allocates size bytes from the requests's memory pool. The memory can be explicitly freed by a call to FREE. If the memory is not explicitly freed, it is automatically freed after processing of the current request has been completed. If pooled memory has been disabled in the configuration file with the built-in SAF pool-init , PERM_MALLOC and MALLOC both obtain their memory from the system heap. However, because memory allocated by MALLOC is automatically freed, it should not be shared between threads.

If pooled memory has been disabled in the configuration file with the built-in SAF pool-init, , PERM_MALLOC and MALLOC both obtain their memory from the system heap.

Syntax

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

Return Values

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()` Macro” on page 82, “`CALLOC()` Macro” on page 67, “`REALLOC()` Macro” on page 118, “`STRDUP()` Macro” on page 127, “`PERM_MALLOC()` Macro” on page 109

N

`net_flush()` Function

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

Syntax

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

Return Values

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

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()` Function” on page 91, “`net_sendfile()` Function” on page 90

net_ip2host() Function

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

Note – This function works only if the DNS directive is enabled in the magnus.conf file.

Syntax

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

Return Values

A new string containing the fully qualified domain name if the transformation is accomplished, or NULL if the transformation is 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. Though this verification requires an extra query, you should use it when checking the access control.

net_read() Function

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);
```

Return Values

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. The purpose of timeout is to limit the amount of time devoted to waiting until some data arrives.

See Also

"net_write() Function" on page 91

net_sendfile() Function

The net_sendfile function sends the contents of a specified file to a specified a socket. Either the whole file or a fraction might be sent, and the contents of the file might 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);
```

Return Values

A positive number indicating 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()` Function" on page 88

`net_write()` Function

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);
```

Return Values

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

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()` Function" on page 89

`netbuf_buf2sd()` Function

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);
```

Return Values

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()` Function" on page 93, "`netbuf_getc()` Function" on page 94, "`netbuf_getbytes()` Function" on page 93, "`netbuf_grab()` Function" on page 95, "`netbuf_open()` Function" on page 95

`netbuf_close()` Function

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.

Never close the `netbuf` parameter in a `session` structure.

Syntax

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

Return Values

`void`

Parameters

`netbuf *buf` is the buffer to close.

See Also

"`netbuf_buf2sd()` Function" on page 92, "`netbuf_getc()` Function" on page 94, "`netbuf_getbytes()` Function" on page 93, "`netbuf_grab()` Function" on page 95, "`netbuf_open()` Function" on page 95

`netbuf_getbytes()` Function

The `netbuf_getbytes` function reads bytes from a network buffer into a caller-supplied buffer. If the network buffer is empty, the function waits to receive data from the network buffer's socket until either at least one byte is available from the socket or the network buffer's timeout has elapsed.

Syntax

```
int netbuf_getbytes(netbuf *buf, char *buffer, int sz);
```

Return Values

The number of bytes placed into the buffer between 1 and `sz` if the operation is successful, the constant `NETBUF_EOF` on end of file, or the constant `NETBUF_ERROR` if an error occurred.

Parameters

`netbuf *buf` is the buffer from which to retrieve bytes.

`char *buffer` is the caller-supplied buffer that receives the bytes.

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

Example

```
int cl = 0;

* Read the entire request body */
for (;;) {
     char mybuf[1024];
     int rv;

     rv = netbuf_getbytes(sn->inbuf, mybuf, sizeof(mybuf));
     if (rv == NETBUF_EOF) {
         log_error(LOG_INFORM, "mysaf", sn, rq,
                   "Received %d byte(s)",
                   cl);
         break;
     }
     if (rv == NETBUF_ERROR) {
         log_error(LOG_FAILURE, "mysaf", sn, rq,
                   "Error reading request body (%s)",
                   cl, system_errmsg());
         break;     }

     cl += rv;
}
```

See Also

"netbuf_buf2sd() Function" on page 92, "netbuf_close() Function" on page 93, "netbuf_getc() Function" on page 94, "netbuf_grab() Function" on page 95, "netbuf_open() Function" on page 95

netbuf_getc() Function

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

Note – Because the constant IO_EOF has a value of 0, netbuf_getc cannot be used to read data that might contain a null character. To read binary data, use "netbuf_getbytes() Function" on page 93 or "netbuf_grab() Function" on page 95.

Syntax

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

Return Values

The integer representing the character if a character is retrieved, or the constant `IO_EOF` or `IO_ERROR` for end of file or an error.

Parameters

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

See Also

"`netbuf_buf2sd()` Function" on page 92, "`netbuf_close()` Function" on page 93, "`netbuf_getbytes()` Function" on page 93, "`netbuf_grab()` Function" on page 95, "`netbuf_open()` Function" on page 95

`netbuf_grab()` Function

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);
```

Return Values

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

Parameters

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

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

See Also

"`netbuf_buf2sd()` Function" on page 92, "`netbuf_close()` Function" on page 93, "`netbuf_getbytes()` Function" on page 93, "`netbuf_getc()` Function" on page 94, "`netbuf_open()` Function" on page 95

`netbuf_open()` Function

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);
```

Return Values

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

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()` Function" on page 92, "`netbuf_close()` Function" on page 93, "`netbuf_getc()` Function" on page 94, "`netbuf_getbytes()` Function" on page 93, "`netbuf_grab()` Function" on page 95

`nsapi_module_init()` Function

Defines the `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.

Use the `nsapi_module_init` function to register SAFs with `func_insert`, create filters with `filter_create`, register virtual server initialization/destruction callbacks with `vs_register_cb`, and perform other initialization tasks.

Syntax

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

Return Values

`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.

See Also

"filter_create() Function" on page 77, "func_insert() Function" on page 84, "vs_register_cb() Function" on page 161

NSAPI_RUNTIME_VERSION() Macro

The NSAPI_RUNTIME_VERSION macro defines the NSAPI version available at runtime. This version 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 iPlanet™ Web Server 6.0, Netscape Enterprise Server 6.0, Sun ONE Web Server 6.1, and Sun Java System Web Server 7.0 Update 2. 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.

Do not attempt to set the value of the NSAPI_RUNTIME_VERSION macro directly. Instead, use 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

"filter_create() Function" on page 77, "func_insert() Function" on page 84, "vs_register_cb() Function" on page 161

NSAPI_VERSION() Macro

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 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.

Do not attempt to set the value of the NSAPI_VERSION macro directly. Instead, use 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() Macro" on page 97, "USE_NSAPI_VERSION() Macro" on page 141

P

param_create() Function

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);
```

Return Values

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() Function” on page 99, “pblock_pinsert() Function” on page 106, “pblock_remove() Function” on page 106

param_free() Function

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);
```

Return Values

1 if the parameter is freed or 0 if the parameter is 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() Function” on page 98, “pblock_pinsert() Function” on page 106, “pblock_remove() Function” on page 106

pblock_copy() Function

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);
```

Return Values

`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()` Function" on page 100, "`pblock_dup()` Function" on page 101, "`pblock_free()` Function" on page 102, "`pblock_find()` Function" on page 101, "`pblock_findval()` Function" on page 102, "`pblock_remove()` Function" on page 106, "`pblock_nvinsert()` Function" on page 104

`pblock_create()` Function

The `pblock_create` function creates a new `pblock`. The `pblock` maintains an internal hash table for fast name-value pair lookups. Because the `pblock` is allocated from the request's memory pool, it should not be shared between threads.

Syntax

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

Return Values

A pointer to a newly allocated `pblock`.

Parameters

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

See Also

"`pblock_copy()` Function" on page 99, "`pblock_dup()` Function" on page 101, "`pblock_find()` Function" on page 101, "`pblock_findval()` Function" on page 102, "`pblock_free()` Function" on page 102, "`pblock_nvinsert()` Function" on page 104, "`pblock_remove()` Function" on page 106

`pblock_dup()` Function

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

Syntax

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

Return Values

A pointer to a newly allocated `pblock`.

Parameters

`pblock *src` is the source `pblock`.

See Also

"`pblock_create()` Function" on page 100, "`pblock_find()` Function" on page 101, "`pblock_findval()` Function" on page 102, "`pblock_free()` Function" on page 102, "`pblock_nvinsert()` Function" on page 104, "`pblock_remove()` Function" on page 106

`pblock_find()` Function

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

Note – Parameter names are case sensitive. By convention, lowercase names are used for parameters that correspond to HTTP header fields.

Syntax

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

Return Values

A pointer to the `pb_param` structure if found, or NULL if the name is 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() Function" on page 99, "pblock_dup() Function" on page 101, "pblock_findval() Function" on page 102, "pblock_free() Function" on page 102, "pblock_nvinsert() Function" on page 104, "pblock_remove() Function" on page 106

pblock_findval() Function

The pblock_findval function finds the value associated with a specified name in a pblock. If you want to find 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 the pointer, do a STRDUP and modify the copy.

Note – Parameter names are case-sensitive. By convention, lowercase names are used for parameters that correspond to HTTP header fields.

Syntax

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

Return Values

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

Parameters

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

pblock *pb is the pblock to be searched.

See Also

"pblock_create() Function" on page 100, "pblock_copy() Function" on page 99, "pblock_find() Function" on page 101, "pblock_free() Function" on page 102, "pblock_nvinsert() Function" on page 104, "pblock_remove() Function" on page 106, "request_header() Function" on page 120

pblock_free() Function

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);
```

Return Values

`void`

Parameters

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

See Also

"`pblock_copy()` Function" on page 99, "`pblock_create()` Function" on page 100, "`pblock_dup()` Function" on page 101, "`pblock_find()` Function" on page 101, "`pblock_findval()` Function" on page 102, "`pblock_nvinsert()` Function" on page 104, "`pblock_remove()` Function" on page 106

`pblock_nninsert()` Function

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.

Note – Parameter names are case sensitive. By convention, lowercase names are used for parameters that correspond to HTTP header fields.

Syntax

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

Return Values

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 you want to assign a non-numerical value, then use the function `pblock_nvinsert` to create the parameter.

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

See Also

"`pblock_copy()` Function" on page 99, "`pblock_create()` Function" on page 100, "`pblock_find()` Function" on page 101, "`pblock_free()` Function" on page 102, "`pblock_nvinsert()` Function" on page 104, "`pblock_remove()` Function" on page 106, "`pblock_str2pblock()` Function" on page 107

`pblock_nvinsert()` Function

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.

Note – Parameter names are case sensitive. By convention, lowercase names are used for parameters that correspond to HTTP header fields.

Syntax

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

Return Values

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()` Function" on page 99, "`pblock_create()` Function" on page 100, "`pblock_find()` Function" on page 101, "`pblock_free()` Function" on page 102, "`pblock_nninsert()` Function" on page 103, "`pblock_remove()` Function" on page 106, "`pblock_str2pblock()` Function" on page 107

`pblock_pb2env()` Function

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);
```

Return Values

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()` Function" on page 99, "`pblock_create()` Function" on page 100, "`pblock_find()` Function" on page 101, "`pblock_free()` Function" on page 102, "`pblock_nvinsert()` Function" on page 104, "`pblock_remove()` Function" on page 106, "`pblock_str2pblock()` Function" on page 107

`pblock_pblock2str()` Function

The `pblock_pblock2str` function copies all parameters of a specified `pblock` into a specified string. The function allocates additional non-heap 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);
```

Return Values

The new version of the `str` parameter. If `str` is NULL, this string is a new string; otherwise, this string is a reallocated string. In either case, this string 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. This 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()` Function" on page 99, "`pblock_create()` Function" on page 100, "`pblock_find()` Function" on page 101, "`pblock_free()` Function" on page 102, "`pblock_nvinsert()` Function" on page 104, "`pblock_remove()` Function" on page 106, "`pblock_str2pblock()` Function" on page 107

`pblock_pinsert()` Function

The function `pblock_pinsert` inserts a `pb_param` structure into a `pblock`.

Note – Parameter names are case sensitive. By convention, lowercase names are used for parameters that correspond to HTTP header fields.

Syntax

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

Return Values

```
void
```

Parameters

`pb_param *pp` is the `pb_param` structure to insert.

`pblock *pb` is the `pblock`.

See Also

"`pblock_copy()` Function" on page 99, "`pblock_create()` Function" on page 100, "`pblock_find()` Function" on page 101, "`pblock_free()` Function" on page 102, "`pblock_nvinsert()` Function" on page 104, "`pblock_remove()` Function" on page 106, "`pblock_str2pblock()` Function" on page 107

`pblock_remove()` Function

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

Syntax

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

Return Values

A pointer to the named `pb_param` structure if it is found, or NULL if the named `pb_param` is 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()` Function" on page 99, "`pblock_create()` Function" on page 100, "`pblock_find()` Function" on page 101, "`pblock_free()` Function" on page 102, "`pblock_nvinsert()` Function" on page 104, "`param_create()` Function" on page 98, "`param_free()` Function" on page 99

`pblock_str2pblock()` Function

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);
```

Return Values

The number of parameter pairs added to the `pblock`, if any, or `-1` if an error occurs.

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=`), the function 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()` Function" on page 99, "`pblock_create()` Function" on page 100, "`pblock_find()` Function" on page 101, "`pblock_free()` Function" on page 102, "`pblock_nvinsert()` Function" on page 104, "`pblock_remove()` Function" on page 106, "`pblock_pblock2str()` Function" on page 105

`PERM_CALLOC()` Macro

The `PERM_CALLOC` macro is a platform-independent substitute for the C library routine `calloc`. This macro allocates `size` bytes of memory and initializes the memory to zeros. The memory persists after processing the current request has been completed. The memory should be explicitly freed by a call to `PERM_FREE`.

Syntax

```
void *PERM_CALLOC(int size)
```

Return Values

A void pointer to a block of memory.

Parameters

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

Example

```
char **name;
name = (char **) PERM_CALLOC(100 * sizeof(char *));
```

See Also

"`CALLOC()` Macro" on page 67, "`PERM_FREE()` Macro" on page 108, "`PERM_STRDUP()` Macro" on page 111, "`PERM_MALLOC()` Macro" on page 109, "`PERM_REALLOC()` Macro" on page 110

`PERM_FREE()` Macro

The `PERM_FREE` macro is a platform-independent substitute for the C library routine `free`. This macro deallocates the persistent space previously allocated by `PERM_MALLOC`, `PERM_CALLOC`, or `PERM_STRDUP`.

Note – Calling PERM_FREE for a block that was allocated with MALLOC, CALLOC, or STRTUP will not work.

Syntax

```
PERM_FREE(void *ptr);
```

Return Values

void

Parameters

void *ptr is a (void *) pointer to a block of memory. If the pointer is not the 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() Macro" on page 82, "PERM_MALLOC() Macro" on page 109, "PERM_CALLOC() Macro" on page 108, "PERM_REALLOC() Macro" on page 110, "PERM_STRDUP() Macro" on page 111

PERM_MALLOC() Macro

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.

Syntax

```
void *PERM_MALLOC(int size)
```

Return Values

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

"MALLOC() Macro" on page 87, "PERM_FREE() Macro" on page 108, "PERM_STRDUP() Macro" on page 111, "PERM_CALLOC() Macro" on page 108, "PERM_REALLOC() Macro" on page 110

PERM_REALLOC() Macro

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 PERM_MALLOC, PERM_CALLOC, or PERM_STRDUP. The content 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.

Caution – Calling PERM_REALLOC for a block that was allocated with MALLOC, CALLOC, or STRDUP does not work.

Syntax

```
void *PERM_REALLOC(vod *ptr, int size)
```

Return Values

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(name, 512);
```

See Also

"REALLOC() Macro" on page 118, "PERM_CALLOC() Macro" on page 108, "PERM_MALLOC() Macro" on page 109, "PERM_FREE() Macro" on page 108, "PERM_STRDUP() Macro" on page 111

PERM_STRDUP() Macro

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 built-in pool-init SAF PERM_STRDUP and STRDUP both obtain their memory from the system heap.

The PERM_STRDUP routine is functionally equivalent to the following code:

```
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);
```

Return Values

A pointer to the new string.

Parameters

char *ptr is a pointer to a string.

See Also

“PERM_MALLOC() Macro” on page 109, “PERM_FREE() Macro” on page 108, “PERM_CALLOC() Macro” on page 108, “PERM_REALLOC() Macro” on page 110, “MALLOC() Macro” on page 87, “FREE() Macro” on page 82, “STRDUP() Macro” on page 127, “CALLOC() Macro” on page 67, “REALLOC() Macro” on page 118

prepare_nsapi_thread() Function

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);
```

Return Values

void

Parameters

`Request *rq` is the request.

`Session *sn` is the session.

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

See Also

"`protocol_start_response()` Function" on page 113

`protocol_dump822()` Function

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);
```

Return Values

A pointer to the buffer, which will be reallocated if necessary.

The function also modifies `*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 inserted.

`int tsz` is the size of the buffer.

See Also

"`protocol_start_response()` Function" on page 113, "`protocol_status()` Function" on page 114

`protocol_set_finfo()` Function

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);
```

Return Values

The constant `REQ_PROCEED` if the request can proceed normally, or the constant `REQ_ABORTED` if the function should treat the request normally but not send any output to the client.

Parameters

`Session *sn` is the session.

`Request *rq` is the request.

The `Session` and `Request` parameters are the same parameters as the ones 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()` Function" on page 113, "`protocol_status()` Function" on page 114

`protocol_start_response()` Function

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 or higher, the function sends a status line followed by the response headers. Because of buffering, the status line and response headers might not be sent immediately. To flush the status line and response headers, use the `net_flush` function. Use this function to set up HTTP and prepare the client and server to receive the body or data of the response.

Note – If you do not want the server to send the status line and response headers, set `rq->senthdrs = 1` before calling `protocol_start_response` or sending any data to the client.

Syntax

```
int protocol_start_response(Session *sn, Request *rq);
```

Return Values

The constant REQ_PROCEED if the operation succeeds, in which case you should send the data you were preparing to send.

The constant REQ_NOACTION if the operation succeeds but the request method is HEAD, in which case no data should be sent to the client.

The constant REQ_ABORTED if the operation fails.

Parameters

Session *sn is the session.

Request *rq is the request.

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

Example

```
/* REQ_NOACTION 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() Function" on page 114, "net_flush() Function" on page 88

protocol_status() Function

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 function finds none, 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 or returning REQ_ABORTED.

For the complete list of valid status code constants, refer to the nsapi.h file.

Syntax

```
void protocol_status(Session *sn, Request *rq, int n, char *r);
```

Return Values

void

Parameters

`Session *sn` is the session.

`Request *rq` is the request.

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

`int n` is one of the HTTP status code constants.

`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()` Function" on page 113

`protocol_uri2url()` Function

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)`. Also 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);
```

Return Values

A new string containing the URL.

Parameters

`char *prefix` is the prefix.

`char *suffix` is the suffix.

See Also

"`pblock_nvinsert()` Function" on page 104, "`protocol_start_response()` Function" on page 113, "`protocol_status()` Function" on page 114, "`protocol_uri2url_dynamic()` Function" on page 116

`protocol_uri2url_dynamic()` Function

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 it constructs refers to the host that the client specified.

Syntax

```
char *protocol_uri2url(char *prefix, char *suffix, Session *sn, Request *rq);
```

Return Values

A new string containing the URL.

Parameters

`char *prefix` is the prefix.

`char *suffix` is the suffix.

`Session *sn` is the Session.

`Request *rq` is the Request.

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

See Also

"`protocol_start_response()` Function" on page 113, "`protocol_status()` Function" on page 114

R

read() Function

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 can be obtained by calling the net_read() function, as shown in the example below.

Syntax

```
int read(FilterLayer *layer, void *buf, int amount, int timeout);
```

Return Values

The number of bytes placed in buf on success. 0 if no data is available, or a negative value if an error occurs.

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 the read operation to return. The purpose of timeout is to limit the amount of time devoted to waiting until some data arrives, not to return because not enough bytes were read in the given time.

Example

```
int myfilter_read(FilterLayer *layer, void *buf, int amount, int timeout)
{
    return net_read(layer->lower, buf, amount, timeout);
}
```

See Also

“net_read() Function” on page 89, “filter_create() Function” on page 77

REALLOC() Macro

The 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.

Caution – 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);
```

Return Values

A pointer to the new space if the request is satisfied.

Parameters

void *ptr is a (void *) pointer to a block of memory. If the pointer is not the one 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(name, 512);
```

See Also

"CALLOC() Macro" on page 67, "MALLOC() Macro" on page 87, "FREE() Macro" on page 82, "STRDUP() Macro" on page 127, "PERM_REALLOC() Macro" on page 110

remove() Function

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 function or remove-filter SAF.

Note – Waiting until the remove method is invoked might be too late to flush buffered data. For this reason, filters that buffer outgoing data should implement the flush filter method.

Syntax

```
void remove(FilterLayer *layer);
```

Return Values

void

Parameters

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

See Also

"flush() Function" on page 82, "filter_remove() Function" on page 81, "filter_create() Function" on page 77

request_get_vs() Function

The request_get_vs function finds the VirtualServer* to which a request is directed.

The returned VirtualServer* is valid only for the current request. To retrieve a virtual server ID that is valid across requests, use vs_get_id().

Syntax

```
const VirtualServer* request_get_vs(Request* rq);
```

Return Values

The VirtualServer* to which the request is directed.

Parameters

Request *rq is the request for which the VirtualServer* is returned.

See Also

"vs_get_id() Function" on page 160

request_header() Function

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);
```

Return Values

A result code, REQ_PROCEED if the header was found, REQ_ABORTED if the header was not found, or REQ_EXIT if there was an error 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 no address is found, the function stores a NULL.

Session *sn is the session.

Request *rq is the request.

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

See Also

request_create, request_free

request_stat_path() Function

The request_stat_path function returns the file information structure for a specified path or, if no path is specified, the path entry in the vars pblock in the specified request structure. If the resulting file name points to a file that the server can read, request_stat_path returns a new file information structure. This structure contains information about the size of the file, its owner, when it was created, and when it was last modified.

Use request_stat_path to retrieve information on the file you are currently accessing instead of calling stat directly, because this function keeps track of previous calls for the same path and returns its cached information.

Syntax

```
struct stat *request_stat_path(char *path, Request *rq);
```

Return Values

Returns a pointer to the file information structure for the file named by the path parameter. Do not free this structure. Returns NULL if the file is not valid or the server cannot read it. In this case, it also leaves an error message describing the problem in rq->staterr.

Parameters

char *path is the string containing the name of the path. If the value of path is NULL, the function uses the path entry in the vars pblock in the request structure denoted by rq.

Request *rq is the request identifier for a Server Application Function call.

Example

```
fi = request_stat_path(path, rq);
```

See Also

request_create, request_free, "request_header() Function" on page 120

request_translate_uri() Function

The request_translate_uri function performs virtual-to-physical mapping on a specified URI during a specified session. Use this function to determine the file to be sent back if a given URI is accessed.

Syntax

```
char *request_translate_uri(char *uri, Session *sn);
```

Return Values

A path string if the function performed the mapping, or NULL if it could not perform the mapping.

Parameters

char *uri is the name of the URI.

Session *sn is the Session parameter that is passed into the SAF.

See Also

request_create, request_free, "request_header() Function" on page 120

S

sendfile() Function

The sendfile filter method is called when the contents of a file are to be sent. Filters that modify or consume outgoing data can choose to 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() calls to net_write() 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);
```

Return Values

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 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() Function" on page 90, "filter_create() Function" on page 77

session_dns() Function

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 that the client matches its claimed identity. The session_dns function does not perform this verification.

Note – This function works only if the `DNS` directive is enabled in the `magnus.conf` file. For more information, see Appendix B, "Alphabetical List of NSAPI Functions and Macros."

Syntax

```
char *session_dns(Session *sn);
```

Return Values

A string containing the host name, or NULL if the DNS name cannot be found for the IP address.

Parameters

`Session *sn` is the session.

The `Session` is the same parameter as the one passed to your SAF.

`session_maxdns()` Function

The `session_maxdns` function resolves the IP address of the client associated with a specified session into its DNS name. This function returns a newly allocated string. You can use `session_maxdns` to change the numeric IP address into something more readable.

Note – This function works only if the `DNS` directive is enabled in the `magnus.conf` file. For more information, see Appendix B, "Alphabetical List of NSAPI Functions and Macros."

Syntax

```
char *session_maxdns(Session *sn);
```

Return Values

A string containing the host name, or NULL if the DNS name cannot be found for the IP address.

Parameters

`Session *sn` is the `Session`.

The `Session` is the same parameter as the one passed to your SAF.

shexp_casecmp() Function

The shexp_casecmp function validates a specified shell expression and compares it with a specified string. This function returns one of three possible values representing match, no match, and invalid comparison. The comparison, in contrast to the comparison made by the shexp_cmp function is not case sensitive.

Use this function if you have a shell expression like *.netscape.com and make sure that a string matches it, such as foo.netscape.com.

Syntax

```
int shexp_casecmp(char *str, char *exp);
```

Return Values

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() Function" on page 124, "shexp_match() Function" on page 125, "shexp_valid() Function" on page 126

shexp_cmp() Function

The shexp_cmp function validates a specified shell expression and compares it with a specified string. This returns one of three possible values representing match, no match, and invalid comparison. The comparison in contrast to the comparison made by the shexp_casecmp function is case sensitive.

Use this function for a shell expression like *.netscape.com and make sure that a string matches it, such as foo.netscape.com.

Syntax

```
int shexp_cmp(char *str, char *exp);
```

Return Values

`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()` Function" on page 124, "`shexp_match()` Function" on page 125, "`shexp_valid()` Function" on page 126

`shexp_match()` Function

The `shexp_match` function compares a specified pre-validated shell expression against a specified string. This function returns one of three possible values representing match, no match, and invalid comparison. The comparison in contrast to the contrast made by the `shexp_casecmp` function is case sensitive.

The `shexp_match` function does not perform validation of the shell expression. To perform validation, use `shexp_valid()`.

Use this function for a shell expression such as `*.netscape.com`, and make sure that a string matches it, such as `foo.netscape.com`.

Syntax

```
int shexp_match(char *str, char *exp);
```

Return Values

`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()` Function" on page 124, "`shexp_cmp()` Function" on page 124, "`shexp_valid()` Function" on page 126

`shexp_valid()` Function

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);
```

Return Values

The constant `NON_SXP` if `exp` is a standard string.

The constant `INVALID_SXP` if `exp` is an invalid shell expression.

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()` Function" on page 124, "`shexp_match()` Function" on page 125, "`shexp_cmp()` Function" on page 124

STRDUP() Macro

The STRDUP macro is a platform-independent substitute for the C library routine strdup. This macro creates a new copy of a string in the request's memory pool. The memory can be explicitly freed by a call to FREE. If the memory is not explicitly freed, it is automatically freed after processing the current request. If pooled memory has been disabled in the configuration file with the built-in pool-init SAF, PERM_STRDUP and STRDUP both obtain their memory from the system heap. However, because the memory allocated by STRDUP is automatically freed, do not share this memory between threads.

The STRDUP routine is functionally equivalent to:

```
newstr = (char *) MALLOC(strlen(str) + 1);
strcpy(newstr, str);
```

Syntax

```
char *STRDUP(char *ptr);
```

Return Values

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

"CALLOC() Macro" on page 67,"MALLOC() Macro" on page 87, "FREE() Macro" on page 82, "REALLOC() Macro" on page 118, "PERM_STRDUP() Macro" on page 111

system_errmsg() Function

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);
```

Return Values

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()` Function" on page 129, "`system_fopenRW()` Function" on page 130, "`system_fopenWA()` Function" on page 130, "`system_lseek()` Function" on page 134, "`system_fread()` Function" on page 131, "`system_fwrite()` Function" on page 132, "`system_fwrite_atomic()` Function" on page 132, "`system_flock()` Function" on page 129, "`system_ulock()` Function" on page 136, "`system_fclose()` Function" on page 128

`system_fclose()` Function

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);
```

Return Values

`0` if the close succeeds, or the constant `IO_ERROR` if the close fails.

Parameters

`SYS_FILE fd` is the platform-independent file descriptor.

Example

```
SYS_FILE logfd;
system_fclose(logfd);
```

See Also

"`system_errmsg()` Function" on page 127, "`system_fopenRO()` Function" on page 129, "`system_fopenRW()` Function" on page 130, "`system_fopenWA()` Function" on page 130, "`system_lseek()` Function" on page 134, "`system_fread()` Function" on page 131, "`system_fwrite()` Function" on page 132, "`system_fwrite_atomic()` Function" on page 132, "`system_flock()` Function" on page 129, "`system_ulock()` Function" on page 136

system_flock() Function

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);
```

Return Values

The constant IO_OKAY if the lock succeeds, or the constant IO_ERROR if the lock fails.

Parameters

SYS_FILE fd is the platform-independent file descriptor.

See Also

"system_errmsg() Function" on page 127, "system_fopenRO() Function" on page 129, "system_fopenRW() Function" on page 130, "system_fopenWA() Function" on page 130, "system_lseek() Function" on page 134, "system_fread() Function" on page 131, "system_fwrite() Function" on page 132, "system_fwrite_atomic() Function" on page 132, "system_ulock() Function" on page 136, "system_fclose() Function" on page 128

system_fopenRO() Function

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);
```

Return Values

The system-independent file descriptor (SYS_FILE) if the open succeeds, or 0 if the open fails.

Parameters

char *path is the file name.

See Also

"`system_errmsg()` Function" on page 127, "`system_fopenRW()` Function" on page 130, "`system_fopenWA()` Function" on page 130, "`system_lseek()` Function" on page 134, "`system_fread()` Function" on page 131, "`system_fwrite()` Function" on page 132, "`system_fwrite_atomic()` Function" on page 132, "`system_flock()` Function" on page 129, "`system_ulock()` Function" on page 136, "`system_fclose()` Function" on page 128

`system_fopenRW()` Function

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 can be read and written by your program.

Syntax

```
SYS_FILE system_fopenRW(char *path);
```

Return Values

The system-independent file descriptor (`SYS_FILE`) if the open succeeds, or `0` if the open fails.

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()` Function" on page 127, "`system_fopenRO()` Function" on page 129, "`system_fopenWA()` Function" on page 130, "`system_lseek()` Function" on page 134, "`system_fread()` Function" on page 131, "`system_fwrite()` Function" on page 132, "`system_fwrite_atomic()` Function" on page 132, "`system_flock()` Function" on page 129, "`system_ulock()` Function" on page 136, "`system_fclose()` Function" on page 128

`system_fopenWA()` Function

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);
```

Return Values

The system-independent file descriptor (`SYS_FILE`) if the open succeeds, or `0` if the open fails.

Parameters

`char *path` is the file name.

See Also

"`system_errmsg()` Function" on page 127, "`system_fopenRO()` Function" on page 129, "`system_fopenRW()` Function" on page 130, "`system_lseek()` Function" on page 134, "`system_fread()` Function" on page 131, "`system_fwrite()` Function" on page 132, "`system_fwrite_atomic()` Function" on page 132, "`system_flock()` Function" on page 129, "`system_ulock()` Function" on page 136, "`system_fclose()` Function" on page 128

`system_fread()` Function

The `system_fread` function reads a specified number of bytes from a specified file into a specified buffer. This function 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);
```

Return Values

The number of bytes read, which might be less than the requested size if an error occurs, or if 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()` Function" on page 127, "`system_fopenRO()` Function" on page 129, "`system_fopenRW()` Function" on page 130, "`system_fopenWA()` Function" on page 130, "`system_lseek()` Function" on page 134, "`system_fwrite()` Function" on page 132, "`system_fwrite_atomic()` Function" on page 132, "`system_flock()` Function" on page 129, "`system_ulock()` Function" on page 136, "`system_fclose()` Function" on page 128

`system_fwrite()` Function

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);
```

Return Values

The constant `IO_OKAY` if the write succeeds, or the constant `IO_ERROR` if the write fails.

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()` Function" on page 127, "`system_fopenRO()` Function" on page 129, "`system_fopenRW()` Function" on page 130, "`system_fopenWA()` Function" on page 130, "`system_lseek()` Function" on page 134, "`system_fread()` Function" on page 131, "`system_fwrite()` Function" on page 132, "`system_fwrite_atomic()` Function" on page 132, "`system_flock()` Function" on page 129, "`system_ulock()` Function" on page 136, "`system_fclose()` Function" on page 128

`system_fwrite_atomic()` Function

The `system_fwrite_atomic` function writes a specified number of bytes from a specified buffer into a specified file. This function also locks the file prior to performing the write, and then unlocks it when done, thereby avoiding 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);
```

Return Values

The constant `IO_OKAY` if the write/lock succeeds, or the constant `IO_ERROR` if the write/lock fails.

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()` Function" on page 127, "`system_fopenRO()` Function" on page 129, "`system_fopenRW()` Function" on page 130, "`system_fopenWA()` Function" on page 130, "`system_lseek()` Function" on page 134, "`system_fread()` Function" on page 131, "`system_fwrite()` Function" on page 132, "`system_flock()` Function" on page 129, "`system_ulock()` Function" on page 136, "`system_fclose()` Function" on page 128

`system_gmtime()` Function

The `system_gmtime` function is a thread-safe version of the standard `gmtime` function. This function returns the current time adjusted to Greenwich Mean Time.

Syntax

```
struct tm *system_gmtime(const time_t *tp, const struct tm *res);
```

Return Values

A pointer to a calendar time (`tm`) structure containing the GMT time. Depending on your system, the pointer might point to the data item represented by the second parameter, or the pointer might 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()` Function" on page 134, "`util_strftime()` Function" on page 152

`system_localtime()` Function

The `system_localtime` function is a thread-safe version of the standard `localtime` function. This function returns the current time in the local time zone.

Syntax

```
struct tm *system_localtime(const time_t *tp, const struct tm *res);
```

Return Values

A pointer to a calendar time (`tm`) structure containing the local time. Depending on your system, the pointer might point to the data item represented by the second parameter, or the pointer might 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()` Function" on page 133, "`util_strftime()` Function" on page 152

`system_lseek()` Function

The `system_lseek` function sets the file position of a file. This position affects where data from `system_fread` or `system_fwrite` is read or written.

Syntax

```
int system_lseek(SYS_FILE fd, int offset, int whence);
```

Return Values

The offset, in bytes, of the new position from the beginning of the file if the operation succeeds, or -1 if the operation fails.

Parameters

SYS_FILE fd is the platform-independent file descriptor.

int offset is a number of bytes relative to whence. This value 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() Function" on page 127, "system_fopenRO() Function" on page 129, "system_fopenRW() Function" on page 130, "system_fopenWA() Function" on page 130, "system_fread() Function" on page 131, "system_fwrite() Function" on page 132, "system_fwrite_atomic() Function" on page 132, "system_flock() Function" on page 129, "system_ulock() Function" on page 136, "system_fclose() Function" on page 128

system_rename() Function

The system_rename function renames a file. This function does not work on directories if the old and new directories are on different file systems.

Syntax

```
int system_rename(char *old, char *new);
```

Return Values

0 if the operation succeeds, or -1 if the operation fails.

Parameters

char *old is the old name of the file.

char *new is the new name for the file.

`system_ulock()` Function

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()` Function" on page 129.

Syntax

```
int system_ulock(SYS_FILE fd);
```

Return Values

The constant `IO_OKAY` if the operation succeeds, or the constant `IO_ERROR` if the operation fails.

Parameters

`SYS_FILE fd` is the platform-independent file descriptor.

See Also

"`system_errmsg()` Function" on page 127, "`system_fopenRO()` Function" on page 129, "`system_fopenRW()` Function" on page 130, "`system_fopenWA()` Function" on page 130, "`system_fread()` Function" on page 131, "`system_fwrite()` Function" on page 132, "`system_fwrite_atomic()` Function" on page 132, "`system_flock()` Function" on page 129, "`system_fclose()` Function" on page 128

`system_unix2local()` Function

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 such as one containing 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 has no effect, but can be called for portability.

Syntax

```
char *system_unix2local(char *path, char *lp);
```

Return Values

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`, which must contain enough space to hold the local path name.

See Also

"`system_fclose()` Function" on page 128, "`system_flock()` Function" on page 129, "`system_fopenRO()` Function" on page 129, "`system_fopenRW()` Function" on page 130, "`system_fopenWA()` Function" on page 130, "`system_fwrite()` Function" on page 132

`systhread_attach()` Function

The `systhread_attach` function makes an existing thread into a platform-independent thread.

Syntax

```
SYS_THREAD systhread_attach(void);
```

Return Values

A `SYS_THREAD` pointer to the platform-independent thread.

Parameters

None

See Also

"`systhread_current()` Function" on page 137, "`systhread_getdata()` Function" on page 138, "`systhread_newkey()` Function" on page 138, "`systhread_setdata()` Function" on page 139, "`systhread_sleep()` Function" on page 139, "`systhread_start()` Function" on page 140, "`systhread_timerset()` Function" on page 140

`systhread_current()` Function

The `systhread_current` function returns a pointer to the current thread.

Syntax

```
SYS_THREAD systhread_current(void);
```

Return Values

A `SYS_THREAD` pointer to the current thread.

Parameters

None

See Also

"`systhread_getdata()` Function" on page 138, "`systhread_newkey()` Function" on page 138, "`systhread_setdata()` Function" on page 139, "`systhread_sleep()` Function" on page 139, "`systhread_start()` Function" on page 140, "`systhread_timerset()` Function" on page 140

`systhread_getdata()` Function

The `systhread_getdata` function gets data that is associated with a specified key in the current thread.

Syntax

```
void *systhread_getdata(int key);
```

Return Values

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. Return Values NULL if the call does 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()` Function" on page 137, "`systhread_newkey()` Function" on page 138, "`systhread_setdata()` Function" on page 139, "`systhread_sleep()` Function" on page 139, "`systhread_start()` Function" on page 140, "`systhread_timerset()` Function" on page 140

`systhread_newkey()` Function

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);
```

Return Values

An integer key.

Parameters

None

See Also

"`systhread_current()` Function" on page 137, "`systhread_getdata()` Function" on page 138, "`systhread_setdata()` Function" on page 139, "`systhread_sleep()` Function" on page 139, "`systhread_start()` Function" on page 140, "`systhread_timerset()` Function" on page 140

`systhread_setdata()` Function

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);
```

Return Values

`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()` Function" on page 137, "`systhread_getdata()` Function" on page 138, "`systhread_newkey()` Function" on page 138, "`systhread_sleep()` Function" on page 139, "`systhread_start()` Function" on page 140, "`systhread_timerset()` Function" on page 140

`systhread_sleep()` Function

The `systhread_sleep` function puts the calling thread to sleep for a given time.

Syntax

```
void systhread_sleep(int milliseconds);
```

Return Values

`void`

Parameters

`int milliseconds` is the number of milliseconds the thread is to sleep.

See Also

“`systhread_current()` Function” on page 137, “`systhread_getdata()` Function” on page 138, “`systhread_newkey()` Function” on page 138, “`systhread_setdata()` Function” on page 139, “`systhread_start()` Function” on page 140, “`systhread_timerset()` Function” on page 140

`systhread_start()` Function

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);
```

Return Values

A new `SYS_THREAD` pointer if the call succeeds, or the constant `SYS_THREAD_ERROR` if the call does 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()` Function” on page 137, “`systhread_getdata()` Function” on page 138, “`systhread_newkey()` Function” on page 138, “`systhread_setdata()` Function” on page 139, “`systhread_sleep()` Function” on page 139, “`systhread_timerset()` Function” on page 140

`systhread_timerset()` Function

The `systhread_timerset` function starts or resets the interrupt timer interval for a thread system.

Note – Because most systems do not allow the timer interval to be changed, this function should be considered a suggestion rather than a command.

Syntax

```
void systhread_timerset(int usec);
```

Return Values

`void`

Parameters

`int usec` is the time in microseconds

See Also

“`systhread_current()` Function” on page 137, “`systhread_getdata()` Function” on page 138, “`systhread_newkey()` Function” on page 138, “`systhread_setdata()` Function” on page 139, “`systhread_sleep()` Function” on page 139, “`systhread_start()` Function” on page 140

U

`USE_NSAPI_VERSION()` Macro

Plug-in developers can define the `USE_NSAPI_VERSION` macro before including the `nsapi.h` header file to request a particular version of NSAPI. The requested NSAPI version is encoded by multiplying the major version number by 100 and then adding the resulting value 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` as 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 6–2 NSAPI Versions Supported by Different Servers

Server Version	NSAPI Version
iPlanet Web Server 4.1	3.0
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 ONE Web Server 6.1	3.2
Sun Java System Web Proxy Server 4.0	3.3
Sun Java System Web Server 7.0 Update 2	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 3.2 or higher.

Syntax

```
int USE_NSAPI_VERSION
```

Example

The following code can be used when building a plug-in designed to work with iPlanet Web Server 4.1 and Sun Java System Web Server 7.0 Update 2:

```
#define USE_NSAPI_VERSION 300 /* We want NSAPI 3.0 (Web Server 4.1) */
#include "nsapi.h"
```

See Also

“`NSAPI_RUNTIME_VERSION()` Macro” on page 97, “`NSAPI_VERSION()` Macro” on page 97

`util_can_exec()` Function (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);
```

Return Values

`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()` Function" on page 144, "`util_getline()` Function" on page 146, "`util_hostname()` Function" on page 147

`util_chdir2path()` Function

The `util_chdir2path` function changes the current working directory. Because a server process can service multiple requests concurrently but has only a single current working directory, this function should not be used.

Syntax

```
int util_chdir2path(char *path);
```

Return Values

`0` if the directory change succeeds, or `-1` if the directory can not be changed.

Parameters

`char *path` is the name of a directory.

The parameter must be a writable string.

`util_cookie_find()` Function

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);
```

Return Values

If successful, this function returns a pointer to the NULL-terminated value of the cookie. Otherwise, this function 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_env_find()` Function

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);
```

Return Values

The value of the environment variable if the string 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()` Function" on page 145, "`util_env_str()` Function" on page 146, "`util_env_free()` Function" on page 145, "`util_env_create()` Function" on page 144

`util_env_create()` Function

The `util_env_create` function creates and allocates the environment specified by `env` and returns 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);
```

Return Values

A pointer to an environment.

Parameters

`char **env` is the 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()` Function" on page 145, "`util_env_str()` Function" on page 146, "`util_env_free()` Function" on page 145, "`util_env_find()` Function" on page 144

`util_env_free()` Function

The `util_env_free` function frees a specified environment. Use this function to deallocate an environment you created using the function `util_env_create`.

Syntax

```
void util_env_free(char **env);
```

Return Values

`void`

Parameters

`char **env` is the environment to be freed.

See Also

"`util_env_replace()` Function" on page 145, "`util_env_str()` Function" on page 146, "`util_env_create()` Function" on page 144, "`util_env_create()` Function" on page 144

`util_env_replace()` Function

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);
```

Return Values

`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()` Function" on page 146, "`util_env_free()` Function" on page 145, "`util_env_create()` Function" on page 144, "`util_env_create()` Function" on page 144

`util_env_str()` Function

The `util_env_str` function creates an environment entry and returns the entry. This function does not check for non-alphanumeric symbols in the name, for example, the equal sign "=". You can use this function to create a new environment entry.

Syntax

```
char *util_env_str(char *name, char *value);
```

Return Values

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()` Function" on page 145, "`util_env_free()` Function" on page 145, "`util_env_create()` Function" on page 144, "`util_env_create()` Function" on page 144

`util_getline()` Function

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);
```

Return Values

`0` if successful, `line` contains the string.

`1` if the end of file is reached, `line` contains the string.

`-1` if an error occurs, `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 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`.

`util_hostname()` Function

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 can reallocate or free this string. Use this function to determine the name of the system you are on.

Syntax

```
char *util_hostname(void);
```

Return Values

A string containing the name, if a fully qualified domain name is found. Otherwise, the function returns NULL.

Parameters

None

util_is_mozilla() Function

The util_is_mozilla function checks whether a specified user-agent header string is a Mozilla browser of at least a specified revision level. The function returns a 1 if the level matches, and 0 otherwise. This 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);
```

Return Values

1 if the user-agent is a Mozilla browser, or 0 if the user-agent is not a Mozilla browser.

Parameters

char *ua is the user-agent string from the request headers.

char *major is the major release number, found to the left of the decimal point.

char *minor is the minor release number, found to the right of the decimal point.

See Also

"util_is_url() Function" on page 148, "util_later_than() Function" on page 149

util_is_url() Function

The util_is_url function checks whether a string is a URL, returns 1 if the string is a URL and 0 otherwise. The string is a URL if it begins with alphabetic characters followed by a colon (:).

Syntax

```
int util_is_url(char *url);
```

Return Values

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() Function" on page 148, "util_later_than() Function" on page 149

`util_itoa()` Function

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);
```

Return Values

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()` Function

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);
```

Return Values

`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()` Function" on page 152

`util_sh_escape()` Function

The `util_sh_escape` function parses a specified string and places a backslash (`\`) in front of any shell-special characters, returning the resulting string. Use this function to ensure that strings from clients do not cause a shell to do anything unexpected.

The shell-special characters are the space plus the following characters:

```
&;‘’"|*?~<>^()[]{}$\#!
```

Syntax

```
char *util_sh_escape(char *s);
```

Return Values

A newly allocated string.

Parameters

`char *s` is the string to be parsed.

See Also

“`util_uri_escape()` Function” on page 154

`util_snprintf()` Function

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. This 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, ...);
```

Return Values

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() Function" on page 151, "util_vsnprintf() Function" on page 156, "util_vsprintf() Function" on page 157

util_sprintf() Function

The util_sprintf function formats a specified string, using a specified format, into a specified buffer, using the printf-style syntax without bounds checking. This function returns the number of characters in the formatted buffer.

Because util_sprintf does not 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, ...);
```

Return Values

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. The function 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() Function" on page 150, "util_vsnprintf() Function" on page 156, "util_vsprintf() Function" on page 157

util_strcasecmp() Function

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 the strings are identical.

The comparison is not case sensitive.

Syntax

```
int util_strcasecmp(const char *s1, const char *s2);
```

Return Values

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() Function" on page 153

util_strftime() Function

The util_strftime function translates a tm structure, which is a structure describing a system time, into a textual representation. util_strftime is a thread-safe version of the standard strftime function.

Syntax

```
int util_strftime(char *s, const char *format, const struct tm *t);
```

Return Values

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. This function does not perform bounds checking, so you must make sure that the buffer is large enough for the text of the date.

`const char *format` is a format string, similar to `printf` string in that it consists of text with certain `%x` substrings. You can 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. For more information on time formats, see the *Sun Java System Web Server 7.0 Update 3 Administrator's Configuration File Reference*.

`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()` Function" on page 134, "`system_gmtime()` Function" on page 133

`util_strncasecmp()` Function

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);
```

Return Values

`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()` Function" on page 152

`util_uri_escape()` Function

The `util_uri_escape` function converts any special characters in the URI into the URI format. This format is `%XX`, where `XX` is the hexadecimal equivalent of the ASCII character. This function 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);
```

Return Values

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()` Function" on page 154, "`util_uri_parse()` Function" on page 155, "`util_uri_unescape()` Function" on page 155

`util_uri_is_evil()` Function

The `util_uri_is_evil` function checks a specified URI for insecure path characters. Insecure path characters include `//`, `/./`, `/../` and`/.`, `/..` (also 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);
```

Return Values

`1` if the URI is insecure, or `0` if the URI is secure.

Parameters

`char *t` is the URI to be checked.

See Also

“`util_uri_parse()` Function” on page 155, “`util_uri_escape()` Function” on page 154

`util_uri_parse()` Function

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 bad sequences into valid ones. First, use the function `util_uri_is_evil` to determine whether the function has a bad sequence.

Syntax

```
void util_uri_parse(char *uri);
```

Return Values

`void`

Parameters

`char *uri` is the URI to be converted.

See Also

“`util_uri_is_evil()` Function” on page 154, “`util_uri_unescape()` Function” on page 155

`util_uri_unescape()` Function

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 does not work.

Syntax

```
void util_uri_unescape(char *uri);
```

Return Values

`void`

Parameters

`char *uri` is the URI to be converted.

See Also

“`util_uri_escape()` Function” on page 154“`util_uri_is_evil()` Function” on page 154, “`util_uri_parse()` Function” on page 155

`util_vsnprintf()` Function

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. This 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);
```

Return Values

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. This function 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_sprintf()` Function” on page 151, “`util_vsprintf()` Function” on page 157

`util_vsprintf()` Function

The `util_vsprintf` function formats a specified string, using a specified format, into a specified buffer using the `vprintf`-style syntax without bounds checking. This 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);
```

Return Values

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()` Function" on page 150, "`util_vsnprintf()` Function" on page 156

V

`vs_alloc_slot()` Function

The `vs_alloc_slot` function allocates a new slot for storing a pointer to data specific to a certain `VirtualServer*`. The returned slot number can be used in subsequent `vs_set_data` and `vs_get_data` calls. The returned slot number is valid for any `VirtualServer*`.

The value of the pointer, which may be returned by a call to `vs_set_data` , defaults to NULL for every `VirtualServer*`.

Syntax

```
int vs_alloc_slot(void);
```

Return Values

A slot number if the function succeeds, or `-1` if fails.

See Also

"vs_get_data() Function" on page 158, "vs_set_data() Function" on page 162

vs_get_data() Function

The vs_get_data function finds the value of a pointer to data for a given VirtualServer* and slot. The slot must be a slot number returned from vs_alloc_slot or vs_set_data.

Syntax

```
void* vs_get_data(const VirtualServer* vs, int slot);
```

Return Values

The value of the pointer previously stored using vs_set_data or NULL on failure.

Parameters

const VirtualServer* vs represents the virtual server to query the pointer for.

int slot is the slot number to retrieve the pointer from.

See Also

"vs_set_data() Function" on page 162, "vs_alloc_slot() Function" on page 157

vs_get_default_httpd_object() Function

The vs_get_default_httpd_object function obtains a pointer to the default (or root) httpd_object from the virtual server's httpd_objset in the configuration defined by the obj.conf file of the virtual server class. The default object is typically named default. Plug-ins may only modify the httpd_object at VSInitFunc time. See "vs_register_cb() Function" on page 161 for an explanation of VSInitFunc time.

Do not FREE the returned object.

Syntax

```
httpd_object* vs_get_default_httpd_object(VirtualServer* vs);
```

Return Values

A pointer the default httpd_object, or NULL on failure. Do not FREE this object.

Parameters

VirtualServer* vs represents the virtual server for which to find the default object.

See Also

"`vs_get_httpd_objset()` Function" on page 159, "`vs_register_cb()` Function" on page 161

`vs_get_doc_root()` Function

The `vs_get_doc_root` function finds the document root for a virtual server. The returned string is the full operating system path to the document root.

The caller should FREE the returned string when done with it.

Syntax

```
char* vs_get_doc_root(const VirtualServer* vs);
```

Return Values

A pointer to a string representing the full operating system path to the document root. The caller must FREE this string.

Parameters

`const VirtualServer*` vs represents the virtual server for which to find the document root.

`vs_get_httpd_objset()` Function

The `vs_get_httpd_objset` function obtains a pointer to the `httpd_objset`, the configuration defined by the `obj.conf` file of the virtual server class for a given virtual server. Plug-ins may only modify the `httpd_objset` at `VSInitFunc` time. See "`vs_register_cb()` Function" on page 161 for an explanation of `VSInitFunc` time.

Do not FREE the returned `objset`.

Syntax

```
httpd_objset* vs_get_httpd_objset(VirtualServer* vs);
```

Return Values

A pointer to the `httpd_objset`, or NULL on failure. Do not FREE this `objset`.

Parameters

`VirtualServer* vs` represents the virtual server for which the function ID to find the `objset`.

See Also

"`vs_get_default_httpd_object()` Function" on page 158, "`vs_register_cb()` Function" on page 161

vs_get_id() Function

The vs_get_id function finds the ID of a VirtualServer*.

The ID of a virtual server is a unique null-terminated string that remains constant across configurations. While IDs remain constant across configurations, the value of VirtualServer* pointers do not.

Do not FREE the virtual server ID string. If called during request processing, the string will remain valid for the duration of the current request. If called during VSInitFunc processing, the string will remain valid until after the corresponding VSDestroyFunc function has returned. For more information, see "vs_register_cb() Function" on page 161.

To retrieve a VirtualServer* that is valid only for the current request, use request_get_vs.

Syntax

```
const char* vs_get_id(const VirtualServer* vs);
```

Return Values

A pointer to a string representing the virtual server ID. Do not FREE this string.

Parameters

const VirtualServer* vs represents the virtual server of interest.

See Also

"vs_register_cb() Function" on page 161, "request_get_vs() Function" on page 119

vs_get_mime_type() Function

The vs_get_mime_type function determines the MIME type that would be returned in the content-type: header for the given URI.

The caller should FREE the returned string when done with it.

Syntax

```
char* vs_get_mime_type(const VirtualServer* vs, const char* uri);
```

Return Values

A pointer to a string representing the MIME type. The caller must FREE this string.

Parameters

`const VirtualServer* vs` represents the virtual server of interest.

`const char* uri` is the URI whose MIME type is of interest.

`vs_lookup_config_var()` Function

The `vs_lookup_config_var` function finds the value of a configuration variable for a given virtual server.

Do not FREE the returned string.

Syntax

```
const char* vs_lookup_config_var(const VirtualServer* vs, const char* name);
```

Return Values

A pointer to a string representing the value of variable name on success, or NULL if variable name was not found. Do not FREE this string.

Parameters

`const VirtualServer*` vs represents the virtual server of interest.

`const char* name` is the name of the configuration variable.

`vs_register_cb()` Function

The `vs_register_cb` function enables a plug-in to register functions that will receive notifications of virtual server initialization and destruction events. The `vs_register_cb` function is typically called from an `Init` SAF in `magnus.conf`.

When a new configuration is loaded, all registered `VSInitFunc` (virtual server initialization) callbacks are called for each of the virtual servers before any requests are served from the new configuration. `VSInitFunc` callbacks are called in the same order they were registered. The first callback registered is the first callback called.

When the last request has been served from an old configuration, all registered `VSDestroyFunc` (virtual server destruction) callbacks are called for each of the virtual servers before any virtual servers are destroyed. `VSDestroyFunc` callbacks are called in reverse order. The first callback registered is the last callback called.

Either `initfn` or `destroyfn` may be NULL if the caller is not interested in callbacks for initialization or destruction, respectively.

Syntax

```
int vs_register_cb(VSInitFunc* initfn, VSDestroyFunc* destroyfn);
```

Return Values

The constant `REQ_PROCEED` if the operation succeeds.

The constant `REQ_ABORTED` if the operation fails.

Parameters

`VSInitFunc* initfn` is a pointer to the function to call at virtual server initialization time, or NULL if the caller is not interested in virtual server initialization events.

`VSDestroyFunc* destroyfn` is a pointer to the function to call at virtual server destruction time, or NULL if the caller is not interested in virtual server destruction events.

`vs_set_data()` Function

The `vs_set_data` function sets the value of a pointer to data for a given virtual server and slot. The `*slot` must be `-1` or a slot number returned from `vs_alloc_slot`. If `*slot` is `-1`, `vs_set_data` calls `vs_alloc_slot` implicitly and returns the new slot number in `*slot`.

The stored pointer is maintained on a per-`VirtualServer*` basis, not a per-ID basis. Distinct `VirtualServer*`s from different configurations might exist simultaneously with the same virtual server IDs. However, because these configurations are distinct `VirtualServer*`s, each configuration has its own `VirtualServer*`-specific data. As a result, `vs_set_data` should generally not be called outside of `VSInitFunc` processing. See “`vs_register_cb()` Function” on page 161 for an explanation of `VSInitFunc` processing.

Syntax

```
void* vs_set_data(const VirtualServer* vs, int* slot, void* data);
```

Return Values

Data on success, or NULL on failure.

Parameters

`const VirtualServer* vs` represents the virtual server to set the pointer for.

`int* slot` is the slot number at which to store the pointer.

`void* data` is the pointer to store.

See Also

"vs_get_data() Function" on page 158, "vs_alloc_slot() Function" on page 157, "vs_register_cb() Function" on page 161

vs_translate_uri() Function

The vs_translate_uri function translates a URI as though the URI were part of a request for a specific virtual server. The returned string is the full operating system path.

The caller should FREE the returned string when done with it.

Syntax

```
char* vs_translate_uri(const VirtualServer* vs, const char* uri);
```

Return Values

A pointer to a string representing the full operating system path for the given URI. The caller must FREE this string.

Parameters

const VirtualServer* vs represents the virtual server for which to translate the URI.

const char* uri is the URI to translate to an operating system path.

W

write() Function

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 filter method.

Syntax

```
int write(FilterLayer *layer, const void *buf, int amount);
```

Return Values

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()` Function" on page 82, "`net_write()` Function" on page 91, "`writev()` Function" on page 164, "`filter_create()` Function" on page 77

`writev()` Function

The `writev` filter method is called when multiple buffers of output data are to be sent. Filters that modify or consume outgoing data can 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` calls. As a result, filters for 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);
```

Return Values

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() Function" on page 82, "net_write() Function" on page 91, "write() Function" on page 163, "filter_create() Function" on page 77

CHAPTER 7

Data Structure Reference

NSAPI uses many data structures that are defined in the `nsapi.h` header file, which is in the *install-dir*/`include` directory.

This chapter describes public data structures in `nsapi.h`.

Note – The data structures in `nsapi.h` that are not described in this chapter are considered private and could change incompatibly in future releases. Some of the data structures described in this chapter might contain additional, undocumented fields. These fields are also considered private and might change incompatibly in future releases. Additional fields may be added in future release, so do not make assumptions regarding the size of data structures.

This chapter has the following sections:

Public Data Structures

This section describes public data structures in `nsapi.h`.

`Session` Data Structure

A session is the time between the opening and closing of the connection between the client and the server.

The following list describes the most important fields in this data structure:

- `sn->client`— Pointer to a `pblock` containing information about the client such as its IP address, DNS name, or certificate.
- `sn->csd`— Platform-independent client socket descriptor. This pointer is passed to the routines for reading from and writing to the client.

The `Session` data structure holds variables that apply to a client, regardless of the requests being sent.

```
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;
} Session;
```

The following list describes the most important fields in the `Session` data structure:

- `client` — Pointer to a `pblock` containing information about the client such as its IP address, DNS name, or SSL certificate. The `ip` parameter contains the client's IP address. Do not modify the contents of this `pblock`.
- `csd` — The platform-independent client socket descriptor used to communicate with the client. This descriptor can be passed to routines such as `net_write` to send output to the client.
- `inbuf` — Pointer to the input buffer for the client socket descriptor. This pointer can be passed to routines such as `netbuf_grab` or `netbuf_getc` to receive input from the client.

Note – The Session NSAPI data structure cannot be used concurrently by multiple threads. Do not retain any reference to a Session or its contents after processing of the current request is complete.

pblock Data Structure

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, pblock 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 6, "NSAPI Function and Macro Reference." You do not need to write code that accesses pblock data fields directly.

```
typedef struct {
    int hsize;
    struct pb_entry **ht;
} pblock;
```

Note – The pblock NSAPI data structure can not be used concurrently by multiple threads. Do not retain any reference to a pblock or its contents after processing of the current request is complete.

pb_entry Data Structure

The pb_entry is a single element in the parameter block.

```
struct pb_entry {
    pb_param *param;
    struct pb_entry *next;
};
```

pb_param Data Structure

The pb_param represents a name-value pair, as stored in a pb_entry.

```
typedef struct {
    char *name,*value;
} pb_param;
```

`Request` Data Structure

The `Request` data structure describes an HTTP transaction, for example, the variables include the client's HTTP request headers.

```
typedef struct{
    */Server working variables */
    pblock *vars;

    /* The method, URI, and protocol revision of this request */
    pblock *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;
```

The following list describes the most important fields in the `Request` data structure:

- `vars`— Pointer to a `pblock` containing information about request-response processing. SAFs may modify the contents of this `pblock` according to the rules established in “Required Behavior of SAFs for Each Directive” on page 35.
- `reqpb` — Pointer to a `pblock` containing information about the client's HTTP request. The `method` parameter contains the HTTP request method, the `uri` parameter contains the path portion of the requested URI, the optional query parameter contains any query string from the requested URI, and the `protocol` parameter contains the HTTP protocol version. Do not modify the contents of this `pblock`.
- `headers` — Pointer to a `pblock` containing the client's HTTP request headers. By convention, all parameter names are lowercase. Do not modify the contents of this `pblock`.
- `senthdrs` — Indicates whether the server has sent HTTP response headers. Service SAFs may set `rq->senthdrs = 1` to prevent the server from sending HTTP response headers.
- `srvhdrs` — Pointer to a `pblock` containing the server's HTTP response headers. By convention, all parameter names are lowercase. SAFs and filters may modify the contents of this `pblock`.

Note – The `Request` NSAPI data structure cannot be used concurrently by multiple threads. Do not retain any references to a `Request` or its contents after processing of the current request.

`stat` Data Structure

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 the implementation of your platform, but the basic outline of the structure is as follows:

```
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` Data Structure

```
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` Data Structure

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 Data Structure

The sendfiledata data structure is used to pass parameters to the net_sendfile function. The parameters are 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 Data Structure

The Filter data structure is an opaque representation of a filter. A Filter structure is created by calling "filter_create() Function" on page 77.

```
typedef struct Filter Filter;
```

FilterContext Data Structure

The FilterContext data structure stores the context associated with a particular filter layer. Filter layers are created by calling "filter_insert() Function" on page 79.

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` Data Structure

The `FilterLayer` data structure represents one layer in a filter stack. The `FilterLayer` structure identifies the filter installed at that layer. This structure 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` Data Structure

The `FilterMethods` data structure is passed to `filter_create` to define the filter methods that a filter supports. Each new `FilterMethods` instance must be initialized with the `FILTER_METHODS_INITIALIZER` macro. For each filter method that 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;
```

CHAPTER 8

Dynamic Results Caching Functions

The functions described in this chapter enables you to write a results caching plug-in for Sun Java System Web Server. A results caching plug-in, which is a `Service` SAF, caches data, a page, or part of a page in the web server address space, which the Web Server can refresh periodically on demand. An `Init` SAF initializes the callback function that performs the refresh.

This chapter has the following sections:

- "About Results Caching Plug-ins" on page 175
- "Dynamic Result Cache Functions" on page 176

About Results Caching Plug-ins

A results caching plug-in can generate a page for a request in three parts:

- A header, such as a page banner, which changes for every request
- A body, which changes less frequently
- A footer, which also changes for every request

Without this feature, a plug-in would have to generate the whole page for every request unless an `IFRAME` is used, where the header or footer is sent in the first response along with an `IFRAME` pointing to the body. In this case, the browser must send another request for the `IFRAME`.

If the body of a page has not changed, the plug-in needs to generate only the header and footer and to call the `dr_net_write` function instead of `net_write`with the following arguments:

- Header
- Footer
- Handle to cache
- Key to identify the cached object

The Web Server constructs the whole page by fetching the body from the cache. If the cache has expired, the Web Server calls the refresh function and sends the refreshed page back to the client.

An `Init` SAF that is visible to the plug-in creates the handle to the cache. The `Init` SAF must pass the following parameters to the `dr_cache_init` function:

- `RefreshFunctionPointer`
- `FreeFunctionPointer`
- `KeyComparatorFunctionPtr`
- `RefreshInterval`

  The `RefreshInterval` value must be a `PRIntervalTime` type. For more information, see the NSPR reference at `http://www.mozilla.org/projects/nspr/reference/html/index.html`.

  As an alternative, if the body is a file that is present in a directory within the web server system machine, the plug-in can generate the header and footer and call the `fc_net_write` function along with the file name.

  This chapter lists the most important functions that a results caching plug-in can use. For more information, see the *install-dir*`/include/drnsapi.h` file.

Dynamic Result Cache Functions

This section describes the dynamic result cache functions.

`dr_cache_destroy()` Function

The `dr_cache_destroy` function destroys and frees resources associated with a previously created and used cache handle. This handle cannot be used in subsequent calls to any of the above functions unless another `dr_cache_init` is performed.

Syntax

```
void dr_cache_destroy(DrHdl *hdl);
```

Parameters

`DrHdl *hdl` is a pointer to a previously initialized handle to a cache. For more information, see “`dr_cache_init()` Function” on page 177.

Returns

`void`

Example

```
dr_cache_destroy(&myHdl);
```

dr_cache_init() Function

The dr_cache_init function creates a persistent handle to the cache, or NULL on failure. This function is called by an Init SAF.

Syntax

```
PRInt32 dr_cache_init(DrHdl *hdl, RefreshFunc_t ref, FreeFunc_t fre,
                      CompareFunc_t cmp, PRUint32 maxEntries,
                      PRIntervalTime maxAge);
```

Returns

1 if successful.

0 if an error occurs.

Parameters

The following table describes parameters for the dr_cache_init function.

DrHdl hdl	Pointer to an unallocated handle.
RefreshFunc_t ref	Pointer to a cache refresh function. This value can be NULL. See the DR_CHECK flag and DR_EXPIR return value for dr_net_write.
FreeFunc_t fre	Pointer to a function that frees an entry.
CompareFunc_t cmp	Pointer to a key comparator function.
PRUint32 maxEntriesp	Maximum number of entries possible in the cache for a given hdl.
PRIntervalTime maxAgep	The maximum amount of time that an entry is valid. If 0, the cache never expires.

Example

```
if(!dr_cache_init(&hdl, (RefreshFunc_t)FnRefresh, (FreeFunc_t)FnFree,
  (CompareFunc_t)FnCompare, 150000, PR_SecondsToInterval(7200)))
{
    ereport(LOG_FAILURE, "dr_cache_init() failed");
    return(REQ_ABORTED);
}
```

dr_cache_refresh() Function

The dr_cache_refresh function provides a way to refresh a cache entry when the plug-in requires it. This refresh can be achieved by passing NULL for the ref parameter in dr_cache_init and by passing DR_CHECK in a dr_net_write call. If DR_CHECK is passed to dr_net_write and it returns with DR_EXPIR, the plug-in should generate new content in the entry and call dr_cache_refresh with that entry before calling dr_net_write again to send the response.

You can use the plug-in to replace the cached entry even if it has not expired based on some other business logic. The dr_cache_refresh function is useful in this case. The plug-in then does the cache refresh management actively.

Syntax

```
PRInt32 dr_cache_refresh(DrHdl hdl, const char *key,
                        PRUint32 klen, PRIntervalTime timeout,
                        Entry *entry, Request *rq, Session *sn);
```

Return Values

1 if successful.

0 if an error occurs.

Parameters

DrHdl hdl	Persistent handle created by the dr_cache_init function.
const char *key	Key to cache, search, or refresh.
PRUint32 klen	Length of the key in bytes.
PRIntervalTime timeout	Expiration time of this entry. If a value of 0 is passed, the maxAge value passed to dr_cache_init is used.
Entry *entry	The not NULL entry to be cached.
Request *rq	Pointer to the request.
Session *sn	Pointer to the session.

Example

```
Entry entry;
char *key = "MOVIES"
GenNewMovieList(&entry.data, &entry.dataLen);  // Implemented by
                                               // plugin developer
```

```
if(!dr_cache_refresh(hdl, key, strlen(key), 0, &entry, rq, sn))
{
    ereport(LOG_FAILURE, "dr_cache_refresh() failed");
    return REQ_ABORTED;
}
```

`dr_net_write()` Function

The `dr_net_write` function sends a response back to the requestor after constructing the full page with `hdr`, the content of the cached entry as the body located using the `key`, and `ftr`. The `hdr`, `ftr`, or `hdl` can be NULL, but not all of them can be NULL. If `hdl` is NULL, no cache lookup is done. The caller must pass `DR_NONE` as the flag.

By default, this function refreshes the cache entry if it has expired by making a call to the `ref` function passed to `dr_cache_init`. If no cache entry is found with the specified `key`, this function adds a new cache entry by calling the `ref` function before sending out the response. However, if the `DR_CHECK` flag is passed in the `flags` parameter and if either the cache entry has expired or the cache entry corresponding to the `key` does not exist, `dr_net_write` does not send any data out. Instead, the function returns with `DR_EXPIR`.

If `ref` which is passed to `dr_cache_init` is NULL, the `DR_CHECK` flag is not passed in the `flags` parameter, and the cache entry corresponding to the `key` has expired or does not exist, then `dr_net_write` fails with `DR_ERROR`. However, `dr_net_write` refreshes the cache if `ref` is not NULL and `DR_CHECK` is not passed.

If `ref` which is passed to `dr_cache_init` is NULL and the `DR_CHECK` flag is not passed but `DR_IGNORE` is passed and the entry is present in the cache, `dr_net_write` sends out the response even if the entry has expired. However, if the entry is not found, `dr_net_write` returns `DR_ERROR`.

If `ref` which is passed to `dr_cache_init` is not NULL and the `DR_CHECK` flag is not passed but `DR_IGNORE` is passed and the entry is present in the cache, `dr_net_write` sends out the response even if the entry has expired. However, if the entry is not found, `dr_net_write` calls the `ref` function and stores the new entry returned from `ref` before sending out the response.

Syntax

```
PRInt32 dr_net_write(DrHdl hdl, const char *key, PRUint32 klen, const char *hdr,
                    const char *ftr, PRUint32 hlen, PRUint32 flen,
                    PRIntervalTime timeout, PRUint32 flags,
                    Request *rq, Session *sn);
```

Return Values

IO_OKAY if successful.

IO_ERROR if an error occurs.

DR_ERROR if an error in cache handling occurs.

DR_EXPIR if the cache has expired.

Parameters

The following table describes parameters for the dr_net_write function.

DrHdl hdl	Persistent handle created by the dr_cache_init function
const char *key	Key to cache, search, or refresh
PRUint32 klen	Length of the key in bytes
const char *hdr	Any header data, which can be NULL.
const char *ftr	Any footer data, which can be NULL.
PRUint32 hlen	Length of the header data in bytes, which can be 0.
PRUint32 flen	Length of the footer data in bytes, which can be 0.
PRIntervalTime timeout	Timeout before this function aborts.
PRUint32 flags	ORed directives for this function. See "Flags" on page 180.
Request *rq	Pointer to the request
Session *sn	Pointer to the session

Flags

This section describes the flags for dr_net_write function.

DR_NONE	Specifies that no cache is used, so the function works as net_write does, DrHdl can be NULL.
DR_FORCE	Forces the cache to refresh, even if it has not expired.
DR_CHECK	Returns DR_EXPIR if the cache has expired. If the calling function has not provided a refresh function and this flag is not used, DR_ERROR is returned.
DR_IGNORE	Ignores cache expiration and sends out the cache entry even if it has expired.
DR_CNTLEN	Supplies the Content-Length header and does a PROTOCOL_START_RESPONSE.
DR_PROTO	Does a PROTOCOL_START_RESPONSE.

Example

```
if(dr_net_write(Dr, szFileName, iLenK, NULL, NULL, 0, 0, 0,
                DR_CNTLEN | DR_PROTO, rq, sn) == IO_ERROR)
{
    return(REQ_EXIT);
}
```

`fc_open()` Function

The `fc_open` function returns a pointer to `PRFileDesc` that refers to an open file (`fileName`). The `fileName` 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 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);
```

Return Values

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.

`fc_close()` Function

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

Syntax

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

Return Values

`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_net_write()` Function

Use `fc_net_write` function is to send a header or a footer and a file that exists somewhere in the system. The `fileName` should be the full path name of a file.

Syntax

```
PRInt32 fc_net_write(const char *fileName, const char *hdr,
                    const char *ftr, PRUint32 hlen,
                    PRUint32 flen, PRUint32 flags,
                    PRIntervalTime timeout, Session *sn, Request *rq);
```

Return Values

`IO_OKAY` if successful.

`IO_ERROR` if an error occurs.

`FC_ERROR` if an error in file handling occurs.

Parameters

`const char *fileName`	File to be inserted
`const char *hdr`	Any header data, which can be NULL

`const char *ftr`	Any footer data, which can be NULL
`PRUint32 hlen`	Length of the header data in bytes, which can be 0
`PRUint32 flen`	Length of the footer data in bytes, which can be 0
`PRUint32 flags`	ORed directives for this function. See "Flags" on page 183.
`PRIntervalTime timeout`	Timeout before this function aborts
`Request *rq`	Pointer to the request
`Session *sn`	Pointer to the session

Flags

This section describes the flags for `fc_net_write()` function.

`FC_CNTLEN`	Supplies the `Content-Length` header and does a `PROTOCOL_START_RESPONSE`
`FC_PROTO`	Does a `PROTOCOL_START_RESPONSE`

Example

```
const char *fileName = "/docs/myads/file1.ad";
char *hdr = GenHdr(); // Implemented by plugin
char *ftr = GenFtr(); // Implemented by plugin

if(fc_net_write(fileName, hdr, ftr, strlen(hdr), strlen(ftr),
    FC_CNTLEN, PR_INTERVAL_NO_TIMEOUT, sn, rq) != IO_OKEY)
{
    ereport(LOG_FAILURE, "fc_net_write() failed");
    return REQ_ABORTED;
}
```

◆ ◆ ◆ APPENDIX A

Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) is a protocol which is a set of rules that describes how information is exchanged, that enables a client such as a web browser and a Web 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 has the following sections:

Compliance

Sun Java System Web Server supports HTTP/1.1. 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 2616) at `http://www.ietf.org/rfc/rfc2616.txt`.

Requests

A request from a browser to a server includes the following information:

- "Request Method, URI, and Protocol Version" on page 186
- "Request Headers" on page 186
- "Request Data" on page 186

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 of these request headers are optional. This section lists some of the commonly used request headers.

`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.
`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, no data exists to send.

Responses

The server's response includes the following:

- "HTTP Protocol Version, Status Code, and Reason Phrase" on page 187
- "Response Headers" on page 188
- "Response Data" on page 189

HTTP Protocol Version, Status Code, and Reason Phrase

The server sends back a status code, which is a three-digit numeric code. The five categories of status codes are:

- `100-199` code indicate a provisional response.
- `200-299` code indicate a successful transaction.
- `300-399` code indicate the requested resource should be retrieved from a different location.
- `400-499` code indicate an error was caused by the browser.
- `500-599` code indicate a serious error occurred in the server.

  The following table lists some common status codes.

TABLE A–1 Common HTTP Status Codes

Status Code	Meaning
`200`	The 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 result is not an error. 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 result 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.

TABLE A–1 Common HTTP Status Codes *(Continued)*

Status Code	Meaning
`401`	Unauthorized. The user requested a document but did not provide a valid user name or password.
`403`	Forbidden. Access to this URL is forbidden.
`404`	Not found. The document requested is not on the server. This code can also be sent if the server is configured to protect the document for unauthorized personnel.
`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 returns this error 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`	A server-related error occurred. The server administrator must check the error log in the server.
`503`	Sent if the quality of service mechanism was enabled and bandwidth or connection limits were attained. The server then serves requests with that code.

Response Headers

The response headers contain information about the server and the response data. This section lists some common response headers.

`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 data that follows
`WWW-Authenticate`	Used during authentication and includes information that tells the browser software what information is necessary for authentication such as user name and password

Response Data

The server sends a blank line after the last header. It then sends the response data such as an image or an HTML page.

APPENDIX B

Alphabetical List of NSAPI Functions and Macros

This appendix provides an alphabetical list for the easy lookup of NSAPI functions and macros.

NSAPI Functions and Macros

Index