# NSAPI Programmer's Guide

Netscape Enterprise Server

Version 3.0

.


Recycled and Recyclable Paper

The Team:
Technical Writing: David H Nelson
Publications: Corey Bridges, Guy K. Haas, Kelly A. Harrison
Engineering:Chris Apple, Mike Barbarino, Mike Belshe, Jim Black, Fred Cox, George Dong, Alex Feygin, Alan Freier, Andy Hakim, Warren Harris, John K. Ho, Ari Luotonen, Mike McCool, Rob McCool, Chuck Neerdaels, Howard Palmer, Ben Polk, Aruna Victor
Marketing: Mike Blakely, Atri Chatterjee, Ben Horowitz, David Pann
Quality Assurance: Saleem Baber, Roopa Cheluvaiah, Shvetal Desai, Noriko Hosoi, Teresa Hsiao, Pramod Khincha, Joy Lenz, Rajesh Menon, Jun Tong, Cathleen Wang, Carol Widra, Ayyaz Yousaf
Technical Support: John Benninghoff, Brian Kendig, Anthony Lee-Masis, Trevor Placker, Bill Reviea, Dan Yang r

Version 3.0



Printed in USA
97 96 10 9 8 7 6 5 4 3 2 1

Netscape Communications Corporation 501 East Middlefield Road, Mountain View, CA 94043

# Contents

Chapter

1

# NSAPI Basics

This chapter provides an introduction to the Netscape Server Application Programming Interface (NSAPI) and how it is used in the Enterprise Server's request-response process to handle HTTP transactions.

Clients send requests to the server when they want access to resources such as HTML documents, images, CGI programs, and imagemap files. Concepts discussed in this chapter include Server Application Functions (SAFs, functions the server calls when processing a request), directives (identifiers for the various steps in the request-response process), and the object configuration file (`obj.conf`).

## Introduction

NSAPI is used to implement the SAFs which provide the core and extended functionality of the Enterprise Server. It allows the server's processing of requests to be divided into small steps which may be arranged in a variety of ways for speed and flexible configuration.

The Server provides a set of built-in SAFs. In addition, you may extend the server's built-in functionality by writing your own custom SAFs using NSAPI. See Chapter 3, "Creating Custom SAFs" for more information.

NSAPI was designed by the creators of the NCSA and CERN web servers. It has been exposed to software developers so that they may take advantage of its speed, tight integration with the server, and flexibility. However, this requires a solid understanding of the server process. If you are converting a CGI or writing a CGI-like server extension, you may not need to use NSAPI. You may use the Web Application Interface (WAI).

WAI is designed to make it easier to implement new server functionality. WAI provides faster performance than CGI. It also has the ability to run in a distributed environment since it is based on Internet Inter-ORB Protocol (IIOP). Refer to the WAI documentation for more information.

If you are interested in developing custom authorization, custom logging, or modifying existing server behavior, then you'll want to use NSAPI. This chapter describes how the Netscape Enterprise Server processes an HTTP request and how SAFs, written with NSAPI, handle the process.

# HTTP Basics

The Netscape Enterprise Server is a web server which accepts and responds to HyperText Transfer Protocol (HTTP) requests. Browsers like Netscape Navigator communicate using several protocols including HTTP, FTP, and gopher. The Enterprise Server handles HTTP specifically.

The HTTP protocol is fast, simple, and extensible. For more detailed information refer to Appendix A, "HyperText Transfer Protocol" and the latest HTTP specification.

As a quick summary, the HTTP protocol works as follows:

- the browser opens a connection to the server and sends a request
- the server processes the request, generates a response, and closes the connection (or leaves the connection open and waits for another request if it finds a `Connection: Keep-alive` header.)

The request consists of a method, Universal Resource Identifier (URI), and HTTP protocol version separated by spaces. This is normally followed by a number of headers, a blank line indicating the end of the headers, and

sometimes body data. Headers may provide various information about the request or the client Body data is typically only sent for POST and PUT methods.

A typical request might look like this:

```
GET /index.html HTTP/1.0
User-agent: Mozilla
Accept: text/html, text/plain, image/jpeg, image/gif, */*
```

Notice that the end of the headers is identified by a blank line and that there is no body data sent for this HTTP GET request.

The server receives the request and processes it. It handles each request individually, although it may process many requests simultaneously. There are certain steps that must be taken to process the request. We call this the "Request-Response process." This is where the SAFs written using NSAPI are executed. We'll cover the process in detail in the next section.

The server generates a response which includes the HTTP protocol version, HTTP status code, and a reason phrase separated by spaces. This is normally followed by a number of headers. The end of the headers is indicated by a blank line. The body data of the response follows. A typical HTTP response might look like this:

```
HTTP/1.0 200 OK
Server: Netscape Enterprise Server/3.0
Content-type: text/html
Content-length: 83

<HTML>
<HEAD><TITLE>Hello World</Title></HEAD>
<BODY>Hello World</BODY>
</HTML>
```

The status code and reason phrase tell the browser how the server handled the request. Normally the status code 200 is returned indicating that the request was handled successfully and the body date contains the requested item. Other result codes indicate redirection to another server or the browser's cache, or various types of HTTP errors such as 404 Not Found.

# The Request-Response Process

When the server first starts up it performs some initialization and then waits for an HTTP request. A request contains a URI (Universal Resource Identifier), an HTTP method, and may contain some number of headers providing various information about the request or the client.

The server handles each request individually, although it may process many requests at the same time. There are certain steps that must be taken to handle the various types of requests. We call this the “Request-Response process.”

The request-response process normally consists of six sequential steps. They are:

1. **AuthTrans** (authorization translation) verify any authorization information (such as name and password) sent in the request.
2. **NameTrans** (name translation) translate the logical URI into a local file system path.
3. **PathCheck** (path checking) check the local file system path for validity and approved access by the requesting user.
4. **ObjectType** (object typing) determine the MIME-type (Multi-purpose Internet Mail Encoding) of the requested resource (eg. text/html, image/gif, etc).
5. **Service** (service) return the response to the client.
6. **AddLog** (adding log entries) add entries to log file(s).

A number of operations may be performed at each of these steps to accomplish its purpose. These operations are performed by Server Application Functions (SAFs). There are quite a variety of SAFs built into the server which handle the core server functionality and its advanced features.

Each SAF has its own settings. It also has access to the request information and any other server variables created or modified by previous SAFs. The SAF performs its operation based on all this information. It may examine, modify, or create server variables based on the current request and its purpose within its step.

Each SAF returns a result code which tells the server whether it succeeded, did nothing, or failed. Based on the result code, the server decides whether to skip to the next request-response step, execute the next SAF in the current step, or abort the process.

Normally, all of the steps are completed in order, the response is sent to the client (in the Service step), and entries are added to the log file(s) (in the AddLog step).

However, if the process is aborted for any reason (such as not finding the requested resource or the user not being authorized), the server skips to a new step called "**Error**". This step is taken in place of the Service step. The Error SAF may send a custom HTML page to the client describing the problem. Then the server continues with the AddLog step.

# The Configuration Files

The Enterprise Server is configured using several text-based configuration files. The configuration files are located in the `config` directory in the server's home directory (`https-<servername>/config`). Three of the configuration files we're interested in are: `magnus.conf`, `obj.conf`, and `mime.types`. Normally the Administration Server is used to change the settings in these files.

The `magnus.conf` file is the primary server configuration file. It determines the server name, port, and other global server settings.

The object configuration file, `obj.conf`, configures initialization and operation of the SAFs for each step in the request-response process. You'll edit this file to install and configure custom SAFs. It will be covered in detail in the next section.

The `mime.types` file determines the mapping of file name extensions to HTTP content-type, content-encoding, and content-language.

# The obj.conf File

The `obj.conf` file controls the initialization and request-response process for the server. It contains directive lines for initialization and for all the request-response steps. The request-response directives are grouped into “objects” with the `<Object>` tag. They're called “objects” because they inherit behavior from other objects.

The “default” object controls the default operation of the server for all requests. Other objects will inherit and/or modify behavior of the “default” object for special types of requests. Some examples are CGI programs, private directories, and custom file types.

Directive lines determine which SAFs will be executed at each step in the request-response process. Every directive line has a `"fn"` parameter that indicates which SAF to execute.

The syntax of each directive line is:

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

`Directive` is one of the server directives.

`func-name` is the name of the SAF to execute.

`nameN="valueN"` are the names and values of paramaters which are passed to the SAF.

Here are the server directives and a description of what each does:

- **Init** - Initializes server subsystems and shared resources.
- **AuthTrans** - Verifies any authorization information (normally sent in the Authorization header) provided in the HTTP request and translates it into a user and/or a group. Server access control occurs in two stages. AuthTrans verifies the authenticity of the user. Later, PathCheck tests the user's access privileges for the requested resource.
- **NameTrans** - Translates the URL specified in the request from a virtual path to a physical file system path for the requested resource. This may also result in redirection to another site.
- **PathCheck** - Performs tests on the physical path determined by the NameTrans step. In general, these tests determine whether the path is valid and whether it is allowed for the client to access the requested resource.

- **ObjectType** - Determines the MIME (Multi-purpose Internet Mail Encoding) type of the requested resource. This is normally done by using the `mime.types` file to map the file name extension into a MIME type. The resulting type may be:
  - A common document type such as "`text/html`" or "`image/gif`" (for example, the file name extension .gif translates to the MIME type "`image/gif`").
  - An internal server type. Internal types always begin with "`magnus-internal/`" and determine which Service directive SAF the server should execute. The Service SAF will report the actual MIME type of the data it returns.
- **Service** - Sends the response to the client. This involves setting the HTTP result status, setting up response headers (such as content-type and content-length), and sending the response data.
- **Error** - Handles an HTTP error (REQ_ABORTED). Typically this involves sending a custom HTML document to the user describing the problem and possible solutions.
- **AddLog** - Records information about the transaction.

The `obj.conf` file is very sensitive to case, extra spaces, indenting, etc. See Chapter 2, “Directives and Built-In SAFs” for details about the file’s syntax.

# Objects and Clients

Request-response directives in the `obj.conf` file are grouped into "objects" which begin with an `<Object>` tag and end with a `</Object>` tag. They are called objects because they inherit and modify the "default" object’s behavior.

The directives in an object are enabled when their parameters match values in the current request. An object tag may have a `name` parameter or a `ppath` parameter. Either parameter may be a wildcard pattern. For example:

```
<Object name="cgi">
```

or

```
<Object ppath="/usr/netscape/suitespot/docs/private/*">
```

The directives in the object with the name "default" are always enabled. When enabled, the directives in additional objects are treated as if they appear before the same type of directives in the default object. This allows the new objects to modify the behavior of the "default" object.

Additional objects with a `name` parameter are normally enabled from a NameTrans SAF such as `pfx2dir` or `assign-name`. Since the AuthTrans and NameTrans steps have already been completed, directives of these types in the new object will not be executed.

Additional objects with a `ppath` parameter are enabled when the physical path matches their wildcard value. The physical path ("path" in the `rq->vars` pblock) is set after the NameTrans step. Once again, the AuthTrans and NameTrans steps have already completed, so directives of these types in the new object will not be executed.

Objects with a `ppath` parameter are typically used in one of two ways. One is to execute directives when the path is within a certain directory. The wildcard pattern is the local file system path ending with a "*" such as `"/usr/netscape/suitespot/docs/private/*"`. The other is to execute directives for a certain file type based on its file name extension. The wildcard pattern is a "*" followed by the extension such as `"*.cgi"`.

The `<Client>` tag may be used within an object to limit a group of directives to specific clients. Directives between a `<Client>` tag and a matching `</Client>` tag will only be executed if the client's information matches the `<Client>` parameters.

A `<Client>` tag may have parameters for `ip`, `dns`, and/or `host`. The value of these parameters are wildcard patterns. For example:

```
<Client ip="198.95.251.*">
```

or

```
<Client dns="*.netscape.com">
```

The directives in the `<Client>` block are only executed if the current client matches all of the parameters.

The `ip` parameter is the IP address of the client. The `dns` parameter is the DNS name of the client.

The `host` parameter is typically used to configure "software virtual servers." These are multiple "virtual" servers on the same machine. There is really only one web server running on the machine, but there may be many DNS names which map to the machines IP address. The web server can tell which "virtual" server was requested because Navigator includes a "Host" header in the request which tells the DNS name of the server that the user requested.

# SAFs in the Object Configuration File

As mentioned earlier, SAFs are functions the server calls when processing requests. Some SAFs take parameters specified on the directive line in the `obj.conf` file (such as `type="application/octet-stream"`). The result code returned to the server by a given SAF determines whether the server executes any subsequent SAFs in the current step, skips to the next step in the process, or takes some other action. Refer to Chapter 2, “Directives and Built-In SAFs” for complete details.

Chapter

2

# Directives and Built-In SAFs

This chapter describes the directives and Server Application Functions (SAFs) that are provided with the server. They are used in the `obj.conf` file to configure the operation of the server.

## The obj.conf file

The obj.conf file contains directive lines which configure the operation of the server. The syntax of each directive line is:

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

`Directive` is one of the server directives.

`func-name` is the name of the SAF to execute.

`nameN="valueN"` are the names and values of paramaters which are passed to the SAF.

Here are the server directives and a description of what each does:

- **Init** - Initializes server subsystems and shared resources.

- **AuthTrans** - Verifies any authorization information (normally sent in the Authorization header) provided in the HTTP request and translates it into a user and/or a group. Server access control occurs in two stages. AuthTrans verifies the authenticity of the user. Later, PathCheck tests the user's access privileges for the requested resource.
- **NameTrans** - Translates the URL specified in the request from a virtual path to a physical file system path for the requested resource. This may also result in redirection to another site.
- **PathCheck** - Performs tests on the physical path determined by the NameTrans step. In general, these tests determine whether the path is valid and whether it is allowed for the client to access the requested resource.
- **ObjectType** - Determines the MIME (Multi-purpose Internet Mail Encoding) type of the requested resource. This is normally done by using the `mime.types` file to map the file name extension into a MIME type. The resulting type may be:
  - A common document type such as "`text/html`" or "`image/gif`" (for example, the file name extension .gif translates to the MIME type "`image/gif`").
  - An internal server type. Internal types always begin with "`magnus-internal/`" and determine which Service directive SAF the server should execute. The Service SAF will report the actual MIME type of the data it returns.
- **Service** - Sends the response to the client. This involves setting the HTTP result status, setting up response headers (such as content-type and content-length), and sending the response data.
- **Error** - Handles an HTTP error (REQ_ABORTED). Typically this involves sending a custom HTML document to the user describing the problem and possible solutions.
- **AddLog** - Records information about the transaction.

# Syntax

Several rules are important in the `obj.conf` file. Be very careful when editting this file. Simple mistakes can make the server fail to start or operate incorrectly.

## Parameters

The number and names of parameters depends on the function. The order of parameters on the line is not important.

## Case Sensitivity

Items in the `obj.conf` file are case-sensitive including function names, parameter names, many parameter values, and path names.

## Separators

The "C" language allows function names to be composed only of letters, digits, and underscores. You may use the hyphen (-) character in the configuration file in place of underscore (_) for your "C" code function names. This is only true for function names.

## Quotes

Quotes (") are only required around value strings when there is a space in the string. Otherwise they are optional. Each open-quote must be matched by a close-quote.

## Spaces

Spaces are not allowed at the beginning of a line except when continuing the previous line. Spaces are not allowed before or after the equal (=) sign that separates the name and value. Spaces are not allowed at the end of a line or on a blank line.

## Line Continuation

A long line may be continued on the next line by beginning the next line with a space or tab.

### Path Names

Always use forward slashes (/) rather than back-slashes (\) in path names under Windows NT. Back-slash escapes the next character.

### Comments

Comments begin with a pound (#) sign. If you administer the server with the Administration Server, your comments will not be preserved.

# The directives and SAFs

## Init directive

`Init` is the directive that is used to initialize server subsystems such as access logging, file typing, and loading of custom SAFs. These functions are called upon server startup or restart.

On Unix platforms, each `Init` directive has an optional `LateInit` parameter. If it is set to "`yes`"or is not provided, the function is executed by the child process after it is forked from the parent. If it is set to "`no`", the function is executed by the parent process before the fork. Any activities that must be performed as the user `root` (such as writing to a root-owned file) must be done before the fork. Any activities involving the creation of threads must be performed after the fork.

Upon failure, Init-class functions return `REQ_ABORTED` and insert into `pb` a variable named `error` that contains a string describing the cause of the error. The server logs this error and terminates. Any other result code is considered success.

The following Init-class functions are described in detail in this section:

- `cache-init` configures server caching for increased performance.
- `cindex-init` changes the default characteristics for fancy indexing.
- `dns-cache-init` configures DNS caching.
- `flex-init` initializes the flexible logging system.

- `IIOPInit` initializes the Web Application Interface (WAI) which uses Internet Inter-ORB Protocol (IIOP)
- `init-cgi` changes the default settings for CGI programs.
- `init-clf` initializes the Common Log subsystem.
- `init-uhome` loads user home directory information.
- `load-modules` loads shared libraries into the server.
- `load-types` loads file extension to MIME type mapping information.
- `pool-init` configures pooled memory allocation.

## cache-init

The `cache-init` function controls file caching. The server caches files to improve performance. File caching is enabled by default.

The `cache-init` function has three arguments:

- `cache-size` specifies the size of the cache in bytes. Valid values for the number of elements in the cache are 32 to 32768; the default is 512.

  The `cache-size` value should be greater than the size of all the documents on your server. You should include any static file such as HTML, text, images, sounds, or any other unchanging data. URLs that are dynamic (such as CGI or NSAPI routines) return different data, depending on who calls them, and should not be counted.

- `mmap-max` specifies the maximum amount of memory set aside for memory-mapped (*mmap*) files the server will keep open at any point. Acceptable values range from 512K to (512*1024)KB; the default is 10000KB (10MB).

  To get maximum speed, the cache keeps many mmap files open. To estimate the optimal value for `mmap-max` on your system, approximately compute the total number of bytes of “static” data that’s on your system. For example, if you have 200 files that are 10K in size, then 2MB should be sufficient for `mmax-map`.

- `disable` specifies, if set to anything but "false", that the cache is to be disabled.

**Note** To optimize server speed, you should ideally have enough RAM for the server and cache because swapping can be slow. Do not allocate a cache that is greater in size than the amount of memory on the system.

**Example**

```
Init fn=cache-init cache-size=512 mmap-max=10000
```

## cindex-init

The function `cindex-init` is used to change the defaut settings for fancy indexing (see `index-common`).

**Parameters**

`opts` (optional) is a string of letters specifying the options to activate:

- `s` tells the server to scan each HTML file in the directory it's indexing in order to place the HTML `<TITLE>` in the description field. The file must be shorter than 255 characters. This option is off by default.

`widths` (optional) specifies the width for each column in the indexing printout. A zero width disables the column. The string is a comma-separated list of numbers that specify the column widths in characters for name, last-modified date, size, and description respectively. The default widths are 22, 1, 1, and 33 respectively.

`ignore` (optional) specifies a wildcard pattern for file names the server should ignore while indexing. File names starting with a period (.) are always ignored. The default is to only ignore file names starting with a period (.).

`icon-uri` (optional) specifies the URI prefix the `index-common` function uses when generating URLs for file icons (.gif files). By default, it's `/mc-icons/`. If `icon-uri` is different from the defaut, the `pfx2dir` function in the NameTrans directive must be changed so that the server can find these icons.

**Examples**

```
Init fn=cindex-init widths=50,1,1,0

Init fn=cindex-init ignore=*.html

Init fn=cindex-init widths=22,0,0,50
```

**See Also**

`index-common`

## dns-cache-init

The `dns-cache-init` function specifies (when DNS lookups are enabled ) that DNS lookups should be cached. If DNS lookups are cached, then when the server gets a client's host name information, it can store the data it receives.

Then, if the server needs information about the client in the future, the information is available to the server without querying for the information again.

You may specify the size of the DNS cache and the time it takes before a cache entry becomes invalid. The DNS cache can contain 32 to 32768 entries; the default value is 1024 entries. Values for the time it takes for a cache entry to expire (specified in seconds) can range from 1 second to 1 year; the default value is 1200 seconds (20 minutes).

**Parameters** `cache-size` (optional) specifies how many entries are contained in the cache. Acceptable values are 32 to 32768; the default value is 1024.

`expire` (optional) specifies how long (in seconds) it takes for a cache entry to expire. Acceptable values are 1 to 31536000 (1 year); the default is 1200 seconds (20 minutes).

**Example**

```
Init fn="dns-cache-init" cache-size="2140" expire="600"
```

## flex-init

The `flex-init` function initializes the flexible logging system. It opens the log file whose name is passed as a parameter, and establishes a record format that is passed as another parameter. The log file stays open until the server is shut down or restarted (at which time all logs are closed and reopened).

As in `init-clf`, use `flex-init` to specify a log-file name (such as `loghttp=/netscape/suitespot/https-servername/logs/loghttp`); then you use the log-file name with the AddLog-class `flex-log` function in `obj.conf` to add a log entry to the file (such as `AddLog fn=flex-log name=loghttp`).

**Note** You may specify multiple log file names in the same `flex-init` function call. Then use multiple `AddLog` directives with the `flex-log` function to log transactions to each log file.

`flex-log` may be called more than once. Each new log file name and format will be added the the list of log files.

If you move, remove, or change the log file without shutting down or restarting the server, client accesses might not be recorded. To save or backup a log file, you need to rename the file and then restart the server. The server first looks for the log file by name, and if it doesn't find it, creates a new one (the renamed original log file is left for you to use).

**Parameters** The `flex-init` function has two parameters: one that names the log file and one that specifies the format of each record in that file.

At least one log file should be specified. The name part of the name-value pair is a unique name for the log file. You will use this name later, as a parameter to the `flex-log` function. The value specifies either the full path to the log file or a file name relative to the server's `logs` directory.

The second parameter is `format.logfilename` which specifies the format of each log entry in the log file. Items contained between percent signs (%) are the names of server pblock entries. (See Chapter 3, "Creating Custom SAFs" for more information about pblocks and Chapter 4, "NSAPI Function Reference" for functions to manipulate pblocks.)

One additional item that is available is %SYSDATE% which is the current system date. It is formated using the time format "%d/%b/%Y:%H:%M:%S" plus the offset from GMT.

If no format paramter is specified for a log file, the common log format is used:

```
"%Ses->client.ip% - %Req->vars.auth-user% [%SYSDATE%] \"%Req-
>reqpb.clf-request%\" %Req->srvhdrs.clf-status% %Req->srvhdrs.content-
length%"
```

Any additional text is treated as literal text, so you can add to the line to make it more readable. Typical components of the formatting parameter are listed in Table 2.1. Certain components might contain spaces, so they should be bounded by escaped quotes (\").

Table 2.1 Typical components of flex-init formatting

| Flex-log option | Component |
|---|---|
| Client DNS name (unless "iponly" is specified in flex-log or DNS name is not available) or IP address | `%Ses->client.ip%` |
| Client DNS name | `%Ses->client.dns%` |

Table 2.1 Typical components of flex-init formatting

| Flex-log option | Component |
|---|---|
| System date | %SYSDATE% |
| Full HTTP request line | %Req->reqpb.clf-request% |
| Status | %Req->srvhdrs.clf-status% |
| Response content length | %Req->srvhdrs.content-length% |
| Response content type | %Req->srvhdrs.content-type% |
| Referer header | %Req->headers.referer% |
| User-agent header | %Req->headers.user-agent% |
| HTTP Method | %Req->reqpb.method% |
| HTTP URI | %Req->reqpb.uri% |
| HTTP query string | %Req->reqpb.query% |
| HTTP protocol version | %Req->reqpb.protocol% |
| Accept header | %Req->headers.accept% |
| Date header | %Req->headers.date% |
| If-Modified-Since header | %Req->headers.if-modified-since% |
| Authorization header | %Req->headers.authorization% |
| Name of authorized user | %Req->vars.auth-user% |

**Examples** The first example below initializes flexible logging into the file `/usr/netscape/suitespot/https-servername/logs/access`.

```
Init fn=flex-init access="/usr/netscape/suitespot/https-servername/logs/access"
format.access="%Ses->client.ip% - %Req->vars.auth-user% [%SYSDATE%] \"%Req->reqpb.clf-
request%\" %Req->srvhdrs.clf-status% %Req->srvhdrs.content-length%"
```

This will record the following items

- ip or hostname, followed by the three characters `" - "`
- the user name, followed by the two characters `" ["`
- the system date, followed by the two characters `"] "`

- the full HTTP request in quotes, followed by a single space
- the HTTP result status in quotes, followed by a single space
- the content length

This is the default format, which corresponds to the Common Log Format (CLF).

It is advisable that the first six elements of any log always be in exactly this format, because a number of log analyzers expect that as output.

This example initializes flexible logging into the file `/user/netscape/suitespot/https-servername/logs/extended`.

```
Init fn=flex-init extended="/usr/netscape/suitespot/https-servername/logs/extended"
format.extended="%Ses->client.ip% - %Req->vars.auth-user% [%SYSDATE%] \"%Req->reqpb.clf-
request%\" %Req->srvhdrs.clf-status% %Req->srvhdrs.content-length% %Req-
>headers.referer% \"%Req->headers.user-agent%\" %Req->reqpb.method% %Req->reqpb.uri% %Req-
>reqpb.query% %Req->reqpb.protocol%"
```

**See Also** `flex-log`

## IIOPinit

Initializes IIOP (Internet Inter-ORB Protocol) support used by the Web Application Interface (WAI).

**Parameters** None.

**Examples**

```
Service fn=IIOPinit
```

**See Also** `IIOPexec, IIOPNameService`

## init-cgi

The `init-cgi` function provides certain initializations for the CGI execution. Two options are provided: timeout of the execution of the CGI script, and establishment of environment variables.

**Parameters** The `init-cgi` function takes two kinds of arguments.

`timeout` (optional) specifies how many seconds the server will wait for CGI output. If the CGI script has not delivered any output in that many seconds, the server terminates the script. The default is 300 seconds.

*`name=value`* (optional) specifies the name and value for an environment variable that the server will place into the environment for the CGI. You can set any number of environment variables in a single `init-cgi` function.

**Example** `Init fn=init-cgi LD_LIBRARY_PATH=/usr/lib;/usr/local/lib`

**See Also** `send-cgi, send-wincgi, send-shellcgi`

## init-clf

The `init-clf` function initializes the Common Log subsystem. It opens the log files whose names are given as parameters. The log files stay open until the server is shut down (at which time the log files are closed) or restarted (at which time the log files are closed and reopened).

Calling this function is required if you are using the common log features.

Log file names are then used by the `common-log` function in an `AddLog` directive to record a transaction to a specific log file.

For example, use `init-clf` to specify a name that refers to a log file (such as `mylog=/usr/netscape/suitespot/https-<servername>/logs/mylogfile`); then use the name `mylog` in an `AddLog` directive to add a log entry to the file (such as `AddLog fn=common-log name=mylog`). If you ever change the path or file name of a log file, you need only change it in one place—in the `init-clf` function.

**Note** You may specify multiple log file names in the same `init-clf` function call. Then use multiple `AddLog` directives with the `common-log` function to log transactions to each log file.

This function should only be called once. If it is called again, the new call will replace log file names from all previous calls.

If you move, remove, or change the log file without shutting down or restarting the server, client accesses might not be recorded. To save or backup a log file, you need to rename the file and then restart the server. The server first looks for the log file by name, and if it doesn't find it, creates a new one (the renamed original log file is left for you to use).

**Parameters** At least one log file should be specified. The name part of the name-value pair should be a unique name for the log file. You will use this name later, as a parameter to the `common-log` function. The value specifies either the full path to the log file or a file name relative to the server's `logs` directory.

**Examples**

```
Init fn=init-clf
   access=/usr/netscape/suitespot/https-<servername>/logs/access
Init fn=init-clf templog=/tmp/mytemplog templog2=/tmp/mytemplog2
```

**See Also** `common-log, record-useragent`

## init-uhome

**Unix Only** The `init-uhome` function loads information about the system's user home directories into internal hash tables. This increases memory usage slightly, but improves performance for servers that have a lot of traffic to home directories.

**Parameters** `pwfile` (optional) specifies the full file system path to a file other than `/etc/passwd`. If not provided, the default Unix path (`/etc/passwd`) is used.

**Examples**

```
Init fn=init-uhome
```

```
Init fn=init-uhome pwfile=/etc/passwd-http
```

**See Also** `unix-home, find-links`

## load-modules

The `load-modules` function loads a shared library/Dynamic Link Library into the server code. Specified functions from the library can then be executed from any subsequent directives.

**Parameters** `shlib` specifies either the full path to the shared library/Dynamic Link Library or a file name relative to the server configuration directory.

`funcs` is a comma separated list of the names of the functions in the shared library/Dynamic Link Library to be made available. The list should not contain any spaces. The dash (-) character may be used in place of the underscore (_) character in function names.

`NativeThread` (optional) specifies which threading model to use. `no` causes the routines in the library to use user-level threading. `yes` enables kernel-level threading. The default is `yes`.

**Examples**

```
Init fn=load-modules shlib="C:/mysrvfns/corpfns.dll" funcs="moveit"
Init fn=load-modules shlib="/mysrvfns/corpfns.so"
   funcs="myinit,myservice"
Init fn=myinit
```

## load-types

The `load-types` function creates a table mapping file-name extensions (`ext`) to a file's content-type (`type`), content-encoding (`enc`), and content-language (`lang`). The content-type is represented by a MIME type. The content-encoding and content-language are indicated by the natural language. indicating a file's content-type. and content-encoding, languages. It scans a file that tells it how to map file-name extensions to MIME types. MIME types and languages are essential to tell browsers what type of data is being sent so they can display it correctly. For example, they are used to tell an HTML file from a GIF file.

The file name extensions (`exts`) are not case-sensitive.

This function must be called in order for the `type-by-extension` and `type-by-exp` SAFs, and the `cinfo_find()` functions to work properly.

**Parameters**

`mime-types` specifies either the full path name to a MIME types file or a path name relative to the server configuration directory. The server comes with a default file called `mime.types` in the server's `config` directory.

`local-types` (optional) specifies either the full path name to a MIME types file or a path name relative to the server configuration directory. The file can be used to maintain types that are applicable only to your server.

MIME types files must begin with the following line or they will not be accepted:

```
#--Netscape Communications Corporation MIME Information
```

**Examples**

```
Init fn=load-types mime-types=mime.types
Init fn=load-types mime-types=mime.types
   local-types=/usr/netscape/suitespot/local.types
```

**See Also** `type-by-extension, type-by-exp, force-type`

### pool-init

The `pool-init` function changes the default values of pooled memory settings. The size of the free block list may be changed or pooled memory may be entirely disabled.

Memory allocation pools allow the server to run significantly faster. If you are programming with the NSAPI, note that `MALLOC`, `REALLOC`, `CALLOC`, `STRDUP`, and `FREE` work slightly differently if pooled memory is disabled.. If pooling is enabled, the server automatically cleans up all memory allocated by these routines when each request completes. In most cases, this will improve performance and prevent memory leaks. If pooling is disabled, all memory is global and there is no clean-up.

If you want persistent memory allocation, add the prefix `PERM_` to the name of each routine (`PERM_MALLOC`, `PERM_REALLOC`, `PERM_CALLOC`, `PERM_STRDUP`, and `PERM_FREE`).

**Note** Any memory you allocate from Init-class functions will be allocated as persistent memory, even if you use `MALLOC`. The server cleans up only the memory that is allocated while processing a request, and because Init-class functions are run before processing any requests, their memory is allocated globally.

**Parameters** `free-size` (optional) maximum size in bytes of free block list. May not be greater than 1048576.

`disable` (optional) flag to disable the use of pooled memory. Should have a value of `true` or `false`. Default value is `false`.

**Example**

```
Init fn=pool-init disable=true
```

# AuthTrans Directive

`AuthTrans` stands for Authorization Translation. Server resources can be protected so that accessing them requires the client to provide authorization information from the user.

The server handles the authorization of client users in two steps.

- AuthTrans - validate authorization information sent by the client in the Authorization header.

- PathCheck - check that the authorized user is allowed access to the requested resource.

It is split into two steps so that multiple authorization schemes can be easily incorporated, as well as providing the flexibility to have resources that record authorization information but do not require it.

If there is more than one `AuthTrans` directive, each function is executed in order until one succeeds in authorizing the user (by returning `REQ_PROCEED`).

The following AuthTrans-class functions are described in detail in this section:

- `basic-auth` calls a custom function to verify user name and password. Optionally calls a custom function to determine the user's group.
- `basic-ncsa` verifies user name and password against an NCSA-style or system DBM database. Optionally determines the user's group.

## basic-auth

The `basic-auth` function calls a custom function to verify authorization information sent by the client. The Authorization header is sent as part of the the "basic" server authorization scheme.

This function is usually used in conjunction with the PathCheck-class function `require-auth`.

**Parameters**

`auth-type` specifies the type of authorization to be used. This should always be `"basic"`.

`userdb` specifies the full path and file name of the user database. This parameter will be passed to the user function.

`userfn` is the name of the user custom function that must have been previously loaded with `load-modules`. It has the same interface as all the SAFs, but it is called with the user name (`user`), password (`pw`), user database (`userdb`), and group database (`groupdb`) if supplied, in the `pb` parameter. The user function should check the name and password using the database and return `REQ_NOACTION` if they are not valid. It should return `REQ_PROCEED` if the name and password are valid. The basic-auth function will then add `auth-type`, `auth-user` (`user`), `auth-db` (`userdb`), and `auth-password` (`pw`, Windows NT only) to the `rq->vars pblock`.

`groupdb` (optional) specifies the full path and file name of the user database. This parameter will be passed to the group function.

`groupfn` (optional) is the name of the group custom function that must have been previously loaded with `load-modules`. It has the same interface as all the SAFs, but it is called with the user name (user), password (pw), user database (`userdb`), and group database (`groupdb`) in the pb parameter. It also has access to the `auth-type`, `auth-user` (`user`), `auth-db` (`userdb`), and `auth-password` (`pw`, Windows NT only) parameters in the `rq->vars pblock.` The group function should determine the user's group using the group database, add it to `rq->vars` as `auth-group`, and return `REQ_PROCEED` if found. It should return `REQ_NOACTION` if the user's group is not found.

**Examples**

```
Init fn=load-modules shlib=/path/to/mycustomauth.so
   funcs=hardcoded_auth
...
AuthTrans fn=basic-auth auth-type=basic userfn=hardcoded_auth
   userdb=unused
```

**See Also** `require-auth`

## basic-ncsa

The `basic-ncsa` function verifies authorization information sent by the client against a database. The Authorization header is sent as part of the the "basic" server authorization scheme.

This function is usually used in conjunction with the PathCheck-class function `require-auth`.

**Parameters**

`auth-type` specifies the type of authorization to be used. This should always be "`basic`".

`dbm` (optional) specifies the full path and base file name of the user database in the server's native format. The native format is a system DBM file, which is a hashed file format allowing instantaneous access to billions of users. If you use this parameter, don't use the `userfile` parameter as well.

`userfile` (optional) specifies the full path name of the user database in the NCSA-style HTTPD user file format. This format consists of lines using the format *`name`*`:`*`password`*, where *`password`* is encrypted. If you use this parameter, don't use `dbm`.

`grpfile` (optional) specifies the NCSA-style HTTPD group file to be used. Each line of a group file consists of *`group`*`:`*`user1 user2 ... userN` where each user is separated by spaces.

**Examples**

```
AuthTrans fn=basic-ncsa auth-type=basic
   dbm=/netscape/suitespot/userdb/rs
AuthTrans fn=basic-ncsa auth-type=basic
   userfile=/netscape/suitespot/.htpasswd
   grpfile=/netscape/suitespot/.grpfile
```

**See Also** `require-auth`

# NameTrans directive

`NameTrans` stands for Name Translation. This directive translates virtual URLs to physical directories on your server. For example, the URL

```
http://www.test.com/some/file.html
```

could be translated to the full file-system path

```
/usr/netscape/suitespot/docs/some/file.html
```

NameTrans directives should appear in the “default” object. If there is more than one NameTrans directive in an object, the server executes each functions in order until one succeeds (by returning REQ_PROCEED).

The following NameTrans-class functions are described in detail in this section:

- `assign-name` enables additional named object directives.
- `document-root` translates a URL into a file system path by prepending the path of the document root directory.
- `home-page` translates a request for the server’s root home page (`/`) to a specific file.
- `mozilla-redirect` redirects the client to a different URL if it is a Netscape browser.
- `pfx2dir` translates any URL beginning with a given prefix to a file system directory and optionally enables additional named object directives.
- `redirect` redirects the client to a different URL.
- `unix-home` translates a URL to a specified directory withing a user’s home directory.

## assign-name

The `assign-name` function associates the name of a configuration object with a path specified by a shell expression. It always returns `REQ_NOACTION`.

**Parameters** `from` is a wildcard pattern that specifies the path to be affected.

`name` is the name of the additional configuration object to enable for the current request.

**Example**

```
# This NameTrans directive is in the default object.
NameTrans fn=assign-name name=personnel from=/httpd/docs/pers
...
<Object name=personnel>
...additional directives..
</Object>
```

## document-root

The `document-root` function specifies the directory that contains your documents. If the path has not been set by a previous function, this path is prepended to the virtual path that the client sent to create the full path name of the file or directory. For example, the client requests `/a/b/file.html`, which is translated to `/usr/netscape/suitespot/docs/a/b/file.html`.

This function always returns `REQ_PROCEED`. NameTrans directives listed after this will never be called.

**Parameters** `root` is the file system path to the server's document root directory.

**Examples**

```
NameTrans fn=document-root root=/usr/netscape/suitespot/docs
```

## home-page

The `home-page` function specifies the home page for your server. Whenever a client requests the server's home page (`/`), they'll get the document specified.

**Parameters** `path` is the path and name of the home page file. If `path` starts with a slash (`/`), it is assumed to be a full path to a file. This function sets the server's `path` variable and returns `REQ_PROCEED`. If `path` does not start with a slash (`/`), it is appended to the URI and and the function returns `REQ_NOACTION` continuing on to the other NameTrans directives.

**Examples**

```
NameTrans fn=home-page path=homepage.html
```

```
NameTrans fn=home-page path=/httpd/docs/home.html
```

## mozilla-redirect

The `mozilla-redirect` function (similar to the `redirect` function) lets you change URLs and send the updated URL to the client for any user who is using the Netscape Navigator, version 0.96 or later.

**Parameters** `from` specifies the URI wildcard pattern which should be redirected.

`url` specifies a complete URL to return to the client.

**Example**

```
NameTrans fn=mozilla-redirect from=/ url=http://newserver/
```

## pfx2dir

`The pfx2dir` function looks for a directory prefix in the path and replaces the prefix with a real directory name.

**Parameters** `from` is the URI prefix to convert. It should not have a trailing slash (/).

`dir` is the local file system directory path that the prefix is converted to. It should not have a trailing slash (/).

`name` (optional) gives a named object (template) from which to derive additional configuration for this request.

**Examples**

```
NameTrans fn=pfx2dir from=/cgi-bin dir=/httpd/cgi-bin name=cgi
NameTrans fn=pfx2dir from=/icons dir=/httpd/mc-icons
```

## redirect

The `redirect` function lets you change URLs and send the updated URL to the client. When a client accesses your server with an old path, they are told to use the new URL you provide.

**Parameters** `from` specifies the prefix of the requested URI to match.

`url` (optional) specifies a complete URL to return to the client. If you use this parameter, don't use `url-prefix` (and vice-versa).

`url-prefix` (optional) is the new URL prefix to return to the client. The `from` prefix is simply replaced by this URL prefix.

escape (optional) is a flag which tells the server to `util_url_escape` the URL before sending it. It should be `yes` or `no`. The default is `yes`.

**Examples**
```
NameTrans fn=redirect from=/ url-prefix=http://tmpserver
NameTrans fn=redirect from=/toopopular
   url=http://bigger/better/stronger/morepopular/new.html
```

## unix-home

**Unix Only** The `unix-home` function translates user names (typically of the form ~username) into the user's home directory on the server's Unix machine. You specify a URL prefix that signals user directories. Any request that begins with the prefix is translated to the user's home directory.

You specify the list of users with either the `/etc/passwd` file or a file with a similar structure. Each line in the file should have this structure (elements in the `passwd` file that aren't needed are indicated with *):

```
username:*:*:groupid:*:homedir:*
```

If you want the server to scan the password file only once at startup, use the Init-class function `init-uhome`.

**Parameters** `from` is the URL prefix to translate, usually `"/~"`.

`subdir` is the subdirectory within the user's home directory that contains their web documents.

`pwfile` (optional) is the full path and file name of the password file if it's different from `/etc/passwd`.

`name` (optional) specifies an additional named object whose directives will be applied to this request.

**Examples**
```
NameTrans fn=unix-home from=/~ subdir=public_html
NameTrans fn=unix-home from /~ pwfile=/mydir/passwd
   subdir=public_html
```

**See Also** `init-uhome, find-links`

# PathCheck directive

`PathCheck` directives check the local file system path that is returned after the `NameTrans` step. The path is checked for things such as CGI path info and for dangerous elements such as `/./`and `/../` and `//`, and then any access restriction is applied.

If there is more than one `PathCheck` directive, each of the functions are executed in order.

The following PathCheck-class functions are described in detail in this section:

- `cert2user` determines the authorized user from the client certificate.
- `check-acl` checks an access control list for authorization.
- `deny-existence` indicates that a resoure was not found.
- `find-index` locates a default file when a directory is requested.
- `find-links` denies access to directories with certain file system links
- `find-pathinfo` locates extra path info beyond the file name for the PATH_INFO CGI environment variable.
- `get-client-cert` gets the authenticated client certificate from the SSL3 session.
- `load-config` finds and loads extra configuration from a file in the requested path
- `nt-uri-clean` denies access to requests with unsafe path names by indicating not found.
- `ntcgicheck` looks for a CGI file with a specified extension.
- `require-auth` denies access to unauthorized users or groups.
- `unix-uri-clean` denies access to requests with unsafe path names by indicating not found.

## cert2user

The `cert2user` function maps the authenticated client certificate from the SSL3 session to a user name, using the certificate-to-user mappings in the user database specified by `userdb`.

**Parameters** `userdb` names the user database from which to obtain the certificate.

`makefrombasic` tells the function to establish a certificate-to-user mapping. If `makefrombasic` is present and is not `0`, the directive uses basic password authentication to authenticate the user and to then create a new certificate-to-user mapping in the specified user database if no such mapping has already been created there.

The server allows the certificate-to-user mapping to be created automatically by:

7. Obtaining and verifying a certificate from the user
8. Obtaining a user name and password using WWW basic authentication.
9. Creating a mapping from that certificate to that user (provided both check out ok).

`require` governs the return value. If the certificate cannot be mapped successfully to a user name, and the value of `require` is `0`, the function returns `REQ_NOACTION` allowing the processing of the request to continue. But if the value of `require` is not `0`, the function returns `REQ_ABORTED` and sets the protocol status to `403 FORBIDDEN,` causing the request to fail and the client to be given the `FORBIDDEN` status. The default value of `require` is `1`.

`method` specifies a wildcard pattern for the HTTP methods for which this function will be applied. If `method` is absent, the function is applied for any method.

**Examples**

```
# Map the client cert to a user using this userdb.
# If a mapping is not present, fail the request.
PathCheck fn="cert2user"
    userdb="/usr/netscape/suitespot/authdb/default"
    require="1"
```

## check-acl

The `check-acl` function attaches an Access Control List to the object in which the directive appears. Regardless of the order of `PathCheck` directives in the object, `check-acl` functions are executed first, and will cause user authentication to be performed, if required by the specified ACL, and will also update the access control state.

**Parameters** `acl` is the name of an Access Control List.

`shexp` (optional) is a wildcard pattern that specifies the path for which to apply the ACL.

`bong-file` (optional) is the path name for a file that will be sent if this ACL is responsible for denying access.

**Examples**

```
PathCheck fn=check-acl acl="HRonly"
```

## deny-existence

The `deny-existence` function sends a “not found” message when a client tries to access a specified path. The server sends “not found” instead of “forbidden,” so the user can’t tell whether the path exists or not.

Use this function inside a <Client> block to deny the existence of a resource to specific users. For example, these lines deny existence of all resources to any user not in the netscape.com domain:

```
<Client dns=*~.netscape.com>
PathCheck fn=deny-existence
</Client>
```

**Parameters** `path` (optional) is a wildcard pattern of the file-system path to hide. If the path does not match, the function does nothing and returns `REQ_NOACTION`. If the path is not provided, it is assumed to match.

`bong-msg` (optional) specifies a file to send rather than responding with the "not found" message. It is a full file-system path.

**Examples**

```
PathCheck fn=deny-existence path=/usr/netscape/suitespot/docs/private
PathCheck fn=deny-existence bong-msg=/svr/msg/go-away.html
```

## find-index

The `find-index` function investigates whether the requested path is a directory. If it is, the function searches for an index file in the directory, and then changes the path to point to the index file. If no index file is found, the server generates a directory listing.

This function does nothing if there is a query string, if the HTTP method is not GET, or if the path is that of a valid file.

If the URI is a directory and it does not end with /, the function creates a "URI" parameter in rq->vars by adding / to the end of the requested URI, sets the HTTP response code to PROTOCOL_REDIRECT, and returns REQ_ABORTED. The server

**Parameters** `index-names` is a comma-separated list of index file names to look for. Use spaces only if they are part of a file name. Do not include spaces before or after the commas. This list is case-sensitive if the file system is case-sensitive.

**Examples**

```
PathCheck fn=find-index index-names=index.html,home.html
```

## find-links

**Unix Only** The `find-links` function searches the current path for symbolic or hard links to other directories or file systems. If any are found, an error is returned. This function is normally used for directories that are not trusted (such as user home directories). It prevents someone from pointing to information that should not be made public.

**Parameters** `disable` is a character string of links to disable:

- `h` is hard links
- `s` is soft links
- `o` allows symbolic links from user home directories only if the user owns the target of the link.

`dir` is the directory to begin checking. If you specify an absolute path, any request to that path and its subdirectories is checked for symbolic links. If you specify a partial path, any request containing that partial path is checked for symbolic links. For example, if you use `/user/` and a request comes in for `some/user/directory`, then that directory is checked for symbolic links.

**Examples**

```
PathCheck fn=find-links disable=sh dir=/foreign-dir
```

```
PathCheck fn=find-links disable=so dir=public_html
```

**See Also** `init-uhome, unix-home`

## find-pathinfo

The `find-pathinfo` function finds any extra path info after the file name in the URL and stores it for use in the CGI environment variable PATH_INFO.

**Parameters** None.

**Examples**
```
PathCheck fn=find-pathinfo
```

## get-client-cert

The `get-client-cert` function gets the authenticated client certificate from the SSL3 session. It can apply to all HTTP methods, or only to those that match a specified pattern. It only works when SSL is enabled on the server.

If the certificate is present or obtained from the SSL3 session, the function returns `REQ_NOACTION`, allowing the request to proceed, otherwise it returns `REQ_ABORTED` and sets the protocol status to `403 FORBIDDEN`, causing the request to fail and the client to be given the `FORBIDDEN` status.

**Parameters** `dorequest` controls whether to actually try to get the certificate, or just test for its presence. If `dorequest` is absent the default value is `0`.

- `1` tells the function to redo the SSL3 handshake and get a certificate. When the client is Netscape Navigator, this request triggers a dialog that lets the client select a certificate to present.
- `0` tells the function to just test for the presence of a certificate in the parameter `auth-cert` in the `Request->vars` parameter block.

If a certificate is obtained from the client and verified successfully by the server, the ASCII base64 encoding of the DER-encoded X.509 certificate is placed in the parameter `auth-cert` in the `Request->vars` pblock, and the function returns `REQ_PROCEED`, allowing the request to proceed.

`method` (optional) specifies a wildcard pattern for the HTTP methods for which the function will be applied. If `method` is absent, the function is applied to all requests.

**Examples**
```
# Get the client cert from the session, request one if
# there is no talready one associated with the session.
# Fail the request if the client does not present a
# valid one.
PathCheck fn="get-client-cert" dorequest="1"
```

## load-config

The `load-config` function searches for configuration files in document directories and adds the files contents to the server's existing configuration.

**Parameters**

`file` (optional) is the name of the file to search for in the document directory. If not provided, the file name is assumed to be `".nsconfig"`.

`disable-types` (optional) specifies a wildcard pattern of types to disable for the base directory. For example, `magnus-internal/cgi`. Requests for resources matching these types are aborted.

`descend` (optional) if present, specifies that the server should search in subdirectories of this directory for configuration files. For example, `descend=1` specifies that the server should search subdirectories. No `descend` parameter specifies that the function should search only the base directory.

`basedir` (optional) specifies a directory in which to look for configuration files. If not specified, the server will start from the directory that was specified in name translation (for example, when accessing `/a/b/home.html`, the filesystem path might be `/docroot/a/b/home.html` and the name translation directory would be `/docroot`).

Returns `REQ_PROCEED` if config files were loaded, `REQ_ABORTED` on error, or `REQ_NOACTION` when no files are loaded.

## nt-uri-clean

**Windows NT Only**

The `nt-uri-clean` function denies access to any resource whose physical path contains `\.\`, `\..\` or `\\` (these are potential security problems).

**Parameters**

None.

**Examples**

```
PathCheck fn=nt-uri-clean
```

**See Also**

`unix-uri-clean`

## ntcgicheck

**Windows NT Only**

The `ntcgicheck` function specifies the file name extension to be added to any file name that does not have an extension, or to be substituted for any file name that has the extension `.cgi`.

**Parameters**

`extension` is the replacement file extension.

**Examples**

```
PathCheck fn=ntcgicheck extension=pl
```

**See Also** `init-cgi, send-cgi, send-wincgi, send-shellcgi`

## require-auth

The `require-auth` function allows access to objects only if the user or group is authorized. Before this function is called, an authorization function (such as `basic-auth`) must be called in an AuthTrans directive.

**Parameters** `path` (optional) is a wildcard local file system path on which this function should operate. If no path is provided, the function applies to all paths.

`auth-type` is the type of HTTP authorization used and must match the auth-type from the previous authorization function in AuthTrans. Currently, `basic` is the only authorization type defined.

`realm` is a string sent to the browser indicating the secure area (or realm) for which a user name and password are requested.

`auth-user` (optional) specifies a wildcard list of users who are allowed access. If this parameter is not provided, then any user authorized by the authorization function is allowed access.

`auth-group` (optional) specifies a wildcard list of groups that are allowed access.

If a USER was authorized in an AuthTrans directive, and the `auth-user` parameter is provided, then the user's name must match the `auth-user` wildcard value. Also, if the `auth-group` parameter is provided, the authorized user must belong to an authorized group which must match the `auth-user` wildcard value.

**Examples**

```
PathCheck fn=require-auth auth-type=basic realm="Marketing Plans"
   auth-group=mktg auth-user=(jdoe|johnd|janed)
```

**See Also** `basic-auth, basic-ncsa`

## unix-uri-clean

**Unix Only** The `unix-uri-clean` function denies access to any resource whose physical path contains `/./`, `/../` or `//` (these are potential security problems).

**Parameters** None.

**Examples** `PathCheck fn=unix-uri-clean`

**See Also** `nt-uri-clean`

# ObjectType directive

`ObjectType` directives determine the MIME type of the file sent to the client. This type is usually sent back to the client to let the client decide what to do. MIME attributes currently sent are `type`, `encoding`, and `language`.

If there is more than one `ObjectType` directive in an object, all of the directives are applied in the order they appear. If a directive sets an attribute and a later directive tries to set that attribute to something else, the first setting is used and the subsequent ones ignored.

The following ObjectType-class functions are described in detail in this section:

- `force-type` sets the content-type header for the response to a specific type.
- `image-switch` attempts to return a .jpg file in place of a .gif file for Netscape browsers.
- `shtml-hacktype` requests that .htm and .html files are parsed for server-parsed html commands.
- `type-by-exp` sets the content-type header for the response based on the requested path.
- `type-by-extension` sets the content-type header for the response based on the files extension and the MIME types database.

## force-type

The `force-type` function assigns a type to objects that do not already have a MIME type. This is used to specify a default object type.

**Parameters** `type` (optional) is the type assigned to a matching request (the "`content-type`" header).

`enc` (optional) is the encoding assigned to a matching request (the "`content-encoding`" header).

`lang` (optional) is the language assigned to a matching request (the "`content-language`" header).

`charset` (optional) is the character set for the `magnus-charset` parameter in `rq->srvhdrs`. If the browser sent the `Accept-charset` header or its `User-agent` is mozilla/1.1 or newer, then append `"; charset=<charset>"` to content-type, where `<charset>` is the value of the `magnus-charset` parameter in `rq->srvhdrs`.

**Examples**

```
ObjectType fn=force-type type=text/plain
```

```
ObjectType fn=force-type lang=en_US
```

**See Also** `load-types, type-by-extension, type-by-exp`

## shtml-hacktype

The `shtml-hacktype` function changes the content-type of any .htm or .html file to "magnus-internal/parsed-html" and returns REQ_PROCEED. This provides backward compatibility with server-side includes for files with .htm or .html extensions. The function may also check the execute bit for the file on Unix systems. This function is not recommended.

**Parameters** `exec-hack` (Unix only, optional) tells the function to change the content-type only if the execute bit is enabled. The value of the parameter is not important. It need only be provided. You may use `exec-hack=true`.

**Examples**

```
ObjectType fn=shtml-hacktype exec-hack=true
```

## type-by-exp

The `type-by-exp` function matches the current path with a wildcard expression. If the two match, the type parameter information is applied to the file. This is the same as type by extension, except you use wildcard patterns for the files or directories specified in the URLs.

**Parameters** `exp` is the wildcard pattern of paths for which this function is applied.

`type` (optional) is the type assigned to a matching request (the "`content-type`" header).

`enc` (optional) is the encoding assigned to a matching request (the "`content-encoding`" header).

`lang` (optional) is the language assigned to a matching request (the "`content-language`" header).

`charset` (optional) is the character set for the `magnus-charset` parameter in `rq->srvhdrs`. If the browser sent the `Accept-charset` header or its `User-agent` is mozilla/1.1 or newer, then append `"; charset=<charset>"` to content-type, where `<charset>` is the value of the `magnus-charset` parameter in `rq->srvhdrs`.

**Examples** `ObjectType fn=type-by-exp exp=*.test type=application/html`

**See Also** `load-types, type-by-extension, force-type`

### type-by-extension

The `type-by-extension` function uses file extensions to determine information about files. (extensions are strings after the last period in a file name.) This matches an incoming request to extensions in the `mime.types` file. The MIME type is added to the "`content-type`" header sent back to the client. The type may be set to an internal server type begining with "magnus-internal/" which directs the server to a special Service function. If the current request is for a directory, the content-type is set to "magnus-internal/directory".

**Parameters** None.

**Examples** `ObjectType fn=type-by-extension`

**See Also** `load-types, type-by-exp, force-type`

## Service directive

The Service class of functions sends the response data and completes the request. This is similar to a CGI program, but much faster.

Every `Service` directive has the following optional parameters to determine whether the function is executed. All of the optional parameters must match the current request for the function to be executed.

- `type` (optional) specifies a wildcard pattern of MIME types for which this function will be executed. The "magnus-internal/*" MIME types are used only to select a Service-class function to execute.

- `method` (optional) specifies a wildcard pattern of HTTP methods for which this function will be executed. Common HTTP methods are GET, HEAD, and POST.

- `query` (optional) specifies a wildcard pattern of query strings for which this function will be executed.

If there is more than one Service-class function, the first one matching the optional parameters above is executed.

The following Service-class functions are described in detail in this section:

- `append-trailer` appends text to the end of all HTML files.
- `IIOPexec` executes a Web Application Service.
- `IIOPNameService` handles name service requests for the Web Application Interface (WAI).
- `image-map` handles server-side image maps.
- `index-common` generates a fancy HTML response representing the files and directories in a requested directory.
- `index-simple` generates a simple HTML response representing the files and directories in a requested directory.
- `key-toosmall` indicates to the client that the provided certificate key size is too small to accept.
- `list-dir` lists the contents of a directory.
- `make-dir` creates a directory.
- `parse-html` parses an HTML file for server-parsed html commands and sends it to the client.
- `query-handler` handles the HTML ISINDEX tag.
- `remove-dir` deletes an empty directory.
- `remove-file` deletes a file.
- `rename-file` renames a file.
- `send-cgi` sets up environment variables, launches a CGI program, and sends the reponse to the client.
- `send-error` sends an HTML file to the client in place of a standard HTTP response status.
- `send-file` sends a local file to the client.
- `send-range` sends a range of bytes of a file to the client.
- `send-shellcgi` sets up environment variables, launches a shell CGI program, and sends the reponse to the client.

- `send-wincgi` sets up environment variables, launches a WinCGI program, and sends the reponse to the client.
- `upload-file` uploads and saves a file.

## append-trailer

The `append-trailer` function sends an HTML file and appends text to the end. It only appends text to HTML fies. This is typically used for author information and copyright text. The date the file was last modified can be inserted.

Returns REQ_ABORTED if a required parameter is missing, if there is extra path info after the file name in the URL, or if the file cannot be opened for read-only access.

**Parameters** `trailer` is the text you want to append to all HTML documents. The string `:LASTMOD:` is replaced by the date the file was last modified; you must also specify a time format with `timefmt`. The string is unescaped with `util_uri_unescape` before being sent. The text can contain HTML tags and can be up to 512 characters long after unescaping and inserting the date.

`timefmt` (optional) is a time format string for `:LASTMOD:`. For details about time formats refer to Appendix C, “Time Formats”. If `timefmt` is not provided, `:LASTMOD:` will not be replaced with the time.

**Examples**

```
Service type=text/html method=GET fn=append-trailer
   trailer="<hr><img src=/logo.gif> Copyright 1995"

# add the trailer with the date in the format: MM/DD/YY
Service type=text/html method=GET fn=append-trailer
   timefmt="%D"
   trailer="<hr>File last updated on: :LASTMOD:"
```

## IIOPexec

Accesses a Web Application Service (with Web Application Interface) to fulfill a request.

**Parameters** `name -` overrides the instance name taken from the URI.

**Examples**

```
Service fn="IIOPexec" name="myinstance"
```

**See Also** `IIOPinit, IIOPNameService`

## IIOPNameService

Provides basic CosNaming-compatible name service to support WAI.

**Parameters** None.

**Examples** `Service fn=IIOPNameService`

**See Also** `IIOPinit, IIOPexec`

## imagemap

The `imagemap` function includes imagemap files.

**Parameters** None.

**Examples**
```
Service type=magnus-internal/imagemap method=(GET|HEAD)
fn=imagemap
```

## index-common

The `index-common` function scans a directory and returns an HTML page to the browser displaying a fancy list of the files/directories in the directory. The list is sorted alphabetically. Files beginning with a period (.) are not displayed. Each item appears as an HTML link. This function displays more information than `index-simple` including the size, date last modified, and an icon for each file. It may also include a header and/or readme file into the listing.

The icons displayed are .gif files dependent on the content-type of the file:

```
"text/*"      text.gif
"image/*"     image.gif
"audio/*"     sound.gif
"video/*"     movie.gif
"application/octet-stream" binary.gif
directory     menu.gif
all others    unknown.gif
```

**Parameters** `header` (optional) is the path (relative to the directory being indexed) and name of a file (HTML or plain text) which is included at the beginning of the directory listing to introduce the contents of the directory. The file is first tried

with ".html" added to the end. If found, it is incorporated near the top of the directory list as HTML. If the file is not found, then it is tried without the ".html" and incorporated as preformatted plain text (bracketed by <PRE> and </PRE>).

`readme` (optional) is the path (relative to the directory being indexed) and name of a file (HTML or plain text) to append to the directory listing. This file might give more information about the contents of the directory, indicate copyrights, authors, or other information. The file is first tried with ".html" added to the end. If found, it is incorporated at the bottom of the directory list as HTML. If the file is not found, then it is tried without the ".html" and incorporated as preformatted plain text (bracketed by <PRE> and </PRE>).

**Examples**

```
Service fn=index-common type=magnus-internal/directory
   method=(GET|HEAD) header=hdr readme=rdme.txt
```

**See Also** `cindex-init, index-simple`

## index-simple

The `index-simple` function scans a directory and returns an HTML page to the browser displaying a bulleted list of the files/directories in the directory. The list is sorted alphabetically. Files beginning with a period (.) are not displayed. Each item appears as an HTML link.

**Parameters** None.

**Examples**

```
Service type=magnus-internal/directory fn=index-simple
```

**See Also** `cindex-init, index-common`

## key-toosmall

The `key-toosmall` function returns a message to the client specifying that the secret key size for SSL communications is too small. This function is designed to be used together with a Client tag to limit access of certain directories to non-exportable browsers.

**Parameters** None.

**Examples**

```
<Object ppath=/mydocs/secret/*>
<Client secret-keysize=40)
Service fn=key-toosmall
</Client>
```

```
</Object>
```

## list-dir

The `list-dir` function returns a sequence of text lines to the client in response to an HTTP INDEX method. The format of the returned lines is:

*name type size* **mimetype**

The *`name`* field is the name of the file or directory. It is relative to the directory being indexed. It is URL-encoded, that is, any character might be represented by `%xx`, where `xx` is the hexadecimal representation of the character's ASCII number.

The *`type`* field is a MIME type such as `text/html`. Directories will be of type `directory`. A file for which the server doesn't have a type will be of type `unknown`.

The *`size`* field is the size of the file, in bytes.

The *`mtime`* field is the numerical representation of the date of last modification of the file. The number is the number of seconds since the epoch (Jan 1, 1970 00:00 UTC) since the last modification of the file.

**Parameters** None.

**Examples**

```
Service fn=list-dir
```

## make-dir

The `make-dir` function creates a directory when the client sends an HTTP MKDIR command. The function can fail if the server can’t write to that directory.

**Parameters** None.

**Examples**

```
Service fn=make-dir
```

## parse-html

The `parse-html` function parses an HTML document, scanning for embedded commands. These commands may provide information from the server, include the contents of other files, or execute a CGI program. Parsing HTML documents will reduce server performance. Refer to Appendix D, “Server-Parsed HTML” for server-parsed HTML commands.

**Parameters** `opts` (optional) are parsing options. The `no-exec` option is the only currently available option—it disables the `exec` command.

**Examples**

```
Service type=magnus-internal/parsed-html
        method=(GET|HEAD) fn=parse-html
```

## query-handler

The `query-handler` function runs a CGI program instead of referencing the path requested. This is used mainly to support the obsolete ISINDEX tag . If possible, use an HTML form instead.

**Parameters** `path` is the full path and file name of the CGI program to run.

**Examples**

```
Service query=* fn=query-handler path=/http/cgi/do-grep
```

```
Service query=* fn=query-handler path=/http/cgi/proc-info
```

## remove-dir

The `remove-dir` function removes a directory when the client sends an HTTP RMDIR method. The directory must be empty (have no files in it). The function will fail if the directory is not empty or if the server doesn’t have the priveleges to remove the directory.

**Parameters** None.

**Examples**

```
Service fn=remove-dir
```

## remove-file

The `remove-file` function deletes a file when the client sends an HTTP DELETE method. It deletes the file indicated by the URL if the user is authorized and the server has the needed file system priveleges.

**Parameters** None.

**Examples**
```
Service fn=remove-file
```

## rename-file

The `rename-file` function renames a file when the client sends an HTTP `MOVE` method and a `New-URL` header. It renames the file indicated by the URL to `New-URL` within the same directory.

**Parameters** None.

**Examples**
```
Service fn=rename-file
```

## send-cgi

The `send-cgi` function sets up the CGI environment variables, runs a file as a CGI program in a new process, and sends the results to the client.

For details about the CGI environment variables and their NSAPI equivalents refer to Chapter 3, “Creating Custom SAFs”.

**Parameters** None.

**Examples**
```
Service fn=send-cgi

Service type=magnus-internal/cgi fn=send-cgi
```

## send-file

The `send-file` function sends the contents of the requested file to the client. It provides the `content-type`, `content-length`, and `last-modified` headers. Most requests are handled by this function.

**Parameters** None.

**Examples**
```
Service type=*~magnus-internal/* method=(GET|HEAD)
        fn=send-file
```

## send-range

When the client requests a portion of a document, by specifying HTTP byte ranges, the `send-range` function returns that portion.

**Parameters** None.

**Examples**
```
Service fn=send-range
```

### send-shellcgi

**Windows NT only** The `send-shellcgi` function runs a file as a shell CGI program and sends the results to the client. Shell CGI is a server configuration that lets you run CGI applications using the file associations set in Windows NT. For information about shell CGI programs, consult the Administrator's Guide.

**Parameters** None.

**Examples**
```
Service fn=send-shellcgi
```

```
Service type=magnus-internal/cgi fn=send-shellcgi
```

### send-wincgi

**Windows NT only** The `send-cgi` function runs a file as a Windows CGI program and sends the results to the client. For information about Windows CGI programs, consult the Administrator's Guide.

**Parameters** None.

**Examples**
```
Service fn=send-wincgi
```

```
Service type=magnus-internal/cgi fn=send-wincgi
```

### upload-file

The `upload-file` function uploads and saves a new file when the client sends an HTTP PUT method.

**Parameters** None.

**Examples**
```
Service fn=upload-file
```

## AddLog directive

After the server has responded to the request, the AddLog directives are executed to record information about the transaction.

If there is more than one `AddLog` directive, all are executed.

The following AddLog-class functions are described in detail in this section:

- `common-log` records information about the request in the common log format.
- `flex-log` records information about the request in a flexible, configurable format.
- `record-useragent` records the client's ip address and user-agent header.

## common-log

The `common-log` function is an `AddLog`-class function that records request-specific data in the common log format (used by most HTTP servers). There is a log analyzer in the `/extras/log_anly` directory. There are also a number of free statistics generators for the common log format.

**Parameters**

`name` (optional) gives the name of a log file, which must have been given as a parameter to the `init-clf` Init function. If no name is given, the entry is recorded in the global log file.

`iponly` (optional) instructs the server to log the IP address of the remote client rather than looking up and logging the DNS name. This will improve performance if DNS is off in the `magnus.conf` file. The value of `iponly` has no significance, as long as it exists; you may use `iponly=1`.

**Examples**

```
# Log all accesses to the global log file
AddLog fn=common-log

# Log accesses from outside our subnet (198.93.5.*) to nonlocallog
<Client ip="*~198.93.5.*">
AddLog fn=common-log name=nonlocallog
</Client>
```

## flex-log

The `flex-log` function is an `AddLog`-class function that records request-specific data in a flexible log format. It may also record requests in the common log format. There is a log analyzer in the `/extras/log_anly` directory. There are also a number of free statistics generators for the common log format.

The log format is specified by the `flex-init` function call.

**Parameters** `name` (optional) gives the name of a log file, which must have been given as a parameter to the `init-clf` Init function. If no name is given, the entry is recorded in the global log file.

`iponly` (optional) instructs the server to log the IP address of the remote client rather than looking up and logging the DNS name. This will improve performance if DNS is off in the magnus.conf file. The value of `iponly` has no significance, as long as it exists; you may use `iponly=1`.

**Examples**

```
# Log all accesses to the global log file
AddLog fn=flex-log

# Log accesses from outside our subnet (198.93.5.*) to nonlocallog
<Client ip="*~198.93.5.*">
AddLog fn=flex-log name=nonlocallog
</Client>
```

**See Also** `flex-init, init-clf, common-log, record-useragent`

### record-useragent

The `record-useragent` function records the IP address of the client, followed by its User-Agent HTTP header. This tells what version of Netscape Navigator (or other client) was used for this transaction.

**Parameters** `name` (optional) gives the name of a log file, which must have been given as a parameter to the `init-clf` Init function. If no name is given, the entry is recorded in the global log file.

**Examples**

```
# Record the client ip address and user-agent to browserlog
AddLog fn=record-useragent name=browserlog
```

**See Also** `init-clf, common-log, record-useragent, flex-init, flex-log`

## Error directive

At any time during a request, conditions may occur that cause the server to stop fulfilling a request and return an "error" (HTTP response status code) to the client. When this occurs, a SAF will set the HTTP response status code and return `REQ_ABORTED`. The server will then search for an Error directive matching the HTTP response status code or its associated reason phrase, and execute the directive's function.

Like `Service` directives, every `Error` directive has optional parameters to determine whether the function is executed. If any of the optional parameters match the current request, the function is executed. If none of the optional paramters are given, the function is executed.

- `reason` (optional) is the text of one of the reason strings (such as “Unauthorized” or “Forbidden”). The string is not case sensitive.
- `code` (optional) is a three-digit number representing the HTTP response status code, such as 401 or 407.

This can be any HTTP response status code or reason phrase according to the HTTP specification. The following is a list of common HTTP response status codes and reason strings.

- `401 Unauthorized`.
- `403 Forbidden`.
- `404 Not Found`.
- `500 Server Error`.

The following Error-class functions are described in detail in this section:

- `send-error` sends an HTML file to the client in place of a specific HTTP response status.

## send-error

The `send-error` function sends an HTML file to the client in place of a specific HTTP response status. This allows the server to present a friendly message describing the problem. The HTML page may contain images and links to the server’s home page or other pages.

**Parameters** `path` specifies the full file system path of an HTML file to send to the client. The file is sent as "text/html" regardless of its name or actual type. If the file does not exist, the server sends a simple default error page.

**Examples**

```
Error fn=send-error code=401
      path=/netscape/suitespot/docs/errors/401.html

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

uses language if "AcceptLanguage on" in magnus.conf and browser sends language header. So do pfx2dir and document-root!!! (plus ContentMgr, WebPublishing, and AgentAPI, and Search API)

```
* Example:
   *    filePath               = "/path/$$LANGDIR/filename.ext"
   *    language               = "language"
   *    GetDefaultLanguage() --> "default"
   *    LANG_DELIMIT           = "_"
   *
   * 1. Try: "/path/language/filename.ext"
   * 2. Try: "/path/filename_language.ext"
   * 3. Try: "/path/default/filename.ext"
   * 4. Try: "/path/filename_default.ext"
   * 5. Try: "/path/filename.ext"
   *   else: ""
   *
   * Example:
   *    language               = "en-us;q=0.6,ja;q=0.8,en-ca"
   *
   * 1. Try: "/path/en-ca/filename.ext"
   * 2. Try: "/path/filename_en_ca.ext"
   * 3. Try: "/path/ja/filename.ext"
   * 4. Try: "/path/filename_ja.ext"
   * 5. Try: "/path/en_us/filename.ext"
   * 6. Try: "/path/filename_en_us.ext"
   * 7. Try: "/path/default/filename.ext"
   * 8. Try: "/path/filename_default.ext"
   * 9. Try: "/path/filename.ext"
   *   else: ""
Using the accept language header
When clients contact a server using HTTP 1.1, they can send header
information that describes the various languages they accept. You can
configure your server to parse this language information.
For example, suppose this feature is set to on, and a client configured
to send
the accept language header sends it with the value en,fr. Now suppose
that
the client requests the following URL:
http://www.someplace.com/somepage.html
The server first looks for:
http://www.someplace.com/en/somepage.html
If it does not find that, it looks for:
```

http://www.someplace.com/fr/somepage.html
If that is not available either, and a ClientLanguage (call it xx) is defined in
the magnus.conf file, the server tries:
http://www.someplace.com/xx/somepage.html
If none of these exist, the server tries:
http://www.someplace.com/somepage.html
For information about configuring the server to parse the accept language
header, see "Parsing the accept language header" on page 55.

Language settings in configuration files

The following directives in the magnus.conf file affect languages:

International settings in magnus.conf

| File | Directive | Values | Description |
|---|---|---|---|
| magnus.conf | ClientLanguage | en, fr, de, ja | Specifies the language in which client messages, such as "Not Found" or "Access denied" are to be expressed. This value is used to identify a directory containing ns-https.db. |
| magnus.conf | DefaultLanguage | en, fr, de, ja | Specifies the language used if a resource cannot be found for the client language or the administration language. |
| magnus.conf | AcceptLanguage | on, off | Enables or disables the Accept language header parsing. |

The following directives in the ns-admin.conf file affect languages:

International settings in ns-admin.conf

| File | Directive | Values | Description |
|---|---|---|---|
| ns-admin.conf | ClientLanguage | en, fr, de, ja | If the client does not send an accept language header, ClientLanguage defines the language of the Directory Server User Information and Password pages. The two-letter value code is used to find the directory containing ns-admin.db. |
| ns-admin.conf | AdminLanguage | en, fr, de, ja | Sets the language used for administrative pages that are accessed through the administration server. |
| ns-admin.conf | DefaultLanguage | en, fr, de, ja | The language used if a value cannot be found for the client or admin languages. |

## load-types:

Chapter

3

# Creating Custom SAFs

This chapter describes how to write and use your own custom Sever Application Functions (SAFs). Creating custom SAFs allows you to modify or extend the server's built-in functionality. For example, to handle user authorization in a special way or generate dynamic HTML pages based on information in a database. Concepts discussed in this chapter include the SAF interface, creating custom SAFs, and configuring the server to use custom SAFs.

Before writing custom SAFs, you should familiarize yourself with the request-response process (see Chapter 1, "NSAPI Basics"), the built-in SAFs (see Chapter 2, "Directives and Built-In SAFs"), and the NSAPI routines you will use to implement your custom SAFs (see Chapter 4, "NSAPI Function Reference" for detailed information.)

## The SAF Interface

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

Here is the "C" interface for a SAF:

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

The pb parameter contains the parameters from the SAF's configuration line in the obj.conf file. The sn parameter contains information relating to a single TCP/IP session. The rq parameter contains information relating to the current request.

The SAF returns a result code which indicates whether and how it succeeded. The server uses the result code from each function to determine how to proceed with processing the request.

# SAF Parameters

SAFs expect 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 (A pblock maintains its data as a collection of name-value pairs. For more information about pblock and the NSAPI routines for manipulating this data structure, see Chapter 5, "Data Structure Reference").

Following are more detailed descriptions of the SAF parameters.

- pb is a pointer to a pblock. pb contains values obtained from the directive line in the obj.conf file where the SAF is configured. For example, the pb passed to basic-ncsa contains the value assigned to auth-type. This parameter is read-only and should not be modified.
- sn is a pointer to a Session data structure. sn contains variables related to an entire session (that is, the time between the opening and closing of the TCP/IP connection between the client and the server). The same sn pointer is passed to each SAF called within each request for an entire session. The following list describes the most important fields in this data structure (See Chapter 4, "NSAPI Function Reference" for information about NSAPI routines for manipulating the Session data structure):
  - sn->client is a pointer to a pblock containing information about the client such as its IP address, DNS name, or certificate. If the client does not have a DNS name or if it cannot be found, it will be set to the client's IP number.
  - sn->csd is a platform-independent client socket descriptor. You will pass this to the routines for reading reading from and writing to the client.

- `rq` is a pointer to a `Request` data structure. `rq` contains variables related to the current request, such as the request headers, URI, and local file system path. The same `rq` pointer is passed to each SAF called in the request-response process for an HTTP request. The following list describes the most important fields in this data structure (See Chapter 4, “NSAPI Function Reference” for information about NSAPI routines for manipulating the `Request` data structure).

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

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

  - `rq->headers` is a pointer to a `pblock` containing all the request headers (such as User-Agent, If-Modified-Since, ...) received from the client in the HTTP request. See Appendix A, “HyperText Transfer Protocol” for more information about request headers. This `pblock` does not normally change throughout the request-response process.

  - `rq->srvhdrs` is a pointer to a `pblock` containing the response headers (such as Server, Date, Content-type, Content-length,...) to be sent to the client in the HTTP response. See Appendix A, “HyperText Transfer Protocol” for more information about response headers.

  - `rq->directive_is_cacheable` is a flag which may be used by your SAF to tell the server that your SAF is cacheable.

    The server attempts to cache requests that will generate the same response when requested by different clients at different times. That is, if a client requests /mfg/proc/item.txt, and then another client requests / mfg/proc/item.txt, the server's response will be the same as long as / mfg/proc/item.txt doesn't change between the requests. When the server can avoid calling the SAFs for a request, it can return the response faster.

The flag is set to 0 on entry to each SAF. If you do not set this flag to 1 before your SAF returns, the server will not try to cache the request, and each subsequent request will call your SAF again. If you set it to 1, and all other SAFs called for this request also set the flag, the server will cache the request and will not call your SAF when another request is made for the same resource.

If your SAF performs access control, logging, depends on the client IP address, the user-agent, or any headers the client sends, you should not set `directive_is_cacheable`. Otherwise you should set `directive_is_cacheable` to 1.

During development, you may disable server caching by adding the following line at the top of the obj.conf file:

```
Init fn=cache-init disable=true
```

Don't forget to stop and start the server after saving the file. This will disable server caching so that your SAF will always be called.

# 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 tells the server to proceed to the next request-response step, skipping any other SAFs in the current step. For the other request-response steps (PathCheck, ObjectType, and AddLog), the server proceeds to the next SAF in the current step.
- `REQ_NOACTION` indicates the SAF took no action. The server continues with the next SAF in the current server step.
- `REQ_ABORTED` indicates that an error occurred and an HTTP response should be sent to the client to indicate the cause of the error. A SAF returning REQ_ABORTED should also set the HTTP response status code. If the server finds an Error directive matching the status code or reason

phrase, it will execute the SAF specified. If not, the server will send a defualt HTTP response with the status code and reason phrase plus a short HTML page reflecting the status code and reason phrase for the user. The server then goes to the first AddLog directive.

- `REQ_EXIT` indicates the connection to the client was lost. This should be returned when the SAF fails in reading or writing to the client. The server then goes to the first AddLog directive.

# Overview of NSAPI C Functions

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

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 4, “NSAPI Function Reference”.

The categories are `pblock` manipulation, protocol utilities, memory management, file I/O, network I/O, threads, and utilities:

- **`pblock` manipulation** provides routines for locating, adding, and removing entries in a `pblock` data structure.
  - `pblock_findval()` returns the value for a given name in a `pblock`.
  - `pblock_nvinsert()` adds a new name-value entry to a `pblock`.
  - `pblock_remove()` removes a `pblock` entry by name from a `pblock`. The entry is not disposed. Use `param_free()`to free the memory used by the entry.
  - `param_free()` frees the memory for the given `pblock` entry.
  - `pblock_pblock2str()` creates a new string containing all the name-value pairs from a `pblock` in the form "`name=value name=value`". This a very useful function for debugging.

- **protocol utilities** provide functionality necessary to implemet Service SAFs.
  - `request_header()` returns the value for a given request header name, reading the headers if necessary. This function must be used when requesting entries from the browser header `pblock` (`rq->headers`).
  - `protocol_status()` sets the HTTP response status code and reason phrase
  - `protocol_start_response()` sends the HTTP response and all HTTP headers to the browser.
- **memory management** routines provide fast, platform-indepent versions of the standard memory management routines. They also prevent memory leaks by allocating from a temporary memory (called "pooled" memory) for each request and then disposing the entire pool after each request. There are wrappers for standard memory routines for using permenant memory. To disable pooled memory for debugging, see the built-in SAF `pool-init` in Chapter 2, “Directives and Built-In SAFs”.
  - MALLOC()
  - FREE()
  - STRDUP()
  - REALLOC()
  - CALLOC()
  - PERM_MALLOC()
  - PERM_FREE()
  - PERM_STRDUP()
  - PERM_REALLOC()
  - PERM_CALLOC()
- **file I/O** provides platform-independent, thread-safe file I/O routines.

  - `system_fopenRO()` opens a file for read-only access.
  - `system_fopenRW()` opens a file for read-write access, creating the file if necessary.
  - `system_fopenWA()` opens a file for write-append access, creating the file if necessary.
  - `system_fopenWT()` opens a file for write access and truncates the file, creating the file if necessary.
  - `system_fclose()` closes a file.
  - `system_fread()` reads from a file.
  - `system_fwrite()` writes to a file.
  - `system_fwrite_atomic()` locks the given file before writing to it. This avoids interference between simultaneous writes by multiple processes or threads.
  - `ereport()` logs an error to the error log file.
- **network I/O** provides platform-independent, thread-safe network I/O routines. These routines work with SSL when it's enabled.
  - `net_grab()` read from the network socket.
  - `netbuf_getc()` gets a character from a network buffer.
  - `net_write()` writes to the network socket.
- **threads** provides functions for creating your own threads which are compatible with the server's threads. There are also routines for critical sections and condition variables.
  - `systhread_start()` creates a new thread.
  - `systhread_sleep()` puts a thread to sleep for a given time.
  - `crit_init()` creates a new critical section variable.
  - `crit_enter()` gains ownership of a critical section.

- `crit_exit()` surrender ownership of a critical section.
- `crit_terminate()` disposes of a critical section variable.
- `condvar_init()` create a new condition variable.
- `condvar_notify()` awaken any threads blocked on a condition variable.
- `condvar_wait()` block on a condition variable.
- `condvar_terminate()` dispose of a condition variable.

- **utilities** provide platform-independent, thread-safe versions of many standard library functions (such as string manipulation) as well as new utilities useful for NSAPI.
  - `daemon_atrestart()` (Unix only) registers a user function to be called when the server is sent a restart signal (HUP) or at shutdown.
  - `util_getline()` get the next line (up to a LF or CRLF) from a buffer.
  - `util_hostname()` gets the local hostname as a fully qualified domain name.
  - `util_later_than()` compares two dates.
  - `util_sprintf()` same as standard library routine `sprintf()`.
  - `util_strftime()` same as standard library routine `strftime()`.
  - `util_uri_escape()` convert the special characters in a string into URI escaped format.
  - `util_uri_unescape()` convert the URI escaped characters in a string back into special characters.

# SAFs for each Directive

As the primary mechanism for passing along information throughout the request-response process, the `rq` parameter merits further discussion. On input to a SAF, `rq` contains whatever values were inserted or modified by previously executed SAFs. On output, `rq` will contain 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` which were previously inserted by an AuthTrans SAF. The following list describes how the SAFs for each directive might use these paramters:

**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`.
- Return `REQ_PROCEED`.

**AuthTrans SAFs**

- Purpose: Verify any authorization information. Only "Basic" authorization is currently defined in the HTTP/1.0 spec.
- Check for "Authorization" header in `rq->headers` which contains the authorization type and uu-encoded user/password information. If header was not sent return `REQ_NOACTION`.
- If header exists, check authenticity of user/password.
- If authentic, create "auth-type", plus "auth-user" and/or "auth-group" parameter in rq->vars to be used later by PathCheck SAFs.

- Return REQ_PROCEED if the user was successfully 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 you provided the full local file system path in "ppath", or REQ_NOACTION if not.
- To redirect browser to another site: Change "ppath" in rq->vars to "/URL". Add "url" to rq->vars with full url (eg. http://home.netscape.com/). Return REQ_PROCEED.

**PathCheck SAFs**

- Purpose: Check path validity and user's access
- Check "auth-type", "auth-user" and/or "auth-group" in rq->vars.
- Return REQ_PROCEED if user (and group) is authorized for this area ("path" in rq->vars).
- If not authorized, insert "WWW-Authenticate" to rq->srvhdrs with a value like: "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

**Service SAFs**

- Purpose: Generate and send response to browser.
- The Service SAF is only called if each of the optional wildcard parameters (type, method, query) match this 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.
- Call `protocol_start_response()` to send HTTP response & headers.
- Generate and send data to the browser using `net_write()`.
- 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 wildcard parameters (`code` & `reason`) match.
- Same as Service SAFs, but only in response to an HTTP status error condition

**AddLog SAFs**

- Purpose: Log the transaction to a log file
- Use any data available in `pb`, `sn`, or `rq` to log this transaction.
- Return `REQ_PROCEED`.

# Creating a Custom SAF

Custom SAFs are functions in shared libraries that are loaded and called by the server. Follow these steps to create a custom SAF:

1. Write the source code for your SAF using the NSAPI functions. Each SAF is written for a specific directive.
2. Compile and link the source code, creating a shared library (`.so, .sl,` or `.dll`) file.
3. Configure the server by editing the `obj.conf` file to:
   - Load your shared library file with your custom SAF(s).
   - Call your custom SAF(s) at the appropriate time.
4. Stop and start the server.
5. Access the server from a browser to test you SAF.

The following sections describe these steps in greater detail.

# Write the Source Code

NSAPI provides routines you may call from your custom SAFs. Built-in SAFs use these routines as well. This section describes a few of the routines you are most likely to use. Chapter 4, “NSAPI Function Reference” provides more complete and detailed information about all of the routines available. For examples of custom SAFs, see `nsapi/examples/` in the server root directory and Chapter 6, “Examples of Custom SAFs”.

The Enterprise Server runs as a multi-threaded single process. On Unix platforms there are actually 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 the HTTP requests.

Keep these things in mind when writing your SAF. Write thread-safe code. Blocking may affect performance. Write small functions with parameters and configure them in `obj.conf`. Carefully check and handle all errors. Also log them so that you can determine the source of problems and fix them.

# Compile and Link

Compile and link your code with the native compiler for the target platform. For Windows NT, use Microsoft Visual C++ 4.2 or newer. You must have an import list that specifies all global variables and functions you want to access from the server binary. Use the correct compiler and linker flags for your platform. Refer to the example Makefile in the `nsapi/examples` directory. You will need the file `httpd.lib` to link your SAF.

The `include` directory within the server root directory, contains the NSAPI header files. All of the NSAPI header information is now contained in one file called "`nsapi.h`". The other header files are for backward compatibility and simply include `nsapi.h`.

# Configure the Server

Configure the server to open your shared library file and load one or more custom SAFs by adding the following line after the other Init directives at the top of the `obj.conf` file:

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

`shlib` is the local file system path to the shared library

`funcs` is a comma-separated list of function names to be loaded from the shared library. Function names are case-sensitive. You may use dash (-) in place of underscore (_) in function names. There should be no spaces in the function name list.

Next configure the server to call one of your custom SAFs at the appropriate time by adding a line like the following to the `obj.conf` file:

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

`Directive` is one of the server directives.

`func-name` is the name of the SAF to execute.

`nameN="valueN"` are the names and values of paramaters which are passed to the SAF.

# Stop and Start the Server

Next, stop and start the server. On Unix you may execute the shell scripts `stop` and `start` in the servers home directory. Do not use restart on Unix since the server will not reload your shared library after it has been loaded once.

On Windows NT you may use the Services Control Panel to stop and start the server. Once you have started the server with your shared library, you'll have to stop it before you can build your shared library again.

If there are problems during startup, check the error log.

# Access the Serer

Test your SAF by accessing your server from a browser with a URL that will trigger your function.

You should disable caching in your browser so that the server is sure to be accessed. In Navigator you may hold the shift key while clicking the Reload button to ensure that the cache is not used.

You may also wish to disable the server cache using the `cache-init` SAF.

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

# CGI to NSAPI Conversion

You may have a need to convert a CGI into a SAF using NSAPI. Since the CGI environment variables are not available to NSAPI, you'll retrieve them from the NSAPI pblocks. The table below indicates where each CGI environment variable can be obtained in NSAPI.

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

Table 3.1

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

Table 3.1

| CGI getenv() | NSAPI |
|---|---|
| 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 | MAGNUS_VERSION_STRING |
| **Netscape specific:** | |
| CLIENT_CERT | pblock_findval("auth-cert", rq->vars); |
| HOST | char *session_maxdns(sn); (may be null) |
| HTTPS | security_active ? "ON" : "OFF"; |
| HTTPS_KEYSIZE | pblock_findval("keysize", sn->client); |
| HTTPS_SECRETKEYSIZE | pblock_findval("secret-keysize", sn->client); |
| QUERY | pblock_findval( query", rq->reqpb); (GET only, POST puts query string in entity-body data) |
| SERVER_URL | http_uri2url_dynamic("","", sn, rq); |

Chapter

4

# NSAPI Function Reference

This chapter lists all the public "C" functions and macros of the Netscape Server Applications Programming Interface (NSAPI) in alphabetic order. These are the functions you use when writing your own Server Application Functions (SAFs). For information on the built-in SAFs, see Chapter 2, "Directives and Built-In SAFs".

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

For more information on data structures, see Chapter 5, "Data Structure Reference" and the nsapi.h header file.

# NSAPI Functions (in Alphabetic Order)

## CALLOC()

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

**Syntax** `void *CALLOC(int num, int size)`

**Returns** A void pointer to a block of memory.

**Parameters** `int num` is the number of elements to allocate.

`int size` is the size in bytes of each element.

**Example**

```
/* Allocate space for an array of 100 char pointers */
char *name;
name = (char *) CALLOC(100, sizeof(char *));
```

**See also** `FREE, REALLOC, STRDUP, PERM_MALLOC, PERM_FREE, PERM_REALLOC, PERM_STRDUP`

## cinfo_find()

The `cinfo_find()` function uses the MIME types information to find the type, encoding, and/or language based on the extension(s) 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 it will be receiving from the server.

The 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 may contain multiple extensions separated by period (.) to indicate type, encoding, or language. For example, the URI "a/b/filename.jp.txt.zip" could represent a Japanese language, text/plain type, zip encoded file.

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

**Returns**

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

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

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

---

## condvar_init()

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

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

**Returns** A newly allocated condition variable (`CONDVAR`).

**Parameters** `CRITICAL id` is a critical-section variable.

**See also** `condvar_notify, condvar_terminate, condvar_wait, crit_init, crit_enter, crit_exit, crit_terminate.`

---

## condvar_notify()

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

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

**Returns** `void`

**Parameters** `CONDVAR cv` is a condition variable.

**See also** `condvar_init, condvar_terminate, condvar_wait, crit_init, crit_enter, crit_exit, crit_terminate.`

---

## condvar_terminate()

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

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

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

**Returns** `void`

**Parameters** `CONDVAR cv` is a condition variable.

**See also** `condvar_init, condvar_notify, condvar_wait, crit_init, crit_enter, crit_exit, crit_terminate.`

---

## condvar_wait()

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

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

**Returns** `void`

**Parameters** `CONDVAR cv` is a condition variable.

**See also** `condvar_init, condvar_notify, condvar_terminate, crit_init, crit_enter, crit_exit, crit_terminate.`

## crit_enter()

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

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

**Returns** `void`

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

**See also** `crit_init, crit_exit, crit_terminate.`

## crit_exit()

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

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

**Returns** `void`

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

**See also** `crit_init, crit_enter, crit_terminate.`

## crit_init()

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

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

**Syntax** `CRITICAL crit_init(void);`

**Returns** A newly allocated critical-section variable (`CRITICAL`)

**Parameters** none.

**See also** `crit_enter, crit_exit, crit_terminate.`

---

## crit_terminate()

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

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

**Returns** `void`

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

**See also** `crit_init, crit_enter, crit_exit.`

---

## daemon_atrestart()

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

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

**Returns** `void`

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

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

**Example**

```
/* Register the brief_terminate 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);
}
```

## filebuf_buf2sd()

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

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

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

**Returns**

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

**Parameters** `filebuf *buf` is the file buffer which must already have been opened.

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

**Example**

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

**See also** `filebuf_close`, `filebuf_open`, `filebuf_open_nostat`, `filebuf_getc`.

## filebuf_close()

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);`

**Returns** void

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

**Example**
```
filebuf_close(buf);
```

**See also** `filebuf_open, filebuf_open_nostat, filebuf_buf2sd, filebuf_getc`

---

## filebuf_getc()

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

Use `filebuf_getc` to sequentially read characters from a buffered file.

**Syntax**
```
filebuf_getc(filebuf b);
```

**Returns**
- An integer containing the character retrieved
- The constant IO_EOF or IO_ERROR upon an end of file or error

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

**See also** `filebuf_close, filebuf_buf2sd, filebuf_open, filebuf_open_nostat`

---

## filebuf_open()

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);
```

**Returns**
- A pointer to a new buffer structure to hold the data, if successful
- NULL if no buffer could be opened

**Parameters** `SYS_FILE fd` is the platform-independent file descriptor of the file which 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`, `filebuf_buf2sd`, `filebuf_close`, `filebuf_open_nostat`

---

## filebuf_open_nostat()

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 `filebuf_open`, but is more efficient, since it does not need to call the `request_stat_path` function. It requires that the stat information be passed in.

**Syntax**

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

**Returns**

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

**Parameters**

`SYS_FILE fd` is the platform-independent file descriptor of the file which 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`, `filebuf_open`, `filebuf_getc`, `filebuf_buf2sd`

## FREE()

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.

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

**Returns** `void`

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

**Example**

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

**See also** MALLOC, CALLOC, REALLOC, STRDUP, PERM_MALLOC, PERM_FREE, PERM_REALLOC, PERM_STRDUP

## func_exec()

The func_exec function executes the function named by the fn entry in a specified pblock. If the function name is not found, it 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);`

**Returns**

- The value returned by the executed function
- The constant REQ_ABORTED if no function was 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 as the ones passed into your SAF.

**See also** `log_error`

---

## func_find()

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

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

**Returns**

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

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

**Example**

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

**See also** `func_exec`

---

## log_error()

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

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

**Returns**

- 0 if the log entry was created.
- -1 if the log entry was not created.

**Parameters** `int degree` specifies the severity of the error. It must be one of the following constants:

LOG_WARN—warning
LOG_MISCONFIG—a syntax error or permission violation
LOG_SECURITY—an authentication failure or 403 error from a host
LOG_FAILURE—an internal problem
LOG_CATASTROPHE—a non-recoverable server error
LOG_INFORM—an informational message

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

`Session *sn` is the Session.

`Request *rq` is the Request.

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

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

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

**Example**

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

**See also** `func_exec`

---

## magnus_atrestart()

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

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

**Syntax**

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

**Returns** `void`

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

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

**Example**

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

## MALLOC()

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

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

**Returns** A void pointer to a block of memory.

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

**Example**

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

**See also** `FREE, CALLOC, REALLOC, STRDUP, PERM_MALLOC, PERM_FREE, PERM_CALLOC, PERM_REALLOC, PERM_STRDUP`

## net_ip2host()

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

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

**Returns**

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

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

`int verify`, if non-zero, specifies that the function should verify the fully-qualified domain name. Though this requires an extra query, you should use it when checking access control.

## net_read()

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

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

**Returns**

- The number of bytes read, which will not exceed the maximum size, `sz`.
- A negative value 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 not to return because not enough bytes were read in the given time, but to limit the amount of time devoted to waiting until some data arrives.

**See also** `net_write`

## net_write()

The `net_write` function writes a specified number of bytes to a specified socket from a specified buffer. It returns the number of bytes written.

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

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

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

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

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

**Example**

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

**See also** `net_read`

---

## netbuf_buf2sd()

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

**Syntax**

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

**Returns**

- The number of bytes transferred to the socket, if successful
- 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`, `netbuf_getc`, `netbuf_grab`, `netbuf_open`

---

## netbuf_close()

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

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

**Syntax**

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

**Returns** `void`

**Parameters** `netbuf *buf` is the buffer to close.

**See also** `netbuf_buf2sd`, `netbuf_getc`, `netbuf_grab`, `netbuf_open`

## netbuf_getc()

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

**Syntax** `netbuf_getc(netbuf b);`

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

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

**See also** `netbuf_buf2sd`, `netbuf_close`, `netbuf_grab`, `netbuf_open`

## netbuf_grab()

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

This function is used by the function `netbuf_buf2sd`.

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

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

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

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

**See also** `netbuf_buf2sd`, `netbuf_close`, `netbuf_getc`, `netbuf_open`

## netbuf_open()

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

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

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

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

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

**See also** `netbuf_buf2sd`, `netbuf_close`, `netbuf_getc`, `netbuf_grab`

---

## param_create()

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

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

**Returns** A pointer to a new pb_param structure.

**Parameters** `char *name` is the string containing the name.

`char *value` is the string containing the value.

**Example**

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

**See also** `param_free, pblock_pinsert, pblock_remove`

---

## param_free()

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

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

**Returns**

- 1 if the parameter was freed
- 0 if the parameter was NULL

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

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

**See also** `param_create, pblock_pinsert, pblock_remove`

---

## pblock_copy()

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

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

**Returns** `void`

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

`pblock *dst` is the destination pblock.

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

**See also** `pblock_create, pblock_dup, pblock_free, pblock_find, pblock_findval, pblock_remove, pblock_nvinsert`

---

## pblock_create()

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

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

**Returns** A pointer to a newly allocated pblock.

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

**See also** `pblock_copy, pblock_dup, pblock_find, pblock_findval, pblock_free, pblock_nvinsert, pblock_remove`

## pblock_dup()

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

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

**Returns** A pointer to a newly allocated `pblock`.

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

**See also** `pblock_create`, `pblock_find`, `pblock_findval`, `pblock_free`, `pblock_find`, `pblock_remove`, `pblock_nvinsert`

## pblock_find()

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

This function is implemented as a macro.

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

**Returns**

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

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

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

**See also** `pblock_copy, pblock_dup, pblock_findval, pblock_free, pblock_nvinsert, pblock_remove`

## pblock_findval()

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

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

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

**Returns**
- A string containing the value associated with the name
- NULL if no match was found

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

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

**Example** see `pblock_nvinsert`.

**See also** `pblock_create, pblock_copy, pblock_find, pblock_free, pblock_nvinsert, pblock_remove, request_header`

---

## pblock_free()

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

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

**Returns** `void`

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

**See also** `pblock_copy, pblock_create, pblock_dup, pblock_find, pblock_findval, pblock_nvinsert, pblock_remove`

---

## pblock_nninsert()

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

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

**Returns** A pointer to the new `pb_param` structure.

**Parameters** `char *name` is the name of the new entry.

`int value` is the numeric value being inserted into the `pblock`

The `pblock_nninsert` function requires that the parameter `value` be an integer. If the value you assign is not a number, then instead use the function `pblock_nvinsert` to create the parameter.

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

**See also** `pblock_copy`, `pblock_create`, `pblock_find`, `pblock_free`, `pblock_nvinsert`, `pblock_remove`, `pblock_str2pblock`

---

## pblock_nvinsert()

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

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

**Returns** A pointer to the newly allocated `pb_param` structure

**Parameters** `char *name` is the name of the new entry.

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

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

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

**See also** `pblock_copy`, `pblock_create`, `pblock_find`, `pblock_free`, `pblock_nninsert`, `pblock_remove`, `pblock_str2pblock`

---

## pblock_pb2env()

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

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

**Returns** A pointer to the evironment.

**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, pblock_create, pblock_find, pblock_free, pblock_nvinsert, pblock_remove, pblock_str2pblock`

---

## pblock_pblock2str()

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);`

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

**Parameters** `pblock *pb` is the `pblock` to be copied.

`char *str` is the string into which the `pblock` is to be copied. It 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, pblock_create, pblock_find, pblock_free, pblock_nvinsert, pblock_remove, pblock_str2pblock`

---

## pblock_pinsert()

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

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

**Returns** `void`

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

`pblock *pb` is the `pblock`.

**See also** `pblock_copy, pblock_create, pblock_find, pblock_free, pblock_nvinsert, pblock_remove, pblock_str2pblock`

---

## pblock_remove()

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

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

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

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

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

**See also** `pblock_copy, pblock_create, pblock_find, pblock_free, pblock_nvinsert, param_create, param_free`

---

## pblock_str2pblock()

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

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

**Returns**
- The number of parameter pairs added to the `pblock`, if any
- -1 if an error occurred

**Parameters** `char *str` is the string to be scanned.

The name-value pairs in the string can have the format `name=value` or `name="value"`.

All back slashes (\) must be followed by a literal character. If string values are found with no unescaped = signs (no `name=`), it assumes the names 1, 2, 3, and so on, depending on the string position. For example, if `pblock_str2pblock` finds `"some strings together"`, the function treats the strings as if they appeared in name-value pairs as `1="some" 2="strings" 3="together"`.

`pblock *pb` is the `pblock` into which the name-value pairs are stored.

**See also** `pblock_copy, pblock_create, pblock_find, pblock_free, pblock_nvinsert, pblock_remove, pblock_pblock2str`

---

## PERM_CALLOC()

The `PERM_CALLOC` macro is a platform-independent substitute for the C library routine `calloc`. It allocates `num*size` bytes of memory that persists after the request that is being processed has been completed. If pooled memory has been disabled in the configuration file (with the `pool-init` built-in SAF), `PERM_CALLOC` and `CALLOC` both obtain their memory from the system heap.

**Syntax** `void *PERM_CALLOC(int num, int size)`

**Returns** A void pointer to a block of memory

**Parameters** `int num` is the number of elements to allocate.

`int size` is the size in bytes of each element.

**Example**

```
/* Allocate 256 bytes for a name */
char **name;
name = (char **) PERM_CALLOC(100, sizeof(char *));
```

**See also** `PERM_FREE, PERM_STRDUP, PERM_MALLOC, PERM_REALLOC, MALLOC, FREE, CALLOC, STRDUP, REALLOC`

## PERM_FREE()

The PERM_FREE macro is a platform-independent substitute for the C library routine free. It deallocates the persistent space previously allocated by PERM_MALLOC, PERM_CALLOC, or PERM_STRDUP. If pooled memory has been disabled in the configuration file (with the pool-init built-in SAF), PERM_FREE and FREE both deallocate memory in the system heap.

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

**Returns** `void`

**Parameters** `void *ptr` is a (void *) pointer to block of memory. If the pointer is not one created by PERM_MALLOC, PERM_CALLOC, or PERM_STRDUP, the behavior is undefined.

**Example**

```
char *name;
name = (char *) PERM_MALLOC(256);
...
PERM_FREE(name);
```

**See also** FREE, MALLOC, CALLOC, REALLOC, STRDUP, PERM_MALLOC, PERM_CALLOC, PERM_REALLOC, PERM_STRDUP

## PERM_MALLOC()

The PERM_MALLOC macro is a platform-independent substitute for the C library routine malloc. It provides allocation of memory that persists after the request that is being processed has been completed. If pooled memory has been disabled in the configuration file (with the pool-init built-in SAF), PERM_MALLOC and MALLOC both obtain their memory from the system heap.

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

**Returns** A void pointer to a block of memory

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

**Example**

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

**See also** PERM_FREE, PERM_STRDUP, PERM_CALLOC, PERM_REALLOC, MALLOC, FREE, CALLOC, STRDUP, REALLOC

---

## PERM_REALLOC()

The PERM_REALLOC macro is a platform-independent substitute for the C library routine realloc. It changes the size of a specified memory block that was originally created by MALLOC, CALLOC, or STRDUP. The contents of the object remains unchanged up to the lesser of the old and new sizes. If the new size is larger, the new space is uninitialized.

**Warning** Calling PERM_REALLOC for a block that was allocated with MALLOC, CALLOC, or STRDUP will not work.

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

**Returns** A void pointer to a block of memory

**Parameters** `void *ptr` a void pointer to a block of memory created by PERM_MALLOC, PERM_CALLOC, or PERM_STRDUP.

`int size` is the number of bytes to which the memory block should be resized.

**Example**

```
char *name;
name = (char *) PERM_MALLOC(256);
if (NotBigEnough())
   name = (char *) PERM_REALLOC(512);
```

**See also** PERM_MALLOC, PERM_FREE, PERM_CALLOC, PERM_STRDUP, MALLOC, FREE, STRDUP, CALLOC, REALLOC

---

## PERM_STRDUP()

The PERM_STRDUP macro is a platform-independent substitute for the C library routine strdup. It creates a new copy of a string in memory that persists after the request that is being processed has been completed. If pooled memory has been disabled in the configuration file (with the pool-init built-in SAF), PERM_STRDUP and STRDUP both obtain their memory from the system heap.

The PERM_STRDUP routine is functionally equivalent to

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

A string created with PERM_STRDUP should be disposed with PERM_FREE.

**Syntax** `char *PERM_STRDUP(char *ptr);`

**Returns** A pointer to the new string

**Parameters** `char *ptr` is a pointer to a string.

**See also** PERM_MALLOC, PERM_FREE, PERM_CALLOC, PERM_REALLOC, MALLOC, FREE, STRDUP, CALLOC, REALLOC

---

## protocol_dump822()

The `protocol_dump822` function prints headers from a specified `pblock` into a specific buffer, with a specified size and position. Use this function to serialize the headers so that they can be sent, for example, in a mail message.

**Syntax** `char *protocol_dump822(pblock *pb, char *t, int *pos, int tsz);`

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

The function also 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 dumped.

`int tsz` is the size of the buffer.

**See also** `protocol_start_response, protocol_status`

---

## protocol_set_finfo()

The `protocol_set_finfo` function retrieves the `content-length` and `last-modified` date from a specified `stat` structure and adds them to the response headers (`rq->srvhdrs`). Call `protocol_set_finfo` before calling `protocol_start_response`.

**Syntax** `int protocol_set_finfo(Session *sn, Request *rq, struct stat *finfo);`

**Returns**

- The constant `REQ_PROCEED` if the request can proceed normally
- 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 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, protocol_status`

---

## protocol_start_response()

The `protocol_start_response` function initiates the HTTP response for a specified session and request. If the protocol version is HTTP/0.9, the function does nothing, because that version has no concept of status. If the protocol version is HTTP/1.0, the function sends a status line followed by the response headers. Use this function to set up HTTP and prepare the client and server to receive the body (or data) of the response.

**Syntax**

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

**Returns**

- The constant `REQ_PROCEED` if the operation succeeded, in which case you should send the data you were preparing to send.
- The constant `REQ_NOACTION` if the operation succeeded, but the request method was HEAD in which case no data should be sent to the client.
- The constant `REQ_ABORTED` if the operation did not succeed.

**Parameters** `Session *sn` is the Session.

`Request *rq` is the Request.

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

**Example**

```
/* A noaction response from this function means the request was HEAD */
if (protocol_start_response(sn, rq) == REQ_NOACTION) {
   filebuf_close(groupbuf);  /* close our file*/
   return REQ_PROCEED;
}
```

**See also** `protocol_status`

---

## protocol_status()

The `protocol_status` function sets the session status to indicate whether an error condition occurred. If the reason string is NULL, the server attempts to find a reason string for the given status code. If it 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`.

The following is a list of valid status code constants:

```
PROTOCOL_CONTINUE
PROTOCOL_SWITCHING
PROTOCOL_OK
PROTOCOL_CREATED
PROTOCOL_NO_RESPONSE
PROTOCOL_PARTIAL_CONTENT
PROTOCOL_REDIRECT
PROTOCOL_NOT_MODIFIED
PROTOCOL_BAD_REQUEST
PROTOCOL_UNAUTHORIZED
PROTOCOL_FORBIDDEN
PROTOCOL_NOT_FOUND
PROTOCOL_METHOD_NOT_ALLOWED
PROTOCOL_PROXY_UNAUTHORIZED
PROTOCOL_CONFLICT
PROTOCOL_LENGTH_REQUIRED
PROTOCOL_PRECONDITION_FAIL
PROTOCOL_ENTITY_TOO_LARGE
PROTOCOL_URI_TOO_LARGE
PROTOCOL_SERVER_ERROR
PROTOCOL_NOT_IMPLEMENTED
PROTOCOL_VERSION_NOT_SUPPORTED
```

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

**Returns** `void`, but it sets values in the Session/Request designated by `sn`/`rq` for the status code and the reason string

**Parameters** `Session *sn` is the Session.

`Request *rq` is the Request.

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

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

`char *r` is the reason string.

**Example**

```
/* if we find extra path-info, the URL was bad so tell the */
/* browser it was not found */
if (t = pblock_findval("path-info", rq->vars)) {
    protocol_status(sn, rq, PROTOCOL_NOT_FOUND, NULL);
    log_error(LOG_WARN, "function-name", sn, rq, "%s not found", path);
    return REQ_ABORTED;
}
```

**See also** `protocol_start_response`

---

## protocol_uri2url()

The `protocol_uri2url` function takes strings containing the given URI prefix and URI suffix, and creates a newly-allocated fully qualified URL in the form `http://(server):(port)(prefix)(suffix)`. See `protocol_uri2url_dynamic`.

If you want to omit either the URI prefix or suffix, use "" instead of NULL as the value for either parameter.

**Syntax** `char *protocol_uri2url(char *prefix, char *suffix);`

**Returns** A new string containing the URL

**Parameters** `char *prefix` is the prefix.

`char *suffix` is the suffix.

**See also** `protocol_start_response, protocol_status, pblock_nvinsert, protocol_uri2url_dynamic`

---

## protocol_uri2url_dynamic()

The `protocol_uri2url` function takes strings containing the given URI prefix and URI suffix, and creates a newly-allocated fully qualified URL in the form `http://(server):(port)(prefix)(suffix)`.

If you want to omit either the URI prefix or suffix, use "" instead of NULL as the value for either parameter.

The `protocol_uri2url_dynamic` function is similar to the `protocol_uri2url` function but should be used whenever the `Session` and `Request` structures are available. This ensures that the URL that it constructs refers to the host that the client specified.

**Syntax**

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

**Returns** A new string containing the URL

**Parameters**

`char *prefix` is the prefix.

`char *suffix` is the suffix.

`Session *sn` is the Session.

`Request *rq` is the Request.

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

**See also** `protocol_start_response, protocol_status, protocol_uri2url`

## REALLOC()

The REALLOC macro is a platform-independent substitute for the C library routine realloc. It changes the size of a specified memory block that was originally created by MALLOC, CALLOC, or STRDUP. The contents of the object remains unchanged up to the lesser of the old and new sizes. If the new size is larger, the new space is uninitialized.

**Warning** Calling REALLOC for a block that was allocated with PERM_MALLOC, PERM_CALLOC, or PERM_STRDUP will not work.

**Syntax**
```
void *REALLOC(void *ptr, int size);
```

**Returns** A pointer to the new space if the request could be satisfied.

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

**See also** MALLOC, FREE, STRDUP, CALLOC, PERM_MALLOC, PERM_FREE, PERM_REALLOC, PERM_CALLOC, PERM_STRDUP

## request_header()

The request_header function finds an entry in the pblock containing the client's HTTP request headers (rq->headers). You must use this function rather than pblock_findval when accessing the client headers since the server may begin processing the request before the headers have been completely

**Syntax**
```
int request_header(char *name, char **value, Session *sn, Request *rq);
```

**Returns** A result code, REQ_PROCEED if the header was found, REQ_ABORTED if the header was not found, 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 none 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 as the ones passed into your SAF.

**See also** `request_create, request_free`

---

## request_stat_path()

The `request_stat_path` function returns the file information structure for a specified path or, if none 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 on the size of the file, its owner, when it was created, and when it was last modified.

You should 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);
```

**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`.
- A pointer to the file information structure for the file named by the `path` parameter. Do not free this structure.

**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`

## request_translate_uri()

The `request_translate_uri` function performs virtual to physical mapping on a specified URI during a specified session. Use this function when you want to determine which file would be sent back if a given URI is accessed.

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

**Returns**
- A path string, if it performed the mapping
- 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 your SAF.

**See also** `request_create, request_free, request_header`

## session_maxdns()

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

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

**Returns**
- A string containing the host name
- NULL if the DNS name cannot be found for the IP address

**Parameters** `Session *sn` is the Session.

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

## shexp_casecmp()

The `shexp_casecmp` function validates a specified shell expression and compares it with a specified string. It returns one of three possible values representing match, no match, and invalid comparison. The comparison (in contrast to that of the `shexp_cmp` function) is not case-sensitive.

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

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

**Returns**

- 0 if a match was found
- 1 if no match was found
- -1 if the comparison resulted in an invalid expression

**Parameters** `char *str` is the string to be compared.

`char *exp` is the shell expression (wildcard pattern) to compare against.

**See also** `shexp_cmp, shexp_match, shexp_valid`

## shexp_cmp()

The `shexp_casecmp` function validates a specified shell expression and compares it with a specified string. It returns one of three possible values representing match, no match, and invalid comparison. The comparison (in contrast to that of the `shexp_casecmp` function) is case-sensitive.

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

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

**Returns**

- 0 if a match was found
- 1 if no match was found
- -1 if the comparison resulted in an invalid expression

**Parameters** `char *str` is the string to be compared.

`char *exp` is the shell expression (wildcard pattern) to compare against.

**Example**

```
/* Use wildcard match to see if this path is one we want */
char *path;
char *match = "/usr/netscape/*";
if (shexp_cmp(path, match) != 0)
   return REQ_NOACTION;   /* no match */
```

**See also** `shexp_casecmp, shexp_match, shexp_valid`

---

## shexp_match()

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

The `shexp_match` function doesn't perform validation of the shell expression; instead the function assumes that you have already called `shexp_valid`.

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

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

**Returns**

- 0 if a match was found
- 1 if no match was found
- -1 if the comparison resulted in an invalid expression

**Parameters**

`char *str` is the string to be compared.

`char *exp` is the pre-validated shell expression (wildcard pattern) to compare against.

**See also** `shexp_casecmp, shexp_cmp, shexp_valid`

---

## shexp_valid()

The `shexp_valid` function validates a specified shell expression named by `exp`. Use this function to validate a shell expression before using the function `shexp_match` to compare the expression with a string.

**Syntax** `int shexp_valid(char *exp);`

**Returns**

- The constant `NON_SXP` if `exp` is a standard string
- The constant `INVALID_SXP` if `exp` is a shell expression, but invalid
- The constant `VALID_SXP` if `exp` is a valid shell expression

**Parameters** `char *exp` is the shell expression (wildcard pattern) to be validated.

**See also** `shexp_casecmp, shexp_match, shexp_cmp`

---

## STRDUP()

The `STRDUP` macro is a platform-independent substitute for the C library routine `strdup`. It creates a new copy of a string in the request's memory pool.

The `STRDUP` routine is functionally equivalent to:

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

A string created with `STRDUP` should be disposed with FREE.

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

**Returns** A pointer to the new string.

**Parameters** `char *ptr` is a pointer to a string.

**Example**

```
char *name1 = "MyName";
char *name2 = STRDUP(name1);
```

**See also** `MALLOC, FREE, CALLOC, REALLOC, PERM_MALLOC, PERM_FREE, PERM_CALOC, PERM_REALLOC, PERM_STRDUP`

---

## system_errmsg()

The `system_errmsg` function returns the last error that occurred from the most recent system call. This function is implemented as a macro that returns an entry from the global array `sys_errlist`. Use this macro to help with I/O error diagnostics.

**Syntax** `char *system_errmsg(int param1);`

**Returns** A string containing the text of the latest error message that resulted from a system call. Do not FREE this string.

**Parameters** `int param1` is reserved, and should always have the value 0.

**See also** `system_fopenRO`, `system_fopenRW`, `system_fopenWA`, `system_lseek`, `system_fread`, `system_fwrite`, `system_fwrite_atomic`, `system_flock`, `system_ulock`, `system_fclose`

---

## system_fclose()

The `system_fclose` function closes a specified file descriptor. The `system_fclose` function must be called for every file descriptor opened by any of the `system_fopen` functions.

**Syntax** `int system_fclose(SYS_FILE fd);`

**Returns**

- 0 if the close succeeded
- The constant `IO_ERROR` if the close failed

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

**Example**

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

**See also** `system_errmsg`, `system_fopenRO`, `system_fopenRW`, `system_fopenWA`, `system_lseek`, `system_fread`, `system_fwrite`, `system_fwrite_atomic`, `system_flock`, `system_ulock`

---

## system_flock()

The `system_flock` function locks the specified file against interference from other processes. Use `system_flock` if you do not want other processes using the file you currently have open. Overusing file locking can cause performance degradation and possibly lead to deadlocks.

**Syntax** `int system_flock(SYS_FILE fd);`

**Returns**

- The constant `IO_OK` if the lock succeeded

- The constant `IO_ERROR` if the lock failed

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

**See also** `system_errmsg`, `system_fopenRO`, `system_fopenRW`, `system_fopenWA`, `system_lseek`, `system_fread`, `system_fwrite`, `system_fwrite_atomic`, `system_ulock`, `system_fclose`

---

## system_fopenRO()

The `system_fopenRO` function opens the file identified by `path` in read-only mode and returns a valid file descriptor. Use this function to open files that will not be modified by your program. In addition, you can use `system_fopenRO` to open a new file buffer structure using `filebuf_open`.

**Syntax**

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

**Returns**

- The system-independent file descriptor (`SYS_FILE`) if the open succeeded
- 0 if the open failed

**Parameters** `char *path` is the file name.

**See also** `system_errmsg`, `system_fopenRW`, `system_fopenWA`, `system_lseek`, `system_fread`, `system_fwrite`, `system_fwrite_atomic`, `system_flock`, `system_ulock`, `system_fclose`

---

## system_fopenRW()

The `system_fopenRW` function opens the file identified by `path` in read-write mode and returns a valid file descriptor. If the file already exists, `system_fopenRW` does not truncate it. Use this function to open files that will be read from and written to by your program.

**Syntax**

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

**Returns**

- The system-independent file descriptor (`SYS_FILE`) if the open succeeded
- 0 if the open failed

**Parameters** `char *path` is the file name.

**Example**

```
SYS_FILE fd;
fd = system_fopenRO(pathname);
if (fd == SYS_ERROR_FD)
    break;
```

**See also** `system_errmsg, system_fopenRO, system_fopenWA, system_lseek, system_fread, system_fwrite, system_fwrite_atomic, system_flock, system_ulock, system_fclose`

---

## system_fopenWA()

The `system_fopenWA` function opens the file identified by `path` in write-append mode and returns a valid file descriptor. Use this function to open those files that your program will append data to.

**Syntax**

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

**Returns**

- The system-independent file descriptor (`SYS_FILE`) if the open succeeded
- 0 if the open failed

**Parameters** `char *path` is the file name.

**See also** `system_errmsg, system_fopenRO, system_fopenRW, system_lseek, system_fread, system_fwrite, system_fwrite_atomic, system_flock, system_ulock, system_fclose`

---

## system_fread()

The `system_fread` function reads a specified number of bytes from a specified file into a specified buffer. It returns the number of bytes read. Before `system_fread` can be used, you must open the file using any of the `system_fopen` functions, except `system_fopenWA`.

**Syntax**

```
int system_fread(SYS_FILE fd, char *buf, int sz);
```

**Returns** The number of bytes read, which may be less than the requested size if an error occurred or the end of the file was reached before that number of characters were obtained.

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

`char *buf` is the buffer to receive the bytes.

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

**See also** `system_errmsg`, `system_fopenRO`, `system_fopenRW`, `system_fopenWA`, `system_lseek`, `system_fwrite`, `system_fwrite_atomic`, `system_flock`, `system_ulock`, `system_fclose`

---

## system_fwrite()

The `system_fwrite` function writes a specified number of bytes from a specified buffer into a specified file.

Before `system_fwrite` can be used, you must open the file using any of the `system_fopen` functions, except `system_fopenRO`.

**Syntax**
```
int system_fwrite(SYS_FILE fd, char *buf, int sz);
```

**Returns**

- The constant `IO_OK` if the write succeeded
- The constant `IO_ERROR` if the write failed

**Parameters**

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

`char *buf` is the buffer containing the bytes to be written.

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

**See also** `system_errmsg`, `system_fopenRO`, `system_fopenRW`, `system_fopenWA`, `system_lseek`, `system_fread`, `system_fwrite_atomic`, `system_flock`, `system_ulock`, `system_fclose`

---

## system_fwrite_atomic()

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

**Returns**

- The constant `IO_OK` if the write/lock succeeded

- The constant `IO_ERROR` if the write/lock failed

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

`char *buf` is the buffer containing the bytes to be written.

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

**Example**
```
SYS_FILE logfd;
char *logmsg = "An error occured.";
system_fwrite_atomic(logfd, logmsg, strlen(logmsg));
```

**See also** `system_errmsg, system_fopenRO, system_fopenRW, system_fopenWA, system_lseek, system_fread, system_fwrite, system_flock, system_ulock, system_fclose`

---

## system_gmtime()

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

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

**Returns** A pointer to a calendar time (`tm`) structure containing the GMT time. Depending on your system, the pointer may point to the data item represented by the second parameter, or it may point to a statically-allocated item. For portability, do not assume either situation.

**Parameters** `time_t *tp` is an arithmetic time.

`tm *res` is a pointer to a calendar time (`tm`) structure.

**Example**
```
time_t tp;
struct tm res, *resp;
tp = time(NULL);
resp = system_gmtime(&tp, &res);
```

**See also** `system_localtime, util_strftime`

---

## system_localtime()

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

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

**Returns** A pointer to a calendar time (`tm`) structure containing the local time. Depending on your system, the pointer may point to the data item represented by the second parameter, or it may point to a statically-allocated item. For portability, do not assume either situation.

**Parameters** `time_t *tp` is an arithmetic time.

`tm *res` is a pointer to a calendar time (`tm`) structure.

**See also** `system_gmtime, util_strftime`

---

## system_lseek()

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

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

**Returns**

- -1 if the operation failed
- if the operation succeeded, the offset, in bytes, of the new position from the beginning of the file.

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

`int offset` is a number of bytes relative to `whence`. It may be negative.

`int whence` is a 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, system_fopenRO, system_fopenRW, system_fopenWA, system_fread, system_fwrite, system_fwrite_atomic, system_flock, system_ulock, system_fclose`

## system_rename()

The `system_rename` function renames a file. It may not work on directories if the old and new directories are on different file systems.

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

**Returns**
- 0 if the operation succeeded
- -1 if the operation failed

**Parameters** `char *old` is the old name of the file.

`char *new` is the new name for the file:

## system_ulock()

The `system_ulock` function unlocks the specified file that has been locked by the function `system_lock`. For more information about locking, see `system_flock`.

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

**Returns**
- The constant `IO_OK` if the operation succeeded
- The constant `IO_ERROR` if the operation failed

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

**See also** `system_errmsg`, `system_fopenRO`, `system_fopenRW`, `system_fopenWA`, `system_fread`, `system_fwrite`, `system_fwrite_atomic`, `system_flock`, `system_fclose`

## system_unix2local()

The `system_unix2local` function converts a specified Unix-style pathname to a local file system pathname. 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 like Windows NT. You can use

`system_unix2local` to convert the Unix file name into the format that Windows NT accepts. In the Unix environment, this function does nothing, but may be called for portability.

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

**Returns** A pointer to the local file system path string

**Parameters** `char *path` is the Unix-style pathname to be converted.

`char *lp` is the local pathname.

You must allocate the parameter `lp`, and it must contain enough space to hold the local pathname.

**See also** `system_fclose, system_flock, system_fopenRO, system_fopenRW, system_fopenWA, system_fwrite`

---

## systhread_attach()

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

**Syntax** `SYS_THREAD systhread_attach(void);`

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

**Parameters** none.

**See also** `systhread_current, systhread_getdata, systhread_init, systhread_newkey, systhread_setdata, systhread_sleep, systhread_start, systhread_terminate, systhread_timerset`

---

## systhread_current()

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

**Syntax** `SYS_THREAD systhread_current(void);`

**Returns** A `SYS_THREAD` pointer to the current thread

**Parameters** none.

**See also** `systhread_getdata, systhread_newkey, systhread_setdata, systhread_sleep, systhread_start, systhread_terminate, systhread_timerset`

---

## systhread_getdata()

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

**Syntax**

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

**Returns**

- A pointer to the data that was earlier used with the `systhread_setkey` function from the current thread, using the same value of `key`
- NULL if the call did not succeed, for example if the `systhread_setkey` function was never called with the specified key during this session

**Parameters** `int key` is the value associated with the stored data by a `systhread_setdata` function. Keys are assigned by the `systhread_newkey` function.

**See also** `systhread_current, systhread_newkey, systhread_setdata, systhread_sleep, systhread_start, systhread_terminate, systhread_timerset`

---

## systhread_newkey()

The `systhread_newkey` function allocates a new integer key (identifier) for thread-private data. Use this key to identify a variable that you want to localize to the current thread; then use the `systhread_setdata` function to associate a value with the key.

**Syntax**

```
int systhread_newkey(void);
```

**Returns** An integer key.

**Parameters** `none.`

**See also** `systhread_current, systhread_getdata, systhread_setdata, systhread_sleep, systhread_start, systhread_terminate, systhread_timerset`

## systhread_setdata()

The `systhread_setdata` function associates data with a specified key number for the current thread. Keys are assigned by the `systhread_newkey` function.

**Syntax** `void systhread_setdata(int key, void *data);`

**Returns** `void`

**Parameters** `int key` is the priority of the thread.

`void *data` is the pointer to the string of data to be associated with the value of `key`.

**See also** `systhread_current, systhread_getdata, systhread_newkey, systhread_sleep, systhread_start, systhread_terminate, systhread_timerset`

## systhread_sleep()

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

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

**Returns** `void`

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

**See also** `systhread_current, systhread_getdata, systhread_newkey, systhread_setdata, systhread_start, systhread_terminate, systhread_timerset`

## systhread_start()

The `systhread_start` function creates a thread with the given priority, allocates a stack of a specified number of bytes, and calls a specified function with a specified argument.

**Syntax** `SYS_THREAD systhread_start(int prio, int stksz,`

```
void (*fn)(void *), void *arg);
```

**Returns**

- A new SYS_THREAD pointer if the call succeeded
- The constant SYS_THREAD_ERROR if the call did not succeed.

**Parameters**

`int prio` is the priority of the thread. Priorities are system-dependent.

`int stksz` is the stack size in bytes. If `stksz` is zero, 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, systhread_getdata, systhread_newkey, systhread_setdata, systhread_sleep, systhread_terminate, systhread_timerset`

---

## systhread_terminate()

The `systhread_terminate` function terminates a specified thread.

**Syntax**

```
void systhread_terminate(SYS_THREAD thr);
```

**Returns** void

**Parameters** SYS_THREAD `thr` is the thread to terminate.

**See also** `systhread_current, systhread_getdata, systhread_newkey, systhread_setdata, systhread_sleep, systhread_start, systhread_timerset`

---

## systhread_timerset()

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

Because most systems don't allow the timer interval to be changed, this should be considered a suggestion, rather than a command.

**Syntax**

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

**Returns** void

**Parameters** `int usec` is the time, in microseconds

**See also** `systhread_current, systhread_getdata, systhread_newkey, systhread_setdata, systhread_sleep,systhread_start, systhread_terminate`

---

## util_can_exec()

**Unix only** The `util_can_exec` function checks that a specified file can be executed, returning either a 1 (executable) or a 0. The function checks to see if the file can be executed by the user with the given user and group ID.

Use this function before executing a program using the `exec` system call.

**Syntax**

```
int util_can_exec(struct stat *finfo, uid_t uid, gid_t gid);
```

**Returns**

- 1 if the file is executable
- 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 determines the permissions of the Unix user.

**See also** `util_env_create, util_getline, util_hostname`

---

## util_chdir2path()

The `util_chdir2path` function changes the current directory to a specified directory, where you will access a file.

When running under Windows NT, use a critical section to ensure that more than one thread does not call this function at the same time.

Use `util_chdir2path` when you want to make file access a little quicker, because you do not need to use a full paths.

**Syntax**

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

**Returns**
- 0 if the directory was changed
- -1 if the directory could not be changed

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

The parameter must be a writable string because it isn't permanently modified.

---

## util_env_create()

The `util_env_create` function creates and allocates the environment specified by `env`, returning a pointer to the environment. If the parameter `env` is NULL, the function allocates a new environment. Use `util_env_create` to create an environment when executing a new program.

**Syntax**
```
char **util_env_create(char **env, int n, int *pos);
```

**Returns** A pointer to an environment.

**Parameters** `char **env` is the existing environment or NULL.

`int n` is the maximum number of environment entries that you want in the environment.

`int *pos` is the an integer that keeps track of the number of entries used in the environment.

**See also** `util_env_replace, util_env_str, util_env_free, util_env_find`

---

## util_env_find()

The `util_env_find` function locates the string denoted by a name in a specified enviroment and returns the associated value. Use this function to find an entry in an environment.

**Syntax**
```
char *util_env_find(char **env, char *name);
```

**Returns**
- The value of the environment variable if it is found
- 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, util_env_str, util_env_free, util_env_create`

---

## util_env_free()

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);`

**Returns** `void`

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

**See also** `util_env_replace, util_env_str, util_env_find, util_env_create`

---

## util_env_replace()

The `util_env_replace` function replaces the occurrence of the variable denoted by a name in a specified environment with a specified value. Use this function to change the value of a setting in an environment.

**Syntax** `void util_env_replace(char **env, char *name, char *value);`

**Returns** `void`

**Parameters** `char **env` is the environment.

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

`char *value` is the new value to be stored.

**See also** `util_env_str, util_env_free, util_env_find, util_env_create`

## util_env_str()

The `util_env_str` function creates an environment entry and returns it. This function does not check for nonalphanumeric symbols in the name (such as the equal sign "="). You can use this function to create a new environment entry.

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

**Returns** A newly-allocated string containing the name-value pair

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

`char *value` is the new value to be stored.

**See also** `util_env_replace, util_env_free, util_env_find, util_env_create`

## util_getline()

The `util_getline` function scans the specified file buffer to find a line-feed or carriage-return/line-feed terminated string. The string is copied into the specified buffer, and NULL-terminates it. The function returns a value that indicates whether the operation stored a string in the buffer, encountered an error, or reached the end of the file.

Use this function to scan lines out of a text file, such as a configuration file.

**Syntax** `int util_getline(filebuf *buf, int lineno, int maxlen, char *l);`

**Returns**

- 0 if successful. `l` contains the string.
- 1 if the end of file was reached. `l` contains the string.
- -1 if an error occured. `l` 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 `l`.

**See also** `util_can_exec, util_env_create, util_hostname`

## util_hostname()

The `util_hostname` function retrieves the local host name and returns it as a string. If the function cannot find a fully-qualified domain name, it returns NULL. You may reallocate or free this string. Use this function to determine the name of the system you are on.

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

**Returns**
- If a fully-qualified domain name was found, a string containing that name
- NULL if the fully-qualified domain name was not found

**Parameters** none.

## util_is_mozilla()

The `util_is_mozilla` function checks whether a specified user-agent header string is a Netscape browser of at least a specified revision level, returning a 1 if it is and 0 otherwise. It uses strings to specify the revision level to avoid ambiguities like 1.56 > 1.5.

**Syntax**
```
int util_is_mozilla(char *ua, char *major, char *minor);
```

**Returns**
- 1 if the user-agent is a Netscape browser
- 0 if the user-agent is not a Netscape browser

**Parameters**

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

`char *major` is the major release number (to the left of the decimal point).

`char *minor` is the minor release number (to the right of the decimal point).

**See also** `util_is_url, util_later_than`

## util_is_url()

The `util_is_url` function checks whether a string is a URL, returning 1 if it is 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);`

**Returns**

- 1 if the string specified by `url` is a URL
- 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, util_later_than`

## util_itoa()

The `util_itoa` function converts a specified integer to a string, and returns the length of the string. Use this function to create a textual representation of a number.

**Syntax** `int util_itoa(int i, char *a);`

**Returns** The length of the string created

**Parameters** `int i` is the integer to be converted.

`char *a` is the ASCII string that represents the value. The user is responsible for the allocation and deallocation of `a`, and it should be at least 32 bytes long.

## util_later_than()

The `util_later_than` function compares the date specified in a time structure against a date specified in a string. If the date in the string is later than or equal to the one in the time structure, the function returns 1. Use this function to handle RFC 822, RFC 850, and ctime formats.

**Syntax** `int util_later_than(struct tm *lms, char *ims);`

**Returns**

- 1 if the date represented by `ims` is the same as or later than that represented by the `lms`
- 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`

---

## util_sh_escape()

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

The shell-special characters are: `&;`'\"|*?~<>^()[]{}$\\ #!`

**Syntax**

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

**Returns** A newly allocated string

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

**See also** `util_uri_escape`

---

## util_snprintf()

The `util_snprintf` function formats a specified string, using a specified format, into a specified buffer using the `printf`-style syntax and performs bounds checking. It returns the number of characters in the formatted buffer.

For more information, see the documentation on the `printf` function for the run-time library of your compiler.

**Syntax**

```
int util_snprintf(char *s, int n, char *fmt, ...);
```

**Returns** The number of characters formatted into the buffer.

**Parameters** `char *s` is the buffer to receive the formatted string.

`int n` is the maximum number of bytes allowed to be copied.

`char *fmt` is the format string. The function handles only `%d` and `%s` strings; it does not handle any width or precision strings.

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

**See also** `util_sprintf, util_vsnprintf, util_vsprintf`

---

## util_sprintf()

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

Because `util_sprintf` doesn't perform bounds checking, use this function only if you are certain that the string fits the buffer. Otherwise, use the function `util_snprintf`. For more information, see the documentation on the `printf` function for the run-time library of your compiler.

**Syntax**
```
int util_sprintf(char *s, char *fmt, ...);
```

**Returns** The number of characters formatted into the buffer.

**Parameters** `char *s` is the buffer to receive the formatted string.

`char *fmt` is the format string. The function handles only `%d` and `%s` strings; it does not handle any width or precision strings.

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

**Example**
```
char *logmsg;
int len;

logmsg = (char *) MALLOC(256);
len = util_sprintf(logmsg, "%s %s %s\n", ip, method, uri);
```

**See also** `util_snprintf, util_vsnprintf, util_vsprintf`

## util_strcasecmp()

The `util_strcasecmp` function performs a comparison of two alpha-numeric strings and returns a -1, 0, or 1 to signal which is larger or that they are identical.

The comparison is not case-sensitive.

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

**Returns**

- 1 if `s1` is greater than `s2`
- 0 if `s1` is equal to `s2`
- -1 if `s1` is less than `s2`

**Parameters** `char *s1` is the first string.

`char *s2` is the second string.

**See also** `util_strncasecmp`

## util_strftime()

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

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

**Returns** The number of characters placed into `s`, not counting the terminating NULL character.

**Parameters** `char *s` is the string buffer to put the text into. There is no bounds checking, so you must make sure that your buffer is large enough for the text of the date.

`const char *format` is a format string, a bit like a `printf` string in that it consists of text with certain `%x` substrings. You may use the constant HTTP_DATE_FMT to create date strings in the standard internet format. For more information, see the documentation on the `printf` function for the run-time library of your compiler. Refer to Appendix C, “Time Formats” for details on time formats.

`const struct tm *t` is a pointer to a calendar time (`tm`) struct, usually created by the function `system_localtime` or `system_gmtime`.

**See also** `system_localtime, system_gmtime`

---

## util_strncasecmp()

The `util_strncasecmp` function performs a comparison of the first `n` characters in the alpha-numeric strings and returns a -1, 0, or 1 to signal which is larger or that they are identical.

The function's comparison is not case-sensitive.

**Syntax**
```
int util_strncasecmp(const char *s1, const char *s2, int n);
```

**Returns**

- 1 if `s1` is greater than `s2`
- 0 if `s1` is equal to `s2`
- -1 if `s1` is less than `s2`

**Parameters**

`char *s1` is the first string.

`char *s2` is the second string.

`int n` is the number of initial characters to compare.

**See also** `util_strcasecmp`

---

## util_uri_escape()

The `util_uri_escape` function converts any special characters in the URI into the URI format (%XX where XX is the hexadecimal equivalent of the ASCII character), and returns the escaped string. The special characters are `%?#:+&*"<>`, space, carriage-return, and line-feed.

Use `util_uri_escape` before sending a URI back to the client.

**Syntax**
```
char *util_uri_escape(char *d, char *s);
```

**Returns** The string (possibly newly allocated) with escaped characters replaced.

**Parameters** `char *d` is a string. If `d` is not NULL, the function copies the formatted string into `d` and returns it. If `d` is NULL, the function allocates a properly-sized string and copies the formatted special characters into the new string, then returns it.

The `util_uri_escape` function does not check bounds for the parameter `d`. Therefore, if `d` is not NULL, it should be at least three times as large as the string `s`.

`char *s` is the string containing the original unescaped URI.

**See also** `util_uri_is_evil, util_uri_parse, util_uri_unescape`

---

## util_uri_is_evil()

The `util_uri_is_evil` function checks a specified URI for insecure path characters. Use this function to see if a URI requested by the client is insecure.

**Syntax**
```
int util_uri_is_evil(char *t);
```

**Returns**

- 1 if the URI is insecure.
- 0 if the URI is OK.

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

**See also** `util_uri_escape, util_uri_parse`

---

## util_uri_parse()

The `util_uri_parse` function converts `//`, `/./`, and `/*/../` into `/` in the specified URI (where * is any character other than `/`). You can use this function to convert a URI's 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);
```

**Returns** `void`

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

**See also** `util_uri_is_evil, util_uri_unescape`

## util_uri_unescape()

The `util_uri_unescape` function converts the encoded characters of a URI into their ASCII equivalents. Encoded characters appear as %XX where XX is a hexadecimal equivalent of the character.

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

**Returns** `void`

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

**See also** `util_uri_escape, util_uri_is_evil, util_uri_parse`

## util_vsnprintf()

The `util_vsnprintf` function formats a specified string, using a specified format, into a specified buffer using the `vprintf`-style syntax and performs bounds checking. It returns the number of characters in the formatted buffer.

For more information, see the documentation on the `printf` function for the run-time library of your compiler.

**Syntax** `int util_vsnprintf(char *s, int n, register char *fmt, va_list args);`

**Returns** The number of characters formatted into the buffer

**Parameters** `char *s` is the buffer to receive the formatted string.

`int n` is the maximum number of bytes allowed to be copied.

`register char *fmt` is the format string. The function handles only `%d` and `%s` strings; it does not handle any width or precision strings.

`va_list args` is an STD argument variable obtained from a previous call to `va_start`.

**See also** `util_sprintf, util_vsprintf`

## util_vsprintf()

The `util_vsprintf` function formats a specified string, using a specified format, into a specified buffer using the `vprintf`-style syntax without bounds checking. It returns the number of characters in the formatted buffer.

For more information, see the documentation on the `printf` function for the run-time library of your compiler.

**Syntax** `int util_vsprintf(char *s, register char *fmt, va_list args);`

**Returns** The number of characters formatted into the buffer.

**Parameters** `char *s` is the buffer to receive the formatted string.

`register char *fmt` is the format string. The function handles only `%d` and `%s` strings; it does not handle any width or precision strings.

`va_list args` is an STD argument variable obtained from a previous call to `va_start`.

**See also** `util_snprintf, util_vsnprintf`

## net_socket()

The `net_socket` function opens a connection to a socket, creating a new socket descriptor. The socket is not connected to anything, and is not listening to any port. A function must use `net_connect` to make a connection, and `net_accept` to listen.

**Syntax** `SYS_NETFD net_socket (int domain, int type, int protocol);`

**Returns** The platform-independent socket descriptor (`SYS_NETFD`) associated with the socket.

**Parameters** `int domain` must be the constant `AF_INET`.

`int type` must be the constant `SOCK_STREAM`.

`int protocol` must be the constant `IPPROTO_TCP`.

**See also** `net_read, net_write, net_connect, net_accept...`
systhread_start creates a thread with the given priority.
systhread_terminate terminates a specified thread.
systhread_current returns a pointer to the current thread.
systhread_sleep puts the calling thread to sleep for a stated time.
systhread_newkey allocates a new integer key for thread-private data.
systhread_getdata obtains the data associated with a specified key in the current thread.
systhread_setdata associates data to a specified key for the current thread.
systhread_timerset starts or resets the interrupt timer interval for a thread system.

Chapter

5

# Data Structure Reference

NSAPI uses many data structures which are defined in the "nsapi.h" header file in the includes directory of the server root. Some of the data structures are described here for your convenience.

## session

A session is the time between the opening and closing of the connection between the client and the server. The **Session** data structure holds variables that apply session wide, regardless of the requests being sent, as shown here:

```
typedef struct {
/* Information about the remote client */
   pblock *client;

   /* The socket descriptor to the remote client */
   SYS_NETFD csd;
   /* The input buffer for that socket descriptor */
   netbuf *inbuf;

/* Raw socket information about the remote */
/* client (for internal use) */
   struct in_addr iaddr;
} Session;
```

## pblock

The parameter block is the hash table that holds **pb_entry** structures. Its contents are transparent to most code.

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

## pb_entry

The **pb_entry** is a single element in the parameter block.

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

## pb_param

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

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

## client

The `Session->client` parameter block structure contains two entries:

- `The ip` entry is the IP address of the client machine.
- `The dns` entry is the DNS name of the remote machine. This member must be accessed through the **session_dns** function call:

```
/*
* session_dns returns the DNS host name of the client for this
* session and inserts it into the client pblock. Returns NULL if
* unavailable.
*/

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

## request

Under HTTP protocol, there is only one request per session. The **Request** structure contains the variables that apply to the request in that session (for example, the variables include the client's HTTP headers).

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

   /* The method, URI, and protocol revision of this request */
   block *reqpb;
   /* Protocol specific headers */
   int loadhdrs;
   pblock *headers;

   /* Server's response headers */
   pblock *srvhdrs;

   /* The object set constructed to fulfill this request */
   httpd_objset *os;

   /* The stat last returned by request_stat_path */
   char *statpath;
   struct stat *finfo;
} Request;
```

## stat

When the program calls the **stat( )** function for a given file, the system returns a structure that provides information about the file. The specific details of the structure should be obtained from your platform's implementation, but the basic outline of the structure is as follows:

```
struct stat {
   dev_t    st_dev;/* device of inode */
   inot_tst_ino;/* inode number */
   shortst_mode;/* mode bits */
   shortst_nlink;/* number of links to file /*
   shortst_uid;/* owner's user id */
   shortst_gid;/* owner's group id */
   dev_tst_rdev;/* for special files */
   off_tst_size;/* file size in characters */
   time_tst_atime;/* time last accessed */
   time_tst_mtime;/* time last modified */
   time_tst_ctime;/* time inode last changed*/
```

}.

The elements that are most significant for server plug-in API activities are `st_size`, `st_atime`, `st_mtime`, and `st_ctime`.

## shmem_s

```
typedef struct {
void *data;   /* the data */
   HANDLE fdmap;
   int size;     /* the maximum length of the data */
   char *name;   /* internal use: filename to unlink if exposed */
   SYS_FILE fd;  /* internal use: file descriptor for region */
} shmem_s;
```

## netbuf

The **netbuf** is a platform-independent network-buffering structure that maintains such members as buffer address, position within buffer, current file size, maximum file size, and so on. Details of its structure vary between implementations.

## filebuffer

The **filebuffer** is a platform-independent file-buffering structure that maintains such members as buffer address, file position, current file size, and so on. Details of its structure vary between implementations.

## cinfo

The cinfo data structure records the content info for a file.

```
typedef struct {
   char *type;/* Identifies what kind of data is in the file*/
   char *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;
```

## SYS_NETFD

The **SYS_NETFD** data structure is a platform-independent socket descriptor. Details of its structure vary between implementations.

## SYS_FILE

The **SYS_FILE** data structure is a platform-independent file descriptor. Details of its structure vary between implementations.

## SEMAPHORE

The **SEMAPHORE** data structure is a platform-independent implementation of semaphores. Details of its structure vary between implementations.

## sockaddr_in

The **socaddr_in** data structure is a platform-dependent socket address. You can find more information in `WINSOCK.`

## CONDVAR

The **CONDVAR** data structure is a platform-independent implementation of a condition variable. Details of its structure may vary between implementations.

## CRITICAL

The **CRITICAL** data structure is a platform-independent implementation of a critical-section variable. Details of its structure may vary between implementations.

## SYS_THREAD

The **SYS_THREAD** data structure is a platform-independent implementation of a system-thread variable. Details of its structure may vary between implementations.

## CacheEntry

The CacheEntry data structure holds all the information about one cache entry. It is created by the **ce_lookup** function and destroyed by the **ce_free** function.

```
typedef struct _CacheEntry {
   CacheState  state;/* state of the cache file; DO NOT refer to any
               * of the other fields in this C struct if state
               * is other than
               *    CACHE_REFRESH or
               *    CACHE_RETURN_FROM_CACHE
               */
SYS_FILE    fd_in;/* do not use: open cache file for reading */
int         fd_out;/* do not use: open (locked) cache file for writing */
struct stat finfo;/* stat info for the cache file */

unsigned char  digest[CACHE_DIGEST_LEN];/* MD5 for the URL */
char * url_dig;/* URL used to for digest; field #8 in CIF */
char *url_cif; /* URL read from CIF file */
char *filname; /* Relative cache file name */
char *dirname; /* Absolute cache directory name */
char *absname; /* Absolute cache file path */
char *lckname; /* Absolute locked cache file path */
char *cifname; /* Absolute CIF path */
int   sect_idx;/* Cache section index */
int   part_idx;/* Cache partition index */
CSect *section;/* Cache section that this file belongs to */
CPart *partition;/* Cache partition that this file belongs to */
int   xfer_time;/* secs *//* Field #2 in CIF */
time_t last_modified;/* GMT *//* Field #3 in CIF */
time_t expires;/* GMT *//* Field #4 in CIF */
time_t last_checked;/* GMT *//* Field #5 in CIF */
long  content_length;   /* Field #6 in CIF */
char *content_type;     /* Field #7 in CIF */
int   is_auth; /* Authenticated data -- always do recheck */
int   auth_sent;/* Client did send the Authorization header */
longmin_size;  /* Min size for a cache file (in KB) */
longmax_size;  /* Max size for a cache file (in KB) */
time_t last_accessed;/* GMT for proxy, local for gc */
time_tcreated; /* localtime (only used by gc, st_mtime) */
```

```
int   removed; /* gc only; file was removed from disk */
long  bytes;   /* from stat(), using this we get hdr len */
long  bytes_written;/* Number of bytes written to disk */
long  bytes_in_media;/* real fs size taken up */
long  blks;    /* size in 512 byte blocks */
int   category;/* Value category; bigger is better */
int   cif_entry_ok;/* CIF entry found and ok */
time_t      ims_c;/* GMT; Client -> proxy if-modified-since */
time_tstart_time;/* Transfer start time */
int   inhibit_caching;/* Bad expires/other reason not to cache */
int         corrupt_cache_file;/* Cache file gone corrupt => remove */
int   write_aborted;/* True if the cache file write was aborted */
int         batch_update;/* We're doing batch update (no real user) */
char *cache_exclude;/* Hdrs not to write to cache (RE) */
char *cache_replace;/* Hdrs to replace with fresh ones from 304 response (RE) */
char *cache_nomerge;/* Hdrs not to merge with the cached ones (RE) */
Session * sn;
Request * rq;
} CacheEntry;
```

## CacheState

The CacheState data structure is actually an enumerated list of constants. Aways use their names, because values would be subject to implementation change.

```
typedef enum {
CACHE_EXISTS_NOT = 0,/* Internal flag -- do not use! */
CACHE_EXISTS,  /* Internal flag -- do not use! */
CACHE_NO,      /* No caching: don't read, don't write cache */
CACHE_CREATE,  /* Create cache; don't read */
CACHE_REFRESH, /* Refresh cache; read if not modified */
CACHE_RETURN_FROM_CACHE,/* Return directly, no check */
CACHE_RETURN_ERROR/* With connect-mode=never when not in cache */
} CacheState;
```

## ConnectMode

The ConnectMode data structure is actually an enumerated list of constants. Aways use their names, because values would be subject to implementation change.

```
typedef enum {
CM_NORMAL = 0,/* normal -- retrieve/refresh when necessary */
CM_FAST_DEMO,/* fast -- retrieve only if not in cache already */
CM_NEVER /* never -- never connect to network */
} ConnectMode;
```

Chapter

6

# Examples of Custom SAFs

This chapter contains source code examples of custom Sever Application Functions (SAFs) for each directive in the request-response process. You may wish to use these examples as the basis for implementing your own custom SAFs. For more information about creating your own custom SAFs, see Chapter 3, “Creating Custom SAFs”.

Before writing custom SAFs, you should be familiar with the request-response process, the built-in SAFs provided by NSAPI, and the NSAPI routines you can use to implement custom SAFs. See Chapter 1, “NSAPI Basics” Chapter 2, “Directives and Built-In SAFs”, and Chapter 4, “NSAPI Function Reference” for detailed information.

## Examples By Directive

The following sections contain source code examples of custom SAFs for each class of functionality. The `nsapi/examples/` subdirectory within the server root directory contains these examples.

# AuthTrans Directive

For information, see the `auth.c` file in the `nsapi/examples/` subdirectory of the server root directory.

# NameTrans Directive

For information, see the `ntrans.c` file in the `nsapi/examples/` subdirectory of the server root directory.

# PathCheck Directive

The example in this section demonstrates how to implement `restrict-by-acf`, a custom SAF for performing path checks. This SAF loads an access control list from a custom file when the server starts up. This information is loaded into static data structures, where it is then consulted by a PathCheck function to verify that the given user is from an allowed host.

The custom file is a list of allowable IP addresses, one per line. All others are denied access. For simplicity, the stdio library is used to scan the IP addresses from the file.

To load the shared object containing your functions and to perform some special initialization, you add the following lines at the beginning of the `obj.conf` file (the file extension designated by `<ext>` would be `so` under UNIX or `dll` under Windows NT):

```
Init fn=load-modules shlib=example.<ext> funcs=acf-init,restrict-by-acf
Init fn=acf-init file=/foo/bar/baz
```

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

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

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

# ObjectType Directive

The example in this section demonstrates how to implement `html2shtml`, a custom SAF for identifying which HTML files you want parsed. This SAF looks at the given file name. If it finds that it ends with `.html`, it looks for a file with the same base name, but with the extension `.shtml` instead. If it finds one, it will use that path and inform the server that the file is parsed HTML instead of regular HTML. Note that this requires an extra stat call for every HTML file accessed.

To load the shared object containing your functions, you add the following lines at the beginning of the `obj.conf` file (the file extension designated by `<ext>` would be `so` under UNIX or `dll` under Windows NT):

```
Init fn=load-modules shlib=example.<ext> funcs=html2shtml
```

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

```
ObjectType fn=html2shtml
```

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

# Service Directive

The example in this section demonstrates how to implement `send-images`, a custom SAF which replaces the doit.cgi demonstration available on the Netscape home pages. When a file is accessed as /dir1/dir2/foo.picgroup, `send-images` checks if the file is being accessed by Mozilla/1.1. If not, it will send a short error message. The file foo.picgroup contains a list of lines, each of which specifies a filename followed by a content-type (for example, `one.gif image/gif`).

To load the shared object containing your function, you add the following line at the beginning of the `obj.conf` file (the file extension designated by `<ext>` would be `so` under UNIX or `dll` under Windows NT):

```
Init fn=load-modules shlib=example.<ext> funcs=send-images
```

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

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

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

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

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

# AddLog Directive

The example in this section demonstrates how to implement `brief-log`, a custom SAF for logging only three items of information about a request: the IP address, the method, and the URI (for example, 198.93.95.99 GET /foo/bar/baz).

To load the shared object containing your functions and to perform some special initialization, you add the following lines at the beginning of the `obj.conf` file (the file extension designated by `<ext>` is `so` under UNIX or `dll` under Windows NT):

```
Init fn=load-modules shlib=/path/example.<ext>
   funcs=brief-init,brief-log
Init fn=brief-init file=/tmp/brief.log
```

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

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

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

Appendix

A

# HyperText Transfer Protocol

The HyperText Transfer Protocol (HTTP) is a protocol (a set of rules that describes how information is exchanged) that allows a web browser and a web server to “talk” to 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 request contains the following: HTTP method, Universal Resource Identifier (URI), and HTTP protocol version. The request may include some header information. The server processes the request and generates a response. The response contains the following: HTTP protocol version, HTP status code & reason phrase. The response may include some header information. Following is the requested data. The server then closes the connection.

The Netscape Enterprise ServerNetscape FastTrack Server 3.0 supports HTTP 1.1. Previous versions of the server supported HTTP 1.0. The server is conditionally compliant with the HTTP 1.1 proposed standard, as approved by the Internet Engineering Steering Group (IESG) and the Internet Engineering Task Force (IETF) HTTP working group. For more information on the criteria for being conditionally compliant, see the Hypertext Transfer Protocol—HTTP/1.1 specification (RFC 2068) at:

`http://www.ietf.org/html.charters/http-charter.html`

or

`ftp://ds.internic.net/rfc/rfc2068.txt`

This appendix 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`.

# Requests

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

- Request method, URI, and protocol version
- Request headers
- Request data

## Request method, URI, and protocol version

A browser can request information using a number of methods. The commonly used methods include the following:

- `GET`—Requests the specified document
- `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 are optional. Some commonly used request headers are shown in Table A.1.

Table A.1 Common request headers

| Request header | Description |
|---|---|
| `Accept` | The file types the browser can accept. |
| `Authorization` | Used if the browser wants to authenticate itself with a server; information such as the username and password are included. |

Table A.1 Common request headers

| Request header | Description |
|---|---|
| `User-agent` | The name and version of the browser software. |
| `Referer` | The URL of the document where the user clicked on the link. |
| `Host` | The Internet host and port number of the resource being requested. |

## Request data

If the browser has made a `POST` or `PUT` request, it will send data after the blank line following the request headers. If the browser sends a `GET` or `HEAD` request, there is no data to send.

# Responses

The server's response includes the following:

HTTP protocol version, status code, and reason phrase

Response headers

Response data

## HTTP protocol version, status code, and reason phrase

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

100-199 a provisional response.

200-299 a successful transaction.

300-399 the requested resource should be retrieved from a different location.

400-499 an error was caused by the browser.

500-599 a serious error occurred in the server.

Table A.2 Common HTTP status codes

| Status code | Meaning |
|---|---|
| `200` | OK; successful transaction. |
| `302` | Found. Redirection to a new URL. The original URL has moved. This 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 is not an error. |
| `401` | Unauthorized. The user requested a document but didn't provide a valid username or password. |
| `403` | Forbidden. Access to this URL is forbidden. |
| `404` | Not found. The document requested isn't on the server. This code can also be sent if the server has been told to protect the document by telling unauthorized people that it doesn't exist. |
| `500` | Server error. A server-related error occurred. The server administrator should check the server's error log to see what happened. |

## Response headers

The response headers contain information about the server and the response data. Common response headers are shown in Table A.3.

Table A.3 Common response headers

| Response header | Description |
|---|---|
| `Server` | The name and version of the web server. |
| `Date` | The current date (in Greenwich Mean Time). |
| `Last-modified` | The date when the document was last modified. |

Table A.3 Common response headers

| Response header | Description |
|---|---|
| `Expires` | The date when the document expires. |
| `Content-length` | The length of the data that follows (in bytes). |
| `Content-type` | The MIME type of the following data. |
| `WWW-authenticate` | Used during authentication and includes information that tells the browser software what is necessary for authentication (such as username and password). |

## Response data

The server sends a blank line after the last header. It then sends the response data which is typically an HTML file.

Appendix

B

# Wildcard Patterns

This appendix describes the format of wildcard patterns used by the Netscape Enterprise Server. These wildcards are used by various built-in SAFs (see Chapter 2, “Directives and Built-In SAFs”) and by some NSAPI functions (see Chapter 4, “NSAPI Function Reference”).

Wildcard patterns use special characters. If you want to use one of these characters without the special meaning, precede it with a backslash (\) character.

Table B.1 Wildcard patterns

| Pattern | Use |
|---|---|
| * | Match zero or more characters. |
| ? | Match exactly one occurrence of any character. |
| \| | An or expression. The substrings used with this operator can contain other special characters such as * or $. The substrings must be enclosed in parentheses, for example, (a\|b\|c), but the parentheses cannot be nested. |
| $ | Match the end of the string. This is useful in or expressions. |

Table B.1 Wildcard patterns

| Pattern | Use |
|---|---|
| [abc] | Match one occurrence of the characters a, b, or c. Within these expressions, the only character that needs to be treated as a special character is ]; all others are not special. |
| [a-z] | Match one occurrence of a character between a and z. |
| [^az] | Match any character except a or z. |
| *~ | This expression, followed by another expression, removes any pattern matching the second expression. |

Table B.2 Wildcard examples

| Pattern | Result |
|---|---|
| *.netscape.com | Matches any string ending with the characters .netscape.com. |
| (quark\|energy).netscape.com | Matches either quark.netscape.com or energy.netscape.com. |
| 198.93.9[23].??? | Matches a numeric string starting with either 198.93.92 or 198.93.93 and ending with any 3 characters. |
| *.* | Matches any string with a period in it. |
| *~netscape-* | Matches any string except those starting with netscape-. |
| *.netscape.com~quark.netscape.com | Matches any host from domain netscape.com except for a single host quark.netscape.com. |
| *.netscape.com~(quark\|energy\|neutrino).netscape.com | Matches any host from domain netscape.com except for hosts quark.netscape.com, energy.netscape.com, and neutrino.netscape.com. |
| *.com~*.netscape.com | Matches any host from domain com except for hosts from subdomain netscape.com. |

Appendix

C

# Time Formats

This appendix describes the format strings used for dates and times. These formats are used by the NSAPI function `util_strftime`, by some built-in SAFs such as `append-trailer`, and by server-parsed HTML (`parse-html`).

The formats are similar to those used by the `strftime` C library routine, but not identical.

Table C.1 Time formats

| Symbol | Meaning |
|---|---|
| %a | Abbreviated weekday name (3 chars) |
| %d | Day of month as decimal number (01-31) |
| %S | Second as decimal number (00-59) |
| %M | Minute as decimal number (00-59) |
| %H | Hour in 24-hour format (00-23) |
| %Y | Year with century, as decimal number, up to 2099 |
| %b | Abbreviated month name (3 chars) |
| %h | Abbreviated month name (3 chars) |
| %T | Time "HH:MM:SS" |

Table C.1 Time formats

| Symbol | Meaning |
|---|---|
| %X | Time "HH:MM:SS" |
| %A | Full weekday name |
| %B | Full month name |
| %C | "%a %b %e %H:%M:%S %Y" |
| %c | Date & time "%m/%d/%y %H:%M:%S" |
| %D | Date "%m/%d/%y" |
| %e | Day of month as decimal number (1-31) without leading zeros |
| %I | Hour in 12-hour format (01-12) |
| %j | Day of year as decimal number (001-366) |
| %k | Hour in 24-hour format (0-23) without leading zeros |
| %l | Hour in 12-hour format (1-12) without leading zeros |
| %m | Month as decimal number (01-12) |
| %n | line feed |
| %p | A.M./P.M. indicator for 12-hour clock |
| %R | Time "%H:%M" |
| %r | Time "%I:%M:%S %p" |
| %t | tab |
| %U | Week of year as decimal number, with Sunday as first day of week (00-51) |
| %w | Weekday as decimal number (0-6; Sunday is 0) |
| %W | Week of year as decimal number, with Monday as first day of week (00-51) |
| %x | Date "%m/%d/%y" |
| %y | Year without century, as decimal number (00-99) |
| %% | Percent sign |

Appendix

D

# Server-Parsed HTML

This appendix describes the commands used in server-parsed HTML. These commands are embedded into HTML files which are processed by the built-in SAF `parse-html`.

The server replaceseach command with data determined by the command and its attributes.

## Commands

The format for a command is:

```
<!--#command attribute1 attribute2 ... -->
```

The format for each `attribute` is a name-value pair such as:

```
name="value"
```

Commands and attribute names should be in lower case.

As you can see, the commands are “hidden” within HTML comments so they are ignored if not parsed by the server. Following are details of each command and its attributes.

# The config command

The `config` command initializes the format for other commands.

- The `errmsg` attribute defines a message sent to the client when an error occurs while parsing the file. This error is also logged in the error log file.
- The `timefmt` attribute determines the format of the date for the `flastmod` command. It uses the same format characters as the `util_strftime()` function. Refer to Appendix C, “Time Formats” for detail about time formats. The default time format is: `"%A, %d-%b-%y %T"`.
- The `sizefmt` attribute determines the format of the file size for the `fsize` command. It may have one of these values:
  - `bytes` to report file size as a whole number in the format 12,345,678.
  - `abbrev` to report file size as a number of KB or MB. This is the default.

**Example**

```
<!--#config timefmt="%r %a %b %e, %Y" sizefmt="abbrev"-->
```

This sets the date format like 08:23:15 AM Wed Apr 15, 1996, and the file size format to the number of KB or MB of characters used by the file.

# The include command

The `include` command inserts a file into the parsed file (it can’t be a CGI program). You can nest files by including another parsed file, which then includes another file, and so on. The user requesting the parsed document must also have access to the included file if your server uses access control for the directories where they reside.

- The `virtual` attribute is the URI of a file on the server.
- The `file` attribute is a relative path name from the current directory. It may not contain elements such as `../` and it may not be an absolute path.

**Example**

```
<!--#include file="bottle.gif"-->
```

# The echo command

The `echo` command inserts the value of an environment variable. The `var` attribute specifies the environment variable to insert. If the variable is not found, "(none)" is inserted. See below for additional environment variables.

**Example** `<!--#echo var="DATE_GMT"-->`

# The fsize command

The `fsize` command sends the size of a file. The attributes are the same as those for the `include` command (`virtual` and `file`). The file size format is determined by the `sizefmt` attribute in the `config` command.

**Example** `<!--#fsize file="bottle.gif"-->`

# The flastmod command

The `flastmod` command prints the date a file was last modified. The attributes are the same as those for the `include` command (`virtual` and `file`). The date format is determined by the `timefmt` attribute in the `config` command.

**Example** `<!--#flastmod file="bottle.gif"-->`

# The exec command

The `exec` command runs a shell command or CGI program.

- The `cmd` attribute (Unix only) runs a command using `/bin/sh`. You may include any special environment variables in the command.
- The `cgi` attribute runs a CGI program and includes its output in the parsed file.

**Example** `<!--#exec cgi="workit.pl"-->`

# Environment variables in commands

In addition to the normal set of environment variables used in CGI, you may include the following variables in your parsed commands:

DOCUMENT_NAMEis the file name of the parsed file.

DOCUMENT_URIis the virtual path to the parsed file (for example, `/shtml/test.shtml`).

QUERY_STRING_UNESCAPEDis the unescaped version of any search query the client sent with all shell-special characters escaped with the \ character.

DATE_LOCALis the current date and local time.

DATE_GMTis the current date and time expressed in Greenwich Mean Time.

LAST_MODIFIEDis the date the file was last modified.

# Index

## A

## B



## C

## D



## E



## F

## G



## H



## I



## K



## L

## M



## N



## O



## P

## Q



## R

## S

## T



## U

## V



## W