Contents

About This Book

This book was last updated 8/12/99.

This book discusses how to use Netscape Server Application Programmer's Interface (NSAPI) to build plugins that define Server Application Functions (SAFs) to extend and modify the Enterprise Server versions 3.x and 4.0. The book also discusses the purpose and use of the configuration files `obj.conf`, `magnus.conf` and `mime.types`, and provides comprehensive lists of the directives and functions that can be used in these configuration files. It also provides a reference of the NSAPI functions you can use to define new plugins.

This book has the following chapters and appendices:

- Chapter 1, "Basics of Enterprise Server Operation."

  This chapter discusses how the Enterprise Server uses configuration files to perform initialization tasks and to process client requests.

- Chapter 2, "Syntax and Use of Obj.conf."

  This chapter goes into detail on the configuration file `obj.conf`. The chapter discusses the syntax and use of directives in this file, which instruct the server how to process requests.

- Chapter 3, "Predefined SAFS for Each Stage in the Request Handling Process."

  This chapter discusses each of the stages in the request handling process, and provides an API reference of the Server Application Functions (SAFs) that can be invoked at each stage.

- Chapter 4, "Creating Custom SAFs."

  This chapter discusses how to create your own plugins that define new SAFs to modify or extend the way the server handles requests.

- Chapter 5, "NSAPI Function Reference."

  This chapter presents a reference of the functions in the Netscape Server Application Programming Interface (API). You use NSAPI functions to define SAFs.

- Chapter 6, "Examples of Custom SAFsS."

  This chapter discusses examples of custom SAFs to use at each stage in the request handling process.

- Appendix A, “Data Structure Reference.”

  This appendix discusses some of the commonly used NSAPI data structures.

- Appendix B, “Variables in magnus.conf.”

  This appendix discusses the variables you can set in the configuration file `magnus.conf` to configure the Enterprise Server during initialization.

- Appendix C, “MIME Types.”

  This appendix discusses the MIME types file, which maps file extensions to file types.

- Appendix D, “Wildcard Patterns.”

  This appendix lists the wildcard patterns you can use when specifying values in `obj.conf`, various predefined SAFs and in some NSAPI functions.

- Appendix E, “Time Formats.”

  This appendix lists time formats.

- Appendix F, “Server-Parsed HTML Tags.”

  This appendix discusses the syntax and use of server-parsed HTML tags.

- Appendix G, “HyperText Transfer Protocol.”

  This appendix gives an overview of HTTP.

- Appendix H, “Alphabetical List of NSAPI Functions and Macros,”
  Appendix I, “Alphabetical List of Directives in magnus.conf,”
  Appendix J, “Alphabetical List of Pre-defined SAFs.”

  These appendices provide alphabetical lists for easy lookup of NSAPI functions, predefined SAFs, and variables in `magnus.conf`.

Chapter

1

Basics of Enterprise Server Operation

The configuration and behavior of Enterprise Server 4.0 is determined by a set of configuration files. You can change the settings in these configuration files either by using the Server Manager interface or by manually editing the files.

The configuration file that contains instructions for how the server processes requests from clients is called `obj.conf`. You can modify and extend the request handling process by adding or changing the instructions in `obj.conf`. You can use the Netscape Server Application Programming Interface (API) to create new Server Application Functions (SAFs) to use in instructions in `obj.conf`.

This chapter discusses the configuration files used by the Enterprise Server. Then the chapter looks in more detail at the server's process for handling requests. The chapter closes by introducing the use of Netscape Server Application Programming Interface (NSAPI) to define new functions to modify the request-handling process.

- Configuration Files
- How the Server Handles Requests from Clients
- Using NSAPI to Write New Server Application Functions

Configuration Files

The configuration and operation of the Enterprise Server is controlled by configuration files. The configuration files reside in the directory *server-root/server-id*/config/. This directory contains various configuration files for controlling different components, such as jsa.conf for configuring server-side JavaScript and netshare.conf for configuring NetShare. The exact number and names of configuration files depends on which components have been enabled or loaded into the server.

However, this directory always contains three configuration files that are essential for the server to operate. These files are:

- magnus.conf -- contains server initialization information.
- obj.conf -- contains instructions for handling requests from clients.
- mime.types -- contains information for determining the content type of requested resources.

magnus.conf

This file sets values of variables that configure the server during initialization. The server looks at this file and executes the settings on startup. The server does not look at this file again until it is restarted.

See Appendix B, “Variables in magnus.conf,” for a list of all the variables that can be set in magnus.conf.

obj.conf

This file contains additional initialization information, and also contains instructions for the server about how to process requests from clients (such as browsers). The server looks at this file every time it processes a request from a client.

The obj.conf file is essential to the operation of the Enterprise Server. When you make changes to server through the Server Manager interface, the system automatically updates obj.conf.

The file `obj.conf` contains a series of instructions (directives) that tell the Enterprise Server what to do at each stage in the request-response process. Each directive invokes a Server Application Function (SAF). These functions are written using the Netscape Server Application Programming Interface (NSAPI). The Enterprise Server comes with a set of pre-defined SAFs, but you can also write your own using NSAPI to create new instructions that modify the way the server handles requests.

For more information about how the server uses `obj.conf`, see Chapter 2, “Syntax and Use of Obj.conf.”.

mime.types

This file maps file extensions to MIME types, to enable the server to determine the content type of a requested resource. For example, requests for resources with `.html` extensions indicate that the client is requesting an HTML file, while requests for resources with `.gif` extensions indicate that the client is requesting an image file in GIF format.

The server loads the `mime.types` file when it starts up. If you make changes to this file, you must restart the server before the changes will take effect.

For more information about how the server uses `mime.types`, see Appendix C, “MIME Types.”.

How the Server Handles Requests from Clients

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

For more information about the HTTP protocol refer to Appendix G, “HyperText Transfer Protocol,”and also the latest HTTP specification.

HTTP Basics

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

- the client (usually a 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 line indicating a method such as `GET` or `POST`, a Universal Resource Identifier (URI) indicating which resource is being requested, and an 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.

The example request shown below would be sent by a Netscape browser to request the server to send back the resource in `/index.html`. In this example, no body data is sent because the method is GET (the point of the request is to get some data, not to send it.)

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

The server receives the request and processes it. It handles each request individually, although it may process many requests simultaneously. Each request is broken down into a series of steps that together make up the request handling process.

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/4.0
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 client how the server handled the request. Normally the status code 200 is returned indicating that the request was handled successfully and the body data 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.

Steps in the Request Handling Process

When the server first starts up it performs some initialization and then waits for an HTTP request from a client (such as a browser). When it receives a request, it handles it in the following steps:

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 check that the requestor has access privileges to the requested resource on the file system.

4. **ObjectType** (object typing)

   determine the MIME-type (Multi-purpose Internet Mail Encoding) of the requested resource (for example. text/html, image/gif, and so on).

5. **Service** (generate the response)

   generate and return the response to the client.

6. **AddLog** (adding log entries)

   add entries to log file(s).

7. **Error** (service)

   This step is executed only if an error occurs in the previous steps. If an error occurs, log an error message and abort the process.

Directives for Handling Requests

The file `obj.conf` contains a series of instructions, known as directives, that tell the Enterprise Server what to do at each stage in the request handling process. Each directive invokes a Server Application Function (SAF) with one or more arguments. Each directive applies either to initialization or to a specific stage in the request handling process. The stages are `Init`, `AuthTrans`, `NameTrans`, `PathCheck`, `ObjectType`, `Service`, and `AddLog`.

For example, the following directive applies during the `NameTrans` stage. It calls the `document-root` function with the `root` argument set to `D:/Netscape/Server4/docs`. (The `document-root` function translates the `http://`*`server_name/`* part of the URL to the document root, which in this example is `D:/Netscape/Server4/docs`.)

```
NameTrans fn="document-root" root="D:/Netscape/Server4/docs"
```

The functions invoked by the directives in `obj.conf` are known as Server Application Functions (SAFs).

Using NSAPI to Write New Server Application Functions

The Enterprise Server comes with a variety of pre-defined SAFs that you can use to create more directives in `obj.conf`. You can also write your own SAFs using the functions provided by the NSAPI. After writing a SAF, you would add a directive to `obj.conf` so that your new function gets invoked by the server at the appropriate time.

Each SAF has its own arguments, which are passed to it by the directive in `obj.conf`. Every SAF is also passed additional arguments that contain information about the request (such as what resource was requested and what

kind of client requested it) and any other server variables created or modified by SAFs called by previously invoked directives. Each SAF may examine, modify, or create server variables.

Each SAF returns a result code which tells the server whether it succeeded, did nothing, or failed.

For more information about `obj.conf`, see Chapter 2, “Syntax and Use of Obj.conf.”.

For more information on the pre-defined SAFs, see Chapter 3, “Predefined SAFS for Each Stage in the Request Handling Process.”.

For more information on writing your own SAFs, see Chapter 4, “Creating Custom SAFs.”

Chapter

2

Syntax and Use of Obj.conf

The `obj.conf` configuration file contains directives that instruct the Enterprise Server how to handle requests from clients. This chapter discusses server instructions in `obj.conf`; the use of `OBJECT` and `CLIENT` tags; the flow of control in `obj.conf`; and the syntax rules for editing `obj.conf`.

The sections in this chapter are:

- Server Instructions in obj.conf
- Object and Client Tags
- Flow of Control in obj.conf
- Syntax Rules for Editing obj.conf

Server Instructions in obj.conf

The `obj.conf` file contains two kinds of directives:

- directives that initialize the Enterprise Server . These directives appear at the start of the file, and are not embedded inside `OBJECT` tags.
- directives that instruct the server how to handle requests received from clients such as browser. These directives appear inside `OBJECT` tags.

Each directive calls a function, indicating when to call it and specifying arguments for it.

The syntax of each directive is:

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

For example:

```
NameTrans fn="document-root" root="D:/Netscape/Server4/docs"
```

`Directive` indicates when this instruction is executed, which is either during server initialization or during a step in the request handling process. If it is to be executed during server initialization, the value is `Init`. Otherwise the value is one of `AuthTrans`, `NameTrans`, `PathCheck`, `ObjectType`, `Service`, `Error`, and `AddLog`.

The value of the `fn` argument is the name of the Server Application Function to execute. All directives must supply a value for the `fn` parameter -- if there's no function, the instruction won't do anything.

The remaining parameters are the arguments needed by the function, and they vary from function to function.

Enterprise Server is shipped with a set of built-in server application functions (SAFs) such as `load-types`, `basic-auth`, and so on, that you can use to create and modify directives in `obj.conf`. You can also define new SAFs, as discussed in Chapter 4, "Creating Custom SAFs.".

Summary of the Directives

Here are the categories of server directives and a description of what each does. Each category corresponds to a stage in the request handling process (except for the Init category which corresponds to the server initialization stage).The section "Flow of Control in obj.conf" explains exactly how the server decides which directive or directives to execute in at each stage.

- **`Init`**

  Initializes server subsystems and shared resources. For example:

  ```
  Init fn="load-types" mime-types="mime.types"
  ```

  This example calls the function `load-types` to load the file `mime.types`, which the server will use for looking up MIME types.

- **`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.

```
AuthTrans fn=basic-auth userfn=ntauth auth-type=basic
userdb=none
```

This example calls the `basic-auth` function, which calls a custom function (in this case `ntauth`, to verify authorization information sent by the client. The Authorization header is sent as part of the basic server authorization scheme.

- **`NameTrans`**

  Translates the URL specified in the request from a logical URL to a physical file system path for the requested resource. This may also result in redirection to another site. For example:

  ```
  NameTrans fn="document-root" root="D:/Netscape/Server4/docs"
  ```

  This example calls the `document-root` function with a `root` argument of `"D:/Netscape/Server4/docs"`. The function `document-root` function translates the `"http://`*`server_name/`*`"` part of the requested to URL to the document root, which in this case is `D:/Netscape/Server4/docs`. Thus a request for `http://`*`server-name`*`/doc1.html` is translated to `D:/Netscape/Server4/docs/doc1.html`.

- **`PathCheck`**

  Performs tests on the physical path determined by the `NameTrans` step. In general, these tests determine whether the path is valid and whether the client is allowed to access the requested resource. For example:

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

  This example calls the `find-index` function with an `index-names` argument of `"index.html,home.html"`. If the requested URL is a directory, this function instructs the server to look for a file called either `index.html` or `home.html` in the requested directory.

- **`ObjectType`**

  Determines the MIME (Multi-purpose Internet Mail Encoding) type of the requested resource. The MIME type has attributes `type` (which indicates content type), `encoding` and `language`. The MIME type is sent in the headers of the response to the client. The MIME type also helps determine which `Service` directive the server should execute.

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`".

For example:

```
ObjectType fn="type-by-extension"
```

This example calls the `type-by-extension` function which causes the server to determine the MIME type according to the requested resource's file extension.

- **`Service`**

  Generates and 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 generating and sending the response data. The default response is to invoke the `send-file` function to send the contents of the requested file along with the appropriate header files to the client.

  The default `Service` directive is:

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

  This directive instructs the server to call the `send-file` function in response to any request whose method is `GET`, `HEAD`, or `POST`, and whose `type` does not begin with `magnus-internal/`. (Note the use of the special characters `*~` to mean "does not match".)

  Another example is:

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

  In this case, if the method of the request is either `GET` or `HEAD`, and the type of the requested resource is "`magnus-internal/imagemap`", the function `imagemap` is called.

- **`AddLog`**

  Adds an entry to a log file to record information about the transaction. For example:

  ```
  AddLog fn="flex-log" name="access"
  ```

This example calls the `flex-log` function to log information about the current request in the log file named `access`.

- **`Error`**

  Handles an HTTP error. This directive is invoked if a previous directive results in an error. Typically the server handles an error by sending a custom HTML document to the user describing the problem and possible solutions.

  For example:

  ```
  Error fn="send-error" reason="Unauthorized"
  path="D:/netscape/server4/errors/unauthorized.html"
  ```

  In this example, the server sends the file in `"D:/netscape/server4/errors/unauthorized.html"` whenever a client requests a resource that it is not authorized to access.

Object and Client Tags

This section discusses the use of `Object` and `Client` tags in the file `obj.conf`. `Object` tags group together directives that apply to requests for particular resources, while `Client` tags group together directives that apply to requests received from particular clients.

- The Object Tag
- The Client Tag

The Object Tag

Directives in the `obj.conf` file are grouped into objects that begin with an `<Object>` tag and end with a `</Object>` tag. The default object provides instructions to the server about how to process requests by default. Each new object modifies the default object's behavior.

An `Object` tag may have a `name` attribute or a `ppath` attribute. Either parameter may be a wildcard pattern. For example:

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

or

```
<Object ppath="/usr/netscape/server4/docs/private/*">
```

The server always starts handling a request by processing the directives in the default object. However, the server switches to processing directives in another object after the `NameTrans` stage of the default object if either of the following conditions is true:

- The successful `NameTrans` directive specifies a `name` argument
- the physical pathname that results from the `NameTrans` stage matches the `ppath` attribute of another object

When the server has been alerted to use an object other than the default object, it processes the directives in the other object before processing the directives in the default object. For some steps in the process, the server stops processing directives in that a particular stage (such as the `Service` stage) as soon as one is successfully executed, whereas for other stages the server processes all directives in that stage, including the ones in the default object as well as those in the additional object. For more details, see the section "Flow of Control in obj.conf."

Objects that Use the Name Attribute

If a `NameTrans` directive in the default object specifies a `name` argument, the server switches to processing the directives in the object of that name before processing the remaining directives in the default object.

For example, the following `NameTrans` directive in the default object assigns the name `cgi` to any request whose URL starts with `http://`*`server_name`*`/cgi/`.

```
<Object name="default">
NameTrans fn="pfx2dir" from="/cgi" dir="D:/netscape/server4/docs/mycgi"
name="cgi"
...
</Object>
```

When that `NameTrans` directive is executed, the server starts processing directives in the object named `cgi`:

```
<Object name="cgi">
more directives...
</Object>
```

Object that Use the Ppath Attribute

When the server finishes processing the `NameTrans` directives in the default object, the logical URL of the request will have been converted to a physical pathname. If this physical pathname matches the `ppath` attribute of another object in `obj.conf`, the server switches to processing the directives in that object before processing the remaining ones in the default object.

For example, the following `NameTrans` directive translates the `http://` *`server_name`*`/` part of the requested URL to `D:/Netscape/Server4/docs/` (which is the document root directory).

```
<Object name="default">
NameTrans fn="document-root" root="D:/Netscape/Server4/docs"
...
</Object>
```

The URL `http://`*`server_name`*`/internalplan1.html` would be translated to `D:/Netscape/Server4/docs/internalplan1.html`. However, suppose that `obj.conf` contains the following additional object:

```
<Object ppath="*internal*">
more directives...
</Object>
```

In this case, the partial path `*internal*` matches the path `D:/Netscape/Server4/docs/`**`internal`**`plan1.html`. So now the server starts processing the directives in this object before processing the remaining directives in the default object.

The Client Tag

The `<Client>` tag may be used within an object to limit a group of directives to requests received from specific clients. Directives between a `<Client>` tag and a matching `</Client>` tag are executed only 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 client that sent the current request matches all 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 clients such as Netscape browsers includes a "Host" header in the request which tells the DNS name of the server that the user requested.

Flow of Control in obj.conf

This section discusses how the server decides which directives to execute in obj.conf.

Init

When the Enterprise Server starts up, it executes the variable settings defined in magnus.conf, then executes the Init directives in obj.conf. The Init section contains directives that initialize the server, such as loading and initializing additional modules and plugins, and initializing log files.

The server executes all the directives in the Init section.

The Init section should always contain a directive that invokes the load-types function. This function loads the MIME types file that the server uses to create a table that maps file extensions to MIME types. The file is usually called mime.types. We don't recommend that you change the name of the MIME types file since most people expect it to be called mime.types. The following directive loads the MIME types file:

```
Init fn="load-types" mime-types="mime.types"
```

The most common way that the server determines the MIME type of a requested resource is by invoking the `type-by-extension` directive in the `ObjectType` section of `obj.conf`. This function will not work if the MIME types file has not been loaded.

AuthTrans

When the server receives a request, it executes the `AuthTrans` directives in the default object to check that the client is authorized to access the server.

If there is more than one AuthTrans directive, the server executes them all (unless one of them results in an error). If an error occurs, the server skips all other directives except for `Error` directives.

NameTrans

Next, the server executes a `NameTrans` directive in the default object to map the logical URL of the requested resource to a physical pathname on the server's file system. The server looks at each `NameTrans` directive in the default object in turn, until it finds one that can be applied.

If there is more than one `NameTrans` directive in the default object, the server considers each directive until one succeeds.

The `NameTrans` section in the default object must contain exactly one directive that invokes the `document-root` function. This functions translates the `http://`*`/server_name/`* part of the requested URL to a physical directory that has been designated as the server's document root. For example:

```
NameTrans fn="document-root" root="D:/Netscape/Server4/docs"
```

The directive that invokes `document-root` must be the last directive in the `NameTrans` section so that it is executed if no other `NameTrans` directive is applicable.

The `pfx2dir` (prefix to directory) function is used to set up additional mappings between URLs and directories. For example, the following directive translates the URL `http://`*`server_name`*`/cgi/` into the directory pathname `D:/netscape/server4/docs/mycgi/`:

```
NameTrans fn="pfx2dir" from="/cgi" dir="D:/netscape/server4/docs/mycgi"
```

Notice that if this directive appeared *after* the one that calls `document-root`, it would never be executed, with the result that the resultant directory pathname would be `D:/netscape/server4/docs/cgi/` (not `mycgi`). This illustrates why the directive that invokes `document-root` must be the last one in the `NameTrans` section.

How the Server Knows to Process Other Objects

As a result of executing a `NameTrans` directive, the server might start processing directives in another object. This happens if the `NameTrans` directive that was successfully executed specifies a name or generates a partial path that matches the `name` or `ppath` attribute of another object.

If the successful `NameTrans` directive assigns a name by specifying a `name` argument, the server starts processing directives in the named object (defined with the `OBJECT` tag) before processing directives in the default object for the rest of the request handling process.

For example, the following `NameTrans` directive in the default object assigns the name `cgi` to any request whose URL starts with `http://`*`server_name`*`/cgi/`.

```
<Object name="default">
...
NameTrans fn="pfx2dir" from="/cgi" dir="D:/netscape/server4/docs/mycgi"
name="cgi"
...
</Object>
```

When that `NameTrans` directive is executed, the server starts processing directives in the object named `cgi`:

```
<Object name="cgi">
more directives...
</Object>
```

When a `NameTrans` directive has been successfully executed, there will be a physical pathname associated with the requested resource. If the resultant pathname matches the `ppath` (partial path) attribute of another object, the server starts processing directives in the other object before processing directives in the default object for the rest of the request handling process.

For example, suppose `obj.conf` contains an object as follows:

```
<Object ppath="*internal*">
more directives...
```

```
</Object>
```

Now suppose the successful `NameTrans` directive translates the requested URL to the pathname `D:/Netscape/Server4/docs/internalplan1.html`. In this case, the partial path `*internal*` matches the path `D:/Netscape/Server4/docs/internal`plan1.html. So now the server would start processing the directives in this object before processing the remaining directives in the default object.

PathCheck

After converting the logical URL of the requested resource to a physical pathname in the `NameTrans` step, the server executes `PathCheck` directives to verify that the client is allowed to access the requested resource.

If there is more than one `PathCheck` directive, the server executes all the directives in the order in which they appear, unless one of the directives denies access. If access is denied, the server switches to executing directives in the Error section.

If the `NameTrans` directive assigned a name or generated a physical pathname that matches the `name` or `ppath` attribute of another object, the server first applies the `PathCheck` directives in the matching object before applying the directives in the default object.

ObjectType

Assuming that the `PathCheck` directives all approve access, the server next executes the `ObjectType` directives to determine the MIME type of the request. The MIME type has three attributes: type, encoding, and language. When the server sends the response to the client, the type, language, and encoding values are transmitted in the headers of the response. The `type` also frequently helps the server to determine which `Service` directive to execute to generate the response to the client.

If there is more than one `ObjectType` directive, the server applies all the directives in the order in which they appear. However, once a directive sets an attribute of the MIME type, further attempts to set the same attribute are

ignored. The reason that all `ObjectType` directives are applied is that one directive may set one attribute, for example `type`, while another directive sets a different attribute, such as `language`.

As with the `PathCheck` directives, if another object has been matched to the request as a result of the `NameTrans` step, the server executes the `ObjectType` directives in the matching object before executing the `ObjectType` directives in the default object.

Setting the Type By File Extension

Usually the default way the server figures out the MIME type is by calling the `type-by-extension` function. This function instructs the server to look up the MIME type according to the requested resource's file extension in the MIME types table. This table was created during the `Init` stage by the `load-mime-types` function, which loads the MIME types file, (which is usually called `mime.types`).

For example, the entry in the MIME types table for the extensions `.html` and `.htm` is usually:

```
type=text/html  exts=htm,html
```

which says that all files that have the extension `.htm` or `.html` are text files formatted as HTML and the `type` is `text/html`.

Note that since the server creates the MIME types table during initialization, if you make changes to the MIME types file, you must restart the server before those changes can take effect.

For more information about MIME types, see Appendix C, "MIME Types."

Forcing the Type

If no previous `ObjectType` directive has set the type, and the server does not find a matching file extension in the `MIME` types table, the `type` still has no value even after `type-by-expression` has been executed. Usually if the server does not recognize the file extension, it is a good idea to force the type to be `text/plain`, so that the content of the resource is treated as plain text. There are also other situations where you might want to set the type regardless of the file extension, such as forcing all resources in the designated CGI directory to have the MIME type `magnus-internal/cgi`.

The function that forces the type is `force-type`.

For example, the following directives first instruct the server to look in the MIME types table for the MIME type, then if the `type` attribute has not been set (that is, the file extension was not found in the MIME types table), set the `type` attribute to `text/plain`.

```
ObjectType fn="type-by-extension"
ObjectType fn="force-type" type="text/plain"
```

If the server receives a request for a file `abc.dogs`, it looks in the MIME types table, does not find a mapping for the extension `.dogs`, and consequently does not set the `type` attribute. Since the `type` attribute has not already been set, the second directive is successful, forcing the `type` attribute to `text/plain`.

The following example illustrates another use of `force-type`. In this example, the `type` is forced to `magnus-internal/cgi` before the server gets a chance to look in the MIME types table. In this case, all requests for resources in `http://`*`server_name`*`/cgi/` are translated into requests for resources in the directory `D:/netscape/server4/docs/mycgi/`. Since a name is assigned to the request, the server processes `ObjectType` directives in the object named `cgi` before processing the ones in the default object. This object has one `ObjectType` directive, which forces the `type` to be `magnus-internal/cgi`.

```
NameTrans fn="pfx2dir" from="/cgi" dir="D:/netscape/server4/docs/mycgi"
name="cgi"
```

```
<Object name="cgi">
ObjectType fn="force-type" type="magnus-internal/cgi"
Service fn="send-cgi"
</Object>
```

The server continues processing all `ObjectType` directives including those in the default object, but since the `type` attribute has already been set, no other directive can set it to another value.

Service

Next, the server needs to execute a `Service` directive to generate the response to send to the client. The server looks at each `Service` directive in turn, to find the first one that matches the type, method and query string. If a `Service` directive does not specify type, method, or query string, then the unspecified attribute matches anything.

If there is more than one `Service` directive, the server applies the first one that matches the conditions of the request, and ignores all remaining `Service` directives.

As with the `PathCheck` and `ObjectType` directives, if another object has been matched to the request as a result of the `NameTrans` step, the server considers the `Service` directives in the matching object before considering the ones in the default object. If the server successfully executes a `Service` directive in the matching object, it will not get round to executing the `Service` directives in the default object, since it only executes one `Service` directive.

Service Examples

For an example of how `Service` directives work, consider what happens when the server receives a request for the URL `D:/`*`server_name`*`/jos.html`. In this case, all directives executed by the server are in the default object.

- The following `NameTrans` directive translates the requested URL to `D:/netscape/server4/docs/jos.html`:

  ```
  NameTrans fn="document-root" root="D:/Netscape/Server4/docs"
  ```

- Assume that the `PathCheck` directives all succeed.

- The following `ObjectType` directive tells the server to look up the resource's MIME type in the MIME types table:

  ```
  ObjectType fn="type-by-extension"
  ```

- The server finds the following entry in the MIME types table, which sets the type attribute to `text/html`:

  ```
  type=text/html exts=htm,html
  ```

- The server invokes the following `Service` directive. The value of the `type` parameter matches anything that does *not* begin with `magnus-internal/`. (For a list of all wildcard patterns, see Appendix D, "Wildcard Patterns.") This directive sends the requested file, `jos.html`, to the client.

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

For an example that involves using another object, consider what happens when the server receives a request for `http://`*`server_name`*`/servlet/doCalculation.class`. This example assumes that servlets have been

activated and the directory `D://netscape/server4/docs/servlet/` has been registered as a servlet directory (that is, the server treats all files in that directory as servlets).

- The following `NameTrans` directive translates the requested URL to `D:netscape/Server4/docs/servlet/doCalculation.class`. This directive also assigns the name `ServletByExt` to the request.

  ```
  NameTrans fn="pfx2dir" from="/servlet"
  dir="D:/Netscape/Server4/docs/servlet" name="ServletByExt"
  ```

- As a result of the `name` assignment, the server switches to processing the directives in the object named `ServletByExt`. This object is defined as:

  ```
  <Object name="ServletByExt">
  ObjectType fn="force-type" type="magnus-internal/servlet"
  Service type="magnus-internal/servlet" fn="NSServletService"
  </Object>
  ```

- The `ServletByExt` object has no `PathCheck` directives, so the server processes the `PathCheck` directives in the default object. Let's assume that all `PathCheck` directives succeed.

- Next, the server processes the `ObjectType` directives, starting with the one in the `ServletByExt` object. This directive sets the `type` attribute to `magnus-internal/servlet`.

  ```
  ObjectType fn="force-type" type="magnus-internal/servlet"
  ```

  The server continues processing all the `ObjectType` directives in the default object, but since the `type` attribute is already set its value cannot be changed.

- When processing `Service` directives, the server starts by considering the `Service` directive in the `ServletByExt` object which is:

  ```
  Service type="magnus-internal/servlet" fn="NSServletService"
  ```

- The `type` argument of this directive matches the `type` value that was set by the `ObjectType` directive. So the server goes ahead and executes this `Service` directive which calls the `NSServletService` function. This function invokes the requested file as a servlet and sends the output from the servlet as the response to the client. (If the requested resource is not a servlet, an error occurs.)

Since a Service directive has now been executed, the server does not process any other Service directives. (However, if the matching object had *not* had a Service directive that was executed, the server would continue looking at Service directives in the default object.)

Default Service Directive

There is usually a Service directive that does the default thing (sends a file) if no other Service directive matches a request sent by a browser. This default directive should come last in the list of Service directives in the default object, to ensure it only gets called if no other Service directives have succeeded. The default Service directive is usually:

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

This directive matches requests whose method is GET, HEAD, or POST, which covers nearly virtually all requests sent by browsers. The value of the type argument uses special pattern-matching characters. For complete information about the special pattern-matching characters, see Appendix D, “Wildcard Patterns.”.

The characters "*~" mean "anything that doesn’t match the following characters", so the expression "*~magnus-internal/" means "anything that doesn’t match "magnus-internal/". An asterisk by itself matches anything, so the whole expression "*~magnus-internal/*" matches anything that does not begin with "magnus-internal/".

So if the server has not already executed a Service directive when it reaches this directive, it executes the directive so long as the request method is GET, HEAD or POST, and the value of the type attribute does not begin with "magnus-internal/". The invoked function is send-file, which simply sends the contents of the requested file to the client.

AddLog

After the server generate the response and sends it to the client, it executes AddLog directives to add entries to the log files.

All AddLog directives are executed. The server can add entries to multiple log files.

Depending on which log files are used and which format they use, the `Init` section may need to have directives that initialize the logs. For example, if one of the `AddLog` directives calls `flex-log`, which uses the extended log format, the `Init` section must contain a directive that invokes `flex-init` to initialize the flexible logging system.

For more information about initializing logs, see the discussion of the functions `flex-init` and `init-clf` in Chapter 3, "Predefined SAFS for Each Stage in the Request Handling Process."

Error

If an error occurs during the request handling process, such as if a `PathCheck` or `AuthTrans` directive denies access to the requested resource, or the requested resource does not exist, then the server immediately stops executing all other directives and immediately starts executing the `Error` directives.

Syntax Rules for Editing obj.conf

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

Order of Directives

The order of directives is important, since the server executes them in the order they appear in `obj.conf` . The outcome of some directives affect the execution of other directives.

For `PathCheck` directives, the order within the `PathCheck` section is not so important, since the server executes all `PathCheck` directives. However, in the `ObjectType` section the order is very important, because if an `ObjectType` directive sets an attribute value, no other `ObjectType` directive can change that value. For example, if the default `ObjectType` directives were listed in the following order (which is the wrong way round), every request would have its `type` value set to `text/plain`, and the server would never have a chance to set the `type` according to the extension of the requested resource.

```
ObjectType fn="force-type" type="text/plain"
ObjectType fn="type-by-extension"
```

Similarly, the order of directives in the `Service` section is very important. The server executes the first `Service` directive that matches the current request and does not execute any others.

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 manually add comments to `obj.conf`, then use the Server Manager interface to make changes to your server, the Server Manager will wipe out your comments when it updates `obj.conf`.

Chapter

3

Predefined SAFS for Each Stage in the Request Handling Process

This chapter describes the directives and pre-defined Server Application Functions (SAFs) that are provided as standard with the Enterprise Server. They are used in the `obj.conf` file to give instructions to the server. For a discussion of the use and syntax of `obj.conf`, see the previous chapter, Chapter 2, "Syntax and Use of Obj.conf."

This chapter includes functions that are part of the core functionality of Enterprise Server. It does not include functions that are available only if additional components, such as servlets, web publishing, WAI, and server-parsed HTML are enabled.

The functions and arguments described here are applicable to Enterprise 3.x and 4.0. Functions and arguments that are new to Enterprise Server 4.0 are indicated as such.

This chapter contains a section for each directive which lists all the pre-defined Server Application Functions that can be used with that directive.

The directives are:

- `Init Stage`
- `AuthTrans Stage`
- `NameTrans Stage`
- `PathCheck Stage`
- `ObjectType Stage`
- `Service Stage`
- `AddLog Stage`
- `Error Stage`

For an alphabetical list of pre-defined SAFs, see Appendix J, "Alphabetical List of Pre-defined SAFs."

The following table lists the SAFs that can be used with each directive.

Table 3.1

Init Stage	`cache-init`
	`cindex-init`
	`dns-cache-init`
	`flex-init`
	`flex-rotate-init`
	`init-cgi`
	`init-clf`
	`init-uhome`
	`load-modules`
	`load-types`
	`pool-init`
	`thread-pool-init`

AuthTrans Stage	`basic-auth`
	`basic-ncsa`
	`get-sslid`

NameTrans Stage	`assign-name`
	`document-root`
	`home-page`
	`pfx2dir`
	`pfx2dir`
	`redirect`
	`unix-home`

PathCheck Stage	`cert2user`
	`check-acl`
	`deny-existence`
	`find-index`
	`find-links`
	`find-pathinfo`
	`get-client-cert`
	`load-config`
	`nt-uri-clean`
	`ntcgicheck`
	`require-auth`
	`ssl-check`
	`ssl-logout`
	`unix-uri-clean`

Table 3.1

ObjectType Stage	`force-type`
	`image-switch`
	`shtml-hacktype`
	`type-by-exp`
	`type-by-extension`

Service Stage	`add-footer`
	`add-header`
	`append-trailer`
	`imagemap`
	`index-common`
	`index-simple`
	`key-toosmall`
	`list-dir`
	`make-dir`
	`parse-html`
	`query-handler`
	`remove-dir`
	`remove-file`
	`rename-file`
	`send-cgi`
	`send-file`
	`send-range`
	`send-shellcgi`
	`send-wincgi`
	`upload-file`

AddLog Stage	`common-log`
	`flex-log`
	`record-useragent`

Error Stage	`send-error`

Init Stage

`Init` directives are invoked during server initialization when the server is started or restarted. These directives perform tasks such as initializing log files and loading plugins.

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`. The server logs the error according to the instructions in the `Error` directives, and terminates. Any other result code is considered a 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.
- `flex-rotate-init` enables rotation for flexible logs.
- `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.
- `thread-pool-init` configures an additional thread pool.

cache-init

Applicable in `Init`-class directives.

The `cache-init` function controls file caching for static files, such as HTML pages, GIF files and sound files. The server caches files to improve performance. If a request is received for a file that is in the cache, the server retrieves the requested resource from the cache, which is more efficient than retrieving it from its source. File caching is enabled by default.

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.

Files can be cached in various ways. Small files might be read into the heap space; larger files might be cached using memory-mapping; and in some circumstance files might be cached as open file descriptors.

Note In Enterprise Server 4.0, much of the functionality of the file cache is controlled by a new configuration file called `nsfc.conf`. For information about `nsfc.conf`, see the tuning chapter in the *Administrator's Guide for Enterprise Server 4.0*.

Parameters:

`disable`	(optional) specifies whether the file cache is disabled or not. If set to anything but "false" the cache is disabled. By default, the cache is enabled.
`PollInterval`	(optional) specifies how often the files in the cache are checked for changes. The default is 5 seconds. In Enterprise Server 4.0, this parameter is ignored -- use the `MaxAge` parameter in the `nsfc.conf` file instead.
`MaxNumberOfCachedFiles`	(optional) maximum number of entries in the accelerator cache. The default is 4096, minimum is 32, maximum is 32K.
`MaxNumberOfOpenCachedFiles`	(optional) Maximum number of memory-mapped cached files that can be open simultaneously. The default is 512, minimum is 32, maximum is 32.
`MaxCachedFileSize`	(optional) maximum size of a file that can be cached as a memory-mapped file. The default is 525K. In Enterprise Server 4.0, this parameter is ignored. Use the `MediumFileSizeLimit` parameter in `nsfc.conf` instead. In Enterprise Server 4.0, this parameter is ignored on NT because it no longer applies to the platform.
`MaxTotalCachedFileSize`	(optional) total size of all files that are cached as memory-mapped files. Default is 10K, minimum is 1K, maximum is 16M. In Enterprise Server 4.0, this parameter is ignored on Unix. Use the `MediumFileSpace` parameter in `nsfc.conf` instead. In Enterprise Server 4.0, this parameter is ignored on NT because it no longers applies to the platform.

`CacheHashSize`	(optional) size of hash table for the file cache accelerator. Default is 8192K, minimum is 32, max is 32K.

Example

```
Init fn=cache-init PollIntervale=2
MaxNumberOfCachedFiles=8192
```

cindex-init

Applicable in `Init`-class directives.

The function `cindex-init` sets the default settings for common indexing. Common indexing (also known as fancy indexing) is performed by the Service function `index-common`. Indexing occurs when the requested URL translates to a directory that does not contain an index file or home page, or no index file or home page has been specified.

In common (fancy) indexing, the directory list shows the name, last modified date, size and description for each indexed file or directory.

Parameters:

`opts` (optional) is a string of letters specifying the options to activate. Currently there is only one possible option:

- `s` tells the server to scan each HTML file in the directory being indexed for the contents of the HTML `<TITLE>` tag to display in the description field. The `<TITLE>` tag must be within the first 255 characters of the file. This option is off by default.

Note: In Enterprise Server 3.x and previously, the search for the `<TITLE>` tag is case sensitive. In Enterprise Server 4.x, the search is no longer case-sensitive.

`widths`	(optional) specifies the width for each column in the indexing display. 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. Note: In Enterprise Server 3.x and previous versions, the `widths` parameter does not work properly. It basically acts as a flag, since the actual widths (for non-zero values) are hardcoded. However, in Enterprise Server 4.x, the widths parameter works correctly. The default values in Enterprise Server 4.0 are 22,18,8,33. The final three values (corresponding to last-modified date, size, and description respectively) can each be set to 0 to turn the display for that column off. The name column cannot be turned off. The minimum size of a column (if the value is non-zero) is specified by the length of its title -- for example, the minimum size of the Date column is 5 (the length of "Date" plus one space). If you set a non-zero value for a column which is less than the length of its title, the width defaults to the minimum required to display the title.
`timezone`	(optional) **Enterprise Server 4.x only.** This indicates whether the last-modified time is shown in local time or in Greenwich Mean Time. The values are `GMT` or `local`. The default is `local`.
`format`	(optional) **Enterprise Server 4.x only.** This parameter determines the format of the last modified date display. It uses the format specification for the UNIX function `strftime()`. The default is `"%d-%b-%Y %H:%M"`
`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 is `/mc-icons/`. If `icon-uri` is different from the default, 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=*private*
Init fn=cindex-init widths=22,0,0,50
```

See Also `index-common, find-index, home-page`

dns-cache-init

Applicable in `Init`-class directives.

The `dns-cache-init` function specifies that DNS lookups should be cached when DNS lookups are enabled. If DNS lookups are cached, then when the server gets a client's host name information, it stores that information in the DNS cache. If the server needs information about the client in the future, the information is available in the DNS cache.

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

Applicable in `Init`-class directives.

The `flex-init` function opens the named log file to be used for flexible logging and establishes a record format for it. The log format is recorded in the first line of the log file. You cannot change the log format while the log file is in use by the server.

The `flex-log` function writes entries into the log file during the `AddLog` stage of the request handling process.

The log file stays open until the server is shut down or restarted (at which time all logs are closed and reopened).

Note: If the server has `AddLog Stage` directives that call `flex-log`, the flexible log file must be initialized by `flex-init` during server initialization.

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.

The `flex-init` function may be called more than once. Each new log file name and format will be added to 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). The exception to this rule is if log rotation has been enabled in Enterprise Server 4.0.

For information on rotating log files, see `flex-rotate-init`.

The `flex-init` function has three parameters: one that names the log file, one that specifies the format of each record in that file, and one that specifies the logging mode.

Parameters

`logFileName`	The name of the parameter is the name of the log file. The value of the parameter specifies either the full path to the log file or a file name relative to the server's `logs` directory. For example:
	`access="/usr/netscape/server4/https-servername/logs/access"`
	`mylogfile = "log1"`
	You will use the log file name later, as a parameter to the `flex-log` function.

`format.`*`logFileName`*	specifies the format of each log entry in the log file. For information about the format, see the section "More on Log Format" below.

`relaxed.`*`logFileName`* **New in Enterprise Server 4.0.**

If you turn on relaxed logging and the logged component is one that would normally block static page acceleration, the server skips logging the component (instead it puts a blank in the log file) if static page acceleration is enabled. However, if static page acceleration is not enabled, the server logs the full value of the component.

If the value is `"true"`, `"on"`, `"yes"` or `"1"` relaxed logging is on otherwise it is off.

More on Log Format

The `flex-init` function recognizes anything contained between percent signs (%) as the name portion of a name-value pair stored in a parameter block in the server. (The one exception to this rule is the %SYSDATE% component which delivers the current system date.) %SYSDATE% is formatted using the time format "%d/%b/%Y:%H:%M:%S" plus the offset from GMT.

(See Chapter 4, “Creating Custom SAFs” for more information about parameter blocks and Chapter 5, “NSAPI Function Reference” for functions to manipulate pblocks.)

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 3.2. Certain components might contain spaces, so they should be bounded by escaped quotes (\")

If no format parameter 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%"
```

New in Enterprise Server 4.0: you can now log cookies by logging the `Req->headers.cookie.`*`name`* component.

Enterprise Server can use cache acceleration for serving static pages (as discussed in `cache-init`). However, some components of log format entries block this acceleration (unless the logging mode is relaxed) causing the server to use the unaccelerated path for serving static pages. (The server always uses the unaccelerated path to serve dynamically-generated pages.) The following table indicates which components of the log format entry allow static page acceleration to proceed for the current request. If the log

format uses any components that do not allow static page acceleration, the performance of serving static pages may decrease significantly (unless the logging mode is relaxed).

In the following table, the components that are enclosed in escaped double quotes (\") are the ones that could potentially resolve to values that have white spaces.

Table 3.2 Typical components of flex-init formatting

Flex-log option	Component	Allows static page acceleration
Client Host name (unless "iponly" is specified in flex-log or DNS name is not available) or IP address	`%Ses->client.ip%`	Yes
Client DNS name	`%Ses->client.dns%`	Yes
System date	`%SYSDATE%`	Yes
Full HTTP request line	`\"%Req->reqpb.clf-request%\"`	Yes
Status	`%Req->srvhdrs.clf-status%`	Yes
Response content length	`%Req->srvhdrs.content-length%`	Yes
Response content type	`%Req->srvhdrs.content-type%`	Yes
Referer header	`\"%Req->headers.referer%\"`	Yes
User-agent header	`\"%Req->headers.user-agent%\"`	Yes
HTTP Method	`%Req->reqpb.method%`	Yes
HTTP URI	`%Req->reqpb.uri%`	Yes

Table 3.2 Typical components of flex-init formatting

Flex-log option	Component	Allows static page acceleration
HTTP query string	`%Req->reqpb.query%`	Yes
HTTP protocol version	`\"%Req->reqpb.protocol%\"`	Yes
Accept header	`%Req->headers.accept%`	No
Date header	`\"%Req->headers.date%\"`	No
If-Modified-Since header	`%Req->headers.if-modified-since%`	No
Authorization header	`%Req->headers.authorization%`	Yes
Any header value	`\"%Req->headers.`*`headername`*`%\"`	No (unless otherwise indicated for specific header names)
Name of authorized user	`%Req->vars.auth-user%`	Yes
Value of a cookie	`\"%Req->headers.cookie.`*`name`*`%\"`	No
Value of any variable in `Req->vars`	`\"%Req->vars.`*`varname`*`%\"`	No

Examples The first example below initializes flexible logging into the file `/usr/netscape/server4/https-servername/logs/access`.

```
Init fn=flex-init access="/usr/netscape/server4/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.

The following example initializes flexible logging into the file `/user/netscape/server4/https-servername/logs/extended`.

```
Init fn=flex-init extended="/usr/netscape/server4/
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-rotate-init, flex-log`

flex-rotate-init

Applicable in `Init`-class directives. **New in Enterprise Server 4.0.**

The `flex-rotate-init` function enables log rotation for logs that use the flexible logging format. Call this function in the `Init` stage of `obj.conf` before calling `flex-init`. The `flex-rotate-init` function allows you to specify a time interval for rotating log files. At the specified time interval, the server moves the log file to a file whose name indicates the time of moving. The `flex-log` function in the `AddLog` stage then starts logging entries in a new log file. The server does not need to be shut down while the log files are being rotated.

Note that the server keeps all rotated log files forever, so you will need to clean them up as necessary to free up disk space.

By default, log rotation is disabled.

Parameters

`rotate-start`	Indicates the time to start rotation. This value is a 4 digit string indicating the time in 24 hour format, for example, 0900 indicates 9 am while 1800 indicates 9 pm.
`rotate-interval`	Indicates the number of minutes to elapse between each log rotation.

Example This example enables log rotation, starting at midnight and occurring every hour.

```
Init fn=flex-rotate-init rotate-start=2400
rotate-intervals=60
```

See Also `flex-init, flex-log`

init-cgi

Applicable in `Init`-class directives.

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

Parameters

`timeout`	(optional) specifies how many seconds the server waits 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.
`env-variable`	(optional) specifies the name and value for an environment variable that the server places 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

Applicable in `Init`-class directives.

The `init-clf` function opens the named log files to be used for common logging. The `common-log` function writes entries into the log files during the `AddLog` stage of the request handling process. 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).

Note: If the server has an `AddLog Stage` directive that calls `common-log`, common log files must be initialized by `init-clf` during the `Init` stage.

Note: 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 for Unix, send the `-HUP` signal) 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

`logFileName`	The name of the parameter is the name of the log file. The value of the parameter specifies either the full path to the log file or a file name relative to the server's `logs` directory. For example:
	`access="/usr/netscape/server4/https-servername/logs/access"`
	`mylogfile = "log1"`
	You will use the log file name later, as a parameter to the `common-log` function.

Examples

```
Init fn=init-clf
access=/usr/netscape/server4/https-boots/logs/access
Init fn=init-clf
templog=/tmp/mytemplog templog2=/tmp/mytemplog2
```

See Also `common-log, record-useragent`

init-uhome

Applicable in `Init`-class directives.

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

Applicable in `Init`-class directives.

The `load-modules` function loads a shared library or Dynamic Link Library into the server code. Specified functions from the library can then be executed from any subsequent directives. Use this function to load new plugins or SAFs.

If you define your own Server Application Functions, you get the server to load them by using the `load-modules` function and specifying the shared library or dll to load.

Parameters

`shlib`	specifies either the full path to the shared library or 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 or dynamic link library to be made available for use by other `Init` or `Service` directives in `obj.conf`. 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`.
`pool`	the name of a custom thread pool, as specified in `thread-pool-init`.

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

Applicable in `Init`-class directives.

The `load-types` function loads the file that the server uses to look up mime types.

More explicitly, this function uses the indicated file to create a table that maps file-name extensions to a file's content-type , content-encoding, and content-language. During the `ObjectType` phase, the function `type-by-extension` instructs the server to look in this table to determine the type of content requested by the client, based on the extension of the requested resource.

If you edit the MIME types file, you will need to restart the server to load the changes.

The file name extensions 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()` NSAPI functions to work properly.

Note: MIME types files must begin with the following line or they will not be accepted:`#--Netscape Communications Corporation MIME Information`

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.

Examples

```
Init fn=load-types mime-types=mime.types
Init fn=load-types mime-types=mime.types
local-types=/usr/netscape/server4/local.types
```

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

pool-init

Applicable in `Init`-class directives.

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
```

thread-pool-init

Applicable in `Init`-class directives.

This function creates a new pool of user threads. To tell a plugin to use the new pool, specify the `pool` parameter when loading the plugin with the Init-class function `load-modules`.

One reason to create a custom thread pool would be if a plugin is not thread-aware, in which case you can set the maximum number of threads in the pool to 1.

Parameters

`name`	name of the thread pool.
`maxthreads`	maximum number of threads in the pool.
`minthreads`	minimum number of threads in the pool.
`queueSize`	size of the queue for the pool. If all the threads in the pool are busy, further request-handling threads that want to get a thread from the pool will wait in the pool queue. The number of request-handling threads that can wait in the queue is limited by the queue size. If the queue is full, the next request-handling thread that comes to the queue is turned away, with the result that the request is turned down, but the request-handling thread remains free to handle another request instead of becoming locked up in the queue.

Example

```
Init fn=thread-pool-init name="my-custom-pool"
maxthreads=100 minthreads=1 queuesize=200
```

```
Init fn=load-modules shlib="C:/mydir/myplugin.dll"
funcs="tracker" pool="my-custom-pool"
```

See also `load-modules`

AuthTrans Stage

`AuthTrans` stands for Authorization Translation. `AuthTrans` directives give the server instructions for checking authorization before allowing a client to access resources. `AuthTrans` directives work in conjunction with `PathCheck` directives. Generally, an `AuthTrans` function checks if the username and password associated with the request are acceptable, but it does not allow or deny access to the request -- it leaves that to a `PathCheck` function.

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

- AuthTrans Directive - validates authorization information sent by the client in the Authorization header.
- PathCheck Stage - checks that the authorized user is allowed access to the requested resource.

The authorization process 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.

`AuthTrans` functions get the username and password from the headers associated with the request. When a client initially makes a request, the username and password are unknown so the `AuthTrans` functions and `PathCheck` functions work together to reject the request, since they can't validate the username and password. When the client receives the rejection, its usual response is to pop up a dialog box asking for the username and password to enter the appropriate realm, and then the client submits the request again, this time including the username and password in the headers.

If there is more than one `AuthTrans` directive in `obj.conf`, each function is executed in order until one succeeds in authorizing the user.

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 determines 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.

- `get-sslid` retrieves a string that is unique to the current SSL session and stores it as the `ssl-id` variable in the `Session->client` parameter block.

basic-auth

Applicable in `AuthTrans`-class directives.

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 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`	(optional) specifies the full path and file name of the user database to be used for user verification. This parameter will be passed to the user function.
`userfn`	is the name of the user custom function to verify authorization. This function 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

PathCheck fn=require-auth auth-type=basic realm="Marketing
Plans"
```

See Also `require-auth`

basic-ncsa

Applicable in `AuthTrans`-class directives.

The `basic-ncsa` function verifies authorization information sent by the client against a database. The Authorization header is sent as part of 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/server4/userdb/rs
PathCheck fn=require-auth auth-type=basic
realm="Marketing Plans"
```

```
AuthTrans fn=basic-ncsa auth-type=basic
userfile=/netscape/server4/.htpasswd
grpfile=/netscape/server4/.grpfile
PathCheck fn=require-auth auth-type=basic
realm="Marketing Plans"
```

See Also `require-auth`

get-sslid

Applicable in `AuthTrans`-class directives.

The `get-sslid` function retrieves a string that is unique to the current SSL session, and stores it as the `ssl-id` variable in the `Session->client` parameter block.

If the variable `ssl-id` is present when a CGI is invoked, it is passed to the CGI as the `HTTPS_SESSIONID` environment variable.

The `get-sslid` function has no parameters and always returns `REQ_NOACTION`. It has no effect if SSL is not enabled.

Note: Enterprise Server 4.x incorporates the functionality of `get-sslid` into the standard processing of an SSL connection, so there should no longer be a need to use `get-sslid` as of Enterprise Server 4.0.

Parameters

none

NameTrans Stage

`NameTrans` stands for Name Translation. `NameTrans` directives translate 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/server4/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 one in order until one succeeds.

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

- `assign-name` tells the server to process directives in a named object.
- `document-root` translates a URL into a file system path by replacing the `http://`*`server-name/`* part of the requested resource with the document root directory.
- `home-page` translates a request for the server's root home page (`/`) to a specific file.
- `pfx2dir` translates any URL beginning with a given prefix to a file system directory and optionally enables directives in an additional named object .
- `redirect` redirects the client to a different URL.
- `unix-home` translates a URL to a specified directory within a user's home directory.

assign-name

Applicable in `NameTrans`-class directives.

The `assign-name` function specifies the name of an object in `obj.conf` that matches the current request. The server then processes the directives in the named object in preference to the ones in the default object.

For example, consider the following directive in the default object:

```
NameTrans fn=assign-name name=personnel from=/personnel
```

Let's suppose the server receives a request for `http://`*`server-name/`*`personnel`. After processing this `NameTrans` directive, the server looks for an object named `personnel` in `obj.conf`, and continues by processing the directives in the `personnel` object.

The `assign-name` function always returns `REQ_NOACTION`,

Parameters

`from`	is a wildcard pattern that specifies the path to be affected.
`name`	specifies an additional named object in `obj.conf` whose directives will be applied to this request.

Example

```
# This NameTrans directive is in the default object.
NameTrans fn=assign-name
name=personnel from=/a/b/c/pers

...
<Object name=personnel>
...additional directives..
</Object>
```

document-root

Applicable in `NameTrans`-class directives.

The `document-root` function specifies the root document directory for the server. If the physical path has not been set by a previous `NameTrans` function, the `http://`*`server-name/`* part of the path is replace by the physical pathname for the document root.

When the server receives a request for `http://`*`server-name`*`/somepath/somefile`, the `document-root` function replaces `http://`*`server-name/`* with the value of its `root` parameter. For example, if the document root directory is `/usr/netscape/server4/docs` , then when the server receives a request for

`http://server-name/a/b/file.html`, the `document-root` function translates the pathname for the requested resource to `/usr/netscape/server4/docs/a/b/file.html`.

This function always returns `REQ_PROCEED`. `NameTrans` directives listed after this will never be called, so be sure that the directive that invokes `document-root` is the last `NameTrans` directive.

There can be only one root document directory. To specify additional document directories, use the `pfx2dir` function to set up additional path name translations.

Parameters

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

Examples

```
NameTrans fn=document-root root=/usr/netscape/server4/docs
```

See also `pfx2dir`

home-page

Applicable in `NameTrans`-class directives.

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 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"
```

pfx2dir

Applicable in `NameTrans`-class directives.

The `pfx2dir` function replaces a directory prefix in the requested URL with a real directory name. It also optionally allows you to specify the name of an object that matches the current request. (See the discussion of `assign-name` for details of using named objects)

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) specifies an additional named object in `obj.conf` whose directives will be applied to this request.

Examples In the first example, the URL `http://`*`server-name`*`/cgi-bin/`*`resource`* (such as `http://x.y.z/cgi-bin/test.cgi`) is translated to the physical pathname `/httpd/cgi-local/`*`resource`*, (such as `/httpd/cgi-local/test.cgi`) and the server also starts processing the directives in the object named `cgi`.

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

In the second example, the URL `http://`*`server-name`*`/icons/`*`resource`* (such as `http://x.y.z/icons/happy/smiley.gif`) is translated to the physical pathname `/users/nikki/images/`*`resource`*, (such as `/users/nikki/images/smiley.gif`)

```
NameTrans fn=pfx2dir from=/icons/happy
dir=/users/nikki/images
```

redirect

Applicable in `NameTrans`-class directives.

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, the server treats the request as a request for the new URL.

Parameters

`from`	specifies the prefix of the requested URI to match.
`url`	(maybe 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`	(maybe optional) is the new URL prefix to return to the client. The `from` prefix is simply replaced by this URL prefix. If you use this parameter, don't use `url` (and vice-versa).
`escape`	(optional) is a flag which tells the server to `util_uri_escape()` the URL before sending it. It should be `yes` or `no`. The default is `yes`.

Examples

In the first example, any request for `http://`*`server-name/whatever`* is translated to a request for `http://tmpserver/`*`whatever`*.

```
NameTrans fn=redirect from=/ url-prefix=http://tmpserver
```

In the second example, any request for `http://`*`server-name/`*`toopopular/`*`whatever`* is translated to a request for `http://bigger/better/stronger/morepopular/`*`whatever`*.

```
NameTrans fn=redirect from=/toopopular
url=http://bigger/better/stronger/morepopular
```

unix-home

Applicable in `NameTrans`-class directives.

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 are not 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 is 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 Stage

`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 information 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 resource 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 information 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.
- `ssl-check` checks the secret keysize.
- `ssl-logout` invalidates the current SSL session in the server's SSL session cache.
- `unix-uri-clean` denies access to requests with unsafe path names by indicating not found.

cert2user

Applicable in `PathCheck`-class directives.

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: • Obtaining and verifying a certificate from the user • Obtaining a user name and password using WWW basic authentication. • 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, the request fails.
PathCheck fn="cert2user"
userdb="/usr/netscape/server4/authdb/default"
require="1"
```

check-acl

Applicable in `PathCheck`-class directives.

The `check-acl` function specifies an Access Control List (ACL) to use to check whether the client is allowed to access the requested resource. An access control list contains information about who is or is not allowed to access a resource, and under what conditions access is allowed.

Regardless of the order of `PathCheck` directives in the object, `check-acl` functions are executed first. They 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 denies access.

Examples

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

deny-existence

Applicable in `PathCheck`-class directives.

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 cannot 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 `personal.com` domain:

```
<Client dns=*~.personal.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/server4/docs/
private
```

```
PathCheck fn=deny-existence bong-msg=/svr/msg/go-away.html
```

find-index

Applicable in `PathCheck`-class directives.

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.

Note that if the file `obj.conf` has a `NameTrans` directive that calls `home-page`, and the requested directory is the root directory, then the home page rather than the index page, is returned to the client.

The `find-index` 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.

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

Applicable in `PathCheck`-class directives.

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

Applicable in `PathCheck`-class directives.

The `find-pathinfo` function finds any extra path information 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

Applicable in `PathCheck`-class directives.

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 to get a client certificate, if the server does not already have the client certificate. This typically causes the client to present a dialog box to the user to select a client certificate. The server may already have the client certificate if it was requested on the initial handshake, or if a cached SSL session has been resumed.
- 0 tells the function not to redo the SSL3 handshake if the server does not already have the client certificate.

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.

`require` — controls whether failure to get a client certificate will abort the HTTP request. If `require` is absent the default value is 1.

- 1 tells the function to abort the HTTP request if the client certificate is not present after `dorequest` is handled. In this case, the HTTP status is set to `PROTOCOL_FORBIDDEN`, and the function returns `REQ_ABORTED`.
- 0 tells the function to return `REQ_NOACTION` if the client certificate is not present after dorequest is handled.

`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 certificate from the session.
# If a certificate is not already associated with the
# session, request one.
# The request fails if the client does not present a
# valid certificate.

PathCheck fn="get-client-cert" dorequest="1"
```

load-config

Applicable in `PathCheck`-class directives.

The `load-config` function searches for configuration files in document directories and adds the file's contents to the server's existing configuration. These configuration files (also known as dynamic configuration files) specify additional access control information for the requested resource. Depending on the rules in the dynamic configuration files, the server might or might not allow the client to access the requested resource.

Each directive that invokes `load-config` is associated with a base directory, which is either stated explicitly through the `basedir` parameter or derived from the root directory for the requested resource. The base directory determines two things:

- the top-most directory for which requests will invoke this call to the `load-config` function.

  For example, if the base directory is `D:/Netscape/Server4/docs/nikki/`, then only requests for resources in this directory or its subdirectories (and their subdirectories and so on) trigger the search for dynamic configuration files. A request for the resource `D:/Netscape/Server4/docs/somefile.html` does not trigger the search in this case, since the requested resource is in a parent directory of the base directory.

- the top-most directory in which the server looks for dynamic configuration files to apply to the requested resource.

  If the base directory is `D:/Netscape/Server4/docs/nikki/`, the server starts its search for dynamic configuration files in this directory. It may or may not also search subdirectories (but never parent directories) depending on other factors.

When you enable dynamic configuration files through the Server Manager interface, the system writes additional objects with `ppath` parameters into the `obj.conf` file. If you manually add directives that invoke `load-config` to the default object (rather than putting them in separate objects), the Server Manager interface might not reflect your changes.

If you manually add `PathCheck` directives that invoke `load-config` to the file `obj.conf`, put them in additional objects (created with the `<OBJECT>` tag) rather than putting them in the default object. Use the `ppath` attribute of the `OBJECT` tag to specify the partial pathname for the resources to be affected by the access rules in the dynamic configuration file. The partial pathname can be any pathname that matches a pattern, which can include wildcard characters.

For example, the following `<OBJECT>` tag specifies that requests for resources in the directory `D:/Netscape/Server4/docs` are subject to the access rules in the file `my.nsconfig`.

```
<Object ppath="D:/Netscape/Server4/docs/*">
PathCheck fn="load-config" file="my.nsconfig" descend=1
basedir="D:/Netscape/Server4/docs"
</Object>
```

Note: If the `ppath` resolves to a resource or directory that is higher in the directory tree (or is in a different branch of the tree) than the base directory, the `load-config` function is not invoked. This is because the base directory specifies the highest-level directory for which requests will invoke the `load-config` function.

The `load-config` function returns `REQ_PROCEED` if configuration files were loaded, `REQ_ABORTED` on error, or `REQ_NOACTION` when no files are loaded.

Parameters

`file`	(optional) is the name of the dynamic configuration file containing the access rules to be applied to the requested resource. 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, such as "`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 dynamic 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 base directory. This is the highest-level directory for which requests will invoke the `load-config` function and is also the directory where the server starts searching for configuration files. If `basedir` is not specified, the base directory is assumed to be the root directory that results from translating the requested resource's URL to a physical pathname. For example, if the request was for `http://`*`server-name`*`/a/b/file.html`, the physical file name would be `/`*`document-root`*`/a/b/file.html`..

Examples In this example, whenever the server receives a request for any resource containing the substring `"secret"` that resides in `D:/Netscape/Server4/docs/nikki/` or a subdirectory thereof, it searches for a configuration file called `checkaccess.nsconfig`.

The server starts the search in the directory `D:/Netscape/Server4/docs/nikki`, and searches subdirectories too. It loads each instance of `checkaccess.nsconfig` that it finds, applying the access control rules contained therein to determine whether the client is allowed to access the requested resource or not.

```
<Object ppath="*secret*">
PathCheck fn="load-config" file="checkaccess.nsconfig"
basedir="D:/Netscape/Server4/docs/nikki" descend="1"
</Object>
```

nt-uri-clean

Applicable in `PathCheck`-class directives.

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

Applicable in `PathCheck`-class directives.

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

Applicable in `PathCheck`-class directives.

The `require-auth` function allows access to resources 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.

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.

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.

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`

ssl-check

Applicable in `PathCheck`-class directives. **New in Enterprise Server 4.0**.

If a restriction is selected that is not consistent with the current cipher settings under Security Preferences, this function opens a popup dialog which warns that ciphers with larger secret keysizes need to be enabled. This function is designed to be used together with a Client tag to limit access of certain directories to non-exportable browsers.

The function returns `REQ_NOACTION` if SSL is not enabled, or if the `secret-keysize` parameter is not specified. If the secret keysize for the current session is less than the specified `secret-keysize` and the `bong-file` parameter is not specified, the function returns `REQ_ABORTED` with a status of `PROTOCOL_FORBIDDEN`. If the bong file is specified, the function returns `REQ_PROCEED`, and the `path` variable is set to the bong filename. Also, when a

keysize restriction is not met, the SSL session cache entry for the current session is invalidated, so that a full SSL handshake will occur the next time the same client connects to the server.

Requests that use `ssl-check` are not cacheable in the accelerator file cache if `ssl-check` returns something other than `REQ_NOACTION`.

This function supersedes the `key-toosmall` Service-class function that was used in Enterprise Server prior to release 4.0.

Parameters

`secret-keysize`	(optional) is the minimum number of bits required in the secret key.
`bong-file`	(optional) is the name of a file (not a URI) to be served if the restriction is not met

ssl-logout

Applicable in `PathCheck`-class directives.

`ssl-logout` invalidates the current SSL session in the server's SSL session cache. This does not affect the current request, but the next time the client connects, a new SSL session will be created. If SSL is enabled, this function returns `REQ_PROCEED` after invalidating the session cache entry. If SSL is not enabled, it returns `REQ_NOACTION`.

Parameters

None.

unix-uri-clean

Applicable in `PathCheck`-class directives.

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 Stage

`ObjectType` directives determine the MIME type of the file to send to the client in response to a request. MIME attributes currently sent are `type`, `encoding`, and `language`. The MIME type sent to the client as the value of the `content-type` header.

`ObjectType` directives also set the `type` parameter, which is used by `Service` directives to determine how to process the request according to what kind of content is being requested.

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

The `obj.conf` file almost always has an `ObjectType` directive that calls the `type-by-extension` function. This function instructs the server to look in a particular file (the MIME types file) to deduce the content type from the extension of the requested resource.

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.
- `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

Applicable in `ObjectType`-class directives.

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

Make sure that the directive that calls this function comes last in the list of `ObjectType` directives so that all other `ObjectType` directives have a chance to set the MIME type first. If there is more than one `ObjectType` directive in an object, all the directives are applied in the order they appear. If a directive sets an attribute and later directives try to set that attribute to something else, the first setting is used and the subsequent ones ignored.

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

Applicable in `ObjectType`-class directives.

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. The use of 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

Applicable in ObjectType-class directives.

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

Applicable in `ObjectType`-class directives.

This function instructs the server to look in a table of MIME type mappings to find the MIME type of the requested resource according to the extension of the requested resource. The MIME type is added to the `content-type` header sent back to the client.

The table of MIME type mappings is created during the server's `Init` stage by the `load-types` function, which loads a MIME types file and creates the mappings. The MIME types file is usually called `mime.types`, but you can specify a different file by providing a different file name to `load-types`.

For example, the following two lines are part of the MIME types file:

```
type=text/html    exts=htm,html
type=text/plain    exts=txt
```

If the extension of the requested resource is `htm` or `html`, the `type-by-extension` file sets the type to `text/html`. If the extension is `txt`, the function sets the type to `text/plain`.

For more information about MIME types, see Appendix C, “MIME Types.”.

Parameters

None.

Examples

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

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

Service Stage

The `Service` class of functions sends the response data to the client.

Every `Service` directive has the following optional parameters to determine whether the function is executed. All 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.

By default, the server sends the requested file to the client by calling the `send-file` function. The directive that sets the default is:

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

This directive usually comes last in the set of `Service`-class directives to give all other Service directives a chance to be invoked. This directive is invoked if the method of the request is `GET`, `HEAD`, or `POST`, and the type does **not** start with `magnus-internal/`. Note here that the pattern `*~` means "does not match." For a list of characters that can be used in patterns, see Appendix D, “Wildcard Patterns.”.

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

- `add-footer` appends a footer specified by a filename or URL to a an HTML file.
- `add-header` prepends a header specified by a filename or URL to an HTML file.
- `append-trailer` appends text to the end of an HTML file.

- `imagemap` handles server-side image maps.
- `index-common` generates a fancy list of the files and directories in a requested directory.
- `index-simple` generates a simple list of 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.
- `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 response to the client.
- `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 response to the client.
- `send-wincgi` sets up environment variables, launches a WinCGI program, and sends the response to the client.
- `upload-file` uploads and saves a file.

add-footer

Applicable in `Service`-class directives. **New in Enterprise Server 4.0**.

This function appends a footer to an HTML file that is sent to the client. The footer is specified either as a filename or a URI -- thus the footer can be dynamically generated. To specify static text as a footer, use the `append-trailer` function.

Parameters

`file`	(optional) The pathname to the file containing the footer. Specify either `file` or `uri`. By default the pathname is relative. If the pathname is absolute, pass the `NSIntAbsFilePath` parameter as `"yes"`.

`uri`	(optional) A URI pointing to the resource containing the footer. Specify either `file` or `uri`.
`NSIntAbsFilePath`	(optional) if the file parameter is specified, the `NSIntAbsFilePath` parameter determines whether the file name is absolute or relative. The default is relative. Set the value to `"yes"` to indicate an absolute file path.
`type`	optional parameter common to all Service-class functions
`method`	optional parameter common to all Service-class functions
`query`	optional parameter common to all Service-class functions

Examples

```
Service type=text/html method=GET fn=add-footer
file="footers/footer1.html"
```

```
Service type=text/html method=GET fn=add-footer
file="D:/netscape/server4/footers/footer1.html"
NSIntAbsFilePath="yes"
```

See Also `append-trailer, add-header`

add-header

Applicable in `Service`-class directives. **New in Enterprise Server 4.0**.

This function prepends a header to an HTML file that is sent to the client. The header is specified either as a filename or a URI -- thus the header can be dynamically generated.

Parameters

`file`	(optional) The pathname to the file containing the header. Specify either `file` or `uri`. By default the pathname is relative. If the pathname is absolute, pass the `NSIntAbsFilePath` parameter as `"yes"`.
`uri`	(optional) A URI pointing to the resource containing the header. Specify either `file` or `uri`.
`NSIntAbsFilePath`	(optional) if the file parameter is specified, the `NSIntAbsFilePath` parameter determines whether the file name is absolute or relative. The default is relative. Set the value to `"yes"` to indicate an absolute file path.
`type`	optional parameter common to all Service-class functions

`method`	optional parameter common to all Service-class functions
`query`	optional parameter common to all Service-class functions

Examples

```
Service type=text/html method=GET fn=add-header
file="headers/header1.html"
```

```
Service type=text/html method=GET fn=add-footer
file="D:/netscape/server4/headers/header1.html"
NSIntAbsFilePath="yes"
```

See Also `add-footer, append-trailer`

append-trailer

Applicable in `Service`-class directives.

The `append-trailer` function sends an HTML file and appends text to the end. It only appends text to HTML files. 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 information after the file name in the URL, or if the file cannot be opened for read-only access.

Parameters

`trailer`	is the text to append to 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 E, “Time Formats.” If `timefmt` is not provided, `:LASTMOD:` will not be replaced with the time.
`type`	optional parameter common to all Service-class functions
`method`	optional parameter common to all Service-class functions
`query`	optional parameter common to all Service-class functions

Examples

```
Service type=text/html method=GET fn=append-trailer
trailer="<hr><img src=/logo.gif> Copyright 1999"

# Add a 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:"
```

See Also `add-footer, add-header`

imagemap

Applicable in `Service`-class directives.

The `imagemap` function responds to requests for imagemaps. Imagemaps are images which are divided into multiple areas that each have an associated URL. The information about which URL is associated with which area is stored in a mapping file.

Parameters

`type`	optional parameter common to all Service-class functions
`method`	optional parameter common to all Service-class functions
`query`	optional parameter common to all Service-class functions

Examples

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

index-common

Applicable in `Service`-class directives.

The `index-common` function generates a fancy (or common) list of files in the requested 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 `Init`-class function `cindex-init` specifies the format for the index list, including where to look for the images.

If `obj.conf` contains a call to `index-common` in the `Service` stage, it must initialize fancy (or common) indexing by invoking `cindex-init` during the `Init` stage.

Indexing occurs when the requested resource is a directory that does not contain an index file or a home page, or no index file or home page has been specified by the functions `find-index` or `home-page`.

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).
`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 (enclosed by <PRE> and </PRE>).
`type`	optional parameter common to all Service-class functions
`method`	optional parameter common to all Service-class functions

`query`	optional parameter common to all Service-class functions

Examples

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

See Also `cindex-init, index-simple, find-index, home-page`

index-simple

Applicable in `Service`-class directives.

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

Indexing occurs when the requested resource is a directory that does not contain either an index file or a home page, or no index file or home page has been specified by the functions `find-index` or `home-page`.

Parameters

`type`	optional parameter common to all Service-class functions
`method`	optional parameter common to all Service-class functions
`query`	optional parameter common to all Service-class functions

Examples

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

See Also `cindex-init, index-common`

key-toosmall

Applicable in `Service`-class directives. This function is deprecated in Enterprise Server 4.0. It is replaced by the PathCheck-class SAF `ssl-check`.

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

`type`	optional parameter common to all Service-class functions
`method`	optional parameter common to all Service-class functions
`query`	optional parameter common to all Service-class functions

Examples

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

list-dir

Applicable in `Service`-class directives.

The `list-dir` function returns a sequence of text lines to the client in response to a request whose method is INDEX. 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.

When remote file manipulation is enabled in the server, the `obj.conf` file contains a `Service`-class function that calls `list-dir` for requests whose method is `INDEX`.

Parameters

`type`	optional parameter common to all Service-class functions
`method`	optional parameter common to all Service-class functions
`query`	optional parameter common to all Service-class functions

Examples

```
Service fn=list-dir method="INDEX"
```

make-dir

Applicable in `Service`-class directives.

The `make-dir` function creates a directory when the client sends a request whose method is MKDIR. The function can fail if the server can't write to that directory.

When remote file manipulation is enabled in the server, the `obj.conf` file contains a `Service`-class function that invokes `make-dir` when the request method is `MKDIR`.

Parameters

`type`	optional parameter common to all Service-class functions
`method`	optional parameter common to all Service-class functions
`query`	optional parameter common to all Service-class functions

Examples

```
Service fn="make-dir" method="MKDIR"
```

parse-html

Applicable in `Service`-class directives.

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. Refer to Appendix F, "Server-Parsed HTML Tags," 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.
`type`	optional parameter common to all Service-class functions
`method`	optional parameter common to all Service-class functions
`query`	optional parameter common to all Service-class functions

Examples

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

query-handler

Applicable in `Service`-class directives.

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.
`type`	optional parameter common to all Service-class functions
`method`	optional parameter common to all Service-class functions
`query`	optional parameter common to all Service-class functions

Examples

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

remove-dir

Applicable in `Service`-class directives.

The `remove-dir` function removes a directory when the client sends an request whose method is `RMDIR`. 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 privileges to remove the directory.

When remote file manipulation is enabled in the server, the `obj.conf` file contains a `Service`-class function that invokes `remove-dir` when the request method is `RMDIR`.

Parameters

`type`	optional parameter common to all Service-class functions
`method`	optional parameter common to all Service-class functions
`query`	optional parameter common to all Service-class functions

Examples

```
Service fn="remove-dir" method="RMDIR"
```

remove-file

Applicable in `Service`-class directives.

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

When remote file manipulation is enabled in the server, the `obj.conf` file contains a `Service`-class function that invokes `remove-file` when the request method is `DELETE`.

Parameters

`type`	optional parameter common to all Service-class functions
`method`	optional parameter common to all Service-class functions
`query`	optional parameter common to all Service-class functions

Examples

```
Service fn="remove-file" method="DELETE"
```

rename-file

Applicable in `Service`-class directives.

The `rename-file` function renames a file when the client sends a request with a `New-URL` header whose method is `MOVE` . It renames the file indicated by the URL to `New-URL` within the same directory if the user is authorized and the server has the needed file system privileges.

When remote file manipulation is enabled in the server, the `obj.conf` file contains a `Service`-class function that invokes `rename-file` when the request method is `MOVE`.

Parameters

`type`	optional parameter common to all Service-class functions
`method`	optional parameter common to all Service-class functions
`query`	optional parameter common to all Service-class functions

Examples

```
Service fn="rename-file" method="MOVE"
```

send-cgi

Applicable in `Service`-class directives.

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 section "CGI to NSAPI Conversion" in Chapter 4, “Creating Custom SAFs”.

Parameters

`type`	optional parameter common to all Service-class functions
`method`	optional parameter common to all Service-class functions
`query`	optional parameter common to all Service-class functions

Examples

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

send-file

Applicable in `Service`-class directives.

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 using the following directive (which usually comes last in the list of `Service`-class directives in the default object so that it acts as a default)

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

This directive is invoked if the method of the request is `GET`, `HEAD`, or `POST`, and the type does **not** start with `magnus-internal/`. Note here that the pattern `*~` means "does not match." For a list of characters that can be used in patterns, see Appendix D, “Wildcard Patterns.”.

Parameters

`type`	optional parameter common to all Service-class functions
`method`	optional parameter common to all Service-class functions
`query`	optional parameter common to all Service-class functions

Examples

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

send-range

Applicable in `Service`-class directives.

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

Parameters

`type`	optional parameter common to all Service-class functions
`method`	optional parameter common to all Service-class functions
`query`	optional parameter common to all Service-class functions

Examples

```
Service fn=send-range
```

send-shellcgi

Applicable in `Service`-class directives.

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 to Enterprise Server 4.0.

Parameters

`type`	optional parameter common to all Service-class functions
`method`	optional parameter common to all Service-class functions
`query`	optional parameter common to all Service-class functions

Examples

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

send-wincgi

Applicable in `Service`-class directives.

Windows NT only. The `send-wincgi` 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 to Enterprise Server 4.0.

Parameters

`type`	optional parameter common to all Service-class functions
`method`	optional parameter common to all Service-class functions
`query`	optional parameter common to all Service-class functions

Examples

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

upload-file

Applicable in `Service`-class directives.

The `upload-file` function uploads and saves a new file when the client sends a request whose method is `PUT` if the user is authorized and the server has the needed file system privileges.

When remote file manipulation is enabled in the server, the `obj.conf` file contains a `Service`-class function that invokes `upload-file` when the request method is `PUT`.

Parameters

`type`	optional parameter common to all Service-class functions
`method`	optional parameter common to all Service-class functions
`query`	optional parameter common to all Service-class functions

Examples

```
Service fn=upload-file
```

AddLog Stage

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

Applicable in `AddLog`-class directives.

This function 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 for Enterprise Server. The common log must have been initialized previously by the `init-cgi` function.

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>
```

See Also `init-clf`

flex-log

Applicable in `AddLog`-class directives.

This function 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/flexanlg` directory for Enterprise Server.

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. For information about rotating logs, see `flex-rotate-init`.

Parameters

`name`	(optional) gives the name of a log file, which must have been given as a parameter to the `flex-init` 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-rotate-init`, `flex-init`, `init-clf`, `common-log`, `record-useragent`

record-useragent

Applicable in `AddLog`-class directives.

The `record-useragent` function records the IP address of the client, followed by its User-Agent HTTP header. This indicates 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 `flex-init`, `init-clf`, `common-log`, `record-useragent`, `flex-log`

Error Stage

If a server application function results in an error, it sets the HTTP response status code and returns the value `REQ_ABORTED`. When this happens, the server stops processing the request. Instead, it searches for an Error directive matching the HTTP response status code or its associated reason phrase, and executes the directive's function. If the server does not find a matching Error directive, it returns the response status code to the client.

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

Applicable in `Error`-class directives.

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.
`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`.

Examples

```
Error fn=send-error code=401
path=/netscape/server4/docs/errors/401.html
```

Chapter

4

Creating Custom SAFs

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

The sections in this chapter are:

- The SAF Interface
- SAF Parameters
- Result Codes
- Creating and Using Custom SAFs
- Overview of NSAPI C Functions
- Required Behavior of SAFs for Each Directive
- CGI to NSAPI Conversion

Before writing custom SAFs, you should familiarize yourself with the request handling process, as described in Chapter 1, "Basics of Enterprise Server Operation.". Also, before writing a custom SAF, check if a built-in SAF already accomplishes the tasks you have in mind. See Chapter 3, "Predefined SAFS for Each Stage in the Request Handling Process," for a list of the pre-defined SAFs.

For a complete list of the NSAPI routines for implementing custom SAFs, see Chapter 5, "NSAPI Function Reference."

The SAF Interface

All SAFs (custom and built-in) have the same C interface regardless of the request-handling 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 directive that invokes them in 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 next section discusses the parameters in detail.

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. See the section "Result Codes" for details of the result codes.

SAF Parameters

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

- `pb (parameter block)`-- contains the parameters from the directive that invokes the SAF in the `obj.conf` file.
- `sn (session)`-- contains information relating to a single TCP/IP session.
- `rq (request)`-- contains information relating to the current request.

pb (parameter block)

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

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

```
AuthTrans fn=basic-ncsa auth-type=basic
dbm=/netscape/server4/userdb/rs
```

In this case, the pb parameter passed to basic-ncsa contains name/value pairs that correspond to auth-type=basic and dbm=/netscape/server4/userdb/rs.

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

sn (session)

The sn parameter is a pointer to a Session data structure. This parameter contains variables related to an entire session (that is, the time between the opening and closing of the TCP/IP connection between the client and the server). The same sn pointer is passed to each SAF called within each request for an entire session. The following list describes the most important fields in this data structure.

(See Chapter 5, “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 from and writing to the client.

rq (request)

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

The following list describes the most important fields in this data structure (See Chapter 5, “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 G, “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 G, “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 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 is 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 does not try to cache the request, and each subsequent request calls your SAF again. If your SAF sets it to 1, and all other SAFs called for this request also set the flag, the server caches the request and does 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, it 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 disables server caching so that your SAF will always be called.

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

Result Codes

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

- **`REQ_PROCEED`**

  indicates that the SAF achieved its objective. For some request-response steps (AuthTrans, NameTrans, Service, and Error), this 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 executes the SAF specified. If not, the server sends a default HTTP response with the status code and reason phrase plus a short HTML page reflecting the status code and reason phrase for the user. The server then goes to the first `AddLog` directive.

- **`REQ_EXIT`**

  indicates 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.

Creating and Using Custom SAFs

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

   using the NSAPI functions. Each SAF is written for a specific directive.

2. Compile and Link

   the source code to create a shared library (`.so`, `.sl`, or `.dll`) file.

3. Load and Initialize the SAF

   by editing the `obj.conf` file to:

   -- Load the shared library file containing your custom SAF(s).

   -- Initialize the SAF if necessary.

4. Instruct the Server to Call the SAFs

   by editing `obj.conf` to call your custom SAF(s) at the appropriate time.

5. Stop and Start the Server.

6. Test the SAF

   by accessing your server from a browser with a URL that triggers your function.

The following sections describe these steps in greater detail.

Write the Source Code

Write your custom SAFs using NSAPI functions. For a summary of some of the most commonly used NSAPI functions, see the section "Overview of NSAPI C Functions." Chapter 5, “NSAPI Function Reference,” provides information about all of the routines available.

For examples of custom SAFs, see `nsapi/examples/` in the server root directory and also see Chapter 6, “Examples of Custom SAFsS.”

The signature for all SAFs is:

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

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

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.

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

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

SAFs 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 a summary of the most commonly used functions for working with `pblock` structures, see "Parameter Block Manipulation Routines."

When defining a SAF, you do not specifically state which directive it is written for. However, each SAF must be written for a specific directive (such as `Init`, `AuthTrans`, `Service` and so on). Each directive expects its SAFs to do particular things, and your SAF must conform to the expectations of the directive for which it was written. For details of what each directive expects of its SAFs, see the section "Required Behavior of SAFs for Each Directive."

Compile and Link

Compile and link your code with the native compiler for the target platform. For Windows NT, use Microsoft Visual C++ 6.0 or newer when compiling for Enterprise Server 4.0. You must have an import list that specifies all global variables and functions to access from the server binary. Use the correct compiler and linker flags for your platform. Refer to the example Makefile in the `nsapi/examples` directory. On Windows NT link to `nshttpd3x.lib` or `nshttpd40.lib` as appropriate in the `plugins/lib` directory.

The `include` directory in the *`server-root`* directory in Enterprise Server 3.x or in *`server-root`*`/plugins` in Enterprise Server 4.0 contains the NSAPI header file. All the NSAPI header information is now contained in one file called `nsapi.h`.

Load and Initialize the SAF

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

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

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

- `shlib` is the local file system path to the shared library (plugin).

- `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.

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

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

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

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

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

Instruct the Server to Call the SAFs

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

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

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

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

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

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

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

For example, suppose your plugin defines two new SAFs, `do_small_anim()` and `do_big_anim()` which both take `speed` parameters. These functions run animations. All files to be treated as small animations reside in the directory `D:/Netscape/server4/docs/animations/small`, while all files to be treated as full screen animations reside in the directory `D:/Netscape/server4/docs/animations/fullscreen`.

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

```
NameTrans fn=pfx2dir from="/animations/small"
dir="D:/Netscape/server4/docs/animations/small" name="small_anim"
```

```
NameTrans fn=pfx2dir from="/animations/fullscreen"
dir="D:/Netscape/server4/docs/animations/fullscreen"
name="fullscreen_anim"
```

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

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

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

Stop and Start the Server

After modifying `obj.conf`, you need to start and stop 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.

You can also use the Server Manager interface to re-load `obj.conf` and to start and stop the server.

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

Test the SAF

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

You should disable caching in your browser so that the server is sure to be accessed. In Navigator you may hold the shift key while clicking the Reload button to ensure that the cache is not used. (Note that the shift-reload trick does not always force the client to fetch images from source if the images are already in the cache.)

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.

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 when defining new SAFs.

This section provides an overview of the function categories available and some of the more commonly used routines. All the public routines are detailed in Chapter 5, “NSAPI Function Reference.”

The main categories of NSAPI functions are:

- Parameter Block Manipulation Routines
- Protocol Utilities for Service SAFs
- Memory Management
- File I/O
- Network I/O
- Threads
- Utilities

Parameter Block Manipulation Routines

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

- `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 can be a useful function for debugging.

Protocol Utilities for Service SAFs

Protocol utilities provide functionality necessary to implement 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

Memory management routines provide fast, platform-independent 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 permanent memory. To disable pooled memory for debugging, see the built-in SAF `pool-init` in Chapter 3, “Predefined SAFS for Each Stage in the Request Handling Process.”

- `MALLOC()`
- `FREE()`
- `STRDUP()`
- `REALLOC()`
- `CALLOC()`
- `PERM_MALLOC()`
- `PERM_FREE()`
- `PERM_STRDUP()`
- `PERM_REALLOC()`
- `PERM_CALLOC()`

File I/O

The file I/O functions 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_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.

Network I/O

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

- `netbuf_grab()` reads from a network buffer's socket into the network buffer.
- `netbuf_getc()` gets a character from a network buffer.
- `net_write()` writes to the network socket.

Threads

Thread functions include 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()` surrenders ownership of a critical section.
- `crit_terminate()` disposes of a critical section variable.
- `condvar_init()` creates a new condition variable.
- `condvar_notify()` awakens any threads blocked on a condition variable.

- `condvar_wait()` blocks on a condition variable.
- `condvar_terminate()` disposes of a condition variable.

Utilities

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

- `daemon_atrestart()` (Unix only) registers a user function to be called when the server is sent a restart signal (HUP) or at shutdown.
- `util_getline()` gets 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()` converts the special characters in a string into URI escaped format.
- `util_uri_unescape()` converts the URI escaped characters in a string back into special characters.

Required Behavior of SAFs for Each Directive

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

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

- Init SAFs
- AuthTrans SAFs
- NameTrans SAFs
- PathCheck SAFs
- ObjectType SAFs

- Service SAFs
- Error SAFs
- AddLog SAFs

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

Init SAFs

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

AuthTrans SAFs

- Purpose: Verify any authorization information. Only basic authorization is currently defined in the HTTP/1.0 specification.
- Check for `Authorization` header in `rq->headers` which contains the authorization type and uu-encoded user and password information. If header was not sent return `REQ_NOACTION`.
- If header exists, check authenticity of user and 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 `ppath` in `rq->vars` contains the full local file system path, or `REQ_NOACTION` if not.
- To redirect the client to another site, change `ppath` in `rq->vars` to `/URL`. Add `url` to `rq->vars` with full URL (for example., `http://home.netscape.com/`). Return `REQ_PROCEED`.

PathCheck SAFs

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

ObjectType SAFs

- Purpose: Determine content-type of data.
- If `content-type` in `rq->srvhdrs` already exists, return `REQ_NOACTION`.

- Determine the MIME type and create `content-type` in `rq->srvhdrs`
- Return `REQ_PROCEED` if `content-type` is created, `REQ_NOACTION` otherwise

Service SAFs

- Purpose: Generate and send the response to the client.
- A Service SAF is only called if each of the optional parameters `type`, `method`, and `query` specified in the directive in `obj.conf` match the request.
- Remove existing "content-type" from `rq->srvhdrs`. Insert correct "content-type" in `rq->srvhdrs`.
- Create any other headers in `rq->srvhdrs`.
- Call `protocol_status()` to set HTTP response status.
- Call `protocol_start_response()` to send HTTP response and headers.
- Generate and send data to the client 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 parameters `code` and `reason` specified in the directive in `obj.conf` match the current error.
- Error SAFs do the same as Service SAFs, but only in response to an HTTP status error condition.

AddLog SAFs

- Purpose: Log the transaction to a log file.

- `AddLog` SAFs can use any data available in `pb`, `sn`, or `rq` to log this transaction.
- Return `REQ_PROCEED`.

CGI to NSAPI Conversion

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 parameter blocks. The table below indicates how each CGI environment variable can be obtained in NSAPI.

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

Table 4.1

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);`

Table 4.1

CGI getenv()	NSAPI
REMOTE_HOST	session_dns(sn) ? session_dns(sn) : pblock_findval("ip", sn->client);
REMOTE_IDENT	pblock_findval("from", rq->headers); (not usually available)
REMOTE_USER	pblock_findval("auth-user", rq->vars);
REQUEST_METHOD	pblock_findval("method", req->reqpb);
SCRIPT_NAME	pblock_findval("uri", rq->reqpb);
SERVER_NAME	char *util_hostname();
SERVER_PORT	conf_getglobals()->Vport; (as a string)
SERVER_PROTOCOL	pblock_findval("protocol", rq->reqpb);
SERVER_SOFTWARE	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

5

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 3, “Predefined SAFS for Each Stage in the Request Handling Process”.

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 Appendix A, “Data Structure Reference,”and also look in the `nsapi.h` header file in the `include` directory in the build for Enterprise Server 4.0

NSAPI Functions (in Alphabetical Order)

For an alphabetical list of function names, see Appendix H, “Alphabetical List of NSAPI Functions and Macros.”

C D F L M N P R S U

C

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 or NULL if no content was found

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

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

condvar_init()

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

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

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

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

See also `condvar_notify, 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.`

D

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);
}
```

F

filebuf_buf2sd()

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

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

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

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

Parameters `filebuf *buf` is the file buffer 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, or the constant `IO_EOF` or `IO_ERROR` upon an end of file or error.

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

See also `filebuf_close, 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 or `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 or 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 or the constant `REQ_ABORTED` if no function was executed.

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

`Session *sn` 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 or NULL if the function could not be found.

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

Example

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

See also `func_exec`

L

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 or -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`

M

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`

N

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 or 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 is returned if an error has occurred, in which case `errno` is set to the constant `ETIMEDOUT` if the operation did not complete before `timeout` seconds elapsed.

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

`char *buf` is the buffer to receive the bytes.

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

`int timeout` is the number of seconds to allow for the read operation before returning. The purpose of `timeout` is 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 or the constant `IO_ERROR` if unsuccessful

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

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

`int len` is the length of the buffer.

See also `netbuf_close`, `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 or the constant `IO_EOF` or `IO_ERROR`, for end of file or error

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

See also `netbuf_buf2sd`, `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 or the constant `IO_EOF` or `IO_ERROR`, for end of file or error

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

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

See also `netbuf_buf2sd, 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`

P

param_create()

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

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

Returns A pointer to a new `pb_param` structure.

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

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

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

See also `param_free, 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 or 0 if the parameter was NULL.

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

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

See also `param_create, 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 or `NULL` if name was not found.

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

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

See also `pblock_copy`, `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 or NULL if no match was found

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

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

Example see `pblock_nvinsert()`.

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`. This parameter must be an integer. If the value you assign is not a number, then instead use the function `pblock_nvinsert` to create the parameter.

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

See also `pblock_copy`, `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 environment.

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

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

See also `pblock_copy, 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 or NULL if the named `pb_param` was not found.

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

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

See also `pblock_copy, 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 or -1 if an error occurred

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

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

All 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 or the constant `REQ_ABORTED` if the function should treat the request normally, but not send any output to the client

Parameters `Session *sn` is the Session.

`Request *rq` is the Request.

The Session and Request parameters are the same 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`

R

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

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

`Request *rq` is the request identifier for a server application function call.

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

See also `request_create, request_free, request_header`

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 or NULL if it could not perform the mapping

Parameters `char *uri` is the name of the URI.

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

See also `request_create, request_free, request_header`

S

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 or 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 or 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 or 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 or `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 or 0 if the open failed.

Parameters `char *path` is the file name.

Example

```
SYS_FILE fd;
fd = system_fopenRO(pathname);
if (fd == SYS_ERROR_FD)
   break;
```

See also `system_errmsg, 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 or 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 or 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 or 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 occurred.";
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 Greenwich Mean Time.

Syntax

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

Returns A pointer to a calendar time (`tm`) structure containing the GMT time. Depending on your system, the pointer may point to the data item represented by the second parameter, or it may point to a statically-allocated item. For portability, do not assume either situation.

Parameters `time_t *tp` is an arithmetic time.

`tm *res` is a pointer to a calendar time (`tm`) structure.

Example
```
time_t tp;
struct tm res, *resp;
tp = time(NULL);
resp = system_gmtime(&tp, &res);
```

See also `system_localtime, 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()

The10`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 the offset, in bytes, of the new position from the beginning of the file if the operation succeeded or -1 if the operation failed.

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

`int offset` is a number of bytes relative to `whence`. It may be negative.

`int whence` is 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 or -1 if the operation failed.

Parameters `char *old` is the old name of the file.

`char *new` is the new name for the file:

system_ulock()

The `system_ulock` function unlocks the specified file that has been locked by the function `system_lock`. For more information about locking, see `system_flock`.

Syntax `int system_ulock(SYS_FILE fd);`

Returns The constant `IO_OK` if the operation succeeded or 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` if the call succeeds. Returns NULL if the call did not succeed, for example if the `systhread_setkey` function was never called with the specified key during this session

Parameters `int key` is the value associated with the stored data by a `systhread_setdata` function. Keys are assigned by the `systhread_newkey` function.

See also `systhread_current, 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 or 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_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`

U

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 or 0 if the file is not executable.

Parameters `stat *finfo` is the stat structure associated with a file.

`uid_t uid` is the Unix user id.

`gid_t gid` is the Unix group id. Together with `uid`, this 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 or -1 if the directory could not be changed.

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

The parameter must be a writable string because it isn't permanently modified.

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 or -1 if the directory could not be changed.

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

The parameter must be a writable string because it isn't permanently modified.

util_cookie_find()

New in Enterprise Server 4.0.

The `util_cookie_find` function finds a specific cookie in a cookie string and returns its value.

Syntax `char *util_cookie_find(char *cookie, char *name);`

Returns If successful, returns a pointer to the `NULL`-terminated value of the cookie. Otherwise, returns `NULL`. This function modifies the cookie string parameter by null-terminating the name and value.

Parameters `char *cookie` is the value of the Cookie: request header.

`char *name` is the name of the cookie whose value is to be retrieved.

See also `util_cookie_next()`

util_cookie_next()

New in Enterprise Server 4.0.

The `util_cookie_next` function can enumerate all the cookie name-value pairs in a cookie string.

Syntax `char *util_cookie_next(char *cookie, char **name, char **value);`

Returns If successful, returns a pointer beyond the first name-value pair (so that it can be used with `util_cookie_next()` to find the next name-value pair), and `*name` and `*value` point to the `NULL`-terminated name and value of the first cookie. If no cookie name-value pair is found, returns `NULL`. This function modifies the cookie string parameter by null-terminating the name and value.

Parameters `char *cookie` is the value of the Cookie request header

`char **name` points to a pointer to the name on successful execution.

`char **value` points to a pointer to the value on successful execution.

See also `util_cookie_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 or NULL if the string was not found.

Parameters `char **env` is the environment.

`char *name` is the name of an environment variable in env.

See also `util_env_replace, 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 non alphanumeric 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 *line);
```

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

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

-1 if an error occurred. `line` contains a description of the error.

Parameters `filebuf *buf` is the file buffer to be scanned.

`int lineno` is used to include the line number in the error message when an error occurs. The caller is responsible for making sure the line number is accurate.

`int maxlen` is the maximum number of characters that can be written into `l`.

`char *l` is the buffer in which to store the string. The user is responsible for allocating and deallocating `line`.

See also `util_can_exec, 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, returns a string containing that name otherwise returns NULL if the fully-qualified domain name was not found.

Parameters none.

util_is_mozilla()

The `util_is_mozilla` function checks whether a specified user-agent header string is a Netscape browser of at least a specified revision level, 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 or 0 if the user-agent is not a Netscape browser

Parameters

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

`char *major` is the major release number (to the left of the decimal point).

`char *minor` is the minor release number (to the right of the decimal point).

See also `util_is_url, 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 or 0 if the string specified by `url` is not a URL.

Parameters `char *url` is the string to be examined.

See also `util_is_mozilla, 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` or 0 if the date represented by `ims` is earlier than that represented by the `lms`.

Parameters `tm *lms` is the time structure containing a date.

`char *ims` is the string containing a date.

See also `util_strftime`

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 E, "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. Insecure path characters include `//`, `/./`, `/../` and`/.`, `/..` (also for NT`./`) at the end of the URI. 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 or 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`

Chapter

6

Examples of Custom SAFsS

This chapter discusses 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 4, “Creating Custom SAFs.”.

Before writing custom SAFs, you should be familiar with the request-response process, (discussed in Chapter 1, “Basics of Enterprise Server Operation,”) and the role of the configuration file `obj.conf` (discussed in Chapter 2, “Syntax and Use of Obj.conf.”)

Before writing your own SAF, check if an existing SAF serves your purpose. The pre-defined SAFs are discussed in Chapter 3, “Predefined SAFS for Each Stage in the Request Handling Process.”

For a list of the NSAPI functions for creating new SAFs, see Chapter 5, “NSAPI Function Reference.”

This chapter has the following sections:

- Examples in the Build
- AuthTrans Example
- NameTrans Example
- PathCheck Example
- ObjectType Example
- Service Example

- AddLog Example

Examples in the Build

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

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

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

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

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

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

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

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

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

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

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

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

After modifying `obj.conf` manually, you'll need to load the configuration files in the Server Manager interface if it is open. If it is not open, you'll need to stop and start the server to have your changes take effect, since the server loads `obj.conf` during initialization.

After adding new `Init` directives to `obj.conf`, you always need to restart the Enterprise Server to load the changes, since `Init` directives are only applied during server initialization.

AuthTrans Example

This simple example of an `AuthTrans` function demonstrate how to use your own custom ways of verifying that the username and password that a remote client provided is accurate. This program uses a hard coded table of usernames and passwords and checks a given user's password against the one in the static data array. The *userdb* parameter is not used in this function.

`AuthTrans` directives work in conjunction with `PathCheck` directives. Generally, an `AuthTrans` function checks if the username and password associated with the request are acceptable, but it does not allow or deny access to the request -- it leaves that to a `PathCheck` function.

`AuthTrans` functions get the username and password from the headers associated with the request. When a client initially makes a request, the username and password are unknown so the `AuthTrans` function and `PathCheck` function work together to reject the request, since they can't validate the username and password. When the client receives the rejection, the usual response is for it to pop up a dialog box asking the user for their username and password, and then the client submits the request again, this time including the username and password in the headers.

In this example, the `hardcoded-auth` function, which is invoked during the `AuthTrans` step, checks if the username and password correspond to an entry in the hardcoded table of users and passwords.

Installing the Example

To install the function on the Enterprise Server, add the following `Init` directive at the beginning of `obj.conf` to load the compiled function:

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

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

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

```
userdb=unused
```

Note that this function does not actually enforce authorization requirements, it only takes given information and tells the server if it's correct or not. The `PathCheck` function `require-auth` performs the enforcement, so add the following `PathCheck` directive also:

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

Source Code

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

```
#include nsapi.h

typedef struct {
   char *name;
   char *pw;
} user_s;

/* This is the array of users and passwords */

static user_s user_set[] = {
   {"nikki", "bones"},
   {"boots", "frisbee"},
   {"jack", "steak"},
   {"topper", "kibble"},
   {"beulah", "rollover"},
   {NULL, NULL}
};

#include "frame/log.h"

#ifdef __cplusplus
extern "C"
#endif

/* hardcoded_auth is our custom SAF */

NSAPI_PUBLIC int hardcoded_auth(pblock *param, Session *sn, Request *rq)
{
   /* Parameters given to us by auth-basic.
   * Use pblock_findval to find the value of a specific parameter.
   */


   /* pwfile will be null, but that's OK because we don't use it */
   char *pwfile = pblock_findval("userdb", param);
```

```
   /* Get the user and password */
   char *user = pblock_findval("user", param);
   char *pw = pblock_findval("pw", param);

   /* Temp variables */
   register int x;


   /* Iterate over the hardcoded array of users and passwords
   * to see if the current user is in there.
   */
   for(x = 0; user_set[x].name != NULL; ++x) {
      /* If this isn't the user we want, keep going */
      if(strcmp(user, user_set[x].name) != 0)
      continue;

         /* If this is the user we want, verify password.
         * If password is wrong, log an error and return REQ_NOACTION.
         */
         if(strcmp(pw, user_set[x].pw)) {
            log_error(LOG_SECURITY, "hardcoded-auth", sn, rq,
                  "user %s entered wrong password", user);
            return REQ_NOACTION;
            }

         /* If username and password are vaild, return REQ_PROCEED */
         return REQ_PROCEED;
   }

   /* If the username was not found in our array, log an error
   * and return REQ_NOACTION.
   */

   log_error(LOG_SECURITY, "hardcoded-auth", sn, rq,
      "unknown user %s", user);

    return REQ_NOACTION;

}
```

NameTrans Example

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

- `explicit_pathinfo`

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

- `https_redirect`

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

This section discusses the first example. Look at the source code in `ntrans.c` for the second example.

Note: The main thing that a `NameTrans` function usually does is to convert the logical URL in `ppath` in `rq->vars` to a physical pathname. However, the example discussed here, `explicit_pathinfo`, does not translate the URL into a physical pathname, it changes the value of the requested URL. See the second example, `https_redirect`, in `ntrans.c` for an example of a `NameTrans` function that converts the value of `ppath` in `rq->vars` from a URL to a physical pathname.

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

For example:

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

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

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

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

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

Normally `NameTrans` directives return `REQ_PROCEED` when they change the path so that the server does not process any more `NameTrans` directives. However, in this case we want name translation to continue after we have extracted the path info, since we have not yet translated the URL to a physical pathname.

Installing the Example

To install the function on the Enterprise Server, add the following `Init` directive at the beginning of `obj.conf` to load the compiled function:

```
Init fn=load-modules shlib=yourlibrary funcs=explicit-pathinfo
```

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

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

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

Source Code

This example is in the `ntrans.c` file in the `nsapi/examples/` or `plugins/nsapi/examples` subdirectory of the server root directory.

```
#include nsapi.h

#include <string.h>      /* strchr */
#include "frame/log.h"   /* log_error */

#ifdef __cplusplus
extern "C"
#endif

/* explicit-pathinfo is our new NameTrans SAF */

NSAPI_PUBLIC int explicit_pathinfo(pblock *pb, Session *sn, Request *rq)
{
   /* The separator parameter is specified in the directive line
   * in obj.conf that invokes this function.
   * The separator separates the URL of the requested resource
   * from the extra path information to put into PATH_INFO
   */

   char *sep = pblock_findval("separator", pb);

   /* Get the ppath from the vars field of the request object*/
   char *ppath = pblock_findval("ppath", rq->vars);

   /* Temp var */
   char *t;

   /* Verify correct usage */
   if(!sep) {
      log_error(LOG_MISCONFIG, "explicit-pathinfo", sn, rq,
```

```
            "missing parameter (need root)");
        /* When we abort, the default status code is 500 Server Error *
        return REQ_ABORTED;
    }

    /* Check for separator. If not there, don't do anything */
    t = strchr(ppath, sep[0]);
    if(!t)
        return REQ_NOACTION;

    /* If path contains separator, truncate path at the separator */
    *t++ = '\0';

    /* Put the extra path info into the path-info field of rq->vars*/
    pblock_nvinsert("path-info", t, rq->vars);

    /* Normally NameTrans functions return REQ_PROCEED when they change
    * the path. However, we want name translation to continue after we
    * have extracted the extra path info since we haven't translated the
    * URL to a physical file name yet.
    */
    return REQ_NOACTION;

}
```

PathCheck Example

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

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

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

Installing the Example

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

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

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

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

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

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

Source Code

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

```
#include nsapi.h

/* Set hosts to NULL to prevent problems if acf-init is not called */
static char **hosts = NULL;

#include <stdio.h>
#include "base/daemon.h"
#include "base/util.h"          /* util_sprintf */
#include "frame/log.h"          /* log_error */
#include "frame/protocol.h"     /* protocol_status */


/* The longest line we'll allow in an access control file */
#define MAX_ACF_LINE 256

#ifdef __cplusplus
extern "C"
#endif

/* Used to free static array on restart */
NSAPI_PUBLIC void acf_free(void *unused)
{
   register int x;
   for(x = 0; hosts[x]; ++x)
      FREE(hosts[x]);
      FREE(hosts);
      hosts = NULL;
}

/* This is the initialization function that gets invoked
* during the Init stage in obj.conf.
* This function opens the custom file and reads the IP addresses
* of the allowed hosts into the global variable hosts.
*/
```

```
NSAPI_PUBLIC int acf_init(pblock *pb, Session *sn, Request *rq)
{
   /* The file parameter is specified in the PathCheck directive
   * that invokes this function.
   */
   char *acf_file = pblock_findval("file", pb);

   /* Working variables */
   int num_hosts;
   FILE *f;
   char err[MAGNUS_ERROR_LEN];
   char buf[MAX_ACF_LINE];

   /* Check usage. Note: Init functions have special error logging */
   if(!acf_file) {
      util_sprintf(err, "missing parameter to acf_init (need file)");
      pblock_nvinsert("error", err, pb);
      return REQ_ABORTED;
   }

   /* Open the file containing the list of allowed hosts */
   f = fopen(acf_file, "r");

   /* Did we open it? */
   if(!f) {
      util_sprintf(err, "can't open access control file %s (%s)",
      acf_file, system_errmsg());
      pblock_nvinsert("error", err, pb);
      return REQ_ABORTED;
   }

   /* Initialize hosts array */
   num_hosts = 0;
   hosts = (char **) MALLOC(1 * sizeof(char *));
   hosts[0] = NULL;

   while(fgets(buf, MAX_ACF_LINE, f)) {
      /* Blast linefeed that stdio helpfully leaves on there *
         buf[strlen(buf) - 1] = '\0';
         hosts = (char **) REALLOC(hosts, (num_hosts + 2) * sizeof(char *));
         hosts[num_hosts++] = STRDUP(buf);
         hosts[num_hosts] = NULL;
   }

   /* Close the file */
   fclose(f);

   /* At restart, free hosts array */
   daemon_atrestart(acf_free, NULL);

   return REQ_PROCEED;

}

/* restrict_by_acf is the new PathCheck SAF.
```

```
* It checks if the requesting host is in the list of allowed hosts.
* The list of hosts is in the hosts[] array which was set up by
* acf-init during server initialization.
*/
NSAPI_PUBLIC int restrict_by_acf(pblock *pb, Session *sn, Request *rq)
{
   /* No need to get any parameters from the directive in obj.conf. */

   /* Working variables */
   /* Get the client's ip address */
   char *remip = pblock_findval("ip", sn->client);
   register int x;

   /* If the hosts variable is not set, it means acf-init was not called
   * so log an error and return REQ_ABORTED
   if(!hosts) {
      log_error(LOG_MISCONFIG, "restrict-by-acf", sn, rq,
         "restrict-by-acf called without call to acf-init");
         /* The default abort status code is 500 Server Error */
      return REQ_ABORTED;
   }

   /* if hosts is defined, iterate through the hosts list to see
   * if the host that sent the request is allowed access.
   * If the host is on the list, all is well, so return REQ_NOACTION.
   */
   for(x = 0; hosts[x] != NULL; ++x) {
      if(!strcmp(remip, hosts[x]))
         return REQ_NOACTION;
      }

   /* If the requesting host is not on the list, access is denied */
   /* Set response code to forbidden and return an error. *
   protocol_status(sn, rq, PROTOCOL_FORBIDDEN, NULL);
   return REQ_ABORTED;
}
```

ObjectType Example

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

A well-behaved `ObjectType` function checks if the content type is already set, and if so, does nothing except return `REQ_NOACTION`.

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

The main thing an `ObjectType` directive needs to do is to set the content type (if it is not already set). This example sets it to `magnus-internal/parsed-html` in the following lines:

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

The `html2shtml` function looks at the requested file name. If it ends with `.html`, the function looks for a file with the same base name, but with the extension `.shtml` instead. If it finds one, it uses that path and informs 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.

Installing the Example

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

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

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

```
ObjectType fn=html2shtml
```

Source Code

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

```
# include nsapi.h

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

#ifdef __cplusplus
extern "C"
#endif

/* This is the custom SAF. Whenever a request is made for an
* html file, this SAF checks if another file with the same base name
* but with a .shtml extension exists. If it does, it sets the
* type to magnus-internal/parsed-html.
*/
```

```
NSAPI_PUBLIC int html2shtml(pblock *pb, Session *sn, Request *rq)
{
   /* No need to get any parameters from the directive in obj.conf. */

   /* Work variables */
   /* Get the path from the request object */

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

   /* This is a nicely behaved ObjectType function, so obey the rules
   * and if the type has already been set, don't do anything.
   */
   if(pblock_findval("content-type", rq->srvhdrs))
      return REQ_NOACTION;

   /* If path does not end in .html, don't do anything */
   baselen = strlen(path->value) - 5;
   if(strcasecmp(&path->value[baselen], ".html") != 0)
      return REQ_NOACTION;

   /* If we got this far, the file ends in .html */

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

   /* If the .shtml version of the file does not exist,
   * don't do anything */

   if(stat(npath, &finfo) == -1) {
      FREE(npath);
      return REQ_NOACTION;
   }

   /* If the .shtml version of the file does exist, change the pathname
   * of the requested file to the .shtml version
   */
   FREE(path->value);
   path->value = npath;

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

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

   /* We have successfully set the type, so return REQ_PROCEED */

   return REQ_PROCEED;
```

```
}
```

Service Example

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

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

Installing the Example

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

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

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

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

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

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

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

Source Code

```
#include <nsapi.h>

static char *simple_msg = "default customized content";
```

```
/* This is the initialization function.
 * It gets the value of the generated-output parameter
 * specified in the Init directive in obj.conf
*/
NSAPI_PUBLIC int init-simple-service(pblock *pb, Session *sn,
Request *rq)
{
   /* Get the message from the parameter in the directive in obj.conf */
   simple_msg = pblock_findval("generated-output", pb);
   return REQ_PROCEED;

}

/* This is the customized Service SAF.
 * It sends the "generated-output" message to the client.
*/
NSAPI_PUBLIC int simple-service(pblock *pb, Session *sn, Request *rq)
{
   int return_value;
   char msg_length[8];

   /* Use the protocol_status function to set the status of the
   * response before calling protocol_start_response.
   */
   protocol_status(sn, rq, PROTOCOL_OK, NULL);

   /* Although we would expect the ObjectType stage to
   * set the content-type, set it here just to be
   * completely sure that it gets set to text/html.
   */
   param_free(pblock_remove("content-type", rq->srvhdrs));
   pblock_nvinsert("content-type", "text/html", rq->srvhdrs);

   /* If you want to use keepalive, need to set content-length header.
   * The util_itoa function converts a specified integer to a string,
   * and returns the length of the string. Use this
   * function to create a textual representation of a number.
   */

   util_itoa(strlen(simple_msg), msg_length);
   pblock_nvinsert("content-length", msg_length, rq->srvhdrs);

   /* Send the headers to the client*/
   return_value = protocol_start_response(sn, rq);
   if (return_value == REQ_NOACTION) {
      /* HTTP HEAD instead of GET */
      return REQ_PROCEED;
   }

   /* Write the output using net_write*/
   return_value = net_write(sn->csd, simple_msg, strlen(simple_msg));
   if (return_value == IO_ERROR) {
      return REQ_EXIT;
```

```
    }
    return REQ_PROCEED;
}
```

More Complex Service Example

The `send-images` function is a custom SAF which replaces the `doit.cgi` demonstration available on the Netscape home pages. When a file is accessed as `/dir1/dir2/something.picgroup`, the `send-images` function checks if the file is being accessed by a `Mozilla/1.1` browser. If not, it sends a short error message. The file `something.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, add the following line at the beginning of the `obj.conf` file:

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

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

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

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

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

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

AddLog Example

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

Installing the Example

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

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

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

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

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

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

Source Code

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

```
#include nsapi.h

#include "base/daemon.h"    /* daemon_atrestart */
#include "base/file.h"      /* system_fopenWA, system_fclose */
#include "base/util.h"         /* sprintf */

/* File descriptor to be shared between the processes */
static SYS_FILE logfd = SYS_ERROR_FD;

#ifdef __cplusplus
extern "C"
#endif

/* brief_terminate closes the log file when the server is restarted */
NSAPI_PUBLIC void brief_terminate(void *parameter)
{
   system_fclose(logfd);
   logfd = SYS_ERROR_FD;
}

/* brief-init opens the log file when the server is initialized */
NSAPI_PUBLIC int brief_init(pblock *pb, Session *sn, Request *rq)
{
   /* Get the file parameter from the directive in obj.conf that
   * invokes this function.
   */
```

```
   char *fn = pblock_findval("file", pb);

   /* If no file name is given, abort the process */
   if(!fn) {
      pblock_nvinsert("error", "brief-init: needs a file name",pb);
      return REQ_ABORTED;
   }

   /* Open the log file */
   logfd = system_fopenWA(fn);

   /* If a sys error occurs, abort the process */
   if(logfd == SYS_ERROR_FD) {
      pblock_nvinsert("error", "brief-init: needs a file name", pb);
      return REQ_ABORTED;
   }

    /* Close log file when server is restarted *
   daemon_atrestart(brief_terminate, NULL);

   return REQ_PROCEED;
}


NSAPI_PUBLIC int brief_log(pblock *pb, Session *sn, Request *rq)
{
    /* No need to get parameters from the directive in obj.conf */

   /* Get the method, uri, and ip from the request object */
   char *method = pblock_findval("method", rq->reqpb);
   char *uri = pblock_findval("uri", rq->reqpb);
   char *ip = pblock_findval("ip", sn->client);

   /* Create the log message string */
   char *logmsg;
   int len;

   /* Put the ip, method, and uri in the log message */
   logmsg = (char *) MALLOC(strlen(ip) + 1 + strlen(method) + 1 +
      strlen(uri) + 1 + 1);

   len = util_sprintf(logmsg, "%s %s %s\n", ip, method, uri);

   /* Write the log message to the log file.
   * The atomic version uses locking to prevent interference
   */
   system_fwrite_atomic(logfd, logmsg, len);

   /* free the log message string */
   FREE(logmsg);

   /* Log entry has been successfully written so proceed */
   return REQ_PROCEED;

}
```

Appendix

A

Data Structure Reference

NSAPI uses many data structures which are defined in the `nsapi.h` header file, which is in the directory *`server-root`*`/include` in Enterprise 3.x and in *`server-root`*`/plugins/include` in Enterprise Server 4.0.

The NSAPI functions described in Chapter 5, “NSAPI Function Reference,” provide access to most of the data structures and data fields. Before directly accessing a data structure in `naspi.h`, check if an accessor function exists for it.

For information about the privatizaton of some data structures in Enterprise Server 4.0, see:

- Privatization of Some Data Structures

The rest of this chapter describes ome of the frequently used public data structures in `nsapi.h` for your convenience. Note that only the most commonly used fields are documented here for each data structure; for complete details look in `nsapi.h`.

- `session`
- `pblock`
- `pb_entry`
- `pb_param`
- `Session->client`
- `request`
- `stat`
- `shmem_s`

- `cinfo`

Privatization of Some Data Structures

In Enterprise Server 4.0, some data structures have been moved from `nsapi.h` to `nsapi_pvt.h`. The data structures in `nsapi_pvt.h` are now considered to be private data structures, and you should not write code that accesses them directly. Instead, use accessor functions. We expect that very few people have written plugins that access these data structures directly, so this change should have very little impact on existing customer-defined plugins. Look in `nsapi_pvt.h` to see which data structures have been removed from the public domain and to see the accessor functions you can use to access them from now on.

Plugins written for Enterprise Server 3.x that access contents of data structures defined in `nsapi_pvt.h` will not be source compatible with ES 4.0, that is, it will be necessary to `#include "nsapi_pvt.h"` in order to build such plugins from source. There is also a small chance that these programs will not be binary compatible with Enteprise Server 4.0, because some of the data structures in `nsapi_pvt.h` have changed size. In particular, the `directive` structure is larger, which means that a plugin that indexes through the directives in a `dtable` will not work without being rebuilt (with `nsapi_pvt.h` included).

We hope that the majority of plugins do not reference the internals of data structures in `nsapi_pvt.h`, and therefore that most existing NSAPI plugins will be both binary and source compatible with Enterprise Server 4.0.

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. This data structure is frequently used in NSAPI; it provides the basic mechanism for packaging up parameters and values. There are many functions for creating and managing parameter blocks, and for extracting, adding, and deleting entries. See the functions whose names start with `pblock_` in Chapter 5, “NSAPI Function Reference.” You should not need to write code that access `pblock` data fields directly.

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

pb_entry

The `pb_entry` is a single element in the parameter block.

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

pb_param

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

```
typedef struct {
```

```
   char *name,*value;
} pb_param;
```

Session->client

The `Session->client` parameter block structure contains two entries:

- The `ip` entry is the IP address of the client machine.
- The `dns` entry is the DNS name of the remote machine. This member must be accessed through the `session_dns` function call:

```
/*
* session_dns returns the DNS host name of the client for this
* session and inserts it into the client pblock. Returns NULL if
* unavailable.
*/

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

request

Under HTTP protocol, 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 a program calls the `stat( )` function for a given file, the system returns a structure that provides information about the file. The specific details of the structure should be obtained from your platform's implementation, but the basic outline of the structure is as follows:

```
struct stat {
   dev_t      st_dev;     /* device of inode */
   inot_t     st_ino;     /* inode number */
   short      st_mode;    /* mode bits */
   short      st_nlink;   /* number of links to file /*
   short      st_uid;     /* owner's user id */
   short      st_gid;     /* owner's group id */
   dev_t      st_rdev;    /* for special files */
   off_t      st_size;    /* file size in characters */
   time_t     st_atime;   /* time last accessed */
   time_t     st_mtime;   /* time last modified */
   time_t     st_ctime;   /* time inode last changed*/
}
```

The elements that are most significant for server plug-in API activities are `st_size`, `st_atime`, `st_mtime`, and `st_ctime`.

shmem_s

```
typedef struct {
   void       *data;   /* the data */
   HANDLE     fdmap;
   int        size;    /* the maximum length of the data */
   char       *name;   /* internal use: filename to unlink if exposed */
   SYS_FILE   fd;      /* internal use: file descriptor for region */
} shmem_s;
```

cinfo

The `cinfo` data structure records the content information for a file.

```
typedef struct {
   char    *type;
           /* Identifies what kind of data is in the file*/
   char    *encoding;
           /* encoding dentifies 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;
```

Appendix

B

Variables in magnus.conf

When the Enterprise Server starts up, it looks in a file called `magnus.conf` in the *`server-id`*`/config` directory to establish a set of global variable settings that affect the server's behavior and configuration.

Each directive in magnus.conf specifies a variable and a value, for example:

```
ServerID https-boots.mcom.com
ServerName boots.mcom.com
Address 123.45.67.89
```

The order of the directives is not important.

This appendix lists the global settings that can be specified in `magnus.conf` in Enterprise Server 3.x and 4.0.

The categories are:

- Server Information
- Object Configuration File
- Language Issues
- DNS Lookup
- Threads, Processes and Connections
- Native Thread Pools
- CGI
- Error Logging and Statistic Collection
- ACL

- Security
- Miscellaneous

For an alphabetical list of directives, see Appendix I, “Alphabetical List of Directives in magnus.conf.”

Note In Enterprise Server 4.0, much of the functionality of the file cache is controlled by a new configuration file called `nsfc.conf`. For information about `nsfc.conf`, see the tuning chapter in the *Administrator's Guide for Enterprise Server 4.0*.

Server Information

This sub-section lists the directives in `magnus.conf` that specify information about the server. They are:

- Address
- Concurrency
- MtaHost
- Port
- ServerID
- ServerName
- ServerRoot
- User
- VirtualServerFile

Address

If a server has multiple IP addresses and you want it listen for requests only at a specific IP address, set the value of this directive.

Concurrency

This directive determines the number of CPU processors that the server uses. By default, the server uses all the CPU processors. You only need to set this directive if you want the server to use less than the available processors.

MtaHost

Specifies the name of the SMTP mail server used by the server's agents. This value must be specified before reports can be sent to a mailing address.

Port

The `Port` directive determines which TCP port the server listens to. There should be only one `Port` directive in `magnus.conf`.

Unix: If you choose a port number less than 1024, the server must be started as root.

Note: The port you choose can affect how users configure their navigators. Users must specify the port number when accessing the server if the port number is anything other than 80 (unsecured servers) or 443 (secured servers).

Syntax `Port number`

`number` is a whole number between 0 and 65535.

Default If no port is specified, the server assumes 80.

Examples `Port 80`

`Port 8080`

`Port 8000` (Unix only)

ServerID

Specifies the server ID, such as `https-boots.mcom.com`.

ServerName

The `ServerName` directive tells the server what to put in the host name section of any URLs it sends to the client. This affects URLs the server automatically generates; it doesn't affect the URLs for directories and files stored in the server. This name is what all clients use to access the server; they need to combine this name with the port number if the port number is anything other than 80.

This name should be the alias name if your server uses an alias. You can't have more than one `ServerName` directive in `magnus.conf`.

Syntax `ServerName host`

`host` is a fully qualified domain name such as `myhost.netscape.com`.

Default If `ServerName` isn't in `magnus.conf`, the server attempts to derive a host name through system calls. If they don't return a qualified domain name (for example, it gets `myhost` instead of `myhost.netscape.com`), the server won't start, and you'll get a message telling you to manually set this value.

Examples

```
ServerName server.netscape.com
```

```
ServerName www.server.anycompany.com
```

```
ServerName www.agency.gov
```

ServerRoot

Specifies the server root, such as `d:/netscape/server4/https-boots.mcom.com`. This directive is set during installation and is commented out. Unlike other directives, the server expects this directive to start with #. Do not change this directive. If you do, the Server Manager may not function properly.

Syntax

```
#ServerRoot d:/netscape/server4/https-boots.mcom.com
```

User

Windows NT: The `User` directive specifies the user account the server runs with. By using a specific user account (other than LocalSystem), you can restrict or enable system features for the server. For example, you can use a user account that can mount files from another machine.

Unix: The `User` directive specifies the Unix user account for the server. If the server is started by the superuser or root user, the server binds to the Port you specify and then switches its user ID to the user account specified with the `User` directive. This directive is ignored if the server isn't started as `root`. The user account you specify should have *read* permission to the server's root and subdirectories. The user account should have write access to the `logs` directory and execute permissions to any CGI programs. The user account should not have write access to the configuration files. This ensures that in the unlikely event that someone compromises the server, they won't be able to change configuration files and gain broader access to your machine. Although you can use the `nobody` user, it isn't recommended.

Syntax `User name`

`name` is the 8-character (or less) login name for the user account.

Default If there is no `User` directive, the server runs with the user account it was started with.

Examples `User http`

`User server`

`User nobody`

VirtualServerFile

The value of this directive is the name of a file that specifies virtual servers. Each line in this file contains an `IP`, `docroot` pair.

Object Configuration File

This subsection lists the directives in `magnus.conf` that provide information about the object configuration file that instructs the server how to handle requests. These directives are:

- LoadObjects
- RootObject

LoadObjects

The `LoadObjects` directive specifies one or more object configuration files to use on startup, most notably `obj.conf`, which contains instructions that tell the server how to handle requests from clients.

Note: Although you can have more than one object configuration file, the Server Manager interface works on only one file and assumes that it is the file `obj.conf` in the `config` directory in the server root directory. If you use the Server Manger interface, don't put the `obj.conf` file in any other directory and don't rename it.

Syntax `LoadObjects filename`

`filename` is either the full path name or a relative path name.

Unix: When the server starts executing, relative path names are resolved from the directory specified with the `-d` command lien flag. If no `-d` flag was given, the server looks in the current directory.

Default There is no default. Make sure that your `magnus.conf` loads the `obj.conf` object, otherwise your server will not be able to process requests from clients.

Examples

```
LoadObjects obj.conf
```

Unix:

```
LoadObjects /var/ns-server/admin/config/local-objs.conf
```

RootObject

The `RootObject` directive tells the server which object loaded from an object file is the server default. The default object is expected to have all the name translation directives for the server; any server behavior that is configured in the default object affects the entire server.

If you specify an object that doesn't exist, the server doesn't report an error until a client tries to retrieve a document. The Server Manager assumes the default to be the object named `default`. Don't deviate from this convention if you use (or plan to use) the Server Manager.

Syntax

```
RootObject name
```

`name` is the name of an object defined in one of the object files loaded with a `LoadObjects` directive.

Default There is no default; that is, if you specify RootObject, you must specify a name with it.

Examples

```
RootObject default
```

Language Issues

This section lists the directives in `magnus.conf` related to language issues. The directives are:

- AcceptLanguage
- AdminLanguage

- ClientLanguage
- DefaultLanguage

AcceptLanguage

This directive determines whether or not the server parses the Accept-Language header sent by the client to indicate which languages the client accepts. If the value is `on`, the server parses this header and sends an appropriate language version based on which language the client can accept. You should set this value to `on` only if the server supports multiple languages.

When this directive is set to `on`, the accelerator cache is disabled since it does not use `AcceptLanguage` in its cache keys.

Default The default value is `off`.

AdminLanguage

For an international version of the server, this directive specifies the language for the Server Manager. Values `en` (English), `fr` (French), `de` (German) or `ja` (Japanese).

ClientLanguage

For an international version of the server, this directive specifies the language client messages (such as File Not Found). Values `en` (English), `fr` (French), `de` (German) or **`ja`** (Japanese).

DefaultLanguage

For an international version of the server, this directive specifies the default language for the server. The default language is used for both the client responses and administration. Values `en` (English), `fr` (French), `de` (German) or **`ja`** (Japanese).

DNS Lookup

This section lists the directives in `magnus.conf` that affect DNS lookup. The directives are:

- AsyncDNS
- DNS

AsyncDNS

Specifies whether asynchronous DNS is allowed. The value is either `on` or `off`. If DNS is enabled, enabling asynchronous DNS improves server performance.

DNS

The `DNS` directive specifies whether the server performs DNS lookups on clients that access the server. When a client connects to your server, the server knows the client's IP address but not its host name (for example, it knows the client as 198.95.251.30, rather than its host name `www.a.com`). The server will resolve the client's IP address into a host name for operations like access control, CGI, error reporting, and access logging.

If your server responds to many requests per day, you might want (or need) to stop host name resolution; doing so can reduce the load on the DNS or NIS server.

Syntax `DNS [on|off]`

Default DNS host name resolution is on as a default.

Example `DNS on`

Threads, Processes and Connections

This subsection lists the directives in `magnus.conf` that affect the number and timeout of threads, processes, and connections. They are:

- BlockingListenSockets
- KeepAliveTimeout
- KernelThreads

- ListenQ
- MaxKeepAliveConnections
- MaxProcs
- PostThreadsEarly
- RcvBufSize
- RqThrottle
- RqThrottleMinPerSocket
- SndBufSize
- StackSize
- TerminateTimeout

Also see the section "Native Thread Pools" for new directives in Enterprise Server 4.0 for controlling the pool of native kernel threads.

BlockingListenSockets

This directive determines whether or not the server's sockets listen in blocking mode. Do not use this directive with SSL.

KeepAliveTimeout

This directive determines the maximum time that the server holds open an HTTP Keep-Alive connection or a persistent connection between the client and the server. The Keep-Alive feature for earlier versions of the server allows the client/server connection to stay open while the server processes the client request. For Enterprise Server 3.0+, the default connection is a persistent connection that remains open until the server closes it or the connection has been open for longer than the time allowed by `KeepAliveTimeout`.

KernelThreads

Enterprise Server can support both kernel-level and user-level threads whenever the operating system supports kernel-level threads. Usually, the standard debugger and compiler are intended for use with kernel-level threads. By setting `KernelThreads` to on, you ensure that the server uses only kernel-level threads, not user-level threads.

ListenQ

Defines the number of incoming connections for a server socket.

MaxKeepAliveConnections

Specifies the maximum number of Keep-Alive and persistent connections that the server can have open simultaneously.

Default 200

MaxProcs

New in Enterprise Server 4.0.

Specifies the maximum number of processes that the server can have running simultaneously. If you don't include `MaxProcs` in your magnus.conf file, the server defaults to running a single process.

There is additional discussion of this and other server configuration and performance tuning issues in the "Configuring the Server for Performance" chapter in the *Enterprise Server 4.0 Administrator's Guide*, which can be found at

`http://home.netscape.com/eng/server/webserver/4.0/ag/esperfrm.htm`

The "*Enterprise Server 4.0 Administrator's Guide* " is also shipped in the Enterprise Server 4.0 build in the `manuals/ag` directory.

PostThreadsEarly

If this directive is set to `on`, the server checks the whether the minimum number of threads are available at a socket (as specified by `RqThrottleMinPerSocket`) after accepting a connection but before sending the response to the request. Use this directive when the server will be handling requests that take a long time to handle, such as those that do long database connections.

RcvBufSize

Controls the size of the receive buffer at the server's sockets.

RqThrottle

Specifies the maximum number of simultaneous requests that the server can handle simultaneously per socket. Each request runs in its own thread.

There is additional discussion of this and other server configuration and performance tuning issues in the “Configuring the Server for Performance” chapter in the *Enterprise Server 4.0 Administrator’s Guide*, which can be found at

`http://home.netscape.com/eng/server/webserver/4.0/ag/esperfrm.htm`

The "*Enterprise Server 4.0 Administrator’s Guide* " is also shipped in the Enterprise Server 4.0 build in the `manuals/ag` directory.

Default 512

RqThrottleMinPerSocket

Specifies the approximate minimum number of threads that wait at each socket for requests to come in.

SndBufSize

Controls the size of the send buffer at the server’s sockets.

StackSize

Determines the maximum stack size for each request handling thread.

TerminateTimeout

Specifies the time that the server waits for all existing connections to terminate before it shuts down.

Native Thread Pools

New in Enterprise Server 4.0.

This section lists the directives for controlling the size of the native kernel thread pool. These directives are all new in Enterprise Server 4.0. In previous versions of the server, you could control the native thread pool by setting the system variables `NSCP_POOL_STACKSIZE`, `NSCP_POOL_THREADMAX`, and `NSCP_POOL_WORKQUEUEMAX`.

Note If you have set these values as environment variables and also in `magnus.conf`, the environment variable values will take precedence.

The directives are:

- NativePoolStackSize
- NativePoolMaxThreads
- NativePoolMinThreads
- NativePoolQueueSize

NativePoolStackSize

New in Enterprise Server 4.0.

Determines the stack size of each thread in the native (kernel) thread pool.

NativePoolMaxThreads

New in Enterprise Server 4.0.

Determines the maximum number of threads in the native (kernel) thread pool.

Default 128

NativePoolMinThreads

New in Enterprise Server 4.0.

Determines the minimum number of threads in the native (kernel) thread pool.

Default 1

NativePoolQueueSize

New in Enterprise Server 4.0.

Determines the number of threads that can wait in the queue for the thread pool. If all threads in the pool are busy, then the next request-handling thread that needs to use a thread in the native pool must wait in the queue. If the queue is full, the next request-handling thread that tries to get in the queue is rejected, with the result that it returns a busy response to the client. It is then free to handle another incoming request instead of being tied up waiting in the queue.

CGI

This section lists the directives in `magnus.conf` that affect requests for CGI programs. The directives are:

- CGIExpirationTimeout
- CGIWaitPid (UNIX Only)

CGIExpirationTimeout

New in Enterprise Server 4.0.

This directive specifies the maximum time in seconds that CGI threads are allowed to run before being killed.

The value of `CGIExpirationTimeout` should not be set too low - 5 minutes would be a good value for most interactive CGIs; but if you have CGIs that are expected to take longer without misbehaving, then you should set it to the maximum duration you expect a CGI program to run normally.

CGIWaitPid (UNIX Only)

This directive is to prevent defunct processes on UNIX systems for each SHTML access. If the value is `on`, the server calls `waitpid` explicitly to pickup terminated shtml or CGI child processes.

Error Logging and Statistic Collection

This section lists the directives in `magnus.conf` that affect error logging and the collection of server statistics. They are:

- DaemonStats (Unix Only)
- ErrorLog
- LogVerbose
- PidLog

DaemonStats (Unix Only)

This directive specifies whether or not the server collects some daemon statistics. The value is `on` or `off`. If the value is `off`, SNMP statistic collection will not work.

ErrorLog

The `ErrorLog` directive specifies the directory where the server logs its errors. If errors are reported to a file, then the file and directory in which the log is kept must be writable by whatever user account the server runs as.

Unix: You can also use the `syslog` facility.

Syntax `ErrorLog logfile`

`logfile` can be either a full path and file name.

On Unix systems, it can be the keyword `SYSLOG` (it must be in all capital letters).

Default There is no default error log.

Examples **Windows NT:**

```
ErrorLog C:\Netscape\ns-home\Logs\Errors
```

Unix:

```
ErrorLog /var/ns-server/logs/errors
```

```
ErrorLog SYSLOG
```

LogVerbose

This directive determines whether verbose logging occurs or not. If the value is `on`, the server logs all server messages including those that are not logged by default (such as WAI initialization messages).

PidLog

`PidLog` specifies a file in which to record the process ID (pid) of the base server process. Some of the server support programs assume that this log is in the server root, in `logs/pid`.

To shut down your server, kill the base server process listed in the pid log file by using a `-TERM` signal. To tell your server to reread its configuration files and reopen its log files, use `kill` with the `-HUP` signal.

If the `PidLog` file isn't writable by the user account that the server uses, the server does not log its process ID anywhere. The server won't start if it can't log the process ID.

Syntax `PidLog file`

`file` is the full path name and file name where the process ID is stored.

Default There is no default.

Examples
```
PidLog /var/ns-server/logs/pid
```
```
PidLog /tmp/ns-server.pid
```

ACL

This section lists the directives in magnus.conf relevant to access control lists (ACLs).

- ACLFile

ACLFile

The `ACLFile` directive specifies an ACL (Access Control List) definition file—a text file that normally resides in the `httpacl` directory. Multiple `ACLFile` directives can appear in the `magnus.conf` file. The server reads all the ACL definitions in all the specified ACL definition files when it starts up. Each ACL file must have a unique name.

Usually the value of `ACLFile` is `generated.https-`*`servername`*`.acl`, and it resides in the `httpacl` directory of the server installation directory.

Syntax `ACLFile name`

`name` is the name of an ACL definition file.

Example `ACLFile d:/netscape/server4/httpacl/generated.https-boots.mcom.com.acl`

Security

This section lists the directives in `magnus.conf` that affect server access and security issues for Enterprise Server. They are:

- Chroot (Unix only)
- Ciphers
- Security
- ServerCert
- ServerKey
- SSLCacheEntries
- SSLClientAuth
- SSLSessionTimeout
- SSL2
- SSL3
- SSL3Ciphers
- SSL3SessionTimeout

Chroot (Unix only)

The `Chroot` directive lets the Unix system administrator place the server under a constraint such that it has access only to files in a given directory, termed the "Chroot directory". This is useful if the server's security is ever compromised. For example, if an intruder somehow obtains shell access on the server machine, the intruder could only affect a very limited set of files on the server machine.

The server must be started as the `superuser` to use the `Chroot` directive. CGI programs must be linked statically, and any binaries (`perl or /bin/sh`) must be copied to the Chroot directory.

The user public information directory feature isn't available unless a copy of `/etc/passwd` is kept in the Chroot directory and all of the users home directories are exactly mirrored within the Chroot directory.

A server using `Chroot` can't be restarted with the `-HUP` signal.

Logs and server configuration files should be kept outside the Chroot directory.

IMPORTANT All paths in `magnus.conf` must be absolute; paths in `obj.conf` must be relative to the Chroot directory.

Syntax `Chroot directory`

`directory` is the full path name to the directory used as the server's root directory.

Default There is no default. You must specify a directory.

Examples
```
Chroot /d/ns-httpd
```
```
Chroot /www
```

Ciphers

The `Ciphers` directive specifies the ciphers enabled for your server.

Syntax
```
Ciphers +rc4 +rc4export -rc2 -rc2export +idea +des +desede3
```

A + means the cipher is active, and a - means the cipher is inactive.

Valid ciphers are `rc4`, `rc4export`, `rc2`, `rc2export`, `idea`, `des`, `desede3`. Any cipher with `export` as port of its name is not stronger than 40 bits.

Security

The `Security` directive tells the server whether encryption (Secure Sockets Layer version 2 or version 3 or both) is enabled or disabled.

If `Security` is set to `on`, and both SSL2 and SSL3 are enabled, then the server tries SSL3 encryption first. If that fails, the server tries SSL2 encryption.

Syntax
```
Security [on|off]
```

Default By default, security is off.

Example
```
Security off
```

ServerCert

The `ServerCert` directive specifies where the certificate file is located.

Syntax `ServerCert certfile`

`certfile` is the server's certificate file, specified as a relative path from the server root or as an absolute path.

ServerKey

The `ServerKey` directive tells the server where the key file is located.

Syntax `ServerKey keyfile`

`keyfile` is the server's key file, specified as a relative path from the server root or as an absolute path.

SSLCacheEntries

Specifies the number of SSL sessions that can be cached.

SSLClientAuth

The `SSLClientAuth` directive causes SSL3 client authentication on all requests.

Syntax `SSL3ClientAuth on|off`

`on` directs that SSL3 client authentication be performed on every request, independent of ACL-based access control.

SSLSessionTimeout

The `SSLSessionTimeout` directive controls SSL2 session caching.

Syntax `SSLSessionTimeout seconds`

`seconds` is the number of seconds until a cached SSL2 session becomes invalid. The default value is 100. If the `SSLSessionTimeout` directive is specified, the value of seconds is silently constrained to be between 5 and 100 seconds.

SSL2

The `SSL2` directive tells the server whether Secure Sockets Layer, version 2 encryption is enabled or disabled. The `Security` directive dominates the `SSL2` directive; if SSL2 encryption is enabled but the `Security` directive is set to `off`, then it is as though SSL2 were disabled.

Syntax `SSL2 [on|off]`

Default By default, security is off.

Example `SSL2 off`

SSL3

The `SSL3` directive tells the server whether Secure Sockets Layer, version 3 security is enabled or disabled. The `Security` directive dominates the `SSL3` directive; if SSL3 security is enabled but the `Security` directive is set to `off`, then it is as though SSL3 were disabled.

Syntax `SSL3 [on|off]`

Default By default, security is off.

Example `SSL3 off`

SSL3Ciphers

The `SSL3Ciphers` directive specifies the SSL3 ciphers enabled for your server.

Syntax `SSL3Ciphers +rc4 +rc4export -rc2 -rc2export +idea +des +desede3`

A + means the cipher is active, and a - means the cipher is inactive.

Valid ciphers are `rsa_rc4_128_md5`, `rsa3des_sha`, `rsa_des_sha`, `rsa_rc4_40_md5`, `rsa_rc2_40_md5`, and `rsa_null_md5`. Any cipher with `40` as part of its name is 40 bits.

SSL3SessionTimeout

The `SSL3SessionTimeout` directive controls SSL3 session caching.

Syntax `SSL3SessionTimeout seconds`

`seconds` is the number of seconds until a cached SSL3 session becomes invalid. The default value is 86400 (24 hours). If the `SSL3SessionTimeout` directive is specified, the value of seconds is silently constrained to be between 5 and 86400 seconds.

Miscellaneous

This section lists miscellaneous other directives in `magnus.conf`.

- Umask (UNIX only)

Umask (UNIX only)

This directive specifies the umask value used by the NSAPI functions `System_fopenWA()` and `System_fopenRW()` to open files in different modes. Valid values for this directive are standard UNIX umask values.

For more information on these functions, see `system_fopenWA()` and `system_fopenRW()` in Chapter 5, “NSAPI Function Reference.”

Appendix

C

MIME Types

This appendix discusses the MIME types file. The sections are:

- Introduction
- Loading the MIME Types File
- Determining the MIME Type
- How the Type Affects the Response
- What Does the Client Do with the MIME Type?
- Syntax of the MIME Types File
- Sample MIME Types File

Introduction

The MIME types file in the `config` directory contains mappings between MIME (Multipurpose Internet Mail Extensions) types and file extensions. For example, the MIME types file maps the extensions `.html` and `.htm` extension to the type `text/html`:

```
type=text/html exts=htm,html
```

When the Enterprise Server receives a request for a resource from a client, it uses the MIME type mappings to determine what kind of resource is being requested.

MIME types can have three attributes: language (`lang`), encoding (`enc`), and content-type (`type`). The most commonly used attribute is `type`. The server frequently considers the `type` when deciding how to generate the response to the client. (The `enc` and `lang` attributes are rarely used).

By default, the MIME types file is called `mime.types`. You should not change the name of this file unless you have a particular reason for doing so -- everyone expects it to be called `mime.types`.

Loading the MIME Types File

When the server is initialized, an `Init` directive in `obj.conf` invokes the `load-mime-types` directive to load the MIME types file:

```
Init fn="load-types" mime-types="mime.types"
```

After loading the MIME types file, the server uses it to create a table of mappings between file extensions and MIME types.

If you make changes to the MIME types file, you will need to restart the server before the changes take effect. The server loads the MIME types file during the initialization step, so it does not notice any changes in the MIME types file until the next time it is initialized.

Determining the MIME Type

During the `ObjectType` step in the request handling process, the server determines the MIME type attributes of the resource requested by the client. Several different server application functions (SAFs) can be used to determine the MIME type, but the most commonly used one is `type-by-extension`. This function tells the server to look up the MIME type according to the requested resource's file extension in the MIME types table.

The directive in `obj.conf` that tells the server to look up the MIME type according to the extension is:

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

If the server uses a different SAF, such as `force-type`, to determine the `type`, then the MIME types table is not used for that particular request.

For more details of the ObjectType step, see Chapter 2, “Syntax and Use of Obj.conf.”

How the Type Affects the Response

The server considers the value of the `type` attribute when deciding which `Service` directive in `obj.conf` to use to generate the response to the client.

By default, if the `type` does not start with `magnus-internal/`, the server just sends the requested file to the client. The directive in `obj.conf` that contains this instruction is:

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

Note here the use of the special characters `*~` to mean "does not match." See Appendix D, “Wildcard Patterns,” for details of special characters.

By convention, all values of `type` that require the server to do something other than just send the requested resource to the client start with `magnus-internal/`.

For example, if the requested resource's file extension is `.map`, the type is mapped to `magnus-internal/imagemap`. If the extension is `.cgi`, `.exe`, or `.bat`, the type is set to `magnus-internal/cgi`:

```
type=magnus-internal/imagemap       exts=map
type=magnus-internal/cgi            exts=cgi,exe,bat
```

If the `type` starts with `magnus-internal/`, the server executes whichever `Service` directive in `obj.conf` matches the specified type. For example, if the type is `magnus-internal/imagemap`, the server uses the `imagemap` function to generate the response to the client, as indicated by the following directive:

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

If the type is `magnus-internal/servlet`, the server uses the `NSServletService` function to generate the response to the client, as indicated by the following directive:

```
Service type="magnus-internal/servlet" fn="NSServletService"
```

What Does the Client Do with the MIME Type?

The `Service` function generates the data and sends it to the client that made the request. When the server sends the data to the client, it also sends headers. These headers include whichever MIME type attributes are known (which is usually `type`).

When the client receives the data, it uses the MIME type to decide what to do with the data. For browser clients, the usual thing is to display the data in the browser window.

If the requested resource cannot be displayed in a browser but needs to be handled by another application, its `type` starts with `application/`, for example `application/octet-stream` (for `.bin` file extensions) or `application/x-maker` (for `.fm` file extensions). The client has its own set of user-editable mappings that tells it which application to use to handle which types of data.

For example, if the type is `application/x-maker`, the client usually handles it by opening Adobe FrameMaker to display the file.

Syntax of the MIME Types File

The first line in the MIME types file identifies the file format and must read:

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

Other non-comment lines have the following format:

```
type=type/subtype exts=[file extensions] icon=icon
```

- `type/subtype` is the type and subtype.
- `exts` are the file extensions associated with this type.
- `icon` is the name of the icon the browser displays. Netscape Navigator keeps these images internally. If you use a browser that doesn't have these icons, the server delivers them.

Sample MIME Types File

Here is an example of a MIME types file:

```
#--Netscape Communications Corporation MIME Information
# Do not delete the above line. It is used to identify the file type.
type=application/octet-stream  exts=bin,exe
type=application/oda           exts=oda
type=application/pdf           exts=pdf
type=application/postscript    exts=ai,eps,ps
type=application/rtf           exts=rtf
type=application/x-mif         exts=mif,fm
type=application/x-gtar        exts=gtar
type=application/x-shar        exts=shar
type=application/x-tar         exts=tar
type=application/mac-binhex40  exts=hqx

type=audio/basic               exts=au,snd
type=audio/x-aiff              exts=aif,aiff,aifc
type=audio/x-wav               exts=wav

type=image/gif                 exts=gif
type=image/ief                 exts=ief
type=image/jpeg                exts=jpeg,jpg,jpe
type=image/tiff                exts=tiff,tif
type=image/x-rgb               exts=rgb
type=image/x-xbitmap           exts=xbm
type=image/x-xpixmap           exts=xpm
type=image/x-xwindowdump       exts=xwd

type=text/html                 exts=htm,html
type=text/plain                exts=txt
type=text/richtext             exts=rtx
type=text/tab-separated-values exts=tsv
type=text/x-setext             exts=etx

type=video/mpeg                exts=mpeg,mpg,mpe
type=video/quicktime           exts=qt,mov
type=video/x-msvideo           exts=avi

enc=x-gzip exts=gz
enc=x-compress exts=z
enc=x-uuencode exts=uu,uue

type=magnus-internal/imagemap  exts=map
type=magnus-internal/parsed-html exts=shtml
type=magnus-internal/cgi       exts=cgi,exe,bat
type=magnus-internal/jsp        exts=jsp
```

Appendix

D

Wildcard Patterns

This appendix describes the format of wildcard patterns used by the Netscape Enterprise Server.

These wildcards are used in:

- directives in the configuration file `obj.conf` (see Chapter 2, “Syntax and Use of Obj.conf.”)
- various built-in SAFs (see Chapter 3, “Predefined SAFS for Each Stage in the Request Handling Process.”)
- some NSAPI functions (see Chapter 5, “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.

Wildcard Patterns

Table 6.1 Wildcard patterns

Pattern	Use
*	Match zero or more characters.
?	Match exactly one occurrence of any character.

Table 6.1 Wildcard patterns

Pattern	Use
|	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.
[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.

Wildcard Examples

Table 6.2 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.

Table 6.2 Wildcard examples

Pattern	Result
~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`.
type=*~magnus-internal/*	Matches any type that does not start with `magnus-internal/`. This wildcard pattern is used in the file `obj.conf` in the catch-all `Service` directive.

Appendix

E

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 6.3 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"
%X	Time "HH:MM:SS"

Table 6.3 Time formats

Symbol	Meaning
%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

F

Server-Parsed HTML Tags

HTML files can contain tags that are executed on the server. This appendix discusses the standard server-side tags you can include in HTML files.

For information about defining your own server-side tags in Enterprise Server 4.0, see the *Programmer's Guide to Enterprise Server 4.0*.

Note: The server parses server-side tags only if server-side parsing has been activated. Use the "Parse HTML" page in the Content Management tab of the Server Manager interface to enable or disable the parsing of server-side tags.

When you activate parsing, you need to be sure that the following directives are added to your `obj.conf` file (Note that native threads are turned off.):

```
Init funcs="shtml_init,shtml_send" shlib="<install_dir>/bin/https/bin/
Shtml.dll" NativeThreads="no" fn="load-modules"

Init LateInit = "yes" fn="shtml_init"
```

Using Server-Parsed Commands

This section describes the HTML commands for including server-parsed tags in HTML files. These commands are embedded into HTML files which are processed by the built-in SAF `parse-html`.

The server replaces each command with data determined by the command and its attributes.

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.

- `config`
- `include`
- `echo`
- `fsize`
- `flastmod`
- `exec`

config

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 E, "Time Formats," for details 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.

include

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"-->`

echo

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"-->`

fsize

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"-->`

flastmod

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"-->`

exec

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_NAME

  is the file name of the parsed file.
- DOCUMENT_URI

  is the virtual path to the parsed file (for example, `/shtml/test.shtml`).
- QUERY_STRING_UNESCAPED

  is the unescaped version of any search query the client sent with all shell-special characters escaped with the \ character.
- DATE_LOCAL

  is the current date and local time.
- DATE_GMT

  is the current date and time expressed in Greenwich Mean Time.
- LAST_MODIFIED

is the date the file was last modified.

Appendix

G

HyperText Transfer Protocol

The HyperText Transfer Protocol (HTTP) is a protocol (a set of rules that describes how information is exchanged) that allows a client (such as a web browser) and a web server to communicate with each other. 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`

Introduction

HTTP is based on a request/response model. The browser opens a connection to the server and sends a request to the server.

The server processes the request and generates a response which it sends to the browser.The server then closes the connection.

Netscape Enterprise Server 3.x and 4.0 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`

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 resource (such as a document or image)
- `HEAD`—Requests only the header information for the document
- `POST`—Requests that the server accept some data from the browser, such as form input for a CGI program
- `PUT`—Replaces the contents of a server's document with data from the browser

Request Headers

The browser can send headers to the server. Most are optional. Some commonly used request headers are shown in Table 6.4.

Table 6.4 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.
`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 sends data after the blank line following the request headers. If the browser sends a `GET` or `HEAD` request, there is no data to send.

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. The five categories of status codes are:

- 100-199 a provisional response.

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

Table 6.5 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 6.6

.
Table 6.6 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.
`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 such as an image or an HTML page.

Appendix

H

Alphabetical List of NSAPI Functions and Macros

C

D

F

L

M

N

P

R

S

U

Appendix

I

Alphabetical List of Directives in magnus.conf

A

B

C

D

E

K

L

M

N

P

R

S

T

U

V

Appendix

J

Alphabetical List of Pre-defined SAFs

A

B

C

D

F

G

H

I

K

L

M

N

P

Q

R

S

T

U

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W