SAS/IntrNet® 9.1 Application Dispatcher

Third Edition

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2007. *SAS/IntrNet®: Application Dispatcher, Third Edition*. Cary, NC: SAS Institute Inc.

SAS/IntrNet®: Application Dispatcher, Third Edition

Copyright © 2007, SAS Institute Inc., Cary, NC, USA

All rights reserved. Produced in the United States of America.

For a Web download or e-book: Your use of this publication shall be governed by the terms established by the vendor at the time you acquire this publication.

U.S. Government Restricted Rights Notice: Use, duplication, or disclosure of this software and related documentation by the U.S. government is subject to the Agreement with SAS Institute and the restrictions set forth in FAR 52.227-19, Commercial Computer Software-Restricted Rights (June 1987).

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

February 2007

SAS® Publishing provides a complete selection of books and electronic products to help customers use SAS software to its fullest potential. For more information about our e-books, e-learning products, CDs, and hard-copy books, visit the SAS Publishing Web site at **support.sas.com/pubs** or call 1-800-727-3228.

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.

Table of Contents

Table of Contents

Table of Contents

Table of Contents

What's New in SAS/IntrNet 9 and 9.1 Application Dispatcher

Overview

Application Dispatcher provides Application Broker and Load Manager enhancements, additional server encodings for z/OS, a new logging feature, new options for the PROC APPSRV statement, file upload capability, and enhanced documentation.

Note:

- This section describes the features of SAS/IntrNet: Application Dispatcher that are new or enhanced since SAS 8.2.
- z/OS is the successor to the OS/390 operating system. SAS/IntrNet 9.1: Application Dispatcher is supported in both the OS/390 and z/OS operating environments and, throughout this document, any reference to z/OS also applies to OS/390, unless otherwise stated.

Details

SAS/IntrNet: Application Dispatcher includes the following enhancements:

- The Application Dispatcher samples are now part of the samples database.
- You can now use Application Dispatcher to upload one or more files to your Application Server.
- The Application Broker default welcome page can be replaced by a customized welcome page.
- SAS Enterprise Guide 3.0 includes experimental support for building SAS/IntrNet: Application Dispatcher applications. You can use SAS Enterprise Guide to generate or modify SAS programs and to generate input HTML forms for your SAS/IntrNet applications. See the Working with SAS/IntrNet Applications section in the SAS Enterprise Guide product Help for more information.
- The _NOLOG_ feature enables you to create special macro symbols that can be sent to the Application Server without publishing the macro values in the APPSRV log.
- SAS/IntrNet: Application Dispatcher now supports the following additional parameters for starting the Load Manager:

  −maxreq=minutes
  : specifies the maximum time it should take for the Application Server to send a BUSY state after the Application Server is allocated to the Application Broker.

  −maxrun=minutes
  : specifies the expected maximum job run−time in minutes before an Application Server is declared as hung.

  −maxstart=minutes
  : specifies the maximum time that it should take an Application Server to start.

  −nokill
  : specifies not to kill a pool server that is marked as hung.

  −workdir=directory
  : enables you to specify the current working directory as a start parameter for the Load Manager.
- You can obtain an Application Server activity report by using the Application Broker and running the LOADCURRENT program.

- The documentation now includes a sample Web application that demonstrates some of the features of Application Dispatcher sessions. The sample application is an online library. Users can login, select one or more items to check out of the library, and request by e−mail that the selected items be delivered. The sample code shows how to create a session and then create, modify, and view macro variables and data sets in that session.
- Two additional versions of the Application Broker have been developed for heavily loaded systems where performance is critical. The two new modules are broker.dll (ISAPI Windows) and broker.so (GWAPI z/OS).
- For z/OS, the SAS 9 Application Broker requires that the IBM Web maintenance patch, PQ47248, be installed if you intend to use a Web server codepage (FSCP) other than `ibm-1047`.
- SAS/IntrNet: Application Broker on z/OS now supports Windows server encodings in addition to the previously supported ISO−8859 encodings. New encodings include

    - `wlatin1` (Western Europe): This value is the default in all cases except when the Web server is using IBM−870 or IBM−1025 encoding.
    - `wlatin2` (Eastern Europe): This value is the default when the Web server is using IBM−870 encoding.
    - `wcyrillic` (Cyrillic): This value is the default when the Web server is using IBM−1025 encoding.

  The z/OS Application Broker also supports a new ISO−8859 encoding:

    - `ISO-8859-15` (Latin9): This encoding is recommended only if your Web server and SAS System are using any of the IBM−114x encodings.

  SAS/IntrNet: Application Broker on z/OS now supports the following additional Web server fscp encodings:

    - `EBCDIC1140` (North America)
    - `EBCDIC1141` (Austria/Germany)
    - `EBCDIC1142` (Denmark/Norway)
    - `EBCDIC1143` (Finland/Sweden)
    - `EBCDIC1144` (Italy)
    - `EBCDIC1145` (Spain)
    - `EBCDIC1146` (United Kingdom)
    - `EBCDIC1147` (France)
    - `EBCDIC1148` (International)
- The _DEBUG option now supports a list of case−insensitive keywords that can be entered to indicate which debug values to enable.
- The ConnectionError directive enables users to specify the message to be displayed when there is an Application Server connection error.
- The SHAREPOLL= and NOSHAREPOLL options enable the PROC APPSRV statement to control the period of SAS/SHARE libref polling and to disable polling of the SAS/SHARE server librefs, respectively.
- Load Manager log filename directives have been added to enable the rollover of Load Manager log files. Special codes inserted into the log filename specify the format and frequency for creating log files.
- Documentation for the BrokerPassword directive was added. The directive enables users to specify a password in order to protect the Application Broker administration interface.

About Application Dispatcher

Application Dispatcher, a SAS/IntrNet component, is a Web gateway from your Web browser to the power of SAS processing. This gateway, written by using the Common Gateway Interface (CGI), provides access to data in combination with a powerful array of analysis and presentation procedures. SAS software does not have to be installed on your machine!

To access and analyze data, a Web user completes an HTML form by selecting items and filling in fields. When the user selects the option to submit the information, the Dispatcher passes the information through the CGI program to a waiting SAS session. SAS software processes the information by using the identified program. Program results return through the CGI to the browser and are displayed to the waiting user.

You do not need CGI programming experience to use the Application Dispatcher. You can create the Web user interface and retrieve SAS data for display on the Web without having to program a CGI script.

An Overview of the Application Dispatcher

Using the Application Dispatcher, you can send information from a Web browser to a SAS session for processing and receive the results on your browser. The results can appear as text, HTML, GIF, JPEG, or any other format that is supported by your browser.

By submitting an HTML form or clicking a hypertext link, you can cause SAS to run a program. The program can be written by you or someone else at your site, or it can be a sample program that is provided by SAS. You can even use the sample forms and code as the foundation for your own applications. You can easily modify almost any SAS batch program to run on the Web and thus add new life to legacy applications and legacy data. Some commercial SAS applications that run by using the Application Dispatcher are included with SAS/IntrNet software.

To get started with Application Dispatcher terminology and concepts, read the following:

- What Is the Application Dispatcher?
- What Are Dispatcher Services?
- What Are Dispatcher Applications?
- What Is the Application Load Manager?
- Who Uses the Application Dispatcher?

What Is the Application Dispatcher?

The Application Dispatcher exchanges and processes information by using the following components:

The *input component* runs on the Web server or the client. It normally consists of static or dynamically generated HTML pages containing URL references or HTML forms. The input component is responsible for selecting what program component to run and what input data to pass to that program component.

The *Application Broker* is a CGI program that resides on your Web server (for example, in the cgi–bin or scripts directory). The Application Broker interprets the information received from the input component and passes it to the Application Server.

The *Application Server* is a SAS session that receives input from the Application Broker. The Application Server accepts information from the Application Broker CGI program and invokes the program component.

The *program component* is a SAS program invoked within the Application Server. The program

1. receives the request from the server
2. processes it
3. returns the results to the Application Broker for delivery to the Web browser and the waiting user.

What Are Dispatcher Services?

Each request from the browser contains the name of a service that will fulfill the request. The Application Broker identifies the service by looking into its configuration file and then determines where and how the request should be forwarded. The configuration file defines the three services (socket, launch, and pool) that are available for Dispatcher applications to use.

A *socket service* runs the Application Server continuously, waiting for new requests, and refers to the protocol that is used (TCP/IP sockets) to communicate between the server and the Application Broker. Using this type of service,

many servers can run at the same time, letting the Application Broker balance the load. As multiple users invoke Dispatcher programs, multiple servers can be utilized to improve application performance. An optional component called the Application Load Manager can be added to assist the Application Broker in balancing the load.

Instead of using the socket service method of running the Application Server, you can use the *launch service*, which starts a new server for each request. Although this method can require more time than the socket service because of the Application Server start−up time, it is easier to administer and provides some security advantages.

Using the Application Load Manager, the *pool service* starts servers from the pool as needed to handle queued jobs. When a job completes, the server becomes available for new requests until an optional idle time−out is reached, at which time the server shuts down.

What Are Dispatcher Applications?

The Application Broker and the Application Server provide the communication and processing mechanisms for Dispatcher applications. A *Dispatcher application* consists of one or more associated input components and program components. The program components for a Dispatcher application can be any of the following:

- a SAS program (an external file that has a *.sas* extension)
- a source entry (a catalog entry that has a *.source* extension)
- an SCL entry (a catalog entry that has a *.scl* extension)
- a compiled macro (a catalog entry that has a *.macro* extension).

The program component that you create must be designed to accept the information that is received from the Web browser. In many cases, this means that you not only create the source program but also the HTML page that passes information to the Application Broker.

What Is the Application Load Manager?

The *Application Load Manager* (LOADMGR) is a separate, optional process that can be used to enhance the distribution of Application Dispatcher resources on a network. If installed, it records the state of all Application Servers and maintains a separate dynamic pool of available servers. These capabilities enable the Load Manager to distribute Application Dispatcher requests most efficiently.

The Load Manager is of greatest use to the person who creates and maintains the configuration file for the Application Broker.

Who Uses the Application Dispatcher?

You should use the Application Dispatcher if you

- want to analyze and display information dynamically on the Web and let your Web users immediately retrieve the information they need.
- have SAS programming experience but have little or no CGI programming experience. Application Dispatcher enables you to create the Web user interface and retrieve the SAS data for display on the Web without writing a CGI script.
- want to create applications that provide Web output without investing a lot of programming time.
- want to create applications that run on a variety of Web browsers.

The Application Dispatcher has several types of users:

- End users enter information in a form, select a link, or view an inline image that displays in a Web browser.
- Web−page authors create the HTML forms or pages, which include unique Dispatcher fields. These individuals may or may not be SAS application developers.
- Dispatcher program developers create the Dispatcher programs that receive information entered on the Web page.
- System administrators, also known as Webmasters, maintain programs on the Web server or maintain the Application Server(s).

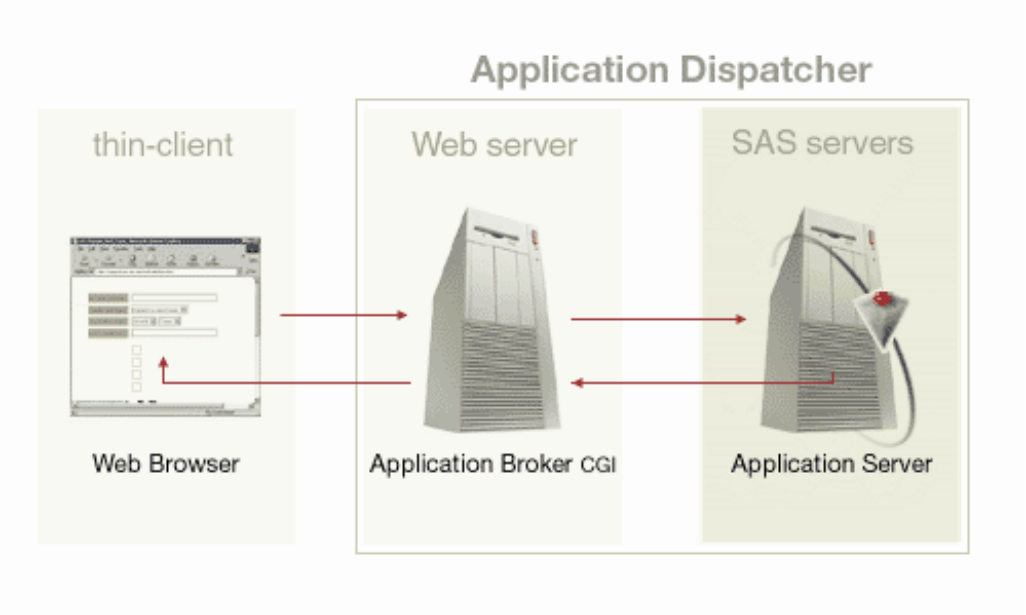
How the Application Dispatcher Works

- How a Request Is Submitted to the Application Broker
- How the Application Broker Processes a Request
- How the Application Server Processes a Request
- How Program Output Is Sent to the Application Broker
- How the Load Manager Works

The SAS/IntrNet: Application Dispatcher enables you to offer the power of SAS to Web users without having them install client software on every desktop. Here's a summary of how it works:

1. Users enter information in an HTML form using their Web browser and then submit it. The information is passed to the Web server, which invokes the first component of the Application Dispatcher, the *Application Broker*.
2. The Application Broker accepts data from the Web server and sends it to the second Application Dispatcher component, the *Application Server*.
3. The Application Server invokes a SAS program that processes the information.
4. The results of the SAS program are sent back through the Application Broker and Web server to the Web browser and the awaiting users.

The following diagram, which is described in detail later, illustrates how the Application Dispatcher submits and processes a communication request.



How a Request Is Submitted to the Application Broker

Web users submit information and processing requests via their Web browsers. The interface to the Application Dispatcher is usually an HTML form that users access from their Web browser, but users can also access the Application Dispatcher through a hypertext link that contains the URL and required parameters that are necessary to run the program.

Depending on the design and purpose of the form, users can

- formulate queries by selecting items from lists, check boxes, or radio buttons

- specify analysis variables, processing options, and reporting options by selecting items from lists, check boxes, or radio buttons, or by completing text entry fields
- input data by completing text entry fields.

When a user selects the **Submit** button or a hypertext link, the Web browser sends the information to the Web server, which immediately invokes the Application Broker. The Application Broker is a Common Gateway Interface (CGI) program that runs on any Web server that supports the CGI standard. The Application Broker then forwards the request to the Application Server.

In addition to providing the user interface, the HTML form provides

- the location of the Application Broker, as defined in the ACTION= attribute of the HTML FORM tag
- the name of the service that is used to process the request, as specified in the _SERVICE field
- the name of the program that executes the request, as specified in the _PROGRAM field
- generic name and value information that is contained in optional fields.

To learn how to enter these fields in HTML code and how to make your own Application Dispatcher form and its associated SAS code, see the Input Component and the Program Component.

How the Application Broker Processes a Request

The Web server invokes the Application Broker each time a user submits a request. The location of the Application Broker CGI program is provided by the ACTION= attribute in the HTML form or by the HREF= attribute in a hypertext link.

The _SERVICE field in the request specifies the service name. The Application Broker reads its configuration file to get the definition of the requested service. The Application Broker then connects to an Application Server that is associated with the service. Application Dispatcher currently supports three types of services:

Socket Service
: specifies one or more Application servers that run permanently on the SAS Server and wait for a request. The Application Broker either picks an available server or queries the Load Manager for an idle server.

Pool Service
: identifies multiple servers (a pool of servers) that might be running. The Application Broker contacts the Load Manager in order to find an available Application Server. If no server is available, the Load Manager can start a new server that will then handle the request. After the request is completed, the new server is added to the service pool and is available for future requests.

Launch Service
: enables the Application Broker to launch a new Application Server for each new request. The Application Server runs on the Web server machine and terminates when the request is complete.

The Application Broker forwards information from input fields and any configuration information that is specified in the Application Broker configuration file. For socket services, the configuration file also specifies the machine name or IP address and the TCP/IP port name or number that will receive the request.

How the Application Server Processes a Request

It can be helpful for you to know the specific steps that the Application Server performs when it handles a request. This information can be useful when you are enhancing server performance or performing administrative tasks. The following list describes how the Application Server processes a request:

1. After the Application Server is started, it listens on the TCP socket for a request.
2. When the server receives a request, it examines the _PROGRAM parameter to determine the type and location of the program that is being run. If the Application Server does not find the program, it generates an error page that displays in the user's browser.
3. The server creates macro variables and a SAS Component Language (SCL) list that contains the input name/value pairs so that the program can access them.
4. The Application Server creates a _WEBOUT fileref connection so that SAS can send data back to the browser.
5. The server runs the program, cleans up, and then waits for a new request. If the server was started for a launch service and no session was created by the user program, the server exits immediately.

How Program Output Is Sent to the Application Broker

The program output is sent directly to the Application Broker by using a predefined file reference. The format of the output is defined by the HTTP standard. See HTTP Headers for more information.

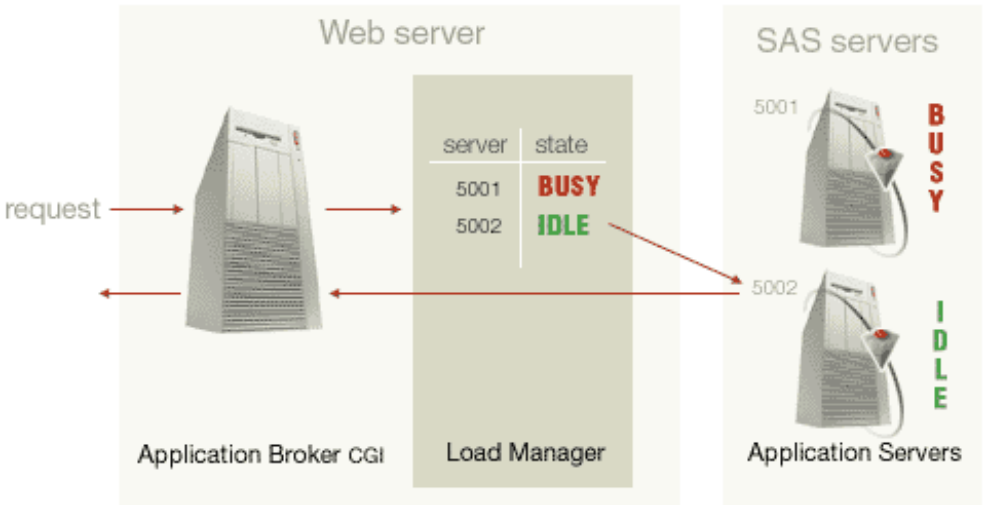
As the Application Broker receives the program output, it does a quick consistency check on the HTTP headers and sends the results back to the Web server, which streams the results back to the browser. Because of the streaming, results begin to appear in the browser before the program has finished processing.

How the Load Manager Works

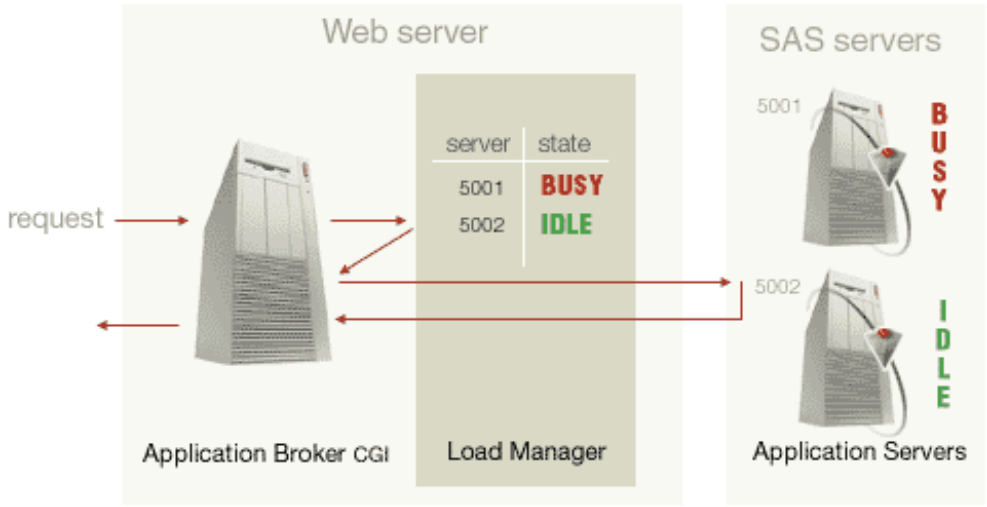
You can use the Load Manager, which is a separate, optional process, to optimize the use of Application Dispatcher resources on a network. The Load Manager can route requests to available idle servers and start additional available servers as needed.

- The Load Manager listens for requests from Application Brokers for Application Servers that are idle.
- If a server is not designated as being busy, it is allocated to the requesting Application Broker.

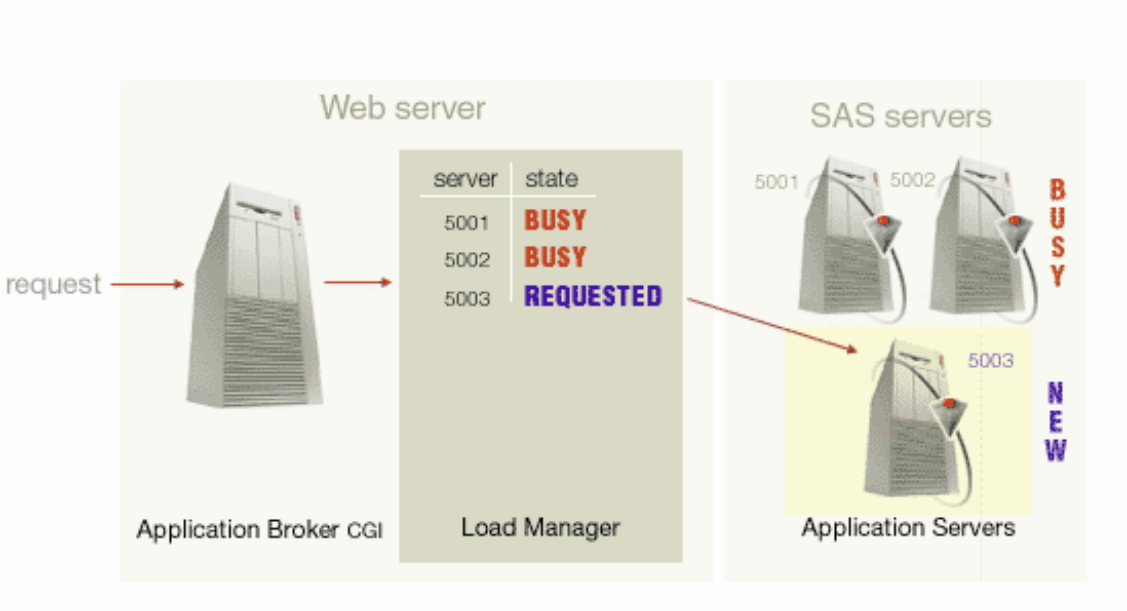
When the Application Server receives the job, it notifies the Load Manager that it is busy processing the request. The following diagram illustrates how the Application Server notifies the Load Manager after it has received a request.



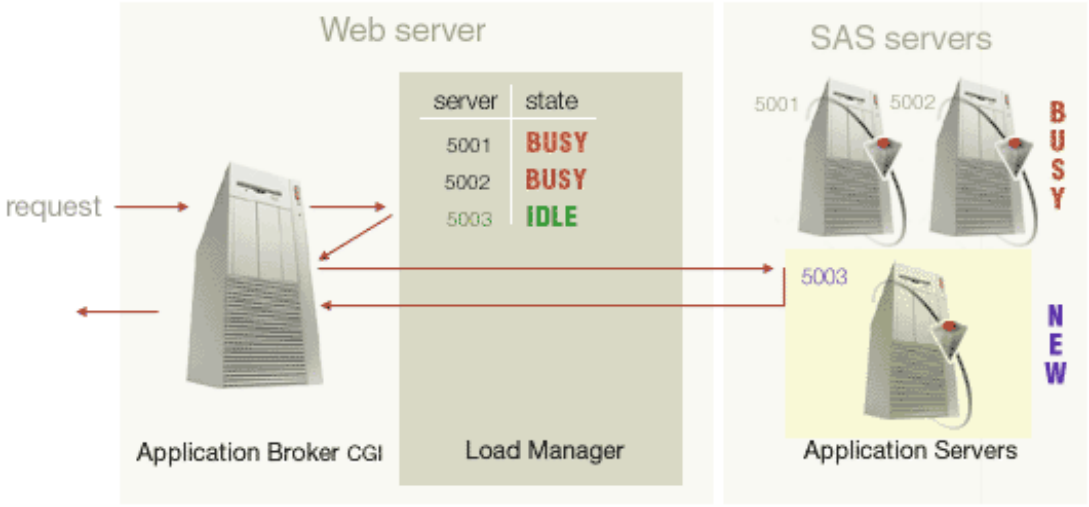
After completing the job, the Application Server again notifies the Load Manager that it is free to work on another request. The following diagram illustrates how the Application Server notifies the Load Manager after it is free to process the next request.



If no servers are free, the request is queued by the Load Manager until a server becomes available or until the time−out value is reached. For pool services, the Load Manager can start a new server to process the request. The following diagram illustrates how the Load Manager starts a new server to process the request.



The new server is then added to the pool of servers for that service. The following diagram illustrates how a new server is added to the pool of servers.



By using a Load Manager to maintain the state of Application Servers, requests are distributed to idle servers. Each Application Server notifies the Load Manager when it starts and completes a job by using the Load Manager socket address that is passed by the Application Broker. This data is used by the Load Manager during subsequent requests to determine which Application Servers are busy and to direct the Application Broker to use an available idle server.

Note: If the Load Manager is not installed, the Application Broker randomly selects an Application Server to process the request. Random selection of an Application Server is adequate for low−traffic environments. However, as traffic increases, requests can become stalled while waiting for a busy server even if another server is idle.

Requirements for the Application Dispatcher

The following are required:

- The end user must have a Web browser.
- The site must have a machine that has a Web server.
- The site must have a SAS server for SAS 9.1 with an Application Server installed.

These requirements (a Web browser, a Web server, and an Application Server) can reside on one machine or on three separate machines.

You must install both the Application Broker and the Application Server components of the Application Dispatcher. Install the Application Server on a machine that has SAS software running, and install the Application Broker on the machine that has your Web server running. A third component of the Dispatcher, the Application Load Manager, is not required for a typical setup. SAS software is not required on the Web server or the local desktop. The installation requirements are

- Requirements for the Application Broker
- Requirements for the Application Server
- Requirements for the Application Load Manager

Requirements for the Application Broker

- Because the Application Broker is invoked by the Web server, the Web server must already be installed on the machine where you plan to install the Application Broker.
- You must have WRITE access to the directory where your CGI scripts and programs are stored.
- You must have already installed at least one Application Server.
- The machine on which you install the Application Broker must have access by means of a TCP/IP–based network to the machine on which you installed the Application Server.
- For pool services, a Load Manager must be running on the network. A SAS spawner is also required if servers are to be started by using username or passwords.
- You must have access to a Web browser to test your installation.

Requirements for the Application Server

This documentation describes the Application Server for SAS 9.1. If your SAS server is an earlier version, refer to the documentation for that version.

- You must install the Application Server on a machine where SAS 9.1 is installed, because the Application Server is a SAS program.
- If you want to use a launch service, both the Web server and the SAS server must reside on the same machine. This is because the Web server invokes the Application Broker and the Application Broker invokes the SAS server.
- A socket service can be configured by using the Web server and the SAS server on the same or on different machines.
- For increased functionality, you can take advantage of other SAS software you already have installed, such as SAS/GRAPH, SAS/SHARE, SAS/ACCESS, or SAS/EIS software.
- The machine on which the Application Server is installed must have access to all of the data that is necessary to run the Dispatcher applications. This data can be stored in local SAS data sets, in third–party databases accessed through SAS/ACCESS software, or on remote servers accessed through SAS/SHARE software.

Requirements for the Application Load Manager

- You must install both the Application Broker and the Application Server in order to use Load Manager.
- Load Manager must have access to the Web server and the Application Server by means of a TCP/IP–based network. You can install Load Manager on your Web server machine, your SAS server machine, or any other machine on your network.
- You can run Load Manager as an unattended, background process, or you can start it as a system service.

Application Dispatcher Security

Security is a complex topic for networked, client/server applications. Security issues include user authentication, authorization, communications security, and writing secure applications. Here are some of the many approaches and tools you can use to secure the Application Dispatcher.

- Application Broker and Web Server Security
  - ♦ Using a Secure Web Server
  - ♦ Hiding Sensitive Information from Web Server Logs
  - ♦ Protecting the Application Broker Configuration File
  - ♦ Creating Encrypted Usernames and Passwords
  - ♦ Authenticating the Application Broker
- Application Server Security
  - ♦ The Application Server Should Not Trust the Application Broker
  - ♦ Application Server May Restrict Application Broker Access
  - ♦ Supplying a Password When Starting the Application Server
  - ♦ Hiding Passwords and Other Sensitive Data from the SAS Log
  - ♦ Restricting Access to Program Libraries
  - ♦ Disabling Sample Programs
  - ♦ Reviewing New or Modified Code
- Controlling Access to SAS Data Sources With the AUTHLIB Data Set
  - ♦ AUTHLIB Functions
  - ♦ Verifying the AUTHLIB Data Set
- Dispatcher Program Security
  - ♦ Using SCL or Compiled Macro Code
  - ♦ Using Password–Protected Data Sets

Related Topics

- Host Authentication
- Firewalls (support.sas.com/rnd/web/intrnet/misc/firewall.html)
- Application Broker Directives. See BrokerPassword under Administrator Directives and Encrypt under General Service Directives.

Application Broker and Web Server Security

- Using a Secure Web Server
- Hiding Sensitive Information from Web Server Logs
- Protecting the Application Broker Configuration File
- Creating Encrypted Usernames and Passwords
- Authenticating the Application Broker

Using a Secure Web Server

One security risk involves the network between the Web browser and the Web server. You can improve security by using a secure Web server. A secure Web server uses an HTTPS protocol, which is HTTP that has secure sockets. This protocol encrypts all the data flowing back–and–forth between the browser and the Web server. Unauthorized users are not able to decipher the secure packets of data as it passes through the various computers between the browser and the Web server.

Hiding Sensitive Information from Web Server Logs

Parameters on a GET request are logged by the Web server. This means that passwords and other sensitive parameters may be captured in the Web server log. Using POST generally prevents submitted form data from appearing in the Web server log files. You should use POST instead of GET to handle sensitive data in Application Dispatcher requests, although POST is not a guarantee that the Web server will not log input parameters.

Protecting the Application Broker Configuration File

As a security precaution, you should protect your Application Broker configuration file. Your first priority should be to restrict file system access so that only specific individuals can update the configuration file. Protecting this file means that you can rely on the settings you define in the file, such as DebugMask and Debug.

Usually the configuration file is stored in the CGI executable directory along with the Application Broker executable. Some Web servers allow files stored in a CGI executable directory to be downloaded the same way as a regular HTML file. To test this, try one of the following URLs, and see if your Web server allows you to download a copy of the configuration file:

UNIX:

```
http://yourserver/cgi-bin/broker.cfg
```

Windows:

```
http://yourserver/scripts/broker.cfg
```

If you are able to download the file, then you need to adjust your Web server configuration to prevent this. Next, make sure that the DebugMask value is set in your configuration file to disallow _DEBUG=4. This debug value displays the contents of the configuration file. Try one of the following URLs, to make sure that this debug flag is disabled:

UNIX:

```
http://yourserver/cgi-bin/broker?_debug=4
```

Windows:

```
http://yourserver/scripts/broker.exe?_debug=4
```

Creating Encrypted Usernames and Passwords

The username and password parameters in the Application Broker configuration file can be encrypted or entered as open text. If a value starts with an exclamation point (!), the value is assumed to be encrypted. To obtain the encrypted equivalent for a username/password, send the values to the Application Broker with a _DEBUG=1. For example,

```
http://abc.def.com/cgi-bin/broker?_service=default&_debug=1&_username=myname&_password=xyzzy
```

should produce output with the fields encrypted following an exclamation point (!). These new values can then be used in place of the original open−text versions.

Note: When you use the Application Broker to encrypt a username or password, the original unencrypted username and password may be saved in the Web server log. You can run the Application Broker from the command line to avoid this issue:

```
broker "_service=default&_debug=1&_username=myname&_password=xyzzy"
```

An alternative to entering this password in plain−text is to use the encrypted version of the password. For example, if your password is `xyzzy`, the encrypted version that you can put into the broker.cfg is `!ci3mC.Xmq.t2Chnx`. By hardcoding the encrypted version in the broker.cfg, the text of your actual password is protected from anyone who has read access to the broker.cfg file.

Authenticating the Application Broker

By default, Web servers enable any client to connect and make an anonymous request for a static page or a CGI program. You can enable Web server authentication for the CGI executable directory that contains the Application Broker. This requires that users supply a user ID and password in their Web browser to run the Application Broker. When a Web server launches a CGI program that is authenticated, it supplies the user ID of the client in the environment variable REMOTE_USER. By examining the corresponding SAS macro variable, _RMTUSER, your programs can determine who the requesting client is. Using this information, you can provide the appropriate access to application features.

Note that it may be easy for users to fake the value of the REMOTE_USER variable without authenticating. Because the Application Broker program accepts this value through an environment variable, users can set the environment variable REMOTE_USER to any value that they want and run the Application Broker from an operating system command prompt. Therefore, they can masquerade as other users and the Application Server does not know the difference. See the section titled Application Server Should Not Trust the Application Broker for an explanation of how to overcome this problem.

Application Server Security

The Application Server Should Not Trust the Application Broker

By enabling Web−server authentication for the Application Broker executable file, your Web server can pass the user ID of the client to the Application Broker in the REMOTE_USER environment variable. By exporting this environment variable, you can pass the remote−user value to SAS as the macro variable _RMTUSER. Then your application can use the value of this variable to activate or de−activate various application features.

Note: You cannot trust that the Application Broker that is connected to your Application Server is the same Application Broker that is executed on your Web server machine. It is possible that another Web server, or a user running the Application Broker from an operating system command prompt, could make a connection to your Application Server and supply a bogus value for REMOTE_USER. In other words, there is not a strong coupling between the Application Broker and the Application Server to ensure the transmission of a security context.

This is not to say that Web server authentication and the REMOTE_USER value are worthless. On the contrary, they are quite valuable as long as their reliability has not been compromised. One way to repair the security model outlined above is to create a strong coupling between the Application Broker and Application Server. If you can ensure that the only Application Broker that can run programs on your Application Server is the same Application Broker that you are authenticating on your Web server, then REMOTE_USER is a trustworthy value.

You can use the ServiceSet directive in the Application Broker configuration file to define a variable name and a constant value. For example,

```
SocketService default "Default service"
   Server appsrv.yourcomp.com
   Port 5001
   ServiceSet passkey "some value that is hard to guess"
```

With each request to the specified service, the Application Broker passes all the normal application data and the additional variable defined by using ServiceSet to the Application Server. In this example, each request would contain the name/value pair `passkey=some value that is hard to guess`. You can check the value of this special variable within your Application Server program. If the value of this variable passed to your program does not match the value you assigned in the configuration file, then you can refuse the request because it violates your security model. Any requests that perform the required Web server authentication will contain the special variable that has the correct value. These requests will pass your initial security check and then you can trust the value of _RMTUSER.

Because the Application Broker merges exported environment variables and variables created by using the Set directives after the Web server receives the request from the Web browser, there is no fear of the special variable appearing in the URL or in the Web server log files. In this example, only the Application Server sees the variable PASSKEY and its value.

There are some important caveats to this technique. Placing this "secret" value in the Application Broker configuration file means that you must protect the configuration file and make sure that no one can read its contents. The "secret"

variable and its value must be treated like a password. Secondly, you need to disable the _DEBUG=1 flag by using the DebugMask directive. This debug flag will display the "secret" variable and its value in the Web browser of the user along with the reset of the data for the request. You also have to include the "secret" variable and its value in your program code. This means that you have to restrict access to your program code just as you restricted access to the configuration file. See also Protecting the Application Broker Configuration File

Application Server May Restrict Application Broker Access

An Application Broker might be restricted from making requests from an Application Server if the Application Server's request syntax is limited to specific IP addresses.

For example, the Application Broker is started on a machine with IP address 12.34.56.78. The Application Server is started on another machine using the following syntax:

```
proc appsrv port=5800;
request fromadr=("12.34.56.99");
run;
```

A request from the Application Broker to this Application Server fails because the Application Broker's IP address does not match the APPSRV FROMADR.

Supplying a Password When Starting the Application Server

When you start the server, you can optionally specify an administrative password by using the ADMINPW option. For example,

```
PROC APPSRV  PORT=5001  ADMINPW='XXXX' ...
```

The password must not contain quotation marks. Specifying a password prevents anyone from running administrative programs such as STOP without supplying the password. By using an HTML form that has a password field, as shown in the following example, you can create an administrative interface to the server:

```
<FORM ACTION= "/cgi-bin/broker" METHOD="POST">
   <INPUT TYPE= "hidden" NAME= "_SERVICE" VALUE="default">
   <INPUT TYPE="hidden" NAME="_PROGRAM" VALUE="stop">
   Password:
   <INPUT TYPE="PASSWORD" NAME="_ADMINPW">
   <INPUT TYPE="SUBMIT" VALUE="Shut down">
   </FORM>
```

Note: Use METHOD=POST when creating forms that supply passwords to the Application Server. Including the password on a URL by using METHOD=GET or an explicit URL will expose the password in potentially unsecure places such as Web server logs, Application Server logs, history files or bookmark files.

Hiding Passwords and Other Sensitive Data from the SAS Log

The SAS log exposes programs and input parameters, which could pose a security issue. There are some actions you can take to hide passwords and other sensitive data from the SAS log. Password values are automatically hidden from the Application Server log. You can disable the SAS log with the DebugMask option. You can also use the prefix _NOLOG_ with macro symbols to hide request variable values.

The _NOLOG_ feature enables you to create special macro symbols that can be sent to the Application Server without publishing the macro values in the APPSRV log. The special macro symbols must start with the prefix _NOLOG_.

The prefix is case insensitive. For example:

```
http://yourserver/cgi-bin/broker.exe?_service=default
   &_program=test.getEmployeeSalary.sas&_nolog_salary=secretpw
```

If _NOLOG_SALARY is displayed in the SAS logs, it shows

```
_NOLOG_SALARY=XXXXXXXX;
```

Restricting Access to Program Libraries

Another recommended security measure is to segregate Dispatcher programs from their data. Dispatcher programs should not be placed in libraries along with the data they read and update. If your operating environment allows you to set read and write permissions for specific directories, then specify or assign the Application Server READ access only to all program libraries. Also, it is best to allocate program libraries by using the ACCESS=READONLY option. If any programs need to create or update data sets or files, then READ and WRITE access must be allowed to those data libraries. Though the Dispatcher functions correctly without these security measures, we strongly recommend that you follow these guidelines.

Do not permit unlimited access to existing application libraries. To achieve a high level of security, restrict access to the Dispatcher application libraries only to those developers who should be allowed to modify application code.

Disabling Sample Programs

The sample programs shipped with SAS/IntrNet are enabled by default. These samples programs include the ability to browse data in any library defined to an Application Server. If you wish to limit access to SAS libraries defined in a server's data libraries, you should disable the sample programs. You can do this by editing the appstart.sas file (or @APSTX*n* members on z/OS) for the Application Dispatcher service. You must remove or comment out the ALLOCATE LIBRARY statement for the SAMPLIB libref, remove or comment out the ALLOCATE FILE statement for the SAMPLE fileref, and remove SAMPLE and SAMPLIB from the PROGLIBS statement.

Reviewing New or Modified Code

To prevent security holes that could be created inadvertently, review any new code or code that has been changed in any way. See the section on Dispatcher Program Security for details about what to look for to find potential problems.

Controlling Access to Data Sources with the AUTHLIB Data Set

The AUTHLIB data set enables you to permit or restrict access to SAS library entities. The default name for the AUTHLIB data set is SASHELP.AUTHLIB. It contains INCLUDE and EXCLUDE rules that declare which data is available and which data is unavailable to a Dispatcher program. The enforcement of these rules is not automatic. A Dispatcher program must call the AUTHLIB functions in order to participate in this access control scheme. It is the responsibility of the programmer to incorporate the AUTHLIB functions into a program. The SAS Design−Time Controls are the only SAS/IntrNet components that automatically utilize the AUTHLIB data set in SAS/IntrNet Software.

The AUTHLIB data set has a specific structure:

Column Name	Type	Length	Description
Rule	character	7	The access rule for this record. Valid values are "INCLUDE" and "EXCLUDE".
Libname	character	8	The library name of the entity to which this rule applies.
Memname	character	32	The member name of the entity to which this rule applies.
Memtype	character	8	The member type of the entity to which this rule applies.
Objname	character	32	The catalog entry name of the entity to which this rule applies.
Objtype	character	8	The catalog entry type of the entity to which this rule applies.
Comment	character	128	An optional comment explaining this rule.

And here is a sample AUTHLIB data set:

Rule	Libname	Memname	Memtype	Objname	Objtype	Comment
INCLUDE	SASHELP	*	DATA	*	*	
INCLUDE	SASHELP	*	VIEW	*	*	
INCLUDE	SASHELP	*	MDDB	*	*	
INCLUDE	SAMPDAT	*	*	*	*	
EXCLUDE	SAMPDAT	MYCAT	CATALOG	*	*	

To customize the access control for your Application Server, you can modify the SASHELP.AUTHLIB data set that is shipped with SAS/IntrNet software, or you can copy this data set to a new name and modify that copy. If you use a data set name other than SASHELP.AUTHLIB for your set of access rules, you must use the APPSRV_AUTHDS function to set the new name.

Here is how the AUTHLIB data set is interpreted. An *entity* is any SAS library, member, or catalog entry.

- An INCLUDE rule indicates that access is allowed for matching entities.
- An EXCLUDE rule indicates that access is not allowed for matching entities.
- All explicit EXCLUDE rules override all INCLUDE rules.
- If an entity does not match any rules, then an implicit EXCLUDE rule is assumed.
- Variable values are not case sensitive.
- A single asterisk in a variable value matches any entity or partial entity name.

Here are a few additional guidelines:

- Keep it simple. Avoid creating an overly complex set of rules. This reduces the chance of unintentionally allowing access to sensitive entities.
- Verify any changes you make to the AUTHLIB data set.
- You cannot combine a text value with an asterisk to create a pattern match. An asterisk is effective only when used by itself.
- Do not leave any variable values blank. This does not evaluate properly. Place an asterisk in any columns that you might expect to leave blank. For example, OBJNAME and OBJTYPE do not make sense when the MEMTYPE is DATA. However, placing asterisks in these columns is required.
- Use a MEMTYPE value of CATALOG when you supply a nonasterisk value for OBJNAME or OBJTYPE. For example, suppose you want to exclude access to all catalog entries of type SCL. That rule would look like

Rule	Libname	Memname	Memtype	Objname	Objtype	Comment
EXCLUDE	*	*	CATALOG	*	SCL	Exclude all SCL entries.

- As stated above, the default rule (if none match) is EXCLUDE. If you add an INCLUDE rule with asterisks in all columns, this changes the default rule to INCLUDE, for example:

Rule	Libname	Memname	Memtype	Objname	Objtype	Comment
INCLUDE	*	*	*	*	*	Now all entities are included by default.

- If you add an EXCLUDE rule with asterisks in all columns, then no access is allowed to any entities, for example:

Rule	Libname	Memname	Memtype	Objname	Objtype	Comment
EXCLUDE	*	*	*	*	*	Turn off all access to SAS library data.

AUTHLIB Functions

The following functions enable you to use the AUTHLIB data set in your Dispatcher programs.

- APPSRV_AUTHLIB checks whether access is allowed for a given entity. The arguments to this function are similar to the columns of the AUTHLIB data set. This function is efficient if you are checking either a single or just a few entities. If you want to check many entities it is more efficient to use the APPSRV_AUTHCLS function.
- APPSRV_AUTHCLS produces various WHERE clauses. These clauses can be used to subset the entities in the current SAS session to only the entities that are authorized by the AUTHLIB data set. If your program needs to check the authorization for a large number of entities, or if your program needs to generate lists of authorized entities, then use this function. The returned WHERE clause can be combined with your own subsetting criteria and applied to the SQL dictionaries or various SASHELP views.
- APPSRV_AUTHDS changes the name of the AUTHLIB data set that is used by the other two functions.

Verifying the AUTHLIB Data Set

It is a good idea to verify all changes you make to the AUTHLIB data set. Fortunately, the APPSRV_AUTHCLS function makes this task easy. By using this function, you can generate lists of included and excluded entities that you can review for correctness. The following program produces a verification report for the AUTHLIB data set.

```
/*generate the different authlib WHERE clauses and store them as macro variables*/

data _null_;
   length clause $ 32767;

   clause = appsrv_authcls('LIBRARY');
   call symput('LIBCLS',clause);

   clause = appsrv_authcls('MEMBER');
   call symput('MEMCLS',clause);

   clause = appsrv_authcls('CATALOGENTRY');
   call symput('ENTRYCLS',clause);
run;


/*create a view of included libraries*/

proc sql;
create view work.inclib as select *
   from sashelp.vslib
   where &libcls;
quit;


/*create a view of the excluded libraries*/

proc sql;
create view work.exclib as select *
   from sashelp.vslib
   where not &libcls;
quit;


/*create a view of the included members*/

proc sql;
create view work.incmem as select *
   from sashelp.vmember
   where &memcls;
quit;


/*create a view of the excluded members*/

proc sql;
create view work.excmem as select *
   from sashelp.vmember
   where not &memcls;
quit;


/*NOTE: THE CATALOG ENTRY VIEWS CAN TAKE A LONG TIME TO RUN
YOU MAY WANT TO SUBSET BY ADDING SOMETHING TO
THE WHERE CLAUSE TO SPEED IT UP SUCH AS
```

```
and libname ne 'SASHELP'

THIS WILL PREVENT YOU FROM OPENING EVERY CATALOG
IN EVERY LIBRARY.*/


/*create a view of the included entries from selected catalogs*/

proc sql;
create view work.incentry as select *
   from sashelp.vcatalg
   where &entrycls;
quit;


/*create a view of the excluded entries from selected catalogs*/

proc sql;
create view work.excentry as select *
   from sashelp.vcatalg
   where not &entrycls;
quit;


/*Now print out the results of the SQL steps*/

proc print data=work.inclib;
proc print data=work.exclib;
proc print data=work.incmem;
proc print data=work.excmem;
proc print data=work.incentry;
proc print data=work.excentry;
run;
```

Dispatcher Program Security

- Using SCL or Compiled Macro Code
- Using Password−Protected Data Sets

Using SCL or Compiled Macro Code

One feature of the Application Dispatcher lets you view the SAS log. This helps when developing an application; however, it creates a potential security risk in a production−level application. Programs of the type .SAS, .SOURCE, and .MACRO all submit statements that appear in the log. SAS Component Language (SCL) statements do not appear in the log, but statements submitted by using an SCL submit block do appear. (SCL is available with SAS/AF software).

You can accomplish many of the same tasks in SCL that you can by using these other program types. SCL is the most secure program type. If you create your Dispatcher program with SCL and the user attempts to return the SAS log, your program statements do not appear. Additionally, SCL is more secure because it is a compiled language. Compiled macros (.MACRO program types) share this feature. Using SCL lets you compile the program and delete the readable source. This prevents someone from reading the program statements even if they gained access to the SAS catalog on the Application Server machine.

Running a .MACRO entry prints the original source to the SAS log if the MPRINT option is set. To prevent this, you can include the following statement in a request init program:

```
options nomprint;
```

Using Password−Protected Data Sets

You can protect access to your data by using password−protected data sets. This feature of SAS software lets you assign a password to a data set. You must then supply the password to access or to modify the data set. You can choose to code the password to the data set in your application or require the user to supply it. If you code the password into your application, ensure that the user cannot view that password by returning the SAS log to the browser or by reading your source code files.

Upgrading from Version 8 to Version 9

The following information details the steps that you must take in order to upgrade from Version 8 to Version 9 of Application Dispatcher.

Note: If you are upgrading from a release prior to Version 8, read the documentation for Release 8.2 of Application Dispatcher (at support.sas.com/rnd/web/intrnet), which describes upgrading from prior releases.

1. You must install the SAS 9 CGI Tools package on your Web server to access the SAS 9 Application Server. Note that the SAS 9 Application Broker and Load Manager will work with existing Version 8 Application Dispatcher services, so you can upgrade one or more services to SAS 9 while keeping other services at previous versions.
2. Upgrade socket services by editing the server start script and changing the path to the SAS executable from your Version 8 installation to the new SAS 9 executable. The following table lists the server start script for each operating environment.

Operating Environment	Server Start Script
Windows	`appstart.bat`
UNIX	`start.pl`
z/OS	`APSTRTn JCL`
OpenVMS	`start.com`

   No changes are required for the `appstart` SAS program or other support files.
3. Upgrade pool and launch services by editing the Application Broker configuration file and changing the path of the SAS executable. No changes are required to the `appstart` SAS program or other support files.
4. If you are using Load Manager, modify your procedure for starting Load Manager to use the SAS 9 executable. On Windows, use the "Create a New IntrNet Service" utility to set up a SAS 9 Load Manager. Uninstall any existing Window Load Manager services from the Version 8 SAS System Start Menu. On UNIX you may need to edit the `loadmgr` script created by the inetcfg.pl utility.
5. If you are using Spawner for pool services, modify your procedure for starting the Spawner to use the SAS 9 executable.
6. Existing programs from earlier versions will run on the SAS 9 Application Server without modifications in most cases.

Completing the Installation

Before you can use the Application Dispatcher, you must perform the following steps:

1. If you are upgrading from a previous version, see Upgrading From Version 8 to Version 9 for more information before completing the installation.
2. Install SAS 9.1 (including SAS/IntrNet.) The SAS/IntrNet software that is installed in this step includes the **Application Server**.
3. Install the **Application Broker**, which is contained in the CGI Tools for Web Server. The instructions for installing the CGI Tools for Web Server package are included with SAS/IntrNet software, which is available from the SAS Client−Side Components CD Volume 2 or Volume 3.
4. Create and start the default service for the Application Server by using the inetcfg utility.
5. Add the default service definition to the Application Application Broker configuration file.
6. Run the sample applications to test your installation.

Note: For z/OS, the SAS 9 or later Application Broker requires that the IBM Web maintenance patch, PQ47248, be installed if you intend to use a Web server codepage (FSCP) other than `ibm-1047`.

Create and Start the Default Service

You must create a default Application Dispatcher service to run any of the sample programs supplied with SAS/IntrNet. Before you create the service, you must reserve a TCP/IP port number for your default Application Dispatcher service. Consult your system administrator or check your services definition file to find an available port number.

You should create the default service with the SAS/IntrNet Configuration Utility (inetcfg). This utility is available for

- Windows platforms
- UNIX platforms
- z/OS
- OpenVMS.

After you create and start the service, continue with Adding the Default Service Definition to the Application Broker configuration file.

Windows Platforms

Services can be created on Windows platforms with a configuration utility that is accessible from the Start menu.

Perform the following steps to create and start the default service:

1. From the Start menu, select **Programs** ➧ **SAS** (or other program group where SAS is installed) ➧ **IntrNet** ➧ **Create a New IntrNet Service**.

   The IntrNet Config Utility Welcome window appears.
2. Read the information in the Welcome window, and then select **Next** to continue.
3. Select **Create a Socket Service**, then select **Next** to continue.
4. Type `default` as the name of the new service. Select **Next** to continue.
5. Specify the directory where you want the configuration utility to place your service directory and control files. The default location (under your SASUSER directory) is recommended. Select **Next** to continue.
6. Type the TCP/IP port number that you reserved for the default Application Dispatcher service. Select **Next** to continue.
7. A password is not necessary for the default service. You can add an administrator password later if you use this service for production applications. Select **Next** to continue.
8. The Create Service window displays all of the information that you specified for this service. Verify that the information is correct and then select **Next** to create the service.
9. Select **Next** and then **Finish** to complete the setup of the default service.
10. From the Start menu, select **Programs** ➧ **SAS** (or other program group where SAS is installed) ➧ **IntrNet** ➧ **default Service** ➧ **Start Interactively**. Your default Application Server should now be running.

UNIX Platforms

On UNIX platforms, the configuration utility is a Perl script. Perform the following steps to create and start the default service:

1. From a system prompt, submit the following command:

   ```
   SASROOT/utilities/bin/inetcfg.pl
   ```

where *SASROOT* is the path to the SAS root directory.

As the configuration utility runs, you are prompted for information about the service that you are creating.

2. Press **Return** to accept the default value, which names the service *default*.
3. The next prompt asks for the name of the directory where all of the service control files should be stored. Press **Return** to accept the suggested value, or type the desired directory name and then press **Return**.
4. Type **S** and press **Return** to define a socket service.
5. Press **Return** to select one server.
6. Type the TCP/IP port number that you reserved for this service and press **Return**.
7. Press **Return** to skip entering an administrator password. You can add an administrator password later if you use this service for production applications.
8. Verify the displayed information and press **Return** to create the service. Note the path for the service directory.
9. The configuration utility created a start.pl file to start the default Application Server. Change to the service directory path and start the server by submitting the following command:

```
./start.pl
```

z/OS

The configuration utility provided for z/OS is a batch job. It is installed as a member named INETCFG in the CNTL data set that you created as the first step in the installation of your SAS software. To use the utility, you must edit the parameter file, member INETEDTP in the CNTL data set, edit the INETCFG job, and then submit the INETCFG job. The INETEDTP member contains the parameters necessary for creating a service.

To create and start the default service on z/OS, perform the following steps:

1. Edit the member INETEDTP.
2. Verify the name of the service that you are creating. The service name is defined by the line beginning with ISVC= and should be DEFAULT.
3. Verify that the service is a socket service. The line containing ISVCTYP=%SOCKETTYP should be uncommented.
4. Locate the line I$−PORT1. Change the value 5001 to the correct port number or network service name for your Application Server.
5. Save and close INETEDTP.
6. Edit INETCFG to verify the job header information. Verify that the service name is DEFAULT. If you make changes, be sure to save them. Do NOT change SASEDITP to INETEDTP because this is your original SAS installation parameters file.
7. Submit the INETCFG JCL job for processing. The INETCFG job will submit another job (INETCFGA). Verify that both jobs completed with a return code of 0. If they completed successfully, you now have the data sets and members necessary for running the default service.

   If the INETCFG job failed, examine the messages and sysprint members for error messages. If you see a message that reads

   ```
   ERROR: THIS REPLACEMENT CAUSES RESULT TO EXCEED OUTPUT LRECL
   ```

   you might have supplied a pathname in one of your INETEDTP parameters that is too long. Try shortening this pathname and rerun INETCFG.

   Note: Before you run INETCFG again, you must delete any data sets created by the previous failure of INETCFG. You can find these data sets by looking in the namespace determined by your original SAS install.

For example, if your SAS software was installed with the prefix name SYS.SAS and your failed INETCFG was trying to create the DEFAULT service, then delete all data sets beginning with the name prefix SYS.SAS.WEB.DEFAULT before running INETCFG again.

8. The configuration utility creates a server root in a partitioned data set (PDS) named *prefix*.WEB.DEFAULT, where *prefix* is the data set prefix that you supplied during your SAS installation. The PDS contains any JCL procedures and server start−up code required for starting the service. You will find the following members:

   APSTRT1
   : contains the JCL necessary to run the corresponding @APSTX1 file as a started task. **You should move this file** to your started task library and enable it as a started task.

   @APSTX1
   : contains the SAS code that invokes the server. This file is called by the JCL in the corresponding APSTRT1 file. This SAS program must remain in the PDS where it was created.

   In addition to the server root PDS, the configuration utility creates an empty PDS named *prefix*.WEB.DEFAULT.TDIR, where *prefix* is the data set prefix that you supplied during your SAS installation. The Application Server will use this PDS as a scratch location.
9. You must modify the permissions for the data sets created above so that the server can write to them as necessary. To modify the permissions, create a special RACF data set profile that applies to all the data sets in this service (*prefix*.WEB.DEFAULT.*). The RACF data set profile should also grant write access to the user ID of the Application Server.
10. Issue a START command from the system console to start the default Application Server.

OpenVMS

To create and start the default service on OpenVMS, perform the following steps:

1. From a system prompt, submit `@sas$root:[tools]inetcfg.com`.

   As the configuration utility runs, you are prompted for information about the service that you are creating.
2. Press **Return** to name the service DEFAULT.
3. Press **Return** to accept the suggested value for the server root directory.
4. Type **S** to define a socket service. Press **Return** to continue.
5. Press **Return** to select one server for this service.
6. Type the TCP/IP port number or name that you reserved for the default Application Server and press **Return**.
7. A password is not necessary for the default service. You can add an administrator password later if you use this service for production applications. Press **Return** to continue.
8. The utility displays the information that you entered for this service. Verify that the information is correct. If the information is correct, press **Return** to create the service. Read the messages to determine if the service was created correctly. One of these messages contains the path for the service directory created by the utility. You should note this path for use later in this process.
9. The configuration utility created a START.COM file that starts the default Application Server. To start the server, change to the service directory created by the utility. Then submit the following command:

   ```
   @START.COM
   ```

Add the Default Service Definition

After you have created the default service, you must add the service definition to the Application Broker configuration file. The configuration file is usually named broker.cfg and lives in the same directory as the Application Broker executable. The following instructions describe just the changes needed for the default service. See Using the Configuration File for more information about this file.

1. Open the configuration file, broker.cfg, in edit mode. The configuration file is in the directory where you installed the Application Broker.
2. Search the file for Global administrator. Change the values for Administrator and AdministratorMail to appropriate values for your site.
3. Search the file for SocketService default.
    - Change the value for Server from appsrv.yourcomp.com to the DNS name or IP address of the machine where you created the default service (where SAS software is installed).
    - Change the value for Port from 5001 to the TCP/IP port that you selected when creating the default service.
4. Save the changes to the configuration file and continue to Testing the Installation.

Testing the Installation

If you follow the instructions for Completing the Installation, you should have an Application Server running and the Application Broker should be installed in your Web server CGI directory. Before trying to write applications of your own, perform the following steps to verify that everything is working correctly.

1. Test the Application Broker by pointing your Web browser to the Application Dispatcher URL. For example,

   Windows:

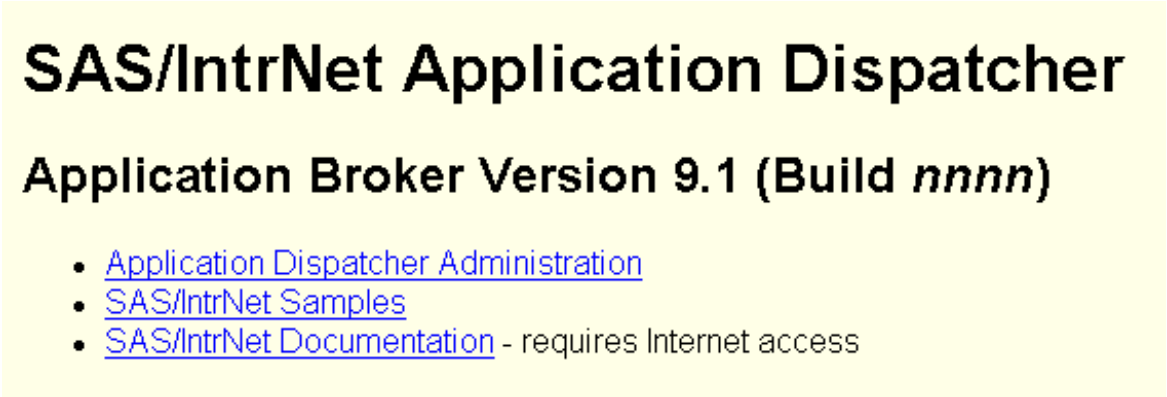
   `http://yourserver/scripts/broker.exe?`

   Other hosts:

   `http://yourserver/cgi-bin/broker?`

   Replace *yourserver* with the name of your Web server. The URL path might also need to be changed if you installed the Application Broker to a different directory.

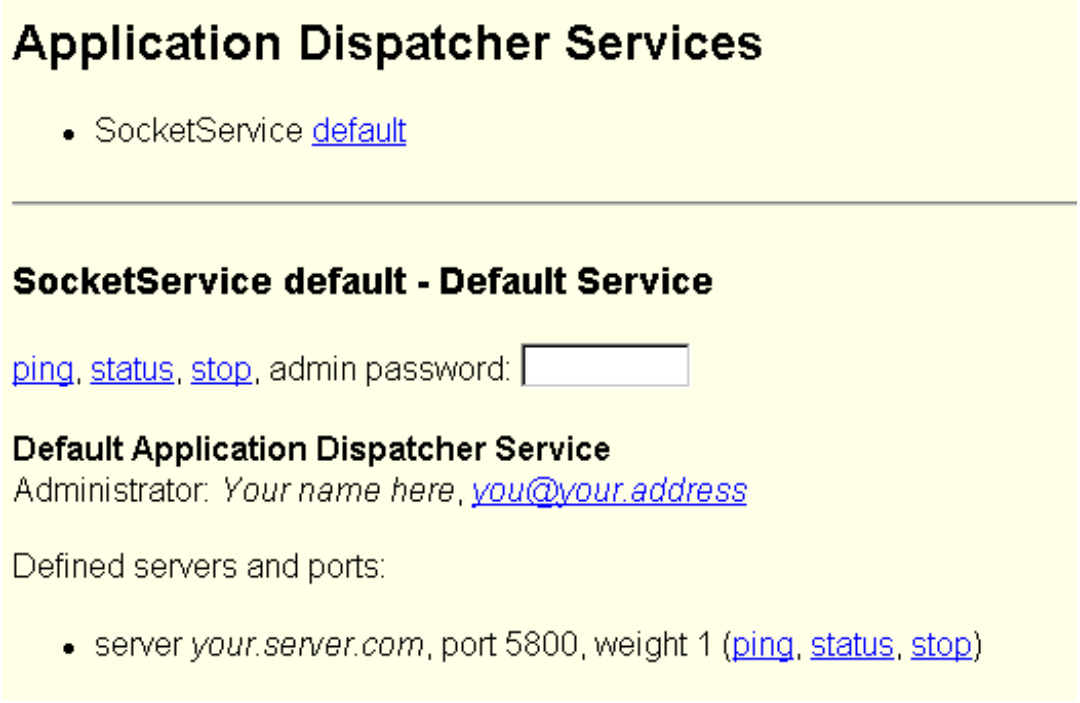
   If the Application Broker is working, you receive a page similar to the following:

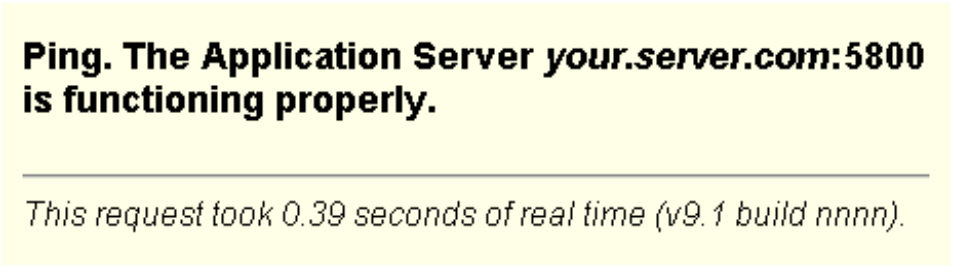
   Note: If there is a customized Application Broker welcome page, then it will display instead of this default welcome page when you enter the Application Broker URL in your browser. If this is the case, and if you want to view the services that are available from the default welcome page, then add _DEBUG=4 to the URL, as follows:

   `http://yourserver/cgi-bin/broker.exe?_debug=4`

2. Click on the Application Dispatcher Administration link to see if the Application Broker can read the Application Broker configuration file. The response looks like



3. Ping the Application Server by clicking on the ping link in the Application Dispatcher Services page. If the server is working correctly, the response is



4. To complete installation testing, return to the main Application Dispatcher page (see step 1) and select **SAS/IntrNet Samples**. Try some of the Application Dispatcher samples to verify the complete installation.

   Note: If the samples fail, stop the Application Server and examine the SAS log file. You can stop the Application Server by clicking on the **stop** link on the Application Dispatcher Services page (see step 2).

Completing the Application Dispatcher Installation

Congratulations! If you followed all the steps, you now have a working Application Dispatcher. While you should find this setup sufficient for many simple applications, the Dispatcher includes additional features that easily handle more complex applications. For details about creating your own Dispatcher applications, see The Input Component and The Program Component. For details about additional settings or customization options, see Customizing the Application Dispatcher.

Customizing the Application Dispatcher

The topics listed here provide information for customizing your Application Dispatcher installation. Follow the instructions in Completing the Installation if you have not already done so.

Customizing the Application Broker

- Using the Application Broker Configuration File
- Creating a Customized Welcome Page
- ISAPI/GWAPI Application Brokers
- Specifying the Global Administrator
- Specifying the Self−Referencing URL
- Specifying HTTP Methods
- Setting the Default Value of _DEBUG
- Using the DebugMask and ServiceDebugMask
- Displaying the Powered by SAS Logo
- Exporting Environment Variables
- Configuration File Directives

Customizing the Application Server

- Running Multiple Application Servers at Your Site
- Application Server Administration Programs
- Application Server Libraries

Using the Application Broker Configuration File

The Application Broker is controlled by the directives in a configuration file. Usually, the configuration file is named broker.cfg and lives in the same directory as the broker executable, but other names or directories can be used in special cases. The Application Broker searches for the configuration file in the following manner:

1. builds the configuration file name by adding .cfg as the file type to the executable name. For example, broker.exe would look for broker.cfg and broker7 or broker7.cgi would look for broker7.cfg.
2. checks for the environment variable BROKER_CFG. If this variable exists, it is assumed to contain the path with the configuration file. If the BROKER_CFG variable exists, the configuration file must exist in this directory or the Application Broker will fail to execute.
3. checks for the configuration file in the same directory as the executable.

Platform Notes

UNIX

- Check /usr/local/lib/IntrNet/broker/ for the configuration file.

z/OS

- Check /usr/local/lib/IntrNet/broker/ for the configuration file.
- Starting with Release 8.2, the Application Broker configuration file is assumed to be in the encoding specified by the Web server's file system codepage option (FSCP).

If the configuration file is not found in any of the locations above, the Application Broker will fail to execute.

Template Configuration File

A template Application Broker configuration file named broker.cfg_v9 is installed with SAS/IntrNet. If this is the first installation of SAS/IntrNet, the template file will be installed as your initial Application Broker configuration file. The template contains example directives to help configure the Application Broker for your site. The following pages describe some of these directives in greater detail.

- Specifying the Global Administrator
- Specifying the Self−Referencing URL
- Specifying HTTP Methods
- Setting the Default Value of _DEBUG
- Using the DebugMask and ServiceDebugMask
- Displaying the Powered by SAS Logo
- Exporting Environment Variables
- Configuration File Directives

Modifying the Application Broker Configuration File

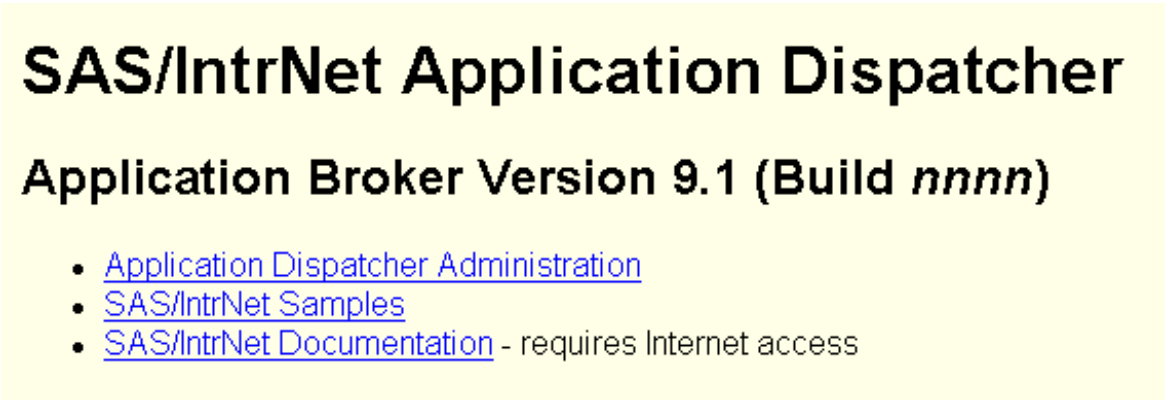
Use the following guidelines when modifying the template configuration file:

- Comments start with # as the first non−blank character.
- Because the Application Broker ignores leading spaces, you can include them to make the file easier for you to read.

- Each line of the configuration file must not extend beyond the first 256 columns. The plus sign (+) at the end of a non−comment line is used for line continuation.
- Quotation marks are required for values that contain blanks.
- You can use single or double quotation marks. Values that might require quotation marks are filenames and descriptions.
- If a configuration file entry accepts multiple values, delimit the values with spaces only.
- To specify a single quotation mark in a value, use \'.
- To specify double quotation marks in a value, use \".
- To specify a single backslash in a value, use \\.
- If an entry in the configuration file begins with #, you can activate that entry by removing the #.
- To complete an entry, delete or modify the text provided in the sample file. Replace this text with information that is valid for your site and installation.

Creating a Customized Welcome Page

When the Application Broker is invoked with no parameters, it displays a default welcome page that looks like this:



To display a customized welcome page, create an HTML file in the same directory as the Application Broker configuration file. The name of the file should be the same as the configuration file with a file type of "html" instead of "cfg". For example, if the configuration file is named broker.cfg, then the customized welcome page should be named broker.html. For most installations, the customized welcome page will be named broker.html and be located in the same directory as the Application Broker executable.

ISAPI/GWAPI Application Brokers

Two additional versions of the Application Broker have been developed for heavily loaded systems where performance is critical. These versions are built as shared libraries that are linked directly into the Web server at run time. When a new request is accepted by the Web server, it starts an Application Broker copy in a Web server thread rather than starting a new CGI process. In this manner, the overhead of process creation is replaced by the creation of a new Web server thread.

The two new modules are broker.dll (ISAPI Windows) and broker.so (GWAPI z/OS). These files are typically installed into the Web server CGI directory. When the first request is made to the Web server, these modules are loaded and linked to the Web server. The Application Broker configuration file is read once and stored in memory. This means that if the configuration file is subsequently changed, the Web server must be stopped and restarted in order to reload the changes. On Windows, the IIS Admin Service must also be stopped and restarted, using the Control Panel/Services dialog box. To see when the configuration file was last read, invoke the Application Broker with a _DEBUG value of 16384.

ISAPI

To use the ISAPI version on Windows, change the URL in the Web browser from broker.exe to broker.dll. A URL of the form

```
http://yourserver/cgi-bin/broker.dll?
```

loads and executes the ISAPI module. If no parameters are specified, then the default or optional customized welcome page is displayed.

Note: For Apache Web servers on Windows, the ISAPI Application Broker will only work with Apache 2.0 or greater and with the following configuration lines added for your CGI directory to the Web server HTTP.CONF file:

```
ISAPICacheFile c:/cgi-bin/broker.dll
Addhandler isapi-isa .dll
```

You must also add the ExecCGI options line to the directory section of the HTTP.CONF file, as follows:

```
<Directory "C:/Program Files/Apache Group/Apache2/cgi-bin">
   AllowOverride None
   Options ExecCGI
   Order allow,deny
   Allow from all
</Directory>
```

For ISAPI, the Application Broker thread typically runs under the user ID IUSR_*nodename*, as it does with the CGI version.

GWAPI

The z/OS GWAPI version requires a Web server configuration change. Add a line of the form

```
Service /cgi-bin/gwbroker* /dept/test/cgi-bin/broker.so:broker
```

to the Web server configuration file httpd.conf. In addition, execute the following command for the broker.so module:

```
extattr +p /dept/test/cgi-bin/broker.so
```

The exact form of the commands depends on the directory specification for the CGI directory. Changing the URL to the form

```
http://yourserver/cgi-bin/gwbroker?
```

loads and executes the GWAPI module, which will look for an Application Broker configuration file named gwbroker.cfg. If no parameters are specified, then the default or optional customized welcome page (gwbroker.html) is displayed.

For GWAPI, the Application Broker runs under the same user ID as the Web server.

Note: The Application Broker encryption option is not available with the GWAPI Application Broker.

Specifying the Global Administrator

The *global administrator* is the designated contact for service definition requests and general Broker problems. You can specify a single person or a group of people. At least one administrator must have write access to the configuration file and understand the information required to define a service.

To specify information about the global administrator

1. Locate the following two lines that appear near the top of the file:

   Administrator "*Your Name*"
   AdministratorMail "*yourname@yoursite*"
2. Replace *Your Name* with the administrator's name. This value can include spaces.
3. Replace *yourname@yoursite* with the fully qualified e−mail address for the administrator.

These two values become the macro variables _ADMIN and _ADMAIL in your Dispatcher program.

Specifying the Self–Referencing URL

The self–referencing URL identifies the Application Broker program URL. In most cases you will not need to set this value. The URL is passed to the SAS program in a macro variable called _URL. The Web server uses the `script-name` environment variable to change the value of _URL. The Application Server uses the variable _URL to generate the variables _THISSRV and _THISSESSION.

You might need to change the self–referencing URL in the following situations:

- if your Web server does not set the `script-name` environment variable or sets it incorrectly.
- if your site uses load balancing with its Web servers. For example, a site might have a Web server `www.company.com` that dynamically refers all browser requests to a range of Web servers from `www1.company.com through www5.company.com`. In this situation, the self–referencing URL might direct the request to `www2.company.com/cgi-bin/broker` rather than to the original, load–balanced Web server. If all the company's Web servers used the Dispatcher, you could set the self–referencing URL to point to `www.company.com/cgi-bin/broker`, which would direct all requests to the load–balanced service to maintain the use of Web server load balancing with each page of the Dispatcher application.

Specifying HTTP Methods

The HTTP methods specified in the ALLOW directive are the two methods used by the HTTP server to pass information to the CGI program (Application Broker). The ALLOW directive lists the allowable values for the request method; this line does not actually set the method. The method names are GET and POST:

- GET tells the server to process the entire form as one long concatenated string of values appended to the URL. Using GET allows users to bookmark the resulting dynamic pages. However, the resulting page's URL can become very long and display variable information that you might prefer not to display.
- POST sends the form data in a long input stream, which is not visible to users. Using POST is helpful when processing a large amount of data. However, users cannot bookmark the resulting pages.

To specify which HTTP methods the Application Broker should allow, locate the following line in the configuration file:

```
Allow get post
```

If you want to allow both methods, leave the line as it is. If you want to allow only one method, delete the method that you do not want to allow. By default, both methods are allowed, so commenting or omitting the directive allows both GET and POST.

As stated, the ALLOW directive does not set the HTTP method. That is done in each HTML page that references the Application Broker. The author of the HTML portion of a Dispatcher application specifies either the GET or POST method in the HTML form tag, for example:

```
<form action=<location of Application Broker> method=post|get>
```

One simple, but not ironclad, security technique is to use the POST method when you invoke the Application Broker. In your HTML form tag, specify ACTION=, which points to the Application Broker. In addition, you can specify a method as shown in the following example:

```
<form action="/cgi-bin/broker" method="post";>
```

The POST method passes all form variables to the Application Broker on standard input, which prevents them from appearing as part of the URL. This method makes it more difficult for users to subvert the values sent to your program.

Note: Using POST prevents the submitted form data from appearing in the Web server log files. POST also prevents you from bookmarking those dynamically generated pages.

Setting the Default Value of _DEBUG

If you are writing your own Dispatcher applications or are having problems with some of the samples we provide, you might want to specify a different default _DEBUG value or keyword. The default is 2, or TIME, which will display the elapsed processing time, the Powered by SAS logo (if available), and the Application Broker build number at the bottom of the Web page.

However, you can set a different default value that will take effect if the _DEBUG field is not included in the HTML page. The config file directives Debug and ServiceDebug set the default values for the _DEBUG field. To set them, follow the directive with the value that you want as the default.

See the List of Valid Debug Values for a complete list of debug values and keywords.

Remember that you can set more than one debug option. To do so, add the option values together. For example, to set both the 2048 and 4 options, enter `2052` as the value for Debug, DebugService, or _DEBUG, as appropriate. You can also use more than one keyword separated by spaces or commas to specify more than one option. For example, to set both the 2048 (TRACE) and 4 (SERVICES) options, specify

```
_DEBUG=TRACE SERVICES
```

Using DebugMask and ServiceDebugMask

The Application Dispatcher has several debugging options that can be turned on and off through the _DEBUG field in Dispatcher requests. Some of these options might represent security risks, including a few that are not documented and are used by Technical Support. For example, the Dispatcher includes an option to show the SAS log (which might contain source code), the host name and port number where the Application Server is running, or a list of all services known to the Application Broker.

To create a secure Dispatcher setup, decide which debugging options you want to allow and set the value of DebugMask or ServiceDebugMask in the Application Broker configuration file to the sum of those options. Add together the debug values that you want to allow and use that number in the directive. For example, if you want to allow only the field echo (1), status message (2), and output dump (16) values, you would set DebugMask to 19 (1+2+16). You can also use keywords to specify these options. For a list of valid debug values and keywords, see the List of Valid Debug Values.

Note: By default, all debugging options are allowed because the DebugMask and ServiceDebugMask directives are global and by−service directives.

The default value for the DebugMask is **32767**, which is acceptable for most sites. The value 32767 indicates that all debug values are allowed. If you comment out the DebugMask option by maintaining the # sign in front of DebugMask, you are also allowing all debug values.

Some debug values pose a security risk, so it is recommended that you selectively disable these values by specifying a different DebugMask value. Setting a different DebugMask value dictates the allowable values for the _DEBUG field in the HTML form or link.

Displaying the Powered by SAS Logo

You can include the Powered by SAS logo at the end of every request on the bottom of the returned page. To enable this, complete the following steps:

1. Download the Powered by SAS logo from the SAS Web site. Review the logo guidelines before downloading the Powered by SAS logo.
2. Edit your Application Broker configuration file (broker.cfg). Find the four directives beginning with #SASPowered and remove the leading pound sign on each line to enable the Powered by SAS logo.
3. Set the _DEBUG=2 flag in your HTML code to show the Powered by SAS logo and the Application Broker build number in addition to the elapsed time. You can enable the logo for all programs by defaulting the debug mask in the broker.cfg file with the Debug 2 directive.

The Application Dispatcher adds the image in the status line at the bottom of each results page.

Exporting Environment Variables

The following table from the sample configuration file lists some standard CGI environment variables. However, you can pass any variables that your Web server supports. (For more information about environment variables, see the CGI area at the World Wide Web Consortium Web site at www.w3.org.)

Note: These SAS macro variable names are suggestions only; you do not need to use these exact names.

Environment variables	SAS macro variable	Description
GATEWAY_INTERFACE	_GATEWAY	Version of the Common Gateway Interface (CGI) that the Web server uses.
SERVER_NAME	_SRVNAME	Web server's DNS (host) name or IP address.
SERVER_SOFTWARE	_SRVSOFT	Web server software name and version.
SERVER_PROTOCOL	_SRVPROT	Name and revision of the HTTP information protocol transmitting the client request.
SERVER_PORT	_SRVPORT	Web server port number.
REQUEST_METHOD	_REQMETH	Method with which the information request was issued, for example, GET or POST. This corresponds with the <FORM...METHOD=GET|POST> statement in the HTML form.
PATH_INFO	_PATHINF	Extra path information after the script passed to a CGI program.
PATH_TRANSLATED	_PATHTRN	Local filename of PATH_INFO.
SCRIPT_NAME	_SCRIPT	Virtual path of the script being executed. In this case, a duplicate of _URL, another macro variable passed to the Dispatcher program.
DOCUMENT_ROOT	_DOCROOT	Directory from which Web documents are served. This variable is unreliable.
QUERY_STRING	_QRYSTR	Query information passed to the program. It is appended to the URL with a question mark (?). In this case, it is an unparsed version of the user macro parameters. Set only with GET.
REMOTE_HOST	_RMTHOST	User's DNS (remote host) name, if known.
REMOTE_ADDR	_RMTADDR	User's IP address.
AUTH_TYPE	_AUTHTYP	Authentication method used to validate a user, usually Basic.
REMOTE_USER	_RMTUSER	Username, if authenticated.
REMOTE_IDENT	_RMTID	Identification of user making request. RFC931 ID, if supported.
CONTENT_TYPE	_CONTTYP	The Internet media type (MIME type) of the query data. Set only with POST.
CONTENT_LENGTH	_CONTLEN	Length of the data (in bytes or number of characters) passed to the CGI program. Set only with POST.
HTTP_FROM	_HTFROM	E−mail address of the user making the request (unreliable).

HTTP_ACCEPT	_HTACPT	Internet media (MIME) types that the client can accept. However, you may find using the HTTP_USER_AGENT variable more reliable than HTTP_ACCEPT.
HTTP_COOKIE	_HTCOOK	Cookies. See also the Set−Cookie header line.
HTTP_USER_AGENT	_HTUA	Browser name.
HTTP_REFERER	_HTREFER	If known, the URL of the document that the client points to before accessing the CGI program.

The Web server makes essential information available to CGI programs as environment variables. You can pass some or all of this information on to your Dispatcher programs by using the Export directive. The syntax is

```
Export <environment-variable> <SAS variable name>
```

The Export directive instructs the Application Broker to retrieve the contents of the specified environment variable and make it available to Dispatcher programs in the specified SAS macro variable or SCL list item.

The sample configuration file includes several Export directives. You can activate a directive by changing the information to match your site and removing the # that appears at the left of the export line.

Some Export directives are activated by default. Export REMOTE_HOST _RMTHOST is one. These directives are not preceded by a # in the default configuration file.

If you omit the SAS name, the name of the environment variable will be used as the SAS macro name.

If the value of the environment variable is greater than your field width (as set in _FLDWDTH), then the variable divides like any field into multiple variables. You can avoid this by using SAS variable names with a leading underscore, such as _RMTHOST. Dispatcher variables that begin with an underscore are not divided according to _FLDWDTH. These variables are truncated at 32767 characters.

Configuration File Directives

The required directives are listed below. For usage tips, see Using the Application Broker Configuration File.

Administrator Directives

Administrator name
ServiceAdmin name
: specifies the name of the person who is the administrator of the entire system or service. The *name* is passed to the Dispatcher program in the _ADMIN variable and is used in error messages.

AdministratorMail e−mail
ServiceAdminMail e−mail
: specifies the fully qualified e−mail address of the system or service administrator. The *e−mail* value is passed to the Dispatcher program in the _ADMAIL variable and is used in error messages.

BrokerPassword string
: specifies a password to protect the administration interface. If the BrokerPassword directive is specified, you must supply the password to access the Application Broker Admin page (debug=4).

ConnectionError "string"
ServiceConnectionError "string"
: specifies the message to be displayed when there is an Application Server connection error. The directive can be specified in the configuration file on a global or service level. The message is not displayed for connection errors in socket service administration programs such as ping and status.

Debugging Directives

Debug flags
ServiceDebug flags
: specifies flags for debugging and output management. This directive can be overridden with the _DEBUG field, which can be specified with a value or a keyword. The default is 2, or TIME, (indicates to display the status line and the Powered by SAS logo). See also List of Valid Debug Values.

DebugMask flags
ServiceDebugMask flags
: specifies the debug values that users are allowed to set. The default value for the DebugMask is 32767, which indicates that all debug values are allowed. If any debug values represent a security risk, you can selectively disable them by specifying a different DebugMask value, and then allow them only on certain services or for troubleshooting. See also List of Valid Debug Values.

File and Variable Manipulation Directives

Allow method−1 ...
: lists the allowable values for the request method. This directive does not actually set the method. The method names are GET and POST. See also HTML Syntax Reference.

AppendFile filepath
ServiceAppendFile filepath
: specifies a file that is added to the bottom of every HTML page that is generated by your application. The file will also be added to requests that generate errors in the Application Server, but will not be added when errors are generated by the Application Broker. Note that this is a host pathname, not a URL. If you use this feature, your applications might not output the </BODY> and </HTML> tags, but most browsers allow this.

Export env−var sas−var
ServiceExport env−var sas−var

specifies environment variables to be made available to Dispatcher programs. The *sas−var* is optional; if omitted, the SAS variable name is the same as the environment variable name (as long as it is a valid SAS name). Variables that do not begin with an underscore are subject to long−value splitting according to the field width. See also Exporting Environment Variables.

Language code

specifies the language used for error messages. The *code* is a two−letter language code. Currently only EN and FR are valid. The default is English.

PrependFile filepath

ServicePrependFile filepath

specifies a file that is added at the top of every HTML page that is generated by your application. The file will also be added to requests that generate errors in the Application Server, but will not be added when errors are generated by the Application Broker. Note that this is a host pathname, not a URL. If you use this feature, your applications might not output the </BODY> and </HTML> tags, but most browsers allow this.

Set variable value

ServiceSet variable value

specifies a variable to define on every request. This is similar to Export, but no environment variable is needed. This enables you to avoid hard−coding values such as the location of htmSQL in your applications. Values that do not begin with an underscore are subject to long−value splitting according to the field width.

General Service Directives

DefaultService

specifies the default service to use when no service name is supplied. DefaultService is the default.

Encrypt algorithm lib−path

ServiceEncrypt algorithm lib−path

specifies the configuration file line that enables encryption. When this line is included for a service and the SAS/SECURE product has been installed, all data sent between the Application Broker and the Application Server will be encrypted by using the specified algorithm.

algorithm

is one of the values SASPROPRIETARY, RC4, RC2, DES, or TRIPLEDES. A special keyword NONE may be entered to disable encryption for a particular service.

lib−path

is the path to where the SAS libraries TCPDENCR.DLL, TCPDEAM.DLL, and TCPDCAPI.DLL (Windows) and TCPENCR and TCPDRSA/TCPDRSAI (all other platforms) reside, for example,

```
"C:\\Program Files\\SAS\\SAS 9.1\\core\\sasexe".
```

Platform Notes

z/OS

- *lib−path* is not used, but the path *must* be specified in an environment variable named STEPLIB.

Windows

- You must install Microsoft Enhanced Cryptographic Provider Version 1.0 or later in order to use DES or TRIPLEDES.
- The *lib−path* value is optional if SAS/SECURE is installed. The software automatically obtains the library location from the registry value:

```
HKEY_LOCAL_MACHINE\SOFTWARE\SAS Institute Inc.\The SAS
System\9.1\Setup\Globals.
```

LoadManager host:port
ServiceLoadManager host:port
: defines the host and port number for the Application Load Manager. The Application Broker attempts to connect to this host and port to request an available Application Server from the Load Manager. You can supply the DNS name (for example, APPSRV.YOURCOMP.COM) or IP address (for example, 127.0.0.1) of the machine for *host*. You can supply a numeric port value or a symbolic name that is defined in the system services file for *port*.

LocalAddress address
: overrides the automatic determination of the local host IP address. Only specify this directive in special cases where the Application Server cannot connect back to the Application Broker host.

ServiceCompatibility version
: specifies the Application Server version number, if not the current version. This directive is useful for transitioning between incompatible releases. It is not needed if the Application Broker and the server releases match. For Version 6 and Version 7 of SAS software, set this value to 1.0.

ServiceDescription description
: provides a long description for the service.

Set variable value
ServiceSet variable value
: specifies a variable to define on every request. This is similar to Export, but no environment variable is needed. This enables you to avoid hard−coding values such as the location of htmSQL in your applications. Values that do not begin with an underscore are subject to long−value splitting according to the field width.

Timeout seconds
ServiceTimeout seconds
: specifies the number of seconds that the Application Broker waits for a response from the Application Server. When the specified time elapses, the Application Broker returns an error message to the browser. If no global timeout is specified, then the timeout default is 60 seconds.

For z/OS Only

ServerEncoding encoding
ServiceServerEncoding encoding
: defines the encoding used for data sent from the Application Broker to the Application Server and returned from the Application Server to the Application Broker. This option is not necessary unless the Web server uses a different encoding from the one used by the Application Server. The default ServerEncoding is automatically set based on the Web server encoding. The server encoding must match the Application Server output encoding. The Application Server output encoding is normally determined by the locale setting of your SAS installation, but may be set directly using the PROC APPSRV ENCODING option.

  Use one of the following values for *encoding*:

  - `wlatin1` (Western Europe): This value is the default in all cases except for when the Web server encoding is IBM−870 or IBM−1025.
  - `wlatin2` (Eastern Europe): This is the default encoding when the Web server is using IBM−870 encoding.
  - `wcyrillic` (Cyrillic): This is the default encoding when the Web server is using IBM−1025 encoding.
  - `ISO-8859-1` (Latin1)
  - `ISO-8859-2` (Eastern Europe)
  - `ISO-8859-5` (Cyrillic)
  - `ISO-8859-15` (Latin9)

Note that the body of the response from the Application Server (whether in HTML or another text format) defaults to the specified encoding but might be changed by the request program. For example, your request program might choose to generate a ISO−8859−1 response even if the ServerEncoding directive specifies `wlatin1`.

Service−Specific Directives

LaunchService

LaunchService name desc
begins a service definition and accepts two values: a name and an optional short description for the service. The name is used as the value for the _SERVICE field that is passed to the Application Broker from the HTML information in the browser. The *name* value is required.

Note: A launch on Windows NT systems does not work if you do not have the TEMP system variable set or if you do not specify `−work` on the SAS command line. Within the SAS configuration file, WORK is defined as

```
/* Setup the default SAS System user the work folder */
−WORK "!TEMP\SAS Temporary Files"
```

Because the Web server uses only system variables, if TEMP is not defined as a system variable, then WORK is not found and SAS does not start.

SasCommand command
specifies the SAS command and arguments that are necessary to invoke a new SAS session. It is usually the fully qualified path to your SAS executable file or a shell script. The argument SYSPARM must be included with the command. It must be specified at the end of the *command* as shown in the template configuration file delivered with SAS/IntrNet software. When you specify SasCommand on a Windows system, you must include the .exe extension for the SAS executable file.

LaunchService Directives for Previous Version Servers

InitCmd
(Version 6 servers only)
specifies the SAS statement necessary to invoke the Application Server. Do not include the PORT= argument, which is valid only with the SocketService.

InitStmt
(Version 7 servers only)
specifies the SAS statement necessary to invoke the Application Server. Do not include the PORT= argument, which is valid only with the SocketService.

SasBin command
(Version 7 and Version 6 servers only)
specifies the SAS command necessary to invoke a new SAS session. It is usually the fully qualified path to your SAS executable file. When you specify SasBin on a Windows system, you must include the .exe extension for the SAS executable file.

SasOpts options
(Version 7 and Version 6 servers only)
specifies the SAS command line options that are used to invoke a SAS session. You must include a −SYSIN file as one of your SAS options or the server will not start. This file *must* exist, but it is empty because the real input to the server session is supplied by the InitStmt directive.

TmpDir directory
(in Version 7 and Version 6 LaunchServices only)

specifies a directory (that must end with a slash) on the Web server machine (with read and write permissions allowed) where the application writes temporary files. All temporary files and directories, including the SASUSER and WORK libraries, log files, and other files used by the Application Broker and the server, are created in TmpDir. The *directory* value is passed to the Dispatcher application in the _TMPDIR variable.

PoolService

Note: You must specify a Load Manager before you use PoolService. Also, if the specified server is not on the same machine as the Load Manager, you must specify a spawner port to use.

FullDuplex True
: indicates that the Application Broker and Application Server use only one socket for communication. Use this only with servers Release 8.1 or later because it will cause previous releases to hang.

IdleTimeout minutes
: specifies the optional pool server timeout (in minutes). The default is 60 minutes. A value of 0 indicates immediate shutdown after processing the job. A server does not shut down until all sessions have expired.

MinRun value
: specifies the minimum number of servers to keep running. This directive is optional.

Password string
: specifies the optional password used with the Username directive to start a new server. If _PASSWORD is specified, the password is taken from the client _PASSWORD field. A password that starts with an exclamation point (!) character is assumed to be encrypted. This directive is valid only if you have specified a spawner port.

PoolService name desc
: begins a service definition and accepts two values: a name and an optional short description for the service. The name specified for the service is used as the value for the _SERVICE field that is passed to the Application Broker from the HTML information in the browser.

Port port1 port1−port3
: specifies the TCP/IP port number(s) or network service name(s) used by the Application Broker to send requests to the Application Servers. You can define multiple ports by separating their values with spaces or by issuing the Port directive multiple times. Numeric port ranges and symbolic names that are defined in the system services file are supported. A number less than 256 indicates a count of the maximum number of servers to start.

SasCommand command
: specifies the SAS command and arguments that are necessary to invoke a new SAS session. It is usually the fully qualified path to your SAS executable file or a shell script. The argument SYSPARM must be included with the command. It must be specified at the end of the *command* as shown in the template configuration file delivered with SAS/IntrNet software. When you specify SasCommand on a Windows system, you must include the .exe extension for the SAS executable file. On UNIX systems, it is recommended that you use the −LOG /DEV/NULL option.

Server host−1 host−2 ...
: specifies the names of the physical machines on which the Application Servers are installed. You can supply the DNS name (for example, APPSRV.YOURCOMP.COM) or IP address (for example, 127.0.0.1) of the machine. This directive is required with the pool service. You can supply a value of LOCALHOST instead of a fully qualified DNS name if the Application Server is running on the same machine as the Web server.

SpawnerPort port
: specifies the port on which the SAS Spawner is listening. The SAS Spawner is used to start new Application servers for this service. This directive is optional.

  Note: Some of the SAS Spawner features cannot be used with pool services. For example, because the Load Manager does not support data encryption, the SAS Spawner cannot be started with `-netencrypt` or `-netencralg`.

StartAhead value

indicates how many SAS servers to start ahead of time when all current servers are busy. The default is 0. This directive is optional.

Username string

specifies an optional username used with the Password directive to start a new server. If _USERNAME is specified, the username is taken from the client _USERNAME field. A username starting with a ! character is assumed to be encrypted. This option is valid only if you have specified a spawner port.

SocketService

FullDuplex True

indicates that the Application Broker and Application Server use only one socket for communication. Use this only with servers Release 8.1 or later because it will cause previous releases to hang.

Port port1 port1−port3 ...

specifies the TCP/IP port number(s) or network service name(s) used by the Application Servers for this service. You can define multiple ports by separating their values with spaces or by issuing the Port directive multiple times. Numeric port ranges and symbolic names defined in the system services file are supported.

Note: For Pool Services, a number less than 256 indicates a count of the maximum number of servers to start.

Server host−1 host−2 ...

specifies the names of the physical machines on which the Application Servers are installed. You can supply the DNS name (for example, APPSRV.YOURCOMP.COM) or IP address (for example, 127.0.0.1) of the machine. This directive is required with the socket service. You can supply a value of LOCALHOST instead of a fully qualified DNS name if the Application Server is running on the same machine as the Web server. See also Enhancing Performance.

SocketService name desc

Begins a service definition and accepts two values: a name and an optional short description for the service. The name specified for the service is used as the value for the _SERVICE field passed to the Application Broker from the HTML information in the browser. The *name* value is required.

URL Directives

SASPoweredAlt text

ServiceSASPoweredAlt text

specifies the alternate text used for the Powered by SAS logo image. This text appears while the image is loading, or if images are disabled or not supported, or in some browsers, when the mouse is held motionless over the image. The default is "SAS Institute Inc."

SASPoweredHref URL

Service SASPoweredHref URL

specifies the destination URL, that is, where you go when you click the image. The default is HTTP://WWW.SAS.COM.

SASPoweredLogo URL

ServiceSASPoweredLogo URL

specifies the location of the Powered by SAS logo image file. See also Displaying the Powered by SAS Logo.

SASPoweredTarget frame

ServiceSASPoweredTarget frame

specifies the frame that is used for the hypertext link on the Powered by SAS logo. The default is no target. Any browser−supported target might be used, such as _TOP, which indicates to take over the whole browser window, _BLANK, which indicates to open a new window, _PARENT, which indicates to use the parent of the current frame, and _SELF, which indicates to use the current frame.

SelfURL URL

specifies the self−referencing URL that identifies the Application Broker program. The default value is the SCRIPT_NAME environment variable set by the Web server. The URL is passed to the SAS program in a macro variable named _URL.

Note: Normally you do not need to set SelfURL. SelfURL may be useful in the following situations:

- ◊ if your Web server does not set the SCRIPT_NAME environment variable or sets it incorrectly.
- ◊ if your site uses DNS load balancing with its Web servers. SelfURL can be used to specify the load−balanced Web server name instead of the particular Web server executing the Application Broker. For example, assume your company Web address HTTP://WWW.COMPANY.COM uses DNS to refer browser requests to one of five servers (WWW1.COMPANY.COM through WWW5.COMPANY.COM). An Application Broker running on WWW2.COMPANY.COM might have a default SCRIPT_NAME value of HTTP://WWW2.COMPANY.COM/SCRIPTS/BROKER.EXE. The SelfURL directive could be used to specify the load−balanced address HTTP://WWW.COMPANY.COM/SCRIPTS/BROKER.EXE instead.

Running Multiple Application Servers at Your Site

Running multiple Application Servers raises the same issues as running multiple SAS sessions. There is little to be concerned with if you set up each server on a different file system. However, quite often, multiple servers could be set up that access the same file system. This raises the general issue of file contention between the Application Servers.

Here are some guidelines to follow if you set up multiple Application Servers on the same file system:

- Servers can share a server root directory.
- Servers can share program libraries. Be sure to use `access=readonly` when allocating shared program libraries. Allocating shared program libraries in this way prevents an Application Server from opening SAS catalogs in update mode.
- Servers on the same machine must use different port numbers.
- Update access to data sets should be done through the use of SAS/SHARE software. If you do not use SAS/SHARE software and two users attempt to update the same SAS data set at the same time, a failure will result.
- Because update access to files can fail due to operating environment contention, Dispatcher programs should be prepared to handle such failures.
- Each server should write to a different log file.

Application Server Administration Programs

The Application Server has a few built−in programs that instruct the server to perform special administrative tasks. Starting with Release 8.2 of Application Dispatcher, you can use an administration interface to perform these tasks. To access the interface, enter the Application Broker URL in your browser. The Application Broker URL depends on your Web server platform and the Application Broker location. Typical URLs might be

```
http://yourserver/cgi-bin/broker           (UNIX or z/OS)
http://yourserver/scripts/broker.exe       (Windows)
```

When you access the Application Broker URL using no parameters, a welcome page appears in your browser. This page gives you access to an administration interface, SAS/IntrNet samples, and SAS/IntrNet documentation. Clicking on the Application Dispatcher Administration link will display information about all defined Application Dispatcher services. Links are provided to administrative and status programs for each service. The available programs are described below. The administration interface can be password protected using the BrokerPassword directive in the Application Broker configuration file. To disable the administration interface, use the DebugMask directive to disable _DEBUG=4.

Note: If there is a customized Application Broker welcome page, then it will display instead of this default welcome page when you enter the Application Broker URL in your browser. If this is the case, and if you want to view the services that are available from the default welcome page, then add _DEBUG=4 to the URL, as follows:

```
http://yourserver/cgi-bin/broker.exe?_debug=4
```

The following administrative programs are available for Application Dispatcher services.

Command	Description
STOP	Stops the Application Server. All currently active requests are allowed to complete. An optional _WAIT parameter can be used to specify a maximum wait time (in seconds) for any sessions to expire. An Application Server that has received a STOP command with a _WAIT parameter will accept requests that access existing sessions, but will not accept other new requests. The default value for _WAIT is zero. BREAK and ENDSAS commands are also supported for compatibility with earlier releases, although the STOP command is recommended.
PING	Executes a simple program that verifies that the Application Server is working correctly.
STATUS	Displays a status page for the server that contains useful information, such as when the server was started and how many jobs it has processed.

In addition to using the administration interface, you can execute an Application Server administration program by supplying the name of the program in an Application Dispatcher request. For example,

```
http://yourserver/cgi-bin/broker?_service=default&_program=ping
```

These special program names are not case sensitive. If you start the Application Server by using a password, you must supply the password to execute the STOP program. For example,

```
http://yourserver/cgi-bin/broker?_service=default&_program=stop&_adminpw=foo
```

Note: If you do not start the Application Server by using a password, any client can run the STOP program and shut down your server.

Application Server Libraries

Program Libraries

Program libraries are directories, partitioned data sets, or SAS libraries that contain Dispatcher programs. Each Dispatcher program must be placed in a program library before the Application Server can run it. One Application Server can access multiple program libraries. They are defined by the ALLOC file, ALLOC library, and PROGLIB statements in PROC APPSRV. These libraries are segregated from data libraries for security reasons.

Data Libraries

Data created or used by Dispatcher programs should not be stored in program libraries. It is a security risk to store programs in the same location as their data. For a more complete discussion on security precautions, see Restricting Access to Program Libraries. The recommended method for accessing data from a Dispatcher program is to issue a LIBNAME or FILENAME statement in the Dispatcher program code. After the program has completed, the Application Server clears any libnames or filenames that the program has left assigned.

Using Services

An Application Dispatcher service is a collection of one or more Application Servers. The servers might be running on one system or could be distributed across multiple systems. All of the servers in a specified service are assumed to have access to the same applications and data so that a particular client request can be fulfilled by any server within the service. All Application Dispatcher requests include a service name (the _SERVICE parameter) that determines which service will perform the request. The Application Broker is responsible for selecting a particular server belonging to that service and forwarding the request to that server.

During the SAS/IntrNet installation process, you will have created a default service. The default service is a socket service with only one server, the simplest type of service to set up and use. This service is adequate for running the sample programs delivered with Application Dispatcher and beginning to develop your own applications, but there are many reasons that you might want to create additional services. You might want to create separate services for different applications so that you can distribute resources (such as memory, disk space, or priority) among applications. You can create different services with different levels of access as a simple form of security. Often, you will need to create separate services for development and production application environments, so that development activities do not affect production applications.

The following pages address some of these issues and describe the process of creating and maintaining services:

- Choosing a Service Type
- Creating and Modifying Services
    - ♦ on OpenVMS
    - ♦ on z/OS
    - ♦ on UNIX platforms
    - ♦ on Windows platforms.
- Enhancing Performance
- Development vs. Production Environments.

Choosing a Service Type

A service can be a socket, pool, or launch service. The features, advantages, and disadvantages of each of these service types is discussed below.

Socket Services

Socket services consist of one or more Application Servers that run continuously, servicing client requests. Socket services are generally started whenever a machine is restarted (either manually or by an operating environment mechanism for starting processes at boot or login time). The service usually runs until the machine is shut down. Socket services are relatively simple to configure and manage and are adequate for most applications.

Advantages

- Socket services are supported on all SAS/IntrNet platforms. Other service types are not supported everywhere.
- The server is already running by the time a client request appears, so clients do not have to wait for a server to start.
- The administrator has explicit control of resources allocated to the service: the administrator can control how many servers are run on each system and which resources are allocated to each server.
- Increasing load can be handled by adding more servers to the service.

Disadvantages

- Servers must be started and stopped manually or by the operating system. No automated start−up and shutdown service is provided by SAS/IntrNet software.
- No dynamic scaling to meet increasing loads is provided. A fixed number of servers are available to handle all client requests. A few long−running requests can slow the entire service for all clients.

Pool Services

Pool services consist of a pool of Application Servers shared by clients. Based on system loading the servers are started and stopped by the Application Load Manager. Numerous options are provided to fine−tune the operation of a pool service. Pool services combine some of the advantages of socket and launch services.

Advantages

- Servers are started as needed. If all servers in the service are busy, the Load Manager can start an additional server.
- Servers can be reused by new clients once they are started. A started server remains in the pool until an idle timeout is reached and the server is stopped.
- Unlike launch services, pool services can be on a different system than the Web server and can be distributed across multiple server systems.
- Using the SAS Spawner, servers can be started under specific usernames to control access to system resources.

Disadvantages

- Installation and configuration are more complex for pool services. The Application Load Manager must be installed. The SAS Spawner must be installed in some cases.

- Client requests might have to wait for a new server to start, although this is typically no worse (and could be better) than waiting for currently executing requests to complete in a socket service.

Launch Services

A launch service starts a new Application Server for each client request. An existing server is reused only for applications that use sessions or the _TMPCAT catalog for IDS output. Most of the features of launch services are better provided by pool services. Launch services are not generally recommended for new installations.

Advantages

- Server start−up is automatic for each request. Once the launch service is configured, little or no additional administration is necessary.
- Requests run in a separate server, so a long−running request will not block access to the service for other clients.
- Many requests can run in parallel, assuming that the system will support the load.
- Ill−behaved applications that "crash" or "hang" a server will not affect other client requests.

Disadvantages

- Launch services are started by the Application Broker and must run on the same system as the Web server.
- Each new request incurs the resource overhead and delay of starting a new server session.
- Launch services are not suitable for high user loads. There are no settable limits on the server load. The service will attempt to start a new server for each new client. In an extreme case, 200 simultaneous users could cause 200 servers to be started, likely causing extreme "memory thrashing" and very slow response for all users. Most Web servers have limits on the number of simultaneous CGI requests that could help to control this problem.
- Each launch service request must incur the additional time for starting a SAS session.
- Launch services are not supported on OpenVMS and z/OS platforms.
- Launched servers can be difficult to shut down. A launched server that creates a session or _TMPCAT catalog will continue running until an idle timeout is reached. These servers cannot be shut down other than by interrupting the server process.

Services on OpenVMS

Creating a Service

To create a service on OpenVMS:

1. From a system prompt, submit `@sas$root:[tools]inetcfg.com`.

   Note: Square brackets in syntax indicates that an attribute is optional; do not include the square brackets in your code.

   As the configuration utility runs, you are prompted for information about the service that you are creating.
2. At the first prompt, type the name of the service. Press **Return** to accept the default value, which names the service *default*. To create a service other than default, type the service name and then press **Return**.
3. The next prompt asks for the name of the directory where all of the service control files should be stored. Press **Return** to accept the suggested value, or type the desired directory name and then press **Return**.
4. Specify the type of service you are defining. Type **S** for a socket service or **P** for a pool service. Press **Return** to continue.
5. If you choose a socket service, specify the number of servers that you want to include in this service. The number you specify here represents only those servers running on this physical machine.

   If there is only one server, press **Return**. If you plan to have multiple servers, type the number and then press **Return**.
6. If you have selected a socket service, you are prompted to enter a TCP/IP port number or name for each of the servers that you requested as the answer to the previous prompt. Type the value and press **Return**.
7. You are asked whether you want to protect this service with an administrator password. Answer **Y** or **N**, and press **Return**. Type a password and press **Return**.
8. The utility displays the information that you entered for this service. Verify that the information is correct.

   If the information is correct, press **Return**. The utility creates the service. Read the messages printed by the utility to determine if the service was created correctly. One of these messages contains the path for the service root directory created by the utility. Note this path for use later in this process.

   If you want to change the information, type **N** and press **Return**. The utility exits and you can start over.
9. If you are creating a pool service, you must install the SAS Spawner. Refer to the SAS/CONNECT documentation for installation instructions for the SAS Spawner.
10. The Application Broker must know about this service for you to access it. Open the Broker configuration file on your Web server in a text editor and add a service definition block.

    The definition block for a socket service might look like `# This service contains one server (port 5801) on yourserv.yyy.com. SocketService service-name "brief-text-desc" ServiceDescription "text-desc" ServiceAdmin "administrator-name" ServiceAdminMail "administrator-email-address@host" Server yourserver Port 5801`

    A pool service definition might look like `# Start up to 5 servers on node yourserv.yyy.com using the spawner started # at port 7777. All servers will be started with the specified username/ # password. At least 1 server will not timeout and be kept running. PoolService service-name "brief-text-desc" ServiceDescription "text-desc"`

```
ServiceAdmin "administrator-name"
ServiceAdminMail "administrator-email-address@host"
ServiceLoadManager load-manager-host:port
SasCommand "sas9 disk:[username.INTRNET.service-name]APPSTART.SAS+
/rsasuser /noterminal /noprint /nolog /SYSPARM "
Server yourserver
Port 6000-6004
Username your-username
Password your-password
SpawnerPort 7777
MinRun 1
```

Starting the Service

Socket services must be started with the START.COM created by the configuration utility. To start the service, change to the service root directory created by the utility. Then submit the command

```
@START.COM
```

Pool services are started automatically by the Application Load Manager. See Using the Load Manager for more details.

Once a service is started, you can test it from a Web browser. The URL depends on the platform and path where your Application Broker is installed. For typical installations, the URL to test (or "ping") a service is one of the following:

Windows:

```
http://yourserver/scripts/broker.exe?_service=service-name&_program=ping
```

UNIX and z/OS:

```
http://yourserver/cgi-bin/broker?_service=service-name&_program=ping
```

Specify your Web server name in place of *yourserver* and your service name in place of *service−name*. You might need to use a different URL path if you chose a different path when you installed the Application Broker. If the service is running, an HTML page will be returned stating that the Application Server is functioning.

Stopping the Service

Services can be stopped from a Web browser. The URL depends on the platform and path where your Application Broker is installed. For typical installations, the URL to stop a service is one of the following:

Windows:

```
http://yourserver/scripts/broker.exe?_service=service-name&_program=stop
```

UNIX and z/OS:

```
http://yourserver/cgi-bin/broker?_service=service-name&_program=stop
```

Specify your Web server name in place of *yourserver* and your service name in place of *service−name*. You might need to use a different URL path if you chose a different path when you installed the Application Broker.

Service Log Files

Log files are placed in the LOGS directory under the service root directory and are named <*day*>_<*port*>.LOG, for example, MON_5001.LOG;1 or TUE_5001.LOG;1. By default, logs are kept for one week (six full days and one

partial day) and then overwritten.

SAS 9 has implemented a set of % codes that can be used in the –log parameter. You must add a –logparm option in order to get the codes translated into the log. For example adding

```
-log 'disk:[service-root.service-name.LOGS]appsrv_%v.log'
-logparm rollover=auto
```

creates log files with unique log filenames. The rollover=auto option causes an automatic "rollover" of the log when the directives in the value of the LOG option change. This is particularly useful for generating log files for pool services. This example creates log filenames such as APPSRV_1.LOG;1, APPSRV_2.LOG;1, and APPSRV_3.LOG;1.

Note: You must use a full path to specify the log file because there is limited control over what path the Load Manager or spawner will use as the current directory for each Application Server.

For more information, see the documentation on the LOGPARM= system option in SAS Language Elements.

Removing a Service

You can remove a service by deleting the service root directory and its contents. Any active servers must be stopped before you delete this directory.

Services on z/OS

Creating a Service

The configuration utility provided for z/OS is a batch job. It is installed as a member named INETCFG in the CNTL data set that you created as the first step in the installation of your SAS software. To use the utility, you must edit the parameter file, member INETEDTP in the CNTL data set, edit the INETCFG job, and then submit the INETCFG job. The INETEDTP member contains the parameters necessary for creating a service. You can read the comments provided and change the default values to the values required for your service.

To create a service under z/OS:

1. Edit the member INETEDTP.
2. Specify the name of the service you are creating. The service name can be a maximum of eight characters. Locate the line containing ISVC= and replace the default value with the name of the service you are creating.
3. Specify the type of service that you are defining. Uncomment the appropriate line containing ISVCTYP=:

   - If you want a socket service, uncomment %SOCKETTYP.
   - If you want a pool service, uncomment %POOLTYP.

   Make sure that the line containing the other service type is commented out by placing an asterisk in the first column.
4. For socket services, specify the TCP/IP port number or network service name for each server in the service. You must specify at least one port, but you can specify up to ten. Port numbers or names are not used for pool services.

   To specify the TCP/IP port, locate the line containing I$−PORT1. Change the value `5001` to the correct port number or network service name for the first server in your service. If you want to use more than one server for this service, remove *NO* from the desired number of I$−PORT entries and change the value to the appropriate value for each server in the service.
5. If you are creating a pool service, you must install the Application Load Manager. You might also want to install the Load Manager if you have a socket service with more than one server. See Using the Load Manager for more information.

   If you want to use the Load Manager on your z/OS system, you must install the SAS/IntrNet CGI Tools for Web Server package. Verify the settings for the Load Manager in INETEDTP:

   - Choose a TCP port number or network service name for the Load Manager. Supply this value on the line containing I$−LDMPORT=.
   - Supply the entire UNIX System Services file path to the Load Manager executable on the line containing I$−LDMPROG=. The Load Manager is named LOADMGR and is installed in the directory corresponding to the URL `http://`*`yourserver`*`/sasweb/IntrNet9/tools/`.
   - Determine where the Load Manager should write its log file. Supply the entire UNIX System Services file path for the log file on the line containing I$−LDMSOUT.
6. Save and close INETEDTP.
7. Edit INETCFG to verify the job header information and the name of the service you are defining. The service name in this Job Control Language (JCL) should match the value you supplied for ISVC in INETEDTP. If you make changes, be sure to save them. Do NOT change SASEDITP to INETEDTP. This filename refers to your original SAS installation parameters file.
8. Submit the INETCFG JCL job for processing.The INETCFG job submits another job (INETCFGA). Verify that both jobs completed with a return code of 0. If they completed successfully, you now have the data sets and members necessary for running your service.

If the INETCFG job failed, examine the messages and sysprint members for error messages. If you see message that reads

```
ERROR: THIS REPLACEMENT CAUSES RESULT TO EXCEED OUTPUT LRECL
```

you might have supplied a pathname in one of your INETEDTP parameters that is too long. Try shortening this pathname and rerun INETCFG.

Note: Before you run INETCFG again, you must delete any data sets created by the previous failure of INETCFG. You can find these data sets by looking in the namespace determined by your original SAS install. For example, if your SAS software was installed with the prefix name SYS.SAS and your failed INETCFG was trying to create the `default` service, then delete all data sets beginning with the name prefix SYS.SAS.WEB.DEFAULT before running INETCFG again.

9. The configuration utility creates a server root in a partitioned data set (PDS) named *prefix*.WEB.*service−name*, where *prefix* is the data set prefix you supplied during your SAS installation and *service−name* is the name of the service that you just created. The PDS contains any JCL procedures and server start−up code required for starting the service. You will find these members:

   APSTRTn
   : contains the JCL necessary to run the corresponding @APSTX*n* file as a started task. These members exist only for socket services. You should move these files to your started task library and enable them as started tasks.

   @APSTXn
   : contains the SAS code that invokes the server. This file is called by the JCL in the corresponding APSTRT*n* file for socket services and by the Spawner for pool services. These SAS programs must remain in the PDS where they were created.

   LOADMGR
   : contains the JCL necessary to run the Load Manager. You should move this file to your started task library and enable it as a started task.

   In addition to the server root PDS, the configuration utility creates an empty PDS named *prefix*.WEB.*service−name*.TDIR, where *prefix* is the data set prefix that you supplied during your SAS installation and *service−name* is the name of the service you just created. All of the servers in the service use this PDS as a scratch location. Each server also has its own scratch SAS data library. These libraries are named TBLIB1 through TBLIB*n*.
10. You must modify the permissions for the data sets created above so that the server can write to them as necessary. To modify the permissions, create a special RACF data set profile that applies to all the data sets in this service (*`prefix`*`.WEB.`*`service-name`*`.*`). The RACF data set profile should also grant write access to the user ID of the Application Server.
11. If you are creating a pool service, you must install the SAS Spawner. Refer to the SAS/CONNECT documentation for installation instructions for the SAS Spawner.
12. The Application Broker must know about this service so that you can access it. Open the Broker configuration file on your Web server in an editor and add a service definition block. Example service definitions are found in the template configuration file installed with the Application Broker. Several examples are shown below. Values that may need to be changed for your site are shown in green.

    The definition block for a socket service might look like the following:

```
# This service contains one server (port 5801) on yourserv.yyy.com.
SocketService service-name
   ServiceDescription "text-desc"
   ServiceAdmin "administrator-name"
   ServiceAdminMail "administrator-email-address@host"
   Server yourserv.yyy.com
   Port 5801
```

A pool service is defined by the following:

```
#  Start up to 5 servers on node yourserv.yyy.com using the spawner started
#  at port 7777. All servers will be started with the specified username/
#  password. At least 1 server will not timeout and be kept running.
PoolService service-name
   ServiceDescription "text-desc"
   ServiceAdmin "administrator-name"
   ServiceAdminMail "administrator-email-address@host"
   ServiceLoadManager load-manager-host:port
#  Change prefix in command to the SAS install prefix name.
#  There is no closing single quote for the CLIST parameter string,
#  because the spawner adds parameters to the end of the string.  Spacing
#  shown in the example command is important.
   SasCommand  "/your/bin/spawnsas.sh NOSASUSER +
      O(''NOTERMINAL SYSIN=prefix.WEB.service(@APSTXn) SYSPARM '') "
   Server yourserv.yyy.com
   Port 6000-6004
   Username appdemo
   Password xyzzy
   SpawnerPort 7777
   MinRun 1
```

Note: The SasCommand line contains two double quotation marks (") and four single quotation marks ('). The double quotation marks are located at the beginning of the command (before /your/bin) and at the end of the entire command, after the parenthesis. Also note that the SAS Spawner must be installed and configured for scripted signons in order for this command to work.

Starting the Service

As stated above, the APSTRT*n* files for a socket service should be moved from your server root PDS to your started task library and enabled as started tasks. To start the service, issue a START command from the system console for each server in the service.

Pool services are started automatically by the Application Load Manager. If you installed the Load Manager on your z/OS system, the LOADMGR started task can be started by a START command from the system console. See Using the Load Manager for more details.

Once a service is started, you can test it from a Web browser. The URL will depend on the platform and path where your Application Broker is installed. For typical installs, the URL to test (or "ping") a service will be one of the following:

UNIX and z/OS:

`http://yourserver/cgi-bin/broker?_service=service-name&_program=ping`

Windows:

`http://yourserver/scripts/broker.exe?_service=service-name&_program=ping`

You must specify your Web server name in place of *yourserver* and your service name in place of *service−name*. You might need to use a different URL path if you chose a different path when you installed the Application Broker. If the service is running, an HTML page will be returned stating that the Application Server is functioning.

Stopping the Service

Socket or pool services can be stopped from a Web browser. The URL will depend on the platform and path where your Application Broker is installed. For typical installs, the URL to stop a service will be one of the following:

UNIX and z/OS:

```
http://yourserver/cgi-bin/broker?_service=service-name&_program=stop
```

Windows:

```
http://yourserver/scripts/broker.exe?_service=service-name&_program=stop
```

You must specify your Web server name in place of *yourserver* and your service name in place of *service−name*. You might need to use a different URL path if you chose a different path when you installed the Application Broker.

Service Log Files

Log files for socket services are saved as JES spool files. Log files for pool services are named *prefix*`.WEB.`*service-name*`.mmddyy.`*port-no*`.LOG`, where *prefix* is the data set prefix that you supplied during your SAS installation, `mmddyy` is the current date (represented as a six digit number), and *port-no* is the TCP/IP port number of the server. Log files are not automatically deleted. You must manually delete them to recover the disk space.

Removing a Service

A service can be removed by deleting all data sets beginning with the name *prefix*`.WEB.`*service-name*, where *prefix* is the data set prefix that you supplied during your SAS installation. For example, if you want to remove the `SVC2` service and your SAS software was installed with the prefix name SYS.SAS, then delete all data sets beginning with the name prefix SYS.SAS.WEB.SVC2.

Services on UNIX Platforms

Creating a Service

To create a service for an Application Server running in a UNIX environment, perform the following steps:

1. From a system prompt, submit the following command:

   ```
   SASROOT/utilities/bin/inetcfg.pl
   ```

   where *SASROOT* is the path to the SAS root directory.

   As the configuration utility runs, you are prompted for information about the service that you are creating.
2. For the first prompt, type the name of the service. Press **Return** to accept the default value, which names the service *default*. To create a service other than default, type the service name, and then press **Return**.
3. The next prompt asks for the name of the directory where all of the service control files should be stored. Press **Return** to accept the suggested value, or type the desired directory name and then press **Return**.
4. Specify the type of service you are defining. Type **S** if you are defining a socket service, **L** for a launch service, or **P** for a pool service. Press **Return** to continue.
5. If you choose a socket service, specify the number of servers that you want to include in this service. Note that the number you specify here represents only those servers running on this physical machine.

   If there is only one server, press **Return**. If you plan to have multiple servers, type the number, and then press **Return**.
6. If you select a socket service, you are prompted to enter a TCP/IP port number or name for each of the servers that you requested as the answer to the previous prompt. Type the value and press **Return**.
7. If you choose a socket or a pool service, the script will ask if you want to protect this service with an administrator password. Answer **Y** or **N** and press **Return**. Supply a password and press **Return**.
8. Next, the utility displays the information that you entered for this service. Verify that the information is correct.

   If the information is correct, press **Return**. The utility creates the service. Read the messages printed by the utility to determine if the service was created correctly. One of these messages contains the path for the service directory created by the utility. You should note this path for use later in this process.

   If you want to change the information, type **N** and press **Return**. The script exits and you can rerun the script.
9. If you are creating a pool service, you must install the Application Load Manager. You might also want to install the Load Manager if you have a socket service with more than one server. See Using the Load Manager for more information.
10. If you are creating a pool service, you might need to install the SAS Spawner. The Spawner is not required if you configure the Application Load Manager to start the service directly. If you choose this method, the servers must all execute on the same system as the Load Manager and under the same user ID. The Spawner is not required if you choose to use the UNIX telnet daemon to start the servers. Refer to the SAS/CONNECT documentation for installation instructions for the SAS Spawner.
11. The Application Broker must know about this service so that you can access it. Open the Broker configuration file on your Web server in an editor and add a service definition block. Example service definitions are found in the template configuration file installed with the Application Broker. Several examples are shown below. Values that may need to be changed for your site are shown in green.
    - The definition block for a socket service might look like

```
# This service contains one server (port 5801) on yourserv.yyy.com.
SocketService service-name "brief-text-desc"
ServiceDescription "text-desc"
ServiceAdmin "administrator-name"
ServiceAdminMail "administrator-email-address@host"
Server yourserv.yyy.com
Port 5801
```

♦ Launch services are defined with

```
LaunchService service-name "brief-text-desc"
ServiceDescription "text-desc"
ServiceAdmin "administrator-name"
ServiceAdminMail "administrator-email-address@host"
SasCommand "/usr/local/bin/sas+
/usr/local/intrnet/service-name/appstart.sas+
-rsasuser -noterminal -noprint -nolog -SYSPARM"
```

♦ A pool service that is started directly by the Load Manager is defined by

```
# Start up to 2 servers on the same node as the Load Manager. No
# username/password or spawner is needed for this case. Servers will
# time out after 30 minutes.
PoolService service-name
ServiceDescription "text-desc"
ServiceAdmin "administrator-name"
ServiceAdminMail "administrator-email-address@host"
ServiceLoadManager yourserv.xxx.com:5555
SasCommand "/usr/local/bin/sas+
/usr/local/intrnet/service-name/appstart.sas+
-rsasuser -noterminal -noprint -nolog -SYSPARM"
IdleTimeout 30
Server yourserv.xxx.com
Port 2
```

♦ A pool service that is started by the SAS Spawner is defined by

```
# Start up to 5 servers on node yourserv.yyy.com using the spawner
# started at port 7777. All servers will be started with the
# specified username/password. At least 1 server will not timeout
# and be kept running.
PoolService service-name
ServiceDescription "text-desc"
ServiceAdmin "administrator-name"
ServiceAdminMail "administrator-email-address@host"
ServiceLoadManager load-manager-host:port
SasCommand "/usr/local/bin/sas+
/usr/local/intrnet/service-name/appstart.sas+
-rsasuser -noterminal -noprint -nolog -SYSPARM"
Server yourserv.yyy.com
Port 6000-6004
Username appdemo
Password xyzzy
SpawnerPort 7777
```

```
MinRun 1
```

Starting the Service

Socket services must be started with the start.pl script created by the configuration utility. To start the service, change to the service root directory created by the utility. Then submit the following:

```
./start.pl
```

Launch services are started automatically by the Application Broker. Pool services are started automatically by the Application Load Manager. See Using the Load Manager for more details.

Once a service is started, you can test it from a Web browser. The URL will depend on the platform and path where your Application Broker is installed. For typical installs, the URL to test (or "ping") a service will be one of the following:

UNIX and z/OS:

`http://`*`yourserver`*`/cgi-bin/broker?_service=`*`service-name`*`&_program=ping`

Windows:

`http://`*`yourserver`*`/scripts/broker.exe?_service=`*`service-name`*`&_program=ping`

You must specify your Web server name in place of *yourserver* and your service name in place of *service−name*. You might need to use a different URL path if you chose a different path when you installed the Application Broker. If the service is running, an HTML page will be returned stating that the Application Server is functioning.

Stopping the Service

Socket or pool services can be stopped from a Web browser. The URL will depend on the platform and path where your Application Broker is installed. For typical installs, the URL to stop a service will be one of the following:

UNIX and z/OS:

`http://`*`yourserver`*`/cgi-bin/broker?_service=`*`service-name`*`&_program=stop`

Windows:

`http://`*`yourserver`*`/scripts/broker.exe?_service=`*`service-name`*`&_program=stop`

You must specify your Web server name in place of *yourserver* and your service name in place of *service−name*. You might need to use a different URL path if you chose a different path when you installed the Application Broker.

Service Log Files

Log files are placed in the `logs` directory under the service root directory and are named `<`*`day`*`>_<`*`port`*`>.log`. For example, Mon_5001.log or Tue_5001.log. By default, logs are kept for one week (six full days and one partial day) and then overwritten.

SAS 9 has implemented a set of % codes that can be used in the −log parameter. You must add a −logparm option in order to get the codes translated into the log. For example adding

```
-log '/full-service-root-path/service-name/logs/appsrv_%v.log'
-logparm rollover=auto
```

creates log files with unique log filenames. The rollover=auto option causes an automatic "rollover" of the log when

the directives in the value of the LOG option change. This is particularly useful for generating log files for pool or launch services. This example creates log filenames such as appsrv_1.log, appsrv_2.log, and appsrv_3.log.

Note: You must use a full path to specify the log file because there is limited control over what path the Load Manager or spawner will use as the current directory for each Application Server.

For more information, see the documentation on the LOGPARM= system option in SAS Language Elements.

Removing a Service

A service can be removed by deleting the service root directory and its contents. Any active servers must be stopped before deleting this directory.

Services on Windows Platforms

Creating a Service

To create a service for an Application Server running in a Windows operating environment, perform the following steps:

1. From the Start menu, select **Programs** ➧ **SAS** ➧ **IntrNet** ➧ **Create a New IntrNet Service**.
2. Read the information in the IntrNet Config Utility Welcome window. Select **Next** to continue.
3. Select the type of service that you wish to create. Select **Next** to continue.
4. Type the name of the new service. (Remember that service names must begin with either a letter or an underscore and may contain letter, number, underscore, or dash characters.)

   The default value for this field is `default`. Create this as your first service because this is what is used when you run the samples. Select **Next** to continue.
5. The configuration utility selects a default service root directory based on the location that you chose for user files when you installed SAS software. This default location is recommended for most users, although you can use the **Browse** button to select a different directory. Select **Next** to continue.
6. If you are defining a socket service, you are prompted to specify the TCP/IP port numbers or names for each Application Server that you want to define as part of this service. If you supply multiple numbers or names, use a space to separate each entry. Select **Next** to continue.
7. Review the contents of the Service Created window for information about your service. Note the Application Broker configuration sample text. You are required to enter similar text in the Application Broker configuration file in a later step. Select **Next** to continue.
8. The Create Service window displays all of the information that you specified for this service. Verify the information in this window before continuing. If it is not correct, select the **Back** button to change the information. If it is correct, select **Next** to create the service.
9. Select **Next** and then select **Finish** to exit the utility.
10. If you are creating a pool service, you must install the Application Load Manager. You might also want to install the Load Manager if you have a socket service with more than one server. See Using the Load Manager for more information.
11. If you are creating a pool service, you might need to install the SAS Spawner. The Spawner is not required if you configure the Application Load Manager to start the service directly. If you choose this method, the servers must all execute on the same system as the Load Manager and under the same user ID. Refer to the SAS/CONNECT documentation for installation instructions for the SAS Spawner.
12. The Application Broker must know about the new service before you can access it. Open the Application Broker configuration file on your Web server in a text editor and add a service definition block. Example service definitions are provided in the template configuration file that is installed with the Application Broker. In addition, sample configuration text was generated by the configuration utility in a preceding step. Several examples are also shown below. Values that may need to be changed for your site are shown in green.

    - The definition block for a socket service might look like

      ```
      # This service contains one server (port 5801) on yourserv.yyy.com.
      SocketService service-name
         ServiceDescription "text-desc"
         ServiceAdmin "administrator-name"
         ServiceAdminMail "administrator-email-address@host"
         Server yourserv.yyy.com
         Port 5801
      ```

    - A pool service that is started directly by the Load Manager is defined by

```
# Start up to 2 servers on the same node as the Load Manager. No
# username/password or spawner is needed for this case. Servers will
# time out after 30 minutes.
PoolService service-name
   ServiceDescription "text-desc"
   ServiceAdmin "administrator-name"
   ServiceAdminMail "administrator-email-address@host"
   ServiceLoadManager yourserv.xxx.com:5555
   SasCommand ""\"C:\\Program Files\\SAS\\SAS 9.1\\sas.exe\"+
      \"C:\\Program Files\\SAS\\IntrNet\\service-name\\appstart.sas\"+
      -rsasuser -noterminal -noprint -nolog -SYSPARM "
   IdleTimeout 30
   Server yourserv.xxx.com
   Port 2
```

♦ A pool service that is started by the SAS Spawner is defined by

```
# Start up to 5 servers on node yourserv.yyy.com using the spawner started
# at port 7777. All servers will be started with the specified
# username/password. At least 1 server will not timeout and
# will be kept running.
PoolService service-name
   ServiceDescription "text-desc"
   ServiceAdmin "administrator-name"
   ServiceAdminMail "administrator-email-address@host"
   ServiceLoadManager load-manager-host:port
   SasCommand ""\"C:\\Program Files\\SAS\\SAS 9.1\\sas.exe\"+
      \"C:\\Program Files\\SAS\\IntrNet\\service-name\\appstart.sas\"+
      -rsasuser -noterminal -noprint -log /dev/null -SYSPARM"
   Server yourserv.yyy.com
   Port 6000-6004
   Username appdemo
   Password xyzzy
   SpawnerPort 7777
   MinRun 1
```

♦ Launch services are defined with

```
LaunchService service-name
   ServiceDescription "text-desc"
   ServiceAdmin "administrator-name"
   ServiceAdminMail "administrator-email-address@host"
   SasCommand ""\"C:\\Program Files\\SAS\\SAS 9.1\\sas.exe\"+
      \"C:\\Program Files\\SAS\\IntrNet\\service-name\\appstart.sas\"+
      -rsasuser -noterminal -noprint -nolog  -SYSPARM "
```

Starting the Service

Socket services must be started manually. From the Start menu, select **Programs** ➧ **SAS** (or your SAS 9.1 program group) ➧ **IntrNet**. Select the entry for the service you just created, and then select **Start Interactively**.

Launch services are started automatically by the Application Broker. Pool services are started automatically by the Application Load Manager. See Using the Load Manager for more details.

After a service is started, you can test it from a Web browser. The URL depends on the platform and path where your Application Broker is installed. For typical installations, the URL to test (or *ping*) a service is one of the following:

Windows

`http://yourserver/scripts/broker.exe?_service=service-name&_program=ping`

UNIX and z/OS

```
http://yourserver/cgi-bin/broker?_service=service-name&_program=ping
```

Specify your Web server name in place of *yourserver* and your service name in place of *service−name*. You might need to use a different URL path if you chose a different path when you installed the Application Broker. If the service is running, an HTML page is returned stating that the Application Server is functioning.

Stopping the Service

Socket or pool services can be stopped from a Web browser. The URL depends on the platform and path where your Application Broker is installed. For typical installations, the URL to stop a service is one of the following:

Windows

```
http://yourserver/scripts/broker.exe?_service=service-name&_program=stop
```

UNIX and z/OS

```
http://yourserver/cgi-bin/broker?_service=service-name&_program=stop
```

Specify your Web server name in place of *yourserver* and your service name in place of *service−name*. You might need to use a different URL path if you chose a different path when you installed the Application Broker.

Modifying a Service or Accessing Service Log Files

Service configuration and log files are kept in a service directory tree. The service root directory can be accessed from the Windows Start menu. From the Start menu, select **Programs** ➔ **SAS** (or your SAS 9.1 program group) ➔ **IntrNet**. Select the entry for the service you just created, and then select **Service Directory**. This directory contains an appstart.sas file with the SAS code used to start the Application Servers for this service. Other scripts and configuration files for this service are also located in this directory.

Log files are placed in the logs directory under the service root directory and are named <*day*>_<*port*>.log. For example, Mon_5001.log or Tue_5001.log. By default, logs are kept for one week (six full days and one partial day) and then overwritten.

SAS 9 has implemented a set of % codes that can be used in the −log parameter. You must add a −logparm option in order to get the codes translated into the log. For example adding

```
-log 'C:\Program Files\Sas\IntrNet\service-name\logs\appsrv_%v.log'
-logparm rollover=auto
```

creates log files with unique log filenames. The rollover=auto option causes an automatic "rollover" of the log when the directives in the value of the LOG option change. This is particularly useful for generating log files for pool or launch services. This example creates log filenames such as appsrv_1.log, appsrv_2.log, and appsrv_3.log.

Note: You must use a full path to specify the log file because there is limited control over what path the Load Manager or spawner will use as the current directory for each Application Server.

For more information, see the documentation on the LOGPARM= system option in SAS Language Elements.

Windows Services

Socket services can be installed as Windows services. Installing as a Windows service enables the service to run automatically whenever the Windows system is booted, even if a user is not logged into the system. As noted above, install the SAS Service Configuration Utility before you create any service that will be installed as a Windows service.

Instructions for installing the SAS Service Configuration Utility are located in the SAS 9.1 user installation guide for Microsoft Windows at support.sas.com/installcenter.

From the Start menu, select **Programs** ➔ **SAS** (or your SAS 9.1 program group) ➔ **IntrNet**. Select the entry for the service you just created, and then select **Install as Windows Service**. This step does not start the service. Once the service has been installed as a Windows service, each time the system reboots the service is started automatically. You can manually start and stop the Windows service by using the **Start Windows Service** and **Stop Windows Service** menu selections. Note that you can still run this service interactively by using the **Start Interactively** menu selection, but only if the Windows service has been stopped.

A Windows service can also be uninstalled from the **Uninstall Windows Service** menu selection. This only uninstalls the Windows service. The SAS/IntrNet service remains and can still be run interactively.

The **Services** icon in the Windows Control Panel enables you to see the current status of all Windows services. Each SAS/IntrNet service display name begins with SAS App Server. Note that a separate Windows service is created for each server in a multiple server socket service. Individual servers can be started and stopped from the Services Control Panel. You can use the Start menu selections to start or stop all servers in the service.

Removing a Service

You can remove a service by deleting the service directory and removing the associated Start menu entry. Follow these steps:

1. If the service is running, stop it.
2. If the service is a socket service and has been installed as a Windows service, uninstall it as a Windows service. From the Start menu, select **Programs** ➔ **SAS** (or your SAS 9.1 program group) ➔ **IntrNet**. Select the entry for the service, and then select **Uninstall as a Windows Service**. Perform this step even if you are not sure if this service is installed as a Windows service.
3. Find the service directory, select **Start Menu**, and select **Programs** ➔ **SAS** (or whatever program group under which you installed SAS 9.1) ➔ **IntrNet**. Select the entry for the service, and then select **Service Directory**.
4. Go up one level to the parent directory and delete the directory for the service that you wish to remove.
5. Remove the Start menu entry for the service. To do this, right click on the Start menu and select **Explore All Users** (or **Explore** on Windows 95/98).
6. Use the Explorer window to find the Programs folder.
7. Select **SAS**, then select **IntrNet**. Delete the service entry in this folder for the service that you want to remove.

Enhancing Performance

As more users access your applications, you may need to fine−tune the performance of your Application Dispatcher setup. You can improve performance by

- Using Multiple Servers (Random Load Balancing)
- Using the Load Manager (Intelligent Load Balancing)
- Increasing Timeout
- Using Server Weights
- Specifying a Backup Machine

Using Multiple Servers (Random Load Balancing)

In a socket service definition in the Application Broker configuration file you can supply multiple names of the physical machines on which Application Servers are installed or multiple TCP/IP port numbers that represent many server processes on a single machine. The following example shows multiple servers and ports:

```
server machine_a  machine_b  machine_c  machine_d
port 5000-5002
server machine_e
port 5001 5003
```

This example provides 14 different Application Servers:

- 3 servers (5000,5001,5002) on machine_a
- 3 servers (5000,5001,5002) on machine_b
- 3 servers (5000,5001,5002) on machine_c
- 3 servers (5000,5001,5002) on machine_d
- 2 servers (5001,5003) on machine_e

For the Application Broker to access these Application Servers, they must be running. If one or more of the servers is not running, the Application Broker tries to access another one. If you need a diagnostic tool, you can set a _DEBUG value to trace the connection attempts. Notice that you can specify a range of ports, for example, 5000−5002. You can also specify any number of single ports or port ranges on one line.

When a Web browser request is made, the Application Broker uses one of two methods for selecting a server out of the specified service:

- If a Load Manager is defined in the configuration file, the Application Broker attempts to ask the Load Manager for an available server in the list that is defined for the service.
- If the Load Manager is unavailable or undefined, the Application Broker randomly chooses a server from the list.

Using the Load Manager (Intelligent Load Balancing)

The Load Manager balances requests from Application Brokers for Application Server processing. Each time an Application Broker is activated, it sends a list of Application Servers that are associated with the service to the Load Manager. The Load Manager checks an in−memory list of server states to determine if any servers are idle. The first server found idle with the least number of outstanding sessions is returned to the Application Broker and marked as in use. If no servers are idle, the request is queued by the Load Manager until a server is free. If the Application Servers are configured with multiple programs enabled, the request is sent to a server that is not handling the maximum

number of programs.

Each Application Server contacts the Load Manager to record its state. As a job is received, the server sends a message to the Load Manager indicating that the server is busy. When the job has been completed, an idle state message is sent. In this manner, the Load Manager can maintain which servers are available for Application Broker requests. The Load Manager also periodically checks server sockets to try to determine whether each server is still functioning. The Load Manager log can be used to track the state changes and job allocation for the Application Servers.

Increasing Timeout

The ServiceTimeout directive specifies the number of seconds that the Application Broker should wait for a response from the Application Server. The default value is 60 seconds. This can be lengthened depending upon your needs. When the specified time elapses, the Application Broker returns an error message to the Browser. To avoid receiving this error message and to increase your chances of connecting when the server is very busy, increase the timeout period. The global directive Timeout overrides the default of 60 seconds. The directive ServiceTimeout further overrides the global timeout for a particular service.

Using Server Weights

If you are *not using the Load Manager* and you want to direct the connection to a particular machine out of a range of the servers that you have predefined, specify weights for each machine. For example, if you have two computers running Application Servers, each machine has a one−in−two chance of receiving a request from the Application Broker. The Application Broker randomly selects one of your predefined servers. To increase the likelihood that a server receives requests, assign a weight to the machine in the server directive, as shown in the following example:

```
server machine_a  machine_b*5
```

Server A now has only a one−in−six chance of receiving a request. Server B now has a five−in−six chance. You could rewrite the previous example to show the following and attain the same result. The weight assignment of 5 saves you from having to type the following line:

```
server machine_a  machine_b  machine_b  machine_b  machine_b  machine_b
```

If you are using the Load Manager, you can use weights to reorder the list of servers in the service. The servers with higher weights appear first in the list that is sent to the Load Manager. Because the Load Manager always selects the first available server in the list, those with higher weights are selected first.

Specifying a Backup Machine

You can also use weights to specify a backup machine that receives requests only if the server that is running on the primary machine is not operating. To do so, set the weight to zero (0), as shown in the following example:

```
server machine_a  machine_b*0
```

Under normal circumstances, server A acts as your primary machine and server B never receives requests. However, if server A is not operating, the Application Broker attempts to connect to server B.

Note: Weights apply to server machines and not to individual ports on those machines. You cannot force the Application Broker to favor one port over another on the same machine.

Development vs. Production Environments

In most cases, you will need separate services for your development and production Application Dispatcher environments. Isolating your development environment from your production applications will help you provide application stability and security to your end users, while providing flexibility and freedom to your application developers.

Development Services

A development environment can typically be provided by a one−server socket service. A single server will generally simplify debugging and will allow testing of most application features (other than performance). The developer is free to start and stop the server as needed. You may wish to create a separate service for each developer to allow independent updates to the application under development.

Note: Applications that are intended to run in multiple−server services must be written and tested with this in mind. Programs must allow for concurrent access to shared resources by multiple servers. Even if the development is performed on a single−server service, some testing should be done on a multiple−server service before deploying the application.

Launch services were recommended for development on previous releases of SAS/IntrNet software. This was because the Version 6 SCL based Application Server was not tolerant of some common development errors (such as mismatched quotes in SAS code). Later releases of the Application Server are much more tolerant of these types of errors. Most programming errors will cause the individual request to fail without affecting the server.

Production Services

Most production applications can be deployed on a socket service with one or more servers. Additional servers can be added as the total system load grows. Servers can easily be distributed across multiple server systems. Once your service uses more than one server, you should consider using the Application Load Manager. The Load Manager provides intelligent load balancing between multiple servers in a socket service.

Applications with widely varying user loads or special security requirements may benefit from pool services. Pool services enable servers to be started and stopped dynamically as the user load grows and shrinks.

Using the Load Manager

The Application Load Manager is a separate process that can be used to control distribution of client requests across multiple Application Servers. The Load Manager can run on any node that is visible to both the Application Brokers and Application Servers. The Load Manager keeps track of which Application Servers are busy. When a new request arrives, it is routed to an idle server. If no servers are idle, the request queues at the Load Manager and waits for the first available server. In the case of pool services, the Load Manager can start another Application Server when all servers are busy. It can also shut down unused servers after an idle timeout.

The Load Manager is not required for socket and launch services, but it is recommended for socket services that have more than one server. The Load Manager is required for pool services. See Using Services for more information about service types. The Load Manager requires only a few command line options to start and that a single directive be added to the Application Broker configuration file. For more information, see

- Application Load Manager Reference
    - ♦ Load Manager on Windows Platforms
- Load Manager Log Files.

Application Load Manager Reference

The Application Load Manager is available for z/OS, UNIX, and Windows systems. The Load Manager executable (loadmgr.exe on Windows, loadmgr on other platforms) is included in the CGI Tools for Web Server package. After you have installed this package, you can find the Load Manager executable in the Web server directory corresponding to the URL

```
http://yourserver/sasweb/IntrNet9/tools
```

The Load Manager executable is also included in the SAS installation on UNIX and Windows systems. The Load Manager can be found at the following locations:

UNIX
```
!SASROOT/utilities/bin/loadmgr
```
Windows
```
!SASROOT\intrnet\sasexe\loadmgr.exe
```

Starting the Load Manager

The syntax for starting the load manager is

loadmgr *<options>*

where ***<options>*** can be any of the following:

–delete
removes a previously installed load manager as a system service on Windows NT.

–install
installs the load manager on Windows NT as a system service.

–log<=filename>
specifies an optional log file. The STDOUT device is used if a filename is not specified. If the Load Manager log file specification contains any of the following directives, the corresponding value will be inserted into the file name:

```
%a  Day of week [Sun – Sat]
%b  Month [Jan – Dec]
%d  day [01 –31]
%H  hour [00 – 23]
%m  month [01 – 12]
%w  day of week [0=Sunday – 6=Saturday]
%Y  full year
%y  2-digit year [00 – 99]
```

Note: Additional codes may be available depending upon the C library function strftime implementation for a given platform.

For example, "/logs/loadmgr_%a.log" creates /logs/loadmgr_Mon.log if the Load Manager starts on a Monday, /logs/loadmgr_Tue.log if it starts on a Tuesday, and so on.

Periodically, the Load Manager regenerates the log file name and checks to see if it is different from the current log file. If it is different, the current log file is closed, and the new log file with the new name is opened. In the example above, shortly after midnight, early Tuesday morning, the log file /logs/loadmgr_Mon.log is closed and the file /logs/loadmgr_Tue.log is opened.

If the Load Manager is started and finds a log file with the current name, it replaces the contents of an existing log file if the last modification date is greater than 5 days, 23 hours ago. If the last modification date is less than that, the Load Manager appends to the existing log file.

−maxreq=minutes
: specifies the maximum time it should take for the Application Server to send a BUSY state after the Application Server is allocated to the Application Broker. The default is 1 minute.

−maxrun=minutes
: specifies the expected maximum job run time in minutes before an Application Server is declared to be hung. The default is 60 minutes.

−maxstart=minutes
: specifies the maximum time that it should take an Application Server to start. The default is 5 minutes.

−nokill
: specifies not to kill a pool server that is marked as hung.

−passwd<=password>
: specifies an optional password controlling access to the ENDLOADMGR and LOADSTAT administration programs. If the −passwd option is specified without a password value, a prompt is issued for the password.

−port=nnnn
: specifies the port number or service name for the socket on which the Load Manager listens. If this parameter is not specified, the /etc/services file is checked for an entry for LOADMGR.

−workdir=directory
: enables you to specify the current working directory as a start parameter for the Load Manager.

On Windows platforms only, a setup wizard is available to configure the Load Manager. The setup wizard allows you to create Start menu shortcuts to start the Load Manager, install or uninstall the Load Manager as a Windows service, or view log files. See Load Manager on Windows Platforms for more information.

Stopping the Load Manager

The Load Manager can be stopped from a Web browser. The URL depends on the platform and path where your Application Broker is installed. For typical installations, the URL to stop a service is one of the following:

Windows
: `http://yourserver/scripts/broker.exe?_service=service-name&_program=endloadmgr`

UNIX and z/OS
: `http://yourserver/cgi-bin/broker?_service=service-name&_program=endloadmgr`

Specify your Web server name in place of *yourserver* and any service that uses the Load Manager in place of *service−name*. You might need to use a different URL path if you chose a different path when you installed the Application Broker. The ENDLOADMGR command also stops any pool service Application Servers that have been started by the Load Manager. If the load manager was started with the −passwd option, the specified password must be appended to the URL. For example:

```
http://yourserver/scripts/broker.exe?_service=service-name&_program=endloadmgr&_passwd=secret
```

Application Broker Directives for the Load Manager

To use the Load Manager, add a directive to the Application Broker configuration file:

```
LoadManager host:port
```

You can override this on a per−service basis with a corresponding ServiceLoadManager directive:

```
ServiceLoadManager host:port
```

No other changes are required. All information in the Application Broker configuration file are passed to the Load Manager by the Application Broker as needed.

Load Manager Statistics

Statistics are recorded for all Application Servers requested through the Load Manager. Data is kept for the length of jobs and for the amount of time required to wait for a server. The job times are based on state changes sent from the Application Servers and vary slightly from the job times reported by the Application Broker.

A statistics report may be obtained by running a special program via the Application Broker. The load manager statistics are returned when _PROGRAM is set to LOADSTAT and _SERVICE specifies any service that uses the desired load manager. If the load manager was started with the −passwd option, _PASSWD must be used to supply the password. For example, the URL

```
http://yourserver/scripts/broker.exe?_service=default&_program=loadstat&_passwd=secret
```

might return the following report:

Load Manager serv.abc.com:5555

Service default

Server	Port	Total Jobs	Max Job Time	Average Job Time	Percent Waited	Average Wait Time
serv.abc.com	5197	2	0.52	0.38	0.00	0.00

Service pool1

Server	Port	Total Jobs	Max Job Time	Average Job Time	Percent Waited	Average Wait Time
poolserv.abc.com	2909	12	10.40	2.20	10.00	1.79
poolserv.abc.com	2940	1	0.08	0.08	100.00	0.19

The columns of the report are defined as

Server
: is the Application Server host.

Port
: is the port number for the Application Server.

Total Jobs
: is the number of complete Application Broker requests that were processed.

Max Job Time
: is the length of the longest−running Application Broker request, in seconds.

Average Job Time

is the average length of all Application Broker requests, in seconds.

Percent Waited
is the percentage of Application Broker requests that had to wait for an Application Server.

Average Wait Time
is the average amount of time, in seconds, that an Application Broker request waited, if it had to wait.

Load Manager Data for Application Server Activity

An Application Server activity report may be obtained by running a special program via the Application Broker. The activity data is returned when _PROGRAM is set to LOADCURRENT and _SERVICE specifies any service that uses the desired load manager. It reports only information contained in the Load Manager and does not contact any Application Servers. If the load manager was started with the −passwd option, _PASSWD must be used to supply the password. For example, the URL

```
http://yourserver/scripts/broker.exe?_service=default&_program=loadcurrent&_passwd=secret
```

might return the following report:

Load Manager serv.abc.com:5555

Service pool1

Server	Port	State	Total Jobs	Last Job
poolserv.abc.com	2990	BUSY	1	Jan 15 14:32:04
poolserv.abc.com	2972	BUSY	3	Jan 15 14:32:01
poolserv.abc.com	2985	BUSY	2	Jan 15 14:32:04

Waiters: 2

The columns of this report are defined as

Server
is the Application Server host.

Port
is the port number for the Application Server.

State
is the current Application Server state, which specifies whether or not the Application Server is busy.

Total Jobs
is the number of complete Application Broker requests that were processed.

Last Job
is the time when the last Application Server was assigned to a job.

Waiters
is the number of clients that are waiting for an available Application Server.

Load Manager on Windows Platforms

Configuring and Starting the Load Manager

To configure the Load Manager in a Windows environment, perform the following steps:

1. From the Start menu, select **Programs** ➧ **SAS** ➧ **IntrNet** ➧ **Create a New IntrNet Service**.
2. Read the information in the IntrNet Config Utility Welcome window. Select **Next** to continue.
3. Select **Configure the Load Manager**. Select **Next** to continue.
4. Specify the TCP/IP port number or name for the Load Manager. Select **Next** to continue.
5. The Configure Load Manager window displays all of the information you specified for this service. Verify the information in this window before continuing. If it is not correct, select the **Back** button to change the information. If it is correct, select **Next** to configure the Load Manager.
6. Select **Finish** to exit the wizard.

The Load Manager is now configured. You can start the Load Manager from the Windows Start menu by selecting **Programs** ➧ **SAS** ➧ **IntrNet** ➧ **Load Manager** ➧ **Start Interactively**.

Return to the Application Load Manager Reference for general information about the Load Manager.

Accessing Log Files

Log files are kept in a Load Manager service directory. The service directory can be accessed from the Windows Start menu. From the Start menu, select **Programs** ➧ **SAS** (or your SAS 9.1 program group) ➧ **IntrNet** ➧ **Load Manager** ➧ **Log Directory**.

Windows Services

The Load Manager can be installed as a Windows service. Installing as a Windows service enables the Load Manager to run automatically whenever the Windows system is booted, even if a user is not logged into the system. As noted above, install the SAS Service Configuration Utility before you Configure a Load Manager that will be installed as a Windows service. Instructions for installing the SAS Service Configuration Utility are located in the SAS 9.1 user installation guide for Microsoft Windows at support.sas.com/installcenter.

From the Start menu, select **Programs** ➧ **SAS** (or your SAS 9.1 program group) ➧ **IntrNet** ➧ **Load Manager** ➧ **Install as Windows Service**. This step does not start the Load Manager. After the Load Manager has been installed as a Windows service, each time the system reboots the Load Manager is started automatically. You can manually start and stop the Load Manager Windows service by using the **Start Windows Service** and **Stop Windows Service** menu selections. Note that you can still run the Load Manager interactively by using the **Start Interactively** menu selection, but only if the Windows service has been stopped.

A Windows service can also be uninstalled from the **Uninstall Windows Service** menu selection. This only uninstalls the Windows service. The Load Manager Start menu shortcut remains, and the Load Manager can still be run interactively.

Selecting the **Services** icon in the Windows Control Panel opens a window that lists the current status of all Windows services. The Load Manager service display name is SAS IntrNet Load Manager.

Application Load Manager Log Files

The Application Load Manager generates a log file that lists requests and events that are processed by the Load Manager. This log can help you determine how Application Dispatcher requests are distributed among available servers or to find problems when starting new servers.

Example 1

Here is a sample log file for a socket service with line numbers added for reference:

```
Line 1 Wed Jun 03 2000 09:16:14 GET SERVER default aaa.bbb.com:5612 3
Line 2 Wed Jun 03 2000 09:16:14 Waiting for default
Line 3 Wed Jun 03 2000 09:16:18 SET STATE default aaa.bbb.com:5612 IDLE 2 0/1
0
Line 4 Wed Jun 03 2000 09:16:19 Returned default aaa.bbb.com:5612 3
Line 5 Wed Jun 03 2000 09:16:23 SET STATE default aaa.bbb.com:5612 BUSY 3 1/1
0
Line 6 Wed Jun 03 2000 09:16:26 SET STATE default aaa.bbb.com:5612 IDLE 3 0/1
0
Line 7 Wed Jun 03 2000 09:16:26 SET STATE default aaa.bbb.com:5612 SHUTDOWN
```

This log file shows the following events:

Line 1	The Application Broker requests an Application Server from the default service. This request is assigned the number 3.
Line 2	No server is available. Load Manager is waiting for an idle server.
Line 3	The server completes a previous request and notifies the Load Manager that the server is IDLE. The 0/1 indicates that the Application Server does not have any programs running but has 1 space available. The trailing 0 is the number of active server sessions.
Line 4	The Load Manager releases the available server to Broker request 3.
Line 5	The Broker has submitted its request to the server. The server notifies the Load Manager that the server is BUSY, with the 1 available space in use and 0 server sessions active.
Line 6	The server completes this request and becomes IDLE again.
Line 7	The Application Server is shut down.

Example 2

Here is another sample log file showing a pool service with line numbers added for reference:

```
Line 1 Mon Aug 07 2000 13:22:53 GET SERVER pool1 xxx.yyy.com:2 5
Line 2 Mon Aug 07 2000 13:22:53 Started pool1 on xxx.yyy.com Pid: 17979
Line 3 Mon Aug 07 2000 13:22:53 Command: /usr/local/bin/sas
/usr/local/intrnet/pool1/appstart.sas
-rsasuser -noterminal -noprint -nolog -SYSPARM "loadmgr=xxx.yyy.com:5555
serviceid=pool1 port=000000"
Line 4 Mon Aug 07 2000 13:22:53 Waiting for: pool1 5
```

```
Line 5 Mon Aug 07 2000 13:22:56 SET STATE pool1 xxx.yyy.com:4525 STARTED
Line 6 Mon Aug 07 2000 13:22:56 Returning: pool1 xxx.yyy.com:4525 5
Line 7 Mon Aug 07 2000 13:22:57 SET STATE pool1 xxx.yyy.com:4525 WORKING 5 1/3
1
Line 8 Mon Aug 07 2000 13:22:58 SET STATE pool1 xxx.yyy.com:4525 IDLE 5 0/3 1
```

This log file shows the following events:

Line 1 The Application Broker requests an Application Server from the pool1 service. This request is assigned the number 5.

Line 2 No server is available so the Load Manager starts a new Application Server.

Line 3 This is the exact command that is used to start the server.

Line 4 The requesting Broker is placed in the wait queue waiting for the server to start.

Line 5 The server notifies the Load Manager that it has started.

Line 6 The Load Manager releases the available server to Broker request 5

Line 7 The Broker has submitted its request to the server. The server notifies the Load Manager that the server is in a WORKING state, with 1 of the 3 available spaces in use and 1 server session active.

Note: The WORKING state will only appear if the Applications Server was started with the PROGRAMS parameter set to be greater than one.

Line 8 The server completes this request and becomes IDLE again.

Using SAS Design−Time Controls

SAS Design−Time Controls are add−in components for your HTML editor that help you to easily add SAS content to your Web pages. Design−Time Controls act like page−component wizards that help you to build parts of your Web page. The controls present a user−friendly, intuitive interface that insulates you from much of the complexity that comes along with sophisticated Web content. You get to control the look and feel of your Web pages in a WYSIWYG editor and at the same time access and surface the power of SAS software on your Web page.

For information about data administration for the SAS Design−Time Controls, see Making Data Available in the SAS Design−Time Controls documentation. For more details about the SAS Design−Time Controls and how to use them, see the SAS Design−Time Controls documentation.

The Input Component

A Dispatcher application is composed of input and program components. The program component is the actual SAS program that runs on the Application Server. The input component is the remainder of the application, which runs on the Web server or the client. It normally consists of static or dynamically generated HTML pages that contain one or more of the following:

- an HTML form that has a **Submit** button. When the Web user provides the required information and submits the request, the browser sends the data that was entered plus data from any hidden fields to the Application Broker.
- a hypertext link to the Application Broker. When the user selects the link, the browser sends a request, which includes parameters that are specified in the link's Universal Resource Locator (URL), to the Application Broker.
- an inline image whose source is a reference to the Application Broker. When the user brings the page up for viewing, the browser loads the image and sends a request to the Application Broker. Similar to the process used for a hypertext link, parameters can be included in the URL.
- a Java applet, ActiveX control, or Plug–in that contains a reference to the Application Broker. Depending on the object, the Application Server may send a request to the Application Broker immediately or wait for a user action, such as clicking a button.

The input component selects what program component to run and passes input data to the program component as a list of name/value pairs. The name/value pairs can be specified in a URL, in input fields in an HTML form, by an object such as a Java applet or ActiveX control, or in the Application Broker configuration file as described below. The Web user, who does not need to know how the Dispatcher passes and processes the information, receives the results of the application in the browser. The results are typically displayed as an HTML page, but they can be presented as a downloaded file in more sophisticated applications.

The Dispatcher uses macro variables to pass name/value pair data to your programs. SAS Component Language (SCL) programs are supplied with an SCL list as an additional mechanism for accessing the data. Usually, both the macro variable names and list–item names match the names supplied in the HTML code. The HTML names that are used to create the macro variable names must be valid SAS names and must be expected by the program. The Dispatcher rejects invalid SAS names.

Because the SAS rules for names are more restrictive than the rules for HTML names, Dispatcher application developers use the following SAS naming rules for all fields:

- Use between 1 and 32 characters.
- Begin the name with a letter or an underscore.
- Continue the name with letters, underscores, or digits.

Reserved Names

Reserved names have special meaning to the Application Dispatcher. For example, every request must include a _PROGRAM name/value pair to identify the program to be run by the Application Dispatcher. In most cases, a _SERVICE name/value pair is required to identify the service that handles the request. More details on these and other special variables (name/value pairs) are available in Reserved or Special Variables.

Specifying Name/Value Pairs in a URL

You can specify name/value pairs in a URL by using Application Broker CGI−parameter syntax. For example, the URL

```
http://yourserver/cgi-bin/broker?_service=default&_program=sample.webhello.sas
```

specifies two name/value pairs. Note the question mark (?) that follows BROKER. The section of the URL that follows the question mark is called the *query string*. The query string contains the name/value pair data that is input to the application. Each name is separated from the following value by an equal sign (=). Multiple name/value pairs are separated by ampersands (&). In this example, the _SERVICE=DEFAULT pair specifies the service that handles this request, and the _PROGRAM=SAMPLE.WEBHELLO.SAS pair specifies the request program that is executed.

The Web browser has strict rules about the format of the query string. Any special characters (including spaces) in a value must be URL encoded. Spaces can be encoded as a plus sign (+) or %20. For example, if you wish to pass the name AUTHOR with a value of John Doe, specify it in the URL as `AUTHOR=John+Doe` or `AUTHOR=John%20Doe`. See the HTML Syntax Reference section and the URLENCODE function for more complete information.

URLs with name/value pairs can be manually typed in a browser location field, saved as a browser bookmark, included as an HREF attribute of an anchor tag, included as an SRC attribute of an IMG tag, or used anywhere a URL may be used. Java or ActiveX components such as the SAS/GRAPH thin−client graphic components might generate URLs with name/value pairs to activate Dispatcher programs.

URLs with name/value pairs that are included in an HTML page (for example, as an HREF= or SRC= attribute) must be properly encoded to prevent incorrect interpretation of the ampersand characters. For example, the anchor tag

```
<A HREF="http://yourserver/cgi-bin/broker?_program=lib.pgm.sas&copy=true">
```

causes the browser to interpret &COPY as the character entity reference for a copyright character. The correct way to encode this URL is

```
<A HREF="http://yourserver/cgi-bin/broker?_program=lib.pgm.sas&amp;copy=true">
```

In addition, some browsers incorrectly identify a character entity reference even if it is not terminated by punctuation. For example, &REGION=EAST might be interpreted as fiION=EAST by some (but not all) browsers. To avoid this problem, encode all ampersands that separate name/value pairs in a URL as `&` when used in an HTML tag.

Specifying Name/Value Pairs in an HTML Form

HTML forms provide the most versatile mechanism for data input in a Dispatcher application. A form definition begins with the <FORM> tag and ends with the </FORM> tag. Between these two tags, other HTML tags define the various components of the form, including input fields, selection lists, push buttons, and more. Several forms of varying complexity are included in the HTML file section of the Application Broker package sample directory. The HTML code in these files, as well as the descriptions in the following sections, helps you learn how to create forms. A detailed list of form requirements and components can be found in the HTML Syntax Reference section.

Hidden fields are name/value pairs that do not appear as buttons, selection lists, and so on in the HTML page. Here is an example of a hidden field:

```
<INPUT TYPE="hidden" NAME="_service" VALUE="default">
```

This HTML tag passes the name/value pair _SERVICE=DEFAULT when the form that contains the name/value pair is submitted. The required Dispatcher fields _SERVICE and _PROGRAM are often passed as hidden fields, but you can also include your own fields as hidden fields. Although hidden fields do not appear visually in the Browser, you can use them to

- pass parameters to the Dispatcher program. For example, you can pass a list of variables to a Dispatcher program for processing. A single Dispatcher program can then be referenced by many HTML files.
- pass name/value pairs from one form to the next. The input component to a complicated application often has more than one form and more than one page, which means that the name/value pair data must be propagated through each of the forms until the final program is invoked. Hidden fields are an easy way to accomplish this.
- capture data generated by user interaction with screen widgets if your application uses JavaScript or Visual Basic Script.

Specifying Name/Value Pairs in the Application Broker Configuration File

You can specify name/value pairs in the Application Broker configuration file (broker.cfg). The Set directive defines a constant name/value pair that is passed to all program components. For example, your broker.cfg might contain

```
Set IMGHOME http://server.xyz.com/images
```

This directive defines the name/value pair `IMGHOME=http://server.xyz.com/images` for all requests executed by this Application Broker. The IMGHOME macro variable can then be used to construct URL links to images in an HTML page without coding a fixed URL path in each Dispatcher program. This feature is used to define the codebase location of SAS/GRAPH Java applets (in the _GRAFLOC name/value pair) in a default SAS/IntrNet installation. The ServiceSet directive defines a name/value pair for a specific service.

You can define name/value pairs by issuing the Export and ServiceExport directives. These directives enable you to export a CGI environment variable as a name/value pair. The default configuration file exports a number of variables. For example, the

```
Export REMOTE_USER _RMTUSER
```

directive exports the REMOTE_USER environment variable as the _RMTUSER name/value pair. See Exporting Environment Variables for more information.

Application Broker directives are documented in Configuration File Directives.

Multiple Value Pairs

In some cases, multiple name/value pairs with the same name are created. Because SAS macro variables do not allow multiple values, the Application Broker creates a unique macro variable name for each value provided. It does this by adding numbers to the end of the name.

As an example, assume you have a group of four check boxes, each named CBOX. The value associated with each box is ONE, TWO, THREE, and FOUR, respectively. The HTML for these check boxes is

```
<input type="CHECKBOX" name="CBOX" value="one">
<input type="CHECKBOX" name="CBOX" value="two">
<input type="CHECKBOX" name="CBOX" value="three">
<input type="CHECKBOX" name="CBOX" value="four">
```

If you select all four boxes, part of the query string that is passed to the Application Broker looks like

```
CBOX=one&CBOX=two&CBOX=three&CBOX=four
```

The Application Broker then sends the following name/value pairs to your application:

Name	Value
CBOX0	4
CBOX	one
CBOX1	one
CBOX2	two
CBOX3	three
CBOX4	four

The CBOX0 value indicates the number of boxes selected. The original variable name is passed with a value equal to the first selection. Though it may seem redundant to have CBOX and CBOX1 with the same value, it is done for consistency in the case of a single selection. This example also applies to a multiple selection list named CBOX that contains the same four selected values.

The input types that can generate multiple values for one name are as follows:

check boxes
: You can select multiple check boxes from a group of boxes. All of the check boxes can have the same HTML name, which can create multiple values for one name.

selection lists
: You can select multiple items from some selection lists. These lists generate multiple values with the same name if more than one item is selected.

text entry fields
: You can enter free−form text in text entry fields. Only one value is passed from the browser to the Application Broker. If the text is too long for a single variable (usually 32000 characters), the Application Broker splits the text into multiple name/value pairs.

HTML Syntax Reference

The information contained in this section is only a partial listing of the HTML tags that your browser understands. For more detailed information about HTML and elements needed to create a basic form, see the World Wide Web Consortium at www.w3.org or the Web Design Group at www.htmlhelp.com. Square brackets in syntax indicates that an attribute is optional; do not include the square brackets in your code.

HTML Tags

Quotation Marks

You can use quotation marks to enclose the values provided in your HTML page, and you should use quotation marks if the values contain blanks. Use double, not single, quotation marks. If you are entering text that contains a single quotation mark, you must enclose the entire string in double quotation marks.

Anchor Tag

<A HREF=*URL*>

The HREF= attribute specifies a hypertext link. When selected by the user, this link invokes the Dispatcher. For example:

```
<A HREF="/cgi-bin/broker?_service=default&program=sample.webhello.sas">click me</A>
```

See URL Syntax for more information on URLs.

FORM Tag

<FORM ACTION=*broker−URL* [METHOD=GET | POST]>

The ACTION= attribute, included in the FORM tag, specifies the location of the Application Broker CGI program. For example:

```
<FORM ACTION="/cgi-bin/broker">
```

The METHOD= attribute is optional. It specifies the value GET or POST. The *broker−URL* cannot contain a question mark or have any parameters. For example:

```
<FORM ACTION="/scripts/broker.exe" METHOD=PUT>
```

- Use GET for nonupdate programs that have no side effects. GET is limited to between 256 and 1024 characters total URL length, depending on your browser. If your application is complex, the resulting page's URL can become very long and may display variable information that you would prefer the user not see. You can bookmark pages by using GET.
- Use POST for operations that have potential side effects (such as writing to a data set). POST is a simple security technique that hides the inner workings of your application from the user and hides the variables that can appear on the URL location line from the users. It also prevents form data from appearing in Web server logs. However, you cannot bookmark these pages.

On some browsers, such as Netscape, the reload button works with both GET and POST. On other browsers, such as Internet Explorer 3.02, refresh does not repost the form data. This works only with GET. If you omit the METHOD= attribute from the FORM tag, the Dispatcher uses the default GET.

Note: If you want to restrict applications from using either the GET or the POST method, use the ALLOW directive in the Application Broker configuration file. If you want to invoke the Application Dispatcher with a hypertext link, an inline image, or other URL, use the default method GET.

IMG Tag

<IMG SRC=*image−URL* [HEIGHT=*value*] [WIDTH=*value*] [ALT=*value*]>

A reference to an inline image causes the Dispatcher to be invoked as soon as the page is viewed. For example:

```
<IMG SRC="/cgi-bin/broker?_service=default&program=sample.grphics.sas">
```

See URL Syntax for more information on URLs.

INPUT Tag

<INPUT TYPE=*input−type* NAME=*input−name* VALUE=*input−value*>

The INPUT tag specifies a simple input element inside a form. The INPUT tag can include the following attributes:

The TYPE= attribute identifies the type of input specified. Valid types are

Value	Description
CHECKBOX	specifies a single toggle button that is either on or off.
HIDDEN	indicates not to display the fields in the form on the browser.
PASSWORD	specifies a text−entry field where the entered characters are represented as asterisks.
RADIO	specifies a single toggle button that is either on or off. Other fields that have the same NAME are grouped into one−of−many behavior.
RESET	specifies a push button that re−sets input elements on the form to their default values.
SUBMIT	specifies a push button that packages data entered in the current form into a request that is sent to the Application Broker CGI (and then to SAS software for processing).
TEXT	A simple text−entry field.

The NAME= attribute identifies the name in a name/value pair that passes to the Application Broker CGI and then on to SAS software for processing.

The VALUE= attribute depends on the TYPE=.

- For TYPE= TEXT or PASSWORD, use VALUE= to specify the default contents of the field.
- For TYPE= CHECKBOX or RADIO, use VALUE= to specify the value that is passed in response to a checked button. Unchecked buttons are not passed by the browser.
- For TYPE= SUBMIT or RESET, use the NAME= attribute to specify the label for the push button.

TEXTAREA Tag

<TEXTAREA NAME=*field−name* [ROWS=*rows−value*] [COLS=*cols−value*]>

The TEXTAREA tag inserts a free−form field for text, which enables the user to enter more than just a single line of text. Use this with the _FLDWDTH attribute.

URL Syntax

URLs must be encoded according to strict rules whether they appear in static HTML pages, are created by htmSQL or the Dispatcher in dynamic pages, or are typed manually into the **Location** field of the browser. This section gives a quick overview. For more information, see W3C's Web Addressing Overview at www.w3.org/Addressing.

Here is a sample URL that is broken into two lines for readability:

```
http://yourcomp.com/cgi-bin/broker?_service=default
&_program=dev.houses.sas&name=Fred%20Jones
```

where

http:	is the protocol (must be http: for Application Broker invocations).
yourcomp.com	indicates the name of the Web server.
cgi−bin	is the path to the Application Broker; an alias or directory mapping set up in the Web server.
broker	is the name of the program to run. For the Application Dispatcher this will usually be broker, broker.exe, or broker.cgi.
? (question mark)	indicates the start of parameters.
name=value	is a name/value pair. URLs can have zero or more name/value pairs, just like an HTML form.
&	separates name/value pairs.
%nn	indicates an escape character in hexadecimal notation. In the example, %20 is a space. This escape notation is used for any characters in a name or value other than alphanumeric characters or one of the following punctuation marks: "−_.!~*'()". Use the URLENCODE Function to escape characters in a URL string.

A *partial URL* results if the protocol, the Web server, or the path is omitted. Partial URLs use information from the currently viewed page to fill in the blanks. For example, if your current page is

```
http://yourcomp.com/cgi-bin/broker?_debug=4
```

and the source code references the URL

```
broker?_service=default&_program=x
```

then the actual URL is

```
http://yourcomp.com/cgi-bin/broker?_service=default&_program=x
```

This is very useful when you move pages between directories or servers, because there are fewer changes to make.

The Program Component

Dispatcher applications are composed of program components and input components. The name *program component* is a convenient name for the part of the Dispatcher application that runs on the SAS server. It will be one of the four types of programs that are listed below. The input component is stored on the Web server and is the interface between users and the program component (or SAS).

The Four Types of Programs

There are four types of Dispatcher programs:

- SAS Programs
- Source Entries
- SCL Entries
- Macro Entries

The input component of the Dispatcher application must pass a special variable named _PROGRAM. This variable names the program to run and also specifies the program type. The value for _PROGRAM is a three– or four–level name delimited by periods(.). The first level in the name indicates the Dispatcher program library where the program is stored. The last level in the name must be `sas`, `source`, `scl`, or `macro`.

SAS Programs

SAS programs are stored in external files, and these files must

- have a .sas filename extension on directory–based platforms. The filename must match the combined second and third levels in the value of _PROGRAM.
- be contained in a partitioned data set (PDS) on z/OS systems. The PDS member name must match the second level in the value of _PROGRAM.

SAS programs can contain a DATA step, procedures, and macro code. This is the only program type not stored in a SAS catalog. The program name is *case sensitive if the Application Server platform is case sensitive*. The proper query string syntax for specifying a SAS program is

```
_program=library.program.sas
```

Source Entries

Source entries are stored in SAS catalog entries with an entry type of SOURCE. They can contain the same code as SAS programs. The program names for source entries are *not case sensitive*. The proper query string syntax for specifying a program of this type is

```
_program=library.catalog.program.source
```

SCL Entries

SCL entries are stored in SAS catalog entries with an entry type of SCL. These entries contain SCL code that must be compiled. The program names for SCL entries are not case sensitive. The proper query string syntax for specifying an SCL entry program is

```
_program=library.catalog.program.scl
```

Note: There are many visual functions, objects, and routines in SCL that require a windowing environment. The Application Server normally does not run in an interactive windowing environment and cannot support visual SCL components. Using visual components in Application Server programs can produce unpredictable results and is not supported.

Macro Entries

Macro entries are stored in SAS catalog entries with an entry type of MACRO. They consist of compiled SAS macro language statements. These programs can be created with the STORE option in the %macro statement along with the SAS option SASMSTORE= to indicate a library. Using macro entries can speed up the execution of macro code when compared to SAS programs or source entries. Because the macro code contained within macro entries is stored in compiled form, there is a performance improvement. Names for macro entries are not case sensitive. The proper query string syntax for specifying a macro entry program is

```
_program=library.catalog.program.macro
```

SAS software automatically creates stored compiled macros in a catalog named SASMACR. The Dispatcher allows you to copy these macro entries to any catalog name and run them. They do not have to be in a catalog named SASMACR for the Dispatcher to access them.

Receiving Input Component Data

The name/value pair data provided by the input component are sent to the program component and made available as macro variables. The Dispatcher creates these variables, assigns their values, and clears their values after the program has completed. The name/value pair data are also supplied in an SCL list to Dispatcher programs written in SCL.

Application developers must write their Dispatcher program to accept the proper macro variable names. The macro variable values can be obtained by direct reference (for example, `&var`) or by using one of the following:

- the SYMGET function of the DATA step
- the SYMGET, SYMGETC, and SYMGETN functions in SCL.

For example, if the HTML name/value pair for a text entry field is `color=blue`, all of the following store the value blue in the DATA step variable `color`:

```
color="&color";
```

or

```
color=%superq(color);
```

or

```
color=symget('color');
```

The left side of each assignment statement is the DATA step variable. The right side shows three different techniques for extracting the macro variable value. All of these techniques return the 'safe' value of the input value. The Application Server will strip any unsafe characters (as defined by the UNSAFE option on PROC APPSRV). This means it is usually safe to use the `&var` reference in Application Dispatcher programs. Use the APPSRV_UNSAFE function to retrieve the full input value, including any 'unsafe' characters:

```
color=appsrv_unsafe('color');
```

Because all macro variables are a character data type, some extra processing is required in DATA step code if the value will be stored in a numeric variable. For example:

```
age=input(symget('age'),12.);
```

If the Dispatcher program is written in SCL, you have another option for accepting the variable values. An SCL list is passed to each Dispatcher program written in SCL. Therefore, each SCL program should contain the following statement:

```
entry inputlist 8;
```

The input list contains named character items that correspond to the macro variables created. If your program is written in SCL, you can use either the input list or macro variables. To access the same name/value pair as above, a statement like this can be used:

```
color=getnitemc(inputlist,'COLOR',1,1,'');
```

As with the macro variables, this SCL list is cleaned up by the Application Server when the Dispatcher program completes.

The Dispatcher automatically creates several variables based on the program request and various information in the Application Broker configuration file. These automatic variables are available to your program as macro variables and SCL list items. For a complete list of these automatic variables, see the sections Reserved or Special Variables and Exporting Environment Variables.

Reserved or Special Variables

Application Dispatcher variables are referred to as name/value pairs, symbols, fields, or variables. You define most variables, such as the name of a data set to graph or a year to use in a WHERE clause. Some fields have special meaning to the Dispatcher and are described here.

Value	Description
_ADMAIL	E−mail address of the administrator. Automatically generated by the Application Broker according to the AdministratorMail directive.
_ADMIN	Name of the administrator. Automatically generated by the Application Broker according to the Administrator directive.
_DEBUG	Debugging flags. Default value is set by the Debug directive. See also Setting the Default Value of _Debug.
_PGM	The next to last level in the value of the _PROGRAM variable. Indicates the name of the external file or catalog entry containing the current program code. This variable is created by the Application Server and is not one of the symbols passed from the Application Broker.
_PGMCAT	The second level in the value of the _PROGRAM variable. It indicates the SAS catalog containing the current program. This variable is blank for programs of type SAS because they have three−level names and are stored in external files. This variable is created by the Application Server and is not one of the symbols passed from the Application Broker.
_PGMLIB	The first level in the value of the _PROGRAM variable. It indicates the program library for the current program. This variable is created by the Application Server and is not one of the symbols passed from the Application Broker.
_PROGRAM	Name of the Dispatcher program that the Application Server should run. The following lists the program types and the syntax for each: *A SAS program (an external file containing SAS source code with a .SAS extension)* Specify the fully qualified, three−level name: *library*.*filename*.sas. *A source entry (a catalog entry with a .SOURCE extension)* Specify the four−level name: *library*.*catalog*.*entry*.source. *A macro entry (a catalog entry with a .MACRO extension)* Specify the four−level name: *library*.*catalog*.*entry*.macro. *An SCL entry (a catalog entry with a .SCL extension)* Specify the four−level name: *library*.*catalog*.*entry*.scl. You must specify a three− or four−level name in the _PROGRAM field, except when using Application Dispatcher−reserved Server Administration Programs, such as STATUS and STOP. For information about Application Server Administration Programs, see Application Server Administration Programs. For more information on _PROGRAM see The Four Types of Programs.
_PGMTYPE	The last level in the value of the _PROGRAM variable. It indicates the type of the current program. This variable is created by the Application Server and is not one of the symbols passed from the Application Broker.

_PORT	The TCP/IP port number of the current Application Server. Together with _Server, it indicates the server selected out of the specified service.
_REPLAY	A complete URL for use with programs that use the Output Delivery System (ODS). It is composed from the values of _URL, _SERVICE, and _TMPCAT. ODS uses this URL to create links that will replay stored output when they are loaded by the user's Web browser. See also Using the Output Delivery System. This variable is created by the Application Server and is not one of the symbols passed from the Application Broker.
_SERVER	The DNS or IP address of the current Application Server. Together with _PORT, it indicates the server selected out of the specified service.
_SERVICE	Name of a Dispatcher Service defined in your configuration file with the LaunchService or SocketService directives.
_STATDATALIBNAME	The LIBNAME, the physical name, and the options of the ALLOCATE FILE statement for the statistics data set library. This variable enables the application to assign a LIBNAME to the library with additional options (for example, ACCESS=READONLY).
_STATDATASET	The library.DATASET setting of the statistics data set for this server.
_STATDATASETAVAIL	The status of the statistics data set. This variable is set to one of the following values: OK, NOADMINPW, or NOSTATS. See also the STATISTICS statement.
_THISSRV	A URL composed from the values of _URL and _SERVICE. This variable is created by the Application Server and is not one of the symbols passed from the Application Broker.
_THISSESSION	A URL composed from the values of _URL, _SERVICE, _SERVER, _PORT, and _SESSIONID. This variable is created by the Application Server and should be used as the base URL for all URL references to the current session.
_TMPCAT	A unique, temporary catalog name. This catalog can be used to store temporary entries to be retrieved later. In socket servers, the _TMPCAT catalog is deleted after a number of minutes specified in the variable _EXPIRE. This variable is created by the Application Server and is not one of the symbols passed from the Application Broker. See Using the Output Delivery System.
_URL	Self−reference to the Application Broker CGI program. Useful for generating pages that have links or inline images that reinvoke the Dispatcher. See also the SelfURL directive.
_VERSION	Application Broker version number. Automatically generated by the Application Broker.

HTTP Headers

All output that is created by Dispatcher programs must contain an abbreviated HTTP header. This header is everything from the beginning of the output up to the first null line.

Starting with Version 8.1, the Application Server provides Automatic Header Generation.

Here is some example output, including the header:

```
Content-type: text/html
Pragma: nocache

<HTML>
<HEAD><TITLE>Application Server Administrative Program</TITLE></HEAD>
<BODY>
<H1>Administrative Program</H1>
<P>The application server has been shut down.</P>
<HR>
</BODY>
</HTML>
```

In this example, the HTTP header contains two lines. The minimal requirements for Dispatcher output are that the header contain `Content-type` or `Location`. The null line that terminates the HTTP header is important. You can create the null line with a PUT statement:

```
put ;
```

This, however, is incorrect because it produces a line containing one blank followed by carriage control:

```
put " ";
```

A line with one blank is not a null line and is not recognized as terminating the header.

The output that follows the HTTP header depends upon the content type. If Location is used, then no output follows the header because this header triggers the browser to redirect to another page. The most common type of output is, of course, HTML. The HTML source for the Web page follows the header when the content type is `text/html`.

No matter whether the program output is plain text, binary graphics, HTML code, or any other content type, all output intended for the Web browser should be sent to the fileref _WEBOUT. This special fileref is actually a TCP/IP socket connection to the Application Broker. Sending output to this socket will stream it back to the browser. Think of the socket like a pipe through which data flows. Because it behaves in this way, the fileref _WEBOUT is in a permanent append mode. It is not possible to write something to _WEBOUT and then reopen the fileref and overwrite the previous output. It all gets appended. Therefore, the `mod` parameter should not be used (and is not allowed) in any FILE _WEBOUT statements. Prior to Version 7 of SAS, an additional fileref _GRPHOUT was necessary on z/OS systems because of translation issues from EBCDIC to ASCII. The two filerefs _GRPHOUT and _WEBOUT were synonyms on all hosts except for z/OS under Version 6 of SAS. In SAS, Version 7 and later, these two filerefs are synonyms for all platforms including z/OS. Though no longer needed, the fileref _GRPHOUT is still present for compatibility reasons.

Automatic Header Generation

Starting with Version 8.1, the Application Server provides Automatic Header Generation. The default header is `Content-type: text/html`.

To add a header to the default header list or to modify a header already in the list, use the DATA step function APPSRV_HEADER.

The Application Server detects whether the user application is writing its own headers. Preexisting applications that write their own headers will continue to work as before. New applications that do not output headers will have default headers generated for them.

Applications that want to use the default headers but also want to modify them or add to them can use the APPSRV_HEADER DATA step function. For example,

```
old = appsrv_header('Header name', 'Header value');
```

Calls to the APPSRV_HEADER function adds headers to the list of default headers when the header name does not already exist in the list of default headers. In this case, the return value of the function call will be an empty string.

If the header name passed into the APPSRV_HEADER function already exists in the list of default headers, the header value of the existing header is replaced with the new value passed in, and the old value of the header is returned as the return value of the function. If the header value passed in is an empty string, then the header is removed from the list of default headers. The old value of the header is returned as the return value of the function.

Example

With default headers of `Content-type: text/html`, the following calls to the APPSRV_HEADER function will modify the default headers as shown:

```
rc = appsrv_header('Expires','Thu, 18 Nov 1999 12:23:34 GMT');
```

results in

- `Content-type: text/html`
- `Expires: Thu, 18 Nov 1999 12:23:34 GMT`

```
rc = appsrv_header('Pragma','nocache');
```

results in

- `Content-type: text/html`
- `Expires: Thu, 18 Nov 1999 12:23:34 GMT`
- `Pragma: nocache`

```
rc = appsrv_header('Expires','');
```

results in

- `Content-type: text/html`
- `Pragma: nocache`

```
rc = appsrv_header('Pragma','nocache');
```

results in

- `Content-type: text/html`
- `Pragma: nocache`

Disabling Automatic Header Generation

To disable Automatic Header Generation completely for a request, call the APPSRVSET DATA step function, as follows:

```
data _NULL_;
   rc = appsrvset("automatic headers", 0);
run;
```

HTTP Output Reference

All Dispatcher output must be in the format of an HTTP header that is followed by a blank line and optional data. This section provides introductory technical information on the most common headers you will use. For detailed information about HTTP, see W3C's HTTP Protocol Area at www.w3.org/Protocols.

- Content−type
- Expires
- Location
- Pragma
- Set−Cookie

Content−type

The most basic HTTP header you can send is the content−type header, for example:

```
Content-type: text/html
```

If you use ODS to generate content that is not HTML, then the header will be defined based on information from the SAS registry or the Windows registry. For example, if you use ODS PDF to generate content, the header will look like

```
Content-type: Application/PDF
```

This informs the browser what kind of output follows by specifying it the Internet Media type (also called MIME type). An unregistered MIME type may be used; just precede it with `x−`. Some of the more important types are listed in the table below.

Content−type	Description
application/octet−stream	Unformatted binary data.
image/gif	Image in the GIF (Graphics Interchange Format) format.
image/jpeg	Image in the JPEG (Joint Photographic Expert Group) format.
text/html	Regular HTML (Hypertext Markup Language).
text/plain	Preformatted text.
text/x−comma−separated−values	Spreadsheet data.
multipart/x−mixed−replace	Differently formatted blocks of data (used for Netscape server push).

Expires

Sometimes browsers cache results when you intend for the Dispatcher to be reinvoked, and sometimes they reinvoke when it is unnecessary. Setting the Expires header gives you control over these conditions by specifying the date/time after which the response should be considered stale, for example:

```
Expires: Thu, 01 Dec 1994 16:00:00 GMT
```

To mark a response as *already expired*, use an Expires date that is equal to or earlier than the current date. To mark a response as *never expires*, use an Expires date approximately one year or more from the time the response is sent. The date format should be followed exactly as given above.

Location

The location header redirects the browser immediately to a different URL. Use this as an alternative to the content−type header. There is no data after a header containing `Location:` but you still need the blank line at the end.

```
Location: http://support.sas.com
```

Pragma

This header informs the browser and proxy servers to not cache the results of your program. It is similar to using the Expires: header with a date in the past but may be somewhat better supported, for example:

```
Pragma: nocache
```

Set−Cookie

The header sends a cookie to the browser to maintain the client−side state. The format is

```
Set-Cookie: NAME=VALUE; expires=DATE; path=PATH; domain=DOMAIN_NAME; secure
```

For example:

```
Set-Cookie: CUSTOMER=WILE_E_COYOTE; path=/cgi-bin/broker;
   expires=Wednesday, 09-Nov-1999 23:12:40 GMT
```

The next time your application is run, any matching cookies are returned in HTTP_COOKIE environment variable (use Export directive to pass to application). You must parse them out in order to retrieve the information that you save. The names and values can be anything you like, but you must devise a method to encode special characters such as the equals sign (=) and the semicolon (;). The date format should be followed exactly as above, with only the GMT time zone allowed, and dashes between the day, month, and year (this is different from Expires:).

Most new browsers support cookies, but studies show that approximately 10% of users disable or disallow them. Some users are concerned about the privacy considerations of using cookies. If you use cookies, be sure to explain to your users why you need them and that they should let them pass through.

Using HTML Formatting Tools

The HTML Formatting Tools are often used to produce Dispatcher program output. There are some important guidelines to follow when using the Formatting Tools in a Dispatcher program.

- When using the Output Formatter and Tabulate Formatter mechanism, do not forget to turn capturing off before the program completes. Each `capture=on` needs to have a corresponding `capture=off`. The Application Server is not guaranteed to turn off these tools for you.
- The listing output destination must be active or these two tools will not work. The listing destination is active by default when you start SAS, and it is also turned on in the default server reset file. However, it is possible that the listing could be closed if you remove the ODS statement from your reset file and explicitly turn it off in one of your programs. If you need to turn the listing on, then submit the following before you invoke the `capture=on` mechanism for either the Tabulate or the Output Formatter:

  ```
  ods listing;
  ```
- Use RUNMODE=S with each of the tools. This RUNMODE value designates that the tools are running in server mode. If `openmode=replace` is used along with this run mode, then the tools will generate a `Content-type: text/html` header automatically.
- Use `openmode=replace` if this call to one of the tools will be creating the first output from this program. This open mode will cause the formatting tools to generate the header (if RUNMODE=S), the HTML head section of the document, and body tags.
- Use `openmode=append` if this call to one of the tools will **not** be creating the first output from this program. If the program has already produced some output, then it has already supplied the content−type header and most likely the HTML head section, as well.

The Application Dispatcher is a very flexible programming environment because it provides procedures and tools that automatically generate output, but it also allows exact control of the output. Many SAS programmers have encountered the situation where they want to generate completely customized output. This is often done by using the DATA step and PUT statements. The Dispatcher technology supports PUT statement reporting and allows you to supplement such reports with powerful procedures and Formatting Tools.

The Output Delivery System (ODS)

The Output Delivery System (ODS) enables SAS procedures to generate output in several different formats. One of these output formats is HTML. You can use ODS in your Dispatcher programs to easily create Web pages containing HTML and graphics. This page discusses features and options of ODS that are appropriate for the Application Dispatcher environment. ODS can be used in other SAS environments and can generate other forms of output. For more information about ODS, refer to the *SAS Output Delivery System: User's Guide*.

Creating Web Output with ODS

HTML output is enabled with the ODS HTML statement. The ODS HTML statement can create

- an HTML file (called the body file) that contains the results from the procedures run in your Application Dispatcher program
- a table of contents that links to the body file
- a table of pages that links to the body file
- a frameset that displays the table of contents, the table of pages, and the body file.

ODS may also generate additional HTML or image files if you split the output across multiple body pages or you use embedded GIF or JPEG images to display graphics.

The HTTP protocol used by the Application Dispatcher can deliver only one output file back to the browser per request. This output file is written to the _WEBOUT fileref. Because ODS generates multiple output files in many cases, the extra files must be stored in a temporary location and retrieved by the browser in subsequent requests. The Application Server automatically creates a unique temporary catalog for every request for this purpose. The two−level catalog name is defined in the special macro variable _TMPCAT. ODS must also put hyperlinks and inline image links into the HTML that it generates that will retrieve the files from the temporary catalog. The special macro variable _REPLAY contains the base URL used to create these links.

To enable the above features, any Dispatcher program that uses ODS to generate HTML should include the following options on the ODS HTML statement:

```
ods html path=&_tmpcat (url=&_replay) rs=none ...;
```

Note: The RS=none option forces ODS to perform record based output and is required when writing to the _WEBOUT fileref or to a catalog entry.

ODS is capable of creating a number of different layouts for your output. All layouts have one thing in common: the "primary" page must be returned directly to your browser (via the _WEBOUT fileref). The page written to the _WEBOUT fileref must be preceded by an HTTP header with the appropriate content−type field.

Starting with Release 8.2, the automatic HTTP header generation feature recognizes some ODS output types and generates appropriate content−type headers. Supported output types include HTML, GIF, and JPEG. An appropriate content type must be manually set with APPSRV_HEADER function for all other output types.

If you are writing to _WEBOUT using PUT statements while ODS has _WEBOUT open, when you execute the code the PUT statement data might be out of sequence with the data generated by ODS. This problem occurs because both your code and ODS are opening the same fileref at the same time. This problem can be fixed by inserting your PUT statements before you open ODS, closing ODS while you write directly to the fileref, or using the `ODS HTML TEXT="string"` option to write data. The following code is an example of how you can use the ODS HTML TEXT="string" option to write data:

```
ods listing close;
ods html body=_webout path=&_tmpcat
   (url=&_replay) Style=Banker;
... other code ...
ods html text='<p align="center"> </p>' ;
ods html text='<p align="center"><b>Test.
   If you see this in order, it worked.</b></p>';
... other code ...
ods html close;
```

Layout Examples

The following annotated examples illustrate how to return various ODS layouts to your browser. For these examples, we will use the following data set:

```
data stocks;
length symbol $4 price 8.;
input @1 symbol price;
label symbol = 'Symbol'
   price  = 'Share Price';
format price 7.2;
cards;
AMD   23.50
BORL   9.31
CA    47.25
CPQ   32.06
DELL 139.88
GTW   44.00
HWP   67.00
IBM  104.44
INTC  89.69
MSFT  84.75
ORCL  24.63
SUNW  47.63
;
run;
```

The examples are

- Body Only
- Body and Table of Contents
- Table of Contents Only
- Graphics and Text.

Body Only

When you return only the body output to your browser, the page will be rendered as a single, unframed page. Sample code to produce such output is shown below.

```
ods listing close;
ods html body=_webout
      path=&_tmpcat (url=&_replay) rs=none;
   title 'Stock Prices';
   proc print data=stocks label noobs; run; quit;
ods html close;
```

Because the body file is the only file created, it is the primary file and is directed to the fileref _WEBOUT.

Body and Table of Contents

You can return the Table of Contents (or Table of Pages) and the body file using ODS framed output. In this case, the file created by the ODS HTML FRAME option is the primary file, and it must be directed to _WEBOUT. Other files will be stored in the temporary catalog in the WORK library and will be replayed automatically at the proper time.

The sample code below illustrates the creation of a Table of Contents in addition to the body file.

```
ods listing close;
ods html frame=_webout
      body=b.html
      contents=c.html
      path=&_tmpcat (url=&_replay)
      rs=none charset=' ';
   title 'Stock Prices';
   proc print data=stocks label noobs; run; quit;
   proc contents data=stocks; run; quit;
ods html close;
```

Note: The `charset=' '` option eliminates the CHARSET value in the <META> tag. For more information about using the CHARSET option, see FAQ #3935 at support.sas.com/faq.

The body and contents files are directed to the temporary catalog and will be named B.HTML and C.HTML, respectively. You can choose any valid SAS name for the entries, but the object type *must* be HTML. Do not enclose the name in quotes or it will be interpreted to be an external file rather than a catalog entry.

Table of Contents Only

You may want to return only the Table of Contents to your browser to avoid using HTML frames. The page will be rendered as a single, unframed page. Links on the Table of Contents page will allow you to load body output to the browser. A simple modification to the previous example will drop the FRAME keyword and make the contents file the primary file returned to _WEBOUT.

```
ods listing close;
ods html contents=_webout
      body=b.html
      path=&_tmpcat (url=&_replay)
      rs=none charset=' ';
   title 'Stock Prices';
   proc print data=stocks label noobs; run; quit;
   proc contents data=stocks; run; quit;
ods html close;
```

Note: The `charset=' '` option eliminates the CHARSET value in the <META> tag. For more information about using the CHARSET option, see FAQ #3935 at support.sas.com/faq.

When you click on an item in the Table of Contents, the body file will be replayed from the temporary WORK catalog.

Graphics and Text

The code below will return framed output consisting of the Table of Contents and the body, which contains integrated graphics and text. Note that no special ODS keywords were required to create and store the graphic images. This example is essentially the same as the Body and Table of Contents example with some SAS/GRAPH code added.

```
ods listing close;
ods html frame=_webout
      body=b.html
      contents=c.html
      path=&_tmpcat (url=&_replay)
      rs=none charset=' ';
   goptions reset=all;
   goptions device=gif
      colors=(red orange yellow ligr green blue)
      ctext=black cback=white;
   title 'Stock Prices';
   axis1 major=none minor=none value=none;
   proc gchart data=stocks;
      hbar3d symbol / sumvar=price
            subgroup=symbol
            shape=cylinder
            patternid=subgroup
            raxis=axis1;
      run; quit;
   title;
   proc print data=stocks label noobs; run; quit;
   proc contents data=stocks; run; quit;
ods html close;
```

Note: The `charset=' '` option eliminates the CHARSET value in the <META> tag. For more information about using the CHARSET option, see FAQ #3935 at support.sas.com/faq.

Cleaning Up

HTML and graphics created by ODS in the _TMPCAT catalog must eventually be deleted. The Application Server will handle this task automatically. By default, a temporary catalog will be deleted if it is not used for a period of 15 minutes. This timeout value can be changed with the SESSION TIMEOUT=*seconds* option on PROC APPSRV, or with the APPSRV_SET('session timeout',*seconds*) DATA step function. All temporary catalogs are deleted immediately when a server is stopped.

Using the REPLAY Program

The REPLAY program replays an existing catalog entry to the Web browser. This program is used with Dispatcher programs that invoke the Output Delivery System (ODS) and with other programs that create output and retrieve it for display later. For a more complete description of ODS, see Using the Output Delivery System (ODS).

A sample invocation of REPLAY would look like

```
http://d5220.us.sas.com/scripts/broker.exe?_service=default
   &_program=replay&_entry=SAMPDAT.WEBSAMP.RETAIL.HTML
```

The user must specify _SERVICE, _PROGRAM=REPLAY, and _ENTRY=*lib.cat.entry.type*.

Advanced Programming Techniques

The techniques described in this section will help you to expand the capabilities of your Dispatcher applications. It is a good idea to review the basic programming techniques that are described in The Four Types of Programs before continuing with this section.

- Data Passing and Program Chaining
- Embedded Graphics
- Browser Referral by Using the Location Header
- Creating Various Date/Time Formats

Data Passing and Program Chaining

Only the simplest Dispatcher applications contain a single page. With the addition of a second and subsequent pages, you face the problem of passing information from one page to another. It is also typical to have an application that contains more than a single program. This means that you must find a way to connect the programs that compose your application and make sure that all the data collected along the way is available in the appropriate places.

It is good programming practice to design applications so that they do not request the same information multiple times. Because HTTP is a stateless environment, each program request is separate from all other requests. If users enter a phone number on the first page of an application and submit the form, that phone number is available only to the first program. But after that program completes, the state of the data values passed is lost. If the third program in the application needs to know the specified phone number, the application must ask for the phone number again or retrieve the data from a stored location. There are several ways to solve this problem. You can store data values

- on the client, in hidden form fields
- on the client, in cookies or Web page scripts
- on the server.

Storing data on the client, in hidden fields, is the simplest technique. To do this, you must dynamically generate all of the HTML pages in your application except for the first HTML page. Because each HTML page functions as a mechanism for transporting data values from the previous program to the next program, it cannot be static HTML stored in a file.

Usually, the process involves the following steps:

1. The first HTML form calls the first program.
2. The first program performs some kind of setup to initialize the user.
3. At the end of the first program, the second HTML page is created by writing to _WEBOUT.
4. When the HTML form on the second page is written out, you dynamically generate a series of HTML fields by using the TYPE="hidden" attribute.

Each hidden field in the second form can contain one name/value data pair passed from the first form. You should use unique names for all of the data values in the entire application. In this way you can pass all of the application data throughout the entire application.

At the same time that you dynamically generate the second form, you can write out the name of the second program in the hidden field _PROGRAM. Because the first program contains the logic to determine the second program, this is referred to as program chaining. Your application may have multiple second programs. The logic in the first program can decide which second program the current user should run.

Here is an example.

First HTML Form

```
<FORM ACTION="/cgi-bin/broker">
Please enter your first name:
<INPUT TYPE="text" NAME="fname"><BR>
<INPUT TYPE="hidden" NAME="_service" VALUE="default">
<INPUT TYPE="hidden" NAME="_program" VALUE="mylib.pgm1.sas">
<INPUT TYPE="submit" VALUE="Run Program">
</FORM>
```

This form passes the first name of the user as the variable FNAME to the program named PGM1.SAS in the program library MYLIB.

First Program (PGM1.SAS)

```
data _null_;
   file _webout;
   put 'Content-type: text/html';
   put;
   put '<HTML>';

      /*create reference to the broker from
      special automatic macro variable _url*/
   url=symget('_url');
   put '<FORM ACTION="' url +(-1) '">';

      /*supply service name*/
   service=symget('_service');
   put '<INPUT TYPE="hidden" NAME="_service" VALUE="'
   service +(-1) '">';

      /*use current program library so that this
      application can be easily moved to another library*/
   pgmlib=symget('_pgmlib');
   program=compress(pgmlib)||'.pgm2.sas';
   put '<INPUT TYPE="hidden" NAME="_program" VALUE="'
   program +(-1) '">';

      /*pass first name value on to next program*/
   fname=symget('fname');
   put '<INPUT TYPE="hidden" NAME="fname" VALUE="'
   fname +(-1) '">';

   put 'What is your favorite color?';
   put '<SELECT SIZE=1 NAME="fcolor">';
   put '<OPTION VALUE="red">red';
   put '<OPTION VALUE="green">green';
   put '<OPTION VALUE="blue">blue';
   put '<OPTION VALUE="other">other';
   put '</SELECT><BR>';
   put '<INPUT TYPE="submit" VALUE="Run Program">';
   put '</FORM>';
   put '</HTML>';
run;
```

This program uses the special variables _URL, _SERVICE, and _PGMLIB to maintain program portability. The second program name PGM2.SAS is hard−coded. The important section of this program is where the variable FNAME is received by calling the SYMGET function and written out as a hidden form variable. This is the key step

that enables the data value to "live" beyond the stateless execution of this first program. In addition to inserting the hidden data value, this program generates a selection list that asks the user to enter a favorite color.

Second Program (PGM2.SAS)

```
data _null_;
   file _webout;
   put 'Content-type: text/html';
   put;
   put '<HTML>';

      /*extract first name and favorite
   color and print them out*/
   fname=symget('fname');
   fcolor=symget('fcolor');
   put 'Your first name is <b>' fname '</b>';
   put '<BR>';
   put 'Your favorite color is <b>' fcolor '</b>';
   put '<BR>';
   put '</HTML>';
run;
```

The second program prints the value of the variables from both the first and second form, illustrating that the data has been correctly passed throughout the entire application. The technique of passing data by using hidden fields has these advantages:

- simple to do
- easy to debug
- state is maintained indefinitely
- works seamlessly across multiple Application Servers.

The major disadvantage of this technique is that it is easy for a user to change the values in the form and submit incorrect or falsified information to the application. Another technique that is nearly equivalent to using hidden fields is to pass name/value pair data as part of the query string in a hyperlink. In the Second Program, (PGM2.SAS), suppose the initial forms were the same, but you want the second page to contain a list of hyperlinks instead of a selection list. In this case, you would generate a list of hyperlinks by using the anchor tag, and the HTML source would look like this:

```
<A HREF="path to Application Broker plus data">Red</a>
<A HREF="path to Application Broker plus data">Green</a>
<A HREF="path to Application Broker plus data">Blue</a>
<A HREF="path to Application Broker plus data">Other</a>
```

Using this example, the *path to Application Broker plus data* would consist of the URL for the Application Broker, which is stored in the special variable _URL followed by all of the name/value pair data that should be passed from the first program to the second program. This data includes the required fields _SERVICE and _PROGRAM. At least one additional parameter is added to each hyperlink that will be used to indicate which link is chosen. In this case, that additional field is COLOR. To use hyperlinks instead of an HTML form that has a select list, the first program must change to PGM1.SAS.

Modified Version of First Program (PGM1.SAS)

```
data _null_;
   file _webout;
   put 'Content-type: text/html';
   put;
```

```
    put '<HTML>';

       /*store broker path in data step variable*/
    url=symget('_url');

       /*store current service name*/
    service=symget('_service');

       /*use current program library so that this
       application can be easily moved to another library*/
    pgmlib=symget('_pgmlib');
    program=compress(pgmlib)||'.pgm2.sas';

       /*pass first name value on to next program*/
    fname=symget('fname');

       /*build partial URL, color will be added later*/
    href=trim(left(url))||'?_service='||trim(left(service))||
       '&_program='||trim(left(program))||
       '&fname='||urlencode(trim(left(fname)));

    put '<HTML>';
    put 'What is your favorite color?<br>';
    put '<A HREF="' href +(-1) '&color=red">red</a><br>';
    put '<A HREF="' href +(-1) '&color=green">green</a><br>';
    put '<A HREF="' href +(-1) '&color=blue">blue</a><br>';
    put '<A HREF="' href +(-1) '&color=other">other</a><br>';
    put '</HTML>';
run;
```

This modified version uses a special function named URLENCODE in this program. The purpose of this function is to encode any special characters that may be contained in the query string. Because the values for _PROGRAM and _SERVICE do not contain any special characters, it is not necessary to encode them.

However, the value that the user supplies for a first name may contain some special characters. For the First Program, (PGM1.SAS) to pass this value safely to the second program, (PGM2.SAS) it should be URL−encoded. It does not harm a value to URL−encode it even when it does not contain special characters. The URLENCODE function was not used because if data is passed through an HTML form, then the Web browser would perform the encoding for you. Aside from the need to URL−encode data and the different HTML syntax, the use of a hyperlink and hidden form fields are essentially the same.

Two alternatives to passing the data throughout every form in the application are: storing the data in a Web browser cookie or storing data within the Application Server environment. Both of these techniques are more difficult than using hidden form fields, but they have different advantages.

HTTP cookies are packets of information that are stored in the client Web browser. They are shuttled back and forth with the CGI requests. In this general sense, they are quite similar to hidden form fields. Cookies have the advantage of being nearly invisible to the user. They contain a built−in expiration mechanism, and they are slightly more secure than hidden fields. They also work seamlessly across multiple Application Servers.

Storing data within the Application Server environment is a tempting way to solve the problem of passing data. You can create a data set and assign each user a unique key variable. All the data collected for that user can be stored in the data set within the Application Server environment. This is a much better mechanism for applications that have a large volume of data to be collected and passed from program−to−program. The key variable still needs to be passed along by the Web browser, and that can be done by using cookies or a hidden form field. If the key is unique and sufficiently difficult to guess, then this data passing mechanism also has an added level of security. The advantages to this technique are

- it is easier if there is a large volume of data needs to be passed
- it is nearly invisible to the user
- it is a significant security improvement
- it reduces duplication of program code.

You may experience contention problems if your service contains multiple servers and all servers try to update the same data set that contains the user data. Using a SAS/SHARE data server to read and write to the data set can overcome this problem.

Embedded Graphics

The typical Web page contains embedded graphic images. This is easy to do in static pages. To your static HTML, add an IMG tag, for example:

```
<IMG SRC="mykids.gif">
```

In a dynamic environment such as the Application Dispatcher, the value for the SRC parameter must be a URL that invokes the Application Broker. For example, inserting the following HTML code in a static HTML page causes the sample graphics program to be run when the browser loads the image:

```
<IMG SRC="/cgi-bin/broker?_service=default&_program=sample.webgraph.sas">
```

Programs called from an IMG tag must respond with a content type of image/gif or image/jpeg. You can add multiple parameters to this type of URL to make your graphics program output more flexible and customizable. Each dynamic image tag in your HTML page represents a separate request to the Dispatcher and a separate program execution. It is important to keep that in mind when you design your application. You may want to show only one or just a few graphics on each page. That will reduce the demand on your Application Server.

Sometimes, you may be generating a dynamic HTML page, and you want that dynamic page to contain one or more embedded graphics that are also dynamic. One way to accomplish this is to use the Output Delivery System (ODS). Using ODS is convenient, because it lets you run procedures that produce HTML and graphics within a single Dispatcher program.

If you choose not to use ODS, you must have separate programs for producing HTML and graphics. The first program that is called produces the HTML page by writing to _WEBOUT. At the appropriate place in the HTML source code, the first program writes an image tag that calls the second program −− the program that produces the graphics. By using the concepts outlined in Data Passing and Program Chaining, the first program passes any name/value pair data to the second program. Because the image tag is not an HTML form, you cannot use hidden fields to pass the name/value pairs. You must encode them in the query string of the URL that you generate for the SRC parameter.

Most of the time your embedded graphics represent some underlying data. If that underlying data changes, you expect the browser to display a new picture.

Note: Unfortunately, there is a serious defect in some browsers that prevents the new graphic image from being displayed. This is not a defect in SAS/IntrNet software. Browsers exhibiting this defect will load the URL, that causes the Dispatcher program to execute and deliver a new image; but the old, cached image is displayed. When the user arrives at the page by selecting a hyperlink, a favorite, or a bookmark, or submits a form, an old image may be seen. By clicking REFRESH or RELOAD in the browser the new image will display. Unfortunately, sending the graphic image by using the Expires or the Pragma: no−cache HTTP headers does not fix this problem.

One solution to this problem is to add a caption to your embedded graphics that tells the user to reload the page for the most up−to−date graphic. If your data does not change too quickly, you may not need this caption. The best solution is

to trick the browser into not using a cached image. You can do this by generating a unique URL every time the program is executed. Because HTML files and images are cached according to their URLs, the browser will never have an old image that matches the unique URL that the program generates.

To create a unique URL, generate the SRC= URL string that you need and append to the end of the URL a name/value pair that has a value of the current SAS datetime, as shown in this example.

```
data _null_;
   file _webout;

      /*store broker path in data step variable*/
   url=symget('_url');

      /*store current service name*/
   service=symget('_service');

      /*use current program library so that this
      application can be easily moved to another library*/
   pgmlib=symget('_pgmlib');
   program=compress(pgmlib)||'.graphit.sas';

      /*create partial URL*/
   src=trim(left(url))||'?_service='||trim(left(service))||
      '&_program='||trim(left(program));

      /*the above URL is enough to embed the graphic but a
      unique string must be added to avoid incorrect caching*/
   nocache=datetime();
   src=src||'&nocache='||trim(left(nocache));

      /*write out image tag*/
   put '<IMG SRC="' src +(-1) '">';
run;
```

Each time this code is executed, a different datetime value is returned. This process results in a unique URL. The NOCACHE parameter is not used by the graphics program because its only purpose is to trick the browser into not using its cache.

Note: On some systems it may be better to call one of the SAS random functions instead of datetime. If your system is fast and you make repeated, close calls to the datetime function, it is possible to get the same value returned.

Browser Referral by Using the Location Header

Another advanced programming technique is to have the output from one Dispatcher program invoke another Dispatcher program without displaying a page from the first program. Suppose that you have a program named PGM1.SAS, and it performs some error checking of the form data that is sent as input. If your program detects an error condition, you want to run some additional code. The additional code is contained in another program file named ERROR.SAS. Instead of copying the code from ERROR.SAS into PGM1.SAS and having to maintain two pieces of identical code, you can invoke the error program via the output of PGM1.SAS. This is another form of program chaining.

The HTTP header Location is a special header that the browser recognizes. This header re–directs the browser to another Web page. The only parameter to this header line is the URL that the browser should load. In your first program, you can dynamically construct this URL and refer the browser to another Dispatcher program. The URL that you supply by using the location header should be fully qualified and can contain any additional name/value pair data that you want to send to the second program shown as shown in this example code.

```
data _null_;
   if error>0 then do;
      /*construct referral URL*/
   file _webout;

      /*because URL is fully qualified get name of the Web server
      from automatic exported variable*/
   srvname=symget('_srvname');

      /*store broker path in data step variable
      _url is a special automatic variable*/
   url=symget('_url');

      /*store current service name
      _service is a special automatic variable*/
   service=symget('_service');

      /*use current program library so that this
      application can be easily moved to another library
      _pgmlib is a special automatic variable*/
   pgmlib=symget('_pgmlib');
   program=compress(pgmlib)||'.error.sas';

      /*create fully qualified URL*/
   loc='http://'||trim(left(srvname))||
      trim(left(url))||'?_service='||
      trim(left(service))||
      '&_program='||trim(left(program));

   /*put out location header instead of content-type header
   make sure to include null line to terminate HTTP header
   no content needed after location header because browser will
   refer to another page*/
   put 'Location: ' loc;
   put ;
end;
run;
```

It is also possible to perform browser referrals by using the <META> tag in the generated HTML page.

Creating Various Date/Time Formats

Occasionally, you may find the need to create various specialized date/time formats as part of your Dispatcher output. These formats may not be compatible with the standard set of SAS date/time formats that require you to create the format yourself. The most common format is the EXPIRES format. This format is used to expire HTTP cookies and pages that are cached in the Web browser. The DATATYPE option in the FORMAT procedure allows you to create specialized date/time formats easily, such the EXPIRES format. By specifying DATATYPE= in the PICTURE statement, you can use a special set of codes to represent both numerical and textual components of the date, the time, or the DATETIME value that you are formatting. Here is an example of how you can use this feature to create DATETIME format that expires:

```
proc format;
   picture expires other='%A, %d-%b-%y %0H:%0M:%0S GMT' (DATATYPE=DATETIME);
run;
```

The special codes in the OTHER= parameter represent locale full weekday name, day of the month, locale abbreviated month name and so on. See the documentation on the FORMAT procedure in the SAS Procedures Guide for more information. The test program below illustrates how to use the DATETIME format.

```
data _null_;
   x=datetime();
      /*adjust local datetime to GMT by adding five hours*/
   x=x+5*3600;
   put x expires33.;
run;
```

This program produces the following output:

```
Tuesday, 29-SEP-98 14:39:29 GMT
```

You can use this formatted output to create Expires headers or to set expiration dates and times for HTTP cookies.

Creating Temporary Files

The Dispatcher program creates a temporary file for each request. When multiple requests are run at the same time, one request may write over the temporary file of another. There are several ways to create a unique file for the life of a request so that files do not interfere with each other:

- Creating a File with a Unique Name
- Creating a File in a Unique Subdirectory
- Storing a File in a Unique Catalog

Creating a File with a Unique Name

To create a uniquely named file, use a random function to generate the filename. Then, use the name in a FILENAME statement:

```
/* Use a data step function RANUNI to generate a random number
and format the number with some text to create the unique name.
The RAN macro variable will be something like 'A2578900.txt'  */

data _null_;
   numb = ceil (ranuni(0)*10000000);
   r = 'A' || put(numb, Z7.) || '.txt';
   call symput ('ran', r);
run;

filename foo "c:\temp\&ran";
/* prepend the temp subdirectory */

data _null_;
   file foo;
   put 'This is output to a uniquely named file';
run;
```

This technique requires that you explicitly delete the file when you are finished.

Creating a File in a Unique Subdirectory

You can avoid file interference by using a unique subdirectory. Beginning with Version 8 of SAS/IntrNet: Application Dispatcher, the WORK library is unique for each request. Find the path to the WORK library and use it as the subdirectory to store the temporary file.

Note: To append text to a macro variable reference, use the period (.) operator.

```
/* use pathname function to get work library's path and store
in macro variable wpath */

%let wpath=%sysfunc(pathname(work));

/* use wpath macro variable and append filename to it in
filename statement  */

filename foo "&wpath.\winapi.txt";

data _null_;
file foo;
put 'This output is going to file in the work subdirectory';
```

```
run;
```

If the SESSIONS feature is used, then you can specify the path of the SAVE library:

```
%let spath=%sysfunc(pathname(save));
filename foo "&spath.\winapi.txt";
```

This technique deletes the file automatically at the end of the request.

Storing a File in a Unique Catalog

You can create a unique file for each request by storing the file in a unique SAS catalog. Text can be easily stored in a catalog SOURCE memtype by using the CATALOG access method.

```
filename foo catalog "work.mytext.foo.source";
data _null_;
file foo;
put 'This output is going to file in the work catalog';
run;
```

Example

You can use any one of these techniques in a Dispatcher program to create unique files for each request. To put it all together, here is a Dispatcher program that reports the username for the current process:

```
/*************************************************************/
/*           S A S   S A M P L E   L I B R A R Y             */
/*                                                           */
/*     NAME: HelloWorld with a twist                         */
/*    TITLE: Hello World                                     */
/*  PRODUCT: SAS/IntrNet (Application Dispatcher)            */
/*   SYSTEM: ALL                                             */
/*     KEYS:                                                 */
/*    PROCS:                                                 */
/*     DATA:                                                 */
/*                                                           */
/*  SUPPORT: Web Tools Group             UPDATE: 20JAN1999   */
/*      REF: http://support.sas.com/rnd/web/intrnet/dispatch/*/
/*     MISC:                                                 */
/*************************************************************/

/* create macro variable containing the path to the work subdirectory  */
%let wpath=%sysfunc(pathname(work));
%put &wpath;

filename sascbtbl "&wpath.\winapi.txt";

/* write WINAPI GetUserNameA parameter list to the file for later use */

data _null_;
file sascbtbl;
input line $char80.;
put line $char80.;
cards4;
routine GetUserNameA
minarg=2
maxarg=2
stackpop=called
```

```
module=advapi32
returns=short;
arg 1 char update format=$cstr20;
arg 2 num  update format=pib4.;
;;;;
run;

/* Here's the DATA step: Modulen will use the filename sascbtbl for
parameter list to the called API  */

data _null_;
   length Name $20.;
   name='';
   Size=20;
   rc=modulen('GetUserNameA',Name,Size);
   put rc= Name=;

/* Store the current process username in the macro named vqpname  */

   call symput('vqpname',name);
run;

/*simply write out a Web page that says "Hello World!"*/

/* use a DATA step variable and a macro variable in displayed text  */

data _null_;

/* store macro variable value in data set variable named vname  */

vname=symget('vqpname');
   file _webout;
   put '<HTML>';

/*  Use data set variable in the output   */

   put '<HEAD><TITLE>Hello World!' vname ' is running dispatcher!
      </TITLE></HEAD>';
   put '<BODY>';

/*  Use macro variable in the output   */

   put "<H1>Hello World!  Your work path is &wpath.
          vname is &vqpname.</H1>";
   put '</BODY>';
   put '</HTML>';
run;
```

Sessions

The Web is a stateless environment. That means that the second request to a server knows nothing of the first request. This creates a simple environment for client/server developers, but it is difficult for application programmers. Often, programmers want to maintain certain information from one request to the next. This is known as *maintaining state*. Sessions provide a convenient way to maintain state across multiple requests.

A *session* is the data that is saved from one program execution to the next. It consists of macro variables and library members (data sets and catalogs) that the user program has explicitly saved. The session data is scoped so that all users have independent sessions. Sessions cannot be shared across multiple Application Servers.

To use this mechanism, the user program must explicitly create a session. This is done with the APPSRV_SESSION function. To create a session you run this code:

```
In macro
%let rc=%sysfunc(appsrv_session(create));

In data step or SCL
rc=appsrv_session('create');
```

Creating a session causes the automatic variables _THISSESSION, _REPLAY, and _SESSIONID to be set with values that reflect the current session id. These variables may be used to construct URLs or HTML forms that run a new request program in the same session.

A session saves all global macro variables whose names begin with SAVE_. For example, the statements

```
%global save_mytext;
%let save_mytext="Text to be saved for the life of the session";
```

cause the macro variable `save_mytext` to be available in later request programs that share the same session.

Data sets and catalogs can also be saved across program requests. Once the session has been created, a library named SAVE is created. By creating or copying data sets and catalogs to this library, the user program can rely on them being there the next time a request is made that uses this session.

Sessions have an expiration time associated with them. Options in the APPSRV procedure set the default and maximum session expiration times. The expiration of an individual session can be set within the maximum time allowed by calling the APPSRVSET function as shown below

```
In macro
%let rc=%sysfunc(appsrvset(session timeout,300));

In DATA step or SCL
rc=appsrvset('session timeout',300);
```

where the number supplied is the number of seconds the session should last beyond the time that it was created. Once the session has expired, the server will delete all session data from memory and from disk.

A session can be explicitly destroyed like this:

```
In macro
%let rc=%sysfunc(appsrv_session(delete));

In DATA step or SCL
rc=appsrv_session('delete');
```

Submitting this code does not immediately destroy the session. The session is only marked for deletion at the time this procedure runs. A session is not deleted until the cleanup routine runs. After the request program completes, the session is placed in the queue to be deleted.

A user will create a session only once throughout an application. The user will reuse the session, but deletion of the session will not occur until the end of the application.

For example, in the example below, a user who tries to create a session and then later delete that session and then try to create a new session in the same test program would get a warning.

```
testa.sas (creates session1 -> calls testb.sas)
testb.sas (uses session1 -> deletes session1 -> creates new session2)
```

The user cannot create session2, because session1 is still being used. Even after a session is marked for deletion, another user cannot access that same session, even before the cleanup process runs.

See Using Sessions: A Sample Web Application for a demonstration of some of the features of Application Dispatcher sessions.

Using Sessions: A Sample Web Application

The following sample Web application demonstrates some of the features of Application Dispatcher sessions. The sample application is an online library. Users can login, select one or more items to check out of the library, and request by e−mail that the selected items be delivered. The sample code shows how to create a session and then create, modify, and view macro variables and data sets in that sesssion.

Sample Data

This sample requires a LIB_INVENTORY data set in the SAMPDAT library that is used for other SAS/IntrNet samples. You can create the data set in Windows using the following code. You can also use the code on other systems by making the appropriate modifications to the SAMPDAT libname statement.

```
libname SAMPDAT '!SASROOT\intrnet\sample';
data SAMPDAT.LIB_INVENTORY;
   length type $10 desc $80;
   input refno 1-5 type 7-16 desc 17-80;
   datalines4;
17834 BOOK      SAS/GRAPH Software: Reference
32345 BOOK      SAS/GRAPH Software: User's Guide
52323 BOOK      SAS Procedures Guide
54337 BOOK      SAS Host Companion for UNIX Environments
35424 BOOK      SAS Host Companion for z/OS Environment
93313 AUDIO     The Zen of SAS
34222 VIDEO     Getting Started with SAS
34223 VIDEO     Introduction to AppDev Studio
34224 VIDEO     Building Web Applications with SAS/IntrNet Software
70001 HARDWARE  Cellphone - Model 5153
70002 HARDWARE  Video Project - Model 79F15
;;;;
```

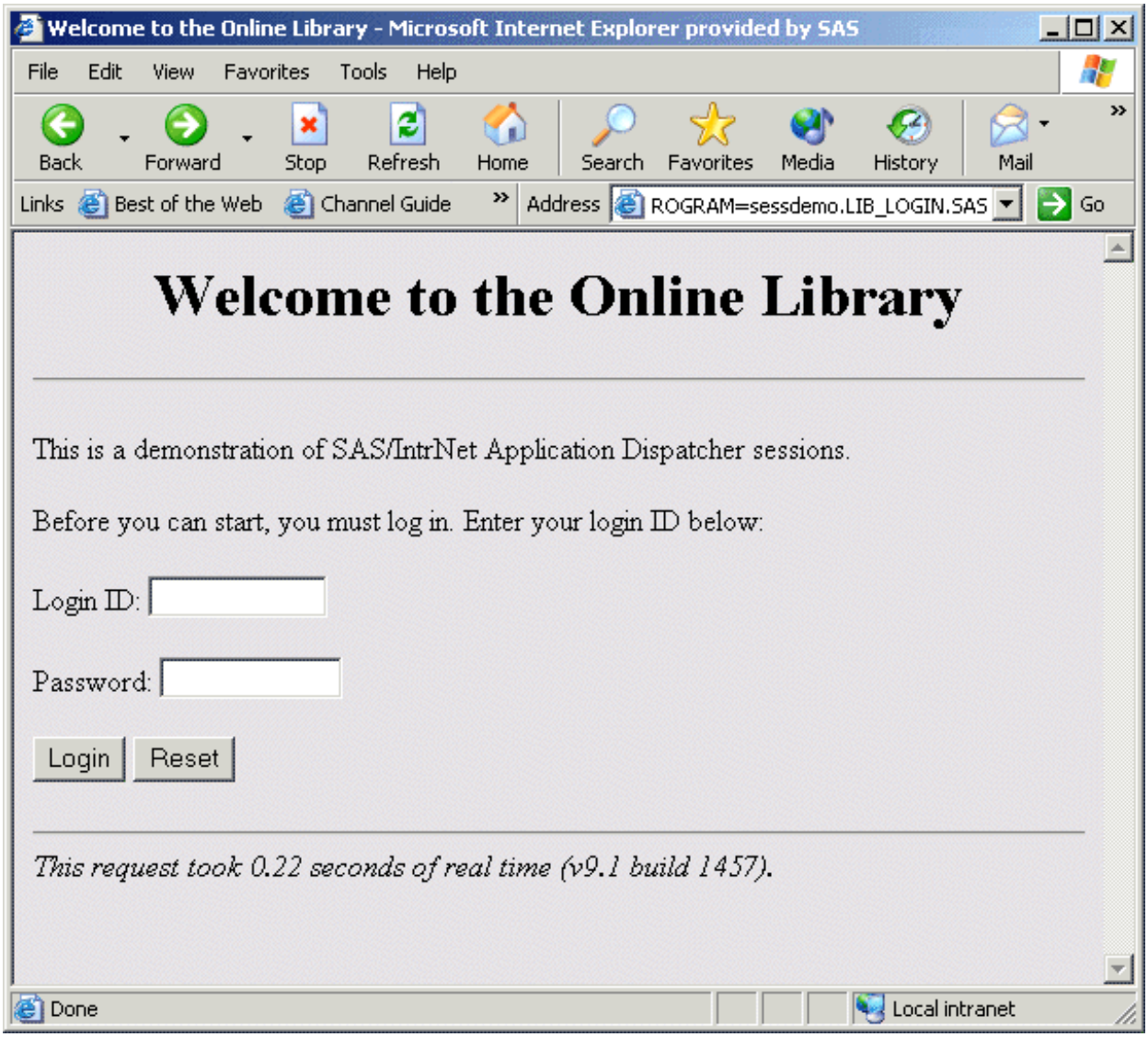
Login

The initial page is the login page. This can be a static HTML page, but in this sample the login page is built at run time. The LIB_LOGIN.SAS program generates the login page as follows:

```
/* LIB_LOGIN.SAS - Welcome to the Online Library */

/* Print a welcome page that is static except for the Application Dispatcher variables
   _URL, _SERVICE, and _PGMLIB. */
data _null_;
   file _webout;
      put '<HEAD>';
      put '<TITLE>Welcome to the Online Library</TITLE>';
      put '</HEAD>';
      put '<BODY vlink="#004488" link="#0066AA" bgcolor="#E0E0E0">';
      put "<H1 ALIGN=CENTER>Welcome to the Online Library</H1>";
      put '<HR>';
      put "<P>This is a demonstration of SAS/IntrNet Application Dispatcher sessions.</P>";
      put '<P>Before you can start, you must log in.  Enter your login ID below: </P>';
      put '<FORM ACTION="' "&_url" '">';
      put '<INPUT TYPE="HIDDEN" NAME="_service" VALUE="' "&_service" '">';
      put '<INPUT TYPE="HIDDEN" NAME="_program" VALUE="' "&_pgmlib" '.lib_main.sas">';
      put 'Login ID: <INPUT SIZE=12 NAME="loginid"><P>';
      put 'Password: <INPUT SIZE=12 TYPE="password" NAME="password"><P>';
      put '<INPUT TYPE="SUBMIT" VALUE="Login">';
      put '<INPUT TYPE="RESET" VALUE="Reset">';
```

```
      put '</FORM>';
      put '</BODY>';
      put '</HTML>';
   run;
```

The LIB_LOGIN.SAS program generates the following page:



The login page is generated dynamically. The FORM ACTION= attribute, the _SERVICE value, and the _PROGRAM library name are all generated at run time based on Application Server macro variables. This enables you to move the application to different Web servers, Application Dispatcher services, or program libraries without editing static HTML. Only the URL for the initial login page changes.

The login page allows the user to enter a login ID and password. Clicking the **Login** button runs the LIB_MAIN.SAS program in order to verify the input data, create a session, and display the Main Aisle page.

Main Aisle

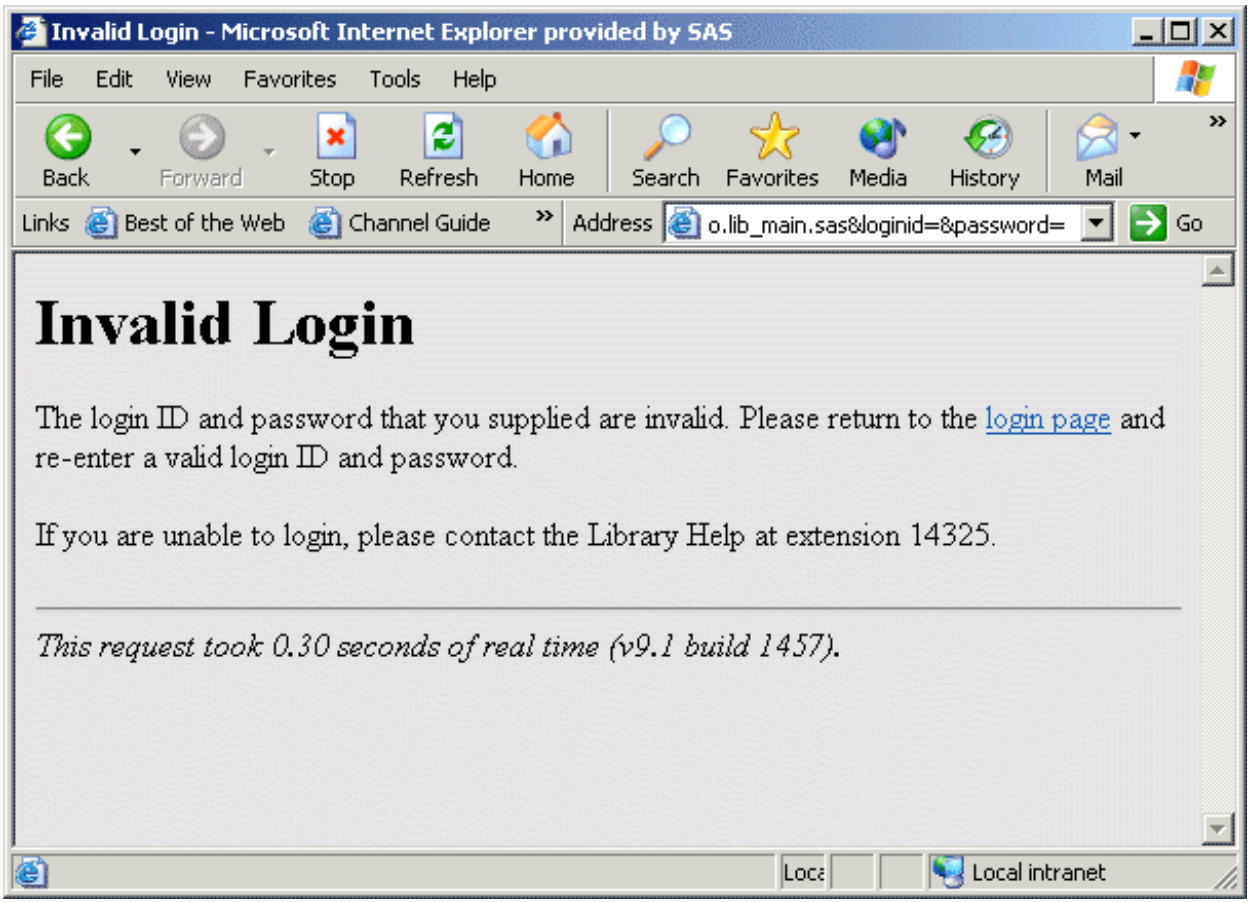
The main aisle page is generated by the LIB_MAIN.SAS program. The LIB_MAIN program also verifies the information that is supplied from the login page.

```
/* LIB_MAIN.SAS - Main Aisle of the Online Library */

/* Use a macro here in order to use conditional logic. */
%macro lib_main;

   /* Check to see if you are already in a session; if so, you don't need
      to validate login info. */
   %if %sysfunc(libref(SAVE)) %then %do;
      /* SAVE libref doesn't exist, so you have not successfully logged in. */

      /* Insert logic here in order to validate the login ID and password.
         For the purposes of this sample, assume that any non-blank password
                     is valid.  This is not usually a good idea; a real application
                     can be expected to insert some real validation logic here. */
      %if &password ne %then %let IDCHECK=PASSED;
      %else %let IDCHECK=FAILED;

      %if &IDCHECK ne PASSED %then %do;

         /* Validation failed - print a failure page. */
         data _null_;
            file _webout;
            put '<HTML>';
            put '<HEAD><TITLE>Invalid Login</TITLE></HEAD>';
            put '<BODY vlink="#004488" link="#0066AA" bgcolor="#E0E0E0">';
            put '<H1>Invalid Login</H1>';
            put;
            put '<P>The login ID and password that you supplied are invalid.  Please';
            put 'return to the <A HREF="' @;
            put "&_URL?_SERVICE=&_service" '&_program=' "&_pgmlib" @;
            put '.lib_login.sas">login page</a> and re-enter a valid login ID';
            put 'and password.</P>';
            put;
            put '<P>If you are unable to login, please contact the Library Help';
            put 'at extension 14325.</P>';
            put '</BODY></HTML>';
            run;
         %end;
         %else %do;
            /* Validation successful - create session and save login ID. */

            %let rc=%sysfunc(appsrv_session(create));

            %global SAVE_LOGINID;         /* SAVE_* variables must be global.  */
            %let SAVE_LOGINID=&loginid;   /* Save the login ID in the session. */
         %end;
      %end;
      %else %let IDCHECK=PASSED;  /* Assume valid login if session already exists. */

      %if &IDCHECK eq PASSED %then %do;

         /* Print the Main Aisle page. */
         data _null_;
            file _webout;
               put '<HTML>';
               put '<HEAD><TITLE>Online Library Main Aisle</TITLE></HEAD>';
               put;
               put '<BODY vlink="#004488" link="#0066AA" bgcolor="#E0E0E0">';
               put '<H1>Online Library Main Aisle</H1>';
               put;
               put 'Select one of the following areas of the library:';
               put '<UL>';
               length hrefroot $400;
```

```
            hrefroot = "%superq(_THISSESSION)" || '&_PROGRAM=' || "&_PGMLIB";
            put '<LI><A HREF="' hrefroot +(-1)
               '.lib_aisle.sas&type=Book">Book Aisle</A></LI>';
            put '<LI><A HREF="' hrefroot +(-1)
               '.lib_aisle.sas&type=Video">Video Aisle</A></LI>';
            put '<LI><A HREF="' hrefroot +(-1)
               '.lib_aisle.sas&type=Audio">Audio Aisle</A></LI>';
            put '<LI><A HREF="' hrefroot +(-1)
               '.lib_aisle.sas&type=Hardware">Hardware Aisle</A></LI>';
            put '<LI><A HREF="' hrefroot +(-1)
               '.lib_cart.sas">View my shopping cart</A></LI>';
            put '<LI><A HREF="' hrefroot +(-1)
               '.lib_logout.sas">Logout</A></LI>';
            put '</UL>';
            put '</BODY>';
            put '</HTML>';
         run;
   %end;
%mend;

%lib_main;
```

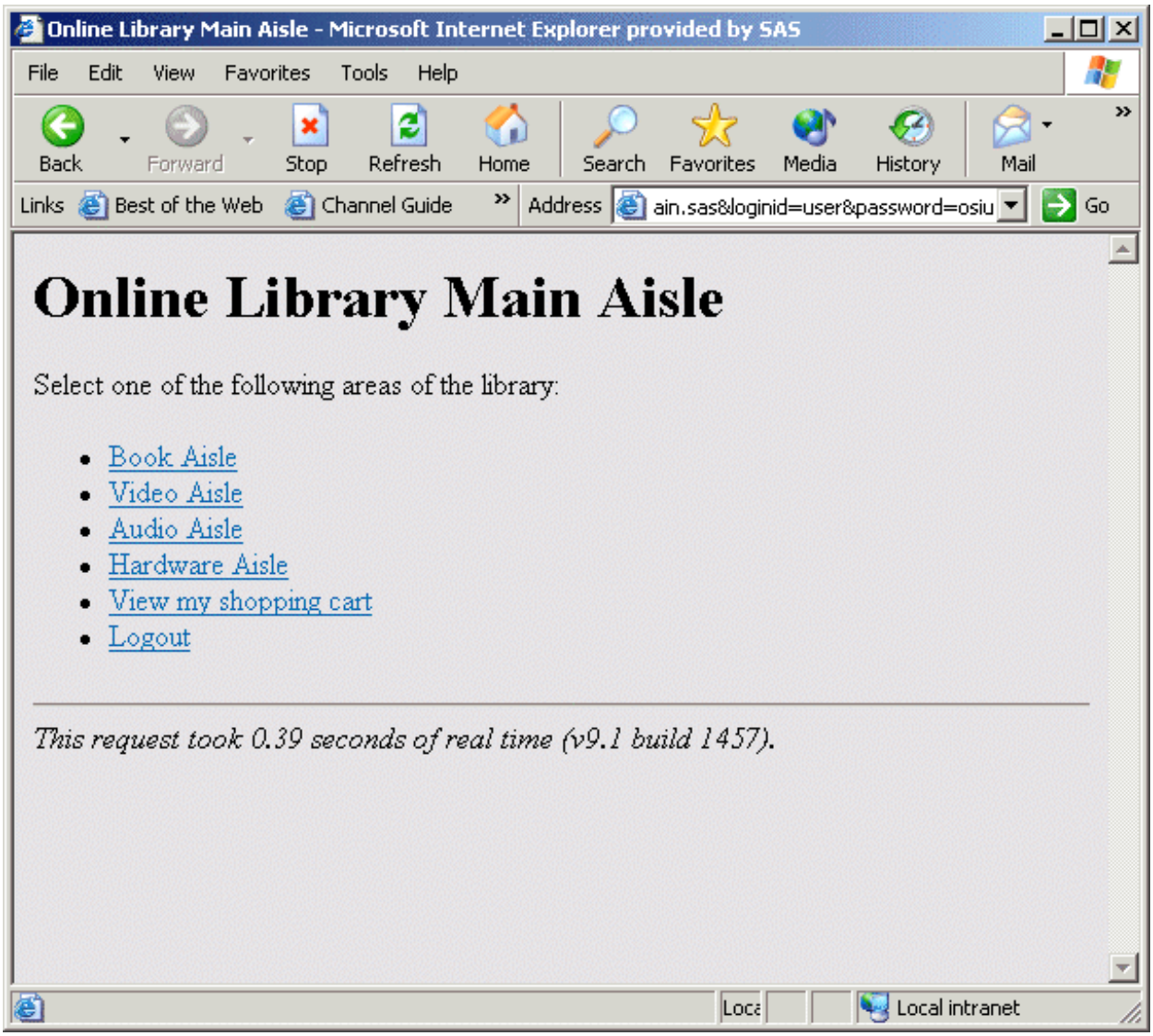
You should display the main aisle page only if the user has logged in; therefore, the first step is to verify user login. First, verify that a session already exists. If the session exists (which is verified by testing whether the SAVE libref exists), then you know that the user has already logged in and you can allow the user to view the main aisle page section of the program.

If the session does not exist, then you can verify valid login information. This sample requires a non−blank login ID and password. If a valid login ID and password are not supplied, the program will display an invalid login page and redirect the user to the login page.



If the supplied login ID and password are valid, the program will create a session and then save the LOGINID value in a session macro variable that is named SAVE_LOGINID. Because the SAVE_LOGINID is declared as a global variable and its name begins with SAVE_, it will be saved for the duration of the session.

After the program verifies that the user has logged in, the main aisle page is displayed. The main aisle page consists of a list of links to specific sections of the Online Library.



Each link in this page is built using the _THISSESSION macro variable. This variable includes all of the values that are necessary in order to run another Dispatcher program in the same session. Use the %SUPERQ function to quote the _THISSESSION variable; this prevents the variable's ampersand characters from being interpreted as SAS macro variables.

Note: By default, sessions are identified entirely in the URLs or HTML form fields that reference the session. You can use the SESSION VERIFY option to provide an increased level of session security.

Library Aisles

The library is divided into aisles for different categories of library items. The pages for each aisle are generated by one shared LIB_AISLE.SAS program. The program accepts a TYPE input variable that determines which items to

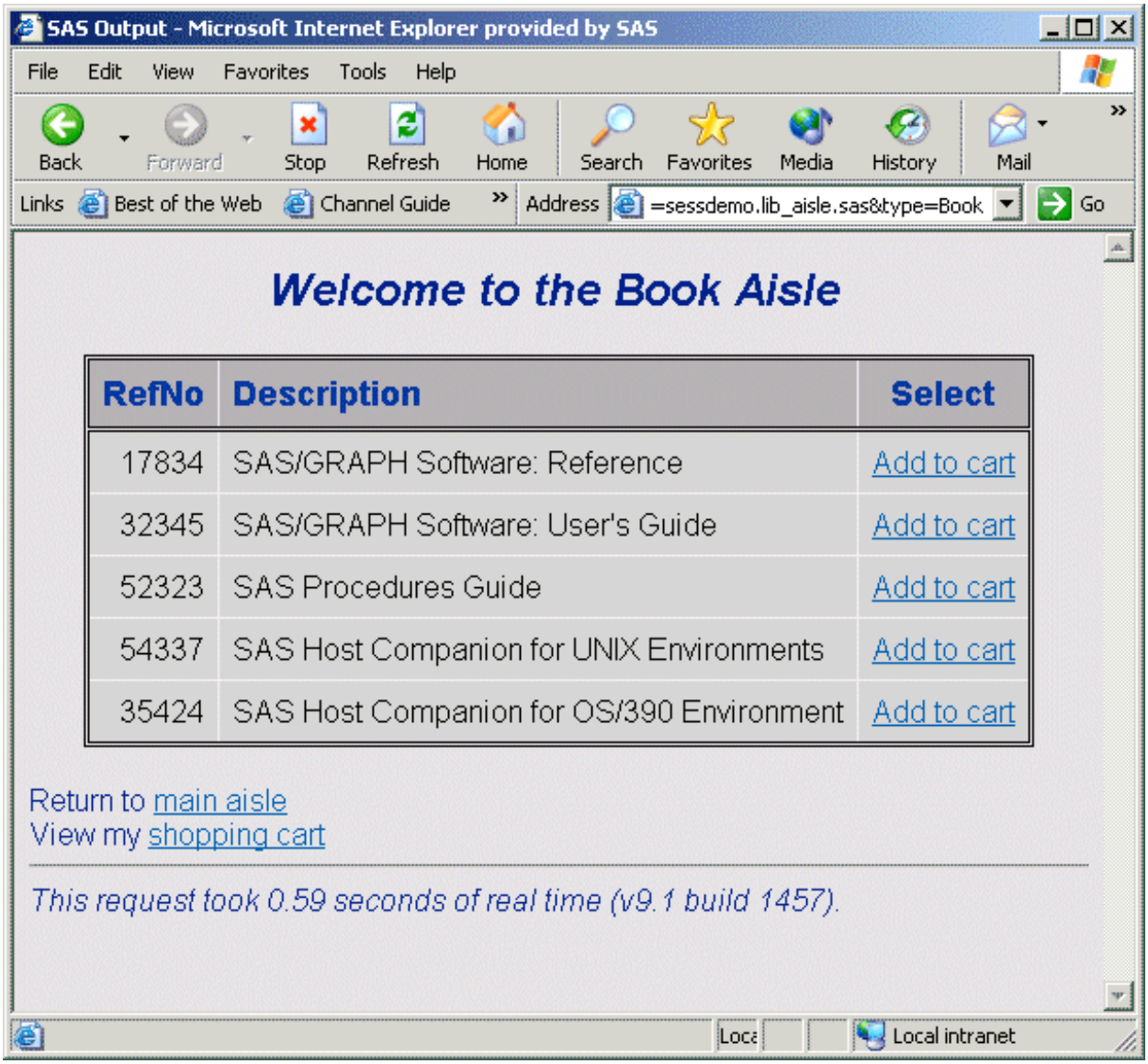
display.

```
/* LIB_AISLE.SAS - List items in a specified aisle.  The aisle
   is specified by the TYPE variable. */

%macro lib_aisle;

   /* Check for a valid session - verifies that the user has logged in. */
   %if %sysfunc(libref(SAVE)) %then %do;

      /* SAVE libref doesn't exist - redirect to login page. */
      data _null_;
         file _webout;
         put '<HTML>';
         put '<HEAD><TITLE>Missing Login</TITLE></HEAD>';
         put '<BODY vlink="#004488" link="#0066AA" bgcolor="#E0E0E0">';
         put '<H1>Missing Login</H1>';
         put;
         put '<P>You must login before you can use this application.  Please';
         put 'return to the <A HREF="' @;
         put "&_URL?_SERVICE=&_service" '&_program=' "&_pgmlib" @;
         put '.lib_login.sas">login page</a> and enter a valid login ID';
         put 'and password.</P>';
         put;
         put '<P>If you are unable to log in, please contact the Library Help';
         put 'at extension 14325.</P>';
         put '</BODY></HTML>';
         run;
   %end;
   %else %do;


      /* Build a temporary data set that contains the selected type, and add
         links for selecting and adding items to the shopping cart. */
      data templist;
         set SAMPDAT.LIB_INVENTORY;
         where type="%UPCASE(&type)";
         length select $200;
         select = '<A HREF="' || "%superq(_THISSESSION)" || '&_program=' ||
            "&_PGMLIB" || '.LIB_ADDITEM.SAS&REFNO=' || trim(left(refno)) ||
            '&TYPE=' || "&TYPE" || '">Add to cart</A>';
         run;

      ods html body=_webout(nobot) rs=none;
      title Welcome to the &type Aisle;
      proc print data=templist noobs label;
         var refno desc select;
         label refno='RefNo' desc='Description' select='Select';
         run;
      ods html close;

      data _null_;
         file _webout;
         put '<P>';
         put 'Return to <A HREF="' "%SUPERQ(_THISSESSION)" '&_PROGRAM='
            "&_PGMLIB" '.LIB_MAIN.SAS">main aisle</A><BR>';
         put 'View my <A HREF="' "%SUPERQ(_THISSESSION)" '&_PROGRAM='
            "&_PGMLIB" '.LIB_CART.SAS">shopping cart</A><BR>';
         put '</BODY>';
         put '</HTML>';
         run;

   %end;
```

```
%mend;

%lib_aisle;
```

The program selects a subset of the LIB_INVENTORY data set using a WHERE clause, and then uses PROC PRINT to create an HTML table. A temporary data set is created that contains the selected items in order for an additional column to be generated that has an HTML link for users to add the item to their shopping cart.

In this program, both ODS and a DATA step are used to generate HTML. The ODS HTML statement includes the NOBOT option that indicates that more HTML will be appended after the ODS HTML CLOSE statement. The navigation links are then added using a DATA step.

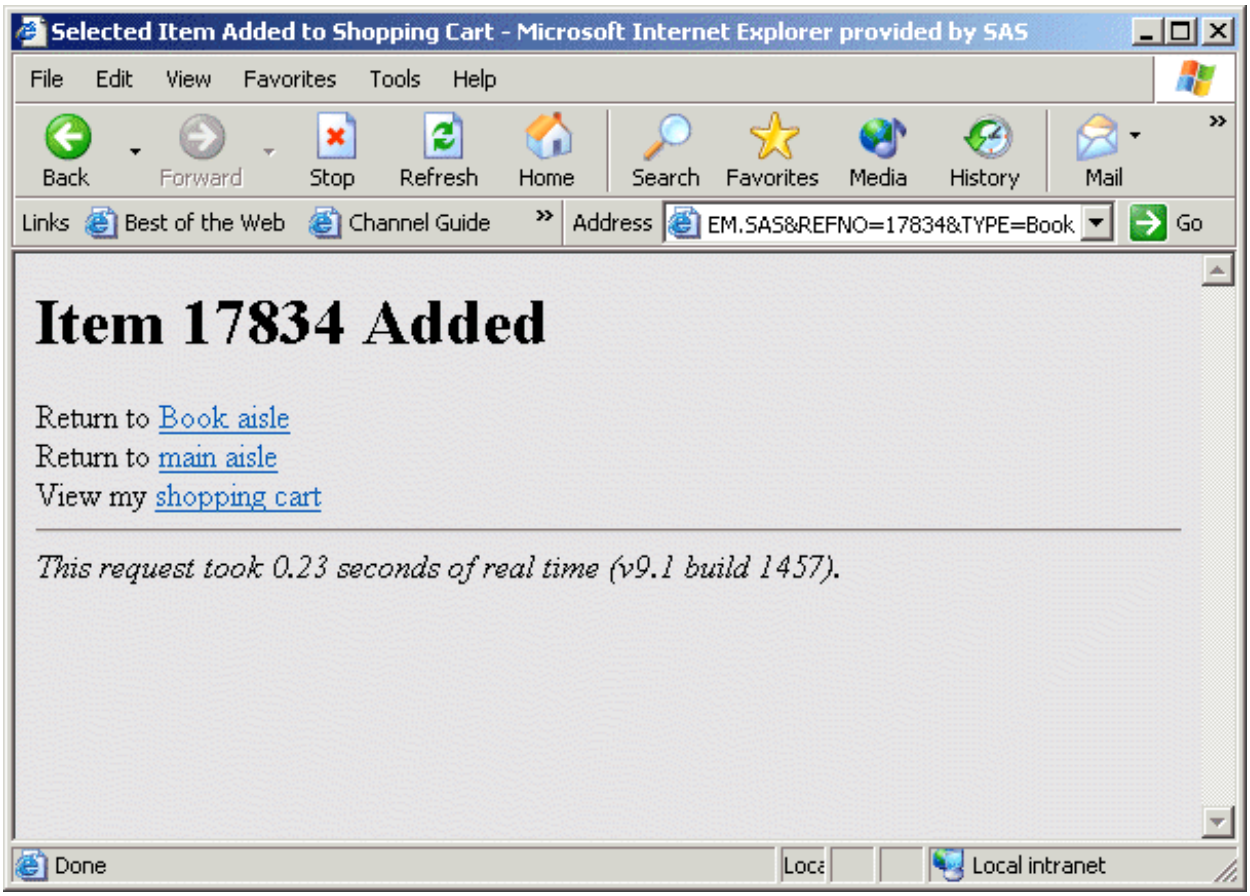


Add Items

The LIB_ADDITEM.SAS program is run when the user clicks the *Add to cart* link in the aisle item table. The specified item is copied from the LIB_INVENTORY data set to a shopping cart data set in the session library (SAVE.CART). The session and the data set will remain accessible to all programs in the same session until the session is deleted or it times out.

```
/* LIB_ADDITEM.SAS - Add a selected item to the shopping cart.  This
   program uses REFNO and TYPE input variables to identify the
   item. */

/* Perform REFNO and TYPE verification here. */

/* Append the selected item. */
proc append base=SAVE.CART data=SAMPDAT.LIB_INVENTORY;
   where refno=&refno;
   run;

/* Print the page. */
data _null_;
   file _webout;
   put '<HTML>';
   put '<HEAD><TITLE>Selected Item Added to Shopping Cart</TITLE></HEAD>';
   put '<BODY vlink="#004488" link="#0066AA" bgcolor="#E0E0E0">';
   put "<H1>Item &refno Added</H1>";
   put 'Return to <A HREF="' "%SUPERQ(_THISSESSION)" '&_PROGRAM='
      "&_PGMLIB" '.LIB_AISLE.SAS&TYPE=' "&TYPE" '">' "&TYPE aisle</A><BR>";
   put 'Return to <A HREF="' "%SUPERQ(_THISSESSION)" '&_PROGRAM='
      "&_PGMLIB" '.LIB_MAIN.SAS">main aisle</A><BR>';
   put 'View my <A HREF="' "%SUPERQ(_THISSESSION)" '&_PROGRAM='
      "&_PGMLIB" '.LIB_CART.SAS">shopping cart</A><BR>';
   put '</BODY>';
   put '</HTML>';
   run;
```

The program prints an information page that has navigation links.

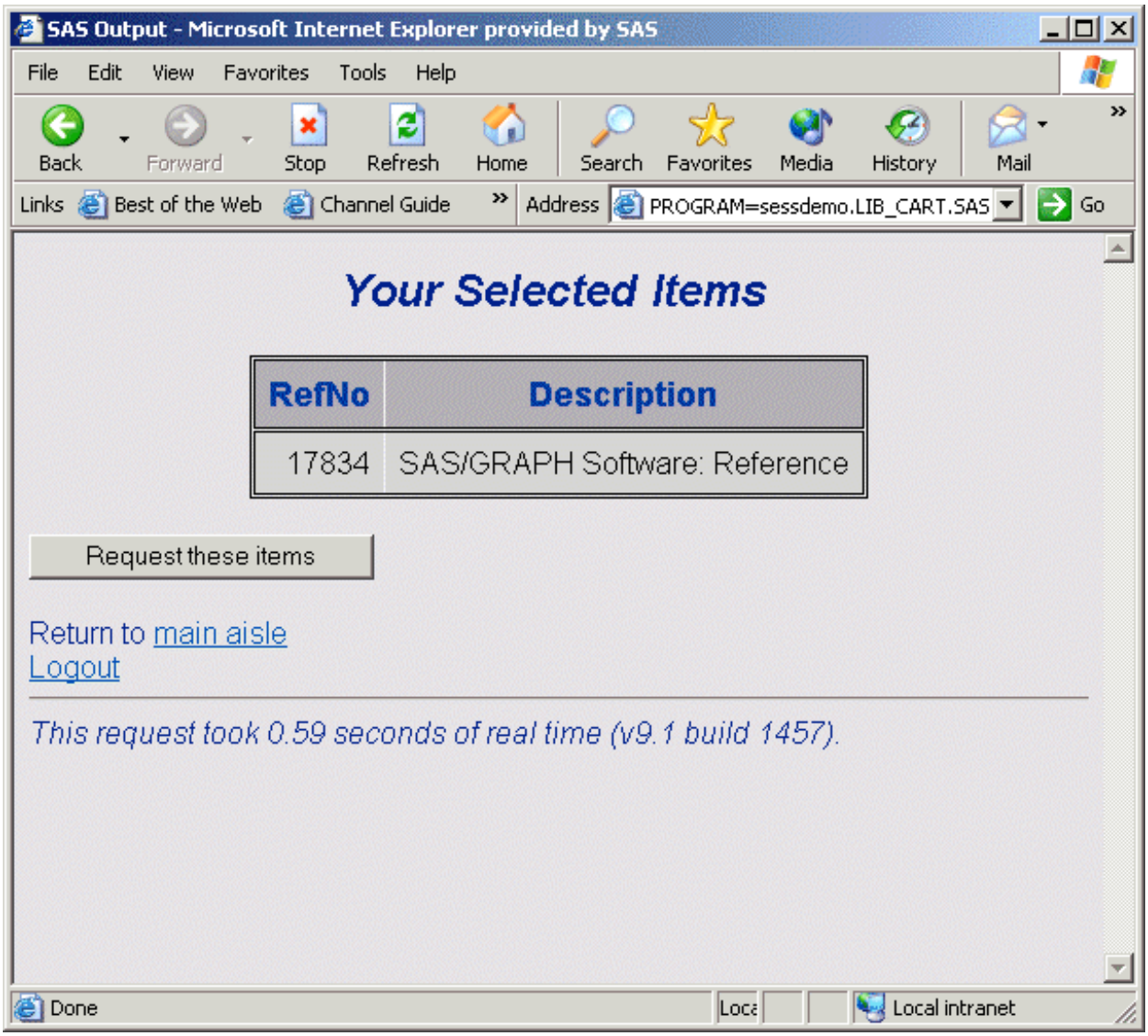


Shopping Cart

The LIB_CART.SAS program displays the contents of the shopping cart.

```
/* LIB_CART.SAS - Display contents of the shopping cart (SAVE.CART data set). */

%macro lib_cart;

   %let CART=%sysfunc(exist(SAVE.CART));
   %if &CART %then %do;
      /* This program could use the same technique as the LIB_AISLE program in order to
         add a link to each line of the table that removes items from the shopping
         cart. */

      /* Print the CART contents. */
      ods html body=_webout(nobot) rs=none;
      title Your Selected Items;
      proc print data=SAVE.CART noobs label;
         var refno desc;
         label refno='RefNo' desc='Description';
         run;
      ods html close;

   %end;
   %else %do;
      /* No items in the cart. */
      data _null_;
         file _webout;
         put '<HTML>';
         put '<HEAD><TITLE>No items selected</TITLE></HEAD>';
         put '<BODY vlink="#004488" link="#0066AA" bgcolor="#E0E0E0">';
         put '<H1>No Items Selected</H1>';
         put;
         run;
   %end;

   /* Print navigation links. */
   data _null_;
      file _webout;
      put '<P>';
      if &CART then do;
         put '<FORM ACTION="' "&_url" '">';
         put '<INPUT TYPE="HIDDEN" NAME="_service" VALUE="' "&_service" '">';
         put '<INPUT TYPE="HIDDEN" NAME="_program" VALUE="' "&_pgmlib" '.LIB_LOGOUT.SAS">';
         put '<INPUT TYPE="HIDDEN" NAME="_server" VALUE="' "&_server" '">';
         put '<INPUT TYPE="HIDDEN" NAME="_port" VALUE="' "&_port" '">';
         put '<INPUT TYPE="HIDDEN" NAME="_sessionid" VALUE="' "&_sessionid" '">';
         put '<INPUT TYPE="HIDDEN" NAME="CHECKOUT" VALUE="YES">';
         put '<INPUT TYPE="SUBMIT" VALUE="Request these items">';
         put '</FORM><P>';
      end;
      put 'Return to <A HREF="' "%SUPERQ(_THISSESSION)" '&_PROGRAM='
         "&_PGMLIB" '.LIB_MAIN.SAS">main aisle</A><BR>';
      put '<A HREF="' "%SUPERQ(_THISSESSION)" '&_PROGRAM='
         "&_PGMLIB" '.LIB_LOGOUT.SAS&CHECKOUT=NO">Logout</A><BR>';
      put '</BODY>';
      put '</HTML>';
      run;
%mend;

%lib_cart;
```

The contents of the shopping cart are displayed using a PROC PRINT statement. The page also includes a request button and navigation links. The request button is part of an HTML form. In order to connect to the same session, include _SERVER, _PORT, and _SESSIONID values in addition to the normal _SERVICE and _PROGRAM values. These values are usually specified as hidden fields and set to the corresponding SAS macro variables. This program has a hidden CHECKOUT field that is initialized to YES in order to indicate that the user is requesting the items in the cart.



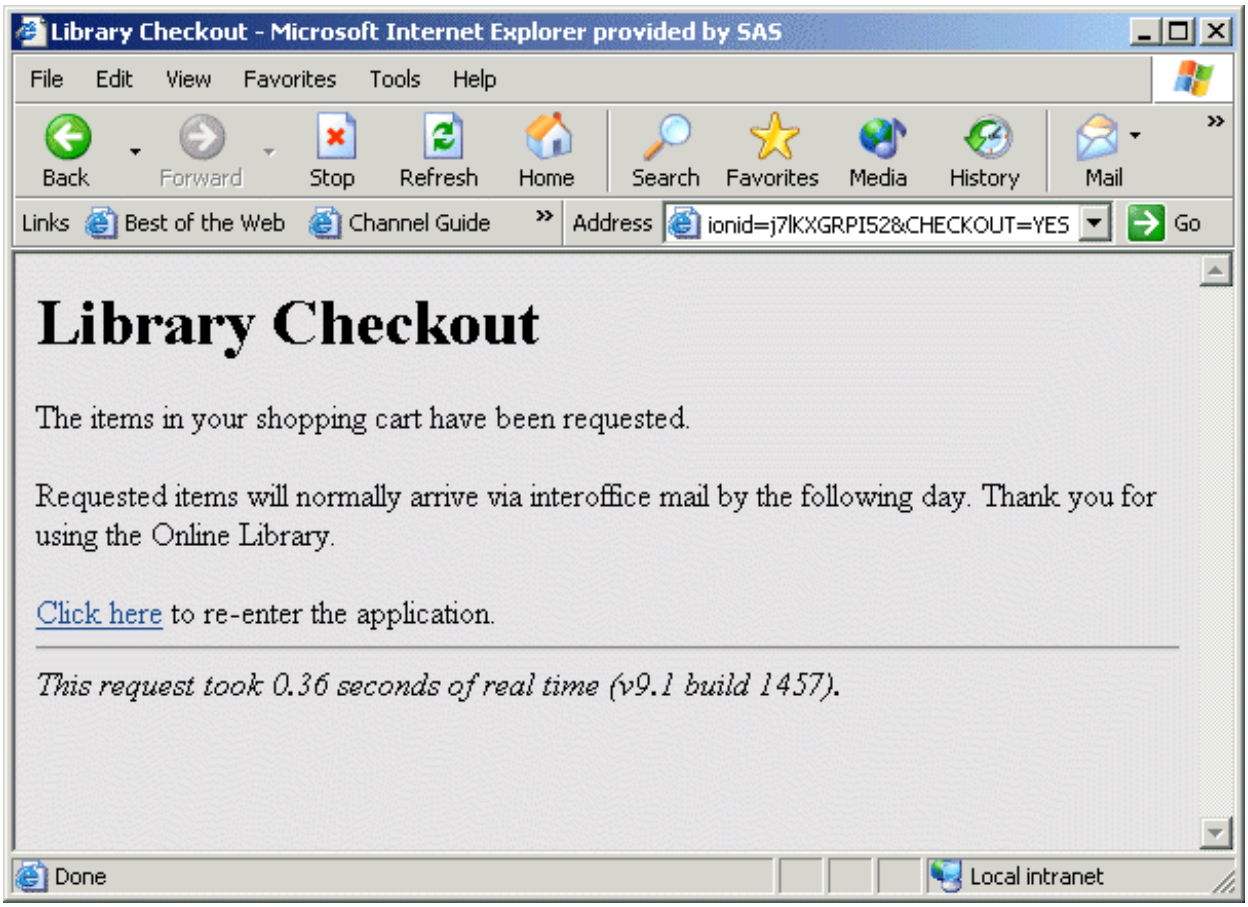
Checkout and Logout

The LIB_LOGOUT.SAS program checks the user out of the Online Library. If the CHECKOUT input variable is YES, then all of the items in the user's shopping cart will be requested via e–mail.

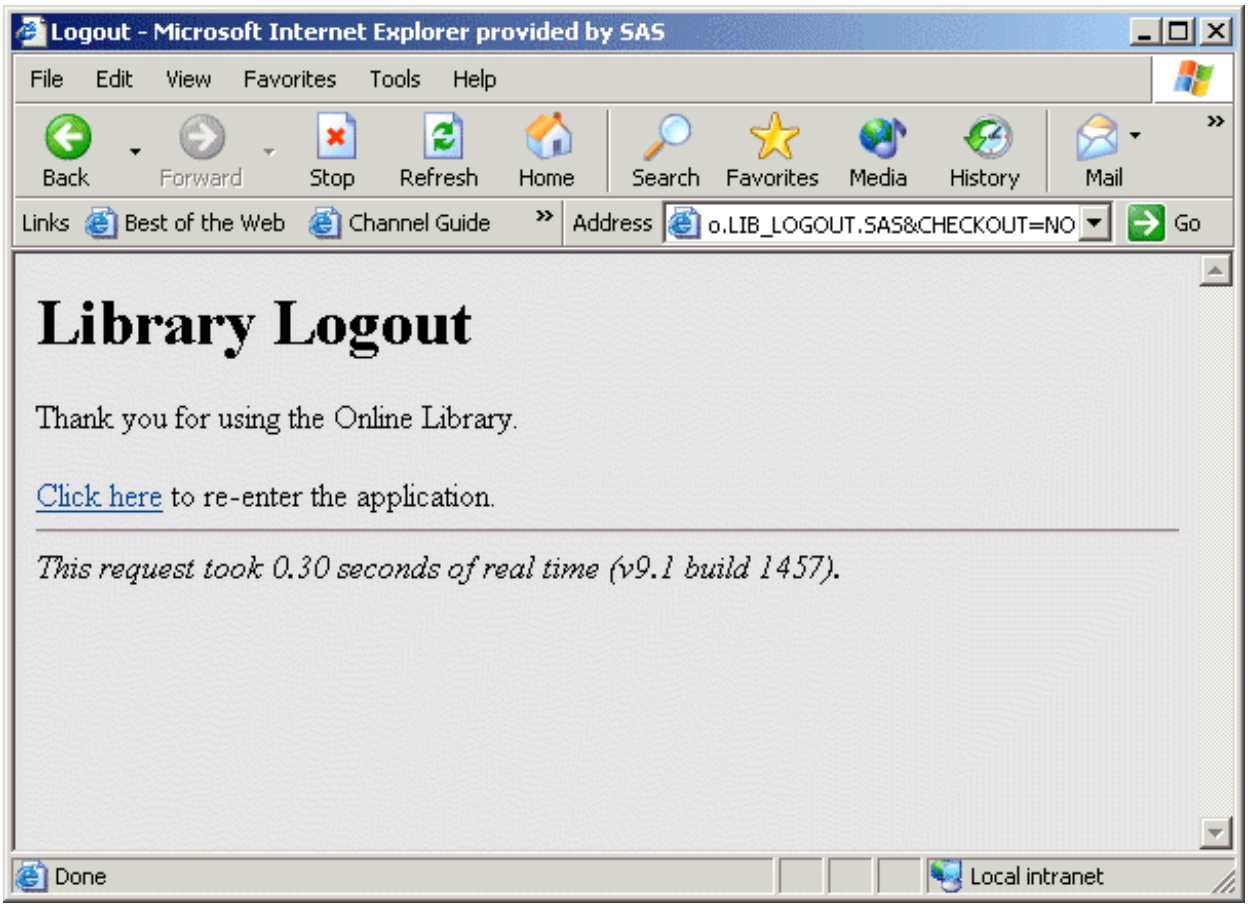
```
/* LIB_LOGOUT.SAS - logout of Online Library application.  Send e-mail to the
   library@abc.com account with requested item if CHECKOUT=YES is specified. */

%macro lib_logout;

   %global CHECKOUT;    /* Define CHECKOUT in case it was not input. */
   %if %UPCASE(&CHECKOUT) eq YES %then %do;
      /* Checkout - send an e-mail request to the library.  E-mail options must
         be specified in order for the Application Server to use the e-mail access method. */
```

```
      filename RQST EMAIL 'library@mybiz.xyz'
         SUBJECT="Online Library Request for &SAVE_LOGINID";
      ods listing body=RQST;
      title Request for &SAVE_LOGINID;
      proc print data=SAVE.CART label;
         var refno type desc;
         label refno='RefNo' type='Type' desc='Description';
         run;
      ods listing close;

      data _null_;
         file _webout;
         put '<HTML>';
         put '<HEAD><TITLE>Library Checkout</TITLE></HEAD>';
         put '<BODY vlink="#004488" link="#0066AA" bgcolor="#E0E0E0">';
         put '<H1>Library Checkout</H1>';
         put;
         put 'The items in your shopping cart have been requested.';
         put '<P>Requested items will normally arrive via interoffice';
         put 'mail by the following day.  Thank you for using the Online Library.';
         put '<P><A HREF="' "&_URL?_SERVICE=&_SERVICE" '&_PROGRAM='
            "&_PGMLIB" '.LIB_LOGIN.SAS">Click here</A> to re-enter the';
         put 'application.';
         put '</BODY>';
         put '</HTML>';
         run;

   %end;
   %else %do;

      /* Logout without requesting anything. */
      data _null_;
         file _webout;
         put '<HTML>';
         put '<HEAD><TITLE>Logout</TITLE></HEAD>';
         put '<BODY vlink="#004488" link="#0066AA" bgcolor="#E0E0E0">';
         put '<H1>Library Logout</H1>';
         put;
         put '<P>Thank you for using the Online Library.';
         put '<P><A HREF="' "&_URL?_SERVICE=&_SERVICE" '&_PROGRAM='
            "&_PGMLIB" '.LIB_LOGIN.SAS">Click here</A> to re-enter the';
         put 'application.';
         put '</BODY>';
         put '</HTML>';
         run;
   %end;

%mend;

%lib_logout;

/* User is finished - delete the session. */
%let rc=%sysfunc(appsrv_session(delete));
```

An information page is displayed if the user chooses to request the shopping cart items.



A simple logout screen is displayed if the user selects the Logout link.



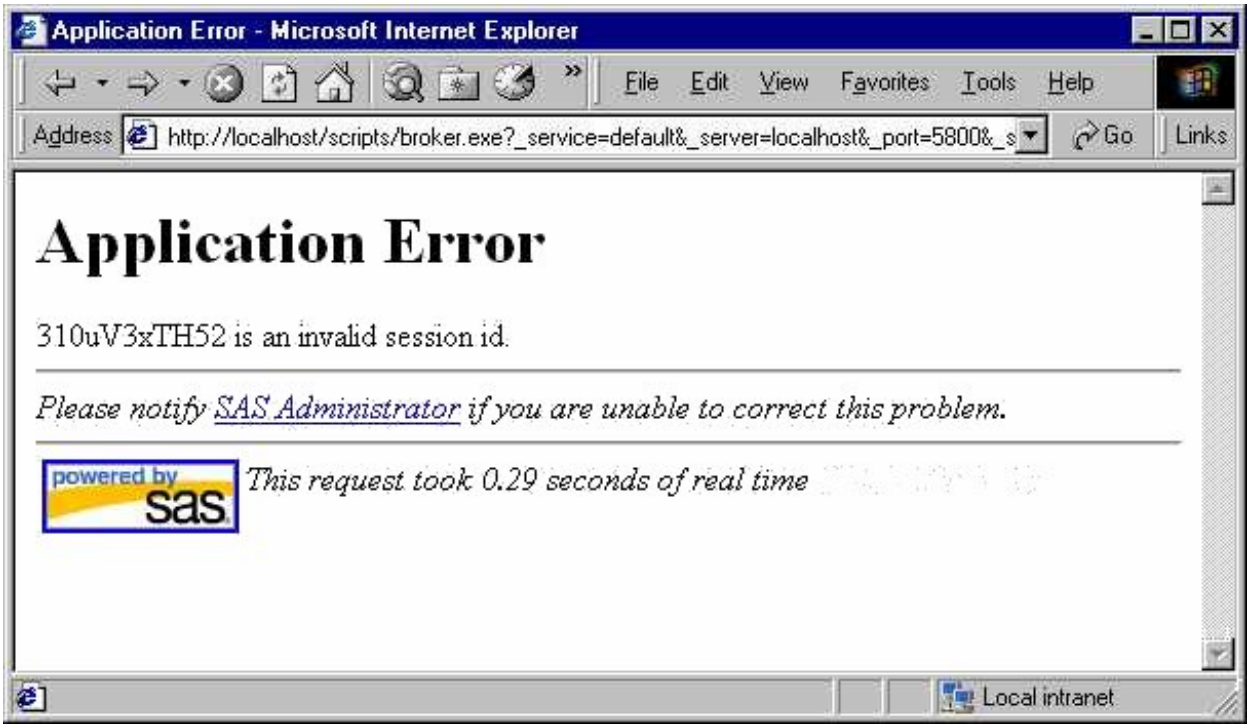
Note: Logging out is not required. All sessions have an associated timeout (the default is 15 minutes). If the session is not accessed for the duration of the timeout, the session and all temporary data in the session will be deleted. In this sample, the SAVE.CART data set and the SAVE_LOGINID macro variable would be automatically deleted when the session timeout is reached. You can change the session timeout using the TIMEOUT option of the PROC APPSRV SESSION statement or by using the APPSRV_SET('session timeout',*seconds*) function inside the program.

Invalid Session Program

Users of the Online Library application might see an invalid session Application Error page. An invalid session might happen for the following reasons:

- The user logged into the application, but then the session expired because the user did not input information during the session timeout period (default is 15 minutes).
- The user logged out of the application and then attempted to re−enter by using the browser's back button or history list.
- The user bookmarked a page in the application and then attempted to return to that bookmark after logging out of the application or after the session had expired.
- The Application Server was stopped and restarted while the user was logged in to the application.

The default invalid session page is not very useful to the typical user because it does not provide specific information.

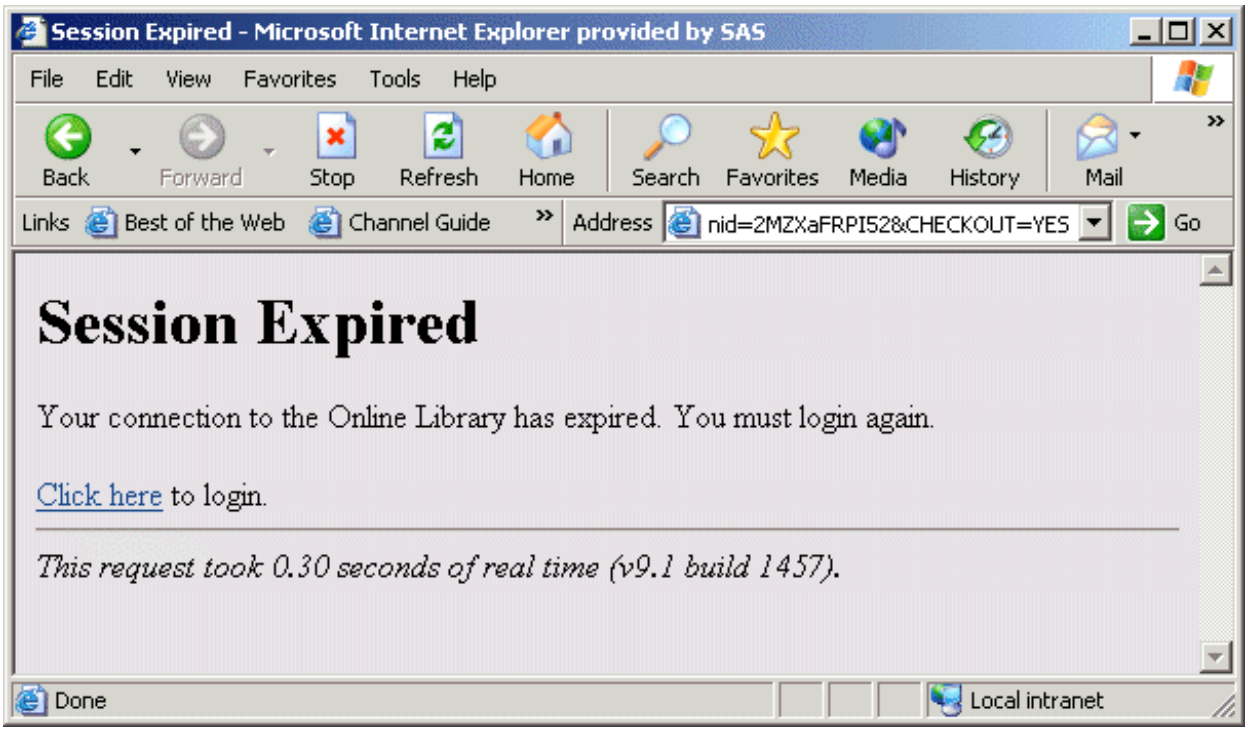


You can replace the default page by using the INVSESS= option of the PROC APPSRV SESSION statement. The INVSESS= option specifies a program that will run when the Application Server finds a nonexistent (invalid or expired) session. Note that the INVSESS program applies to all applications that use a particular Application Dispatcher service. The INVSESS program can use the _USERPROGRAM variable to determine which program or application the user was attempting to run, and then print a suitable error page. An INVSESS program for the Online Library application follows:

```
/* LIB_INVSESS.SAS - Display useful message for expired or invalid sessions. */

%macro lib_invsess;

   %if %upcase(%substr(%scan(&_USERPROGRAM,2,.),1,4)) eq LIB_ and
         %upcase(%scan(&_USERPROGRAM,3,.)) eq SAS %then %do;
      /* If the program name (second part of three part name) starts with LIB_
         and the program type (third part) is SAS, then this is the On-Line
         Library application and you can print an application-specific message. */
      data _null_;
         file _webout;
         put '<HTML>';
         put '<HEAD><TITLE>Session Expired</TITLE></HEAD>';
         put '<BODY vlink="#004488" link="#0066AA" bgcolor="#E0E0E0">';
         put '<H1>Session Expired</H1>';
         put;
         put 'Your connection to the Online Library has expired.  You must';
         put 'login again.';
         put '<P>';
         put '<A HREF="' "&_URL?_SERVICE=&_SERVICE" '&_PROGRAM='
            "%scan(&_USERPROGRAM,1,.).LIB_LOGIN.SAS"
            '">Click here</A> to login.';
         put '</BODY>';
         put '</HTML>';
         run;
   %end;
   %else %do;
      /* Otherwise, this is an unknown application; print a generic
         error page. */
```

```
        data _null_;
           file _webout;
           put '<HTML>';
           put '<HEAD><TITLE>Invalid or Expired Session</TITLE></HEAD>';
           put '<BODY vlink="#004488" link="#0066AA" bgcolor="#E0E0E0">';
           put '<H1>Invalid or Expired Session</H1>';
           put;
           put 'Your application session has expired.  In order to continue,';
           put 'you must restart this application.';
           put '</BODY>';
           put '</HTML>';
           run;
     %end;
%mend;

%lib_invsess;
```

After the Application Server is configured to use the LIB_INVSESS.SAS program, an expired session message for the Online Library application will display:



Uploading Files

Starting with SAS 9.1.3 Service Pack 4, you can use Application Dispatcher to upload one or more files to your Application Server. The upload process is usually initiated by an HTML page that contains an INPUT tag with the attribute TYPE set to `"file"`:

```
<input type="file" name="myfile">
```

This tag enables you to specify the file that you want to upload. After the form data is submitted, the file you chose and any other name/value pairs that are contained in the HTML form are sent to the Application Server. Your SAS program can then use both the name/value pairs and the file that was uploaded.

Reserved Macro Variables

The reserved SAS macro variables that are associated with uploading files all start with `_WEBIN_`.

_WEBIN_CONTENT_LENGTH
: specifies the length, in bytes, of the file that was uploaded.

_WEBIN_CONTENT_TYPE
: specifies the content type that is associated with the file.

_WEBIN_FILE_COUNT
: specifies the number of files that were uploaded. If no files were uploaded, the value of this variable will be set to zero.

_WEBIN_FILEEXT
: specifies the extension of the file that was uploaded.

_WEBIN_FILENAME
: specifies the original location of the file.

_WEBIN_FILEREF
: specifies the SAS FILEREF that is automatically assigned to the uploaded file. You can use this FILEREF to access the file. The uploaded file is stored in a temporary location on the Application Server, and will be deleted when the request is completed. Be sure to copy the file to a permanent location if you need to access it at a later date.

_WEBIN_NAME
: specifies the value that is specified in the NAME attribute of the INPUT tag. In the example above, the value would be `myfile`.

_WEBIN_SASNAME
: specifies a unique name for the uploaded SAS table, view, or catalog. A value is set for this macro variable only if a SAS table, view, or catalog was uploaded. All SAS data types are stored in the Work library. The type of SAS file that was uploaded is stored in the _WEBIN_SASTYPE macro variable. See also _WEBIN_SASNAME_ORI.

_WEBIN_SASNAME_ORI
: specifies the original name of the uploaded SAS table, view, or catalog. If a SAS table named `mydata.sas7bdat` was uploaded, then _WEBIN_SASNAME_ORI would contain the value `mydata`. A value is set for this macro variable only if a SAS table, view, or catalog was uploaded. All SAS data types are stored in the Work library. The type of SAS file that was uploaded is stored in the _WEBIN_SASTYPE macro variable. See also _WEBIN_SASNAME.

_WEBIN_SASTYPE
: specifies the type of SAS file that was uploaded: DATA for SAS tables, VIEW for SAS views, and CATALOG for SAS catalogs. A value is set for this macro variable only if a SAS table, view, or catalog was uploaded. The name of the uploaded file is stored in the _WEBIN_SASNAME macro variable.

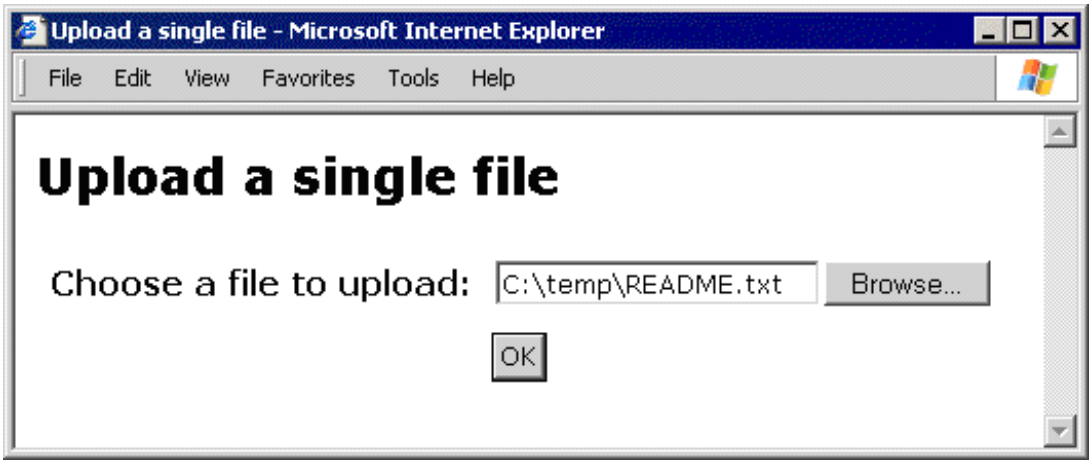
If you are uploading more than one file, unique macro variables will be created for each file. This applies to all of the previous reserved macro variables except _WEBIN_FILE_COUNT. See Multiple Value Pairs for more information.

Note: For z/OS, the SAS server must be invoked with the FILESYSTEM=HFS option in order to be able to upload SAS file types.

Examples of How to Upload Files

Example 1: Uploading a single file

The following figure shows an HTML page that can be used to upload a single file to the Application Server:



The HTML for performing the upload might look like this:

```
<form action="<BrokerURL>" method="post" enctype="multipart/form-data">
<input type="hidden" name="_service" value="<ServiceName>">
<input type="hidden" name="_program" value="<ProgramName>">
<table border="0" cellpadding="5">
   <tr>
      <th>Choose a file to upload:</th>
      <td><input type="file" name="myfile"></td>
   </tr>
   <tr>
      <td colspan="2" align="center"><input type="submit" value="OK"></td>
   </tr>
</table>
</form>
```

In the previous lines of HTML, you must replace "*<BrokerURL>*" with the path to the SAS/IntrNet Application Broker. For example, on Windows, this path is usually `http://YourServer/scripts/broker.exe`, where *YourServer* corresponds to the domain name of your Web server. Similarly, you will need to specify the service name and the program that you want to execute after the file has been uploaded. You should specify the exact values that are shown for the METHOD and ENCTYPE attributes of the FORM tag.

The INPUT tag in the previous lines of HTML is used to create the **Browse** button and text entry field in the previous figure. The appearance of this control might be different depending on which Web browser you use, but the functionality should be the same. Clicking the **Browse** button enables you to navigate to the file that you want to upload. You can choose any file that you have access to. This example uses the file `readme.txt`, which resides in the Windows directory `C:\temp`.

After you select a file and click **OK**, all form data is sent to the Application Broker, which in turn, forwards the data to the Application Server. As a result, the following SAS macro variables are created:

Variable Name	Value	Description
_WEBIN_CONTENT_LENGTH	1465	Specifies the size of the uploaded file in bytes (supplied automatically by the Web browser).
_WEBIN_CONTENT_TYPE	text/plain	Specifies the content type that corresponds to the uploaded file (supplied automatically by the Web browser).
_WEBIN_FILE_COUNT	1	Specifies the number of files that were uploaded.
_WEBIN_FILEEXT	txt	Specifies the extension of the file that was uploaded.
_WEBIN_FILENAME	C:\temp\README.txt	Specifies the name and original location of the file that was uploaded.
_WEBIN_FILEREF	#LN00197	Specifies the SAS FILEREF that you can use to access the uploaded file. This FILEREF is assigned for you by the SAS server.
_WEBIN_NAME	myfile	Specifies the value that corresponds to the NAME attribute of the INPUT tag.

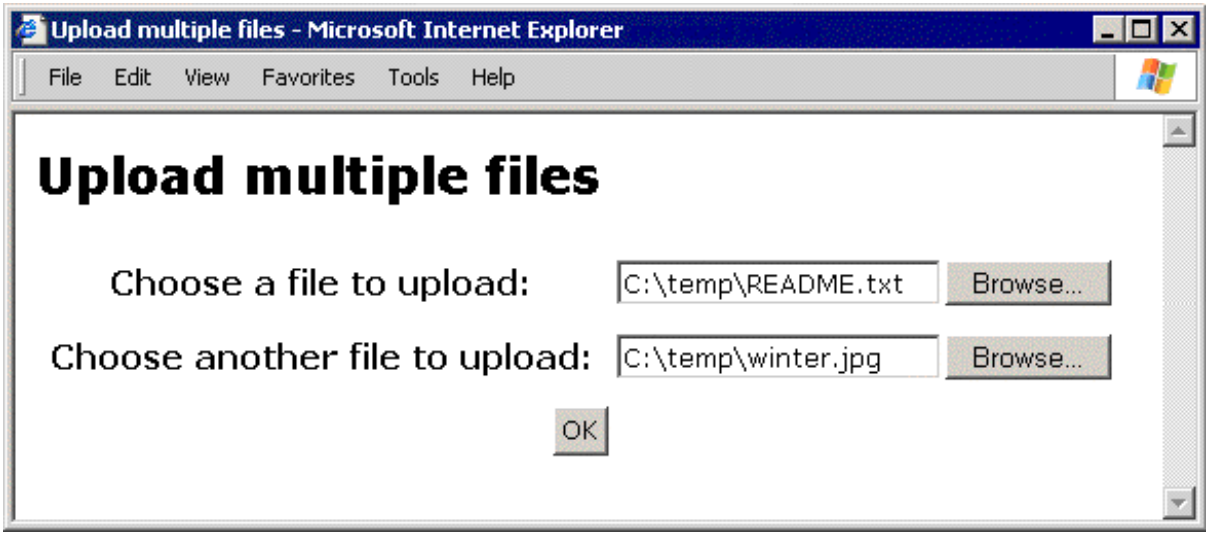
Your SAS/IntrNet program has access to the uploaded file via the FILEREF that is stored in the value of the _WEBIN_FILEREF macro variable. The following code example returns the uploaded file to the client:

```
*  Set the Content-type header;
%let RV = %sysfunc(appsrv_header(Content-type, &_WEBIN_CONTENT_TYPE));

*  Write the file back to the Web browser;
data _null_;
   length data $1;

   infile &_WEBIN_FILEREF recfm=n;
   file _webout recfm=n;
   input data $char1. @@;
   put data $char1. @@;
run;
```

The previous code shows how to use the _WEBIN_CONTENT_TYPE macro variable to set the content−type header. The previous code also shows how to use the _WEBIN_FILEREF macro variable to access the uploaded file.

Example 2: Uploading multiple files

The following figure shows an HTML page that can be used to upload multiple files to the Application Server:



The HTML for performing the upload might look like this:

```
<form action="<BrokerURL>" method="post" enctype="multipart/form-data">
<input type="hidden" name="_service" value="<ServiceName>">
<input type="hidden" name="_program" value="<ProgramName>">
<table border="0" cellpadding="5">
   <tr>
      <th>Choose a file to upload:</th>
      <td><input type="file" name="firstfile"></td>
   </tr>
   <tr>
      <th>Choose another file to upload:</th>
      <td><input type="file" name="secondfile"></td>
   </tr>
   <tr>
      <td colspan="2" align="center"><input type="submit" value="OK"></td>
   </tr>
</table>
</form>
```

Refer to Example 1 for a basic discussion of the previous lines of HTML. This example uses the files `readme.txt` and `winter.jpg`, which reside in the Windows directory `C:\temp`. Note that the two input files do not need to be in the same directory.

After you select a file and click **OK**, all form data is sent to the Application Broker, which in turn, forwards the data to the Application Server. As a result, the following SAS macro variables are created:

Variable Name	Value	Description
_WEBIN_CONTENT_LENGTH	1465	Specifies the size of the first uploaded file in bytes (supplied automatically by the Web browser).
_WEBIN_CONTENT_LENGTH0	2	Specifies the number of files that were uploaded.
_WEBIN_CONTENT_LENGTH1	1465	Specifies the size of the first uploaded file in bytes (supplied automatically by the Web browser).
_WEBIN_CONTENT_LENGTH2	5367	Specifies the size of the second uploaded file in bytes (supplied automatically by the Web browser).
_WEBIN_CONTENT_TYPE	text/plain	

		Specifies the content type that corresponds to the first uploaded file (supplied automatically by the Web browser).
_WEBIN_CONTENT_TYPE0	2	Specifies the number of files that were uploaded.
_WEBIN_CONTENT_TYPE1	text/plain	Specifies the content type that corresponds to the first uploaded file (supplied automatically by the Web browser).
_WEBIN_CONTENT_TYPE2	image/pjpeg	Specifies the content type that corresponds to the second uploaded file (supplied automatically by the Web browser).
_WEBIN_FILE_COUNT	2	Specifies the number of files that were uploaded.
_WEBIN_FILEEXT	txt	Specifies the extension of the first file that was uploaded.
_WEBIN_FILEEXT0	2	Specifies the number of files that were uploaded.
_WEBIN_FILEEXT1	txt	Specifies the extension of the first file that was uploaded.
_WEBIN_FILEEXT2	jpg	Specifies the extension of the second file that was uploaded.
_WEBIN_FILENAME	C:\temp\README.txt	Specifies the name and original location of the first file that was uploaded.
_WEBIN_FILENAME0	2	Specifies the number of files that were uploaded.
_WEBIN_FILENAME1	C:\temp\README.txt	Specifies the name and original location of the first file that was uploaded.
_WEBIN_FILENAME2	C:\temp\winter.jpg	Specifies the name and original location of the second file that was uploaded.
_WEBIN_FILEREF	#LN00014	Specifies the SAS FILEREF that you can use to access the first uploaded file.
_WEBIN_FILEREF0	2	Specifies the number of files that were uploaded.
_WEBIN_FILEREF1	#LN00014	Specifies the SAS FILEREF that you can use to access the first uploaded file.
_WEBIN_FILEREF2	#LN00016	Specifies the SAS FILEREF that you can use to access the second uploaded file.
_WEBIN_NAME	firstfile	Specifies the value that corresponds to the NAME attribute of the first INPUT tag.
_WEBIN_NAME0	2	Specifies the number of files that were uploaded.
_WEBIN_NAME1	firstfile	Specifies the value that corresponds to the NAME attribute of the first INPUT tag.
_WEBIN_NAME2	secondfile	Specifies the value that corresponds to the NAME attribute of the second INPUT tag.

Examples of How to Use Uploaded Files

Example 3: Uploading a CSV file to a SAS table

After you have uploaded a CSV file, you can use the IMPORT procedure to import the file to a SAS table. The following sample code shows one way of achieving this:

```
%let CSVFILE=%sysfunc(pathname(&_WEBIN_FILEREF));

proc import datafile="&CSVFILE"
   out=work.mydata
   dbms=csv
   replace;
   getnames=yes;
run;

title 'First 10 records of CSV file after importing to a SAS table.';

ods html body=_webout style=Seaside path=&_tmpcat (url=&_replay) rs=none;
   proc print data=work.mydata(obs=10); run; quit;
ods html close;
```

Because the IMPORT procedure requires a full path to the CSV file, you must first use the PATHNAME function to get the path to the file. The GETNAMES statement uses the data in the first row of the CSV file for the SAS column names. See the IMPORT procedure in the *Base SAS Procedures Guide* for further details.

An alternative method would be to write a DATA step to import the CSV file. This method would require only Base SAS. The following code is an example of how to do this:

```
data work.mydata;
   infile &_WEBIN_FILEREF dlm=',' dsd;
   *  Your code to read the CSV file;
run;
```

Example 4: Uploading an Excel XML workbook to multiple SAS tables

Starting with Excel XP (Excel 2002), a workbook can be saved as an XML file. This XML file can be read into SAS using the SAS XML LIBNAME Engine and a SAS XMLMap. Each worksheet in the workbook will be imported to a SAS table with the same name, and the column headings in the worksheets will be used for the column names in the SAS tables. The following code is an example of how to do this. Be sure to include the appropriate directory paths.

```
%let XMLFILE=%sysfunc(pathname(&_WEBIN_FILEREF));

*  Include the XLXP2SAS macro;
%include 'loadxl.sas';
*  Import the workbook into SAS tables;
%XLXP2SAS(excelfile=&XMLFILE,
   mapfile=excelxp.map);
```

The %INCLUDE statement makes the XLXP2SAS macro available to SAS. The %XLXP2SAS macro imports the data from all the worksheets into separate SAS tables with the help of a SAS XMLMap. For more details, see the paper Moving Data and Analytical Results between SAS and Microsoft Office at support.sas.com/rnd/papers. There are links available for you to download both the macro and the XMLMap.

Example 5: Uploading a SAS table or view

When a SAS data type (table, view, or catalog) has been uploaded, additional reserved macro variables are created. For example, the following macro variables will be created if the file `C:\temp\djia.sas7bdat` has been uploaded:

Variable Name	Value	Description
_WEBIN_SASNAME	_B3FF5FCAF39482D93793AEEF05BB15F	Specifies a unique name for the uploaded SAS table, which is stored in the Work library.
_WEBIN_SASNAME_ORI	djia	Specifies the original name of the uploaded SAS table.
_WEBIN_SASTYPE	DATA	Specifies the type of SAS file that was uploaded: DATA for a SAS table; VIEW for a SAS view.

To print the SAS table or view that has been uploaded, use the following code:

```
title 'First 10 records of uploaded SAS data file.';

ods listing close;
ods html body=_webout style=Seaside path=&_TMPCAT (url=&_REPLAY) rs=none;
   proc print data=&_WEBIN_SASNAME(obs=10); run; quit;
ods html close;
```

Example 6: Uploading a SAS catalog

You can use the following sample code to list the contents of a SAS catalog that has been uploaded:

```
ods listing close;
ods html body=_webout style=Seaside path=&_TMPCAT (url=&_REPLAY) rs=none;
   proc catalog c=&_WEBIN_SASNAME;
      contents;
   run; quit;
ods html close;
```

Example 7: Uploading a SAS table, view, or catalog and saving a permanent copy

You can use the following sample code to make a permanent copy of an uploaded SAS table, view, or catalog and to retain the name of the original uploaded file:

```
proc datasets library=<YourLibrary>;
   copy in=work out=<YourLibrary> memtype=&_WEBIN_SASTYPE;
      select &_WEBIN_SASNAME;
   run;
      change &_WEBIN_SASNAME=&_WEBIN_SASNAME_ORI;
run;
quit;
```

In the previous lines of SAS code, you must replace <*YourLibrary*> with the name of the SAS library in which you want to store the SAS table, view, or catalog.

Example 8: Uploading an Excel workbook to a SAS table

You can use the IMPORT procedure to import an uploaded Excel workbook file to a SAS table. The following sample code shows one way of achieving this:

```
%let XLSFILE=%sysfunc(pathname(&_WEBIN_FILEREF));

proc import datafile="&XLSFILE"
   out=work.mydata
   dbms=excel
   replace ;
   getnames=yes;
run; quit;

title 'First 10 records of Excel workbook after importing to a SAS table.';

ods listing close;
ods html body=_webout style=Seaside path=&_tmpcat (url=&_replay) rs=none;
   proc print data=work.mydata(obs=10); run; quit;
ods html close;
```

Because the IMPORT procedure requires a full path to the Excel workbook, you must first use the PATHNAME function to get the path to the file. The GETNAMES statement uses the data in the first row of the workbook for the SAS column names. See the IMPORT procedure in the *Base SAS Procedures Guide* for further details.

Application Server Functions

Application Server functions are Data step functions that you use to define character, numeric, and alphanumeric strings to generate output in the desired format within a PROC APPSRV statement. The following list of Application Server functions can be used to return the correct character, numeric, or alphanumeric value of a PROC APPSRV parameter setting.

- APPSRVGETC
- APPSRVGETN
- APPSRVSET
- APPSRV_AUTHCLS
- APPSRV_AUTHDS
- APPSRV_AUTHLIB
- APPSRV_HEADER
- APPSRV_SESSION
- APPSRV_UNSAFE

APPSRVGETC

Returns the character value of a PROC APPSRV parameter setting

Syntax

VALUE = APPSRVGETC(*valuecode*)

Arguments

valuecode
is the character string name of the parameter.

Details

The APPSRVGETC function takes one character string parameter and returns a character string result.

Examples

SAS Statements	Results
`auth=appsrvgetc('auth');` `put auth=;`	`auth=NONE`
`initpgm=appsrvgetc('request init');` `put initpgm=;`	`initpgm=MYLIB.MYINIT.SAS`
`termpgm=appsrvgetc('request term');` `put termpgm=;`	`termpgm=MYLIB.MYCAT.MYTERM.SCL`
`initpgm=appsrvgetc('session init');` `put initpgm=;`	`initpgm=MYLIB.MYCAT.SESSINIT.SOURCE`
`termpgm=appsrvgetc('session term');` `put termpgm=;`	`termpgm=MYLIB.MYCAT.MYTERM.SCL`
`version=appsrvgetc('version');` `put version=;`	`version=SAS/IntrNet Application Server Release 9.1 (Build 527)`
`fname = appsrvgetc('log file');` `put fname=;`	`fname=/u/intrnet/services/default/logs/Fri_5801.log`

APPSRVGETN

Returns the numeric value of a PROC APPSRV parameter setting

Syntax

VALUE = APPSRVGETN(*valuecode*)

Arguments

valuecode
is the character string name of the parameter.

Details

The APPSRVGETN function takes one character string parameter and returns a numeric string result.

Examples

SAS Statements	Results
`maxtimeout=appsrvgetn('request maxtimeout');` `put maxtimeout=;`	`maxtimeout=900`
`timeout=appsrvgetn('request timeout');` `put timeout=;`	`timeout=300`
`sessmaxtimeout=appsrvgetn('session maxtimeout');` `put sessmaxtimeout=;`	`sessmaxtimeout=900`
`session=appsrvgetn('session timeout');` `put session=;`	`session=900`
`starttime=appsrvgetn('server starttime');` `put starttime=datetime.;`	`starttime=01SEP02:08:15:59`
`version=appsrvgetn('version');` `put version=;`	`version=9.1`

APPSRVSET

Returns the numeric value of a PROC APPSRV parameter setting

Syntax

RC = APPSRVSET(*valuecode*, *newvalue*)

Arguments

valuecode
is the character string name of the parameter.

newvalue
is the numeric string name of the parameter.

The following table lists the valid parameters for *valuecode* and provides a description of each.

Valuecode	Description
AUTOMATIC HEADERS	specifies whether the APPSRV procedure returns headers. The value must be 0 (disabled) or 1 (enabled). Automatic header generation is enabled by default.
PROGRAM ERROR	specifies the return code when there is an error. This can be set to any value.
REQUEST TIMEOUT	specifies the number of seconds that a requested program is allowed to run before it is terminated by the server. The default session timeout is 300 (5 minutes).
SESSION TIMEOUT	specifies the number of seconds that elapse before a session expires. The default session timeout is 900 (15 minutes).

Details

The APPSRVSET function takes one character string parameter and one numeric string parameter and returns a numeric string result.

Examples

SAS Statements
`rc=appsrvset('request timeout',300);`
`rc=appsrvset('session timeout',900);`
`rc=appsrvset('program error',256);`

```
/* disable generation of MIME headers */

rc=appsrvset('automatic headers',0);
```

APPSRV_AUTHCLS

Reads the AUTHLIB data set and returns a WHERE clause

Syntax

CLAUSE = APPSRV_AUTHCLS(*type*)

Arguments

type
is one of the following character strings: LIBRARY, MEMBER, CATALOGENTRY.

Details

The APPSRV_AUTHCLS function reads the AUTHLIB data set and returns a WHERE clause. This clause references the variable names LIBNAME, MEMNAME, MEMTYPE, OBJNAME, and OBJTYPE. It can be applied to the SQL dictionary views and other views in the SASHELP library. The returned clause can be used to subset the entities in the current SAS session to only the entities that are authorized by the AUTHLIB data set. The returned clause can be combined with a user−determined clause by using the "and" token to create a compound clause that selects the desired entities, provided that access is authorized.

If the value of *type* is LIBRARY, then the returned clause contains only the LIBNAME variable. If the value of *type* is MEMBER, then the returned clause contains the LIBNAME, MEMNAME, and MEMTYPE variables. If the value of *type* is CATALOGENTRY, then the returned clause contains the LIBNAME, MEMNAME, MEMTYPE, OBJNAME, and OBJTYPE variables.

Examples

For the examples in Table 2, refer to the contents of the SASHELP.AUTHLIB data set in Table 1. ***Entities are excluded by default, and all exclude rules supersede all include rules.***

Table 1: Contents of SASHELP.AUTHLIB Data Set

Rule	Libname	Memname	Memtype	Objname	Objtype
INCLUDE	SASHELP	*	DATA	*	*
INCLUDE	SASHELP	*	VIEW	*	*
INCLUDE	SASHELP	*	MDDB	*	*
INCLUDE	SAMPDAT	*	*	*	*
EXCLUDE	SAMPDAT	MYCAT	CATALOG	*	*

Table 2: Examples

SAS Statements	Results
clause=appsrv_authcls('LIBRARY'); put clause=;	clause=((upcase(libname)='SASHELP')or (upcase(libname)='SASHELP')or (upcase(libname)='SASHELP')or (upcase(libname)='SAMPDAT'))
clause=appsrv_authcls('MEMBER'); put clause=;	clause=(((upcase(libname)='SASHELP' and upcase(memtype)='DATA') or (upcase(libname)='SASHELP' and upcase(memtype)='VIEW') or (upcase(libname)='SASHELP' and upcase(memtype)='MDDB') or (upcase(libname)='SAMPDAT')) and ((upcase(libname) ne 'SAMPDAT' or upcase(memname) ne 'MYCAT' or upcase(memtype) ne 'CATALOG')))

data null;
length clause $ 32767;
clause=appsrv_authcls('MEMBER');
call symput('CLAUSE',clause);
run;
 	
/*create a data set listing all allowed data sets
excluding views*/
 	
proc sql;
create table work.allowed as
select * from dictionary.tables
where memtype='DATA' and &clause;
quit;
	Data set WORK.ALLOWED is created as a
	subset from dictionary.tables. It contains
	only data sets that are allowed according to
	the AUTHLIB data set. Views are excluded
	from this table by the addition of the
	"memtype='DATA'" clause.

APPSRV_AUTHDS

Enables the user to change the AUTHLIB data set name

Syntax

RC = APPSRV_AUTHDS(*dataset*)

Arguments

dataset
is a character string that is the SAS data set name.

Details

The APPSRV_AUTHDS function enables the user to change the AUTHLIB data set name. The AUTHLIB data set is the table that is examined by the APPSRV_AUTHLIB and APPSRV_AUTHCLS functions. By default, these functions operate on the data set SASHELP.AUTHLIB. Calling the APPSRV_AUTHDS function causes subsequent calls to these functions to use the same data set name. In an Application Server environment, the effect of calling APPSRV_AUTHDS lasts for the duration of the request. To change the AUTHLIB data set name for all requests, call the APPSRV_AUTHDS function in the program that is specified by the INIT argument of the REQUEST statement. The function returns 1 if successful and 0 if unsuccessful.

Examples

SAS Statements	Results
`rc=appsrv_authds('MYLIB.MYAUTH');` `put rc=;`	`rc=1`

APPSRV_AUTHLIB

Determines whether the Application Server program is authorized to access a specified data source

Syntax

RC = APPSRV_AUTHLIB(*libname*, *memname*, *memtype*, *objname*, *objtype*)

Arguments

All arguments to this function are optional.

libname
is a character string that is the SAS libref.
memname
is a character string that is the SAS member name.
memtype
is a character string that is the SAS member type.
objname
is a character string that is the SAS catalog entry name.
objtype
is a character string that is the SAS catalog entry type.

Details

The APPSRV_AUTHLIB function determines if the Application Server program is authorized to access the specified data source. The function returns a value of 1 if authorized and 0 if not authorized. Authorization is determined by the contents of the AUTHLIB data set. This data set contains rules for including and excluding various data sources. For more details on the AUTHLIB data set, see Controlling Access to Data Sources with the AUTHLIB Data Set. An asterisk (*) can be supplied for any of the arguments to mean "any." If an argument is omitted, then an asterisk is assumed.

Examples

For the examples in Table 2, refer to the contents of the SASHELP.AUTHLIB data set in Table 1. ***Entities are excluded by default, and all exclude rules supersede all include rules.***

Table 1: Contents of SASHELP.AUTHLIB Data Set

Rule	Libname	Memname	Memtype	Objname	Objtype
INCLUDE	SASHELP	*	DATA	*	*
INCLUDE	SASHELP	*	VIEW	*	*

INCLUDE	SASHELP	*	MDDB	*	*
INCLUDE	SAMPDAT	*	*	*	*
EXCLUDE	SAMPDAT	MYCAT	CATALOG	*	*

Table 2: Examples

SAS Statements	Results

`rc=appsrv_authlib('SASHELP','RETAIL','DATA');`
`put rc=;`
 	
`/*equivalent to`
`  rc=appsrv_authlib('SASHELP','RETAIL','DATA','*','*');`
`*/`
	`rc=1`

`if (appsrv_authlib('SASHELP','CORE','CATALOG'))`
`  put 'You may proceed ...';`
`else put 'You are not authorized to access this`
`  SAS catalog';`
 	
`/*equivalent to`
`  if (appsrv_authlib('SASHELP','CORE','CATALOG','*','*'))`
`*/`
	`You are not authorized to`
	`  access this SAS catalog.`

`/*Check to see if access to any SCL catalog`
`  entries is allowed*/`
`/*NOTE: A true (1) response does not mean that`
`  you can see ALL SCL entries, just some.*/`
`/*This returns true because some catalogs in`
`  SAMPDAT are included*/`
 	
`rc = appsrv_authlib('*','*','CATALOG','*','SCL');`
`put rc=;`
	`rc=1`

`/*Check to see if access to any of the entries`
`  in the MYDATA.MYCAT catalog is allowed*/`
 	
`rc = appsrv_authlib('SAMPDAT','MYCAT','CATALOG');`
`if (rc = 1) then put 'You can access at least some`
`  of the entries';`
`else put 'Access to this entire catalog is`
`  restricted';`
	`Access to this entire catalog`
	`  is restricted.`

APPSRV_HEADER

The DATA step function used to add or modify a header

Syntax

OLD−HEADER = APPSRV_HEADER(*Header Name*,*Header Value*);

Arguments

Header name
The name of the header to set or reset.
Header Value
The new value for the header.

Details

The APPSRV_HEADER function enables automatic header generation. You can add a header to the default list or modify an existing header from the list. When you modify the value of an existing header, the old value becomes the return value of the function.

The automatic HTTP header generation feature recognizes Output Delivery System (ODS) output types and generates appropriate default content−type headers. If no content type is specified with APPSRV_HEADER, ODS is not used and no HTTP header is written to _WEBOUT, a default `Content-type: text/html` header is generated.

Examples

SAS Statements	Resulting Headers
No calls to appsrv_header	`Content-type: text/html`

`/* add expires header */`
`rc = appsrv_header('Expires','Thu,`
`  18 Nov 1999 12:23:34 GMT');`
	`Content-type: text/html`
	`Expires: Thu, 18 Nov 1999 12:23:34 GMT`

`/* add expires header */`
`rc = appsrv_header('Expires','Thu,`
`  18 Nov 1999 12:23:34 GMT');`
`/* add pragma header*/`
`rc = appsrv_header('Cache-control','no-cache');`
	`Content-type: text/html`
	`Expires: Thu, 18 Nov 1999 12:23:34 GMT`
	`Cache-control: no-cache`

`/* add expires header */`
`rc = appsrv_header('Expires','Thu,`
`  18 Nov 1999 12:23:34 GMT');`
`/* add pragma header*/`
`rc = appsrv_header('Cache-control','no-cache');`
	`Content-type: text/html`
	`Cache-control: no-cache`

```
...
/* remove expires header, rc contains old value */
rc = appsrv_header('Expires','');
```

Disabling Automatic Header Generation

To completely disable Automatic Header Generation for a request, call the APPSRVSET DATA step function, as so:

```
data _NULL_;
   rc = appsrvset("automatic headers", 0);
run;
```

APPSRV_SESSION

Creates or deletes a session

Syntax

RC = APPSRV_SESSION('*command*' <, *timeout*>)

Arguments

command
is the command to be performed. Allowed values are "CREATE" and "DELETE."

timeout
is the optional session timeout. This parameter is valid only when you specify a value of "CREATE" for the command parameter.

Details

The APPSRV_SESSION function creates or deletes a session. The function returns zero for a successful completion. A non−zero return value indicates an error condition.

Examples

SAS Statements
`rc=appsrv_session('create', 600);`
`rc=appsrv_session('delete');`

APPSRV_UNSAFE

Returns the character value of a PROC APPSRV parameter setting

Syntax

VALUE = APPSRV_UNSAFE(*valuecode*)

Arguments

valuecode
is the name of an input name/value pair.

Details

The APPSRV_UNSAFE function returns the complete, original value of an input name/value pair. Normally unsafe characters (see the UNSAFE option) are stripped from input values before creating the input macro variables. This is done so that macro variables may be freely used in a application program without any unwanted SAS macro language processing. In some cases, such as processing free−format input text, you may want to retrieve the complete, original value using the APPSRV_UNSAFE function.

Examples

SAS Statements	Results

`/* In this example assume the recommended UNSAFE option is specified and the user specified a value of "Elwood's Bait &Tackle Shop" in the input HTML form. */`
 	
`safename=symget('company');`
`put safename=;`
`fullname=appsrv_unsafe('company');`
`put fullname=;`
	`safename=Elwoods Bait  Tackle Shop`
	
	`fullname=Elwood's Bait &Tackle Shop`

Application Dispatcher Debugging

Debug flags are not used only for diagnostic purposes. At many sites, it may be necessary to disable debug flags for security reasons. This section defines all of the debug flags, gives you some usage scenarios, and provides diagnostic information. It also helps you determine which debug flags should be disabled and how to identify valid debug values, which options are best suited for the four types of programs that constitute the input component, and which debugging methods is best suited for your needs, (based on the debugging options that are available in the programming component).

Debugging in the Input Component

The special variable _DEBUG provides you with several diagnostic options. Using this variable is a convenient way to debug a problem, because you can supply the debug values by using the Web browser to modify your HTML or by editing the URL in your browser location field.

List of Valid Debug Values

You can activate the various debugging options by passing the _DEBUG variable to the Application Broker just as you pass the special variables _SERVICE and _PROGRAM. You can set more than one debug option by adding the flag values together. For example, to set both the options 2048 and 2, use the value (2048 + 2) = 2050.

Keywords can also be used to set debug options. To set these same options using the keywords, you would specify

```
_DEBUG=TIME,TRACE
```

Multiple parameters (values or keywords) can be specified separated by commas or spaces. These values are then logically put together to form one debug number.

Some debug flags may be locked out at your site in the Application Broker configuration file for security reasons. Verify with your administrator which flags are locked out at your site. See Setting the Default Value of _DEBUG for more information on setting the debug value. The following chart is a list of valid debug values:

Value	Keyword	Description
1	FIELDS	Echoes all fields. This is useful for debugging value−splitting problems.
2	TIME	Prints the Application Broker version number and elapsed time after each run, for example, "This request took 2.46 seconds of real time (v9.1 build 1457)." Also, this value displays the Powered by SAS logo if you provide additional settings as described in Displaying the Powered by SAS Logo.
4	SERVICES	Lists definitions of all services as defined by the administrator, but does not run the program.
8		Skips all execution processing.
16	DUMP	Displays output in hexadecimal. This is extremely helpful for debugging problems with the HTTP header or graphics.
32		Displays the Powered by SAS logo without the Application Broker version or elapsed time information. See also Displaying the SAS Powered Logo.
128	LOG	Returns log file. This is useful for diagnosing problems in the SAS code.
256		Is not used in SAS 9 or later. Previous version socket debug data incorporated into debug 2048.
512		Shows socket host and port number in status message (by default off for security reasons).

1024	ECHO	Echoes data usually sent from the Application Broker to the Application Server. It does not run the program. In the case of a launch service, this also shows the SAS command that would have been invoked by the Application Broker.
2048	TRACE	Traces socket connection attempts. This is helpful for diagnosing the machine communication process.
4096		Prevents the deletion of temporary files that are created for launch. This is useful for debugging configuration problems in a launch service (prior to Version 8).
8192		Returns entire SAS log file from a launched service (prior to Version 8).
16384	ENV	Displays a selection of the Application Broker environment parameters.

Disabling Debug Flags

In the Application Broker configuration file, you can specify the debug values that you are and are not allowed to set. The DebugMask and ServiceDebugMask directives control this. The information below describes the DebugMask directive but it also applies to the ServiceDebugMask directive.

The default value for the DebugMask is 32767. This is adequate for most sites. The value 32767 indicates that all debug values are allowed. This means that commenting out the DebugMask directive is the same as allowing all debug values.

Note: Some debug values may pose a security risk. To avoid potential security risks, selectively disable them by specifying a DebugMask value that is the sum of the values that you want to allow. The safest approach is to set DebugMask to 0, 2, 32, or 34 (sum of 2 and 32). These values do not pose a security risk.

Below is an illustration of how the Application Broker uses DebugMask to restrict certain values. The value 32767 is the sum of all the allowed debug values (1 through 16384). The following chart shows all the bits enabled for the DebugMask=32767 value (in binary notation):

16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

If you want to prevent the use of debug value 2048, disable the DebugMask bit for 2048, as shown next:

16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
1	1	1	0	1	1	1	1	1	1	1	1	1	1	1

To allow all debug values *except* 2048, specify a DebugMask value of 30719 (which is 32767 − 2048).

To allow *only* debug values of 2 and 2048, specify a DebugMask value of 2050 (which is 2 + 2048).

Note: The last technique is safer because additional debug values exist that are not documented. But if you do enable them, these debug values could pose a security risk.

The Application Broker displays an error if the binary values for _DEBUG and DebugMask do not equal 0 using binary logic. The Application Broker performs error checking only on the _DEBUG value in the HTML form or link. It does not check to confirm that Debug and DebugMask do not overlap. In other words, it does not check Debug to see if it has an allowed value in DebugMask. DebugMask is used to check values of _DEBUG only.

Special Cases

Some combinations of content types and debug values do not produce the expected results. The debug values 2, 32, 128, and 8192 do not function correctly if the content type is anything other than TEXT or HTML. For example, if your program sends a content type of IMAGE or GIF, then the browser expects the data that follows to be binary graphic data. If you supply one of these debug values, then the Application Broker tries to append HTML code to the end of the binary graphic data. This HTML is not displayed in the browser; only the image is displayed. One way to work around this is to supply the debug value 1 in addition to the value(s) above. This causes the Application Broker to send the content type TEXT or HTML before your program can send the IMAGE or GIF content type. Because the browser sees the TEXT or HTML content type first, it displays the debug output that you have requested. However, because the binary graphic data is combined with HTML data, garbage characters will take the place of the expected image on the Web page.

Keep in mind that setting the debug variable to 1 or 128 generates an HTTP header. This will affect the behavior of a test that changes the HTTP header in any way.

Some HTML formatter macros create their own headers. For example, if you are using SETCOOKIE on a header, the cookies do not work when DEBUG=1 or DEBUG=128 is set.

Debugging Application Broker Installation Problems

1. To verify that the Application Broker can be executed, run the image from a command line on the Web server machine:

   ```
   broker "_service=default&_program=ping"
   ```

   If the broker.cfg file is set up correctly and there is an Application Server running for service "default", the results will be similar to

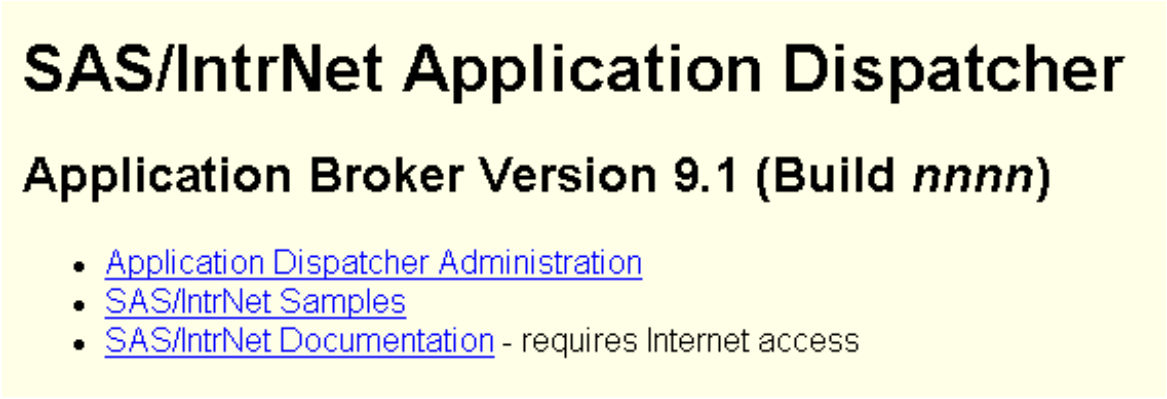
   ```
   Content-type: text/html
   Pragma: no-cache

   Ping. The Application Server your.server.com:5800 is functioning properly.

   ------------------------------------------------------------

   This request took 0.28 seconds of real time (v9.1 build nnnn).
   ```

   Because CGI programs typically run as special nonprivileged users, it is sometimes useful to perform this test with a guest account. On z/OS machines there are occasional problems with the installation of the SAS/C transient library that will be revealed by performing this test.
2. From a browser on the client machine, try to access the Application Broker image via a URL of the form

   ```
   http://yourserver/cgi-bin/broker.exe?
   ```

   The Application Broker program in the URL will be the correct image to use for the Web server machine. If the browser tries to download the program as a file, then you have discovered the problem: the Web server is not configured correctly to execute CGI programs. For UNIX Web servers, typically you must add a line to the Web server HTTP config file for the CGI directory.

   When this command works correctly, the browser displays the following welcome page, which contains a link to an administration page:



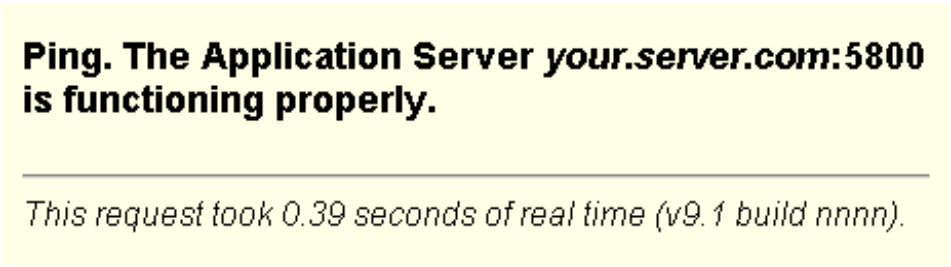
Note: If there is a customized Application Broker welcome page, then it will display instead of this default welcome page when you enter the Application Broker URL in your browser. If this is the case, and if you want to view the services that are available from the default welcome page, then add _DEBUG=4 to the URL, as follows:

```
http://yourserver/cgi-bin/broker.exe?_debug=4
```

3. To test the Applications Server from the browser, enter a URL of the form

```
http://yourserver/cgi-bin/broker.exe?_service=default&_program=ping
```

If the broker.cfg file is set up correctly and there is an Application Server running for service "default", the results will look like this:



If this test fails but the test in the first step succeeds, there is usually a permission problem with the CGI running from the Web server. CGI programs are typically started under a nonprivileged account that may not have access to required system resources. Try totally disabling anonymous access for the CGI directory in the Web server. This forces the Application Broker to run with an authenticated user name. On Windows systems, check the permissions on the entire \winnt\system32 directory tree and verify that the user account IUSR_nodename has the user rights "Access this computer from network" and "Log on locally."

4. For pool service problems, verify that the Load Manager is running and a log is being created. When a pool service request is initiated, the actual command that starts the Application Server is written to the log. It is often useful to copy this command from the log and issue it on a command line to verify that the server starts. If the SAS Spawner is not being used, the Applications Server runs under the username that started the Load Manager with the corresponding privileges and permissions. Occasionally the work directory does not get set correctly and you must add a "–work /tmp" parameter to the SAS command.

   If the server starts but the Application Broker times out, verify that there are not multiple TCP/IP node names defined for a given host. The Application Broker and Load Manager hosts must resolve all node names to the same value. Host name mismatches can cause various error messages to appear in the Load Manager log.
5. To debug problems with launch and pool services, it is useful to obtain the SAS log and investigate any errors concerning the SasCommand that was executed.

To see the SasCommand that is being processed, add the parameter _DEBUG=1024 to the URL that is being used to start the service. This shows any errors with quotes or non−escaped back slashes.

Under UNIX, using the following syntax puts the SAS log, STDOUT, and STDERR messages into files using the Application Broker PID for the file names:

```
SasCommand "/bin/ksh -c '/usr/local/bin/sas  /user/web/launchsas.sas +
   -rsasuser -noterminal -noprint -log $$.log  -SYSPARM $$.out 2>$$.err'"
```

Note: The username that the Application Broker runs under must be able to write to the directory where the Application Broker runs.

Debugging in the Program Component

There are four techniques for debugging Dispatcher programs:

- Examining the SAS Log
- Using SAS Options
- The DATA Step Debugger
- The SCL Debugger

Examining the SAS Log

Passing a name/value pair of _DEBUG=128 to the Dispatcher will cause the SAS log to be returned to the Web browser. This is useful because you will be able to check for errors, which typically show up in the log. The Dispatcher will highlight them in red. If the code that you are debugging contains macro statements, turn on various options such as MPRINT and MLOGIC. Sometimes the Dispatcher will return an error message that suggests to send a debug value of 131. This is the values 1, 2, and 128 combined. See the List of Valid Debug Values section for a complete list of debug values.

If you are unable to retrieve the SAS log with _DEBUG=128, then you should contact your Application Server administrator. The administrator may need to examine the SAS log file that the Application Server writes to disk. This file can often contain more information than is displayed in your browser. If you are unable to determine the problem by examining this log file, see FAQs & Troubleshooting (support.sas.com/rnd/web/intrnet/misc/support.html).

Using SAS Options

There are several SAS options that can help you debug problems in your Dispatcher programs. If you can return the SAS log to your browser, activating some of these options can make that log more useful. If you are debugging a program that contains macro code, you should supply one or more of these options at the beginning of your program: `MPRINT`, `SYMBOLGEN`, `MLOGIC`, `MERROR`.

If, for security reasons, you have disabled the display of submitted source code in your program using the NOSOURCE option when you are debugging, you should enable this feature by supplying the `SOURCE` option. You can then see your submitted SAS code in the log that is returned to your browser. After you are done, you can revert to using `NOSOURCE` if your security model requires it.

The DATA Step Debugger

To use the DATA step debugger, you must start the Application Server in the SAS windowing environment, and you must be working on the computer where SAS software will display the debugger windows. To debug a DATA step, add `/debug` to the DATA statement, like this:

```
data mylib.mydata/debug;
```

When the Application Server executes the program, the DATA step debugger windows will pop up and pause, waiting for you to step through the code.

The SCL Debugger

To use the SCL debugger, you must start the Application Server with the `AFPARMS='debug=yes'` option. The SCL program that you want to debug must be compiled with the debug option. The Application Server must be

running in the SAS windowing environment, and you must be working on the computer where SAS software will display the debugger windows. When the Application Server executes the SCL program, the debugger windows will appear and allow you to step through the code.

The APPSRV Procedure

The APPSRV procedure invokes the Application Server, which is the server component of the SAS/IntrNet: Application Dispatcher. It is recommended that you use the inetcfg utility to set up a new Application Server. This utility creates a standard PROC APPSRV statement with reasonable default options that can be modified to meet your requirements. See the syntax documentation below for more information about these options.

Syntax

PROC APPSRV PORT=*n* <*options*>;

ADMINLIBS *libref–1* | *libref–1.catalog–1* | *fileref–1* <*...libref–n* | *libref–n.catalog–n* | *fileref–n*>;

ALLOCATE FILE *fileref* <*device–type*> '*directory–or–PDS–path*' <*host–options*>;

ALLOCATE LIBRARY *libref* <*engine*> '*SAS–data–library*' <*options*>;

DATALIBS *libref–1* | *fileref–1* <*...libref–n* | *fileref–n*>;

LOG <DISPLAY=NONE | ERRORS | ALL> <SYMBOLS=NONE | ERRORS | ALL> <FILE=*fileref*> <APPEND | REPLACE>;

PROGLIBS *libref–1* | *libref–1.catalog–1* | *fileref–1* <*...libref–n* | *libref–n.catalog–n* | *fileref–n*>;

REQUEST <INIT=*program–name*> <TERM=*program–name*> <LOGIN=*program–name*> <TIMEOUT=*seconds*> <MAXTIMEOUT=*seconds*> <READ=*seconds*> <FROMADR=("*IP–address–1*" <..."*IP–address–n*">)>;

SESSION <INIT=*program–name*> <INVSESS=*program–name*> <TERM=*program–name*> <TIMEOUT=*seconds*> <MAXTIMEOUT=*seconds*> <VERIFY=(*variable–1* <*...variable–n*>)>;

STATISTICS CREATE=*library.dataset* <(*data–set–options*)>;

STATISTICS DATA=*library.dataset* <(*data–set–options*)> <ADDPORT> <EXITONERROR> <TEMPLATE=*library.dataset* <(*data–set–options*)>> <WRITECOUNT=*n*> <WRITEEVERY=*n*>;

To do this	Use this statement
Declare which libraries, filerefs, and catalogs contain programs that can be run by an administrator using the _ADMINPW password	ADMINLIBS
	ALLOCATE FILE

Define a file that the Application Server assigns	
Define a library that the Application Server assigns	ALLOCATE LIBRARY
Define librefs and filerefs that are available to all programs that are run by the Application Server	DATALIBS
Control content and behavior of the Application Server log	LOG
Declare which libraries, filerefs, and catalogs contain programs that can be run on the Application Server	PROGLIBS
Control how a request is processed by the Application Server	REQUEST
Control how a session is administered by the Application Server	SESSION
Control writing of request statistics to a data set	STATISTICS

PROC APPSRV Statement

PROC APPSRV PORT=*n* <*options*>;

Option	Definition
ADMINPW='*password*'	The optional server administration password. This option does not have a default setting.
AFPARMS='*string*'	An optional quoted string of parameters that are passed when invoking SAS/AF to run SCL programs. Users will pass AFPARMS='debug=yes' to invoke the SCL debugger.
AUTH=*scheme*	The authentication scheme. The two values that can be used with this option are HOST (denotes a secure Application Server) or NONE. The default is NONE.
ENCODING=*encoding-value*	The default character−set encoding for all data sent to and received from the Application Broker.
GUESTP2='*password*'	An optional second password to use for guest access.
GUESTPASS='*password*'	The password to use for guest access.
GUESTUSER='*username*'	The username to use for guest access.
LOCALIP=*IP−address*	A manual override for GETSOCKNAME.
LRECL=*n*	The logical record length for _WEBOUT and _GRPHOUT filerefs.
NETBUFFK=*n*	The buffer size (in kilobytes) for _WEBOUT and _GRPHOUT output buffering.
NOSHAREPOLL	Disables polling of the SAS/SHARE server librefs.
PORT=*n*	The *only required option*. The port number or name. Zero is used for dynamic ports. PORT=*n* does not have a default setting.
PROGRAMS=*n*	The maximum number of requests that can run concurrently. The default setting is 1.
SHAREPOLL=*n*	controls the period of SAS/SHARE server libref polling. The period is equal to *n*, which is a positive integer representing seconds. The default setting is 300 seconds (5 minutes).
UNSAFE='*string*'	An optional list of characters that when used, enhances security by compressing name/value pairs.

PROC APPSRV Arguments

ADMINPW=password

allows the user to restrict access to specific administrator programs. The server has several built−in programs, such as STATUS and STOP. If ADMINPW is specified, the user must supply the password in the request (using the _ADMINPW variable) in order to run the STOP program. When the request is received, the server performs the following tasks:

- ◊ verifies that the request is one of the administrator programs
- ◊ searches the request data for the variable _ADMINPW
- ◊ determines if the variable value matches the ADMINPW password that is specified in the PROC APPSRV statement
- ◊ if it is a match, the request is returned; if it is not a match, the request is rejected.

In addition to built−in programs, the ADMINLIBS statement can be used to declare various librefs and filerefs as containing administrator−only programs. Programs in these libraries are not executed unless _ADMINPW is passed and is verified.

Note: If a libref or fileref has been defined in both a PROGLIBS statement and an ADMINLIBS statement, then the ADMINPW is not required for programs in that libref or fileref. General users will have access to programs that might have been intended only for administrators.

AFPARMS='string'

is a quoted string that is appended to the SAS/AF command when users invoke user programs that are written in SCL. It can be used to pass a variety of parameters to the SAS/AF environment, but the primary use in the Application Server is to enable the SCL debugger. To invoke the SCL debugger, compile your SCL program with debug on and then start the server with

```
AFPARMS='debug=yes'
```

AUTH=scheme

specifies the authentication scheme. The default scheme (AUTH=NONE without GUESTUSER being specified) causes all requests to be run with the credentials of the username under which PROC APPSRV was started. Specifying GUESTUSER (and the corresponding GUESTPASS) with the default AUTH=NONE scheme causes all requests to be run with the credentials of the GUESTUSER username. All access to catalogs, datasets, and external files are checked against this username. Note that the AUTH=HOST special requirements listed below also apply to AUTH=NONE when GUESTUSER and GUESTPASS are specified.

The AUTH=HOST scheme requires a username and password with each request, which will run using the credentials of the authenticated username. All access to catalogs, datasets, and external files are checked against this username. The username and password can be specified with the reserved variables _USERNAME and _PASSWORD (and optionally _PASSWORD2). The GUESTUSER and GUESTPASS (and optionally GUESTP2) options can be used to specify default values if they are not specified with the request. If the username is not specified by either the _USERNAME variable or by the GUESTUSER option, the request is rejected (unless the LOGIN option is used.) Usernames and passwords are saved with sessions, so requests that connect to an existing session do not need to and cannot specify a new username and password.

The AUTH schemes do not apply to administration programs. Unprotected administration programs such as PING and STATUS can be run by any client without specifying a username or password. Protected administration programs such as STOP require only the _ADMINPW parameter (for more details, see the ADMINPW option). ADMINPW is required if AUTH=HOST is specified.

See the Special Requirements section for more information about the AUTH=HOST option.

ENCODING=encoding−name

specifies the default character−set encoding for all data sent to and received from the Application Broker. This option is not normally required unless the Web server uses a different encoding from the one used by the Application Server. PROC APPSRV ENCODING defaults to the appropriate Windows encoding regardless of the platform. The default output encoding is automatically set based on the SAS session encoding. The SAS session encoding is normally determined by the locale setting of your SAS installation, but may be set directly using the SAS ENCODING option.

The following are the default Windows SAS encodings based on the Application Server's locale.

SAS Locale	Default PROC APPSRV ENCODING
Western Europe and the Americas	wlatin1
Eastern Europe	wlatin2
Cyrillic	wcyrillic
Japanese	ms−932

Encodings whose names include a dash (–) must be enclosed in quotation marks (').

GUESTP2='password'

See AUTH. This option is used only in OpenVMS environments because OpenVMS can accept two passwords.

GUESTPASS='password'

See AUTH.

GUESTUSER='username'

See AUTH.

LOCALIP=IP–address

allows you to manually override the local IP address used by the Application Server. In rare cases, the local IP address returned by the operating environment is not usable, and a manual override is necessary.

LRECL=n

is the logical record length for _WEBOUT and _GRPHOUT filerefs. The default is `65535`.

NETBUFFK=n

is the buffer size in kilobytes (KB) for _WEBOUT and _GRPHOUT output buffering. The buffer size must be a value between 4 and 128. Output buffering is disabled by default. Use of this option is not recommended without consulting SAS Technical Support.

NOSHAREPOLL

disables polling of the SAS/SHARE server librefs. This option cannot be used at the same time as the SHAREPOLL= option.

PORT=n

specifies the request socket for the Application Server.

- ◊ If a numeric value other than zero is supplied, the value is used as the TCP/IP port number on which the server listens for requests.
- ◊ If an alphanumeric value is supplied, it is assumed to be a network service name. The name is searched in the system services file (for example, /etc/services) and translated to a port number.
- ◊ If zero is supplied, PROC APPSRV chooses an available port. This feature is used only for launch or pool services.

PROGRAMS=n

specifies the maximum number of requests that can execute concurrently. The default setting is 1.

Note: This option should *not* be used if PROC APPSRV is run in the SAS windowing environment.

SHAREPOLL=n

controls the period of SAS/SHARE server libref polling. The period is equal to *n*, which is a positive integer representing seconds. The default setting is 300 seconds (5 minutes). The SHAREPOLL setting should be interpreted as the minimum amount of time between polls of the SAS/SHARE server. SHARE polling has a lower priority than the servicing of client requests so in periods of high client activity the SHARE polling will be delayed beyond the period specified by *n*. The SHAREPOLL= option cannot be used at the same time as the NOSHAREPOLL option.

UNSAFE='string'

specifies a quoted string listing characters that should be stripped from values in the request data (the name/value pairs). This option is normally used to strip characters from input values that could cause unwanted SAS macro language processing.

The characters that users most often want to mark as unsafe are the following:

- ◊ single quotation mark
- ◊ double quotation mark
- ◊ ampersand
- ◊ percent
- ◊ semicolon.

Because this list is enclosed by single quotation marks, you can represent a single quotation mark by placing

two single quotation marks within the quoted string in the following manner:

```
UNSAFE='&"%;'''
```

There are times, such as processing free−format text input, when you might want to use the original, complete value for an input name/value pair. The APPSRV_UNSAFE function can be used for this purpose. For example, the complete text of an input variable named MYTEXT can be accessed in a DATA step or SCL program with APPSRV_UNSAFE, as in the following:

```
fulltext = appsrv_unsafe('MYTEXT');
```

The APPSRV_UNSAFE function can be called from macro with the %sysfunc function:

```
%let fulltext = %sysfunc(appsrv_unsafe(MYTEXT));
```

Note: If you are using programs developed before Version 8 of SAS, you may need to omit the UNSAFE option for proper operation of your application. If the UNSAFE option is not specified, no unsafe processing is performed and all name/value pairs are passed unmodified to the request program.

Special Requirements for AUTH=HOST

Using AUTH=HOST on OpenVMS systems

The AUTH=HOST option requires that the account that is running PROC APPSRV must have SYSPRV privilege enabled to allow the server to verify login information. Note that all client requests will be rejected as invalid if the server account does not have this privilege.

Using AUTH=HOST on z/OS systems

The AUTH=HOST option requires that the SAS SVC routine be installed on z/OS systems. The SAS SVC control program routine is an interface between the z/OS operating environment and a specific request, such as third−party checking. This facility provides verification in the form of calls for authentication of both the user ID and password and of library authority. Perform the following steps before using the AUTH=HOST option.

1. Install the SAS SVC routine, if necessary.
    - If you have already installed the SAS SVC routine for SAS 9.1, do not repeat the step here. If you need to perform the installation, see the installation instructions for SAS under z/OS at support.sas.com/installcenter for details.
    - Because SAS SVC 9.1 is backward compatible, it replaces the SAS SVC routines from previous releases. You can continue using previous releases of Base SAS and SAS/IntrNet or SAS/SHARE with SAS SVC 9.1.
2. Verify the SAS options for the SVC routine.
    - You must verify that the SAS options for the SVC routine accurately reflect the way that the SAS SVC is installed. The SAS option SVC0SVC should be set to the number at which the SAS SVC is installed (for example, 251 or 109). If the SAS SVC is installed at 109 as an ESR SVC, set the SAS option SVC0R15 to the ESR code (for example, 4).
3. Verify installation on all CPUs, as needed.
    - If you have more than one CPU, verify that the SAS SVC routine is installed on the systems that will be running the Application Server at your site.

Using AUTH=HOST on UNIX systems

The AUTH=HOST option requires that the SAS User Authorization utilities (`sasauth` and `sasperm`) be configured properly. See the section on configuring user authorization in the SAS 9.1 post–installation instructions for UNIX at support.sas.com/installcenter for more information on these utilities.

Using AUTH=HOST on Windows systems

The AUTH=HOST option requires special user rights on Windows systems. Review the following requirements carefully before enabling the AUTH=HOST option.

- Any username specified by a client (including the default GUESTUSER) must have **Log on as a batch job** advanced user right enabled. If this permission is not enabled, the client request is rejected as an invalid login.
- On Windows NT and Windows 2000 only, the account that is running PROC APPSRV must have **Act as part of the operating system** advanced user right enabled to allow the server to verify login information. Note that all client requests are rejected as invalid if the server account does not have this permission.

ADMINLIBS Statement

Declares which libraries, filerefs, and catalogs contain programs that can be run by an administrator using the _ADMINPW password

Syntax

ADMINLIBS *libref−1* | *libref−1.catalog−1* | *fileref−1* <*...libref−n* | *libref−n.catalog−n* | *fileref−n*>;

Arguments

Libraries, filerefs, and catalogs listed here can be run on the Application Server only if a valid _ADMINPW value is passed in the request data *and* a password is specified.

Note: If a libref or fileref has been defined in both a PROGLIBS statement and an ADMINLIBS statement, then the ADMINPW is not required for programs in that libref or fileref. General users will have access to programs that might have been intended only for administrators.

libref−1
specifies a library that contains one or more catalogs that contain programs that can be run by the Application Server. Programs must be SCL, SOURCE, or MACRO catalog entries.

libref−1.catalog−1
specifies a catalog that contains SCL, SOURCE, and MACRO programs that can be run by the Application Server. If a libref is listed as a data library and a program library, then the library is globally available and can contain programs. If you want to enable programs from one catalog in a given library to be run without enabling everything in the library to be run, then list just that catalog in a two−level name as in the following example:

```
ADMINLIBS MYLIB.MYCAT . . .;
```

Listing both the library and a specific catalog within that library is redundant. For example,

```
ADMINLIBS MYLIB MYLIB.MYCAT . . .;
```

enables all programs in MYLIB to run.

fileref−1
specifies a host directory or PDS that contains SAS programs that can be run by the Application Server.

...libref−n | libref−n.catalog−n | fileref−n
specifies that you can list multiple librefs, catalogs, and filerefs for this statement.

Note: See also the ADMINPW option of the PROC APPSRV statement.

ALLOCATE FILE Statement

Defines a file that the Application Server assigns

Syntax

ALLOCATE FILE *fileref* <*device−type*> '*directory−or−PDS−path*' <*host−options*>;

Note: The syntax of the ALLOCATE FILE statement is identical to that of the global FILENAME statement. The above syntax is simplified. For a complete listing of arguments and explanations, see the FILENAME statement in *SAS Language Reference: Dictionary*.

Arguments

fileref

associates a SAS fileref with an external file or directory. You can use any SAS name when you are assigning a new fileref. You can list SAS filerefs that are defined in the ALLOCATE FILE statement in a DATALIBS, PROGLIBS, or ADMINLIBS statement. SAS filerefs that are listed in a DATALIBS statement are available to all programs that are run by the Application Server.

Note: SAS filerefs that are defined outside PROC APPSRV by using FILENAME statements are not accessible by Application Server programs and cannot be listed in a DATALIBS, PROGLIBS, or ADMINLIBS statement.

device−type

specifies the type of device. Values include

DISK

specifies that the device is a disk drive. When assigning a fileref to a file on a disk, you are not required to specify DISK.

TAPE

specifies a tape drive.

DUMMY

specifies a bit bucket or null device.

directory−or−PDS−path

specifies a directory or partitioned data set (PDS) that is the pathname for a SAS fileref that is used in a PROGLIBS or ADMINLIBS statement. A directory is assumed to contain SAS source code in individual .sas flat files. A PDS is assumed to contain SAS source code in individual members. You must enter the directory or path in one of the following forms, depending on which operating environment the SAS server is using:

Operating Environment	Example Directory or Path
UNIX	/u/jdoe/samples
Windows	C:\samples
z/OS (HFS directory)	/u/jdoe/samples
z/OS (PDS)	SAS.INTRNET.SAMPLES

host−options

indicates host− and device−specific details, such as file attributes and processing attributes. For details about

host and device options, see *SAS Language Reference: Dictionary* and the SAS documentation for your operating environment.

ALLOCATE LIBRARY Statement

Defines a library that the Application Server assigns

Syntax

ALLOCATE LIBRARY *libref* <*engine*> '*SAS−data−library*' <*options*>;

Note: The syntax of the ALLOCATE LIBRARY statement is identical to that of the global LIBNAME statement. The above syntax is simplified. For a complete listing of arguments and explanations, see the LIBNAME statement in *SAS Language Reference: Dictionary*.

Arguments

libref

associates a SAS libref (shortcut name) with a SAS data library. The libref specifies either the name of an existing server library or the name of a new library reference that is defined when you enter this statement. You can list SAS librefs that are defined in the ALLOCATE LIBRARY statement in a DATALIBS, PROGLIBS, or ADMINLIBS statement.

SAS librefs that are listed in PROGLIBS or ADMINLIBS statements are assumed to contain catalogs that contain SAS programs that can be executed by the Application Server. The SAS programs can be SOURCE, MACRO, or SCL catalog entries.

SAS librefs that are listed in a DATALIBS statement are available to all programs that are run by the Application Server.

Note: SAS librefs that are defined outside PROC APPSRV by using LIBNAME statements are not accessible by Application Server programs and cannot be listed in a DATALIBS, PROGLIBS, or ADMINLIBS statement.

engine

specifies the name of a valid SAS engine that you want to use to access the server library. Specify this option only if you want to override the SAS default for a specific server, or if you want to reduce the time that is needed for the client to determine which engine to use to access a specific server.

SAS−data−library

must be a valid physical name for the SAS data library on your host system. You must enclose the physical name in single or double quotation marks.

The physical name of the SAS data library is the name that is recognized by the operating environment.

options

See the LIBNAME statement in the *SAS Language Reference: Dictionary* for a complete list of options.

Nesting Library Names in Concatenated Libraries

Concatenated Data Libraries

You must list all data libraries that are nested in a concatenated library in DATALIBS.

Single−level nested data libraries work properly regardless of the order of the libraries in the DATALIBS statement. For example,

```
PROC APPSRV;
   ALLOC LIBRARY ONE '/path/one';
   ALLOC LIBRARY TWO ('/path/two' ONE);
   DATALIBS ONE TWO;
```

works whether the order of the libraries is coded as `DATALIBS ONE TWO` or `DATALIBS TWO ONE`.

Multilevel nested libraries work *only* if the order in the DATALIBS statement is correct. The following code does not work because library THREE is assigned before library TWO:

```
PROC APPSRV;
   ALLOC LIBRARY ONE '/path/one';
   ALLOC LIBRARY TWO ('/path/two' ONE);
   ALLOC LIBRARY THREE ('/path/three' TWO);
   DATALIBS THREE TWO ONE;
```

Instead, use the following code:

```
DATALIBS ONE TWO THREE;  /* or DATALIBS TWO ONE THREE; */
```

ALLOC statements are order dependent. PROC APPSRV performs an automatic check on library and file assignments during its startup phase. The code

```
PROC APPSRV;
   ALLOC LIBRARY TWO ('/path/two' ONE);
   ALLOC LIBRARY ONE '/path/one';
```

fails because library ONE is not defined when the library TWO assignment is tested. This happens regardless of how the libraries are listed in the DATALIBS, PROGLIBS, or ADMINLIBS statements. Remember that the syntax is identical to that of a LIBNAME statement in SAS open code.

Every library that is used must be defined as a data library. The following code does not work because library ONE is not defined as a data library:

```
PROC APPSRV;
   ALLOC LIBRARY ONE '/path/one';
   ALLOC LIBRARY TWO ('/path/two' ONE);
   DATALIBS TWO;
```

Concatenated Program Libraries

Nested program libraries generally do not work as expected. For example, the following code does not work when you attempt to run a program in library TWO. This is because library ONE is not assigned in the request executive when you attempt to assign library TWO.

```
PROC APPSRV;
   ALLOC LIBRARY ONE '/path/one';
   ALLOC LIBRARY TWO ('/path/two' ONE);
   PROGLIBS TWO ONE;
```

You must change library ONE to a DATALIB by using the following code:

```
PROC APPSRV;
   ALLOC LIBRARY ONE '/path/one';
   ALLOC LIBRARY TWO ('/path/two' ONE);
   DATALIBS ONE;
   PROGLIBS TWO;
```

The same problem can occur when you use ADMINLIBS. This can cause the most confusion because it is not always obvious what can be causing the problem.

DATALIBS Statement

Defines librefs and filerefs that are available to all programs that are run by the Application Server

Syntax

DATALIBS *libref–1* | *fileref–1* <*...libref–n* | *fileref–n*>;

Assign these logical libraries in an ALLOCATE statement in the same server procedure. Any libraries that are defined externally to SAS (such as, in JCL code) are automatically permanent data libraries and should not be listed in the DATALIBS statement. In previous versions of the Application Server, global data libraries or files were allocated in the permdata.sas file. The Application Server now enables you to

- assign the logical libraries externally to SAS
- allocate them with an ALLOCATE statement and then list them in the DATALIBS statement.

Arguments

libref–1
specifies that the libref is assigned and accessible to all programs that run on the server. The libref cannot be cleared by the user code. Use DATALIBS for globally accessible data repositories that contain non–sensitive data. Keep private or application–specific data in its own library and assign it by using a LIBNAME statement.

fileref–1
specifies that the fileref is assigned and accessible to all programs that run on the server. The fileref cannot be cleared by the user code. Use DATALIBS for globally accessible data repositories that contain non–sensitive data. Keep private or application–specific data in its own file and assign it by using a FILENAME statement.

...libref–n* | *fileref–n
specifies that you can list multiple librefs and filerefs for this statement.

LOG Statement

Controls content and behavior of the Application Server log

Syntax

LOG <DISPLAY=NONE | ERRORS | ALL> <SYMBOLS=NONE | ERRORS | ALL> <FILE=*fileref*> <APPEND | REPLACE>;

The Application Server has several options to control the content and operation of the SAS log. The SAS log can be re−directed to a new file based on the date, the day of the week, or the time. The log contains information about each client request. The log can be limited to a brief note for each request, or can capture the complete SAS log for the request.

By default, the configuration utility (inetcfg) sets up the server so that a new log file is created each day of the week. Separate log files are created for each unique port so that there are no conflicts when two or more servers are active. Each time a server is started, it appends to an existing log file unless the log file has not been modified in the last six days. If the log file is at least six days old, it is replaced by a new log file. See Default Log File Append Behavior for a more complete description. You can change this behavior by editing the LOG statement in the appstart.sas file that is created by the configuration utility and by using the options described below.

Note: z/OS Application Servers always append to existing log files regardless of their last modified date unless the REPLACE option is specified.

Arguments

DISPLAY=NONE | ERRORS | ALL
controls whether the SAS log for each client request is written to the Application Server log. The request log can be ignored for all requests, written only for requests that complete with errors, or written for all requests.

SYMBOLS=NONE | ERRORS | ALL
controls whether the client request symbols are written to the Application Server log. The symbols can be logged for all requests, logged only for requests that complete with errors, or never logged.

FILE=fileref
enables you to re−direct the SAS log file of the Application Server to another file. The *fileref* should be defined in an ALLOCATE FILE statement. The physical path of the fileref can contain any of the following date and time directives.

Date and Time Directives	
%a	Day of week [Sun − Sat]
%b	Month [Jan − Dec]
%d	day [01 −31]
%H	hour [00 − 23]
%m	month [01 − 12]
%w	day of week [1=Sunday − 7=Saturday]
%Y	full year
%y	2−digit year [00 − 99]

%p	port number of listen port
%n	nodename up to first period (.)

For example:

```
allocate file one "/u/username/%a_%p.log";

...

log file=one;
```

creates /u/*username*/Mon_5001.log if the Application Server starts on a Monday, /u/*username*/Tue_5001.log if it starts on a Tuesday, and so on.

Periodically, the Application Server regenerates the log file name and checks to see if it is different from the current log file. If it is different, the current log file is closed, and the new log file with the new name is opened. In the previous example, shortly after midnight, early Tuesday morning, the log file /u/*username*/Mon_5001.log is closed and the file /u/*username*/Tue_5001.log is opened.

Note: On z/OS, this feature is supported only if the log file is specified as a hierarchical file system (HFS) path, as shown in the previous ALLOCATE FILE statement. (In order to correctly specify an HFS fileref for an Application Server on z/OS, SAS must be started with the HFS option.) You cannot use partitioned data set members for log files on z/OS.

APPEND | REPLACE

specifies whether the Application Server *always* appends (APPEND) to an existing log file or replaces (REPLACE) the contents of an existing log file.

Default Log File Append Behavior

The Application Server has special default behavior to simplify the management of server logs. If neither the APPEND nor the REPLACE options are specified, the server replaces the contents of an existing log file if the last modification date is greater than six days ago (actually, 5 days, 23 hours). If the last modification date is less than six days ago, the server appends to the existing log file.

For example, if log file Mon_5001.log has a last modification date of 5:00 p.m., Monday, June 14, and the Application Server is re−started at 8:00 p.m. on the same day, the server appends to the existing log. If the server is restarted on Monday, June 21, the server replaces the contents of the log file. This behavior, together with the service files that are created by the inetcfg utility, ensures that server logs are kept for six days and then are automatically overwritten.

Note: z/OS Application Servers always append to an existing log file unless the REPLACE option is specified. In addition, you cannot use partitioned data set members for log files on z/OS.

PROGLIBS Statement

Declares which libraries, catalogs, and filerefs contain programs that can be run on an Application Server

Syntax

PROGLIBS *libref–1* | *libref–1.catalog–1* | *fileref–1* <...*libref–n* | *libref–n.catalog–n* | *fileref–n*>;

When a request is received by the Application Server, the PROGLIBS list is scanned for a match on the first one or two levels in the program name that is supplied in the special request variable _PROGRAM. If a match is found, then the program is executed.

Arguments

libref–1
specifies a library that contains one or more catalogs that contain programs that can be run by the Application Server. Programs must be SCL, SOURCE, or MACRO catalog entries.

libref–1.catalog–1
specifies a catalog that contains SCL, SOURCE, and MACRO programs that can be run by the Application Server. If a libref is listed as a data library and a program library, then the library is both globally available and can contain programs. If you want to enable programs from one catalog in a specified library to be executed without enabling everything in the library to be executed, then list just that catalog in a two–level name like this:

```
PROGLIBS MYLIB.MYCAT . . .;
```

Listing both the library and a specific catalog within that library is redundant. For example:

```
PROGLIBS MYLIB MYLIB.MYCAT . . .;
```

enables all programs in MYLIB to run.

fileref–1
specifies a fileref that corresponds to a host directory or a PDS that contains SAS programs that can be executed by the Application Server.

...libref–n | libref–n.catalog–n | fileref–n
specifies that you can list multiple librefs, catalogs, and filerefs for this statement.

REQUEST Statement

Controls how a request is processed by the Application Server

Syntax

REQUEST <INIT=*program−name*> <TERM=*program−name*> <LOGIN=*program−name*> <TIMEOUT=*seconds*> <MAXTIMEOUT=*seconds*> <READ=*seconds*> <FROMADR=("*IP−address−1*" <..."*IP−address−n*">)>;

Arguments

INIT=program−name
specifies the name of a program to run before each requested program. By default, no program is run before each request.

TERM=program−name
specifies the name of a program to run after each requested program. By default, no program is run after each request.

LOGIN=program−name
specifies the name of a program to run when the server is running with AUTH=HOST (a secure Application Server) and when _USERNAME and _PASSWORD are missing or incorrect. If the option is omitted, the user will receive a default response stating that the login information is missing or incorrect.

TIMEOUT=seconds
specifies the number of seconds that a requested program is allowed to run before it is terminated by the server. By default, the TIMEOUT is set to 300 (5 minutes). This value can be changed in a request program by calling the Application Dispatcher APPSRVSET function.

MAXTIMEOUT=seconds
is the maximum number of seconds that a timeout can be set to using the APPSRVSET function. The default value is 900 (15 minutes). For more information about setting the request timeout from a request program, see the SAS/IntrNet: Application Dispatcher documentation for the APPSRVSET function.

READ=seconds
sets the number of seconds the server waits for a request to be read. The default value for READ is 30 seconds. The majority of requests are read in less than one second.

FROMADR=("IP−address−1" <..."IP−address−n">)
specifies a space−delimited list of IP addresses from which the server accepts requests. Each address *must* be enclosed in quotation marks. By default, requests are accepted from any address. This option accepts numeric IP numbers only. Names and wildcards in addresses are not supported. Enclose each IP address in quotation marks. Separate the IP addresses from each other with a white space and enclose the complete list in parentheses.

SESSION Statement

Controls how a session is administered by the Application Server

Syntax

SESSION <INIT=*program−name*> <INVSESS=*program−name*> <TERM=*program−name*> <TIMEOUT=*seconds*> <MAXTIMEOUT=*seconds*> <VERIFY=(*variable−1* <...*variable−n*>)>;

Arguments

INIT=program−name
specifies programs to be run when a session is created and destroyed (including those that expire by timing out). By default, no program is run at the creation and destruction of a session. The program names referenced here must be in the same format as the _PROGRAM variable. Also, the libraries or files that contain these programs must be allocated in a previous ALLOCATE statement.

INVSESS=program−name
specifies a program that is to be run in the place of the requested program if the session that is specified by _SESSIONID does not exist. This can happen if the session expired or if the session ID was modified by the client.

Two special macro variables are created for the invalid session program. The _USERPROGRAM variable contains the name of the program that was requested by the user. This is the value that is specified by the _PROGRAM variable in the original request. The _INVSESSREASON variable has `NOSESSION` as its value, which means that the session specified by _SESSIONID does not exist.

The invalid session program can be used to display an informative response when a user session has expired or is otherwise inaccessible. The response can redirect the user to an application login screen, explain how to restart the application, or provide a friendlier error message.

TERM=program−name
specifies programs to be run when a session is created and destroyed (including those that expire by timing out). By default, no program is run at the creation and destruction of a session. The program names referenced here must be in the same format as the _PROGRAM variable. Also, the libraries or files that contain these programs must be allocated in a previous ALLOCATE statement.

Remember that when you delete a session, it is only *marked* for deletion. A session is not deleted until the clean−up routine runs. A user creates a session only once throughout an application. The user can reuse the session, but deletion of the session does not occur until the end of the application.

In the following example, a user creates a session and then deletes that session. When the user tries to create a new session in the same test program they get a warning.

```
testa.sas (creates session1 -> calls testb.sas)
testb.sas (uses session1 -> deletes session1 -> creates new session2)
```

The user cannot create session2, because session1 is still being used. Furthermore, after a session is marked for deletion, another user cannot access that same session, even before the clean−up process runs.

TIMEOUT=seconds
specifies the number of seconds that elapse before a session expires. The default session timeout is 900 (15 minutes). This value can be changed in a request program by calling the Application Dispatcher APPSRVSET

function. An Application Server does not honor a pool service idle timeout stop request from the Load Manager until all sessions have expired.

MAXTIMEOUT=seconds

is the maximum number of seconds that a timeout can be set to using the APPSRVSET function. For more information about setting the session timeout from a request program, see the SAS/IntrNet: Application Dispatcher documentation for the APPSRVSET function.

VERIFY=(variable−1 <...variable−n>)

is a space−delimited list of variable names. A *session reconnect* is a request for a _SESSIONID for an existing session. For enhanced security, the Application Server can verify other request variables in addition to validating the _SESSIONID for all session re−connects. For example, the Application Server can ensure that the variable _RMTUSER is the same for all session re−connects. This makes it more difficult for one client to steal another client's URL and access the first client's session information. Enclose the list of variables in parentheses.

STATISTICS Statement

Controls writing of request statistics to a data set

Syntax

STATISTICS CREATE=*library.dataset* <(*data−set−options*)>;

STATISTICS DATA=*library.dataset* <(*data−set−options*)> <ADDPORT> <EXITONERROR> <TEMPLATE=*library.dataset* <(*data−set−options*)>> <WRITECOUNT=*n*> <WRITEEVERY=*n*>;

Server administrators can use the default data set variables that are supplied by the Application Server, or they can modify the variables that are written to the data set, by removing default variables and adding variables of their own.

Arguments

CREATE=library.dataset <(data−set−options)>

creates the specified data set with the default set of statistics variables. The use of the CREATE= option is NOT required. The Application Server (when started with the STATISTICS DATA=library.dataset statement) will create a default data set even if the CREATE= option is not specified.

This option is only used in a standalone SAS session. Do NOT use it when starting the Application Server for application processing. The CREATE= option creates an empty SAS data set containing the default statistics data set variables. The data set can then be modified using a DATA step (to add new variables or drop existing default variables). The modified data set can then be used with the STATISTICS DATA= or TEMPLATE= option. See the Customizing the Statistics Data Set for an example of how to create and modify the statistics dataset.

DATA=library.dataset <(data−set−options)>

specifies the data set to which statistics are written. These data set options are not required. If the specified data set exists, it is opened and used. If the specified data set does not exist, it is created. If TEMPLATE= is specified, all of the variable definitions from the TEMPLATE data set are copied into the new DATA data set. If TEMPLATE= is not specified, the new data set has the default variables. Data set options should include options for creating the data set and for opening the data set for update.

The library specification must be a library that is specified in one of the Application Server ALLOCATE statements. This enables the Application Server to define the library for the server and for administrative requests.

Note: On z/OS, the library containing the statistics data set cannot use the DISP=NEW option because the library might be assigned multiple times. If you wish to create a new statistics data set when running PROC APPSRV, issue the LIBNAME statement with DISP=NEW before the PROC APPSRV statement. For example,

```
libname statds '...' disp=new;

proc appsrv port=5800;
allocate library statds '...';
statistics data=statds.stats;
run;
```

ADDPORT

tells the Application Server to append the port number of the current server to the member name of the

STATISTICS data set. This is useful when you want pool servers to write to different data sets.

EXITONERROR

causes the Application Server to exit when there is a failure writing to the statistics data set. Failures can occur due to a disk−full condition, a SAS/SHARE server shutdown or other conditions that disable access to the data set. Use this option if you must collect statistics on all server accesses (for example, for security auditing purposes).

If EXITONERROR is not set, a write failure causes the Application Server to queue STATISTICS observations and periodically attempt to write them to the data set. The Application Server will continue to process client requests. If the error condition is not corrected and the Application Server is stopped, any queued STATISTICS observations are lost.

TEMPLATE=library.dataset <(data−set−options)>

specifies a template data set. When the statistics data set is created, the variable definitions from the template data set are copied into the new statistics data set.

The specified library must be defined externally to PROC APPSRV.

WRITECOUNT=n

specifies the number of observations to place in the queue before writing to the data set. When the data set is temporarily unavailable (for example, when a SAS/SHARE server is restarting) the queue might grow larger than WRITECOUNT. The default value for WRITECOUNT is 50 observations.

WRITEEVERY=n

writes all observations that are in the queue every *n* minutes. The default value for WRITEEVERY is 5 minutes.

Default Contents of the Data Set

The following table shows the default variables:

Variable Name	Variable Type	Description
Obstype	Character length 1	R = request, I = Internal, U = startup, D = shutdown, T = trace
Okay	Character length 1	1 = request ran okay, 0 = error
Duplex	Character length 1	H = half duplex, F = full duplex
Http	Character length 1	1 = http request, 0= normal broker request
Program	Character length 32	_PROGRAM variable
Peeraddr	Character length 16	Peer address
Hostname	Character length 20	Node name of the server
Username	Character length 12	_USERNAME variable, if any
Entry	Character length 32	_ENTRY variable, if any
Sessionid	Character length 12	_SESSIONID, if any
Service	Character length 12	Service name
Starttime	Number	Time the request started
Runtime	Number	Run time of the request

Port	Number	Server port number
Bytesin	Number	Number of input bytes (read from client)
Bytesout	Number	Number of output bytes (written to client)
Cputime	Number	Amount of usage time for CPU for the request. This field is only available on z/OS systems. The STIMER option must be enabled to get valid Cputime values. The STIMER option is the default for z/OS.

Customizing the Statistics Data Set

In some cases, you might want to modify the list of variables in the data set. The example below shows how to

- eliminate the default SERVICE variable
- make the PROGRAM variable larger in size
- and add two new variables, EMPNO and EMPDEPT (input values used by applications that run in this service).

First, create a default data set using the following code:

```
proc appsrv port=0;
   statistics create=work.stdstat;
   run;
```

This command creates the data set called STDSTAT in the WORK library, and writes the default list of variables to it. Next, use a DATA step to create a modified data set, as follows:

```
libname statlib 'path-to-library';
data statlib.stats;

   /* change program length to 40 - you must change var defns before anything else */
   length program $40;

   /* start with the default data set */
   set work.stdstat;

   /* set up EMPNO and EMPDEPT variables */
   attrib empno   length=$8  label='Employee Number';
   attrib empdept length=$32 label='Employee Department';

   /* drop service */
   drop service;

   /* do not select any observations (there are none) from the previous data set */
   stop;
run;
```

The STATLIB.STATS data set now contains the desired variables. Modify your appstart.sas file to save statistics to this data set, as follows:

```
proc appsrv ...;
   ...
   allocate library statlib 'path-to-library';
   statistics data=statlib.stats;
```

```
run;
```

Application Server Access to the Data Set

The Application Server opens the statistics data set for WRITE access. This means that usually each server needs to write to its own data set. However, if a data set is accessed by using SAS/SHARE software, multiple servers can write to a single data set.

The following code is an example of a PROC APPSRV command that specifies that a single server accesses a single data set:

```
proc appsrv ... ;
   ...
   allocate library data '.';
   STATISTICS DATA=data.stats;
   ...
   run;
```

The following is an example of a PROC APPSRV command that specifies that a server write statistics to a data set on a SAS/SHARE server:

```
proc appsrv ... ;
   ...
   allocate library data '.' server=host.sasapp11;
   STATISTICS DATA=data.stats;
   ...
   run;
```

Refer to SAS/SHARE documentation for more information about configuring, running, and accessing SAS/SHARE servers.

Application Access to the Data Set

User applications can access the data set by using the _STATDATASET and _STATDATALIBNAME macro variables. The _STATDATASET macro variable contains the library.DATASET setting of the statistics data set for this server. The _STATDATALIBNAME contains the LIBNAME, the physical name, and the options of the ALLOCATE FILE statement for the statistics data set library. This enables the application to assign a LIBNAME to the library with additional options (for example, ACCESS=READONLY).

Before you access the data set with the _STATDATASET or _STATDATALIBNAME macro variables, check the status of the _STATDATASETAVAIL macro variable. It is set to one of the following values:

Value	Description
OK	The statistics data set is enabled and available for use.
NOADMINPW	The statistics data set is enabled, but the _ADMINPW password was not supplied for this request or was incorrect. The _STATDATASET and _STATDATALIBNAME variables are not defined in this case.
NOSTATS	The statistics data set is not enabled.

The following code is an example of assigning a libname and data access authority to the statistics data set library:

```
&_STATDATALIBNAME access=readonly;
data RSTATS;
   set &_STATDATASET;
   where obstype='R';
   keep program starttime runtime;
   run;
```

Note: Only requests with administrator privileges (if the PROC APPSRV ADMINPW option is not specified, all requests, otherwise only requests with a valid _ADMINPW) get these macro variables set. In this way, an administrator can control access to the data set.

Samples

The Application Dispatcher includes some sample applications to help you understand how to create your own applications. These samples are documented at ***http://support.sas.com/samples***. You can also use these applications to ensure that you installed and configured the Application Dispatcher correctly. The input component for each sample application is installed along with the Application Broker in a default location of http://*yourserver*/sasweb/IntrNet9/dispatch/. The program component is installed along with the Application Server in the sample and samplib program libraries.

The Xplore sample application uses Application Dispatcher and Web Publishing Tools to browse data sets, catalogs, catalog entries, perform drill−down on PROC SUMMARY data sets, and even download data sets as comma−separated value files directly into your spread sheet program. This sample application provides an explorer interface to SAS libraries and their contents. Xplore uses HTML frames, so it requires three HTML files as input. Point your browser at http://*your_server*/sasweb/IntrNet9/xplore/webxplor.html to start this application.