

SAS[®] 9.3 Publishing Framework Developer's Guide



The correct bibliographic citation for this manual is as follows: SAS Institute Inc 2011. *SAS® 9.3 Publishing Framework: Developer's Guide*. Cary, NC: SAS Institute Inc.

SAS® 9.3 Publishing Framework: Developer's Guide

Copyright © 2011, SAS Institute Inc., Cary, NC, USA.

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

For a hardcopy book: No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

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.

The scanning, uploading, and distribution of this book via the Internet or any other means without the permission of the publisher is illegal and punishable by law. Please purchase only authorized electronic editions and do not participate in or encourage electronic piracy of copyrighted materials. Your support of others' rights is appreciated.

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.

1st electronic book, July 2011

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/publishing 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.

Contents

<i>What's New in SAS 9.3 Publishing Framework</i>	<i>v</i>
Chapter 1 • Overview of the Publishing Framework	1
What Is the Publishing Framework?	1
Channel Definition and Subscription Management	2
Package Publishing	2
Package Retrieval and Viewing	2
Event Publishing	3
Chapter 2 • Overview of Packages	5
What Is a Package?	5
Rendering a Package	6
Package Transports	7
Chapter 3 • Retrieving Packages and URLs	11
Retrieving Packages from Different Transports	11
SAS Package Retriever	12
SAS Package Reader	13
Retrieving URLs	13
Chapter 4 • Viewer Processing	15
Overview of Viewers	15
When to Use a Viewer	16
How to Create a Viewer	17
Using the Publish Package Interface to Apply a Viewer	22
Dictionary	23
Examples.	31
Chapter 5 • Publishing Packages	39
Package Publishing	40
Using a Third-Party Client Application	41
Using the Publish Package Interface	41
Publish and Retrieve Encoding Behavior	43
Filtering Packages and Package Entries	44
Specifying Name/Value Pairs	48
Dictionary	50
Examples.	139
Chapter 6 • Generating and Publishing Events	149
What Is an Event?	149
Overview of Generating and Publishing Events	150
Using the Publish Event Interface	151
XML Specification for Generic Events	151
XML Specification for SASPackage Events	153
Dictionary	158
Examples.	168
Appendix 1 • Planning and Implementing Your Publishing Solution	173
Plan the Information Architecture	173

Establish the Content Pipeline	174
Configure Channels and Subscribers	175
Implement Content Restrictions in the SAS Metadata Authorization Layer	175
Announce Solution and Train Users	176
Index	177

What's New in SAS 9.3 Publishing Framework

Overview

SAS 9.3 Publishing Framework provides the ability to publish to Microsoft SharePoint, as well as new properties for CALL routines.

General Enhancements

The following general enhancements have been added to the SAS Publishing Framework:

- You can use the Publish Package CALL routines to publish and retrieve packages to and from SharePoint. Publishing to SharePoint is similar to publishing to WebDAV in that it facilitates concurrent access to and update of package data on the Internet. For more information about publishing packages to SharePoint, see [“PACKAGE_PUBLISH \(Publish Package to Microsoft SharePoint\)” on page 86](#). For more information about retrieving packages from SharePoint, see [“RETRIEVE_PACKAGE” on page 130](#).
- The INSERT_FILE CALL routine can have a DESTINATION_FILENAME property defined. This property enables you to specify a different destination filename when publishing a package. For more information, see [“INSERT_FILE” on page 54](#).
- The HTTP_TOKENAUTH property can be used when publishing to WebDAV. This property enables token authentication to a SAS Content Server. For more information, see [“PACKAGE_PUBLISH \(Publish Package to a WebDAV-Compliant Server\)” on page 103](#).
- The GENERATED_NAME property is supported when publishing to WebDAV, archive, or SharePoint. This property enables you to obtain the name of the generated package or archive. For more information, see [“PACKAGE_PUBLISH \(Publish Package to a WebDAV-Compliant Server\)” on page 103](#).

Chapter 1

Overview of the Publishing Framework

What Is the Publishing Framework?	1
Channel Definition and Subscription Management	2
Package Publishing	2
Package Retrieval and Viewing	2
Event Publishing	3

What Is the Publishing Framework?

The Publishing Framework feature of SAS Integration Technologies provides a complete and robust publishing environment for enterprise-wide information delivery. The Publishing Framework consists of SAS CALL routines, application programming interfaces (APIs), and graphical user interfaces that enable both users and applications to publish SAS files (including data sets, catalogs, and database views), other digital content, and system-generated events to a variety of destinations including the following:

- e-mail addresses
- message queues
- publication channels and subscribers
- WebDAV-compliant servers
- archive locations
- SharePoint servers

The Publishing Framework also provides tools that enable both users and applications to receive and process published information. For example, users can receive packages with content, such as charts and graphs, that is ready for viewing. SAS programs can receive packages with SAS data sets that might in turn trigger additional analyses on that data.

The functions of the Publishing Framework include channel definition, subscription management, package publishing, package retrieval, package viewing, and event publishing.

Channel Definition and Subscription Management

The Publishing Framework enables you to define SAS publication channels, which are conduits for publishing particular categories of information. You can set up a channel for a particular topic, organizational group, user audience, or any other category. Once publication channels have been defined, authorized users can subscribe to them and automatically receive information whenever it is published to those channels.

For more information about defining channels and managing subscriptions, refer to the help for the Publishing Framework plug-in for SAS Management Console.

The SAS Information Delivery Portal also enables users to manage their subscriptions. The portal enables users to select channels to subscribe to, specify the desired delivery transport (such as an e-mail address or message queue), and specify filters that indicate which information is to be included or excluded.

Package Publishing

The Publishing Framework enables you to create packages that contain one or more information entities, including SAS data sets, SAS catalogs, SAS databases, and almost any other type of digital content. You can also define viewers that make the information entities easier to display.

After creating a package, you can publish the package and its associated viewers to one or more channels. This causes the information to be delivered to each user who has subscribed to those channels, if the package and its contents meet the subscriber's filtering criteria. In addition to channels, you can publish packages directly to one or more e-mail addresses, message queues, WebDAV-compliant servers, archive locations, and SharePoint servers.

To create and publish packages, you can use either of the following methods:

- Use the publish CALL routines to create packages and publish them from within a SAS program.
- Use the Java APIs that are provided with SAS Integration Technologies to create packages and publish them from within a third-party application.

You can also use SAS Enterprise Guide or SAS Information Delivery Portal to create and publish packages via the Publishing Framework.

Package Retrieval and Viewing

The Publishing Framework provides the SAS Package Retriever, which is a graphical user interface to enable users to extract and save information from packages that have been published through the Publishing Framework. The SAS Package Reader user interface enables users to display the contents of packages. Users can also view published information from the SAS Information Delivery Portal.

In addition, you can use CALL routines to extract and process published information from within SAS programs. APIs are provided to enable third-party programs to extract and process published information.

Event Publishing

SAS Integration Technologies includes the Publish Event Interface. This interface consists of CALL routines that enable you to write SAS programs, including stored processes, that create and publish events. Events can be generated and published explicitly, or they can be generated implicitly as part of the publication of a package. Implicit event generation occurs when packages are published to a channel that has event subscribers defined.

Events are published as XML documents. They can be published to HTTP servers, message queues, or to publication channels and their associated subscribers. You can develop custom applications that receive and process events that are generated by the Publishing Framework.

Chapter 2

Overview of Packages

What Is a Package?	5
Definition of Package	5
SAS Results	5
Unstructured Content	6
Filename Extensions for Package Entry Types	6
Rendering a Package	6
Package Transports	7
Overview of Package Transports	7
Persisted Packages	8
Subscription Channels	8

What Is a Package?

Definition of Package

A package is a container for information, or digital content, that is generated or collected for delivery to a consumer.

Knowledge takes the form of package entries, which can be either of two types:

- SAS results
- unstructured content

SAS Results

A category of package entry type is SAS results, which can take the form of any of the following:

- SAS catalog
- SAS data set
- SAS database (such as MDDb)
- SAS SQL view

Unstructured Content

Unstructured content is any package entry that is created external to SAS. Supported unstructured content types include the following:

- binary file (such as Excel, GIF, JPG, PDF, PowerPoint, and Word)
- HTML file (including ODS output)
- reference string (such as URL)
- text file (such as a SAS program)
- viewer file (such as an HTML or plain text template that formats SAS file items for viewing in e-mail)

Filename Extensions for Package Entry Types

Each entry in a package is implicitly contained in a file whose filename extension reflects the entry type. Knowing filename extensions might be useful when retrieving packages from the archive, WebDAV, and SharePoint transports.

Default filename extensions are as follows:

```
.csv - Comma-separated values
.ref - Reference
.sac - SAS catalog (for archive and WebDAV transports)
.sad - SAS data set (for archive and WebDAV transports)
.sam - SAS MDDB
.sav - SAS SQL View
.spk - Archive package
.stc - SAS catalog or SAS data set (for SharePoint transport)
```

Rendering a Package

When determining how to render packages, the publisher should consider the following:

- the company's business requirements
- the configuration of the business enterprise (for example, hardware, software, business processes, and communications protocols)
- the package content (structured and unstructured data)
- the transport (such as archive, channel, e-mail, message queue, or Web) that is used to deliver the package

The following scenarios depict business factors that might affect how a package is rendered:

Table 2.1 *Package Rendering to Meet Consumer Needs*

Consumer Need	Publisher Solution
Access to packages, but have limited system storage resources	Render the package as an archive.

Consumer Need	Publisher Solution
Access to package without having SAS software installed	Render the package as an archive and attach the archive to e-mail for access by using SAS Package Reader.
Only executive-level summaries (for example, text reports, graphics, and Web links)	Render the package as unstructured content to known consumers via e-mail or to unknown consumers via subscription-based channels.
Access to SAS results, but do not want to access the package for continued processing	Apply a template to the SAS data package entry for viewing in e-mail and on the Web.
Access to SAS results, but do not have Web access or do not use HTML	Apply a template in plain-text format to the SAS results for viewing in e-mail.
Direct access to SAS results for continued data processing	Deliver SAS results package entries to message queues or archives to enable programmatic access to SAS data.
To span a broad professional range (executive, manager, programmer, and knowledge worker)	Apply name/value metadata to the package and package entries to enable consumers to filter packages or package entries for relevancy.

Before the publisher can begin the publishing process, the administrator must first configure the publishing environment, which might include archives, channels, subscribers, and subscriptions.

Package Transports

Overview of Package Transports

The destination (or transport) for delivering a package is defined programmatically in a SAS application by using `PACKAGE_PUBLISH CALL` routines.

Transports are as follows:

archive

a single binary collection of all the items in a package, which is compressed and saved to a directory file. The archive contains the contents of a package and metadata that is necessary for extracting the contents. An archived package is also referred to as an SPK file, which is short for SAS Package.

channel

a conduit through which the defined transport (either e-mail or message queue) delivers package items to the subscribers of the channel. The subscriber defines the preferred transport by using personal subscription properties. Whereas publishing to e-mail is identity-centered (the publisher delivers packages to recipients whose identities are known), publishing to channels is subject-centered, allowing both the publisher and the consumer to concentrate on the subject of the package rather than on the recipients for the package.

e-mail

mechanism for delivering selected package items to identified recipients.

message queue

in application messaging, a place where the publisher can send a message (or a package) that can be retrieved by another program for continued processing.

SharePoint

Microsoft product that enables users to share files, collaborate, and publish to the Web. Publishing to SharePoint is similar to publishing to WebDAV in that it facilitates concurrent access to and update of package data on the Internet.

WebDAV-compliant server

an acronym for Web Distributed Authoring and Versioning. Whereas the traditional transports (archive, channel, e-mail, and message queue) are repositories for published package data that can be retrieved and reprocessed in a synchronous fashion, a WebDAV-compliant server facilitates concurrent access to and update of package data on the Internet.

WebDAV is not only a delivery mechanism, but also a core technology that extends the HTTP network protocol, enabling distributed Web authoring tools to be broadly interoperable. WebDAV extends the capability of the Web from that of a primarily read-only service, to a writable, collaborative medium.

Persisted Packages

Publishing a package to an archive or to a WebDAV server provides the following advantages:

- You conserve disk resources.
- The package stays in a static location, allowing consumers or programs to retrieve the package at their convenience.
- The SAS data package entries are available to consumers who do not have SAS software installed on their systems.

After the administrator has configured channels with archive paths or base paths, the publisher can publish packages directly to an archive or WebDAV server. The publisher can use the following methods to publish to a persistent store (an archive or a WebDAV server):

- publishing [programmatically by using SAS](#)
- using a [third-party client application](#)

The consumer can use the stand-alone product SAS Package Reader subsequently to uncompress, or unzip, and use an archived package. SAS Package Retriever can be used to access the package from the persisted location and to store the package elsewhere. The RETRIEVE SCL application, CALL routines, and SAS Information Delivery Portal can also be used to retrieve persisted packages.

Subscription Channels

Creating Channels

The administrator uses SAS Management Console to create a channel object with the attributes that are specified in the SAS Metadata Repository. The administrator must create a channel for each distinct topic or audience. For example, users of a particular application might want a channel for discussion and data exchange, while the

programmers of that application might want another channel to discuss technical problems and future enhancements. Although the topic is the same application, the discussion about the topic is different. Therefore, two separate channels might best serve the needs of the two groups of users.

Creating Subscribers

The administrator must create a subscriber for each potential user of a channel. Subscribers must be defined before subscriptions to channels can be created.

Creating Subscriptions

The association of a subscriber to a channel is a subscription. A subscription enables the information that is published to a channel to be delivered to the interested (subscribed) subscribers.

Subscriptions can be created by either the administrator or the subscriber. The administrator can create subscriptions when a publishing environment is initially configured. Individual subscribers can create personal subscriptions after the publishing environment has been configured.

Chapter 3

Retrieving Packages and URLs

Retrieving Packages from Different Transports	11
SAS Package Retriever	12
SAS Package Reader	13
Retrieving URLs	13

Retrieving Packages from Different Transports

After a package is created, the transport, or destination, and other properties control how the package is delivered to the consumer.

Packages can be retrieved from the following destinations:

- archive
- e-mail
- message queue
- WebDAV-compliant server
- SharePoint

For archives, you can use the stand-alone product [SAS Package Reader](#) to uncompress, or unzip, and use SPK files. SAS Package Reader can be used to read packages whether or not the consumer has SAS installed.

Depending on your needs and on whether you have SAS installed, you can choose from the following products to access a package on an archive, message queue, or WebDAV-compliant server:

- The consumer can use [SAS Package Retriever](#) to access a package and to store it elsewhere for continued use. SAS must be installed in order to use the SAS Package Retriever.
- If SAS is installed, then you can use the [Publish Package CALL routines](#) in order to write SAS programs, including stored processes, that create, populate, publish, and retrieve packages. You can also use the Publish Package CALL routines to publish and retrieve packages to and from SharePoint.
- If SAS is not installed, then you can use [third-party client software](#) in order to write a third-party client application that uses SAS Integration Technologies to access Integrated Object Model (IOM) servers. The Integrated Object Model provides distributed object interfaces for conventional SAS features. This enables you to

develop component-based applications that integrate SAS features into the enterprise application.

- You can also use SAS Information Delivery Portal to retrieve packages from archives or WebDAV-compliant servers. For more information, see the product Help.

For more information about configuring and publishing to an archive, see “[Persisted Packages](#)” on page 8.

When the publisher publishes a package via e-mail, the package is delivered to a list of recipients. Choosing e-mail gives the publisher authority over who receives the package. The recipient, however, requires no knowledge about the publishing environment from which the package was sent, nor must the recipient subscribe to a delivery channel. Also, recipients do not have to be SAS users. If the e-mail has a package file attachment, or if the e-mail contains a link to the persisted WebDAV package, then SAS is not needed to consume the package. The recipient can use SAS Package Reader or a Web browser in order to read the package.

Note: There are two methods for using e-mail to publish packages. The method described in this section refers to publishing packages explicitly to e-mail addresses. The other method is to publish a package to a channel, which can have e-mail subscribers. In that case, the recipient must be subscribed to a delivery channel.

Although e-mail is suited for delivering reports and views of data to a limited audience, a message queue is best used for collecting package data entries for continued processing and publishing in time-critical environments. Publishing to a queue, and retrieval from a queue, are entirely independent activities. The publishing software (programmatic software) and the retrieval software (SAS Package Retriever or programmatic software) communicate asynchronously without any knowledge of the location of the other software, or even whether the other software is running.

Whereas the traditional transports (archive, channel, e-mail, and message queue) are repositories for published package data that can be retrieved and reprocessed in a synchronous fashion, package delivery to SharePoint or to a WebDAV-compliant server facilitates concurrent access to and update of package data on the Internet.

SAS Package Retriever

SAS Package Retriever is an SCL application that enables you to retrieve package data from a transport—archive, message queue, or WebDAV-compliant server—to an appropriate storage location on your SAS system or an external file location. After you designate a storage location for the entry (for example, at a libref, fileref, or a file location), you can reference the entry for inclusion in a SAS program for continued package publishing in the business enterprise.

Examples of package data include SAS data (such as a SAS data set, SAS catalog, or a SAS database) and external data (such as an HTML file, binary file, text file, or viewer file).

Underlying the functions of SAS Package Retriever is a subset of the [Publish Package CALL routines](#) that relate explicitly to retrieving packages, which you can use directly for programmatic package retrieval.

To invoke SAS Package Retriever, enter the following command on the SAS command line:

```
retrievepackage
```

For more information about SAS Package Retriever, see the product Help.

SAS Package Reader

The SAS Package Reader application enables you to retrieve the contents of a SAS package as an archive file from an archival location or from an e-mail attachment without having to run SAS. An archive is denoted by a .spk file extension, which is an abbreviation for SAS Package.

A read-only tool, SAS Package Reader is useful for viewing individual package entries and saving them to local files. SAS Package Reader launches an appropriate viewer to enable you to see the content of the package entry. For SAS data sets, it starts a built-in data set viewer; for all other viewable data, it starts the Web browser that is already configured on your system.

Note: Some entry types cannot be viewed. Examples include viewer files, SAS catalogs, and SAS databases (MDDDB and DMDB files). If the selected entry type is not viewable, then the View icon does not appear in the toolbar. In addition, if you try to view a SAS data set that is password-protected, a message is displayed saying that the data set cannot be accessed.

What you do with a package corresponds to the type of consumer that you are and the type of information that is contained in the package. Packages are created for specific target consumers for definite purposes.

Because you do not need SAS running in order to use SAS Package Reader, you do not need additional SAS software licensed in order to retrieve packages.

For more information about SAS Package Reader, see the product Help.

Retrieving URLs

When a package is published to a channel that is defined with a WebDAV persistent store, then e-mail subscribers receive e-mail with a link to the persisted location. Here is an example of an e-mail message in which the URL reference is identified:

Weekly sales

Published on 24MAR2000:20:14:19 GMT

The package contains graphs, ticket sales data,
and an executive summary.

URL:

<http://www.AlphaliteAirways/weeklysales/031200>

Clicking the link automatically invokes the consumer's configured Web browser and the URL package entry is presented for viewing in the Web browser window.

Chapter 4

Viewer Processing

Overview of Viewers	15
When to Use a Viewer	16
How to Create a Viewer	17
The Sample Package	17
SAS Data Set	17
Basic Viewer	17
Using the Publish Package Interface to Apply a Viewer	22
Dictionary	23
<SASINSERT> Tag	23
Substitution Syntax	24
<SASTABLE> Tag	28
<SASREPEAT> Tag	29
<SASECHO> Tag	31
Examples	31
Example 1: Sample: Using the <SASINSERT> and <SASTABLE> Tags to Build a List	31
Example 2: Sample: Using the <SASINSERT> and <SASTABLE> Tags to Build a Table	32
Example 3: Sample HTML Viewer	32
Example 4: Sample SAS Program with an HTML Viewer	33
Example 5: Sample Viewer Template	35

Overview of Viewers

SAS Integration Technologies provides a viewer facility that combines the robust text rendering capabilities of HTML and plain text with the efficiency of e-mail delivery. This facility enables you to create and apply a *viewer*, which is a template that contains formatting directives for rendering a specific view to an entire package or to selected package entries.

The viewer facility consists of a set of tagging extensions to HTML, which you can use to create a unique template according to the specific package data that is being rendered. For example, you can write formatting directives to stream package entries (such as a text file or a URL reference) or to extract SAS data file entries for presentation in e-mail. A viewer creates a presentation quality look and feel to package data entries for distribution to a view-only audience.

A primary benefit of applying viewers to packages is that e-mail recipients can now view package entries that otherwise would not be viewable. For example, an archive that contains a SAS data set can be attached to e-mail, but is not viewable in e-mail unless a viewer is applied. The viewer renders the SAS data set as a populated table.

Furthermore, a viewer facilitates publishing in the traditional sense using, for example, an electronic newsletter. An electronic newsletter template that is coded in HTML or plain text format can dynamically build your content, which can consist of links to Web sites for up-to-date information about topics of interest to its readers.

When to Use a Viewer

A viewer is useful under the following conditions:

Publishing to the E-mail Transport

You want to publish a package that contains a data set, for delivery to consumers who use a view-only transport (such as e-mail). Because a SAS data set is not viewable in e-mail, an HTML or text viewer is needed to format the SAS data for a view-only presentation.

Publishing to Channel Subscribers

If you are publishing to a channel, the transports that are used by subscribers are unknown to you. Therefore, you might decide to format the entire package with the aid of a viewer to ensure maximum viewability for the broadest consumer audience. The viewer is applied to a package that is published to subscribers who use e-mail delivery, WebDAV subscribers, or channels that have a WebDAV persistent store. The viewer is not applied to a package that is published to subscribers who specify delivery to message queues.

Extracting and Formatting SAS Data

With a viewer, you can extract specific package items and variables from a SAS data set entry and distribute to subscribers who use e-mail. Subscribers who use e-mail receive the package entries that the viewer extracts and formats. Subscribers who use queues receive the full package.

Formatting an Entire Package

Besides formatting a SAS data set package entry, you can also use a viewer to format other entries in the package (such as another HTML file, a text file, a binary file, or a reference) as input streams. Applying a viewer to the entire package provides a comprehensive presentation for viewing purposes only.

Publishing an Electronic Newsletter

A popular form of package output is an electronic newsletter. The basic template that imposes the look and feel of the document can contain static text or HTML coding. However, you can code the dynamic information (in the form of news articles or SAS data) as links to Web sites whose source data is continuously refreshed.

Publishing an Executive Level Summary

Delivery of SAS result sets and other text and graphical information via e-mail has the greatest value for an executive level consumer. The executive might have a requirement to view the data (for example, in the form of summary tables) and to read text but might not necessarily need access to the raw data for continued processing, extraction, and delivery throughout the enterprise.

How to Create a Viewer

The Sample Package

The publisher (or the person who creates the viewer template) must have a thorough understanding of the contents of the package in order to successfully create the template. You choose the appropriate viewer tags in order to design a template that formats the rendered view of the package.

This sample package contains four entries, in the following order:

1. SAS data set
2. text file
3. reference URL
4. HTML file

This package also contains the following description:

AlphaliteAirways Sales Force Briefing

SAS Data Set

Here is an example of a SAS data set that contains six observations, each containing four variables: FNAME, LNAME, YEARS, and TERRITORY.

John	Smith	32	NE
Gary	DeStephano	20	SE
Arthur	Allen	40	MW
Jean	Forest	3	NW
Tony	Votta	30	SW
Dakota	Smith	3	HA

Basic Viewer

Overview of Basic Viewer

The file that contains the viewer template contains code that is surrounded by tags < SASINSERT> open tag and end with the </SASINSERT> closing tag.

The viewer contains sections that format each package entry by using the appropriate technique.

- [extracting and formatting a variable from a SAS data set into a list](#)
- [extracting and formatting a SAS data set into a table](#)
- [streaming a text file and a reference URL](#)
- [filtering package entries](#)

For a single, comprehensive viewer that contains all of the preceding examples, see [“Example 3: Sample HTML Viewer” on page 32](#).

Extracting and Formatting a Variable from a SAS Data Set into a List

Delivery of a single variable from all observations in a SAS data set is suitable for an unordered list.

Here is the first section from the sample template that formats a single variable from a SAS data set into a list.

```
<!--Section 1: Formatting a Data Set
      Variable in an HTML List-->
<SASINSERT>
<h2>Congratulations!</h2>
$(entry=1 attribute=description)
<ul>
<SASTABLE ENTRY=1>
<li>$(VARIABLE=fname)</li>
</SASTABLE>
</ul>
</SASINSERT>
```

The ENTRY=1 attribute identifies the SAS data set as the first entry in the package. The description attribute extracts the description of the package.

The HTML tag specifies an unordered list after which the <SASTABLE> tag with the ENTRY=1 option are necessary to identify the SAS data set as the first entry in the package. The HTML tag is used with variable substitution syntax to identify that the variable fname is to be extracted from the SAS data file and formatted as a list entry in the rendered view. Implicit in the <SASTABLE> construct is looping. Each observation in the data set is read and formatted until the end of file is reached.

The following SAS HTML tags are used in this example:

- [<SASINSERT> tag](#)
- [Substitution syntax](#)
- [<SASTABLE> tag](#)

This section of the template is rendered for viewing in e-mail as follows:

```
Congratulations!

AlphaliteAirways Sales Force Briefing
  John
  Gary
  Arthur
  Jean
  Tony
  Dakota
```

Extracting and Formatting a SAS Data Set into a Table

Delivery of multiple variables or all variables from the observations in a SAS data set is suitable for a tabular presentation.

Here is an example of a template that extracts three of four variables from a SAS data set into a table.

```
<!--Section 2: Formatting a SAS Data
      Set into a Table-->
<SASINSERT>
<h2>Record Sales from these Salespeople</h2>
$(entry=1 attribute=description)
```



```

<table border cellspacing=0 cellpadding=5
      rules=groups>
<thead>
<tr>
<th>First Name</th>
<th>Last Name</th>
<th>Territory</th>
</tr>
</thead>
<tbody>
<SASTABLE ENTRY=1>
<tr>
<td> $(Variable=fname)</td>
<td> $(Variable=lname)</td>
<td> $(Variable=territory)</td>
</tr>
</tbody>
</SASTABLE>
</table>
</SASINSERT>

```

The ENTRY=1 attribute identifies the SAS data set as the first entry in the package. The description attribute extracts the description of the entry from the package. Standard HTML table tags set up the tabular framework that defines a row with three columns of header text and accompanying tabular data. The <TD> tag is used with the variable substitution syntax to identify the following variables for extraction and insertion into the table: fname, lname, and territory. Implicit in the <SASTABLE> construct is looping. Each observation in the data set is read and formatted until the end of file is reached.

The following SAS HTML tags are used in this example:

- [<SASINSERT> tag](#)
- [Substitution syntax](#)
- [<SASTABLE> tag](#)

This section of the template is rendered for viewing in e-mail as follows:

Record Sales from These Salespeople

Table 4.1 *AlphaliteAirways Sales Force Briefing*

First Name	Last Name	Territory
John	Smith	NE
Gary	DeStephano	SE
Arthur	Allen	MW
Jean	Forest	NW
Tony	Votta	SW
Dakota	Smith	HA

Streaming a Text File and a Reference URL

The viewer template might also include the entire contents of a text file, another HTML file, a reference URL, or a binary file.

Here is the third section from the sample template that inserts a text file and a reference URL into the viewer.

```
<!--Section 3: Inserting a Text File
      and a Reference URL-->
<SASINSERT>
<h2>Letter of Congratulations</h2>
<p>Below is a copy of the letter that was sent
to each recipient of the top sales award.</p>
$(entry=2 attribute=stream)
<p>
See <a href="$(entry=3 attribute=stream)">
for detailed sales figures.</p>
</SASINSERT>
```

The <H2> tag defines a descriptive heading for the text document and the reference URL. The ENTRY=2 attribute identifies the entry (a text document) to be substituted as an input stream to the HTML output. The ENTRY=3 attribute identifies the reference URL.

The following SAS HTML tags are used in this example:

- [<SASINSERT> tag](#)
- [Substitution syntax](#)

This section of the template is rendered for viewing in e-mail as follows:

Letter of Congratulations

Below is a copy of the letter that was sent to each recipient of the top sales award.

December 30, 2000

International Sales
AlphaliteAirways Headquarters

Dear AlphaliteAirways Salesperson,

Congratulations on your much deserved recognition as outstanding salesperson for AlphaliteAirways for 2000.

To express our gratitude for your excellent contribution, we wish to present you with 25 stock options in AlphaliteAirways.

Wishing you continued success in your career with AlphaliteAirways.

Sincerely,

Alvin O. Taft, Jr.
Director-in-Chief

See <http://www.AlphaliteAirways.com/headquarters/sales>
for detailed sales figures.

Filtering Package Entries

Another method for locating package entries for inclusion in the viewer is name/value filtering. You can filter package entries that are assigned an optional name/value pair when they are created according to specified criteria. Entries that match are included in the rendered view. Filtering is especially powerful for searching large, nested packages.

In our example, we filter for all entries that have a name/value pair of type=report and include the matching entries in the viewer. In our fictitious package, one HTML entry matches the name/value pair and so it is filtered for inclusion in the viewer.

Here is the fourth section from the sample template that inserts an HTML file (according to matched criterion) into the viewer.

```
<!--Section 4: Filtering an Entry-->
<SASINSERT>
<h2>Message from the President</h2>
<SASREPEAT>
$(entry="(type=report)" attribute=stream)
</SASREPEAT>
</SASINSERT>
```

The ENTRY="(type=report)" attribute filters all package entries that contain a name/value pair of type=report. The <SASREPEAT> open tag and the </SASREPEAT> closing tag surround the search string in order to perform a repetitive search for the name/value pair. Without this tag, the search would end after the first match. In this example, only one HTML entry is matched. This entry is substituted as an input stream to the HTML output.

The following SAS HTML tags are used in this example:

- [<SASINSERT> tag](#)
- [Substitution syntax](#)
- [<SASREPEAT> tag](#)

This section of the template is rendered for viewing in e-mail as follows:

Message from the President

AlphaliteAirways delivers service. AlphaliteAirways is the recognized industry leader according to its safety record, volume of passengers served, and number of routes serviced.

How are we able to live up to such high expectations consistently? First and foremost, we do it through the abilities of our top salespeople. We owe a huge debt to these hard-working individuals who actively pursue revenue for this company.

Using the Publish Package Interface to Apply a Viewer

After you create a viewer template for a package, the publisher can apply it when publishing the package to e-mail by using the Publish Package Interface. For the e-mail, channel subscriber, and WebDAV delivery types only, you specify a viewer as a property to the `PACKAGE_PUBLISH` SAS CALL routine.

You specify the `VIEWER_NAME` property and assign to it a viewer in the form of either an external filename or a SAS fileref.

For example, the following code shows the application of an HTML viewer to a package that is published to e-mail:

```
publishType = "TO_EMAIL";
properties = "VIEWER_NAME";
viewerFile = "filename:c:\dept\saletemp.html";
emailAddress = "JohnDoe@alphalite.com";
CALL PACKAGE_PUBLISH(packageId, publishType, rc,
    properties, viewerFile, emailAddress);
```

The following code shows the application of a text viewer to a package that is published to e-mail:

```
publishType = "TO_EMAIL";
properties = "TEXT_VIEWER_NAME";
viewerFile = "filename:c:\dept\saletemp.txt";
emailAddress = "JohnDoe@alphalite.com";
CALL PACKAGE_PUBLISH(packageId, publishType, rc,
    properties, viewerFile, emailAddress);
```

The following code publishes the package (to which an HTML viewer is applied) to all subscribers of the HR channel. The subject property is specified so that all e-mail subscribers will receive the message with the specified subject.

```
publishType = "TO_SUBSCRIBERS";
storeInfo =
    "SAS-OMA://alpair02.sys.com:8561";
viewerFile = "filename:c:\dept\saletemp.html";
channel = 'HR';
subject = "Weekly HR Updates:";
user = "myUserName";
password = "myPassword";
props = "VIEWER_NAME, SUBJECT, CHANNEL_STORE, METAUSER,
    METAPASS";
CALL PACKAGE_PUBLISH(packageId, "TO_SUBSCRIBERS", rc,
    props, viewerFile, subject, storeInfo, user, password,
    channel);
```

For complete details about how to programmatically specify a viewer when you publish to the e-mail and the channel subscriber types, see [“PACKAGE_PUBLISH \(Publish Package to E-mail\)” on page 73](#) and [“PACKAGE_PUBLISH \(Publish Package to Subscribers\)” on page 94](#).

The following code publishes the package (to which an HTML viewer is applied) to a WebDAV-compliant server:

```
rc = 0;
publishType = "TO_WEBDAV";
properties= "VIEWER_NAME, COLLECTION_URL";
viewerFile = "filename:c:\dept\saletemp.html";
cUrl = "http://www.alpair.web/NightlyMaintReport";
CALL PACKAGE_PUBLISH(packageId, publishType,
    rc, properties, viewerFile, cUrl);
```

The following code publishes the package (to which an HTML viewer is applied) to SharePoint:

```
rc = 0;
publishType = 'TO_SHAREPOINT';
properties= 'COLLECTION_URL, HTTP_USER, HTTP_PASSWORD, TEXT_VIEWER_NAME,
    TARGET_VIEWER_NAME, TARGET_VIEWER_MIMETYPE, APPLIED_TEXT_VIEWER_NAME';
collectionUrl = http://www.alpair.web/NightlyMaintReport;
username = 'user_name';
password = 'password';
viewerFileTemplate = 'FILENAME:C:\dept\saletemp.html';
viewerFileName = 'packageIndex.html';
viewerMimetype = 'application/html';
viewerAppliedOutput = 'FILENAME:C:\dept\saletemp.output.html';
CALL PACKAGE_PUBLISH(packageId, publishType, rc, properties,
    collectionUrl, username, password, viewerFileTemplate,
    viewerFileName, viewerMimetype, viewerAppliedOutput );
```

Dictionary

<SASINSERT> Tag

Marks a section of the viewer file for viewer processing

Syntax

```
<SASINSERT> </SASINSERT>
```

Details

All viewer processing occurs within the opening <SASINSERT> tag and the closing </SASINSERT> tag. SAS tags and substitution statements are recognized only when they appear within the <SASINSERT> and </SASINSERT> tags.

The data that is inserted into the rendered view comes from a specified package entry. The data that is extracted from the entry can be any of the following:

- value of a SAS variable
- description of the entry or package
- entire entry, which is to be streamed into the HTML file
- reference to the entry
- package or nested package abstract

See “[Example 1: Sample: Using the <SASINSERT> and <SASTABLE> Tags to Build a List](#)” on page 31 and “[Example 2: Sample: Using the <SASINSERT> and <SASTABLE> Tags to Build a Table](#)” on page 32..

Substitution Syntax

A substitution statement within the <SASINSERT> opening tag and the </SASINSERT> closing tag inserts data from the specified package entry into a viewer file for delivery as HTML or text output

Syntax

```
<SASINSERT> $(nested=z entry=x attribute=value)</SASINSERT>
```

Required Arguments

\$()

indicates the start of substitution mode using the dollar sign (\$) followed by the open parenthesis. The close parenthesis indicates the end of substitution mode.

nested=z

identifies the nested package within the main package that is to be involved in the substitution. If the NESTED attribute is not specified, then only the entries in the main package are involved in the substitution. For information about the syntax of the z value, see “[Specifying Values for the NESTED and ENTRY Attributes](#)” on page 25.

entry=x

identifies the entry within the specified package that is to be targeted for the substitution. For information about the syntax of the x value, see “[Specifying Values for the NESTED and ENTRY Attributes](#)” on page 25.

name=someName

identifies a name of a name/value pair. The value of this name/value pair will be substituted. The name= and attribute= keywords cannot be specified on the same substitution string.

If entry= is specified along with name=, the entry's name/value specification will be used to make the substitution. For example, the following substitution takes the first entry in the package and determines the value of the name "title". This value is inserted into the HTML or text output:

```
$(entry=1 name="title")
```

This example evaluates the name/value that is specified at the main package level. The value for the name "title" is substituted:

```
($name="title")
```

attribute=value

identifies the attributes of the specified entry that are to be inserted into the HTML or text output. The value that is associated with this attribute can be any of the following:

description

inserts the description of the specified entry. For example, the following substitution inserts the description of the specified entry into the HTML or text output:

```
$(entry=1 attribute=description)
```

stream

streams the specified entry into the HTML or text output. The streamed entry must be one of the following entry types:

- reference string (added to the package with the INSERT_REFERENCE CALL routine)
- text file (added with INSERT_FILE routine)
- binary file (added with INSERT_FILE routine)
- HTML file (added with INSERT_HTML routine).

For example:

```
$(entry=1 attribute=stream)
```

reference

inserts a reference by substituting the entry's filename into the rendered view. For example, the following substitution inserts the filename of the first entry:

```
$(entry=1 attribute=reference)
```

abstract

Insert the package abstract at this location. If the NESTED attribute is not specified, the abstract of the main package is inserted in the HTML or text output. If the NESTED attribute is specified, then the abstract of the nested package is inserted in the HTML or text output. The ENTRY attribute is not valid when the abstract attribute is specified. For example, the following substitution inserts the main package abstract into the HTML or text output:

```
$(attribute=abstract)
```

Details

Specifying Values for the NESTED and ENTRY Attributes

The NESTED and ENTRY attributes are used in substitution syntax within the `<SASINSERT>` tag and as attributes on the `<SASTABLE>` tag. The examples that appear in this section apply to substitution syntax within the `<SASINSERT>` tag, but all of the syntax rules also apply to the use of the NESTED and ENTRY attributes in the `<SASTABLE>` tag.

You can specify the values of the NESTED and ENTRY attributes in two forms, numeric or name/value.

Note: Variable substitution is another type of substitution. It must be specified within the `<SASTABLE>` tag.

Identifying an Entry by Its Order in the Package

You use the entry's numerical order in the package to identify which entry is to be involved in a substitution operation.

An example of package entry order follows:

1. SAS data set
2. binary file
3. reference string
4. HTML file

The SAS data set is the first entry, the binary file is the second entry, and so on.

For the NESTED attribute, a numeric value identifies the package that is involved in the substitution based on order of nesting into the package. For example, nested=3 specifies the third package that is nested in the main package. To accommodate packages with multiple levels of nesting, a period (.) differentiates levels of nesting. For example, nested=2.5 specifies the fifth package that is nested in the second package that is nested in the main package.

For the ENTRY attribute, a numeric value identifies the entry that is to be used in the substitution that is based on the order of insertion into the package. For example, \$ (entry=2) specifies the second entry in the package.

If the NESTED attribute is not specified, then the specified entry in the main package is used for the substitution.

Identifying an Entry by Filtering the Package

Name/value pairs are used in the NESTED and ENTRY attributes to specify filters that determine which nested packages and entries are to be involved in a substitution operation. You must quote the name/value pair and contain it within parentheses. For example,

```
$ (nested=" (type=report) " entry=" (a=b) ")
```

When the name/value pair is specified outside the <SASREPEAT> tags, only the first entry that matches the filter is substituted. When the name/value pair is used inside the <SASREPEAT> tags, all entries that match the filter are substituted into the HTML or text output.

To limit the search for an entry to the main package only, omit the NESTED attribute. For example, \$ (entry=" (type=report) ") specifies that the entry that is to be involved in the substitution operation is the first entry in the main package that has a name/value pair of type=report.

Entries in the main package are always candidates for name/value substitution, even when the NESTED attribute is specified. In the following example, the entry that is involved in the substitution is either the first entry in the main package that matches the a=b name/value pair or it is the first entry that matches a=b in the first nested package with the type=report name/value pair.

```
$ (nested=" (type=report) " entry=" (a=b) ")
```

To substitute all entries that match the name/value pairs, enclose the substitution within the <SASREPEAT> tag. If the preceding example were enclosed in <SASREPEAT> tags, the entries that are involved in the substitution would be all those in the main package and the nested packages that match the a=b name/value pair.

The name/value syntax also supports the asterisk (*) wildcard on the NESTED attribute. The asterisk indicates "all levels below." For example, to substitute "all entries in all nested packages beneath this level," use a period (.) and an asterisk (*) in the NESTED attribute, as follows:

```
$ (nested=" (type=report) . *" entry=" (a=b) ")
```

The preceding example identifies for the substitution all entries that match the a=b name/value pair in the following packages:

- the main package
- the first nested package that contains a match of the type=report name/value pair, regardless of the nesting level of that package
- any package, regardless of name/value pair, that is nested beneath the first nested package

To substitute all matching entries in the main package and in all nested packages, use an asterisk in the NESTED attribute, as shown in the following example:

```
$(nested="*" entry="(a=b) ")
```

The preceding example substitutes all entries in the main package and in all nested packages at any level that match the name/value pair `a=b`.

Example: Using Substitution Syntax

```
$(entry=1 attribute=description)
```

indicates that the description for package entry 1 is to be substituted at this location.

```
$(nested=1 entry=4 attribute=stream)
```

indicates that the fourth entry within the first nested package should be streamed at this location. The entry must be either a reference, a text file, a binary file, or an HTML file.

```
$(nested=1.2 entry=2 attribute=stream)
```

identifies for streaming the second entry in the second package that is nested in the first package that is nested in the main package.

```
$(nested="*" entry="(type=report)" attribute=description)
```

indicates that the description of the first entry within the main package, or any nested packages, that matches the `type=report` name/value pair is to be substituted into the HTML or text output. If the substitution is contained within `<SASREPEAT>` tags, then all entries in the main and nested packages that match the `type=report` name/value pair are substituted into the HTML or text output.

```
$(nested="(type=report)" attribute=abstract)
```

indicates that the abstract from the nested package within the main package that matches the `type=report` name/value pair is to be substituted into the HTML or text output. If this substitution were specified within `<SASREPEAT>` tags, then the abstracts of all matching nested package entries in the main package are inserted into the HTML or text output.

```
$(name=title entry=1)
```

indicates that the first entry in the package is used for the substitution. Because `name=` is specified, a name/value substitution occurs. `Name=` identifies the name of a name/value pair. Therefore, in this case, it indicates a name of title. If the first entry's name/value specification contains a name of title, its value is substituted.

```
$(name=Definition entry="(type=report)")
```

indicates that the substitution occurs for the first entry within the main package that has the `type=report` name/value pair. The `name=` syntax indicates that a name/value substitution occurs. If the entry that matches the `type=report` filter has a name/value pair with the name of Definition, then its value is substituted. If this substitution is contained in a `<SASREPEAT>` tag, then the name/value substitution will occur for all entries in the main package that match the `type=report` filter.

```
$(name=title)
```

indicates that because `entry=` is not used in this substitution string, the name/value for the main package is used for the substitution. `Name=` identifies the name of a name/value pair, so in this case it indicates a name of title. If the package's name/value specification contains a name of title, then its value is substituted.

<SASTABLE> Tag

Populates HTML or text tables and lists

Syntax

```
<SASTABLE nested=z entry=x attribute=value> /* insert HTML tags or static text as needed here
    */ $(variable=variableName) /* insert HTML tags or static text as needed here */ </SASTABLE>
```

Required Arguments

nested=z

identifies an optional nested package within the main package that is to be used to build the table or list. If the NESTED attribute is not specified, then only the main package is used to build the table or list. The numerical and name/value syntax options that are available for the *z* value are defined in [“Specifying Values for the NESTED and ENTRY Attributes” on page 25](#).

entry=x

identifies the entry within the specified package that is to be used to build the table or list. The numerical and name/value syntax options that are available for the *x* value are defined in [“Specifying Values for the NESTED and ENTRY Attributes” on page 25](#).

first=a

specifies an optional numeric that designates the first row that is to be inserted into the table or list. The default value is 1.

last=b

specifies the last row of optional numeric data that is to be inserted into the table or list. The default is the last row in the specified entry.

Note that the LAST attribute and the N attribute are mutually exclusive. If both are specified, the last one is used.

n=c

specifies the total number of optional rows that are to be inserted into the table or list, beginning with the first row in the entry. The default is all rows in the entry.

Note that the N attribute and the LAST attribute are mutually exclusive. If both are specified, then the last one is used.

Details

Within the <SASINSERT> tags, the <SASTABLE> tag supports the development of HTML lists and tables. The <SASTABLE> tag populates tables and lists by repetitively inserting HTML tags or static text and specified data into HTML or text output. The insertion repeats for each row of data that has been specified for insertion. The location of the data and the rows of data to be inserted are determined by the attributes of the <SASTABLE> tag.

Variable Substitution

Variable substitution is valid within the <SASTABLE> open tag and the </SASTABLE> closing tag. The variable substitution syntax is as follows:

```
$(VARIABLE=variableName)
```

The variable substitution syntax specifies that the value of the VARIABLE attribute in the data set is to be substituted into the tables and lists in either HTML or plain text format. This attribute is valid only within the <SASTABLE> tag. The entry that is named in the <SASTABLE> tag must be a valid SAS data set. Any number of variable substitutions can be specified within the <SASTABLE> tag as long as each one references a valid variable in the SAS data set.

Examples

Example 1: Using <SASINSERT> and <SASTABLE> to Build a List

The following example uses the <SASINSERT> and <SASTABLE> tags to build a list. The SAS data set that is used is the second entry that is added to the main package. The value of the fileName variable is substituted on each repetition.

```
<p>
<SASINSERT>
<ul>
<SASTABLE ENTRY=2>
<li>$(VARIABLE=fileName)</li>
</SASTABLE>
</ul>
</SASINSERT>
```

Example 2: Using <SASINSERT> and <SASTABLE> to Build a Table

The following example uses the <SASINSERT> and <SASTABLE> tags to build a table. The SAS data set entry is the first entry in the main package. The value of the variables fname, lname, state, and homepage are used to create the table. The newly created table will contain one row for each row in the main package.

```
<SASINSERT>
<h1>Table Example using SASTABLE</h1>
<table border cellpadding=5
  rules=groups>
  <thead>
  <tr><th>First Name</th>
  <th>Last Name</th>
  <th>State </th>
  <th>HomePage</th></tr>
  <tbody>
  <SASTABLE ENTRY=1>
  <tr> <td> $(VARIABLE=fname)</td>
  <td> $(VARIABLE=lname)</td>
  <td> $(VARIABLE=state)</td>
  <td> <a href=$(VARIABLE=homepage)>
    $(VARIABLE=homepage)</a></td></tr>
  </SASTABLE>
  </table>
</SASINSERT>
```

<SASREPEAT> Tag

Repeats a substitution for all entries that match given criteria

Syntax

`<SASREPEAT> </SASREPEAT>`

Details

The `<SASREPEAT>` tag causes a substitution that is enclosed within the tag to repeat for all entries that match the specified name/value pair, as described in [“Specifying Name/Value Pairs” on page 48](#). Without the `<SASREPEAT>` tag, the substitution stops after matching the first entry.

Any HTML tags or static text that are included in the `<SASREPEAT>` tag are inserted into the output along with the substitution data, and those tags are repeatedly inserted each time a new entry matches the name/value pair.

The `<SASREPEAT>` tag is recognized only within the `<SASINSERT>` tag and is relevant only when it is used with name/value pair substitutions.

Examples

Example 1: Using `<SASREPEAT>` to Build a List of Reports

The following example uses the `<SASREPEAT>` tag to build a list of reports. The substitutions and the HTML tag within the `<SASREPEAT>` tag are repeated for each entry that matches the `type=report` name/value pair.

```
<SASINSERT>
Available reports include the following:
<ul>
<SASREPEAT>
<li> $(entry="(type=report)"
attribute=description)</li>
</SASREPEAT>
</ul>
</SASINSERT>
```

An example of the rendered view follows:

```
Available reports include the following:
  President's State of the Union address
  AlphaliteAirways Annual Report
  Sales Quotas for Midwest Territory
```

Example 2: Using `<SASREPEAT>` to Build a Table

The next example uses the `<SASREPEAT>` tag to build a table. The substitutions and the HTML tags within the `<SASREPEAT>` tag are repeated for each entry in the main package that matches the `type=report` name/value pair.

```
<table border="1" cellspacing="0" cellpadding="3">
<SASINSERT>
<SASREPEAT>
<tr><td>$(entry="(type=report)"
attribute=description)</td></tr>
</SASREPEAT>
</SASINSERT>
</table>
```

An example of the rendered view follows:

Table 4.2 Table Built Using <SASREPEAT> Tag

President's State of the Union Address	AlphaliteAirways Annual Report	Sales Quotas for Midwest Territory
---	-----------------------------------	---------------------------------------

<SASECHO> Tag

Stores a text string to send to the SAS Log

Syntax

```
<SASECHO text="text">
```

Details

The <SASECHO> tag aids in the diagnosis of viewer parsing and processing problems by printing a message to the SAS Log window as the viewer file is processed.

The <SASECHO> tag is recognized only within the <SASINSERT> tags. If the text value contains embedded punctuation and spaces, surround the text with quotation marks. The <SASECHO> open tag has no corresponding closing tag.

Example: Using <SASECHO>

```
<SASECHO text="Correctly executed first segment.">
```

Examples

Example 1: Sample: Using the <SASINSERT> and <SASTABLE> Tags to Build a List

The following example uses the <SASINSERT> and <SASTABLE> tags to build a list. The SAS data set that is used is the second entry that is added to the main package. The value of the **fileName** variable is substituted on each repetition. To see formatted output from <SASINSERT> and <SASTABLE> examples, see [“Example 5: Sample Viewer Template” on page 35](#).

```
<p>
<SASINSERT>
<ul>
<SASTABLE ENTRY=2>
<li>$(VARIABLE=fileName)</li>
</SASTABLE>
</ul>
</SASINSERT>
```

Example 2: Sample: Using the <SASINSERT> and <SASTABLE> Tags to Build a Table

The following example uses the <SASINSERT> and <SASTABLE> tags to build a table. The SAS data set entry is the first entry in the main package. The value of the variables **fname**, **lname**, **state**, and **homepage** are used to create the table. The newly created table will contain one row for each row in the main package. To see formatted output from <SASINSERT> and <SASTABLE> examples, see [“Example 5: Sample Viewer Template” on page 35](#).

```
<SASINSERT>
<h1>Table Example Using SASTABLE</h1>
<table border cellspacing=0 cellpadding=5
      rules=groups>
  <thead>
    <tr><th>First Name</th>
    <th>Last Name</th>
    <th>State </th>
    <th>HomePage</th></tr>
  <tbody>
    <SASTABLE ENTRY=1>
    <tr> <td> $(Variable=fname)</td>
    <td> $(Variable=lname)</td>
    <td> $(Variable=state)</td>
    <td> <a href=$(Variable=homepage)>
      $(variable=homepage)</a></td></tr>
    </SASTABLE>
    <tr><td colspan=4 align=center>
    Note: Simple table example.</td></tr>
  </table>
</SASTABLE>
```

Example 3: Sample HTML Viewer

This sample HTML viewer example is a collection of viewer coding sections. To see the e-mail output, see the output from the viewer coding sections that are described in [“How to Create a Viewer” on page 17](#). The e-mail output from this sample HTML viewer is a collection of the output from these sections.

```
<!--Section 1: Formatting a Data Set
      Variable in an HTML List-->
<SASINSERT>
<h2>Congratulations!</h2>
$(entry=1 attribute=description)
<p>
<ul>
<SASTABLE ENTRY=2>
<li>$(VARIABLE=fname)
</SASTABLE>
</ul>

<!--Section 2: Formatting a SAS Data Set
      in a Table-->
```

```

<h2>Record Sales from these Salespeople</h2>
$(entry=1 attribute=description)
<table border cellspacing=0 cellpadding=5
      rules=groups>
<thead>
<tr>
<th>First Name</th>
<th>Last Name</th>
<th>Territory</th>
</tr>
</thead>
<tbody>
<SASTABLE ENTRY=1>
<tr>
<td> $(VARIABLE=fname)</td>
<td> $(VARIABLE=lname)</td>
<td> $(VARIABLE=territory)</td>
</tr>
</tbody>
</SASTABLE>
</table>

<!--Section 3: Inserting a Text File
      and a Reference-->
<h2>Letter of Congratulations</h2>
Below is a copy of the letter that was sent
to each recipient of the top sales award.
$(entry=2 attribute=stream)
<p>
See <a href="$(entry=3 attribute=stream)">$(entry=3
attribute=stream)</a> for detailed sales figures.

<!--Section 4: Filtering an Entry-->
<h2>Message from the President</h2>
<SASREPEAT>
$(entry="(type=report)" attribute=stream)
</SASREPEAT>
</SASINSERT>

```

Example 4: Sample SAS Program with an HTML Viewer

The following SAS program example includes two parts:

- SAS code that creates two SAS data sets
- package publishing CALL routines that create a package, insert package entries, and publish the package to e-mail with the aid of a viewer file

The PACKAGE_PUBLISH CALL routine applies a viewer that is named realview.html to the package that is rendered in e-mail. The following code shows the viewer properties and attributes. To look at the content of the viewer template, see “[Example 5: Sample Viewer Template](#)” on page 35. To look at a rendered view of the package that is delivered to e-mail, see the output from the viewer coding sections that are described in “[How to Create a Viewer](#)” on page 17. The e-mail output from this sample HTML viewer is a collection of the output from these sections.

```

data empInfo;
length homePage $256;
input fname $ lname $ ages state $ siblings homePage $;
datalines;
John Smith 32 NY 4 http://alphaliteairways.com/~jsmith
Gary DeStephano 20 NY 2 http://alphaliteairways.com/~gdest
Arthur Allen 40 CA 2 http://alphaliteairways.com/~aallen
Jean Forest 3 CA 1 http://alphaliteairways.com/~jforest
Tony Votta 30 NC 2 http://www.pizza.com/~tova
Dakota Smith 3 NC 1 http://~alphaliteairways.com/~dakota
;
run;
quit;

data fileInfo;
length fileName $256;
input fileName $;
datalines;
Sales
Marketing
R&D
;
run;
quit;

data _null_;
rc=0; packageId = 0;

CALL PACKAGE_BEGIN(packageId,"Daily Orders Report.",'', rc);
if (rc eq 0) then put 'Package begin successful.';
else do;
    msg = sysmsg();
    put msg;
end;

CALL INSERT_REF(packageId, "HTML",
    "http://www.alphaliteairways.com",
    "Check out the Alphalite Airways Web site
    for more information." , "", rc);
if (rc eq 0) then put 'Insert Reference successful.';
else do;
    msg = sysmsg();
    put msg;
end;

CALL INSERT_DATASET(packageId, "work", "empInfo",
    "Data Set empInfo" , "", rc);
if (rc eq 0) then put 'Insert Data Set successful.';
else do;
    msg = sysmsg();
    put msg;
end;

CALL INSERT_DATASET(packageId, "work", "fileInfo",
    "Data Set fileInfo" , "", rc);
if (rc eq 0) then put 'Insert Data Set successful.';

```



```

else do;
    msg = sysmsg();
    put msg;
end;

viewerName='filename:realview.html';
prop='VIEWER_NAME';
address="John.Smith@alphaliteairways.com";
CALL PACKAGE_PUBLISH(packageId, "TO_EMAIL", rc,
    prop, viewerName, address);
if rc ne 0 then do;
    msg = sysmsg();
    put msg;
end;
else
    put 'Publish successful';

CALL PACKAGE_END(packageId,rc);
if rc ne 0 then do;
    msg = sysmsg();
    put msg;
end;
else
    put 'Package termination successful';

run;

```

Example 5: Sample Viewer Template

A SAS program creates a package and applies a viewer template that is named `realview.html`. During package publishing, viewer tag processing renders a view of the package for delivery via e-mail.

```

<html>
<HEAD>
<META HTTP-EQUIV="Content-Type" CONTENT="text/html;
    charset=ISO-8859-1">
<TITLE>Daily Purchase Summary</TITLE>
</HEAD>
<BODY>
<p>

<SASINSERT>
<h1>Table Example Using SASTABLE</h1>
<table border cellpadding=5
    rules=groups>
<thead>
<tr><th>First Name</th>
<th>Last Name</th>
<th>State </th>
<th>HomePage</th></tr>
</thead>
<tbody>
<SASTABLE ENTRY=2>
<tr><td> $(VARIABLE=fname)</td>

```

```

<td> $(VARIABLE=lname)</td>
<td> $(VARIABLE=state)</td>
<td> <a href="$(VARIABLE=homepage)">
    $(VARIABLE=homepage) </a> </td>
</tr>
</tbody>
</SASTABLE>
</table>

<p>
<h1>List Example Using SASTABLE</h1>
<ul>
<SASTABLE ENTRY=3>
<li>$(VARIABLE=org)</li>
</SASTABLE>
</ul>

<P>
<h2>Example Using Stream</h2>
<SASINSERT>
<a href="$(ENTRY=1 ATTRIBUTE=STREAM)">$(ENTRY=1
    ATTRIBUTE=STREAM) </a>
</SASINSERT>
<p>
</BODY>
</html>

```

Simulated Rendered View of the Package in E-mail

The following table is an example of the information that might be displayed by the preceding viewer template:

Table 4.3 Table Example Using SASTABLE

First Name	Last Name	State	HomePage
John	Smith	NY	http://alphaliteairways.com/~jsmith
Gary	DeStephano	NY	http://alphaliteairways.com/~gdest
Arthur	Allen	CA	http://alphaliteairways.com/~aallen
Jean	Forest	CA	http://alphaliteairways.com/~jforest
Tony	Votta	NC	http://pizza.com/~tova
Dakota	Smith	NC	http://alphaliteairways.com/~dakota

List Example Using SASTABLE

Sales

Marketing

R&D

Example Using Stream

<http://alphaliteairways.com>

Chapter 5

Publishing Packages

Package Publishing	40
Using a Third-Party Client Application	41
Using the Publish Package Interface	41
Publish and Retrieve Encoding Behavior	43
Default Publish and Retrieve Behavior	43
Rules for Determining File Encoding	43
Specifying an Encoding on the Retrieve	44
Filtering Packages and Package Entries	44
Overview of Filtering	44
Enabling Filtering When Publishing Packages	45
Implementing MIME-Type Filters	45
Implementing Entry-Type Filters	45
Implementing Name/Value Filters	45
Specifying Name/Value Pairs	48
Overview of Name/Value Pairs	48
Specifying Name/Value Pairs for a Package Item	48
Specifying Name/Value Pairs for an Entire Package	49
Dictionary	50
INSERT_CATALOG	50
INSERT_DATASET	51
INSERT_FILE	54
INSERT_HTML	57
INSERT_MDDb	61
INSERT_PACKAGE	62
INSERT_REF	64
INSERT_SQLVIEW	65
INSERT_VIEWER	66
PACKAGE_BEGIN	68
PACKAGE_END	71
PACKAGE_PUBLISH (Publish Package to Archive)	71
PACKAGE_PUBLISH (Publish Package to E-mail)	73
PACKAGE_PUBLISH (Publish Package to Queues)	83
PACKAGE_PUBLISH (Publish Package to Microsoft SharePoint)	86
PACKAGE_PUBLISH (Publish Package to Subscribers)	94
PACKAGE_PUBLISH (Publish Package to a WebDAV-Compliant Server) . . .	103
COMPANION_NEXT	111
ENTRY_FIRST	113
ENTRY_NEXT	114

PACKAGE_DESTROY	116
PACKAGE_FIRST	117
PACKAGE_NEXT	119
PACKAGE_TERM	121
RETRIEVE_CATALOG	121
RETRIEVE_DATASET	122
RETRIEVE_FILE	124
RETRIEVE_HTML	125
RETRIEVE_Mddb	128
RETRIEVE_NESTED	129
RETRIEVE_PACKAGE	130
RETRIEVE_REF	136
RETRIEVE_SQLVIEW	137
RETRIEVE_VIEWER	138
Examples	139
Example 1: Example: Publishing in the DATA Step	139
Example 2: Example: Publishing in a Macro	142
Example 3: Example: Publishing with the FTP Access Method	145

Package Publishing

The following activities are performed in order to publish a package:

1. Entries are inserted into the package.
2. The transport for delivering the package to the consumer is defined.
3. Other properties are defined that are specific to the transport or the rendering of the package.
4. The package is published.

The following scenarios depict how the package publishing method can depend on your role in the business enterprise or your experience as a programmer:

Table 5.1 *Package Publishing Methods for Different Publishers*

Type of Publisher	Package Publishing Method
Novice user or someone who prefers to use a GUI	Publish by using SAS Enterprise Guide or SAS Information Delivery Portal. For more information, see the product Help.
SAS programmer	Publish programmatically by using the Publish Package CALL routines. See “Using the Publish Package Interface” on page 41.
Programmer who uses a language other than SAS	Publish by writing a third-party client application. See “Using a Third-Party Client Application” on page 41.

Using a Third-Party Client Application

The publisher can write a third-party client application that uses SAS Integration Technologies to access Integrated Object Model (IOM) servers.

The IOM provides distributed object interfaces for conventional SAS features. The distributed object interfaces enable the publisher to develop component-based applications that integrate SAS features into the enterprise application.

Client development in the Java environment enables the publisher to write applets, stand-alone applications, servlets, and even Enterprise JavaBeans that interact with IOM servers. By supporting industry standards (such as Java Database Connectivity [JDBC] and CORBA), the SAS Integration Technologies software enables the publisher to take advantage of existing programming skills and toolsets for writing client applications. For more information, see the *SAS Integration Technologies: Java Client Developer's Guide*.

Client development in the Windows environment is based on the Microsoft Component Object Model (COM). Because COM support is built into the operating system and in all the leading programming language products, the publisher can integrate SAS (and existing SAS programs) into client applications. SAS Integration Technologies software provides the type libraries that are necessary to use the IOM server with Visual Basic and Visual C++. For more information, see the *SAS Integration Technologies: Windows Client Developer's Guide*.

Using the Publish Package Interface

The Publish Package Interface consists of SAS CALL routines that enable you to write SAS programs, including stored processes, that create, populate, publish, and retrieve collections of information known as packages.

The process of publishing a package follows:

1. A package is created by using the [PACKAGE_BEGIN](#) CALL routine. For example,

```
CALL PACKAGE_BEGIN(packageId, desc, nameValue, rc);
```

This CALL routine assigns a name to the package and any optional name/value pairs that are associated with it. Name/value pairs are used to assign metadata to a package or individual package entries. This metadata enables you to create filters that aid in information retrieval. The filters can be used both by subscribers to channels and by programs that search the package archive.

2. A package is populated by adding package entries by using the [INSERT_*](#) CALL routines. An entry can be a SAS file (for example, data set, catalog, or SAS MDDB), or almost any other type of file, including HTML and images. CALL routines fall into two categories of item types:
 - SAS results:
 - [INSERT_CATALOG](#)
 - [INSERT_DATASET](#)
 - [INSERT_MDDB](#)
 - [INSERT_PACKAGE](#)

- [INSERT_SQLVIEW](#)
- unstructured content:
 - [INSERT_FILE](#)
 - [INSERT_HTML](#)
 - [INSERT_REF](#)
 - [INSERT_VIEWER](#)

For example,

```
CALL INSERT_DATASET(packageId, libname, memname, description, NameValue, rc);
```

You can also nest packages by including a package as an entry in another package. Entries are referenced in the order in which they were added to the package.

Note: If inserting HTML file entries, see “[Publish and Retrieve Encoding Behavior](#)” on page 43.

3. A package is published to a delivery transport by using the [PACKAGE_PUBLISH](#) CALL routine. Supported transports are archives, e-mail addresses, message queues, SharePoint, subscribers to a pre-defined channel, and WebDAV-Compliant servers. CALL routines for supported transports are as follows:

- [TO_ARCHIVE](#)
- [TO_EMAIL](#)
- [TO_QUEUE](#)
- [TO_SHAREPOINT](#)
- [TO_SUBSCRIBERS](#)
- [TO_WEBDAV](#)

For example,

```
publishType = "TO_ARCHIVE"
.
.
.
CALL PACKAGE_PUBLISH (packageId, publishType, rc, properties, archivePath,
archiveName);
```

4. The end of the published package is defined by using the [PACKAGE_END](#) CALL routine. For example,

```
CALL PACKAGE_END(packageId, rc);
```

5. A package is retrieved from a delivery transport by using the following CALL routines:

- [COMPANION_NEXT](#)
- [ENTRY_FIRST](#)
- [ENTRY_NEXT](#)
- [PACKAGE_DESTROY](#)
- [PACKAGE_FIRST](#)
- [PACKAGE_NEXT](#)
- [PACKAGE_TERM](#)

- [RETRIEVE_CATALOG](#)
- [RETRIEVE_DATASET](#)
- [RETRIEVE_FILE](#)
- [RETRIEVE_HTML](#)
- [RETRIEVE_MDDDB](#)
- [RETRIEVE_NESTED](#)
- [RETRIEVE_PACKAGE](#)
- [RETRIEVE_REF](#)
- [RETRIEVE_SQLVIEW](#)
- [RETRIEVE_VIEWER](#)

Publish and Retrieve Encoding Behavior

Default Publish and Retrieve Behavior

All HTML files are published with a file encoding that indicates the character set of the HTML file. This encoding is either automatically generated or user-specified. All published files are read as binary data.

When retrieved, all HTML files are written as binary data. By default, no translation occurs. However, translation does occur when a file encoding is specified in the retrieve CALL routine (such as `RETRIEVE_PACKAGE`, for example).

Rules for Determining File Encoding

You can specify an encoding on the `INSERT_HTML` CALL routine to indicate the file's character set. The encoding values of `ASCII`, `EBCDIC_R15`, and `EBCDIC_R25` are treated as special cases in the following encoding rules. The file encoding that is published with each HTML file is determined by the following rules:

1. The HTML file is searched for `charset=` within the META tags. The following rules govern the search:
 - The search covers only the META tags found within the HEAD portion of the document.
 - META tags within comments are ignored.
 - By default, the search uses the encoding of the native session. If a special encoding is specified (`ASCII`, `EBCDIC_RS25`, or `EBCDIC_RS15`), the search uses that encoding rather than the native session encoding.
 - The encoding specified within the META tag always takes precedence over user-specified encodings on the `INSERT_HTML` CALL routine.
2. If the encoding value is found within the HTML file, then that value is published as the encoding value.
3. If the encoding value is not found within the HTML, and if a user-specified encoding value was not provided on the `INSERT_HTML` CALL routine, then the native session encoding is published as the encoding value.

4. If the encoding value is not found within the HTML, and if the user-specified encoding is not a special case (not ASCII, EBCDIC_RS25, or EBCDIC_RS15), then the user-specified encoding value is published as the encoding value.
5. If the encoding value is not found within the HTML file, and if a special encoding value of ASCII was specified, then the following rules apply:
 - If running on an ASCII host at publish time, then an attempt is made to use the current locale information to determine the flavor of ASCII encoding. If the locale information is unavailable, then the native session encoding is used.
 - If running on an EBCDIC host at publish time, then an attempt is made to use the current locale information to determine the transport format. If set, then the transport format is the encoding that is used. If not set, then the default becomes ISO-8859-1.
6. If the encoding value is not found within the HTML file, and if a special encoding value of EBCDIC_RS15 is specified, then an encoding value of OPEN_ED-1047 is used, regardless of the host operating environment.
7. If the encoding value is not found within the HTML file, and if a special encoding value of EBCDIC_RS25 is specified, then an encoding value of EBCDIC1047 is used, regardless of the host operating environment.

Specifying an Encoding on the Retrieve

By default, no translation occurs when HTML files are retrieved; the files are written as binary data. To override the default at retrieve time, supply an encoding property. This property indicates that the HTML files should be translated into the specified character set encoding. The encoding that is published with the file is used as the source encoding, and the user-specified encoding is used as the destination encoding.

Filtering Packages and Package Entries

Overview of Filtering

A *filter* is a property of a subscriber that enables that subscriber to receive only content that meets certain criteria. Filters can be used to exclude content that the subscriber is not interested in, or that the subscriber's computing resources cannot handle. Filters can be defined based on the entry type, MIME type, or one or more name/value pairs that are defined for the content. A filter can be an include filter, which means that the subscriber receives all content that meets the filter criteria, or an exclude filter, which means that the subscriber receives all content that does not meet the filter criteria.

When packages are published to channels, name/value filters can be used to limit the packages that are published to individual subscribers. Subscriber-specified name/value filters are compared to the name/value pairs in the published packages. If the filters match the package, then the package is published to the subscriber.

Subscribers use the Publishing Framework plug-in for SAS Management Console to define subscribers. If a subscriber specifies a delivery transport of **queue**, then that subscriber can specify additional filters to limit the package entries that are included in the packages that are published to that subscriber. Package entry or MIME type filters are compared to the entry type or MIME type of each package entry. If the package entry

type or MIME type matches the subscriber's entry type or MIME type filters, then that package entry is included in the package that is published to that queue subscriber.

Note: For each type of filter (entry type, MIME type, or name/value pair), you can define either inclusion or exclusion filters (but not both). If you have previously defined exclusion name/value filters, for example, and then specify an inclusion filter, then all of the previously defined exclusion filters are deleted from the repository.

Enabling Filtering When Publishing Packages

During package development, user-defined name/value pairs are added to packages in the [PACKAGE_BEGIN](#) CALL routine. Entry types are added to package entries automatically in the various INSERT CALL routines. User-defined MIME types are added to package entries in the [INSERT_FILE](#) CALL routine.

At publish time, filtering takes place when a package is published with the [PACKAGE_PUBLISH](#) CALL routine with a publishType of TO_SUBSCRIBERS.

Implementing MIME-Type Filters

MIME types provide details about the information that is being published. For example, specifying the MIME type `audio/basic` indicates that the file is an audio file and requires software that can interpret such content.

You can define a filter that determines the type of information the subscriber receives. For example, a subscriber who is connecting with a modem might not want to receive some data types that might be large or unwieldy, such as movies or audio. By excluding those MIME types, the subscriber never encounters those types of information.

The MIME type filters are case-insensitive filters. Like name/value pairs, MIME types are user-defined and as such need to be maintained globally to ensure consistent filtering. See the [INSERT_FILE CALL routine on page 54](#) for a list of suggested MIME types.

Implementing Entry-Type Filters

Each published package contains one or more entries. Each entry is one of several possible types. You can create a filter to include or exclude one or more entry types. Entry types are specified automatically in the various INSERT CALL routines. For a list of available package entry types, see the syntax description of the [ENTRY_FIRST CALL routine on page 113](#).

Implementing Name/Value Filters

Publishers can specify name/value pairs that describe the package that is being published. Knowledge of name/value pairs enables you to define filters for a subscriber that determine the packages that are received. If an inclusion name/value filter is defined for a subscriber, then the subscriber receives only those packages that match the name/value filter.

To implement name/value filters across your enterprise, the name/value pairs applied to packages must agree with the name/value pairs that appear in subscriber filters. Maintaining a global list of agreed-upon name/value pairs and including definitions and usage information for each name/value pair enables accurate package description and subscriber filtering in your enterprise.

The name/value filters used in your enterprise depend on the types of packages that you publish and on the types of subscribers that receive those packages. For example, you could define a channel called Maintenance that includes e-mail subscribers and an archive subscriber named MaintReports. You could add a name/value filter to the subscriber definition for the MaintReports archive subscriber that would refuse to accept packages that contain a name/value pair of **noarchive**. For this filter to be effective, packages published to the Maintenance channel would need to include the **noarchive** name/value pair in the appropriate way in order to keep unwanted packages out of the MaintReports archive. A global list of name/value pairs would help ensure that the filters and the packages both used the **noarchive** name/value pair appropriately.

A wide variety of syntax options for name/value filters gives subscribers many filtering options, including filtering based on logical relationships between multiple name/value pairs.

A name/value pair is expressed as either a name or a relationship between a name and a value in the following form:

```
name < operator value >
```

where

- *name* is a variable to which a value can be assigned. *name* is not case sensitive.
- *operator* relates the variable to the value.
- *value* is a character string or numeric value. *value* is case sensitive.

The following table lists commonly used operators:

Table 5.2 Commonly Used Operators

Comparison Operators	Logical Operators
= (equals)	& (AND)
!= (not equal)	(OR)
? (contains)	

The following is an example of a package description that uses name/value pairs that a publisher has assigned to a published package:

```
market=(Mexico, US) type=report Quarter4 sales _priority_=low
```

Subscribers can write meaningful filters if they know the conventions that a publisher uses to describe packages. The following examples illustrate filter strings that determine whether the preceding example entity would be selected by the filter. If the package meets the filter conditions, then the package is delivered to the subscriber.

```
market=(US, Asia, Europe)
```

No match. Because the equals operator (=) is used, the subscriber values and the publisher values that are assigned to the variable name MARKET must match exactly. In this example, the subscriber filters for US, Asia, and Europe, whereas the publisher assigns a value of Mexico and US. The conditions for selection are not met. Therefore, the package is not delivered to the subscriber.

```
market=(mexico, us)
```

No match. Because the equals operator (=) is used, the subscriber values and the publisher values that are assigned to the variable name MARKET must match

exactly. In this example, the subscriber values do not match the publisher values because of case differences.

market=US | market=Asia | market=Mexico

No match. Because the equals operator (=) is used, the subscriber values and the publisher values that are assigned to the variable name MARKET must match exactly. In this example, although the OR operator (|) might seem to cause a matching condition, the equals operator (=) requires that each name/value pair that is separated by an OR operator (|) match the publisher name/value pair entirely. A match would result if the subscriber values were written as follows:

market=Mexico, US | market=Asia | market=Mexico The first name/value pair in the series would match.

market=(Mexico, US)

Match. Because the equals operator (=) is used, the subscriber values and the publisher values that are assigned to the variable name MARKET must match exactly. In this example, the value set does match.

market=(US, Mexico)

Match. Because the equals operator (=) is used, the subscriber values and the publisher values that are assigned to the variable name MARKET must match exactly. In this example, the value set matches, regardless of the order of values within the value set.

market?US & market?Asia & market?Mexico

No match. The conditions that are specified in the subscriber name/value pair read: Variable name MARKET must contain the values US and Asia and Mexico. The contains comparison operator (?) identifies the eligible values for consideration. In this example, although the publisher variable MARKET contains US and Mexico, it does not also contain Asia. Because the logical AND operator (&) is used, its condition is not satisfied.

market?US | market?Asia | market?Mexico

Match. The conditions that are specified in the subscriber name/value pair read: Variable name MARKET must contain the values US or Asia or Mexico. The contains comparison operator (?) identifies the eligible values for consideration. In this example, the publisher variable MARKET contains US, and the logical OR operator (|) condition is satisfied.

Quarter4=sales

No match. Because the equals operator (=) is used, the subscriber values and the publisher values that are assigned to the variable name QUARTER4 must match exactly. In this example, because the publisher variable name QUARTER4 does not contain a value and the subscriber variable name QUARTER4 does contain a value of sales, the value sets do not match.

Quarter4

Match. Variable names are not required to have values. In this example, because the publisher variable name QUARTER4 does not have an assigned value and the subscriber variable name QUARTER4 does not have an assigned value, the value sets match.

type=report & forecast

No match. Two conditions must be met. The equals operator (=) requires that the subscriber values and the publisher values that are assigned to variable name TYPE match. In this example, the first condition is met because both the publisher and the subscriber assign the value report to variable TYPE. However, the AND logical operator (&) requires that the variable name TYPE also be assigned the value forecast. Because the publisher variable name TYPE is not assigned a value of forecast, the final condition is not met.

type=report & sales

Match. Two conditions must be met. The equals operator (=) requires that the subscriber value and the publisher value that are assigned to variable name TYPE match. In this example, the values match. Both assign the value report to the variable name TYPE. The AND logical operator (&) also requires that the variable name SALES match. Because both the publisher and the subscriber identify a variable name sales with no assigned value, the final condition is also met.

For more information about name/value pairs, see [“Specifying Name/Value Pairs” on page 48](#).

Specifying Name/Value Pairs

Overview of Name/Value Pairs

Publishers can specify name/value pairs that describe the contents of the entire package and of individual package items. With these descriptors, SAS channel subscribers can use the Publishing Framework plug-in for SAS Management Console to construct filters. For determining what packages get delivered to them in their entirety, see [“Filtering Packages and Package Entries” on page 44](#). Although subscribers can filter at the package item level for the message queue only, a developer can write retrieval programs that filter at both the package level and the package item level for all transports.

The publisher can specify one or more space-separated name/value pairs for a [package item](#) or for an [entire package](#) in the following forms:

- name
- name=value
- name="value"
- name="single value with spaces"
- name=(value)
- name=("value")
- name=(value1, "value 2",... valueN).

Specifying Name/Value Pairs for a Package Item

Here is an example of specifying a single name/value pair for a package item:

```
type=dataset
```

The publisher identifies the item in the package as a data set.

To describe the package item with finer granularity, the publisher can specify multiple name/value pairs. A space separates each name/value pair. Here is an example of specifying multiple name/value pairs for a package item:

```
type=dataset hub=RDU
```

The publisher identifies the item in the package as a data set, which is relevant only to the RDU hub.

Although a subscriber can filter at the package item level for a message queue only, a developer can write a retrieval program that filters at the package item level for all transports.

The publisher can specify name/value pairs when publishing a package item using the Publish Package Interface. When creating a package entry, you assign name/value pairs to the nameValue property in the `INSERT_entry-type` SAS CALL routine, where values for *entry-type* are as follows:

- [CATALOG](#)
- [DATASET](#)
- [FILE](#)
- [HTML](#)
- [MDDDB](#)
- [REF](#)
- [SQLVIEW](#)
- [VIEWER](#)

The following code shows the assignment of name/value pairs to a data set package entry.

```
libname = "HR";
memname = "capacityHistory";
description = "Flight Capacity History (Data)";
nameValue = "type=dataset hub=RDU";
CALL INSERT_DATASET(packageId, libname, memname,
description, nameValue, rc);
```

This nameValue property specifies a data set whose data is relevant only to the RDU hub.

For complete details about programmatically specifying name/value pairs, see [PACKAGE_BEGIN CALL routine syntax on page 68](#).

Specifying Name/Value Pairs for an Entire Package

Here is an example of specifying a single name/value pair for an entire package:

```
market=US
```

The publisher identifies the entire package as relevant only to a US market.

To describe the contents of an entire package with finer granularity, the publisher can specify multiple name/value pairs. A space separates each name/value pair. Here is another example of specifying multiple name/value pairs for an entire package:

```
market=US type=report content=ticketsales
Quarter4 priority=high
```

This high-priority package contains one or more reports about fourth-quarter ticket sales that is relevant only to a US market.

When both subscribers and developers of package-retrieval applications know about package name/value pairs, they can construct and apply filters that control package delivery. See [“Filtering Packages and Package Entries” on page 44](#).

The publisher can specify name/value pairs when publishing the package by using the Publish Package Interface. For the archive, message queue, and SAS channel subscriber

delivery types only, you assign name/value pairs to the nameValue property in the PACKAGE_BEGIN CALL routine.

The following code shows the assignment of name/value pairs to an entire package:

```
packageID=0;
rc=0;
desc = "Nightly run.";
nameValue = "market=US type=report content=ticketsales
Quarter4 priority=high";
CALL PACKAGE_BEGIN(packageId, desc, nameValue, rc);
```

This nameValue property specifies a high-priority package that contains one or more reports about fourth-quarter ticket sales that are relevant only to a US market.

For complete details about programmatically specifying name/value pairs for an entire package, see [PACKAGE_BEGIN CALL routine syntax on page 68](#).

Dictionary

INSERT_CATALOG

Inserts a SAS catalog into a package

Syntax

```
CALL INSERT_CATALOG(packageId, libname, memname, desc, nameValue, rc);
```

Required Arguments

packageID

identifies the package into which the catalog will be inserted.

Type: Numeric, Input

libname

names the library that contains the catalog.

Type: Numeric, Input

memname

specifies the name of the catalog.

Type: Character, Input

desc

describes the catalog.

Type: Character, Input

nameValue

identifies a list of one or more space-separated name/value pairs, each in one of the following forms:

- *name*
- *name=value*
- *name="value"*

- *name*="single value with spaces"
- *name*=(value)
- *name*=("value")
- *name*=(value1, "value 2",... valueN)

Name/value pairs are site-specific; they are used for the purpose of [filtering](#).

Type: Character, Input

rc

receives a return code.

Type: Numeric, Output

Example: Using INSERT_CATALOG

The following example inserts the catalog ALPHELP.PUBSUB into the PACKAGEID package.

```
libname = 'alphelp';
memname = 'pubsub';
desc = 'Publication's catalog';
nameValue='';
CALL INSERT_CATALOG(packageId, libname,
    memname, desc, nameValue, rc);
```

INSERT_DATASET

Inserts a SAS data set into a package

Syntax

CALL INSERT_DATASET(*packageId*, *libname*, *memname*, *desc*, *nameValue*, *rc*
<, *properties*, *propValue1*, ... *propValueN*>);

Required Arguments

packageID

identifies the package.

Type: Numeric, Input

libname

names the library that contains the data set.

Type: Character, Input

memname

names the data set.

Type: Character, Input

desc

describes the data set.

Type: Character, Input

nameValue

identifies a list of one or more space-separated name/value pairs, each in one of the following forms:

- *name*
- *name=value*
- *name="value"*
- *name="single value with spaces"*
- *name=(value)*
- *name=("value")*
- *name=(value1, "value 2", ... valueN)*

Name/value pairs are site-specific; they are used for the purpose of [filtering](#).

Type: Character, Input

rc

receives a return code.

Type: Numeric, Output

Optional Arguments***properties***

identifies a comma-separated list of optional property names. Valid property names are as follows:

- ALLOW_READ_PROTECTED_MEMBER
- DATASET_OPTIONS
- TRANSFORMATION_TYPE
- CSV_SEPARATOR
- CSV_FLAG

Type: Character, Input

propValue1, ...propValueN

specifies one value for each specified property. The order of the values matches the order of the property names in the *properties* parameter. Valid property values are defined as follows:

ALLOW_READ_PROTECTED_MEMBER	specifies a value of "YES". It is important to note that the password and encryption attributes are not preserved in the intermediate published format (whether on a queue or in an archive). Because of this exposure, take care when publishing data sets that are password protected, encrypted or both. The ALLOW_READ_PROTECTED_MEMBER property must be asserted on read-protected data sets in order to be published. This property ensures that the publisher realizes that this is a read-protected data set, and that the read password and encryption attributes are not preserved when stored in the
-----------------------------	---

	intermediate format. If this property is not applied, then the publish operation fails when trying to publish the read-protected data set.
DATASET_OPTIONS	specifies data set options. For a complete list of data set options, see the SAS Data Set Options topic in the SAS Help.
TRANSFORMATION_TYPE	indicates that the data set should be transformed to the specified type when published. At this time, the only supported value for this property is CSV, for Comma-Separated-Value.
CSV_SEPARATOR	indicates the separator to use when creating the CSV file. The default separator is a comma (,).
CSV_FLAG	indicates a CSV override flag. Supported values include NO_VARIABLES, NO_LABELS, and EXTENDED. By default, when writing numeric variable values into the CSV file, BEST is used to format numerics that have no format associated with them. To override this default, specify the property value EXTENDED on the CSV_FLAG property. This extends the number of digits used as the precision level. By default, if the data set is transformed into a CSV file, then the file's first line contains all of the specified variables. The second line contains all of the specified labels. To override this default behavior, specify flags with values "NO_VARIABLES" or "NO_LABELS". To specify both values, a CSV_FLAG property must be specified for each.

Type: Character or Numeric, Input

Details

When the data set is published, data set attributes are cloned so that when it is retrieved back into SAS, the created data set will have similar attributes. Attributes that are cloned include encryption, passwords, data set label, data set type, indexes and integrity constraints. It is important to know that the password and encryption attributes are not preserved in the intermediate format (whether on a queue or in an archive). Because of this exposure, take care when publishing data sets that are password-protected, encrypted, or both.

Examples

Example 1: Transforming Data into a CSV File and Publishing

The following example specifies a transformation type of CSV and two CSV_FLAG properties. The data set is transformed into a CSV file and published in CSV format.

```
prop='TRANSFORMATION_TYPE,CSV_SEPARATOR,CSV_FLAG,CSV_FLAG';
ttype='CSV';
separator='/';
flag1 = 'NO_VARIABLES';
flag2 = 'NO_LABELS';
CALL INSERT_DATASET(packageId, libname, memname, desc,
    nameValue, rc, prop, ttype, separator, flag1, flag2);
```

Example 2: Publishing a SAS Data Set in a Package

The following example inserts the SAS data set FINANCE.PAYROLL into a package.

```
libname = 'finance';
memname = 'payroll';
desc = 'Monthly payroll data.';
nameValue='';
CALL INSERT_DATASET(packageId, libname,
    memname, desc, nameValue, rc);
```

Example 3: Publishing a SAS Data Set and Applying a Password

The following example uses the DATASET_OPTIONS property to apply a password for read access and to apply a subsetting WHERE statement to the data set when publishing the package. Because the data set is read-protected, you must specify the ALLOW_READ_PROTECTED_MEMBER property. Package publishing fails without this property.

```
libname = 'hr';
memname = 'employee';
desc = 'Employee database.';
nameValue='';
properties="DATASET_OPTIONS, ALLOW_READ_PROTECTED_MEMBER";
opt="READ=abc Where=(x<10)";
allow="yes";
CALL INSERT_DATASET(packageId, libname, memname,
    desc, nameValue, rc, properties, opt, allow);
```

Example 4: Using INSERT_DATASET with the TRANSFORMATION_TYPE Property

The following example uses the TRANSFORMATION_TYPE property to publish a data set in CSV format.

```
libname = 'hr';
memname = 'employee';
desc = 'Employee database.';
nameValue='';
ttype = 'CSV';
prop = "TRANSFORMATION_TYPE";
CALL INSERT_DATASET(packageId, libname, memname,
    desc, nameValue, rc, prop, ttype);
```

INSERT_FILE

Inserts a file into a package

Syntax

CALL INSERT_FILE(*packageId*, *filename*, *filetype*, *mimeType*, *desc*, *nameValue*, *rc*
 <, *properties*, *propValue1*, ... *propValueN*>);

Required Arguments

packageID

identifies the package.

Type: Numeric, Input

filename

names the file, using the following syntax:

- FILENAME: *external_filename*
- FILEREF: *sas_fileref*

Type: Character, Input

filetype

specifies the file type, which must be TEXT or BINARY.

Type: Character, Input

mimeType

specifies the MIME type, the value of which is determined by the user. Subscribers can filter packages based on MIME type. See [“Filtering Packages and Package Entries” on page 44](#). For suggested values, see the Details section.

Type: Character, Input

desc

describes the file.

Type: Character, Input

nameValue

identifies a list of one or more space-separated name/value pairs, each in one of the following forms:

- *name*
- *name=value*
- *name="value"*
- *name="single value with spaces"*
- *name=(value)*
- *name=("value")*
- *name=(value1, "value 2", ... valueN)*

Name/value pairs are site-specific; they are used for the purpose of [filtering](#).

Type: Character, Input

rc

receives a return code.

Type: Numeric, Output

Optional Arguments

properties

identifies a comma-separated list of optional property names. Valid property names are as follows:

- DESTINATION_FILENAME
- PATH

Type: Character, Input

propValue1, ...propValueN

specifies one value for each specified property name. The order of the property values must match the order of the property names in the *properties* parameter. Valid property values are defined as follows:

DESTINATION_FILENAME specifies a valid filename that will be the destination of the file being inserted into the package. The filename should not contain any path information. This property is supported by the archive, WebDAV, and SharePoint transports.

PATH indicates the relative path information for this file. The relative path is included as the name of the file when defined in the ZIP file. The specified path should not contain a drive or device letter, or a leading slash. All slashes should be forward slashes '/' as opposed to backslashes '\'. This property is recognized only by the archive transport. It is ignored by all other transports.

Type: Character, Input

Details

The *mimeType* parameter is a user-specified MIME type that specifies the type of binary file or text file that is being published. Users might choose to document the supported values in order for publishers to use them or to use their own content strings.

Suggested MIME types include the following:

- application/msword
- application/octet-stream
- application/pdf
- application/postscript
- application/zip
- audio/basic
- image/jpeg
- image/gif
- image/tiff
- model/vrml
- text/html

- text/plain
- text/richtext
- video/mpeg
- video/quicktime

Example: Using INSERT_FILE

The following example supplies a content string of **Image/gif** to provide more information about the type of binary file that is being inserted.

```
filename = 'filename:/tmp/Report.gif';
filetype = 'binary';
desc = 'Report information';
nameValue = '';
mimetype = 'Image/gif';
CALL INSERT_FILE(packageId, filename, filetype,
  mimetype, desc, nameValue, rc);
```

INSERT_HTML

Inserts HTML files into a package

Syntax

CALL INSERT_HTML(*packageId*, *body*, *bodyUrl*, *frame*, *frameUrl*, *contents*, *contentsUrl*, *page*, *pageUrl*, *desc*, *nameValue*, *rc*<, *properties*, *propValue1*, ...*propValueN*>);

Required Arguments

packageId

identifies the package.

Type: Numeric, Input

body

names the HTML body file, using the following syntax:

- FILEREF: *SAS_fileref*
- FILENAME: *external_filename*

For information about inserting multiple body files, see the Details section.

Type: Character, Input

bodyURL

specifies the URL to be used for the body file.

Type: Character, Input

frame

names the HTML frame file, using the following syntax:

- FILEREF: *SAS_fileref*
- FILENAME: *external_filename*

Type: Character, Input

frameURL

specifies the URL to be used for the frame file.

Type: Character, Input

contents

names the HTML contents file, using the following syntax:

- FILEREF: *SAS_fileref*
- FILENAME: *external_filename*

Type: Character, Input

contentsURL

specifies the URL to be used for the contents file.

Type: Character, Input

page

names the HTML page file, using the following syntax:

- FILEREF: *SAS_fileref*
- FILENAME: *external_filename*

Type: Character, Input

pageURL

specifies the URL to be used for the page file.

Type: Character, Input

desc

describes the inserted HTML package entry.

Type: Character, Input

nameValue

identifies a list of one or more space-separated name/value pairs, each in the form of *name=value*. Name/value pairs are site-specific; they are used for the purpose of [filtering](#).

Type: Character, Input

rc

receives a return code.

Type: Numeric, Output

Optional Arguments***properties***

identifies a comma-separated list of optional property names. Valid property names are as follows:

- ENCODING
- COMPANION_FILE
- COMPANION_MIMETYPE
- COMPANION_URL
- GPATH
- GPATH_URL
- NESTED_NAME

Type: Character, Input

propValue1, ...propValueN

specifies one value for each specified property name. The order of the property values must match the order of the property names in the *properties* parameter. Valid property values are defined as follows:

ENCODING	indicates the character set of the HTML files, such as ISO-8859-1. The default encoding is assumed from the native session.
COMPANION_FILE	indicates the name of an additional HTML file that is to be added to this set of HTML files. Multiple COMPANION_FILE properties and values can be specified. Name the companion files, using either FILEREF : <i>SAS_fileref</i> or FILENAME : <i>external_filename</i> .
COMPANION_MIMETYPE	indicates the MIME type of the companion file that is to be added to the inserted HTML entry. If specified, then this property must be preceded by the COMPANION_FILE property.
COMPANION_URL	indicates the URL of an HTML file that is to be added to the inserted HTML entry. If specified, then this property must be preceded by the COMPANION_FILE property.
GPATH	indicates the name of a single directory that contains the ODS-generated graphical files for inclusion as companion files to the HTML file set. Note that all files in the specified directory are included as companion files.
GPATH_URL	indicates the URL of the directory that contains the ODS-generated graphical files. An example of a URL might be <i>~ods-output/images</i> . Alternatively, you can specify "NONE" as the GPATH_URL property value. If the value of "NONE" is specified, then only the filename is used as the URL. Note that if GPATH_URL is specified, then you must also specify the GPATH property.
NESTED_NAME	indicates the name of the nested directory to create for the storage of the set of HTML files. If you do not specify a value for this property, then a name is generated automatically. Note that the NESTED_NAME property is valid only when publishing to the WebDAV-compliant server transport.

Type: Character, Input

Details

The files that can be inserted include the body, frame, contents, and page files.

When the NEWFILE= option is specified in the ODS HTML statement, ODS might generate multiple body files. When ODS generates multiple body files, it uses a numeric file naming sequence of the general form: *bodyfilenameNumber*, as in *body1.html*,

body2.html, body3.html. To insert an entire sequence of body files, use the following syntax:

```
FILENAME: bodyFilename*.extension
```

When an asterisk is specified in the **body** parameter, an asterisk should also be specified in the **bodyUrl** parameter. For further information about ODS, see *SAS Output Delivery System: User's Guide*.

Note: As a best practice, it is suggested that a MIME type be provided for any companion files inserted into the HTML entry. The MIME type is useful for applications that will later consume or display the published package.

Examples

Example 1: Using INSERT_HTML

The following example generates ODS files and inserts those files into a package.

```
Desc='HTML output for payroll processing';
nameValue = '';
filename f '/users/f.html';
filename c '/users/c.html';
filename b '/users/b.html';
filename p '/users/p.html';
ods html frame=f contents=c(url='c.html')
      body=b(url='b.html') page=p(url='p.html');

/* insert SAS statements here to generate ODS output */

ods html close;

CALL INSERT_HTML(packageId, 'fileref:b', "b.html",
  'fileref:f', "f.html", 'fileref:c', "c.html",
  'fileref:p', "p.html", desc, nameValue, rc);
```

Example 2: Using INSERT_HTML with the ENCODING Property

The following example replaces the INSERT_HTML CALL routine in the example above with another version of the CALL routine that inserts ODS files by using the ENCODING property. In this case, the ENCODING property specifies the ISO-Latin-1 character set.

```
Desc='HTML output for payroll processing';
nameValue = '';
CALL INSERT_HTML(packageId, 'fileref:b', "b.html",
  'fileref:f', "f.html", 'fileref:c', "c.html",
  'fileref:p', "p.html", desc, nameValue, rc,
  "encoding", "ISO-8859-1");
```

Example 3: Using INSERT_HTML with a Specified Character Set Encoding

The following example specifies a character set encoding and adds two HTML files to the original set of inserted files.

```
Desc='HTML output for payroll processing';
nameValue = '';
properties='encoding, companion_file, companion_file';
```

```

encodingV = "ISO-88591-1";
file1 = "filename: report.html";
file2 = "filename: dept.html";
CALL INSERT_HTML(packageId, 'fileref:b', "b.html",
    'fileref:f', "f.html", 'fileref:c', "c.html",
    'fileref:p', "p.html", desc, nameValue, rc,
    properties, encodingV, file1, file2);

```

Example 4: Using an Asterisk (*) to Specify a Set of Files

The following example uses an asterisk (*) to specify that all body files are to be included in the set of inserted HTML files. The naming sequence used is the same as the naming sequence used in ODS. So the files body.html, body1.html, body2.html, and so on (for all files found in this sequence), will be published. For further information about the ODS naming sequence used in conjunction with the NEWLINE= option, see the *SAS Language Reference: Concepts*.

```

Desc='HTML output for payroll processing';
nameValue = '';
CALL INSERT_HTML(packageId,
    'filename:/users/jsmith/body*.html', "body*.html",
    'fileref:f', "f.html", 'fileref:c', "c.html",
    'fileref:p', "p.html", desc, nameValue, rc);

```

INSERT_MDDB

Inserts a SAS multidimensional database into a package

Syntax

```
CALL INSERT_MDDB(packageId, libname, memname, desc, nameValue, rc);
```

Required Arguments

packageID

identifies the package.

Type: Numeric, Input

libname

names the library that contains the MDDB.

Type: Character, Input

memname

names the MDDB.

Type: Character, Input

desc

describes the MDDB.

Type: Character, Input

nameValue

identifies a list of one or more space-separated name/value pairs, each in one of the following forms:

- *name*

- *name=value*
- *name="value"*
- *name="single value with spaces"*
- *name=(value)*
- *name=("value")*
- *name=(value1, "value 2", ... valueN)*

Name/value pairs are site-specific; they are used for the purpose of [filtering](#).

Type: Character, Input

rc

receives a return code.

Type: Numeric, Output

Details

An MDDB is a multidimensional database (not a data set) offered by SAS. It is a specialized storage facility where data can be pulled from a data warehouse or other data sources and stored in a matrix-like format for fast and easy access by tools such as multidimensional data viewers.

Example: Using INSERT_MDDB

The following example inserts the MDDB FINANCE.PAYROLL into the package returned in **packageId**.

```
libname = 'finance';
memname = 'payroll';
desc = 'Monthly payroll data.';
nameValue='';
CALL INSERT_MDDB(packageId, libname,
  memname, desc, nameValue, rc);
```

INSERT_PACKAGE

Inserts a package into another package

Syntax

```
CALL INSERT_PACKAGE(packageId, insertPackageId, rc
  <, properties, propValue1, ...propValueN> );
```

Required Arguments

packageId

identifies the package.

Type: Numeric, Input

insertPackageId

identifies the package that will be nested in the package identified by *packageID*.

Type: Numeric, Input

rc

receives a return code.

Type: Numeric, Output

Optional Arguments

properties

identifies a comma-separated list of optional property names. At present, only one property is supported:

- NESTED_NAME

Type: Character, Input***propValue1, ...propValueN***specifies one value for each specified property name. The order of the property values must match the order of the property names in the *properties* parameter. Valid property values are defined as follows:

NESTED_NAME indicates the name of the nested directory to create for the storage of the nested package. If you do not specify a value for this property, then a name is generated automatically. Note that the NESTED_NAME property is valid only when publishing to the WebDAV-compliant server transport.

Type: Character, Input

Details

Description and name/value parameters are not allowed on this CALL routine. Instead, this CALL routine uses the description and name/value parameters that are specified in the [PACKAGE_BEGIN](#) CALL routine.

Example: Using INSERT_PACKAGE

The following example initializes two packages (PACKAGEID and DSPID). All data sets are inserted into the package that is identified by DSPID. The package that is identified by DSPID is nested within the main package that is identified by PACKAGEID.

```
CALL PACKAGE_BEGIN(packageId,
    "Main package", '', '', rc);

CALL PACKAGE_BEGIN(dsPid, "Package
    of just data sets.", '', '', rc);

libname = 'sasuser';
memname = 'payroll';
desc = 'Monthly payroll data.';
CALL INSERT_DATASET(dsPid, libname,
    memname, desc, '', rc);

libname = 'sasuser';
memname = 'employees';
desc = 'Employee data.';
CALL INSERT_DATASET(dsPid, libname,
    memname, desc, "", rc);
```

```
/* nest data set package in main package */
CALL INSERT_PACKAGE(packageId, dsPid, rc);
```

INSERT_REF

Inserts a reference into a package

Syntax

CALL INSERT_REF(*packageId*, *referenceType*, *reference*, *desc*, *nameValue*, *rc*);

Required Arguments

packageId

identifies the package.

Type: Numeric, Input

referenceType

specifies the type of the reference. Specify HTML or URL.

Type: Character, Input

reference

specifies the reference that is to be inserted.

Type: Character, Input

desc

describes the reference.

Type: Character, Input

nameValue

identifies a list of one or more space-separated name/value pairs, each in one of the following forms:

- *name*
- *name=value*
- *name="value"*
- *name="single value with spaces"*
- *name=(value)*
- *name=("value")*
- *name=(value1, "value 2", ... valueN)*

Name/value pairs are site-specific; they are used for the purpose of filtering. See the section about [filtering](#) for more information.

Type: Character, Input

rc

receives a return code.

Type: Numeric, Output

Example: Using INSERT_REF

The following example inserts links to newly created HTML files. The package is sent by using the EMAIL transport so that subscribers receive embedded links within their e-mail messages.

```
filename myfram ftp 'odsftpf.htm';

filename mybody ftp 'odsftpb.htm';

filename mypage ftp 'odsftpp.htm';

filename mycont ftp 'odsftpc.htm';

ods listing close;
ods html frame=myfram body=mybody
      page=mypage contents=mycont;

/* insert SAS statements here to develop ODS output*/
ods html close;

desc="The SORT procedure creates a variety of ODS generated
      html output." || "An example can be viewed at :";
CALL INSERT_REF(packageId, "HTML",
      "http://alpair01.sys.com/odsftpf.htm", desc, "", rc);
if rc ne 0 then do;
      msg = sysmsg();
      put msg;
end;
else
      put 'Insert reference OK';
```

For another example, see [“Example 3: Example: Publishing with the FTP Access Method” on page 145](#).

INSERT_SQLVIEW

Inserts a PROC SQL view into a package

Syntax

```
CALL INSERT_SQLVIEW(packageId, libname, memname, desc, nameValue, rc);
```

Required Arguments

packageId

identifies the package.

Type: Numeric, Input

libname

names the library that contains the PROC SQL view.

Type: Character, Input

memname

names the PROC SQL view.

Type: Character, Input

desc

describes the PROC SQL view.

Type: Character, Input

nameValue

identifies a list of one or more space-separated name/value pairs, each in one of the following forms:

- *name*
- *name=value*
- *name="value"*
- *name="single value with spaces"*
- *name=(value)*
- *name=("value")*
- *name=(value1, "value 2", ... valueN)*

Name/value pairs are site-specific; they are used for the purpose of [filtering](#).

Type: Character, Input

rc

receives a return code.

Type: Numeric, Output

Example: Using INSERT_SQLVIEW

The following example inserts the PROC SQL view FINANCE.PAYROLL into the package that is returned in **packageId**.

```
libname = 'finance';
memname = 'payroll';
desc = 'Monthly payroll data.';
nameValue='';
CALL INSERT_SQLVIEW(packageId, libname,
    memname, desc, nameValue, rc);
```

INSERT_VIEWER

Inserts a viewer into a package

Syntax

```
CALL INSERT_VIEWER(packageId, filename, mimeType, desc, nameValue, rc
    <, properties, propValue1, ...propValueN> );
```

Required Arguments

packageID

identifies the package.

Type: Numeric, Input

filename

names the viewer, using the following syntax:

- FILENAME: *external_filename*
- FILEREF: *sas_fileref*

Type: Character, Input

mimeType

specifies the MIME type, the value of which is determined by the user. Subscribers can [filter](#) packages based on MIME type. For suggested values, see the Details section for [INSERT_FILE](#) on page 54.

Type: Character, Input

desc

describes the viewer.

Type: Character, Input

nameValue

identifies a list of one or more space-separated name/value pairs, each in one of the following forms:

- *name*
- *name=value*
- *name="value"*
- *name="single value with spaces"*
- *name=(value)*
- *name=("value")*
- *name=(value1, "value 2", ... valueN)*

Name/value pairs are site-specific; they are used for the purpose of [filtering](#).

Type: Character, Input

rc

receives a return code.

Type: Numeric, Output

Optional Arguments***properties***

identifies a comma-separated list of optional property names. Valid property names are as follows:

- ENCODING
- VIEWER_TYPE

Type: Character, Input

propValue1, ...propValueN

specifies one value for each specified property. The order of the values matches the order of the property names in the *properties* parameter. Valid property values are defined as follows:

ENCODING	indicates the character set of the viewer file, such as ISO-8859-1.
----------	---

VIEWER_TYPE indicates the type of the viewer. Valid values are HTML and TEXT. The default value is HTML.

Type: Character, Input

Example: Using INSERT_VIEWER

The following example inserts the external file HVIEWER.HTML into the package that is specified by **packageId**.

```
filename = 'filename:/tmp/hviewer.html';
desc = 'HTML viewer';
nameValue = '';
mimeType = 'text/html';
CALL INSERT_VIEWER(packageId, filename,
    mimeType, desc, nameValue, rc);
```

PACKAGE_BEGIN

Initializes a package and returns a unique package identifier

Syntax

CALL PACKAGE_BEGIN(*packageId*, *desc*, *nameValue*, *rc*
 <, *properties*, *propValue1*, ... *propValueN*>);

Required Arguments

packageId

identifies the new package.

Type: Numeric, Output

desc

describes the package.

Type: Character, Input

nameValue

identifies a list of one or more space-separated name/value pairs, each in one of the following forms:

- *name*
- *name=value*
- *name="value"*
- *name="single value with spaces"*
- *name=(value)*
- *name=("value")*
- *name=(value1, "value 2", ... valueN)*

Name/value pairs are site-specific; they are used for the purpose of [filtering](#).

Type: Character, Input

rc

receives a return code.

Type: Numeric, Output**Optional Arguments*****properties***

identifies a comma-separated list of optional property names. Valid property names are as follows:

- ABSTRACT
- EXPIRATION_DATETIME
- NAMESPACES

Type: Character, Input***propValue1, ...propValueN***specifies one value for each specified property name. The order of the property values must match the order of the property names in the *properties* parameter. Valid property values are defined as follows:

ABSTRACT	provides an abstract (short summary) of the inserted package.
EXPIRATION_DATETIME	numeric SAS datetime value. This value should be specified in GMT format.
NAMESPACES	specifies unique names that associate published packages with specific contexts on a WebDAV-compliant server. The association of a namespace with a package organizes package data on a server according to meaningful criteria or contexts. A namespace is an additional scoping criterion for a name/value description of a package or package entry. When you publish a package to WebDAV, the name/value descriptors are stored with the package or its entries to the specified WebDAV namespaces. For example, a package might be described as containing first quarter profits that were generated by the Houston office. The specified description and scope uniquely define the package so that consumers can filter name/value pairs on packages or entries unambiguously. An example of a namespace definition that you enter in the Namespaces field follows: HOUSTON='http://www.AlphaLiteAirways.com/revenue/final' . A namespace specification is case sensitive with single quotation marks surrounding embedded values. To specify multiple namespaces, separate each namespace definition with a space.

Type: Character or Numeric, Input

Details

The package identifier returned by this CALL routine is used in subsequent INSERT and PACKAGE CALL routines.

Examples

Example 1: Using PACKAGE_BEGIN

The following example initializes a package and returns the package identifier in *packageId*.

```
packageId=0;
rc=0;
desc = "Nightly run.";
nameValue='';
CALL PACKAGE_BEGIN(packageId, desc, nameValue, rc);
```

Example 2: Using PACKAGE_BEGIN and Setting an Expiration Date

The following example initializes a package with an expiration date and returns the package identifier in *packageId*.

```
packageId=0;
rc=0;
desc = "Nightly run.";
nameValue='';
dtValue = '20apr2010:08:30:00'dt;
CALL PACKAGE_BEGIN(packageId, desc, nameValue,
    rc, "EXPIRATION_DATETIME", dtValue);
```

Example 3: Using PACKAGE_BEGIN and Specifying an Abstract

The following example initializes a package with an expiration date and an abstract character string and returns the package identifier in *packageId*.

```
packageId=0;
rc=0;
desc = "Nightly run.";
nameValue='';
dtValue = '20apr2010:08:30:00'dt;
abstract = "This package contains company
    confidential information.";
properties="EXPIRATION_DATETIME, ABSTRACT";
CALL PACKAGE_BEGIN(packageId, desc, nameValue,
    rc, properties, dtValue, abstract);
```

Example 4: Using PACKAGE_BEGIN and Specifying Namespaces

The following example initializes a package with two namespaces and returns the package identifier in *packageId*.

```
packageId=0;
rc=0;
desc = "Nightly run.";
nameValue='';
namespaces = 'A="http://www.alpair.com/myNamespace1"
    B="http://www.alpair.com/myNamespace2"';
```

```
CALL PACKAGE_BEGIN(packageId, desc, nameValue,
rc, "NAMESPACES", namespaces);
```

PACKAGE_END

Frees the resources that are associated with a package

Syntax

```
CALL PACKAGE_END(packageId, rc);
```

Required Arguments

packageID

identifies the package.

Type: Numeric, Input

rc

receives a return code.

Type: Numeric, Output

Details

This CALL should be made after the completion of package publishing.

Example: Using PACKAGE_END

The following example frees the resources that are associated with the package.

```
CALL PACKAGE_END(packageId, rc);
```

PACKAGE_PUBLISH (Publish Package to Archive)

Publishes a package to an archive

Syntax

```
CALL PACKAGE_PUBLISH(packageId, publishType, rc, properties,
<propValue1, ...propValueN> );
```

Required Arguments

packageID

identifies the package that is to be published.

Type: Numeric, Input

publishType

indicates how to publish the package. To publish the package by using the archive transport, specify TO_ARCHIVE.

Type: Character, Input

rc

receives a return code.

Type: Numeric, Output***properties***

identifies a comma-separated list of optional property names. Specify any of the following property names, or specify " to indicate that no properties are to be applied:

- ARCHIVE_FULLPATH
- ARCHIVE_NAME
- ARCHIVE_PATH
- FTP_PASSWORD
- FTP_USER
- GENERATED_NAME
- HTTP_PASSWORD
- HTTP_PROXY_URL
- HTTP_USER

Type: Character, Input***Optional Argument******propValue1, ...propValueN***

specifies a value for each specified property name. The order of the property values must match the order of the property names in the *properties* parameter. Valid property values are defined as follows:

ARCHIVE_FULLPATH	returns the complete URL path of the published archive on the server. The URL path includes the name of the archive, as specified by ARCHIVE_NAME or the generated name if ARCHIVE_NAME is not specified. This output property is returned only if ARCHIVE_PATH is specified.
ARCHIVE_NAME	specifies a character string that indicates the name of the archive file.
ARCHIVE_PATH	specifies a character string that indicates the path where the archive should be created.
FTP_PASSWORD	indicates the password that is needed to log on to the remote host at which the archive will be stored. Specify this property only when the remote host is secured.
FTP_USER	indicates the user ID that is needed to log on to the remote host at which the archive will be stored. Specify this property only when the remote host is secured.
GENERATED_NAME	returns the name of the package, whether this value was generated by SAS or specified by another property. This property is an output property. If the package is published with a PARENT_URL, and

ARCHIVE_PATH is not specified, then the package is published as a folder that contains the contents of the package and not as a .spk file. In this case, the return value for GENERATED_NAME will not be the name of the archive but the name of the package. For SharePoint, this is a folder name.

HTTP_PASSWORD	indicates the password that is needed to bind to the Web server on which the package is published. Specify this property only when the Web server is secured.
HTTP_PROXY_URL	indicates the URL of the proxy server.
HTTP_USER	indicates the user ID that is needed to bind to the Web server on which the package is published. Specify this property only when the Web server is secured.

Type: Character, Input or Output

Details

The ARCHIVE_NAME property identifies the name of the archive file to create. If this property is omitted, then the archive transport generates a unique name by default.

The ARCHIVE_PATH property identifies where the archive is created. This property can be a physical pathname, an FTP URL, or an HTTP URL. If ARCHIVE_PATH is an HTTP URL on a secured server, you must specify the HTTP_USER and HTTP_PASSWORD properties. Specifying the HTTP_PROXY_URL property is optional. If ARCHIVE_PATH is an FTP URL on a secured host, then you must specify the FTP_USER and FTP_PASSWORD properties.

Note: In the z/OS operating environment, an archive can be published only to UNIX System Services directories.

Example: Publishing a Package to an Archive

```
publishType = "TO_ARCHIVE";
properties='archive_path, archive_name';
path = '/u/users';
name = 'results';
CALL PACKAGE_PUBLISH(packageId, publishType,
    rc, properties, path, name);
```

PACKAGE_PUBLISH (Publish Package to E-mail)

Publishes a package using the e-mail transport

Syntax

```
CALL PACKAGE_PUBLISH(packageId, publishType, rc, properties,
    <propValue1, ...propValueN>, address1<, ...addressN> );
```

Required Arguments***packageID***

identifies the package that is to be published.

Type: Numeric, Input

publishType

indicates how to publish the package. To publish the package by using the e-mail transport, specify TO_EMAIL.

Type: Character, Input

rc

specifies a return code.

Type: Numeric, Output

properties

identifies a comma-separated list of optional property names. Specify any of the following property names, or specify " to indicate that no properties are to be applied:

- ADDRESSLIST_DATASET_LIBNAME
- ADDRESSLIST_DATASET_MEMNAME
- ADDRESSLIST_VARIABLE_NAME
- APPLIED_TEXT_VIEWER_NAME
- APPLIED_VIEWER_NAME
- ARCHIVE_NAME
- ARCHIVE_PATH
- COLLECTION_URL
- DATASET_OPTIONS
- ENCODING
- FROM
- FTP_PASSWORD
- FTP_USER
- HTTP_PASSWORD
- HTTP_PROXY_URL
- HTTP_USER
- IF_EXISTS
- PARENT_URL
- PROCESS_VIEWER
- REPLYTO
- SENDER
- SUBJECT
- TARGET_VIEW_MIMETYPE
- TARGET_VIEW_NAME
- TARGET_VIEWER_MIMETYPE

- TARGET_VIEWER_NAME
- TEXT_VIEWER_NAME
- VIEWER_NAME

Type: Character, Input

address1 <...addressN>

specifies one or more e-mail addresses to use when publishing the package.

Type: Character, Input

Optional Argument

propValue1, ...propValueN

specifies a value for each specified property name. The order of the property values must match the order of the property names in the properties parameter. Valid property values are defined as follows:

ADDRESSLIST_DATASET_LIBNAME	an alternative to specifying explicit e-mail addresses, specifies a character string that indicates the name of the SAS library in which resides the data set from which an e-mail list can be extracted.
ADDRESSLIST_DATASET_MEMNAME	an alternative to specifying explicit e-mail addresses, specifies a character string that indicates the name of the SAS member in which resides the data set from which an e-mail list can be extracted. The data set is fully specified by library.member.
ADDRESSLIST_VARIABLE_NAME	specifies a character string that indicates the name of the variable (or column) in the data set that contains the e-mail addresses.
APPLIED_TEXT_VIEWER_NAME	specifies a character string that names the rendered package view, which results from the application of the text viewer template to the package for viewing in e-mail. To specify the name of the rendered package view, use either FILEREF : <i>SAS_fileref</i> or FILENAME : <i>external_filename</i> . This property is valid only when the TEXT_VIEWER_NAME property is also specified. By default, the rendered view is created as a temporary file. This property overrides the default, causing the rendered view to be saved permanently to a file.

APPLIED_VIEWER_NAME	specifies a character string that indicates the name of the rendered package view, which results from the application of the HTML viewer template to the package for viewing in e-mail. To specify the name of the rendered package view, use either FILEREF : <i>SAS_fileref</i> or FILENAME : <i>external_filename</i> . This property is valid only when the VIEWER_NAME property is also specified. By default, the rendered view is created as a temporary file. This property overrides the default, causing the rendered view to be saved permanently to a file.
ARCHIVE_NAME	specifies a character string that indicates the name of the archive file.
ARCHIVE_PATH	specifies a character string that indicates the path where the archive should be created.
COLLECTION_URL	specifies a character string that indicates the URL in which the WebDAV collection is placed. You assign an explicit filename to the collection. Note that when you use COLLECTION_URL, the default behavior is to replace the existing collection at that location.
CORRELATIONID	specifies a binary character string correlator that is used on the package header message.
DATASET_OPTIONS	specifies a character string that indicates the options to use for opening and accessing a SAS data set that contains e-mail addresses that are used to populate <i>addressn</i> . Specify this property as <i>valuevalueoption1=option2=</i>
ENCODING	specifies a character string that indicates the text encoding to use for the message body.
FROM	specifies a character string that indicates the name or e-mail address of the sender (or package publisher) of the e-mail message. This value is the name or e-mail

	address that the e-mail will appear to be from. Note that the FROM field is valid only with the SMTP e-mail interface.
FTP_PASSWORD	indicates the password that is needed to log on to the remote host at which the archive will be stored. Specify this property only when the remote host is secured.
FTP_USER	indicates the user ID that is needed to log on to the remote host at which the archive will be stored. Specify this property only when the remote host is secured.
HTTP_PASSWORD	indicates the password that is needed to bind to the Web server on which the package is published. Specify this property only when the Web server is secured.
HTTP_PROXY_URL	indicates the URL of the proxy server.
HTTP_USER	indicates the user ID that is needed to bind to the Web server on which the package is published. Specify this property only when the Web server is secured.
IF_EXISTS	specifies one of the following character strings. Use the IF_EXISTS property to control the treatment of same-named collections already existing on the server. "NOREPLACE" indicates that if the package being published contains a collection that already exists on the server, the PUBLISH_PACKAGE call is to return immediately without affecting the contents of the existing collection. "UPDATE" indicates that if the collection already exists on the server, the PUBLISH_PACKAGE call is to update the existing collection by replacing like-named entities and adding newly named entities. If "UPDATE" is specified and both the package to publish and the existing collection have an HTML set (created with INSERT_HTML) with the same NESTED_NAME, then the

HTML set in the published package replaces the HTML set in the existing collection. "UPDATEANY" is identical to "UPDATE" except that the PUBLISH_PACKAGE CALL routine can be used to update a collection that SAS did not create. A consequence of using "UPDATEANY" is that SAS will be unable to retrieve the published package. Note that when names are generated automatically for HTML set collections, the publish code ensures that name collisions will not occur.

PARENT_URL

specifies a character string that indicates the URL under which the WebDAV collection is placed. The collection is automatically assigned a unique name.

PROCESS_VIEWER

specifies a character string of "yes" to indicate that the rendered view will be delivered in e-mail. If you specify the PROCESS_VIEWER property with the ARCHIVE_PATH property, then the archive is created but is not sent as an attachment in e-mail. Instead, viewer processing occurs and the rendered view is sent in e-mail.

REPLYTO

specifies a character string that indicates the designated e-mail address to which package recipients might respond. Note that the REPLYTO field is valid only with the SMTP e-mail interface.

SENDER

specifies a character string that indicates the e-mail address of the sender (or package publisher) of the e-mail message. A valid e-mail address should be specified. This address will receive any bounced or undeliverable e-mail. This value is the actual e-mail address that the e-mail is sent from.

SUBJECT	specifies a character string that provides the subject line for the e-mail message.
TARGET_VIEW_MIMETYPE	specifies a character string that indicates the MIME type of the rendered view for delivery to a WebDAV-compliant server. The target view MIME type overrides the default view MIME type, which is automatically inferred from the viewer. Typical MIME types are HTML (.htm) and plain text (.txt) files. If this field remains blank, then the viewer filename extension is used to locate the MIME type in the appropriate registry. Windows hosts use the Windows Registry; other hosts use the SAS Registry.
TARGET_VIEW_NAME	specifies a character string that indicates the name of the rendered view for delivery to a WebDAV-compliant server. The specified target view name overrides the default name, which is index.html.
TARGET_VIEWER_MIMETYPE	see TARGET_VIEW_MIMETYPE.
TARGET_VIEWER_NAME	see TARGET_VIEW_NAME.
TEXT_VIEWER_NAME	specifies a character string that indicates the name of a text viewer template that formats package content for viewing in e-mail by using either FILEREF : <i>SAS_fileref</i> or FILENAME : <i>external_filename</i> . A text viewer template might be necessary if the destination e-mail program does not support the HTML MIME type.
VIEWER_NAME	specifies a character string that indicates the name of the HTML viewer template to be applied when publishing e-mail by using either FILEREF : <i>SAS_fileref</i> or FILENAME : <i>external_filename</i> .

Type: Character or Numeric, Input

Details

Default Behavior

When publishing to e-mail, the e-mail message is sent in plain text format by default. Only inserted reference entries are published to e-mail. For details about inserting reference entries, see the [INSERT_REF CALL routine on page 64](#).

The package description field precedes the reference value in the e-mail message. All other entries that are inserted into the package are ignored.

To override the default behavior, you can specify the ARCHIVE_PATH, COLLECTION_URL, PARENT_URL, TEXT_VIEWER_NAME, or VIEWER_NAME properties.

Note: If the mailer is not running in a Windows NT operating environment, then you will be prompted for the mail profile to use when you send the e-mail message. To avoid being prompted, specify the EMAILID and EMAILPW options at SAS invocation. For example:

```
sas -EMAILID "Microsoft Outlook"
```

Archive Path Properties

If you specify the ARCHIVE_PATH property, then an archive is created and published as an e-mail attachment. All entries that are inserted into the package are published as an archive. If you specify a value for ARCHIVE_PATH, then the created archive is stored at the designated location. To create a temporary archive that is deleted after the package is published, specify an ARCHIVE_PATH value of "" or "tempfile".

If you specify ARCHIVE_PATH as an FTP URL or as an HTTP URL, and need details about archive properties, see [PACKAGE_PUBLISH \(Publish Package to Archive\) on page 71](#).

Note: In order to create an archive under the z/OS operating environment, the z/OS environment must support UNIX System Services directories.

If you specify the PROCESS_VIEWER property (with either the VIEWER_NAME or TEXT_VIEWER_NAME property) along with the ARCHIVE_PATH property, then the archive is created but is not sent as an attachment in e-mail. Instead, viewer processing occurs and the rendered view is sent in e-mail.

For more information about the application of viewer properties, see [Chapter 4, “Viewer Processing,” on page 15](#).

When publishing to an archive with the e-mail transport, you can specify the following archive properties: ARCHIVE_NAME, ARCHIVE_PATH, FTP_PASSWORD, FTP_USER, HTTP_PASSWORD, HTTP_PROXY_URL, or HTTP_USER.

Viewer Properties

If you specify the VIEWER_NAME or TEXT_VIEWER_NAME property, then the viewer is used to create the e-mail message and to apply substitutions. VIEWER_NAME renders the view in HTML format. TEXT_VIEWER_NAME renders the view in text format. Only the package information that is rendered by the viewer is published.

If you specify the PROCESS_VIEWER property (with either the VIEWER_NAME or TEXT_VIEWER_NAME property) along with the ARCHIVE_PATH property, then the archive is created but is not sent as an attachment in e-mail. Instead, viewer processing occurs and the rendered view is sent in e-mail.

WebDAV Properties

If you specify the `COLLECTION_URL` property, then the package is published to the specified URL on a WebDAV-compliant Web server. An example of a collection URL is `http://www.host.com/AlphaliteAirways/revenue/quarter1`. The collection is named `quarter1`. The e-mail message that is sent to subscribers will contain a reference to the URL that is specified in the `COLLECTION_URL` property.

The `PARENT_URL` property is similar to the `COLLECTION_URL` property except that it specifies the location under which the new WebDAV collection is to be placed. The `PUBLISH_PACKAGE CALL` routine generates a unique name for the new collection. The unique name is limited to eight characters, with the first character as an `s`. An example of a parent URL directory location is `http://www.host.com/AlphaliteAirways/revenue`. An example of a collection name that is automatically generated might be `s9811239`. The e-mail message contains a reference to the collection, which is the URL that you specified in the `PARENT_URL` property.

The specifications of `COLLECTION_URL` and `PARENT_URL` are mutually exclusive.

When publishing to a WebDAV-compliant server with the e-mail transport, you can specify the following WebDAV properties: `HTTP_PASSWORD`, `HTTP_PROXY_URL`, `HTTP_USER`, `IF_EXISTS`, `TARGET_VIEW_MIMETYPE`, `TARGET_VIEW_NAME`, and `VIEWER_NAME` (or `TEXT_VIEWER_NAME`).

WebDAV publishing uses the following file extensions for each item type:

Table 5.3 File Extensions for Item Types

Item Type	File Extension
CATALOG	.sac
DATA	.sad
MDDDB	.sam
REFERENCE	.ref
VIEW	.sav

Examples

Example 1: Using `PACKAGE_PUBLISH` to Publish to E-mail

```
publishType = "TO_EMAIL";
properties='';
CALL PACKAGE_PUBLISH(packageId, publishType, rc, properties,
    "user1@alphaliteairways.com", "John Smith",
    "jsmith@alphaliteairways.com");
```

Example 2: Using `PACKAGE_PUBLISH` to Publish to E-mail with the `SUBJECT` Property

```
publishType = "TO_EMAIL";
subject = "Nightly Builds Update";
properties="SUBJECT";
Addr = "admins-l@alphaliteair03.vm.com";
```

```
CALL PACKAGE_PUBLISH(packageId, publishType,
rc, properties, subject, Addr);
```

Example 3: Using `PACKAGE_PUBLISH` to Publish to Two E-mail Addresses with the `APPLIED_VIEWER_NAME` Property

The following example publishes a package to two e-mail addresses and designates the viewer to be used when formatting the e-mail message. The e-mail message will contain only content that can be rendered in a view. The rendered view is deleted after it is published.

In order to save the rendered view explicitly, you can specify the `APPLIED_VIEWER_NAME` property and a filename value.

```
publishType = "TO_EMAIL";
properties="SUBJECT, VIEWER_NAME";
subject = "Nightly Build Updates";
viewer = "filename:template.html";
Addr = "admins-l@alphaliteair03.vm.com";
CALL PACKAGE_PUBLISH(packageId, publishType,
rc, properties, subject, viewer,
"buildmonitor@alphaliteairways.com", Addr);
```

Example 4: Using `PACKAGE_PUBLISH` to Publish to E-mail with the `ARCHIVE_PATH` Property

```
publishType = "TO_EMAIL";
properties="ARCHIVE_PATH";
apath = "/u/users1";
Addr = "admins-l@alphaliteair05";
CALL PACKAGE_PUBLISH(packageId, publishType,
rc, properties, apath, Addr);
```

Example 5: Using `PACKAGE_PUBLISH` to Publish a Collection URL on a WebDAV-Compliant Server to E-mail

The following example uses the e-mail transport to publish a collection URL on a WebDAV-compliant server. The HTTP user ID and password enable the publisher to bind to the secured HTTP server. All e-mail recipients who are members of the mail list receive the e-mail announcement that the best rates are accessible at the specified URL.

```
publishType = "TO_EMAIL";
properties="COLLECTION_URL, SUBJECT",
"HTTP_USER", "HTTP_PASSWORD";
collurl="http://www.alphaliteairways/fares/discount";
subj="Announcing Best Rates Yet";
http_user="vicdamone";
http_password="myway";
Addr = "admins-l@alphaliteair05";
CALL PACKAGE_PUBLISH(packageId, publishType, rc, properties,
collurl, subj, http_user, http_password, Addr);
```

Example 6: Using `PACKAGE_PUBLISH` to Publish to E-mail Addresses That Are in a Password-Protected SAS Data Set

The following example specifies e-mail addresses that are stored in a variable in a password-protected SAS data set.


```

publishType = "TO_EMAIL";
properties = "SUBJECT, ADDRESS_DATASET_LIBNAME,
             ADDRESS_DATASET_MEMNAME, ADDRESSLIST_VARIABLE_NAME,
             DATASET_OPTIONS";
subject = "Get out and Vote!";
lib = "voterreg";
mem = "northeast";
var = "emailaddr";
opt = "pw='born2run'";
CALL PACKAGE_PUBLISH(packageId, publishType, rc,
                     properties, subject, lib, mem, var, opt);

```

PACKAGE_PUBLISH (Publish Package to Queues)

Publishes a package to one or more message queues

Syntax

CALL PACKAGE_PUBLISH(*packageId*, *publishType*, *rc*, *properties*,
 <*propValue1*, ...*propValueN*> , *queue1*<, ...*queueN*>);

Required Arguments

packageID

identifies the package that is to be published.

Type: Numeric, Input

publishType

indicates how to publish the package. To publish the package by using the queue transport, specify a *publishType* of TO_QUEUE.

Type: Character, Input

rc

receives a return code.

Type: Numeric, Output

properties

identifies a comma-separated list of optional property names. Specify any of the following property names, or specify " to indicate that no properties are to be applied:

- ARCHIVE_NAME
- ARCHIVE_PATH
- CORRELATIONID
- FTP_PASSWORD
- FTP_USER
- HTTP_PASSWORD
- HTTP_PROXY_URL
- HTTP_USER

Type: Character, Input

queue1 <...queueN>

character string that specifies the queue(s) that will be used to publish the package. When publishing to MSMQ queues, use the following syntax:

```
MSMQ://queueHostMachineName\queueName
```

When publishing to IBM WebSphere MQ queues, use the following syntax:

```
MQSERIES://queueManager:queueName
```

or

```
MQSERIES-C://queueManager:queueName
```

MQSERIES specifies the server interface that is used to connect to a queue manager that is local. MQSERIES-C specifies the client interface that is used, allowing the queue manager to be local or remote.

Type: Character, Input

Optional Argument***propValue1, ...propValueN***

specifies one value for each specified property name. The order of the property values must match the order of the property names in the properties parameter. Valid property values are defined as follows:

ARCHIVE_NAME	specifies a character string that indicates the name of the archive file.
ARCHIVE_PATH	specifies a character string that indicates the path where the archive should be created.
CORRELATIONID	specifies a binary character string correlator that is used on the package header message.
FTP_PASSWORD	indicates the password that is needed to log on to the remote host at which the archive will be stored. Specify this property only when the remote host is secured.
FTP_USER	indicates the user ID that is needed to log on to the remote host at which the archive will be stored. Specify this property only when the remote host is secured.
HTTP_PASSWORD	indicates the password that is needed to bind to the Web server on which the package is published. Specify this property only when the Web server is secured.
HTTP_PROXY_URL	indicates the URL of the proxy server.
HTTP_USER	indicates the user ID that is needed to bind to the Web server on which the package is published. Specify this property only when the Web server is secured.

Type: Character or Numeric, Input

Details

When publishing to a queue, all entries in the package are published to the queue by default. To override this default, specify the ARCHIVE_PATH property, which indicates that an archive is to be created and only the archive will be published to the queue. The archive will contain all package entries.

If you specify a value for ARCHIVE_PATH, then the archive is stored at the designated location. To create a temporary archive that is deleted after the package is published, specify an ARCHIVE_PATH value of "" or "tempfile".

If you specify ARCHIVE_PATH as an FTP URL or as an HTTP URL, and need details about specifying archive properties, see [PACKAGE_PUBLISH \(Publish Package to Archive\) on page 71](#).

Note: In the z/OS operating environment, you can publish archives only to UNIX System Services directories.

Queues that support transactional units of work are recommended. By using these types of queues, the queue transport prevents partial packages from remaining on the queue in cases where errors are encountered during package publishing. For MSMQ, this means that the queue should be transactional. For IBM WebSphere MQ, this means that the queue should support synchronization points.

When you specify the CORRELATIONID property, the package message uses the specified CORRELATIONID value. You can retrieve packages from the queue by correlation ID.

Examples

Example 1: Using PACKAGE_PUBLISH to Publish to Two Queues with the CORRELATIONID Property

The following example publishes a package to two queues. One queue is an IBM WebSphere MQ queue that is named PCONE; the second queue is an MSMQ queue that is specified by the queue manager, who is named JSMITH. A CORRELATIONID of 12345678901234567890 is assigned to the package to be published to both queues.

```
publishType = "TO_QUEUE";
FirstQ = "MQSERIES://PCONE:LOCAL";
SecondQ = "MSMQ://JSMITH:TRANSQ";
CorrValue = "12345678901234567890";
CALL PACKAGE_PUBLISH(packageId, publishType, rc,
    "CORRELATIONID", CorrValue, firstQ, secondQ);
```

Example 2: Using PACKAGE_PUBLISH to Publish to a Queue

The following example publishes the package to one queue and does not apply any additional queue properties:

```
publishType = "TO_QUEUE";
firstQ = "MQSERIES://PCONE:MYQ";
CALL PACKAGE_PUBLISH(packageId,
    publishType, rc, '', firstQ);
```

Example 3: Using PACKAGE_PUBLISH to Publish an Archive to a Queue with the ARCHIVE_PATH Property

The following example creates an archive and publishes it to a queue. The ARCHIVE_PATH property is specified as "tempfile". After the archive is published to the queue, the temporary, local copy is deleted automatically. The archive contains all entries that are inserted into the package.

```
publishType = "TO_QUEUE";
firstQ = "MQSERIES://PCONE:MYQ";
prop = "ARCHIVE_PATH";
archivePath = "tempfile";
```

```
CALL PACKAGE_PUBLISH(packageId, publishType,
    rc, prop, archivePath, firstQ);
```

PACKAGE_PUBLISH (Publish Package to Microsoft SharePoint)

Publishes a package to SharePoint

Syntax

```
CALL PACKAGE_PUBLISH(packageId, publishType, rc, properties,
    <propValue1, ...propValueN> );
```

Required Arguments

packageId

identifies the package that is to be published.

Type: Numeric, Input

publishType

indicates how to publish the package. To publish the package using the SharePoint transport, specify a *publishType* of TO_SHAREPOINT.

Type: Character, Input

rc

receives a return code.

Type: Numeric, Output

properties

identifies a comma-separated list of optional property names. Specify any of the following property names, or specify " to indicate that no properties are to be applied:

- APPLIED_TEXT_VIEWER_NAME
- APPLIED_VIEWER_NAME
- ARCHIVE_FULLPATH
- ARCHIVE_NAME
- ARCHIVE_PATH
- COLLECTION_FOLDER
- COLLECTION_URL
- DEBUG_FILE
- GENERATED_NAME
- HTTP_PASSWORD
- HTTP_USER
- IF_EXISTS
- INITIALIZE_SITE
- LIST_NAME
- PARENT_FOLDER

- PARENT_URL
- SITE_URL
- TARGET_VIEW_MIMETYPE
- TARGET_VIEW_NAME
- TARGET_VIEWER_MIMETYPE
- TARGET_VIEWER_NAME
- TEXT_VIEWER_NAME
- VIEWER_NAME

Type: Character, Input

Optional Argument

propValue1, ...propValueN

specifies one value for each specified property name. The order of the property values must match the order of the property names in the properties parameter. Valid property values are defined as follows:

APPLIED_TEXT_VIEWER_NAME specifies a character string that names the rendered package view, which results from the application of the text viewer template. To specify the name of the rendered package view, use either **FILEREF**:*SAS_fileref* or **FILENAME**:*external_filename*. This property is valid only when the TEXT_VIEWER_NAME property is also specified. By default, the rendered view is created as a temporary file. This property overrides the default, causing the rendered view to be saved permanently to a file.

APPLIED_VIEWER_NAME specifies a character string that indicates the name of the rendered package view, which results from the application of the HTML viewer template. To specify the name of the rendered package view, use either **FILEREF**:*SAS_fileref* or **FILENAME**:*external_filename*. This property is valid only when the VIEWER_NAME property is also specified. By default, the rendered view is created as a temporary file. This property overrides the default, causing the rendered view to be saved permanently to a file.

ARCHIVE_FULLPATH returns the complete URL path of the published archive on the server. The URL path includes the name of the archive, as specified by ARCHIVE_NAME or the generated name if ARCHIVE_NAME is not

	specified. This output property is returned only if ARCHIVE_PATH is specified.
ARCHIVE_NAME	specifies a character string that indicates the name of the archive file.
ARCHIVE_PATH	specifies a character string that indicates the path where the archive should be created.
COLLECTION_FOLDER	specifies both the parent folder and the collection folder together. This property is relative to the list name. Do not specify a value for this property if you are using the COLLECTION_URL property.
COLLECTION_URL	specifies a character string that indicates the URL in which the SharePoint collection is placed. The COLLECTION_URL is specified in the following format: <i>http://host/site/list-name/folder-name</i> . This is the equivalent of specifying the SITE_URL, LIST_NAME, and COLLECTION_FOLDER properties separately. Do not specify a value for this property if you are using the SITE_URL, LIST_NAME, or COLLECTION_FOLDER properties. Note that when you use COLLECTION_URL, the default behavior is to replace the existing collection at that location. Do not use this property if the SharePoint URL has one of the following formats: <i>http://host/site/subsiteN/list-name/folder-name</i> or <i>http://host-is-the-sitename/list-name/folder-name</i> .
DEBUG_FILE	specifies the name of the file that will contain debug wire trace output.
GENERATED_NAME	returns the name of the package, whether this value was generated by SAS or specified by another property. This property is an output property. If the package is published with a PARENT_URL, and ARCHIVE_PATH is not specified, then the package is published as a folder that contains the contents of the package and not as a .spk file. In this case, the return value for GENERATED_NAME will not be the name of the archive but the name of the package. For SharePoint, this is a folder name.

HTTP_PASSWORD	indicates the password that is needed to bind to the Web server on which the package is published. Specify this property only when the Web server is secured.
HTTP_USER	indicates the user ID that is needed to bind to the Web server on which the package is published. Specify this property only when the Web server is secured.
IF_EXISTS	specifies one of the following character strings. Use the IF_EXISTS property to control the treatment of same-named collections already existing on the server. "NOREPLACE" indicates that if the package being published contains a collection that already exists on the server, the PUBLISH_PACKAGE call is to return immediately without affecting the contents of the existing collection. "UPDATE" indicates that if the collection already exists on the server, the PUBLISH_PACKAGE call is to update the existing collection by replacing like-named entities and adding newly named entities. If "UPDATE" is specified and both the package to publish and the existing collection have an HTML set (created with INSERT_HTML) with the same NESTED_NAME, then the HTML set in the published package replaces the HTML set in the existing collection. "UPDATEANY" is identical to "UPDATE" except that the PUBLISH_PACKAGE CALL routine can be used to update a collection that SAS did not create. A consequence of using "UPDATEANY" is that SAS will be unable to retrieve the published package. Note that when names are generated automatically for HTML set collections, the publish code ensures that name collisions will not occur.
INITIALIZE_SITE	enables an administrator to initialize a SharePoint site. Before a SharePoint site is used for the first time, it must be initialized with SharePoint content types and column metadata defined by SAS.
LIST_NAME	specifies a document library in the SharePoint site. SharePoint document libraries are a special type of list that is

	used for ordering folders and files. Do not specify a value for this property if you are using the <code>COLLECTION_URL</code> or <code>PARENT_URL</code> properties.
<code>PARENT_FOLDER</code>	specifies the parent folder of a generated collection name. This property is relative to the list name. Do not specify a value for this property if you are using the <code>PARENT_URL</code> property.
<code>PARENT_URL</code>	specifies a character string that indicates the URL under which the WebDAV collection is placed. The collection is automatically assigned a unique name. Do not specify a value for this property if you are using the <code>SITE_URL</code> , <code>LIST_NAME</code> , or <code>PARENT_FOLDER</code> properties.
<code>SITE_URL</code>	defines the transfer protocol (HTTP or HTTPS), the host name, and the SharePoint site. Do not specify a value for this property if you are using the <code>COLLECTION_URL</code> or <code>PARENT_URL</code> properties.
<code>TARGET_VIEW_MIMETYPE</code>	specifies a character string that indicates the MIME type of the rendered view for delivery to a SharePoint server. The target view MIME type overrides the default view MIME type, which is automatically inferred from the viewer. Typical MIME types are HTML (.htm) and plain text (.txt) files. If this field remains blank, then the viewer filename extension is used to locate the MIME type in the appropriate registry. Windows hosts use the Windows Registry; other hosts use the SAS Registry. The <code>TARGET_VIEW_MIMETYPE</code> and <code>TARGET_VIEWER_MIMETYPE</code> properties are synonymous.
<code>TARGET_VIEW_NAME</code>	specifies a character string that indicates the name of the rendered view that is published with the package to the SharePoint server. The specified target view name overrides the default name, which is <code>index.html</code> . The <code>TARGET_VIEW_NAME</code> and <code>TARGET_VIEWER_NAME</code> properties are synonymous.
<code>TARGET_VIEWER_MIMETYPE</code>	see <code>TARGET_VIEW_MIMETYPE</code> .
<code>TARGET_VIEWER_NAME</code>	see <code>TARGET_VIEW_NAME</code> .

TEXT_VIEWER_NAME	specifies a character string that indicates the name of a text viewer template that formats package content for later viewing by a Web browser or text editor or other program that is specific to the viewer template. Syntax is either FILEREF : <i>SAS_fileref</i> or FILENAME : <i>external_filename</i> .
VIEWER_NAME	specifies a character string that indicates the name of the HTML viewer template to be applied by using either FILEREF : <i>SAS_fileref</i> or FILENAME : <i>external_filename</i> .

Type: Character or Numeric, Input or Output

Details

Default Behavior

Publishing with a *publishType* of TO_SHAREPOINT publishes a package to a specified URL on a SharePoint server. The HTTPS protocol is supported when publishing to a SharePoint site. SharePoint enables distributed authoring and versioning, which enables collaborative development of Web files on remote servers.

The SharePoint transport stores package entries as members of a collection.

If you specify the COLLECTION_URL property, then the package is published to the specified URL on a SharePoint server. When you use COLLECTION_URL, the default behavior is to replace the existing collection and its nested directories at that location. If you do not want to replace an existing collection and its nested directories, then you must use the IF_EXISTS property. An example of a collection URL is

```
http://www.host.com/AlphaliteAirways/revenue/quarter1
```

The collection is named quarter1.

The PARENT_URL property is similar to the COLLECTION_URL property except that it specifies the location under which the new SharePoint collection is to be placed. The PUBLISH_PACKAGE CALL routine generates a unique name for the new collection. The unique name is limited to eight characters with the first character as an s. An example of a parent URL directory location is `http://www.host.com/AlphaliteAirways/revenue`. An example of a collection name that is automatically generated might be s9811239.

The specifications of the COLLECTION_URL property and the PARENT_URL property are mutually exclusive.

Publishing to SharePoint is similar to publishing to WebDAV. The primary difference is how the file destination is specified in the PACKAGE_PUBLISH CALL routine. Instead of the package destination being a single URL, SharePoint requires three pieces of information: SITE_URL, LIST_NAME, and PARENT_FOLDER or COLLECTION_FOLDER. However, if the SharePoint site is not nested, a single COLLECTION_URL or PARENT_URL property, which contains all information, can be used (for example, `http://host/site/list-name/folder-name`). SharePoint sites can be nested. For example, a site could be `http://myhost/MyMainSite/SiteUnderneath`. Because SharePoint sites can be nested, the SITE_NAME, LIST_NAME, and COLLECTION_FOLDER properties must be

specified separately. To view a collection with a Web browser, combine all three pieces of information (for example, `http://host/site/list-name/folder-name`).

A `COLLECTION_URL` can represent only a top-level site because the SharePoint URL syntax does not allow for parsing to determine how many nested levels there are. For example, `http://host/topsite/nestedsite/Documents` is a nested site URL. For a nested site, the `SITE_NAME`, `LIST_NAME`, and `COLLECTION_FOLDER` properties must be used. If a `COLLECTION_URL` is mistakenly used with a nested site URL, the caller will receive an HTTP status of 404 (NOT FOUND) because the URL will not be properly parsed.

Similar to `COLLECTION_URL`, `PARENT_URL` can be used only with a top-level SharePoint site. For nested sites, `SITE_NAME`, `LIST_NAME`, and `PARENT_FOLDER` properties must be used.

SharePoint publishing uses the following file extensions for each item type:

Table 5.4 File Extensions for Item Types

Item Type	File Extension
ARCHIVE	.spk
CATALOG	.stc
COMMA-SEPARATED VALUES	.csv
DATA	.stc
MDDb	.sam
REFERENCE	.ref
VIEW	.sav

Initializing the Site

Before a SharePoint site is used for the first time, it must be initialized with SharePoint content types and column metadata defined by SAS. Use the `INITIALIZE_SITE` property to initialize the SharePoint site.

Note that after a document library list is initialized, you can manually enable the use of content types. This step is useful if you want to view the SharePoint site with a Web browser and view the properties on the file, such as the description or expiration date. To enable the use of content types, go to **Site Actions** and select **Site Libraries and Lists**. Then customize **Shared Documents** and click **Advanced Settings**. Select **Yes** to allow management of content types.

To update an existing package in a site document library, you must first make sure the site library allows updates to existing packages. Go to **Site Actions** and select **Site Settings**. Next, select **Site Libraries and Lists** ⇒ **Documents** (or Shared Documents). Under **Versioning settings**, set **Require documents to be checked out before they can be edited?** to **No**.

Viewer Properties

If you specify the `VIEWER_NAME` property with the `COLLECTION_URL` or `PARENT_URL` property, then the view is rendered in HTML format. If you specify the

TEXT_VIEWER_NAME with the COLLECTION_URL or PARENT_URL properties, then the view is rendered in text format.

The specified viewer is used to create a rendered view that is named index.html. To override the default name that is assigned to the rendered view, use the APPLIED_VIEWER_NAME or APPLIED_TEXT_VIEWER_NAME, as appropriate, to specify a filename for the rendered view.

Archive Path Properties

If you specify the ARCHIVE_PATH property, then an archive is created and published as a binary package on a SharePoint server. All entries that are inserted into the package are published as an archive. If you specify a value for ARCHIVE_PATH, then the created archive is stored at the designated location. To create a temporary archive that is deleted after the package is published, specify an ARCHIVE_PATH value of "" or "tempfile".

For more details about how to use the archive properties, see [“PACKAGE_PUBLISH \(Publish Package to Archive\)” on page 71](#).

Note: In order to create an archive under the z/OS operating environment, the z/OS environment must support UNIX System Services directories.

When publishing a binary package with the SharePoint transport, you can specify the following archive properties: ARCHIVE_NAME, ARCHIVE_PATH, or ARCHIVE_FULLPATH.

Applying a Name/Value Pair to a Package and a Package Item

When publishing to SharePoint, optional name/value pairs are transmitted to the SharePoint server in XML format.

For details about specifying the *nameValue* parameter for a single package item, see the applicable INSERT_item CALL routine, where *item* can be any of the following:

- [CATALOG](#)
- [DATASET](#)
- [FILE](#)
- [HTML](#)
- [MDDDB](#)
- [PROC SQL VIEW](#)
- [REFERENCE](#)
- [VIEWER](#)

Examples

Example 1: Using PACKAGE_PUBLISH with the COLLECTION_FOLDER Property to Publish to SharePoint

The following example uses the COLLECTION_FOLDER property:

```
publishType = 'TO_SHAREPOINT';
properties  = 'SITE_URL, LIST_NAME, COLLECTION_FOLDER, HTTP_USER, HTTP_PASSWORD';
siteUrl     = 'http://www.alphaliteairways.com/sale';
listName    = 'fares';
collectionFolder = 'Fare Lists';
```

```

userName      = 'JohnSmith';
password      = 'secret';

CALL PACKAGE_PUBLISH(packageId, publishType, rc, properties,
    siteUrl, listName, collectionFolder, userName, password);

```

Example 2: Using PACKAGE_PUBLISH with the COLLECTION_URL Property to Publish to SharePoint

The following example uses the COLLECTION_URL property:

```

publishType = 'TO_SHAREPOINT';
properties  = 'COLLECTION_URL, HTTP_USER, HTTP_PASSWORD';
collectionUrl = 'http://www.alphaliteairways.com/sale/fares/Fare Lists';
userName    = 'vicdamone';
password    = 'myway';

CALL PACKAGE_PUBLISH(packageId, publishType, rc,
    properties, collectionUrl, userName, password);

```

Example 3: Using PACKAGE_PUBLISH with the INITIALIZE_SITE Property to Initialize a Site and Publish to SharePoint

Before a SharePoint site can be used, it must be initialized by an administrator. To initialize the site, use the INITIALIZE_SITE property, as follows:

```

publishType = 'TO_SHAREPOINT';
properties  = 'SITE_URL, LIST_NAME, HTTP_USER, HTTP_PASSWORD, INITIALIZE_SITE';
siteUrl     = 'http://www.alphaliteairways.com/';
userName    = 'vicdamone';
password    = 'myway';
initialize  = 'TRUE';

CALL PACKAGE_PUBLISH(packageId, publishType, rc, properties,
    siteUrl, listName, userName, password, initialize);

```

PACKAGE_PUBLISH (Publish Package to Subscribers)

Publishes a package to subscribers who are associated with specified channel

Syntax

```
CALL PACKAGE_PUBLISH(packageId, publishType, rc, properties,
    <propValue1, ...propValueN>, channel);
```

Required Arguments

packageID

identifies the package that is to be published.

Type: Numeric, Input

publishType

indicates how to publish the package. To publish a package to the subscribers of a channel, specify a *publishType* value of TO_SUBSCRIBERS.

Type: Character, Input

rc

receives a return code.

Type: Numeric, Output

properties

identifies a comma-separated list of optional property names. Specify any of the following property names, or specify " to indicate that no properties are to be applied:

- APPLIED_TEXT_VIEWER_NAME
- APPLIED_VIEWER_NAME
- ARCHIVE_NAME
- ARCHIVE_PATH
- CHANNEL_STORE
- COLLECTION_URL
- CORRELATIONID
- ENCODING
- FOLDER_PATH
- FROM
- FTP_PASSWORD
- FTP_USER
- GENERATED_NAME
- HTTP_PASSWORD
- HTTP_PROXY_URL
- HTTP_USER
- IF_EXISTS
- METAPASS
- METAUSER
- PARENT_URL
- PROCESS_VIEWER
- REPLYTO
- SUBJECT
- TARGET_VIEW_MIMETYPE
- TARGET_VIEW_NAME
- TARGET_VIEWER_MIMETYPE
- TARGET_VIEWER_NAME
- TEXT_VIEWER_NAME
- VIEWER_NAME

Type: Character, Input

channel

specifies the name of the channel as it is defined in the SAS Metadata Repository. The channel contains a list of subscribers to whom the package will be published.

Type: Character, Input

Optional Argument

propValue1, ...propValueN

specifies one value for each specified property name. The order of the property values must match the order of the property names in the properties parameter. Valid property values are defined as follows:

APPLIED_TEXT_VIEWER_NAME	specifies a character string that names the rendered package view, which results from the application of the text viewer template to the package for viewing in e-mail. To specify the name of the rendered package view, use either FILEREF: <i>SAS_fileref</i> or FILENAME: <i>external_filename</i> . This property is valid only when the TEXT_VIEWER_NAME property is also specified. By default, the rendered view is created as a temporary file. This property overrides the default, causing the rendered view to be saved permanently to a file.
APPLIED_VIEWER_NAME	specifies a character string that indicates the name of the rendered package view, which results from the application of the HTML viewer template to the package for viewing in e-mail. To specify the name of the rendered package view, use either FILEREF: <i>SAS_fileref</i> or FILENAME: <i>external_filename</i> . This property is valid only when the VIEWER_NAME property is also specified. By default, the rendered view is created as a temporary file. This property overrides the default, causing the rendered view to be saved permanently to a file.
ARCHIVE_NAME	specifies a character string that indicates the name of the archive file.
ARCHIVE_PATH	specifies a character string that indicates the path where the archive should be created.
CHANNEL_STORE	specifies a character string that indicates the SAS Metadata Repository containing the channel and subscriber metadata. If channel definitions and subscriber definitions are maintained in a SAS Metadata Repository, then the syntax for the CHANNEL_STORE property is as follows: SAS-OMA:// <i>hostname[:port]</i> reposname=repositoryName; The <i>hostname</i> is the name of SAS Metadata Server that contains channel information. HOSTNAME must be a DNS name or IP address of a host that is running a SAS Metadata Server. The <i>port</i> is the TCP port of the SAS Metadata Server. If no port is

	specified, then 8561 is used as a default. The reposname is the name of the repository.
COLLECTION_URL	specifies a character string that indicates the URL in which the WebDAV collection is placed. You assign an explicit filename to the collection. Note that when you use COLLECTION_URL, the default behavior is to replace the existing collection at that location.
CORRELATIONID	specifies a binary character string correlator that is used on the package header message.
ENCODING	specifies a character string that indicates the text encoding to use for the message body.
FOLDER_PATH	specifies the folder path for the channel of interest. This value is used to search for channels with specific names that exist in specific folder locations. When a user defines a channel via SAS Management Console, all channels by default exist in the /Channels folder. SAS Management Console allows the user to define multiple folders and subfolders. All FOLDER_PATH properties must start with /Channels and then can identify subfolders if necessary. For example, a channel named "Sales" might be defined in two different folders: /Channels/Reports/US/ or /Channels/Reports/Europe/
FROM	specifies a character string that indicates the sender (or package publisher) of the e-mail message. Note that the FROM field is valid only with the SMTP e-mail interface.
FTP_PASSWORD	indicates the password that is needed to log on to the remote host at which the archive will be stored. Specify this property only when the remote host is secured.
FTP_USER	indicates the user ID that is needed to log on to the remote host at which the archive will be stored. Specify this property only when the remote host is secured.
GENERATED_NAME	returns the name of the package, whether this value was generated by SAS or specified by another property. This property is an output property. If the package is published with a PARENT_URL, and ARCHIVE_PATH is not specified, then the package is published as a folder that contains the contents of the package and not as a .spk file. In this case, the return value for GENERATED_NAME will not be the

HTTP_PASSWORD

name of the archive but the name of the package. For SharePoint, this is a folder name.

indicates the password that is needed to bind to the Web server on which the package is published. Specify this property only when the Web server is secured.

HTTP_PROXY_URL

indicates the URL of the proxy server.

HTTP_USER

indicates the user ID that is needed to bind to the Web server on which the package is published. Specify this property only when the Web server is secured.

IF_EXISTS

specifies one of the following character strings. Use the IF_EXISTS property to control the treatment of same-named collections already existing on the server. "NOREPLACE" indicates that if the package being published contains a collection that already exists on the server, the PUBLISH_PACKAGE call is to return immediately without affecting the contents of the existing collection. "UPDATE" indicates that if the collection already exists on the server, the PUBLISH_PACKAGE call is to update the existing collection by replacing like-named entities and adding newly named entities. If "UPDATE" is specified and both the package to publish and the existing collection have an HTML set (created with INSERT_HTML) with the same NESTED_NAME, then the HTML set in the published package replaces the HTML set in the existing collection. "UPDATEANY" is identical to "UPDATE" except that the PUBLISH_PACKAGE CALL routine can be used to update a collection that SAS did not create. A consequence of using "UPDATEANY" is that SAS will be unable to retrieve the published package. Note that when names are generated automatically for HTML set collections, the publish code ensures that name collisions will not occur.

METAPASS

specifies the password to use when binding to the SAS Metadata Server. If the METAPASS property is not specified on the PACKAGE_PUBLISH CALL routine, then the METAPASS system option, if set, will be used when binding to the SAS Metadata Server.

METAUSER

specifies the user name to use when binding to the SAS Metadata Server. If the METAUSER property is not specified on

	the PACKAGE_PUBLISH CALL routine, then the METAUSER system option, if set, will be used when binding to the SAS Metadata Server.
PARENT_URL	specifies a character string that indicates the URL under which the WebDAV collection is placed. The collection is automatically assigned a unique name.
PROCESS_VIEWER	specifies a character string of "yes" to indicate that the rendered view will be delivered in e-mail. If you specify the PROCESS_VIEWER property with the ARCHIVE_PATH property, then the archive is created but is not sent as an attachment in e-mail. Instead, viewer processing occurs and the rendered view is sent in e-mail.
REPLYTO	specifies a character string that indicates the designated e-mail address to which package recipients might respond. Note that the REPLYTO field is valid only with the SMTP e-mail interface.
SUBJECT	specifies a character string that provides the subject line for the e-mail message.
TARGET_VIEW_MIMETYPE	specifies a character string that indicates the MIME type of the rendered view for delivery to a WebDAV-compliant server. The target view MIME type overrides the default view MIME type, which is automatically inferred from the viewer. Typical MIME types are HTML (.htm) and plain text (.txt) files. If this field remains blank, then the viewer filename extension is used to locate the MIME type in the appropriate registry. Windows hosts use the Windows Registry; other hosts use the SAS Registry.
TARGET_VIEW_NAME	specifies a character string that indicates the name of the rendered view for delivery to a WebDAV-compliant server. The specified target view name overrides the default name, which is index.html.
TARGET_VIEWER_MIMETYPE	see TARGET_VIEW_MIMETYPE.
TARGET_VIEWER_NAME	see TARGET_VIEW_NAME.
TEXT_VIEWER_NAME	specifies a character string that indicates the name of a text viewer template that formats package content for viewing in e-mail by using either FILEREF:SAS_fileref or FILENAME:external_filename . A text viewer template might be necessary if the

destination e-mail program does not support the HTML MIME type.

VIEWER_NAME

specifies a character string that indicates the name of the HTML viewer template to be applied when publishing e-mail by using either **FILEREF**:*SAS_fileref* or **FILENAME**:*external_filename*.

Type: Character or Numeric, Input or Output

Details

Overview of Publishing to a Channel

When a package is published to a channel, the package is published to each subscriber of the channel. Each subscriber's entry contains an attribute that specifies the publishing transport method: e-mail, message queue, WebDAV-Compliant server, or none.

You can use the Publishing Framework plug-in for SAS Management Console to define and manage channels and subscribers. This plug-in also enables subscribers to define filters that determine what packages are published to them. For more information about filters, see [“Filtering Packages and Package Entries” on page 44](#).

When publishing to subscribers, the PACKAGE_PUBLISH CALL routine ensures that the package is published to each subscriber only once, thus eliminating any duplication. When the delivery transport is a message queue, the queue name is used as the key to enforce uniqueness. When the delivery transport is WebDAV, the collection URL is used as the key to enforce uniqueness. A parent URL is always unique because the WebDAV transport always creates a unique collection name for parent URLs. When the delivery transport is e-mail, the subscriber's e-mail address is used as the key to enforce uniqueness.

In order to publish to a channel, the publisher must have Write permission. For information about permissions by task for working with publishing channels, see *SAS Intelligence Platform: Security Administration Guide*.

Note: You can use the package cleanup utility to delete packages that have been published to a channel. This utility is part of the Web Infrastructure Platform. For more information, see the Javadoc.

Default Properties

For channel subscribers who specify an e-mail delivery transport, the default action is to publish the e-mail message in plain text format. Only inserted references are published to the e-mail subscriber. For details, see the [INSERT_REF CALL routine on page 64](#).

The package description field precedes the reference value in the e-mail message. All other inserted entries are ignored. For channel subscribers who specify a queue delivery transport, the default action is to publish all inserted entries to the queue.

Viewer Properties

To override the default e-mail behavior, you can specify the VIEWER_NAME or TEXT_VIEWER_NAME property on the PACKAGE_PUBLISH CALL routine. The specified viewer is used to create the content of the e-mail message and to apply substitutions. If you specify VIEWER_NAME, then the e-mail message is published in HTML format. If you specify TEXT_VIEWER_NAME, then the e-mail message is published in text format. Only the package information that is rendered by the viewer is published.

E-mail subscribers can configure the format in which they want to receive the e-mail, either in HTML or text format. The default behavior is that the message is published in HTML format. If the e-mail subscriber specifies text format, then the viewer is not used, and the subscriber receives reference entries only. For more information about the viewer facility, see the chapter about [viewer processing on page 15](#).

The VIEWER_NAME and TEXT_VIEWER_NAME properties override the default behavior for WebDAV subscribers as well. If you specify VIEWER_NAME, then the view is rendered in HTML format. If you specify TEXT_VIEWER_NAME, then the view is rendered in text format. The specified viewer is used to create a rendered view that is named index.html. To override the default name that is assigned the rendered view, use the APPLIED_VIEWER_NAME or APPLIED_TEXT_VIEWER_NAME, as appropriate, to specify a filename for the rendered view.

The VIEWER_NAME and TEXT_VIEWER_NAME properties are ignored by the queue and archive transports.

If you specify the VIEWER_NAME or TEXT_VIEWER_NAME property with the COLLECTION_URL or PARENT_URL property, then the e-mail message contains a reference to a URL. The specified viewer is used to create a rendered view that is named index.html. To override the default name that is assigned to the rendered view, use the TARGET_VIEW_NAME or TARGET_VIEW_MIMETYPE, as appropriate, to specify a filename for the rendered view. The package is published to a WebDAV-compliant server. For channel subscribers who specify an e-mail delivery transport, the default action is to notify subscribers of the URL of the published package. For channel subscribers who specify a message queue delivery transport, no notification is given to indicate the package's availability on the Web.

Archive Path Property

When publishing to subscribers, the ARCHIVE_PATH property indicates that the package is to be persisted to an archive using the specified archive path. The ARCHIVE_PATH property identifies where the archive is to be persisted. This property can be a physical pathname, an FTP URL, or an HTTP URL. The channel metadata can be defined with a default persistent store. A persistent store identifies a default transport that is used to persist the package before publishing to the channel subscribers. The persistent store can be defined as a default archive path. If you specify a blank value for the ARCHIVE_PATH property, then the channel's default archive path is used to determine where the archive is to be persisted.

For channel subscribers who specify e-mail as the delivery transport, the created archive is included as an attachment to the e-mail message. If you specify the PROCESS_VIEWER property along with the ARCHIVE_PATH property, then the archive is created but is not sent as an attachment in e-mail. Instead, viewer processing occurs and the rendered view is sent in e-mail. For channel subscribers who specify a queue delivery transport, the created archive is published to the queue. For channel subscribers who specify a WebDAV delivery transport, the archive is published as a binary package to the WebDAV server.

If the ARCHIVE_PATH property is specified with a blank value, then the channel's default archive path metadata is used to determine where the archive is to be persisted. The name of the archive is automatically generated and the archive metadata is then cataloged in the channel metadata. For details about how to define a channel's default archive, see the help in the Publishing Framework plug-in for SAS Management Console.

If the ARCHIVE_PATH is an HTTP URL, then the URL identifies the HTTP server to use when persisting the archive. If it is a secured server, then you must specify the HTTP_USER and HTTP_PASSWORD properties. Specifying the HTTP_PROXY_URL

property is optional. If the ARCHIVE_PATH is an FTP URL, then the URL identifies the FTP server to use when persisting the archive. If it is a secured host, then you must specify the FTP_USER and FTP_PASSWORD properties.

Note: If you specify both the ARCHIVE_PATH and either the VIEWER_NAME or TEXT_VIEWER_NAME properties, then the viewer property is ignored.

Note: In order to create an archive under the z/OS operating environment, the z/OS environment must support UNIX System Services directories.

WebDAV Properties

The channel metadata can be defined with a default persistent store. A persistent store identifies a default transport that is used to persist the package before publishing to the channel subscribers. The persistent store can be defined as a default WebDAV server.

If the COLLECTION_URL or PARENT_URL property value is blank, then the package is published to the default WebDAV server configured in the channel metadata. If you specify a non-blank COLLECTION_URL or PARENT_URL property value, then the specified URL is used as the persisted location. When a non-blank value is specified for COLLECTION_URL, the URL identifies the full path and the explicit collection name. When a non-blank value is specified for PARENT_URL, the URL identifies the full path and a unique name is assigned to the collection automatically.

Channel subscribers who specify an e-mail delivery transport are notified about the availability of the new collection. The e-mail message contains a reference to the value of the COLLECTION_URL or PARENT_URL property, which specifies the URL to which the package is published. For channel subscribers who specify a message queue delivery transport, no notification is given to announce the collection's availability.

The COLLECTION_URL (or PARENT_URL) property and the ARCHIVE_PATH property are mutually exclusive.

When publishing to a WebDAV-compliant server with the COLLECTION_URL or PARENT_URL properties, you can specify the following WebDAV properties: HTTP_PASSWORD, HTTP_PROXY_URL, HTTP_USER, IF_EXISTS, TARGET_VIEW_MIMETYPE, TARGET_VIEW_NAME, and VIEWER_NAME (or TEXT_VIEWER_NAME).

WebDAV publishing uses the following file extensions for each item type:

Table 5.5 File Extensions for Item Types

Item Type	File Extension
CATALOG	.sac
DATA	.sad
Mddb	.sam
REFERENCE	.ref
VIEW	.sav

Examples

Example 1: Using PACKAGE_PUBLISH to Publish to Subscribers

The following example publishes the specified package to all subscribers of the Report channel. The SAS Metadata Server on ALPAIR03 is searched for the stored channel and subscriber information. The SAS Metadata Server is using port 4059 and the repository to use is MyRepos.

```
channelStore =
  "SAS-OMA://alpair03.sys.com:4059";
channelName = "Report";
prop = "channel_store,metauser,metapass";
user = "myUserName";
password = "myPassword";
CALL PACKAGE_PUBLISH(packageId, "TO_SUBSCRIBERS", rc, prop,
  channelStore, user, password, channelName);
```

Example 2: Using PACKAGE_PUBLISH to Publish to Subscribers with the Subject Property

The following example publishes the package to all subscribers of the HR channel. The subject property is specified so that all e-mail subscribers will receive the message with the specified subject.

```
publishType = "TO_SUBSCRIBERS";
storeInfo =
  "SAS-OMA://alpair03.sys.com:8561";
channel = 'HR';
property = "SUBJECT, CHANNEL_STORE, METAUSER, METAPASS";
subject = "Weekly HR Updates:"
user = "myUserName";
password = "myPassword";
CALL PACKAGE_PUBLISH(packageId, "TO_SUBSCRIBERS",
  rc, property, subject, storeInfo, user, password, channel);
```

PACKAGE_PUBLISH (Publish Package to a WebDAV-Compliant Server)

Publishes a package to a WebDAV-compliant server

Syntax

```
CALL PACKAGE_PUBLISH(packageId, publishType, rc, properties,
  <propValue1, ...propValueN> );
```

Required Arguments

packageID

identifies the package that is to be published.

Type: Numeric, Input

publishType

indicates how to publish the package. To publish the package using the WebDAV transport, specify a *publishType* of TO_WEBDAV.

Type: Character, Input

rc

receives a return code.

Type: Numeric, Output

properties

identifies a comma-separated list of optional property names. Specify any of the following property names, or specify " to indicate that no properties are to be applied:

- APPLIED_TEXT_VIEWER_NAME
- APPLIED_VIEWER_NAME
- ARCHIVE_FULLPATH
- ARCHIVE_NAME
- ARCHIVE_PATH
- COLLECTION_URL
- GENERATED_NAME
- HTTP_PASSWORD
- HTTP_PROXY_URL
- HTTP_TOKENAUTH
- HTTP_USER
- IF_EXISTS
- PARENT_URL
- TARGET_VIEW_MIMETYPE
- TARGET_VIEW_NAME
- TARGET_VIEWER_MIMETYPE
- TARGET_VIEWER_NAME
- TEXT_VIEWER_NAME
- VIEWER_NAME

Type: Character, Input

Optional Argument

propValue1, ...propValueN

specifies one value for each specified property name. The order of the property values must match the order of the property names in the properties parameter. Valid property values are defined as follows:

APPLIED_TEXT_VIEWER_NAME	specifies a character string that names the rendered package view, which results from the application of the text viewer template. To specify the name of the rendered package view, use either FILEREF : <i>SAS_fileref</i> or FILENAME : <i>external_filename</i> . This property is valid only when the TEXT_VIEWER_NAME property is
--------------------------	---

	also specified. By default, the rendered view is created as a temporary file. This property overrides the default, causing the rendered view to be saved permanently to a file.
APPLIED_VIEWER_NAME	specifies a character string that indicates the name of the rendered package view, which results from the application of the HTML viewer template. To specify the name of the rendered package view, use either FILEREF : <i>SAS_fileref</i> or FILENAME : <i>external_filename</i> . This property is valid only when the VIEWER_NAME property is also specified. By default, the rendered view is created as a temporary file. This property overrides the default, causing the rendered view to be saved permanently to a file.
ARCHIVE_FULLPATH	returns the complete URL path of the published archive on the server. The URL path includes the name of the archive, as specified by ARCHIVE_NAME or the generated name if ARCHIVE_NAME is not specified. This output property is returned only if ARCHIVE_PATH is specified.
ARCHIVE_NAME	specifies a character string that indicates the name of the archive file.
ARCHIVE_PATH	specifies a character string that indicates the path where the archive should be created.
COLLECTION_URL	specifies a character string that indicates the URL in which the WebDAV collection is placed. You assign an explicit filename to the collection. Note that when you use COLLECTION_URL, the default behavior is to replace the existing collection at that location.
GENERATED_NAME	returns the name of the package, whether this value was generated by SAS or specified by another property. This property is an output property. If the package is published with a PARENT_URL, and ARCHIVE_PATH is not specified, then the package is published as a folder that contains the contents of the package and not as a .spk file. In this case, the return value for GENERATED_NAME will not be the name of the archive but the name of

HTTP_PASSWORD	indicates the password that is needed to bind to the Web server on which the package is published. Specify this property only when the Web server is secured.
HTTP_PROXY_URL	indicates the URL of the proxy server.
HTTP_TOKENAUTH	enables token authentication to a SAS Content Server. Value must be set to TRUE , FALSE , YES , or NO . Do not specify values for HTTP_USER or HTTP_PASSWORD if you are using this property.
HTTP_USER	indicates the user ID that is needed to bind to the Web server on which the package is published. Specify this property only when the Web server is secured.
IF_EXISTS	specifies one of the following character strings. Use the IF_EXISTS property to control the treatment of same-named collections already existing on the server. "NOREPLACE" indicates that if the package being published contains a collection that already exists on the server, the PUBLISH_PACKAGE call is to return immediately without affecting the contents of the existing collection. "UPDATE" indicates that if the collection already exists on the server, the PUBLISH_PACKAGE call is to update the existing collection by replacing like-named entities and adding newly named entities. If "UPDATE" is specified and both the package to publish and the existing collection have an HTML set (created with INSERT_HTML) with the same NESTED_NAME, then the HTML set in the published package replaces the HTML set in the existing collection. "UPDATEANY" is identical to "UPDATE" except that the PUBLISH_PACKAGE CALL routine can be used to update a collection that SAS did not create. A consequence of using "UPDATEANY" is that SAS will be unable to retrieve the published package. Note that when names are generated automatically for HTML set collections, the publish code ensures that name collisions will not occur.

PARENT_URL	specifies a character string that indicates the URL under which the WebDAV collection is placed. The collection is automatically assigned a unique name.
TARGET_VIEW_MIMETYPE	specifies a character string that indicates the MIME type of the rendered view for delivery to a WebDAV-compliant server. The target view MIME type overrides the default view MIME type, which is automatically inferred from the viewer. Typical MIME types are HTML (.htm) and plain text (.txt) files. If this field remains blank, then the viewer filename extension is used to locate the MIME type in the appropriate registry. Windows hosts use the Windows Registry; other hosts use the SAS Registry.
TARGET_VIEW_NAME	specifies a character string that indicates the name of the rendered view for delivery to a WebDAV-compliant server. The specified target view name overrides the default name, which is index.html.
TEXT_VIEWER_NAME	specifies a character string that indicates the name of a text viewer template that formats package content for viewing in e-mail by using either FILEREF : <i>SAS_filename</i> or FILENAME : <i>external_filename</i> . A text viewer template might be necessary if the destination e-mail program does not support the HTML MIME type.
VIEWER_NAME	specifies a character string that indicates the name of the HTML viewer template to be applied when publishing e-mail by using either FILEREF : <i>SAS_filename</i> or FILENAME : <i>external_filename</i> .

Type: Character or Numeric, Input or Output

Details

Default Behavior

Publishing with a *publishType* of TO_WEBDAV publishes a package to a specified URL on a WebDAV-compliant server. Starting with SAS 9.2, the HTTPS protocol is supported when publishing to a WebDAV server. WebDAV servers enable distributed authoring and versioning, which enables collaborative development of Web files on remote servers.

The WebDAV transport stores package entries as members of a collection.

If you specify the COLLECTION_URL property, then the package is published to the specified URL on a WebDAV-compliant Web server. When you use

COLLECTION_URL, the default behavior is to replace the existing collection and its nested directories at that location. If you do not want to replace an existing collection and its nested directories, then you must use the IF_EXISTS property. An example of a collection URL is

```
http://www.host.com/AlphaliteAirways/revenue/quarter1
```

The collection is named quarter1.

The PARENT_URL property is similar to the COLLECTION_URL property except that it specifies the location under which the new WebDAV collection is to be placed. The PUBLISH_PACKAGE CALL routine generates a unique name for the new collection. The unique name is limited to eight characters with the first character as an *s*. An example of a parent URL directory location is `http://www.host.com/AlphaliteAirways/revenue`. An example of a collection name that is automatically generated might be `s9811239`.

The specifications of the COLLECTION_URL property and the PARENT_URL property are mutually exclusive.

To announce the availability of new WebDAV collections on WebDAV-compliant servers, use a *publishType* of TO_SUBSCRIBERS or TO_EMAIL.

WebDAV publishing uses the following file extensions for each item type:

Table 5.6 File Extensions for Item Types

Item Type	File Extension
ARCHIVE	.spk
CATALOG	.sac
COMMA-SEPARATED VALUES	.csv
DATA	.sad
MDDDB	.sam
REFERENCE	.ref
VIEW	.sav

Viewer Properties

If you specify the VIEWER_NAME property with the COLLECTION_URL or PARENT_URL property, then the view is rendered in HTML format. If you specify the TEXT_VIEWER_NAME with the COLLECTION_URL or PARENT_URL properties, then the view is rendered in text format.

The specified viewer is used to create a rendered view that is named `index.html`. To override the default name that is assigned to the rendered view, use the APPLIED_VIEWER_NAME or APPLIED_TEXT_VIEWER_NAME, as appropriate, to specify a filename for the rendered view.

Archive Path Properties

If you specify the ARCHIVE_PATH property, then an archive is created and published as a binary package on a WebDAV-compliant server. All entries that are inserted into

the package are published as an archive. If you specify a value for ARCHIVE_PATH, then the created archive is stored at the designated location. To create a temporary archive that is deleted after the package is published, specify an ARCHIVE_PATH value of "" or "tempfile".

For more details about how to use the archive properties, see [“PACKAGE_PUBLISH \(Publish Package to Archive\)” on page 71](#).

Note: In order to create an archive under the z/OS operating environment, the z/OS environment must support UNIX System Services directories.

When publishing a binary package with the WEBDAV transport, you can specify the following archive properties: ARCHIVE_NAME, ARCHIVE_PATH, or ARCHIVE_FULLPATH.

Applying a Name/Value Pair to a Package and a Package Item

When publishing to a WebDAV-compliant server, optional name/value pairs are transmitted to the WebDAV server in XML format. XML format requires that the name portion of the name/value pair specification follow these conventions:

- It must begin with an alphabetic character or an underscore.
- It can contain only alphabetic characters, numeric characters, and these special characters: . (period), - (hyphen), and _ (underscore).

If a namespace is associated with the name portion of a name/value pair, then the name can also include a colon (:). Name/value pairs not explicitly associated with a namespace might not be retained by the WebDAV server. For details about the NAMESPACE property or about specifying the *nameValue* parameter for an entire package, see [PACKAGE_BEGIN on page 68](#).

For details about specifying the *nameValue* parameter for a single package item, see the applicable INSERT_ *item* CALL routine, where *item* can be any of the following:

- [CATALOG](#)
- [DATASET](#)
- [FILE](#)
- [HTML](#)
- [MDDDB](#)
- [PROC SQL VIEW](#)
- [REFERENCE](#)
- [VIEWER](#)

Examples

Example 1: Using PACKAGE_PUBLISH to Publish to a WebDAV Server

The following example uses the HTTPS protocol when publishing to the WebDAV server:

```
rc = 0;
publishType = "TO_WEBDAV";
http_user="vicdamone";
http_password="myway";
properties="COLLECTION_URL, http_user, http_password";
```

```
cUrl = "https://www.alpair.web/NightlyMaintReport";
CALL PACKAGE_PUBLISH(packageId, publishType,
    rc, properties, cUrl, http_user, http_password);
```

Example 2: Using PACKAGE_PUBLISH to Publish to a Specified Proxy Server

The following example publishes a package to a URL via the specified proxy server by using the specified credentials:

```
rc = 0;
publishType = "TO_WEBDAV";
properties="COLLECTION_URL,HTTP_PROXY_URL,
    IF_EXISTS,HTTP_USER,HTTP_PASSWORD";
cUrl = "http://www.alpair.secureweb/NightlyMaintReport";
pUrl = "http://www.alpair.proxy:8000/";
exists = "update";
user = "JohnSmith";
password = "secret";
CALL PACKAGE_PUBLISH(packageId, publishType, rc, properties,
    cUrl, pUrl, exists, user, password);
```

Example 3: Using PACKAGE_PUBLISH to Publish a Collection URL on a WebDAV-Compliant Server to a WebDAV Server

The following example uses the e-mail transport to publish a collection URL on a WebDAV-compliant server. The HTTP user ID and password enable the publisher to bind to the secured HTTP server. All e-mail recipients who are members of the mail list receive the e-mail announcement that the best rates are accessible at the specified URL.

```
publishType = "TO_EMAIL";
properties="COLLECTION_URL, SUBJECT,
    HTTP_USER, HTTP_PASSWORD";
collurl="http://www.alphaliteairways/fares/discount.html";
subj="Announcing Best Rates Yet";
http_user="vicdamone";
http_password="myway";
Addr = "admins-l@alphaliteair05";
CALL PACKAGE_PUBLISH(packageId, publishType, rc, properties,
    collurl, subj, http_user, http_password, Addr);
```

Example 4: Using PACKAGE_PUBLISH to Publish to a WebDAV Server with the ARCHIVE_PATH Property

The following example uses the ARCHIVE_PATH property to publish a binary package to the WebDAV-compliant server. The archive path is specified as "tempfile" so that the locally created archive file will be deleted once it has been published to the WebDAV server.

```
publishType = "TO_WEBDAV";
properties="COLLECTION_URL, ARCHIVE_PATH";
cUrl = "http://www.alpair.secureweb/Reports";
apath = "tempfile";
CALL PACKAGE_PUBLISH(packageId, publishType, rc,
    properties, cUrl, apath);
```

COMPANION_NEXT

Retrieves the next companion HTML file in the ODS HTML set

Syntax

```
CALL COMPANION_NEXT(entryId, path, filename, url, rc
    <, properties, propValue1, ...propValueN> );
```

Required Arguments

entryId

identifies the companion HTML file entry.

Type: Numeric, Input

path

specifies the full path of the location that will receive the retrieved file.

Type: Character, Input

filename

returns the name of the new file.

Type: Character, Output

url

returns the URL of the companion file.

Type: Character, Output

rc

receives a return code.

Type: Numeric, Output

Optional Arguments

properties

identifies a comma-separated list of optional property names. Valid property names are as follows:

- ENCODING
- MIMETYPE

Type: Character, Input

propValue1, ...propValueN

specifies one value for each specified property name. The order of the property values must match the order of the property names in the *properties* parameter. Valid property values are defined as follows:

ENCODING	input character string that specifies the target encoding for the companion file. The companion file is translated into the specified encoding. An example of a target encoding value is ISO-8859-1.
MIMETYPE	character output parameter that identifies the MIME type of the companion file. The MIME type is returned in the MIMETYPE variable. The publisher of the companion file can set the user-

specified MIME type after the companion file is published. If the publisher does not specify the MIME type, then the returned value is blank.

Type: Character, Input

Details

The publisher can choose to publish any combination of the HTML files. Included in the set of published files can be any number of additional HTML files or companion files.

The **filename** and **url** parameters are character variables that are updated by the CALL routine. Because they are updated, they must be initialized with a length large enough to contain the name of the file or the URL that is being returned. If not, the returned value will be truncated and a warning will be printed indicating that one or more parameters were truncated. When called from within the DATA step, use the LENGTH statement to define the length of the variable. When called from within a macro, initialize the variable to some value so that it will have an appropriate length.

For details about how HTML files are published and how the optional encoding property can be used to provide encoding information to package recipients, see [“Publish and Retrieve Encoding Behavior” on page 43](#).

Examples

Example 1: Using COMPANION_NEXT

The following example retrieves an HTML file and then retrieves the next companion HTML file in the set.

```
data _null_;
length contents $64 frame $64 pages $64
      body $64 contentsUrl $256 frameUrl $256
      PagesUrl $256 bodyUrl $256;

path = '/finance/accounting/doc';
CALL RETRIEVE_HTML(entryId, path, body, bodyUrl, frame,
      frameUrl, contents, contentsUrl, pages, pagesUrl, rc);

CALL COMPANION_NEXT(entryId, path, fname, url, rc);
```

Example 2: Using COMPANION_NEXT with Specified MIME Type

The following example retrieves an HTML file and then retrieves the next companion HTML file in the set. If the publisher specifies a MIME type when publishing a package, then the optional MIMETYPE property is specified in order for its MIME type to be returned. The MIME type will be returned in the mime variable.

```
data _null_;
length contents $64 frame $64 pages $64 body $64
      contentsUrl $256 frameUrl $256 PagesUrl $256
      bodyUrl $256 mime $64;

path = '/finance/accounting/doc';
CALL RETRIEVE_HTML(entryId, path, body, bodyUrl, frame,
      frameUrl, contents, contentsUrl, pages, pagesUrl, rc);

properties="MIMETYPE";
```

```
CALL COMPANION_NEXT(entryId, path, fname,
    url, rc, properties, mime);
```

ENTRY_FIRST

Returns header information for the first entry in a package

Syntax

```
CALL ENTRY_FIRST(packageId, entryId, entryType, userSpecString, desc, nameValue,
    rc<, properties, propValue1, ...propValueN> );
```

Required Arguments

packageId

identifies the package.

Type: Numeric, Input

entryId

returns the identifier of the entry.

Type: Numeric, Output

entryType

returns the type of the entry. Available types include the following:

- BINARY
- CATALOG
- DATASET
- HTML
- MDDb
- NESTED_PACKAGE
- REFERENCE
- SQLVIEW
- TEXT
- VIEWER

Type: Character, Output

userSpecString

returns a string from the specified entry. For string content, see the Details section.

Type: Character, Output

desc

returns the entry description from the specified entry.

Type: Character, Output

nameValue

returns the name/value pairs assigned to the specified entry. Name/value pairs are site-specific; they are used for the purpose of [filtering](#).

Type: Character, Output

rc

receives a return code.

Type: Numeric, Output**Optional Arguments*****properties***

identifies a comma-separated list of optional property names. Valid property names are as follows:

- FILENAME

Type: Character, Input***propValue1, ...propValueN***

returns one value for each specified property name. Valid property names are supported as follows:

FILENAME output character string variable that returns the name of the file (as it exists in the package).

Type: Character, Output**Details**

The header information returned by this CALL routine identifies the type of the entry and provides descriptive information.

The ENTRY_FIRST CALL routine repositions the entry cursor to the start of the list of entries. When the packages are retrieved by way of the [RETRIEVE_PACKAGE](#) CALL routine, the entry cursor is positioned at the start of the entry list by default. As a consequence, the ENTRY_FIRST CALL routine does not have to be called before the [ENTRY_NEXT](#) CALL routine.

The **userSpecString** parameter is returned to provide further content information about the entry. The value returned is the value that was provided by the publisher at insert time. At this time, only file entries can return a value for this parameter. All other entry types return a blank value. For file entries, this field is the user-specified MIME type.

Example: Using ENTRY_FIRST

The following example returns header information for the first entry in the package.

```
CALL ENTRY_FIRST(packageId, entryid, type,
  uSpec, desc, nv, rc);
```

ENTRY_NEXTReturns header information from the next entry in a package

Syntax

```
CALL ENTRY_NEXT(packageId, entryId, entryType, userSpecString, desc, nameValue,
  rc<, properties, propValue1, ...propValueN> );
```


Required Arguments

packageId

identifies the package.

Type: Numeric, Input

entryId

returns the identifier of the entry.

Type: Numeric, Output

entryType

returns the type of the entry. Available types include the following:

- BINARY
- CATALOG
- DATASET
- HTML
- MDDB
- NESTED_PACKAGE
- REFERENCE
- SQLVIEW
- TEXT
- VIEWER

Type: Character, Output

userSpecString

returns a string from the specified entry. For string content, see the Details section.

Type: Character, Output

desc

returns the entry description from the specified entry.

Type: Character, Output

nameValue

returns the name/value pairs assigned to the specified entry. Name/value pairs are site-specific; they are used for the purpose of [filtering](#).

Type: Character, Output

rc

receives a return code.

Type: Numeric, Output

Optional Arguments

properties

identifies a comma-separated list of optional property names. Valid property names are as follows:

- FILENAME

Type: Character, Input

propValue1, ...propValueN

returns one value for each specified property name. Valid property names are supported as follows:

FILENAME output character string variable that returns the name of the file (as it exists in the package).

Type: Character, Output

Details

The header information returned by this CALL routine identifies the type of the entry and provides descriptive information.

The **userSpecString** parameter provides content information about the entry. The value returned is the value that was provided by the publisher when the entry was inserted in the package. For this release, only file entries can return a value for this parameter. All other entry types return a blank value. For file entries, this field is the user-specified MIME type.

When a package is retrieved, the entry cursor is positioned at the start of the entry list by default. As a consequence, the [ENTRY_FIRST](#) CALL routine does not have to be called before [ENTRY_NEXT](#) CALL routine. The [ENTRY_FIRST](#) CALL routine can be used at a later time in order to move the entry cursor back to the start of the entry list.

Example: Using ENTRY_NEXT

The following example positions the cursor at the start of an entry list.

```
CALL ENTRY_NEXT(packageId, entryid, type,
               uSpec, desc, nv, rc);
```

PACKAGE_DESTROY

Deletes a package

Syntax

```
CALL PACKAGE_DESTROY(packageId, rc);
```

Required Arguments

packageId

identifies the package to be deleted.

Type: Numeric, Input

rc

receives a return code.

Type: Numeric, Output

Details

If the queue transport is used, then the package is removed from the queue, along with all messages that are associated with the package. If the package contains nested packages, then all entries that are contained within the nested packages are also removed from the queue. If the archive transport is used, then the archive is deleted. If the WebDAV transport is used, then the package and its contents are deleted from the

WebDAV server. If the SharePoint transport is used, then the package and its contents are removed from the SharePoint server.

The PACKAGE_DESTROY CALL routine does not support package identifiers that represent nested packages, which are returned by way of the RETRIEVE_NESTED CALL routine. The PACKAGE_DESTROY CALL routine supports only top-level package identifiers, which are returned by [PACKAGE_FIRST](#) and [PACKAGE_NEXT](#).

Example: Using PACKAGE_DESTROY

The following example removes a package from a queue.

```
rc=0;
CALL PACKAGE_DESTROY(packageId, rc);
```

PACKAGE_FIRST

Returns the header information for the first package in the package list

Syntax

```
CALL PACKAGE_FIRST(pkgListId, packageId, numEntries, desc, dateTime, nameValue,
    channel, rc<, properties, propValue1, ...propValueN> );
```

Required Arguments

pkgListId

identifies the list of retrieved packages.

Type: Numeric, Output

packageId

identifies the retrieved package.

Type: Numeric, Output

numEntries

returns the number of entries in the package.

Type: Numeric, Output

desc

returns a description of the package.

Type: Character, Output

dateTime

returns the date and time that the package was published, in GMT format.

Type: Numeric, Output

nameValue

returns the name/value pairs assigned to the package. Name/value pairs are site-specific; they are used for the purpose of [filtering](#).

Type: Character, Output

channel

returns the name of a channel to which the package was published.

Type: Character, Output

rc

receives a return code.

Type: Numeric, Output**Optional Arguments*****properties***

identifies a comma-separated list of optional property names to be returned from the package. Valid property names are as follows:

- ABSTRACT
- EXPIRATION_DATETIME

Type: Character, Input***propValue1, ...propValueN***returns one value for each specified property. The order of the values matches the order of the property names in the *properties* parameter. Valid property values are defined as follows:

ABSTRACT	character string variable, if specified, is returned to the ABSTRACT variable.
EXPIRATION_DATETIME	numeric variable, if specified, is returned as the package expiration date-and-time stamp to the EXPIRATION_DATETIME variable. The date-and-time stamp is in GMT format.

Type: Character or Numeric, Output**Examples*****Example 1: Using PACKAGE_FIRST***

The following example opens the JSMITH queue, retrieves the descriptive header information for all packages, and then returns the header information for the first package.

```

plist=0;
qname = "MQSERIES://LOCAL:JSMITH";
rc=0;
total=0;
nameValue='';
CALL RETRIEVE_PACKAGE(plist, "FROM_QUEUE",
    qname, total, rc);

packageId = 0;
desc='';
num=0;
dt=0;
nv='';
ch='';
rc=0;
CALL PACKAGE_FIRST(plist, packageId,
    num, desc, dt, nv, ch, rc);

```

Example 2: Using PACKAGE_FIRST with Properties

The following example demonstrates the use of properties.

```

plist=0;
qname = "MQSERIES://LOCAL:JSMITH";
rc=0;
total=0;
nameValue='';
CALL RETRIEVE_PACKAGE(list, "FROM_QUEUE",
    qname, total, rc);

packageId = 0;
desc='';
num=0;
exp=0;
abstract='';
dt=0;
nv='';
ch='';
rc=0;
props='ABSTRACT, EXPIRATION_DATETIME';
CALL PACKAGE_FIRST(plist, packageId, num, desc,
    dt, nv, ch, rc, props, abstract, exp);

```

PACKAGE_NEXT

Returns the header information for the next package in the package list

Syntax

CALL PACKAGE_NEXT(pkgListId, packageId, numEntries, desc, dateTime, nameValue, channel, rc<, properties, propValue1, ...propValueN>);

Required Arguments**pkgListId**

identifies the list of retrieved packages.

Type: Numeric, Input

packageId

returns the name of the retrieved package.

Type: Numeric, Output

numEntries

returns the total number of entries in the package.

Type: Numeric, Output

desc

describes the package.

Type: Character, Output

dateTime

returns the date and time value that the package was published, in GMT format.

Type: Numeric, Output

nameValue

returns the name/value pairs assigned to the package. Name/value pairs are site-specific; they are used for the purpose of [filtering](#).

Type: Character, Output

channel

returns the name of the channel to which the package was published.

Type: Character, Output

rc

receives a return code.

Type: Numeric, Output

Optional Arguments***properties***

identifies a comma-separated list of optional property names to be returned from the package. Valid property names are as follows:

- ABSTRACT
- EXPIRATION_DATETIME

Type: Character, Input

propValue1, ...propValueN

returns one value for each specified property. The order of the values matches the order of the property names in the *properties* parameter. Valid property values are defined as follows:

ABSTRACT	character string variable, if specified, is returned to the ABSTRACT variable.
EXPIRATION_DATETIME	numeric variable, if specified, is returned as the package expiration date/time stamp to the EXPIRATION_DATETIME variable. The date/time stamp is in GMT format.

Type: Character or Numeric, Output

Examples***Example 1: Using PACKAGE_NEXT***

The following example returns the header information for the next package that is associated with the list of packages named PLIST.

```
packageId = 0;
desc='';
num=0;
exp=0;
dt=0;
nv='';
ch='';
rc=0;
CALL PACKAGE_NEXT(plist, packageId,
    num, desc, dt, nv, ch, rc);
```

Example 2: Using PACKAGE_NEXT with the ABSTRACT Property

The following example uses the ABSTRACT property so that the abstract value is returned in the **abs** variable.

```

packageId = 0;
desc='';
num=0;
exp=0;
dt=0;
nv='';
ch='';
abs='';
props="ABSTRACT";
rc=0;
CALL PACKAGE_NEXT(plist, packageId, num,
  desc, dt, nv, ch, rc, props, abs);

```

PACKAGE_TERM

Frees all resources associated with the package list identifier

Syntax

```
CALL PACKAGE_TERM(pkgListId, rc);
```

Required Arguments***pkgListId***

identifies the list of packages.

Type: Numeric, Input

rc

receives a return code.

Type: Numeric, Output

Details

This CALL routine is used when publishing a package.

Example: Using PACKAGE_TERM

The following example frees all resources that are associated with **pkgListId**.

```
CALL PACKAGE_TERM(pkgListId, rc);
```

RETRIEVE_CATALOG

Retrieves a catalog from a package

Syntax

CALL RETRIEVE_CATALOG(*entryId*, *libname*, *memname*, *rc*);

Required Arguments

entryId

identifies the catalog entry.

Type: Numeric, Input

libname

specifies the SAS library that will contain the retrieved catalog.

Type: Character, Input

memname

names the retrieved catalog.

Type: Character, Input

rc

receives a return code.

Type: Numeric, Output

Details

If the **memname** parameter is blank, then the RETRIEVE_CATALOG CALL routine creates the catalog by using the original member name as it was defined at publish time.

Example: Using RETRIEVE_CATALOG

The following example retrieves a catalog from the package and creates the catalog WORK.TMPCAT.

```
lib = 'work';
mem = 'tmpcat';
CALL RETRIEVE_CATALOG(entryId, lib, mem, rc);
```

RETRIEVE_DATASET

This CALL routine retrieves a data set entry from a package

Syntax

CALL RETRIEVE_DATASET(*entryId*, *libname*, *memname*, *rc*
<, *properties*, *propValue1*, ...*propValueN*>);

Required Arguments

entryId

identifies the data set entry.

Type: Numeric, Input

libname

specifies the SAS library that will contain the retrieved data set.

Type: Character, Input

memname

names the retrieved data set.

Type: Character, Input

rc

receives a return code.

Type: Numeric, Output

Optional Arguments***properties***

identifies a comma-separated list of optional property names. Valid property names are as follows:

- DATASET_OPTIONS
- CSV_SEPARATOR
- CSV_FLAG

Type: Character, Input

propValue1, ...propValueN

specifies one value for each specified property name. The order of the property values must match the order of the property names in the *properties* parameter. Valid property values are defined as follows:

DATASET_OPTIONS character parameter SAS data set options that are to be applied to the retrieved data set. For a complete list of data set options, see the SAS Data Set Options topic in the SAS Help.

CSV_SEPARATOR character property that applies only when the RETRIEVE_DATASET CALL routine is called on a CSV file entry. When this occurs, the CSV file is transformed into a SAS data set. A binary CSV file is identified by a MIME type of *application/x-comma-separated-values*. Use the CSV_SEPARATOR property to indicate the separator to be used when creating the CSV file. The default separator is a comma. If the CSV file was created at publish time by transforming a SAS data set into a CSV file, then the separator used to create the CSV file will always take precedence. If the CSV file was not created at publish time, then the CSV_SEPARATOR property can be used to specify the separator value used. If the CSV file was not created at publish time and no separator property is specified, then the separator is specified as a comma, by default.

CSV_FLAG character property that only applies when calling the RETRIEVE_DATASET CALL routine for a binary file entry. A binary CSV file is identified by a MIME type of *application/x-comma-separated-values*. This property is a CSV override flag. By default when converting this binary CSV file into a SAS data set, the first line will be processed as variable names. The second line will be processed as variable label names. All remaining lines will be processed as data. To

override this default behavior, the CSV_FLAG value must be NO_VARIABLES or NO_LABELS. To specify both values, specify two CSV_FLAG properties, one with a value of NO_VARIABLES, the other with a value of NO_LABELS. By default, when a CSV file is converted into a data set, the variable lengths are determined by the first row of data. If subsequent rows have greater lengths, then the variable data is truncated. To override this default behavior, specify the CSV_FLAG with a property of NO_TRUNCATION. When this flag value is specified, truncation will not occur, but multiple passes of the data might be necessary in order to perform the resizing.

Type: Character, Input

Details

If the MEMNAME parameter is blank, then the RETRIEVE_DATASET CALL routine creates the data set using the original member name as it was defined at publish time.

Examples

Example 1: Using RETRIEVE_DATASET

The following example retrieves the data set WORK.OUTDATA entry from the package.

```
lib = 'work';
mem = 'outdata';
CALL RETRIEVE_DATASET(rid, lib, mem, rc);
```

Example 2: Using RETRIEVE_DATASET with CSV_FLAG Properties

The following example specifies two CSV_FLAG properties.

```
prop='CSV_SEPARATOR,CSV_FLAG,CSV_FLAG';
separator='/';
flag1 = 'NO_VARIABLES';
flag2 = 'NO_LABELS';
CALL RETRIEVE_DATASET(entryId, libname, memname,
    rc, prop, separator, flag1, flag2);
```

RETRIEVE_FILE

Retrieves an external binary or text file from a package

Syntax

```
CALL RETRIEVE_FILE(entryId, filename, rc);
```

Required Arguments

entryId

identifies the file entry.

Type: Numeric, Input

filename

specifies the name of the file or fileref, using the following syntax:

- FILENAME: *external_filename*
- FILEREF: *SAS_fileref*

Type: Character, Input

rc

receives a return code.

Type: Numeric, Output

Details

Specifying "**FILENAME:** ", without a filename, applies to the retrieved file the same name that was used when the file was initially inserted into the package.

Example: Using RETRIEVE_FILE

The following example retrieves a binary file from a queue.

```
fname = "filename: /users/jsmith.bin";
CALL RETRIEVE_FILE(entryId, fname, rc);
```

RETRIEVE_HTML

Retrieves an HTML entry from a package

Syntax

```
CALL RETRIEVE_HTML(entryId, path, body, bodyUrl, frame, frameUrl, contents,  
                   contentsUrl, pages, pagesUrl, rc<, properties, propValue1, ...propValueN> );
```

Required Arguments

entryId

identifies the HTML entry.

Type: Numeric, Input

path

specifies the full designation of the location that will receive the retrieved files.

Type: Character, Input

body

returns the name of the HTML body file.

Type: Character, Output

bodyUrl

returns the URL of the HTML body file.

Type: Character, Output

frame

returns the name of the HTML frame file.

Type: Character, Output

frameUrl

returns the URL of the HTML frame file.

Type: Character, Output

contents

returns the name of the HTML contents file.

Type: Character, Output

contentsUrl

returns the URL of the HTML contents file.

Type: Character, Output

pages

returns the name of the HTML page file.

Type: Character, Output

pagesUrl

returns the URL of the HTML page file.

Type: Character, Output

rc

receives a return code.

Type: Numeric, Output

Optional Arguments

properties

identifies a comma-separated list of optional property names. Valid property names are as follows:

- ENCODING
- BODY_TOTAL
- FILE_TOTAL
- COMPANION_TOTAL

Type: Character, Input

propValue1, ...propValueN

specifies one value for each specified property name. The order of the property values must match the order of the property names in the *properties* parameter. Valid property values are defined as follows:

ENCODING	input character string that indicates the target encoding for the retrieved HTML file. An example of a target encoding value is ISO-8859-1.
BODY_TOTAL	numeric output parameter that returns the total number of HTML body files published as part of this set.
FILE_TOTAL	numeric output parameter that returns the total number of all HTML files published as part of this

set. This includes all body, page, contents, frame, and additional HTML files and companion files.

COMPANION_TOTAL numeric output parameter that returns the total number of extraneous HTML files that were published as part of this set.

Type: Character or Numeric, Input or Output

Details

The ODS entry can contain any combination of the following: ODS HTML file, contents file, pages file, or frame file.

The publisher can choose to publish any combination of the HTML files. To indicate those files that were not published as part of this set, the output parameter that contains the created filename will be updated to "". For example, if only the body was published, then the page, contents, and frame parameters will be returned as "".

The **pages**, **pagesUrl**, **body**, **bodyUrl**, **frame**, **frameUrl**, **contents**, and **contentsUrl** parameters are character variables that are updated by the CALL routine. Because they are updated, they must be initialized with a length large enough to contain the name of the returned filename or URL. If the length of the character variable is less than the length of the returned filename or URL, the filename or URL will be truncated and a warning will be issued. When calling the RETRIEVE_HTML CALL routine from within the DATA step, use the LENGTH statement to define the length of the character variable. When calling RETRIEVE_HTML from within a macro, initialize the variable to some value so that it will have an appropriate length, as shown in the second example below.

For information on how HTML files are published and how the optional encoding property can be used to provide encoding information to package recipients, see [“Publish and Retrieve Encoding Behavior” on page 43](#).

Examples

Example 1: Using RETRIEVE_HTML

The following example retrieves HTML entry information from the package.

```
data _null_;
  length contents $64 frame $64 pages $64 body $64
    contentsUrl $256 frameUrl $256
    pagesUrl $256 bodyUrl $256;

  path = '/maintenance/schedule/doc';
  CALL RETRIEVE_HTML(entryId, path, body,
    bodyUrl, frame, frameUrl, contents,
    contentsUrl, pages, pagesUrl, rc);
```

Example 2: Using RETRIEVE_HTML with Macro Variables

The following example uses a macro to initialize a variable to a specific length and then retrieves HTML information from the package.

```
%macro initLen(variable, len);
  %let &variable=.;
  %do i=2 %to &len
    %let &variable=&&&variable
  %end;
```

```

%mend;

%initLen(contents, 64);
%initLen(contentsUrl, 256);
%initLen(pages, 64);
%initLen(pagesUrl, 256);
%initLen(body, 64);
%initLen(bodyUrl, 256);
%initLen(frame, 64);
%initLen(frameUrl, 256);
%let path =/users/maintenance/doc;
%let rc=0;
%syscall RETRIEVE_HTML(entryId, path, body,
    bodyUrl, frame, frameUrl, contents, contentsUrl,
    pages, pagesUrl, rc);

```

RETRIEVE_MDDB

Retrieves an MDDB entry from a package

Syntax

CALL RETRIEVE_MDDB(*entryId*, *libname*, *memname*, *rc*);

Required Arguments

entryId

identifies the MDDB entry.

Type: Numeric, Input

libname

specifies the SAS library that will contain the retrieved MDDB.

Type: Character, Input

memname

specifies the name of the retrieved MDDB.

Type: Character, Input

rc

receives a return code.

Type: Numeric, Output

Details

An MDDB is a multidimensional database (not a data set) offered by SAS. An MDDB is a specialized storage facility that can be created by tools such as multidimensional data viewers, which populate the MDDB with data that is retrieved from sources such as a data warehouse. The matrix format of MDDBs allows the viewer to access data quickly and easily.

If the **memname** parameter is blank, then the RETRIEVE_MDDB CALL routine creates the MDDB using the original member name as it was defined at publish time.

Example: Using RETRIEVE_Mddb

The following example retrieves an Mddb entry WORK.OUTDATA from the package:

```
lib = 'work';
mem = 'outdata';
CALL RETRIEVE_Mddb(entryId, lib, mem, rc);
```

RETRIEVE_NESTED

Retrieves the descriptive header information for a nested package entry

Syntax

CALL RETRIEVE_NESTED(*entryId*, *packageId*, *numEntries*, *desc*, *dateTime*, *nameValue*, *rc*);

Required Arguments

entryId

identifies the nested package entry.

Type: Numeric, Input

packageId

returns the identifier of the nested package.

Type: Numeric, Output

numEntries

returns the number of entries in the nested package.

Type: Numeric, Output

desc

returns the description of the nested package entry.

Type: Character, Output

dateTime

returns the date and time that the nested package was published, in GMT format.

Type: Numeric, Output

nameValue

returns the name/value pairs assigned to the specified entry. Name/value pairs are site-specific; they are used for the purpose of [filtering](#).

Type: Character, Output

rc

receives a return code.

Type: Numeric, Output

Details

The descriptive header information that is returned on this CALL routine includes the nested package description, name/value string, datetime stamp, and total number of entries that are contained in the nested package.

The returned **packageId** can then be used on subsequent [ENTRY_FIRST](#) and [ENTRY_NEXT](#) calls to retrieve the entry information.

Package identifiers that are returned on the RETRIEVE_NESTED CALL routine cannot be used on the PACKAGE_DESTROY CALL routine. The RETRIEVE_NESTED CALL routine supports only top-level packages, excluding nested packages. When PACKAGE_DESTROY is called on a top-level package, all entries, including all nested packages and their entries, are removed from the queue.

Example: Using RETRIEVE_NESTED

The following example retrieves the descriptive header information for the nested package entry that is associated with **entryId**.

```
packageId=0;
num=0;
desc=' ';
dt=0;
nv=' ';
rc=0;
CALL RETRIEVE_NESTED(entryId, packageId,
    num, desc, dt, nv, rc);
```

RETRIEVE_PACKAGE

This CALL routine retrieves descriptive header information for all packages

Syntax

CALL RETRIEVE_PACKAGE(*pkgListId*, *retrievalType*, *retrievalInfo*, *totalPackages*,
rc<, *properties*, *propValue1*, ...*propValueN*>);

Required Arguments

pkgListId

identifies the list of packages.

Type: Numeric, Output

retrievalType

specifies the transport to use when retrieving a package. Valid values include the following:

- FROM_QUEUE
- FROM_ARCHIVE
- FROM_WEBDAV
- FROM_SHAREPOINT

Type: Character, Input

retrievalInfo

specifies transport-specific information that determines the package to retrieve. When retrieving from an archive, specify the physical path and name of the archive, excluding the extension. When retrieving from a WebDAV-compliant server, specify the URL that identifies the package to retrieve. When retrieving from MSMQ queues, use the following syntax:

MSMQ://queueHostMachineName\queueName

When retrieving from IBM WebSphere MQ queues, use the following syntax:

`MQSERIES://queueManager:queueName`

or

`MQSERIES-C://queueManager:queueName`

MQSERIES specifies the server interface that is used to connect to a queue manager that is local. MQSERIES-C specifies the client interface that is used, allowing the queue manager to be local or remote.

When retrieving from SharePoint, use the value for SITE_URL.

Type: Character, Input

totalPackages

provides the total number of packages found.

Type: Numeric, Output

rc

receives a return code.

Type: Numeric, Output

Optional Arguments

properties

identifies a comma-separated list of optional property names. Valid property names are as follows:

- COLLECTION_NAME
- CORRELATIONID
- FTP_PASSWORD
- FTP_USER
- HTTP_PASSWORD
- HTTP_PROXY_URL
- HTTP_USER
- LIST_NAME
- NAMESPACES
- QUEUE_TIMEOUT

Type: Character, Input

propValue1, ...propValueN

specifies a value for each specified property name. The order of the property values must match the order of the property names specified in the *properties* parameter. Valid property values are defined as follows:

COLLECTION_NAME

CORRELATIONID

This character string specifies retrieval of only those packages that have the specified correlation identifier. (Applies only to the message queue transport.)

FTP_PASSWORD

When retrieving with the archive transport (FROM_ARCHIVE), this character string indicates the password that is used to connect to the remote

	host. Specify this property only when the host does not accept anonymous access. (Applies to the FROM_ARCHIVE property when the FTP protocol is used.)
FTP_USER	When retrieving with the archive transport, this character string indicates the name of the user to connect to the remote host. (Applies to the FROM_ARCHIVE property when the FTP protocol is used.)
HTTP_PASSWORD	When retrieving with the WebDAV transport (FROM_WEBDAV), this character string indicates the password used to bind to the Web server. Specify this property only when the Web server does not accept anonymous access. (Applies to the FROM_ARCHIVE property when the HTTP protocol is used.)
HTTP_PROXY_URL	When retrieving with the WebDAV transport, this character string indicates the URL of the proxy server. (Applies to the archive transport when the HTTP protocol is used with archive specifications.)
HTTP_USER	When retrieving with the WebDAV transport, this character string indicates the name of the user to bind to the Web server. (Applies to the FROM_ARCHIVE property when the HTTP protocol is used.)
LIST_NAME	When retrieving with the SharePoint transport, this character string indicates a document library in the SharePoint site. SharePoint document libraries are a special type of list that is used for ordering folders and files.
NAMESPACES	When retrieving with the WebDAV transport, this character string lists one or more namespaces that you are interested in, using the syntax shown in the following example: <code>a="http://www.host.com/myNamespace" A="http://www.host.com/myNamespace1" B="http://www.host.com/myNamespace2"</code>
QUEUE_TIMEOUT	This numeric value identifies the number of seconds for the poll time-out. By default, if this property is not specified, the RETRIEVE_PACKAGE CALL routine polls and returns immediately with the number of packages found, if any. To override this default, specify the QUEUE_TIMEOUT property so that the RETRIEVE_PACKAGE CALL routine will continue to poll for packages until at least one package is found on the queue or until the time-out occurs, whichever occurs first.

Type: Character or Numeric, Input

Details

When retrieving FROM_QUEUE, this CALL routine retrieves descriptive header information for all packages that are found on the specified queue. The total number of packages found is returned. The descriptive header information for each package can be obtained by executing the [PACKAGE_FIRST](#) and [PACKAGE_NEXT](#) CALL routines.

When retrieving from the queue transport, no entries or packages are removed or deleted from the queue. Packages can be removed by calling the PACKAGE_DESTROY CALL routine.

The ARCHIVE_PATH property identifies where the archive is created. When retrieving from an archive, the name of the archive can be specified as a physical pathname, an FTP URL, or an HTTP URL.

When retrieving from a WebDAV-compliant server, the name of the archive can be specified as an FTP URL or an HTTP URL.

In the z/OS operating environment, archives can be retrieved only from UNIX System Services directories.

Examples

Example 1: Using RETRIEVE_PACKAGE to Retrieve Information for Packages on a Queue

The following example opens the queue JSMITH and retrieves the descriptive header information for all packages on that queue and the total number of packages on the queue.

```
plist=0;
qname = "MQSERIES://LOCAL:JSMITH";
rc=0;
total=0;
nameValue='';
CALL RETRIEVE_PACKAGE(plist, "FROM_QUEUE",
    qname, total, rc);
```

Example 2: Using RETRIEVE_PACKAGE to Retrieve an Archive

The following example retrieves the archive named STATUS.

```
plist=0;
archiveName = "/maintenance/status";
rc=0;
total=0;
CALL RETRIEVE_PACKAGE(plist, "FROM_ARCHIVE",
    archiveName, total, rc);
```

Example 3: Using RETRIEVE_PACKAGE to Retrieve a Package from a WebDAV Server

The following example retrieves the package from WebDAV by using the specified URL.

```
plist=0;
url = "http://AlphaliteAirways.host.com/~flights";
rc=0;
total=0;
property='namespaces';
```

```

ns="homePage='http://AlphaliteAirs.com/'";
CALL RETRIEVE_PACKAGE(plist, "FROM_WEBDAV",
    url, total, rc, property, ns);

```

Example 4: Using RETRIEVE_PACKAGE with a Queue Polling Time-out

The following example applies a queue polling time-out value of 120 seconds. The RETRIEVE routine seeks packages on the queue until at least one package is located or the 120-second time-out expires, whichever occurs first.

```

plist=0;
qname = "MQSERIES://LOCAL:JSMITH";
rc=0;
total=0;
nameValue='';
prop='queue_timeout';
timeout = 120;
CALL RETRIEVE_PACKAGE(plist, "FROM_QUEUE",
    qname, total, rc, prop, timeout);

```

Example 5: Using RETRIEVE_PACKAGE to Extract Entries from an Archive

The following example is a SAS program that extracts entries from an archive. The RETRIEVE_PACKAGE routine opens the archive file and retrieves the headers for all package entries. Subsequent RETRIEVE routines are called to retrieve the contents of the entries (in this example, data sets, catalogs, and MDDBs) and to dispose of them appropriately.

```

data _null_;
    length description nameValue type userSpec msg $255;
    length libname $8;
    length memname $32;
    CALL RETRIEVE_PACKAGE(packageId, "FROM_ARCHIVE",
        "AlphaliteAir", tp, rc);
    do while (rc = 0);
        CALL ENTRY_NEXT(packageId, eid, type, userSpec,
            description, nameValue, rc);
        if (rc = 0) then select (type);
            when ("DATASET")
                CALL RETRIEVE_DATASET(eid, libname, memname, rc);
            when ("CATALOG")
                CALL RETRIEVE_CATALOG(eid, libname, memname, rc);
            when ("MDDB")
                CALL RETRIEVE_MDDb(eid, libname, memname, rc);
            otherwise;
        end;
    end;
    CALL PACKAGE_TERM(packageId, rc);
run;

```

Example 6: Using RETRIEVE_PACKAGE to Retrieve a Package from SharePoint

The following example is a SAS program that retrieves a package from SharePoint.

```

data _null_;
  length properties nameValue entryType userSpec filename filepath channel $ 255;
  length abstract desc msg $2048;
  abstract      = '';
  channel       = '';
  entryCount    = 0;
  entryId       = 0;
  packageId     = 0;
  rc            = 0;
  totalPackages = 0;
  packageId     = 0;
  numEntries    = 0;

  retrievalType = 'FROM_SHAREPOINT';
  siteUrl       = 'http://host/site';
  retrievalInfo = siteUrl;
  properties    = 'LIST_NAME, COLLECTION_FOLDER, HTTP_USER, HTTP_PASSWORD';
  listName      = 'Shared Documents';
  folderName    = 'File Collection';
  userName      = 'username';
  password      = 'password';

  CALL RETRIEVE_PACKAGE(packageId, retrievalType, retrievalInfo, totalPackages, rc,
                        properties, listName, folderName, userName, password);

  if rc ne 0 then do;
    msg = sysmsg();
    put msg;
    ABORT;
  end;

  properties = 'ABSTRACT';

  CALL PACKAGE_FIRST(packageId, numEntries, desc, dateTime, nameValue, channel, rc,
                    properties, abstract);

  if (rc = 0) then do;
    put "packageId: "    packageId;
    put "  numEntries: " numEntries;
    put " Description: " desc;
    put " nameValue: "  nameValue;
    put "  dateTime: "  dateTime datetime21.2 " GMT";
    put "  abstract: "  abstract;
  end;
else do;
  msg = sysmsg();
  put msg;
  ABORT;
end;

properties = 'FILENAME';
do while (rc = 0);
  CALL ENTRY_NEXT(packageId, entryId, entryType, userSpec, desc, nameValue, rc,
                  properties, filename);
  if (rc = 0) then do;
    entryCount = entryCount + 1;
    put 'Entry: ' entryCount;
    put '      EntryId: '    entryId;
  end;
end;

```

```

        put '          Type: '          entryType;
        put '          UserSpec: '      userSpec;
        put '          Description: ' desc;
        put '          nameValue: '     nameValue;

        if (entryType = 'TEXT' OR entryType = 'BINARY') then do;
            filepath = 'FILENAME: C:\Public\' || filename;
            CALL RETRIEVE_FILE(entryId, filepath, rc2);
            if (rc2 ne 0) then do;
                msg = sysmsg();
                put msg;
            end;
            else do;
                put 'File retrieved: ' filepath;
            end;
        end;
    end;
else do;
    msg = sysmsg();
    put msg;
end;
end; /* do while */

CALL PACKAGE_TERM(packageId, rc);
if rc ne 0 then do;
    msg = sysmsg();
    put msg;
    ABORT;
end;
run;

```

RETRIEVE_REF

Retrieves a reference from a package

Syntax

CALL RETRIEVE_REF(*entryId*, *referenceType*, *reference*, *rc*);

Required Arguments

entryId

identifies the reference entry to be retrieved.

Type: Numeric, Input

referenceType

returns the type of the reference, the value of which can be HTML or URL.

Type: Character, Output

reference

returns the value of the reference.

Type: Character, Output

rc
 receives a return code.
Type: Numeric, Output

Example: Using RETRIEVE_REF

The following example retrieves a reference entry from a package.

```
refType='';
ref='';
CALL RETRIEVE_REF(rid, refType, ref, rc);
```

RETRIEVE_SQLVIEW

Retrieves a PROC SQL view from a package

Syntax

```
CALL RETRIEVE_SQLVIEW(entryId, libname, memname, rc);
```

Required Arguments

entryId
 identifies the PROC SQL view entry.
Type: Numeric, Input

libname
 specifies the SAS library that will contain the retrieved PROC SQL view.
Type: Character, Input

memname
 specifies the member name of the PROC SQL view.
Type: Character, Input

rc
 receives a return code.
Type: Numeric, Output

Details

If the ***memname*** parameter is blank, then the RETRIEVE_SQLVIEW CALL routine creates the PROC SQL view by using the original member name as it was defined at publish time.

Example: Using RETRIEVE_SQLVIEW

The following example retrieves the PROC SQL view WORK.OUTDATA from the package.

```
lib = 'work';
mem = 'outdata';
CALL RETRIEVE_SQLVIEW(entryId, lib, mem, rc);
```

RETRIEVE_VIEWER

Retrieves a viewer entry from a package

Syntax

CALL RETRIEVE_VIEWER(*entryId*, *filename*, *rc*<, *properties*, *propValue1*, ...*propValueN*>);

Required Arguments

entryId

identifies the file entry.

Type: Numeric, Input

filename

specifies the name of the file or fileref, using the following syntax:

- FILENAME: *external_filename*
- FILEREF: *SAS_fileref*

Type: Character, Input

rc

receives a return code.

Type: Numeric, Output

Optional Arguments

properties

identifies a comma-separated list of optional property names. Valid property names are as follows:

- ENCODING

Type: Character, Input

propValue1, ...propValueN

specifies one value for each specified property name. The order of the property values must match the order of the property names in the *properties* parameter. Valid property values are defined as follows:

ENCODING input character string that indicates the target encoding for the retrieved viewer file. An example of a target encoding value is ISO-8859-1.

Type: Character, Input

Details

Specifying "**FILENAME:** ", without an external filename, applies to the retrieved file the same name that was used when the file was initially inserted into the package.

Example: Using RETRIEVE_VIEWER

The following example retrieves a viewer from a package.


```
fname = "filename: /users/jsmith.bin";
CALL RETRIEVE_VIEWER(entryId, fname, rc);
```

Examples

Example 1: Example: Publishing in the DATA Step

The following example builds a package, explicitly publishes to two queues, and then publishes to a channel that is defined in the SAS Metadata Repository. This example uses the DATA step, but easily can be changed to use the macro interface.

```
filename f 'c:\frame.html';
filename c 'c:\contents.html';
filename p 'c:\page.html';
filename b 'c:\body.html';
ods html frame = f contents =c(url="contents.html")
    body = b(url="body.html") page=p(url="page.html");

/* run some data steps/procs to generate ODS output */
data b;
    do i= 1 to 1000;
        output;
    end;
run;

data emp;
input fname $ lname $ ages state $ siblings;
cards;
Steph Lyons 32 NC 4
Richard Jones 40 NC 2
Mary White 74 NC 1
Kai Burns 3 NC 1
Dakota Nelson 1 NC 1
Paul Black 79 NC 8
Digger Harris 5 NC 0
Coby Thomas 1 NC 0
Julie Mack 6 NC 1
Amy Gill 3 NC 1
Brian Meadows 16 HA 1
Richard Wills 17 HA 1
Diane Brown 48 CO 4
Pamela Smith 42 HA 4
Joe Thompson 44 WA 10
Michael Grant 23 IL 1
Michael Ford 31 NM 4
Ken Bush 28 NC 1
Brian Carter 27 NC 1
Laurie Clinton 32 NC 1
Kevin Anderson 33 NC 1
Steve Kennedy 33 NC 1
run;
```

```

quit;

proc print;run;
proc contents;run;
ods html close;

data _null_;
  rc=0;a='a';b='b';c='c';d='d';
  length filename $64 mimeType $24 fileType
    $10 nameValue $100 description $100;

  packageId = 0;
  nameValue="type=(test) coverage=(filtering,
    transports)";
  CALL PACKAGE_BEGIN(packageId,"Main results package.",
    nameValue, rc);
  if (rc eq 0) then put 'Package begin successful.';
  else do;
    msg = sysmsg();
    put msg;
  end;

  gifpid=0;
  CALL PACKAGE_BEGIN(gifpid,"Gif nested package.",'', rc);
  if (rc eq 0) then put 'Gif package begin successful.';
  else do;
    msg = sysmsg();
    put msg;
  end;

  nameValue="type=report, topic=census";
  description="North Carolina residents.";
  libname = "WORK";
  memname="EMP";
  CALL INSERT_DATASET(packageId, libname, memname,
    description, nameValue, rc);
  if rc ne 0 then do;
    msg = sysmsg();
    put msg;
  end;
  else
    put 'Insert data set successful.';

  CALL INSERT_HTML(packageId,"fileref:b", "",
    "fileref:f", "",
    "fileref:c", "",
    "fileref:p", "", "ODS HTML Output","", rc);
  if rc ne 0 then do;
    msg = sysmsg();
    put msg;
  end;
  else
    put 'Insert html successful.';

  CALL INSERT_DATASET(packageId,"WORK","b",
    "B dataset...","", rc);

```

```

if rc ne 0 then do;
    msg = sysmsg();
    put msg;
end;
else
    put 'Insert data set successful';

/* insert package (nested package */
CALL INSERT_PACKAGE(packageId, gifpid,rc);
if rc eq 0 then put 'Insert package successful';
else do;
    msg = sysmsg();
    put msg;
end;

description = "Gif for marketing campaign.";
filename = "filename:campaign01.01.gif";
fileType = "Binary";
mimeType = "Image/Gif";
CALL INSERT_FILE(gifpid, filename, fileType,
    mimeType, description, '', rc);
if rc ne 0 then do;
    msg = sysmsg();
    put msg;
end;
else
    put 'Insert file (gif) successful.';

description = "Test VRML file.";
filename = "filename:test.wrl";
fileType = "text";
mimeType = "model/vrml";
CALL INSERT_FILE(packageId,filename, fileType,
    mimeType, description, '', rc);
if rc ne 0 then do;
    msg = sysmsg();
    put msg;
end;
else
    put 'Insert file (vrml) successful.';

/* send package to the two queues that are specified */
properties='';
CALL PACKAGE_PUBLISH(packageId, "TO_QUEUE", rc, properties,
    "MQSERIES://JSMITH.LOCAL", "MSMQ://JSMITH\transq");
if rc ne 0 then do;
    msg = sysmsg();
    put msg;
end;
else
    put 'Publish successful';

/* publish to the filter test channel
   defined in the SAS Metadata Repository */
storeinfo=
    "SAS-OMA://alpair01.sys.com:8561";

```

```

channel1= 'FilterTest1';
properties='channel_store, subject, metauser, metapass';
subject="Filter Testing Results";
user = "myUserName";
password = "myPassword";
CALL PACKAGE_PUBLISH(packageId, "TO_SUBSCRIBERS", rc,
    properties, storeinfo, subject, user, password,
    channel1);
if rc ne 0 then do;
    msg = sysmsg();
    put msg;
end;
else
    put 'Publish successful';

CALL PACKAGE_END(packageId,rc);
if rc ne 0 then do;
    msg = sysmsg();
    put msg;
end;
else
    put 'Package end successful';
run;
quit;

```

Example 2: Example: Publishing in a Macro

The following example builds a package, publishes the package to a channel that is defined in the SAS Metadata Repository, and then explicitly publishes to one queue. This example uses macro variables rather than the DATA step, which allows you the flexibility to use CALL routines throughout your code, as other processing completes. This example also illustrates how publish CALL routines are used with other SAS statements.

```

%macro chkrc(function);
%if &rc = 0 %then %put "NOTE: &function succeeded.";
%else %do;
    %let msg= %sysfunc(sysmsg());
    %put &function failed - &msg;
%end;
%mend;

%let packageId = 0;
%let nameValue=type=(test) coverage=(filtering,
    transports);
%let rc = 0;
%let packageId = 0;
%let description=main results package;
%syscall package_begin(packageId,description, nameValue, rc);
%chkrc(Package Begin);

%let gifpid=0;
%let description=Gif nested package. ;
%let nameValue=;
%syscall package_begin(gifpid, description,

```

```

        nameValue, rc);
%chkrc(Package Begin);

filename f 'c:\frame.html';
filename c 'c:\contents.html';
filename p 'c:\page.html';
filename b 'c:\body.html';
ods html frame = f
    contents =c(url="contents.html")
    body = b(url="body.html")
    page=p(url="page.html");

/* run some data steps/procs to generate ODS output */
data b;
    do i= 1 to 1000;
        output;
    end;run;

%let nameValue=;
%let description= B, testing dataset;
%let libname = WORK;
%let memname= B;
%syscall insert_dataset(packageId,libname, memname,
    description, nameValue, rc);
%chkrc(Insert Dataset);

data emp;
input fname $ lname $ ages state $ siblings;
cards;
Steph Jones 32 NC 4
Richard Long 40 NC 2
Mary Robins 74 NC 1
Kai Mack 3 NC 1
Dakota Wills 1 NC 1
Paul Thomas 79 NC 8
Digger Johnson 5 NC 0
Coby Gregson 1 NC 0
Julie Thomson 6 NC 1
Amy Billins 3 NC 1
Brian Gere 16 HA 1
Richard Carter 17 HA 1
Diane Ford 48 CO 4
Pamela Aarons 42 HA 4
Joe Ashford 44 WA 10
Michael Clark 23 IL 1
Michael Harris 31 NM 4
Ken Porter 28 NC 1
Brian Harrison 27 NC 1
Laurie Smith 32 NC 1
Kevin Black 33 NC 1
Steve Jackson 33 NC 1
run;
quit;

%let nameValue= type=report, topic=census;
%let description=North Carolina residents.;

```

```

%let libname = WORK;
%let memname= EMP;
%syscall insert_dataset(packageId, libname, memname,
    description, nameValue, rc);
%chkrc(Insert Dataset);

proc print;run;
proc contents;run;
ods html close;

%let body=fileref:b;
%let frame=fileref:f;
%let contents=fileref:c;
%let pages=fileref:p;
%let description=Generated ODS output.;
%let curl="contents.html";
%let burl = "body.html";
%let furl = "frame.html";
%let purl = "page.html";
%syscall insert_html(packageId, body, burl, frame,
    frameurl, contents, curl, pages, purl,
    description, nameValue, rc);
%chkrc(Insert Html);

/* insert nested package */
%syscall insert_package(packageId, gifpid,rc);
%chkrc(Insert Package);

%let giffile=filename:Boeing747.gif;
%let description=New 747 paint scheme.;
%let filetype = text;
%let mimetype = %quote(Image/gif);
%syscall insert_file(gifpid, giffile, filetype,
    mimetype, description, nameValue, rc);
%chkrc(Insert File);

/* publish to the filter test channel
   defined in the SAS Metadata Repository */
%let storeinfo=
    "SAS-OMA://alpair01.sys.com:8561";
%let channel1=FilterTest;
%let properties='channel_store, subject, metauser, metapass';
%let subject=Filter Testing Results;
%let user = myUserName;
%let password = myPassword;
%let publishType=to_subscribers;
CALL PACKAGE_PUBLISH(packageId, "TO_SUBSCRIBERS", rc,
    properties, storeinfo, subject, user, password,
    channel1);
%chkrc(publish package);

/* publish one queue */
%let property=;
%let publishType = to_queue;
%let queueName=mqseries://JSMITH:LOCAL;
%syscall package_publish(packageId, publishType,

```

```

rc, property, queueName);
%chkrc(publish package);

%syscall package_end(packageId,rc);
%chkrc(Package End);

run;
quit;

```

Example 3: Example: Publishing with the FTP Access Method

The following example uses the FTP access method to put ODS-generated output onto the server jsmith.pc.alpair.com in the Windows operating environment. Note that the INSERT_REFERENCE CALL routine is used so that only references to the newly created HTML files are inserted into the package. Subscribers using the EMAIL transport engine receive an e-mail message, with an embedded link to the HTML files within it.

```

filename myfram ftp 'odsftpf.htm'
    host='jsmith.pc.alpair.com'
    user='anonymous'
    pass='smi96Gth';

filename mybody ftp 'odsftpb.htm'
    host='jsmith.pc.alpair.com'
    user='anonymous'
    pass='smi96Gth';

filename mypage ftp 'odsftpp.htm'
    host='jsmith.pc.alpair.com'
    user='anonymous'
    pass='smi96Gth';

filename mycont ftp 'odsftpc.htm'
    host='jsmith.pc.alpair.com'
    user='anonymous'
    pass='smi96Gth';

ods listing close;
ods html frame=myfram body=mybody page=mypage
    contents=mycont;

title 'ODS HTML Generation to PC Files via
    FTP Access Method';
data retail;
    set alpairhelp.retail;
    decade = floor(year/10) * 10;
run;

proc format;
    /* maps foreground colors for total sales */
    value salecol
        low-1500 = 'red'
        1500-2000 = 'yellow'
        2000-high = 'green';

```

```

/* gives the row labels some color */
value decbg
  1980 = '#00aaaa'
  1990 = '#00cccc';

/* gives the decade flyovers */
value decfly
  1980 = 'Me Me Me Generation'
  1990 = 'Kinder Gentler Generation';
run;

proc tabulate data=retail;

class year decade;
classlev decade / s={background=decbg.
  flyover=decfly.};
classlev year / s=<parent>;

var sales;
table decade * year ,
  sales *
    ( sum * f=dollar12. *
      {style={foreground=salecol. font_weight=bold}}
    median * f=dollar12. * {style={foreground=black}}
    );
run;

data a;
  do j=1 to 5;
    do k=1 to 5;
      do i = 1 to 10;
        x=ranuni(i);
        output;end; end; end;
run;

proc sort data=a; by j;
run;

proc sort data=a; by j k;
run;

proc univariate; by j k; var i;
run;

title1;
quit;

ods html close;

data _null_;
length desc $ 1000;
rc=0;a='a';b='b';c='c';d='d';

packageId = 0;

```



```

CALL PACKAGE_BEGIN(packageId,"Weekly Activities Report",
    'Type=Report', rc);
if (rc eq 0) then put 'Package begin successful.';
else do;
    msg = sysmsg();
    put msg;
end;

desc="Retail sales information and miscellaneous
    statistics viewed at :";
nv="";
CALL INSERT_REF(packageId, "HTML",
    "http://jsmith.pc.alpair.com/odsftpf.htm",
    desc, nv, rc);
if rc ne 0 then do;
    msg = sysmsg();
    put msg;
end;
else
    put 'Insert reference ok';

storeinfo =
    "SAS-OMA://alpsrv03.sys.com:8561";
channell= 'emailonly';
subject = "Monday's Update";
properties = 'subject, channel_store, metauser, metapass';
user = "myUserName";
password = "myPassword";
CALL PACKAGE_PUBLISH(packageId, "TO_SUBSCRIBERS", rc,
    properties, subject, storeinfo, user, password,
    channell);
if rc ne 0 then do;
    msg = sysmsg();
    put msg;
end;
else
    put 'Publish successful';

CALL PACKAGE_END(packageId,rc);
if rc ne 0 then do;
    msg = sysmsg();
    put msg;
end;
else
    put 'Package term successful';

run;
quit;

```


Chapter 6

Generating and Publishing Events

What Is an Event?	149
Overview of Generating and Publishing Events	150
Using Explicit Event Publication	150
Using Implicit Event Publication	150
XML Specifications for Events	150
Using the Publish Event Interface	151
XML Specification for Generic Events	151
XML Specification for SASPackage Events	153
Sample Code	153
Header Elements	155
Body Elements	156
Dictionary	158
EVENT_BEGIN	158
EVENT_BODY	161
EVENT_PUBLISH (Publish Event to HTTP)	162
EVENT_PUBLISH (Publish Event to Queues)	163
EVENT_PUBLISH (Publish Event to Subscribers)	166
EVENT_END	168
Examples	168
Example 1: Explicitly Generated Event	168
Example 2: Implicitly Published Event Containing an External File and a Reference	169
Example 3: Implicitly Published Event Containing a SAS Data Set, Text File, and HTML File	170

What Is an Event?

You can use the Publishing Framework to generate and publish events. In this context, an *event* is an action or occurrence that can be detected by a computer program. The Publishing Framework creates events by using XML specifications for events that contain the name of the event, a set of associated properties, and a message body.

The Publishing Framework provides two methods for publishing an event:

explicit event publication

- enables a SAS program to generate any type of event and publish it using HTTP, message queuing, or a publication channel.

implicit event publication

enables a channel's subscribers to be designated as event subscribers. The Publishing Framework then automatically notifies event subscribers whenever new information is published to the channel.

Overview of Generating and Publishing Events

Using Explicit Event Publication

Explicit event publication enables a SAS program to generate an event of any kind and publish it explicitly. First, the event is defined by using the CALL routines EVENT_BEGIN and EVENT_BODY. The CALL routine EVENT_PUBLISH is used to generate the event and publish it by using HTTP, message queuing, or a publication channel. The event is generated by using an [XML specification for generic events](#). The CALL routine EVENT_END is then used to free all resources associated with the event.

Note: The Publishing Framework does not currently support event retrieval. However, custom programs can be developed to provide this functionality.

To collect and process events that the Publishing Framework generates, you can develop customized programs by using the Event Broker service. The Event Broker is one of the Foundation Services that is provided with SAS Integration Technologies.

Using Implicit Event Publication

The PACKAGE_PUBLISH CALL routine automatically generates a SASPackage event whenever a package is published to a channel. The SASPackage event is captured as a well-formed XML document that describes the package. For details, see the [“XML Specification for SASPackage Events” on page 153](#).

Subscribers to the channel who are designated as event subscribers then receive the event via their chosen transport methods. This feature enables subscribers to be aware that new information has been published to the channel. To designate a subscriber as an event subscriber, use the Publishing Framework plug-in for SAS Management Console.

To collect and process events that the Publishing Framework generates, you can develop customized programs by using the Event Broker service. The Event Broker is one of the Foundation Services that is provided with SAS Integration Technologies.

XML Specifications for Events

The Publishing Framework uses well-formed XML specifications to generate events. For detailed information, see the following topics:

- [XML specification for generic events](#), which is used for explicit event publishing
- [XML specification for SASPackage events](#), which is used for implicit event publishing
- the examples of implicitly and explicitly generated events at the end of this chapter, which use both of these specifications

Using the Publish Event Interface

The Publishing Framework supports the generation and publication of events. Explicit event publication enables a SAS program to generate an event of any kind and publish it explicitly.

First, you use the following CALL routines to define the event:

- [EVENT_BEGIN](#)
- [EVENT_BODY](#)

Once the event is defined, you can do the following:

- Use the [EVENT_PUBLISH](#) CALL routine to generate an event that can be published by using HTTP, message queuing, or a publication channel. The event is generated by using an XML specification that contains the name of the event, a set of associated properties, and a message body. For detailed information, see the “[XML Specification for Generic Events](#)” on page 151 and the examples of implicitly and explicitly generated events at the end of this chapter.
- Use the [EVENT_END](#) CALL routine to free all resources that are associated with the event.

Note: The Publishing Framework does not currently support event retrieval.

To collect and process events that the Publishing Framework generates, you can develop customized programs by using the Event Broker service. The Event Broker is one of the Platform Services that is provided with SAS Integration Technologies.

The Publishing Framework also supports implicit event publication. This feature enables a channel's subscribers to be designated as event subscribers. The Publishing Framework then automatically notifies event subscribers whenever new information is published to the channel.

XML Specification for Generic Events

Events are published by using XML documents. The following XML specification is used for generic events, that is, events that are generated explicitly by using the event publishing CALL routines.

The [EVENT_PUBLISH](#) CALL routine automatically generates the header portion of the document by using information that you provide in the [EVENT_BEGIN](#) CALL routine properties. It creates the body portion of the document using information that you provide in the [EVENT_BODY](#) CALL routine properties.

```
<?xml version="1.0" encoding="UTF-8"?>
<sas-event:Event
  xmlns:sas-event=
    "http://support.sas.com/xml/namespace/services.events-1.1"
  sas-event:name='event_name">
  <sas-event:Header>
    <sas-event:Version>1.0</Version>
    <sas-event:Identity>guid</sas-event:Identity>
    <sas-event:Credentials
```

```

        sas-event:name='userid"
        sas-event:password='password"
        sas-event:domain='domain"/>

<sas-event:Priority>priority</sas-event:Priority>

<sas-event:SentFrom>originator_description</sas-event:SentFrom>

<sas-event:SentAt:timestamp</sas-event:SentAt>
    <sas-event:Response
        sas-event:type='none|ack|result"/>
    <sas-event:Properties>

        <PropertyName>property_value</PropertyName>
        ....
    </sas-event:Properties>
</sas-event:Header>
<sas-event:Body>
    body content
    ...
</sas-event:Body>
</sas-event:Event>

```

The following table explains the elements:

Table 6.1 Elements

Element	Description
Event	Specifies the root element that names the event. The sas-event namespace is defined in this element.
Header	Begins the event header properties.
Version	Specifies the version of the event message that is required to support multiple specifications as the service matures.
Identity	Specifies the unique identifier. Responses echo this identifier so that the originator can use it for correlation.
Credentials	Specifies the credentials that are used for authentication and authorization checks. Events that are defined at a broker can be configured so that the sender of the event must be authenticated and authorized to run the event. A configured event can also specify a security context under which it should be run.
Priority	Is mapped to a Java thread priority so that process flows and listener dispatchers can be run at different priority levels.
SentFrom	Specifies the description of event originator that is used for logging purposes if available.

Element	Description
SentAt	Specifies the time that the event was sent. This information is echoed to the sender along with completion times. All times should be formatted according to the International Standard ISO 8601 standard. A brief summary is available on the W3C Web site at: http://www.w3.org/TR/NOTE-datetime.html .
Response	Specifies the type of response that the sender of an event wants. The supported types are no response (NONE), acknowledgment (ACK), and request/response (RESULT). Both NONE and ACK act as broadcast events.
Properties	Specifies user-defined properties that the originator can send to be used during event execution.
PropertyName	Specifies the value assigned to the property called PropertyName.
Body	Specifies the actual message content to be acted upon.

XML Specification for SASPackage Events

Sample Code

Implicitly generated events (SASPackage events) are published by using well-formed XML documents whose details are generated as a result of the package publishing process.

In the following example, the published package contains each entry type: SAS catalog, SAS data set, external file, MDDb, HTML file, file reference, SQL view, viewer, and nested package.

```
<?xml version="1.0" encoding="UTF-8">
<sas-event:Event
  xmlns:sas-event=
    "http://support.sas.com/xml/namespace/services.events-1.1"
  sas-event:name="SASPackage.ChannelName">
  <sas-event:Header>
    <sas-event:Version>1.0</sas-event:Version>
    <sas-event:SentAt>timestamp</sas-event:SentAt>
  </sas-event:Header>
  <sas-event:Body>
    <sas-publish:Package
      xmlns:sas-publish=
        "http://support.sas.com/xml/namespace/services.publish-1.1"

      xmlns:userSpecPkgNamespace='userSpecPkgNamespaceValue'
      sas-publish:version="1.0"
      sas-publish:packageUrl='URL to saved package'
```

```

sas-publish:description='package description'
sas-publish:abstract='package abstract'
sas-publish:channel='channel on which the package was
published"
    userSpecName='value">
<sas-publish:Entries>
  <sas-publish:Entry sas-publish:type="catalog"
    sas-publish:description='description'
    userSpecName='value">
    <sas-publish:Catalog
      sas-publish:name='member name"/>
    </sas-publish:Entry>
  <sas-publish:Entry sas-publish:type="dataset"
    sas-publish:description='description'
    userSpecName='value">
    <sas-publish:Dataset
      sas-publish:name='member name"/>
    </sas-publish:Entry>
  <sas-publish:Entry sas-publish:type="file"
    sas-publish:description='description'
    userSpecName='value">
    <sas-publish:File
      sas-publish:type='text or binary'
      sas-publish:name='file name'
      sas-publish:mimetype='MIME type"/>
    </sas-publish:Entry>
  <sas-publish:Entry sas-publish:type="html"
    sas-publish:description='description'
    userSpecName='value">
    <sas-publish:HTML
      sas-publish:type='body, frame, contents or
page"
      sas-publish:name='file name'
      sas-publish:url='URL"/>
      <sas-publish:Companion
        sas-publish:name='file name'
        sas-publish:url='URL'
        sas-publish:mimetype='MIME
type"/>
    </sas-publish:Entry>
  <sas-publish:Entry sas-publish:type="mddb"
    sas-publish:description='description'
    userSpecName='value">
    <sas-publish:Mddb
      sas-publish:name='member name"/>
    </sas-publish:Entry>
  <sas-publish:Entry
    sas-publish:type="reference"
    sas-publish:description='description'
    userSpecName='value">
    <sas-publish:Reference
      sas-publish:type='html or url'
      sas-publish:reference='reference"/>
    </sas-publish:Entry>
  <sas-publish:Entry
    sas-publish:type="sqlview"

```



```

        sas-publish:description='description'
        userSpecName='value">
        <sas-publish:SQLview
            sas-publish:name='member name"/>
    </sas-publish:Entry>
    <sas-publish:Entry
        sas-publish:type="viewer"
        sas-publish:description='description'
        userSpecName='value">
        <sas-publish:Viewer
            sas-publish:type='text or html"
            sas-publish:name='file name"
            sas-publish:mimetype='MIME type"/>
    </sas-publish:Entry>
    <sas-publish:Entry
        sas-publish:type="nestedpackage"
        sas-publish:description='description'
        userSpecName='value">
        <sas-publish:Package
            xmlns:userSpecPackageNamespace=
                'userSpecPackageNamespaceValue"
            sas-publish:description='package
description"
            sas-publish:abstract='package abstract"
            userSpecName='value" >
            <sas-publish:Entries>
                <sas-publish:Entry
                    sas-publish:type="entry type"

sas-publish:description='description"
                userSpecName='value">
                </sas-publish:Entry>
                .
                .
                additional Entry elements for this nested
package
                .
                .
            </sas-publish:Entries>
        </sas-publish:Package>
    </sas-publish:Entry>
</sas-publish:Entries/>
</sas-publish:Package>
</sas-event:Body>
</sas-event:Event>

```

Header Elements

The header elements are standard event elements. The event name is specified as:

```
"SASPackage.ChannelName"
```

The following table explains the header elements:

Table 6.2 Header Elements

Element	Description
Version	Specifies the version of the event message.
SentAt	Is a timestamp that specifies when the event was sent.

Body Elements

The following table explains the body elements:

Table 6.3 Body Elements

Element	Description
Package	<p>Begins the package definition. The following attributes are supported for this element:</p> <ul style="list-style-type: none"> version the version of SASPackage. packageUrl URL to saved package. description the package description. abstract the package abstract. channel the channel that the package was published to. packageUrl a URL to the saved package. Packages are saved to an archive or to a WebDAV server. <i>user-specified</i> one or more user-specified name/value pairs. <p>The sas-publish namespace is defined in this element. Also, other user-specified namespaces can be declared in this element via the xmlns: prefix. These will be included if the user specified the NAMESPACES property when creating the package.</p>
Entries	Defines the entries that are contained within the package.

Element	Description
Entry	<p>Defines an individual package entry. This element contains the required attribute, type. This attribute identifies the type of entry. Valid types include catalog, dataset, file, html, mddb, reference, sqlview, viewer, and nestedpackage.</p> <p>If the entry description was specified at publish time, the description attribute will be included. If a name/value is specified for the entry, one or more user-specified name/value attributes are included in the element. The name portion of the name/value specification is provided as the attribute, and the value portion is provided as the attribute value.</p>
Catalog	Defines an individual catalog entry. The required name attribute provides the catalog member name.
Dataset	Defines an individual data set entry. The required name attribute provides the data set member name.
File	Defines an individual file entry. The name and type attributes are required. They provide the filename and the file type. Valid file types include text or binary . The user-specified MIME type of the file entry will be provided in the mimetype attribute.
HTML	Defines an individual HTML entry. The required type and name attributes identify the type of HTML file and its filename. Valid types include body , fname , contents , and page . The url attribute identifies the URL.
Companion	Defines an individual companion file in an HTML entry. The required name attribute identifies name of the file. The url and mimetype attributes can be provided. They identify the URL and the MIME type of the file, respectively.
Mddb	Defines an individual Mddb entry. The required name attribute provides the Mddb member name.
Reference	Defines an individual reference entry. The required type attribute identifies the type of reference, either html or url . The actual reference that is inserted into the package is provided in the reference attribute.
SQLview	Defines an individual SQL view. The required name attribute provides the SQL view member name.
Viewer	Defines an individual viewer entry. The viewer type is provided in the type attribute. Valid types include text or html . The name attribute identifies the name of the viewer file. The user-specified MIME type for the viewer entry will be provided in the mimetype attribute.

Dictionary

EVENT_BEGIN

Initializes an event and returns an event identifier that uniquely identifies it

Syntax

```
CALL EVENT_BEGIN(eventId, name, rc<,properties, propValue1, ...propValueN> );
```

Required Arguments

eventId

identifies the new event.

Type: Numeric, Output

name

identifies a user-specified name of the event. The name should correspond to the name of an event that is defined in the Event Broker Service Process Flow Configuration. For more information, see **com.sas.services.events.broker** in the Foundation Services class documentation.

Type: Character, Input

rc

receives a return code.

Type: Numeric, Input

Optional Arguments

properties

identifies a comma-separated list of optional property names. There are two types of properties: well-known and user-defined. EVENT CALL routines recognize and process well-known properties. Some of the well-known properties are used to build the header portion of the event. Other well-known properties are used by the CALL routines to manage results that are returned as a result of the event execution. Well-known properties are as follows:

- DOMAIN
- IDENTITY
- PASSWORD
- PRIORITY
- RESPONSE
- RESULT_URL
- SENT_FROM
- USER

In addition to the well-known properties, you can also specify user-defined properties. The user-defined properties are passed to the Event Broker, which passes

these properties to the processing nodes, as needed. If a user-defined property is specified, the property value can be any user-specified character string.

Type: Character, Input

propValue1, ...propValueN

specifies a value for each specified property name. The order of the property values must match the order of the property names in the *properties* parameter. A value must be specified for each property that is specified in the *properties* parameter. Valid property values are defined as follows:

DOMAIN	recognized and processed by the Event Broker, it is the domain for authenticating the user ID and password that are associated with a process flow. If this property is not specified, then the default domain that is configured with the UserService is used.
IDENTITY	recognized and processed by the Event Broker, it uniquely identifies the message. It enables the client to distinguish among possible responses.
PASSWORD	recognized and processed by the Event Broker, it is the password that is associated with a process flow. For example, if a node in your process flow communicates with IOM, then PASSWORD can be used to authenticate the user who is attempting to access the server.
PRIORITY	recognized and processed by the Event Broker, it specifies the Java priority. The default is 10.
RESPONSE	recognized and processed by the Event Broker, it identifies whether the Event Broker sends an acknowledgment or a result of the event execution. RESPONSE is not supported when publishing the event to subscribers. It is supported only when the event is published to an explicit delivery transport, such as to a queue or to an HTTP server. Valid values are as follows: ACK, which specifies that an acknowledgment message will be sent; NONE, which specifies that no response will be sent; and RESULT, which specifies that a complete result set will be sent.
RESULT_URL	recognized and processed by the CALL routines, it manages results that are returned. If the RESPONSE property is specified with a value of RESULT or ACK, then the event execution returns results or an acknowledgment message, respectively. If a result or acknowledgment is expected, then the RESULT_URL property must be specified. This property is a URL that identifies where to write the results to. At this time, only a URL is supported.
SENT_FROM	recognized and processed by the Event Broker, it is a user-specified text string that identifies where the event was sent from. This property is used by the Event Broker to log the origination of the event message.
USER	recognized and processed by the Event Broker, it is the user ID that is associated with a process flow. For example, if a node in your process flow communicates with IOM, then USER can be used to authenticate the user who is attempting to access the server.

Type: Character or Numeric, Input

Examples

Example 1: Using EVENT_BEGIN

The following example initializes an event and returns the event identifier in *eventId*. No properties are specified.

```
eventId=0;
rc=0;
name = "startEvent";
CALL EVENT_BEGIN(eventId, name, rc);
```

Example 2: Using EVENT_BEGIN with User-Defined Properties

The following example sets two user-defined properties. The user-defined property COMPANY has a value of **Alphalite Airways, Inc.** The user-defined property YEAR has a value of **1993**.

```
name = "Salary";
prop = "Company, Year";
value1 = "Alphalite Airways, Inc";
value2 = "1993";
CALL EVENT_BEGIN(eventId, name,
rc, prop, value1, value2);
```

Example 3: Using EVENT_BEGIN with the PRIORITY Property

The following example sets the well-known property PRIORITY.

```
name = "Sales Figures";
prop = "Priority"
priority = "10";
CALL EVENT_BEGIN(eventId, name,
rc, prop, priority);
```

Example 4: Using EVENT_BEGIN with the SENT_FROM Property and a User-Defined Property

The following example sets a combination of well-known and user-defined properties. It specifies the well-known property SENT_FROM and a user-defined property STATE.

```
name = "Regional Figures";
prop = "sent_From, State";
from = "d1234.us.apex.com";
state = "NC";
CALL EVENT_BEGIN(eventId, name,
rc, prop, from, state);
```

Example 5: Using EVENT_BEGIN with the RESPONSE and RESULT_URL Properties

The following example sets the RESPONSE property to "Result" because results are expected from the event execution. Because the RESPONSE property is specified, the destination for the response must also be specified. Therefore, the RESULT_URL property must also be set to indicate where the response should be written to.

```
name = "Regional Figures";
prop = "Response, Result_Url";
```

```

resp = "Result";
furl = "file:/c:/testsrc/output.xml";
CALL EVENT_BEGIN(eventId, name,
    rc, prop, resp, furl);

```

EVENT_BODY

Sets the body of the event message, which should be specified as a file that contains an XML document fragment

Syntax

CALL EVENT_BODY(*eventId*, *filename*, *rc*);

Required Arguments

eventId

identifies the event.

Type: Numeric, Output

filename

identifies the name of the file that contains the XML fragment that constitutes the event message. The *filename* parameter can be specified as either:

- FILENAME: external_filename
- FILEREF: sas_fileref

Type: Character, Input

rc

identifies the return code.

Type: Numeric, Input

Details

The EVENT_BODY CALL routine can be omitted if the event body is intended to be empty.

Examples

Example 1: Using EVENT_BODY with an XML Fragment

The following example uses an XML fragment that might be defined as the body. This XML fragment is located in the file that is specified by FILENAME or FILEREF in the CALL routine.

```

<Company name='Alphalite Airways'>
  <Sales region='South'>
    <Projection>1000000</Projection>
    <Actual>1000050</Actual>
  </Sales>
  <Sales region='West'>
    <Projection>750000</Projection>
    <Actual>685000</Actual>
  </Sales>

```

```

    <Sales region='North'>
      <Projection>500000</Projection>
      <Actual>600000</Actual>
    </Sales>
    <Sales region='East'>
      <Projection>1000000</Projection>
      <Actual>950000</Actual>
    </Sales>
  </Company>

```

Example 2: Using *EVENT_BODY* with the *FILENAME* Property

The following example uses *FILENAME* to specify the name of the file that contains the body portion of the event.

```

fname = "filename:c:\eventBody.xml";
CALL EVENT_BODY(eventId, fname, rc);

```

EVENT_PUBLISH (Publish Event to HTTP)

Publishes an event using the HTTP protocol

Syntax

```

CALL EVENT_PUBLISH(eventId, publishType, rc, properties,
  <propValue1, ...propValueN> url <url2, ...urlN> );

```

Required Arguments

eventId

identifies the event that is to be published.

Type: Numeric, Input

publishType

indicates how to publish the event. To publish the event by using the HTTP protocol, specify a *publishType* of *TO_HTTP*.

Type: Character, Input

rc

receives a return code.

Type: Numeric, Output

properties

identifies a comma-separated list of optional property names. Specify any of the following property names, or specify double quotation marks to indicate that no properties are to be applied:

- HTTP_PASSWORD
- HTTP_PROXY_URL
- HTTP_USER

Type: Character, Input

url <url2, ...urlN>

identifies the URL(s) that will be used to publish the event.

Type: Character, Input

Optional Argument

propValue1, ...propValueN

specifies one value for each specified property name. The order of the property values must match the order of the property names in the *properties* parameter. A value must be specified for each property that is specified in the *properties* parameter. Valid property values are defined as follows:

HTTP_PASSWORD	specifies the password that is needed to bind to the network resources.
HTTP_PROXY_URL	specifies the URL of the proxy server.
HTTP_USER	specifies the user ID that is needed to bind to the network resources.

Type: Character or Numeric, Input

Details

If you specify multiple URLs, and if you specify the RESPONSE property in the EVENT_BEGIN CALL routine, then the response will be received and processed only for the URL that you specified first. The event is published to the first URL, and the response is written to the RESULT_URL location. For all remaining URLs, the event will be published, but the EVENT_PUBLISH CALL routine will not write the response to the RESULT_URL location. To process results from multiple URLs, issue EVENT_PUBLISH for each URL. Executing an EVENT_PUBLISH for each URL creates an explicit RESULT_URL for each response.

Example: Publishing an Event to HTTP

The following example publishes the event to the network resource by using the HTTP protocol. HTTP URL identifies the machine and port to use.

```
publishType = "TO_HTTP";
url = "http://myhost.com:40";
CALL EVENT_PUBLISH(eventId,
    publishType, rc, '', url);
```

EVENT_PUBLISH (Publish Event to Queues)

Publishes an event to one or more message queues. After EVENT_PUBLISH creates the event and delivers it to a message queue, the Event Broker then retrieves the event and distributes it to other applications. EVENT_PUBLISH supports event delivery to IBM WebSphere MQ and MSMQ destinations.

Syntax

```
CALL EVENT_PUBLISH(eventId, publishType, rc, properties,
    <propValue1, ...propValueN>, queue1 <, ...queueN> );
```

Required Arguments***eventID***

specifies the event that is to be published.

Type: Numeric, Input

publishType

specifies how to publish the event. To publish the event by using the queue transport, specify a *publishType* of TO_QUEUE.

Type: Character, Input

rc

receives a return code.

Type: Numeric, Output

properties

identifies a comma-separated list of optional property names. Specify any of the following property names, or specify double quotation marks to indicate that no properties are to be applied:

- QUEUE_TIMEOUT
- RESPONSE_QUEUE
- SELECTOR

Type: Character, Input

queue1 <, ...queueN>

character string that specifies the queue(s) that will be used to publish the event. When publishing to MSMQ queues, use the following syntax:

MSMQ://queueHostMachineName\queueName

When publishing to IBM WebSphere MQ queues, use the following syntax:

MQSERIES://queueManager:queueName

or

MQSERIES-C://queueManager:queueName

MQSERIES specifies the server interface that is used to connect to a queue manager that is local. MQSERIES-C specifies the client interface that is used, allowing the queue manager to be local or remote.

Type: Character, Input

Optional Argument***propValue1, ...propValueN***

specifies one value for each specified property name. The order of the property values must match the order of the property names in the *properties* parameter. A value must be specified for each property that is specified in the *properties* parameter. Valid property values are defined as follows:

QUEUE_TIMEOUT	specifies the poll time-out value in seconds. If the event expects a result or acknowledgment response, then the response queue identifies the queue to receive the response. By default, the EVENT_PUBLISH CALL routine continues to poll until the response is received or until a 15-second time-out occurs. You can override the default time-out value by specifying an
---------------	--

	explicit queue time-out value. The value must be greater than zero.
RESPONSE_QUEUE	used by the Event Broker, it specifies the name of a queue to receive the result or an acknowledgment. After the event is published, the queue is queried for the response, which is written to a target file that is specified by the RESULT_URL property value.
SELECTOR	specified on the CALL routine, it identifies the name/value properties to define on the event messages that are published. These properties can be used by the Event Broker to determine what messages should be removed from the queue. The Event Broker can be configured so that it removes only messages from the queue that match particular name/value selectors. If the Event Broker configures selectors, then only messages that have properties that match the configured selector are delivered to the Event Broker. Other messages remain on the queue.

Type: Character or Numeric, Input

Details

If you specify multiple queues, and if you specify the RESPONSE property in the EVENT_BEGIN CALL routine, then the response is received and processed only for the queue that you specified first. The event is published to the first queue, and the response is written to the RESULT_URL location. For all remaining queues, the event is published, but the EVENT_PUBLISH CALL routine does not query the response queue for a result. To process results from multiple queues, issue EVENT_PUBLISH for each queue. Executing an EVENT_PUBLISH for each queue creates an explicit RESULT_URL for each response.

Examples

Example 1: Publishing an Event to a Queue

The following example publishes the event to one queue and does not apply any additional queue properties.

```
publishType = "TO_QUEUE";
firstQ = "MQSERIES://PCONE:MYQ";
CALL EVENT_PUBLISH(eventId,
    publishType, rc, '', firstQ);
```

Example 2: Using EVENT_PUBLISH with the RESPONSE_QUEUE Property

The following example publishes the event to one queue. Because a response is expected, the RESPONSE_QUEUE property and a time-out value of 30 seconds are specified. If the response is not received in 30 seconds, a time-out occurs.

```
publishType = "TO_QUEUE";
firstQ = "MQSERIES://PCONE:MYQ";
prop="RESPONSE_QUEUE, QUEUE_TIMEOUT";
respQ = "PCONE:MYQ";
timeout = "30";
```

```
CALL EVENT_PUBLISH(eventId, publishType, rc,
    prop, respQ, timeout, firstQ);
```

Example 3: Using EVENT_PUBLISH with the SELECTOR Property

The following example publishes the event to one queue. The SELECTOR property is used in this example to publish only event messages that are fourth quarter reports from the HR department. The value of the SELECTOR property should be one or more name/value pairs.

```
publishType = "TO_QUEUE";
firstQ = "MQSERIES://PCONE:MYQ";
prop="SELECTOR";
propValue="type=report quarter=four dept=hr";
qName="MQSERIES://QMGR:LocalQ";
CALL EVENT_PUBLISH(packageId, "TO_QUEUE",
    rc, prop, propValue, qName);
```

EVENT_PUBLISH (Publish Event to Subscribers)

Publishes an event to subscribers of the specified channel

Syntax

```
CALL EVENT_PUBLISH(eventId, publishType, rc, properties,
    <propValue1, ...propValueN> , channel);
```

Required Arguments

eventId

specifies the event that is to be published.

Type: Numeric, Input

publishType

indicates how to publish the event. To publish an event to the subscribers of a channel, specify a *publishType* value of TO_SUBSCRIBERS.

Type: Character, Input

rc

receives a return code.

Type: Numeric, Output

properties

specifies the following property name, or specifies double quotation marks to indicate that the property is not to be applied:

- CHANNEL_STORE
- FOLDER_PATH
- SELECTOR

Type: Character, Input

propValue1, ...propValueN

specifies one value for each specified property name. Valid property values are defined as follows:

CHANNEL_STORE	specifies a character string that indicates the SAS Metadata Repository that contains the channel and subscriber metadata. If channel definitions and subscriber definitions are maintained in a SAS Metadata Repository, then the syntax for the CHANNEL_STORE property is as follows: SAS-OMA:// <i>hostname</i> [: <i>port</i>]/ <i>reposname</i> =repositoryName; Where <i>hostname</i> is the name of SAS Metadata Server that contains channel information. HOSTNAME must be a DNS name or IP address of a host that is running a SAS Metadata Server. The <i>port</i> specifies the TCP port of the SAS Metadata Server. If no port is specified, then 8561 is used as a default. The <i>reposname</i> is the name of the repository.
FOLDER_PATH	specifies the folder path for the channel of interest. This value is used to search for channels with specific names that exist in specific folder locations. When a user defines a channel via SAS Management Console, all channels by default exist in the / Channels folder. SAS Management Console allows the user to define multiple folders and subfolders. All FOLDER_PATH properties must start with / Channels and then can identify subfolders if necessary. For example, a channel named "Sales" might be defined in two different folders: / Channels/Reports/US/ or / Channels/Reports/Europe/ .
SELECTOR	specified on the CALL routine, it identifies the name/value properties to define on the event messages that are published. These properties can be used by the Event Broker to determine what messages should be removed from the queue. The Event Broker can be configured so that it removes only messages from the queue that match particular name/value selectors. If the Event Broker configures selectors, then only messages that have properties that match the configured selector are delivered to the Event Broker. Other messages remain on the queue.

Type: Character or Numeric, Input

channel

specifies the name of the channel where the event will be published.

Type: Character, Input

Details

When an event is published to a channel, the event is published to each subscriber of the channel. Each subscriber definition specifies the event publishing transport method to use for the subscriber. Valid transports are HTTP and message queues.

PUBLISH_EVENT ensures that the event is published to each subscriber only once, thus eliminating any duplication. For the message queue transport, the name of the queue is used as the key to enforce uniqueness. For an HTTP server transport, the URL is used as the key to enforce uniqueness.

Publishing an event to subscribers does not support the RESPONSE property.

Example: Publishing an Event to Subscribers

The following example publishes the event to all subscribers of the WeeklyPayroll channel:

```
channelStore =
    "SAS-OMA://alpair03.sys.com:4059";
channelName = "WeeklyPayroll";
prop = "CHANNEL_STORE,METASUSER,METAPASS";
user = "myUserName";
password = "myPassword";
CALL EVENT_PUBLISH(eventId, "TO_SUBSCRIBERS",
    rc, prop, channelStore, user, password, channelName);
```

EVENT_END

Frees all resources that are associated with the specified event

Syntax

```
CALL EVENT_END(eventId, rc);
```

Required Arguments

eventId

identifies the event that is to be published.

Type: Numeric, Input

rc

receives a return code.

Type: Numeric, Output

Details

Freeing resources closes all queues and files that are associated with the specified event.

Example: Using EVENT_END

The following example frees all resources that are associated with the specified event:

```
CALL EVENT_END(eventId, rc);
```

Examples

Example 1: Explicitly Generated Event

In the following example, a company's sales information is reported in an explicitly generated event.

```

<sas-event:Event xmlns:sas-event=
  "http://support.sas.com/xml/namespace/services.events-1.1"
  sas-event:name="event1">
  <sas-event:Header>
    <sas-event:Version>1.0</sas-event:Version>
    <sas-event:Identity>
      7FBBA000-32C4-11D6-8001-363139363230
    </sas-event:Identity>
    <sas-event:Response type="result"/>
    <sas-event:Properties>
      <Company>Alphalite Airways</Company>
    </sas-event:Properties>
  </sas-event:Header>
  <sas-event:Body>
    <Company name="Alphalite Airways">
      <Sales region="South">
        <Projection>1000000</Projection>
        <Actual>1000050</Actual>
      </Sales>
      <Sales region="West">
        <Projection>750000</Projection>
        <Actual>685000</Actual>
      </Sales>
      <Sales region="North">
        <Projection>500000</Projection>
        <Actual>600000</Actual>
      </Sales>
      <Sales region="East">
        <Projection>1000000</Projection>
        <Actual>950000</Actual>
      </Sales>
    </Company>
  </sas-event:Body>
</sas-event:Event>

```

Example 2: Implicitly Published Event Containing an External File and a Reference

In the following example, the published package contains an external file and a reference. Because the package is published to a WebDAV server, the `sas-publish:packageUrl` attribute is specified. This attribute is a URL to an archived package. The package was published with a name/value specification of "report=revenue department=research".

```

<?xml version="1.0" encoding="UTF-8"?>
<sas-event:Event xmlns:sas-event=
  "http://support.sas.com/xml/namespace/services.events-1.1"
  sas-event:name='SASPackage.AirlineChannel'>
  <sas-event:Header>
    <sas-event:Version>1.0<sas-event:Version>
    <sas-event:SentAt>26SEP2001:19:15:37</sas-event:SentAt>
  </sas-event:Header>
  <sas-event:Body>
    <sas-publish:Package
      xmlns:sas-publish=

```

```

"http://support.sas.com/xml/
  namespace/services.publish-1.1"
sas-publish:version="1.0"
sas-publish:description="Revenue Info"
sas-publish:channel="Revenue Channel"
sas-publish:packageUrl=
  "http://alphaliteAirways.com/
    revenue/reports/2001/quarter3"
report="revenue"
department="research">
<sas-publish:Entries>
  <sas-publish:Entry sas-publish:type="file"
    sas-publish:description="Revenue graph">
    <sas-publish:File
      sas-publish:type="binary"
      sas-publish:name="revenue.gif"
      sas-publish:mimetype="image/gif" />
    </sas-publish:Entry>
  <sas-publish:Entry sas-publish:type="reference"
    sas-publish:description="Revenue details.">
    <sas-publish:Reference sas-publish:type="html"
      sas-publish:reference=
        "http://www.alphaliteAirways.com/revenue.html" />
    </sas-publish:Entry>
  </sas-publish:Entries>
</sas-publish:Package>
</sas-event:Body>
</sas-event:Event>

```

Example 3: Implicitly Published Event Containing a SAS Data Set, Text File, and HTML File

In the following example, the published package contains a SAS data set, an external text file, and an HTML file. Because the published package is not archived, the `sas-publish:packageUrl` attribute is not specified. The SAS data set is defined by using the name/value specification of "quarter=third region=south quarterly". The HTML file contains a body, a frame, the contents, a page, and a companion file.

```

<?xml version="1.0" encoding="UTF-8"?>
<sas-event:Event xmlns:sas-event=
  "http://support.sas.com/xml/namespace/services.events-1.1"
  sas-event:name='SASPackage.ReportChannel'>
<sas-event:Header>
  <sas-event:Version>1.0</sas-event:Version>
  <sas-event:SentAt>26SEP2001:19:15:37
  </sas-event:SentAt>
</sas-event:Header>
<sas-event:Body>
  <sas-publish:Package version="1.0"
    xmlns:sas-publish=
      "http://support.sas.com/xml/
        namespace/services.publish-1.1"
    sas-publish:version="1.0"
    sas-publish:Description="Sales Reporting Data"
    sas-publish:Abstract="Data necessary to create and

```



```
manage the Sales reports."
sas-publish:Channel="SalesChannel">
<sas-publish:Entries>
  <sas-publish:Entry sas-publish:type="dataset"
    sas-publish:Description="Employee
    information data set"
    quarter="third" region="south"
    quarterly="">
    <sas-publish:Dataset
      sas-publish:name="SalesData"/>
  </sas-publish:Entry>
  <sas-publish:Entry sas-publish:type="file"
    sas-publish:description="Defects SAS job.">
    sas-publish:File sas-publish:type="text"
      sas-publish:name="defects.sas"
      sas-publish:mimetype="application/sas"/>
  </sas-publish:Entry>
  <sas-publish:Entry sas-publish:type="html"
    sas-publish:description="ODS
    generated HTML.">
    <sas-publish:HTML sas-publish:type="body"
      sas-publish:name="body.html"
      sas-publish:url="body.html"/>
    <sas-publish:HTML sas-publish:type="frame"
      sas-publish:name="frame.html"
      sas-publish:url="frame.html"/>
    <sas-publish:HTML
      sas-publish:type="contents"
      sas-publish:name="contents.html"
      sas-publish:url="contents.html"/>
    <sas-publish:HTML sas-publish:type="page"
      sas-publish:name="page.html"
      sas-publish:url="page.html"/>
    <sas-publish:Companion
      sas-publish:name="graph.gif"
      sas-publish:url="graph.gif"
      sas-publish:mimetype="image/gif"/>
  </sas-publish:Entry>
</sas-publish:Entries>
</sas-publish:Package>
</sas-event:Body>
</sas-event:Event>
```


Appendix 1

Planning and Implementing Your Publishing Solution

Plan the Information Architecture	173
Establish the Content Pipeline	174
Configure Channels and Subscribers	175
Implement Content Restrictions in the SAS Metadata Authorization Layer . . .	175
Announce Solution and Train Users	176

Plan the Information Architecture

Designing a successful publish and subscribe implementation starts with an understanding of why your organization is implementing the system. You will need to know what type of information needs to be distributed to users and how widely that information needs to be distributed. The two main considerations in planning are efficiency (helping users to avoid information overload) and security (enforcing any site-required content restrictions).

For example, you could start the planning process by understanding that your organization needs to disseminate sales information throughout the marketing organization and inventory data to the production organization. Starting with this knowledge, you can begin the process of breaking down the general categories of information into specific information channels by using a hierarchical model.

How you divide and subset the categories depends on your organization's needs, but you should work toward creating information channels as focused as possible, without making them too tightly focused to be useful. Channels that are broadly defined leave users not knowing whether information that is delivered over the channel will be useful to them. Channels that are too precisely defined force users to subscribe to a long list of channels in order to ensure that they receive the information that they need.

To help focus the information that users receive, set up policies for name/value keywords. Name/value pairs are attributes that are specified when a package is published and that help to identify the package contents. Each subscriber definition can include a name/value filter that allows only packages that meet the subscriber's needs to be delivered.

For example, if you publish a package with a name/value attribute of **market= (Mexico)**, that package is seen only by those subscribers whose name/value filter indicates that they are interested in information about the Mexican market. Although the names and associated values can be anything that your organization finds useful, you must establish a list of acceptable keywords and values for those keywords.

This list is essential in order for publishers to be able to provide consistent metadata that identifies published content and for subscribers to be able to filter published content in order to focus on the information that they need.

When you define your information channels, you must also consider the users that will be accessing those channels as well as any restrictions that need to be placed on the channels. Although these aspects of planning are discussed separately, in practice, they are examined at the same time as you are defining your channels. You cannot define an information channel without first knowing who needs to see the information and how that information should be restricted.

The hierarchical model that you use can be based on both subject and access level. For example, it is often appropriate to use group or department-level distinctions. Identify any channels that must be restricted for either who can contribute or who can subscribe. Restrictions are defined on channels, so do not mix access levels within a channel (for example, do not include sensitive and non-sensitive content in a channel). For example, if you plan for a single channel to distribute accounting information throughout your organization, you will encounter a security problem when the accounting department needs to publish sensitive information (such as employee salaries). With only a single, unrestricted channel, you cannot publish the information to a specific set of users. In your consultations with users, you must identify information channels whose access needs to be controlled.

Establish the Content Pipeline

To establish the content pipeline, perform the following steps:

1. Develop or modify applications that will be used to create the content to be published. These applications can take the form of stand-alone applications that are written in a visual programming language or SAS programs. Publishers must obtain or install the appropriate publishing application for their needs. For example, an individual or department that needs to publish data-intensive reports on a regular basis might use a SAS program for publishing, while a user who needs to send information to a changing number of users on an occasional basis might use the publishing functionality that is provided by SAS Enterprise Guide or SAS Information Delivery Portal.
2. For the initial set of information channels, identify the users and groups that are initially subscribed to those channels. If the publishing framework has open access, then users can subscribe themselves to channels. Otherwise, administrators can define the subscribers for each channel.
3. Determine how information is to be distributed to subscribers (whether by text-formatted e-mail or HTML-formatted e-mail, with a WebDAV server, or through a queue).
4. Gather address information, which is necessary for defining subscribers.
5. Create a PUBLISHERS group, and enable the PUBLISHERS group to authenticate to the content server (if it is a secured HTTP, FTP, or WebDAV server). Credentials can be included in your code or stored in metadata. The following example scenarios all require the publisher to have server credentials:
 - publishing to a subscriber with a delivery transport that is defined as a secured WebDAV server
 - publishing to a channel's persistent store that is defined as a secured WebDAV server

- publishing to a channel's persistent store that is defined as an archive path that is a secured HTTP server
- publishing to a channel's persistent store that is defined as an archive path that is a secured FTP server

Note: Token authentication is supported, beginning with SAS 9.2. For more information about SAS token authentication, see *SAS Intelligence Platform: Security Administration Guide*.

It is usually most efficient to create one metadata group that includes all publishers as members and give that group one login for each secured HTTP, FTP, or WebDAV server. Each server must be registered in the metadata in its own authentication domain. For example, the contents of the group's **Accounts** tab might as shown in the following table:

Table A1.1 Credentials for Group

Authentication Domain	User ID	Password
IISauth	sharedIISid	sharedIISpassword
FTPAuth	sharedFTPid	sharedFTPpassword

Note: If you publish directly to subscribers who have their own WebDAV servers, each of those servers must be registered in its own authentication domain. The group's **Accounts** tab must include a login for each such server. For more information about credential management, see *SAS Intelligence Platform: Security Administration Guide*.

Configure Channels and Subscribers

Use the New Subscriber and New Channel wizards in the Publishing Framework plug-in for SAS Management Console to define the channels and subscribers that you identified during the planning phase. Begin by defining the subscribers; the New Channel wizard enables you to associate defined subscribers to a channel. For more information, see the help for the Publishing Framework plug-in.

Implement Content Restrictions in the SAS Metadata Authorization Layer

You can implement content restrictions in the SAS metadata authorization layer in order to:

- control who can publish to a channel
- control who can create a new channel
- control who can self-subscribe to a channel

For more information about authorization and permissions by task, see *SAS Intelligence Platform: Security Administration Guide*.

Note: In a new deployment, only the SAS Administrators group can add channels, subscribers, and content. To enable all registered users to update and publish to a particular channel, navigate on the **Folders** tab to **System** ⇒ **Publishing** ⇒ **Channels** and grant W and WM to SASUSERS on that channel's **Authorization** tab (WM is required to publish only if a channel has an archive persistent store). To enable all registered users to add channels or subscribers, grant WMM on the relevant parent folder (for example, on the **System** ⇒ **Publishing** ⇒ **Subscribers** ⇒ **Content Subscribers** folder). For more information about using permissions, see the *SAS Intelligence Platform: Security Administration Guide*.

Announce Solution and Train Users

After the publishers and subscribers install the necessary applications, you can announce your implementation to your organization. You will also need to follow up the announcement with training for both publishers and subscribers, with training broken down by publishing methods, publishing needs, and subscriber applications.

Index

A

archive path properties [80, 93, 101, 108](#)
 archive transports [7](#)
 archives
 publishing packages to [71](#)
 retrieving packages from [11](#)

C

catalogs
 inserting into packages [50](#)
 retrieving from packages [121](#)
 channel definition [2](#)
 channel subscribers
 publishing to, and viewers [16](#)
 channel transports [7](#)
 channels
 configuring [175](#)
 creating [8](#)
 publishing to [100](#)
 COMPANION_NEXT CALL routine
 [111](#)
 companion HTML files
 retrieving next file in ODS HTML set
 [111](#)
 content pipeline [174](#)
 content restrictions [175](#)

D

data set entries
 retrieving from packages [122](#)
 data sets
 creating viewers and [17](#)
 extracting and formatting into tables [18](#)
 extracting and formatting variables from
 data set to list [18](#)
 inserting into packages [51](#)
 DATA step
 package publishing in [139](#)
 descriptive header information

retrieving for all packages [130](#)
 retrieving for nested package entries
 [129](#)

E

e-mail
 publishing packages to [73](#)
 retrieving packages from [12](#)
 e-mail transport [8](#)
 publishing to, and viewers [16](#)
 electronic newsletters
 publishing, and viewers [16](#)
 ENTRY_FIRST CALL routine [113](#)
 ENTRY_NEXT CALL routine [114](#)
 entry-type filters [45](#)
 EVENT_BEGIN CALL routine [158](#)
 EVENT_BODY CALL routine [161](#)
 EVENT_END CALL routine [168](#)
 EVENT_PUBLISH CALL routine [162,](#)
 [163, 166](#)
 publishing event to HTTP [162](#)
 publishing event to queues [163](#)
 publishing event to subscribers [166](#)
 event identifiers [158](#)
 event publishing [3](#)
 events [149](#)
 body of event message [161](#)
 defined [149](#)
 example of a generated event [168, 169,](#)
 [170](#)
 explicit event publication [150](#)
 freeing resources [168](#)
 generating and publishing [150](#)
 implicit event publication [150](#)
 initializing [158](#)
 methods for publishing [149](#)
 Publish Event Interface [151](#)
 publishing to HTTP [162](#)
 publishing to message queues [163](#)
 publishing to subscribers [166](#)

- publishing with HTTP protocol 162
- XML specification for 150
- XML specification for generic events 151
- XML specification for SASPackage events 153
- executive level summaries
 - publishing, and viewers 16
- explicit event publication 149, 150
- external binary files
 - retrieving from packages 124
- extracting and formatting
 - data sets into tables 18
 - SAS data and viewers 16
 - variable from data set to list 18

F

- file encoding 44
 - default publish and retrieve behavior 43
 - rules for determining 43
 - specifying on the retrieve 44
- file extensions
 - for item types 81, 92, 102, 108
- filename extensions
 - for package entry types 6
- files
 - inserting into packages 54
 - retrieving from packages 124
- filtering 44
 - enabling, when publishing packages 45
 - entry-type filters 45
 - MIME-type filters 45
 - name/value filters 45
 - overview 44
 - package entries 21
 - packages and package entries 44
 - SASINSERT tag and 26
- formatting packages, and viewers 16
- FTP access method
 - package publishing with 145

G

- generated events
 - explicitly generated 168
 - implicitly published 169, 170

H

- header information
 - for first package entry 113
 - for first package in package list 117
 - for next package entry 114
 - for next package in package list 119
 - retrieving for all packages 130

- retrieving for nested package entries 129
- HTML entries
 - retrieving from packages 125
- HTML files
 - inserting into packages 57
 - retrieving next companion file in ODS HTML set 111
- HTML protocol
 - publishing events with 162
- HTML tables and lists
 - populating 28
- HTML viewer 32
 - sample SAS program with 33

I

- implicit event publication 150
- information architecture 173
- INSERT_CATALOG CALL routine 50
- INSERT_DATASET CALL routine 51
- INSERT_FILE CALL routine 54
- INSERT_HTML CALL routine 57
- INSERT_MDDDB CALL routine 61
- INSERT_PACKAGE CALL routine 62
- INSERT_REF CALL routine 64
- INSERT_SQLVIEW CALL routine 65
- INSERT_VIEWER CALL routine 66
- IOM servers 41
- item types
 - file extensions for 81, 92, 102, 108

L

- lists
 - building with SASINSERT and SASTABLE tags 31
 - extracting and formatting variables from data set to 18
- log
 - storing a text string for 31

M

- macros
 - package publishing in 142
- MDDDB entries
 - retrieving from packages 128
- message queue transports 8
- message queues
 - publishing events to 163
 - publishing packages to 83
 - retrieving packages from 11
- MIME-type filters 45
- multidimensional database
 - inserting into packages 61

N

name/value filters 45
 operators and 46
 name/value pairs 48
 applying to packages and package items 93, 109
 specifying 49
 specifying for entire package 49
 specifying for package items 48
 nested package entries
 retrieving descriptive header information for 129

O

ODS HTML set
 retrieving next companion file in 111
 operators
 name/value filters and 46

P

PACKAGE_BEGIN CALL routine 68
 PACKAGE_DESTROY CALL routine 116
 PACKAGE_END CALL routine 71
 PACKAGE_FIRST CALL routine 117
 PACKAGE_NEXT CALL routine 119
 PACKAGE_PUBLISH CALL routine 71, 73, 83, 86, 94, 103
 publishing package to archive 71
 publishing package to e-mail 73
 publishing package to queues 83
 publishing package to SharePoint 86
 publishing package to subscribers 94
 publishing package to WebDAV-compliant server 103
 PACKAGE_TERM CALL routine 121
 package entries
 filtering 21
 header information for first entry 113
 header information for next entry 114
 package entry types 5
 filename extensions for 6
 SAS results 5
 unstructured content 6
 package identifiers 68
 package items
 name/value pairs and 48, 93, 109
 package list
 header information for first package in 117
 header information for next package in 119
 package list identifier
 freeing resources associated with 121

package publishing 2, 40
 examples 139, 142, 145
 filtering packages and package entries 44
 in DATA step 139
 in macros 142
 methods for different publishers 40
 process of 41
 publish and retrieve encoding behavior 44
 Publish Package Interface and 41
 specifying name/value pairs 49
 with FTP 145
 with third-party client application 41
 package retrieval and viewing 2
 package transports 9
 persisted packages 8
 subscription channels 8
 packages 6
 creating viewers and 17
 deleting 116
 filtering 44
 formatting, and viewers 16
 freeing resources 71
 header information for first package in package list 117
 header information for next package in package list 119
 initializing 68
 inserting catalogs into 50
 inserting data sets into 51
 inserting files into 54
 inserting HTML files into 57
 inserting into packages 62
 inserting multidimensional database into 61
 inserting PROC SQL views into 65
 inserting references into 64
 inserting viewers into 66
 name/value pairs, applying to 93, 109
 name/value pairs for entire package 49
 persisted 8
 publishing to archives 71
 publishing to e-mail 73
 publishing to message queues 83
 publishing to SharePoint 86
 publishing to subscribers 94
 publishing to WebDAV-compliant servers 103
 rendering 6
 retrieving catalogs from 121
 retrieving data set entries from 122
 retrieving descriptive header information for all packages 130
 retrieving external binary files from 124

- retrieving from different transports 11
- retrieving HTML entries from 125
- retrieving MDDB entries from 128
- retrieving PROC SQL views from 137
- retrieving references from 136
- retrieving text files from 124
- retrieving viewer entries from 138
- simulated rendered view of package in e-mail 36
- persisted packages 8
- PROC SQL views
 - inserting into packages 65
 - retrieving from packages 137
- publish and retrieve encoding behavior 44
 - default behavior 43
 - rules for determining file encoding 43
 - specifying encoding on the retrieve 44
- Publish Event Interface 3, 151
- Publish Package Interface 22, 41
- publishing 16
 - electronic newsletters, and viewers 16
 - executive level summaries, and viewers 16
 - to channel subscribers 16
 - to channels 100
 - to e-mail transport 16
- Publishing Framework 1

Q

- queues
 - publishing events to 163
 - publishing packages to 83

R

- reference URLs
 - streaming text files and 20
- references
 - inserting into packages 64
 - retrieving from packages 136
- rendering packages 6
- restricted content 175
- RETRIEVE_CATALOG CALL routine 121
- RETRIEVE_DATASET CALL routine 122
- RETRIEVE_FILE CALL routine 124
- RETRIEVE_HTML CALL routine 125
- RETRIEVE_MDDB CALL routine 128
- RETRIEVE_NESTED CALL routine 129
- RETRIEVE_PACKAGE CALL routine 130
- RETRIEVE_REF CALL routine 136

- RETRIEVE_SQLVIEW CALL routine 137
- RETRIEVE_VIEWER CALL routine 138
- retrieving packages 11
 - SAS Package Reader 13
 - SAS Package Retriever 12
- retrieving URLs 13

S

- SAS Integration Technologies Publishing Framework 1
- SAS metadata authorization layer
 - content restrictions in 175
- SAS Package Reader 2, 13
- SAS Package Retriever 2, 12
- SAS results 5
- SASECHO tag 31
- SASINSERT tag 23
 - building a list 31
- SASPackage events 153
- SASREPEAT tag 30
- SASTABLE tag 28
 - building a list 31
- SharePoint
 - publishing to 86
- SharePoint transport 8
- streaming text files and reference URLs 20
- subscribers
 - configuring 175
 - creating 9
 - publishing events to 166
 - publishing packages to 94
- subscription channels
 - creating channels 8
 - creating subscribers 9
 - creating subscriptions 9
- subscription management 2
- subscriptions
 - creating 9
- substitutions in tags 30

T

- tables
 - extracting and formatting data sets into 18
- tag substitutions 30
- text files
 - retrieving from packages 124
 - streaming text files and reference URLs 20
- text strings
 - storing for log 31

- text tables and lists
 - populating 28
- third-party client applications 41
- token authentication 175
- training users 176
- transports
 - retrieving packages from 11

U

- unstructured content 6
- URLs
 - retrieving 13
 - streaming text files and reference URLs 20
- user training 176

V

- variables
 - extracting and formatting from data set to list 18
- viewer entries
 - retrieving from packages 138
- viewer files
 - marking for viewer processing 23
- viewer properties 80, 92, 100, 108
- viewer templates 35
- viewers 15

- applying with Publish Package Interface 22
- building a list with SASINSERT and SASTABLE tags 31
- creating 21
- creating basic viewers 17
- inserting into packages 66
- sample HTML viewer 32
- sample SAS program with HTML viewer 33
- sample viewer template 35
- when to use 16

W

- WebDAV properties 81, 102
- WebDAV-compliant server transports 8
- WebDAV-compliant servers
 - publishing to 103
 - retrieving packages from 11

X

- XML
 - specification for events 150
 - specification for generic events 151
 - specification for SASPackage events 153

