Axeda® Platform

Axeda® Agent Embedded Toolkit

Reference

Version 6.1, Build 364
updated January 2012

Axeda Corporation
25 Forbes Boulevard, Suite 3
Foxborough, MA 02035
USA
(508) 337-9200

Axeda® ServiceLink™ and Axeda® Platform software and services (“Axeda Products”) and Questra® IDM software (“IDM Software”) are protected by contract law, copyright laws, and international treaties. Axeda Products and IDM Software are supplied under license and/or services contracts with Axeda’s customers and only users authorized under the applicable contract are permitted to access and use the Axeda Products and IDM Software. Unauthorized use and distribution may result in civil and criminal penalties.

Use, duplication, or disclosure of Axeda software by the U.S. government is subject to FAR 12.211 and 12.212 which state that Government shall acquire only the technical data and the rights in that data customarily provided to the public with a commercial item or process and commercial computer software or commercial computer software documentation shall be acquired by the Government under licenses customarily provided to the public to the extent such licenses are consistent with Federal law.

Portions of the Axeda Products include one or more open source or other third party software programs. Please refer to the Open_Source_License_Requirements.pdf included in your Axeda product(s) for important notices and licensing information related to such programs. This information is also available through the Axeda Support Site for authorized customers.

The information in this manual is subject to change without notice and does not represent a commitment on the part of Axeda Corporation. This manual and the accompanying Axeda products are copyrighted by Axeda Corporation and contain proprietary and confidential information of Axeda Corporation. All rights are reserved. Any reproduction and/or distribution without the prior written consent from Axeda Corporation is strictly prohibited, except as permitted under the contract between your company and Axeda Corporation. Please refer to the contract for details.

"Axeda" is a registered trademark and the "Axeda" logo is a trademark of Axeda Corporation. “Questra” is a registered trademark of Axeda Corporation. Axeda ServiceLink, Safe Access, Firewall-Friendly, and Maximum Support are trademarks or service marks of Axeda Corporation. Questra IDM Application Suite, Questra RemoteService, Questra SmartMonitor, Questra SoftwareDirector, Questra SoftwareCourier, Questra TotalAccess, A2B, and Asset-to-Business are trademarks of Axeda Corporation.

All third party brand or product names are trademarks or registered trademarks of their respective companies or organizations and are hereby acknowledged, including without limitation the following:

Microsoft, .Net logo, Access (database software), Active Desktop, Active Directory, Internet Explorer, the Microsoft Internet Explorer logo (graphic only), SQL Server, Terminal Services RemoteApp, Visual C++, Visual InterDev, Visual Studio, Visual Studio logo (graphic only), Win32, Windows, the Windows logo (aka the flag logo, graphic only), Windows NT, Windows start button, Windows Start logo (design), and Windows Vista are registered trademarks of Microsoft Corporation. Sun, Solaris, iPlanet, Java, and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. Axeda Corporation is independent of Sun Microsystems, Inc. Oracle and Siebel are registered trademarks of Oracle Corporation and/or its affiliates. IBM and Cognos are registered trademarks of International Business Machines, Incorporated, in the United States and other countries. Linux is a trademark of Linus Torvalds. RED HAT and JBOSS are registered trademarks of Red Hat, Inc. and its subsidiaries in the US and other countries. SuSE Linux is a trademark of SuSE, Inc. Apache Tomcat and Tomcat are trademarks of the Apache Software Foundation. VxWorks is a registered trademark of Wind River Systems, Inc. Netscape is a registered trademark of Netscape Communications Corporation in the U.S. and other countries. Navigator is also a trademark of Netscape Communications Corporation and may be registered outside the U.S. Accelerated Technology and Nucleus are registered trademarks of Accelerated Technology, Inc. ThreadX is a registered trademark of Express Logic, Inc. NetSilicon and NET+OS are trademarks of NetSilicon, Inc. ARM is a registered trademark of ARM Limited. This product includes software developed by the OpenSSL Project for use in the OpenSSL toolkit (http:// www.openssl.org/). Contains software developed by the University of California, Berkeley and its contributors. All other brand or product names are trademarks or registered trademarks of their respective companies or organizations and are hereby acknowledged.

Document Number: AgentEmbedded/Reference/6.1B364/0112

Table of Contents

Chapter 1, Using This Reference

Chapter 2, Introduction to Axeda® Gateway and Axeda® Connector Agents

Chapter 3, Using Agent Embedded 3-1

Chapter 4, Agent Embedded Operations 4-1

Chapter 5, Function Reference 5-1

Chapter 1
Using This Reference

This chapter contains the following major sections:

- ***About This Reference*** explains the intended audience of this reference and summarizes the contents of each chapter.
- ***Getting Help*** explains where to find answers to your questions about Axeda® Agent Embedded.
- ***Documentation Feedback*** explains how to help us improve Axeda documentation.

About This Reference

This section describes the intended audience for this reference (including assumptions about prerequisites), presents a summary of the contents of this reference, and definitions of terms used frequently.

Audience

This manual is intended as a reference for software developers when creating embedded agents for their products that will operate with the Axeda Platform.

Prerequisites

Using Agent Embedded requires an advanced understanding of ANSI C and the asset application that will communicate with the Axeda® Enterprise Server. You must have experience with the operating system on which Agent Embedded will run as well as the development tools for creating loadable libraries.

Contents

This reference contains the following chapters and appendices:

Chapter 1, "Using This Reference"**,** describes what information is contained in this manual and how to use this manual.

Chapter 2, "Introduction to Axeda® Gateway and Axeda® Connector Agents", presents an overview of the main features of Axeda Agents and explains the subset of features that are available through this toolkit.

Chapter 3, "Using Agent Embedded", provides general instructions for using the APIs of this toolkit.

Chapter 4, "Agent Embedded Operations", explains in more detail how your device application can use the APIs of this toolkit to initiate communications with the Axeda Enterprise Server. This chapter lists the SOAP methods that Agent Embedded processes, explains the rules for SOAP method callbacks, and explains synchronous versus asynchronous operations. Finally, this chapter provides information about the interaction of Agent Embedded with the operating systems of devices and thread safety.

Chapter 5, " Function Reference", lists and describes all the APIs available with the Agent Embedded Toolkit.

An index completes this reference.

Definition of Terms

Throughout this document, the term “Agent” refers to the embedded agent that you create using the Axeda® Agent Embedded Toolkit product.

Throughout the Axeda Agent Embedded Toolkit, you will see the term “Device”, which refers to the item, asset, or equipment that runs your Agent Embedded agent and application. For Axeda releases after version 5.3, the term “asset” replaces the term “device” and the term “Configuration application” replaces the term “Device application” in the Axeda® Connected Product Management Applications (hereafter referred to as “Axeda Applications”).

To ensure backwards-compatibility with existing Agent Embedded agents, the classes and methods operations available prior to v5.3 have not changed and continue to identify these managed items as “devices”. This document uses the terms “device” and “asset” interchangeably.

Getting Help

If you still have questions after reading the documentation for your Axeda product, you can contact Axeda at http://help.axeda.com for help or more information.

Documentation Feedback

As part of our ongoing efforts to produce effective documentation, Axeda asks that you send us any comments, additions or corrections for the documentation. Your feedback is very important to us and we will use it to improve our products and services.

Please send your comments to documentation@axeda.com. Thank you for your help.

Chapter 2
Introduction to Axeda® Gateway and Axeda® Connector Agents

This chapter provides an overview of the features of Axeda Gateway and Axeda Connector Agents and then explains the subset of these features that Axeda Agent Embedded supports. It also provides an overview of the Agent Embedded APIs. This chapter contains the following sections:

- ***Axeda Gateway and Axeda Connector*** provides a brief overview of Axeda Agents.
- ***Connectivity Support*** shows the features of Axeda Agents that are supported by Agent Embedded.
- ***Overview of the Agent Embedded Toolkit*** provides a brief overview of Agent Embedded.
- ***How Agent Embedded Works*** explains how Agent Embedded communicates with your asset application and the Axeda Enterprise Server, enabling you to monitor and manage your assets through the Axeda Applications.
- ***Components of the Agent Embedded Toolkit Library*** explains the components of the Agent Embedded library.
- ***Agent Embedded APIs*** explain the categories of APIs and the purpose of each group of functions.

Axeda Gateway and Axeda Connector

The Axeda Gateway and Axeda Connector Agents provide intelligence at the asset level within the distributed architecture of the Axeda Enterprise Server. These Axeda Agents are modular software servers that provide two-way, Firewall-Friendly™ monitoring, communications, and control of asset data and events in real time. Each Agent provides a physical connection to an asset to communicate process data between the asset and the Enterprise Server. These agents can perform a wide variety of operations beyond simple data collection and server communications. Two types of these Agents are available:

- Embedded agents that you create using the Agent Embedded Toolkit, which is a library of portable C-code that can be embedded in small footprint systems.
- Axeda Connector and Axeda Gateway, which are full-featured servers that provide local intelligence on larger systems.

When the full range of functionality provided by the Agents is not needed and more than the application requires, the Axeda Agent Embedded Toolkit provides the solution. These Agents have been developed to operate on different platforms and operating systems. For information on the supported platforms, go to http://help.axeda.com.

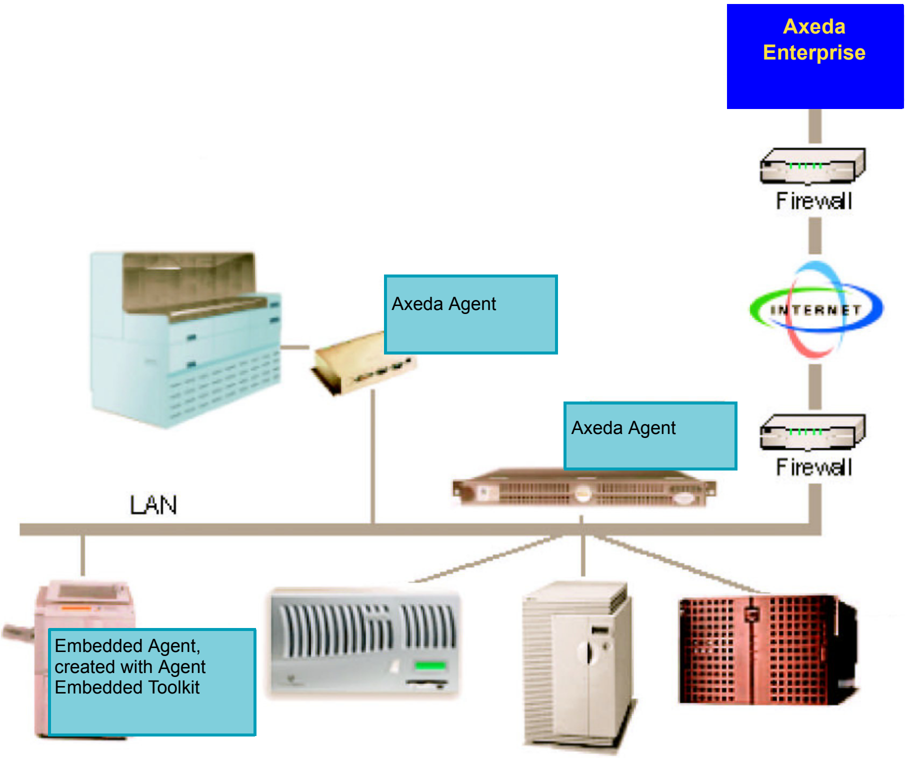

Figure 2-1. Axeda Agents communicating with Axeda Enterprise Server

Like their full-featured counterparts (Axeda Connector and Axeda Gateway), embedded agents provide two-way, Firewall-Friendly monitoring, communications, and control of asset data and events in real time. Included in these services are the abilities to establish and maintain secure communication between the Axeda Enterprise Server and the asset.

Operating as a middleman, Agent Embedded brokers the communications between the asset and the Axeda Enterprise Server. The Agent Embedded is integrated with an existing asset application to enable the asset to communicate its data, events, and alarms to the Enterprise Server, and then receive commands back for execution on the asset.

Using this toolkit, you can develop solutions for any 8-, 16-, or 32- bit processor assets, on most platforms or operating systems. You can easily port Agent Embedded to platforms that have C compilers and an implementation of the TCP/IP stack because this library is implemented in ANSI C and not bound to any platform-specific functionality.

Connectivity Support

The following table illustrates the different connectivity features that both types of Axeda Agents support. This book explains in full the features that Agent Embedded supports. Use this table as a reference to the similarities and differences between the Agents.

Table 2-1. Connectivity Features of Agent Embedded and Full Agents

Feature	Axeda Gateway and Axeda Connector	Agent Embedded
Firewall-Friendly^{TM} communication	yes	yes
Support for 8-, 16-, and 32-bit processors, small footprint	no	yes
Support for 32-bit processors only	yes	no
Linkable with embedded asset application	no	yes
Standards based encryption, 40-bit through 128-bit	yes	yes
Axeda® Access and Axeda® Desktop support	yes	yes (all features except auditing of remote sessions are supported)
Axeda® Policy Server support	yes	no
Large File transfer support	yes	yes
Axeda® Software Management support	yes	Agent Embedded supports file upload, download, and restart instructions only. Agent Embedded does NOT support rollback instructions nor does it support wildcards in the file specification. Agent Embedded does support automatic deployments.
Remote Sessions	yes	yes
Remote script execution	yes	no

Table 2-1. Connectivity Features of Agent Embedded and Full Agents *(continued...)*

Feature	Axeda Gateway and Axeda Connector	Agent Embedded
Intelligent data processing components	yes	no
Self update capability	yes	no
Customization through Axeda® Builder	yes	no
Connect to multiple managed assets	yes (Gateway only)	yes
Automatic discovery of managed assets	yes (Gateway only)	yes

Overview of the Agent Embedded Toolkit

The Agent Embedded Toolkit provides APIs for the basic functionality needed to connect assets with the Axeda Enterprise Server. It also provides APIs for a few additional operations required or useful for your Axeda® Connected Product Management Applications (Axeda Applications). This library extends your own asset applications to communicate with and be monitored by the Enterprise Server, based on Axeda Corporation's Firewall-Friendly technology. Intentionally small to accommodate assets with footprint limitations, this library hides the complexity of network data transfers and provides a simple programming interface to enable custom functionality. Using this toolkit, you can create embedded agents for most platforms and footprints.

The Agent Embedded Toolkit library enables asset applications to send asset data to the Enterprise Server and process commands received from the server. Agent Embedded can post data as HTTP requests in eMessage XML format. The Axeda Enterprise Server sends HTTP responses as SOAP method-formatted commands to Agent Embedded.

For example, the library includes support for sending asset data to the Enterprise Server and for setting callbacks invoked on asset registration. How you implement the asset application depends on your asset and process needs and the functionality supported by your asset's operating system.

How Agent Embedded Works

Your asset has information that you want to make available, for example, to remote service technicians. You want to be able to monitor that asset and its activities. In addition, you want to manage it so that, for example, you can prevent downtime by performing routine maintenance. The Agent Embedded Toolkit provides APIs to which your asset application can link for communicating data to and receiving commands from the Axeda Enterprise Server. To link your asset application with Agent Embedded, you provide implementations for certain OS-dependent functions. Then, during runtime, your asset pushes its raw data to Agent Embedded, which formats the data into XML messages and delivers them, through HTTPS (secure, using SSL) or HTTP (non-secure), to the Enterprise Server. Examples of XML messages include data, alarms, events, and asset registration. You can also create custom XML messages.

When it receives an Agent request, the Axeda Enterprise Server processes the data messages and stores them in its database. When replying to the Agent, the Platform can send commands, using the SOAP protocol. Agent Embedded can parse the incoming SOAP commands and make the necessary function calls to the asset application. Examples of SOAP messages include setting a data item, changing the Agent ping rate (how frequently it contacts the Platform for messages), executing a command, and file upload/download requests. If required, you can also create custom SOAP commands.

Agent Embedded performs the control message processing required by the eMessage protocol. For example, Agent Embedded automatically registers your assets with the Axeda Enterprise Server.

As with the standard Axeda Gateway and Connector solutions, Agent Embedded initiates all communications with the Platform, maintaining firewall transparency and network security. Figure 2-2 shows a high-level model of communications among your asset application, Agent Embedded, and the Axedad Platform.

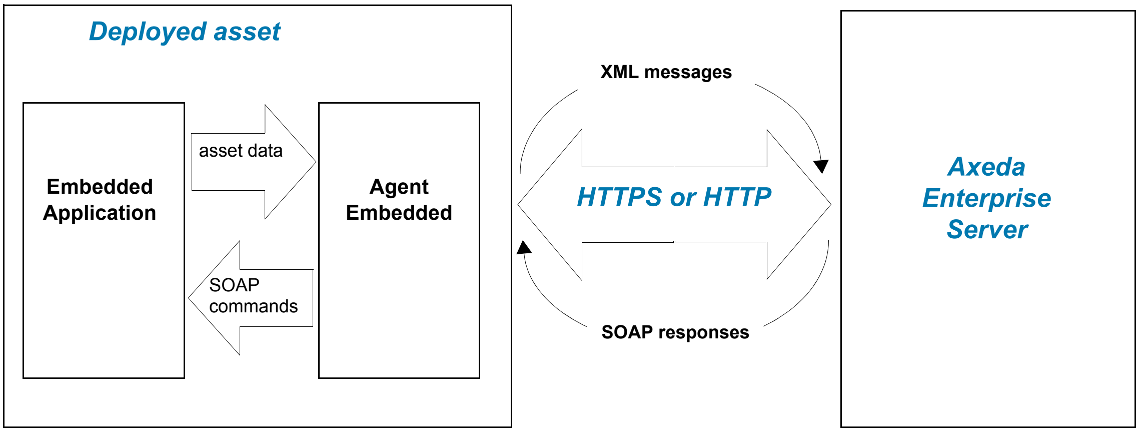

Figure 2-2. High-level communications model for Agent Embedded

Components of the Agent Embedded Toolkit Library

The components of the Agent Embedded Toolkit library enable you to collect data from the asset, communicate that data to the Axeda Enterprise Server, and execute commands from the Platform. Figure 2-3 shows the components available for implementation in the Agent Embedded Toolkit. The components shown in pale yellow (Web Client, XML/SOAP Parse, and Queue Manager) are loaded automatically. The components shown in light blue (Encryption, Remote Sessions, and File Transfers) are optional.

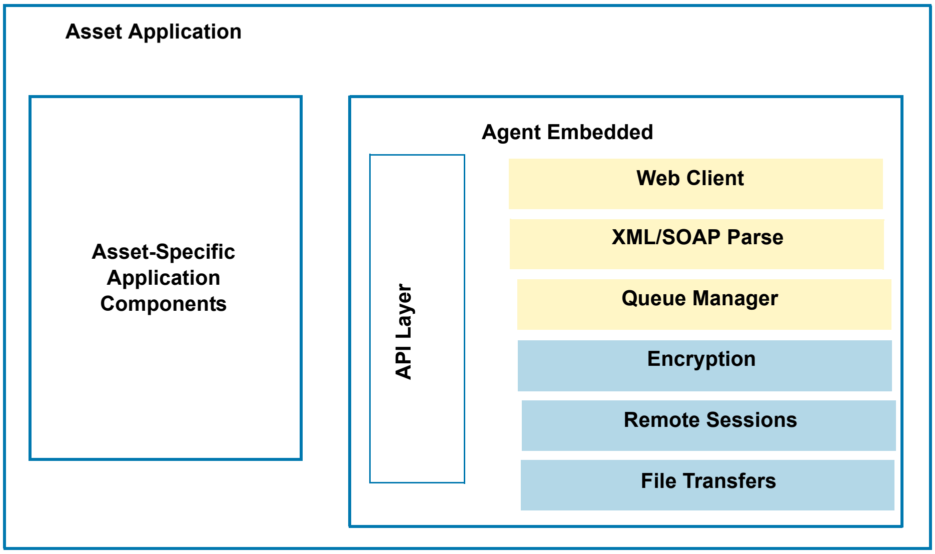

Figure 2-3. Components of an Asset Application Developed with Agent Embedded Toolkit

Most of the library components are enabled automatically at compile time. However, to minimize the footprint of Agent Embedded, the following components are optional: Remote Sessions, File Transfers, and Encryption. If your application does not need to perform remote sessions or to transfer files, you do not need to include those components of the Agent Embedded library. Although not recommended, you can also choose *not* to include portions of the library's Encryption support (based on the OpenSSL Open Source toolkit) if you do not need to ensure data security in your process. For information about this component, refer to *"Encryption Component (optional)"* on page 2-9.

To read and write data between your asset application and the Axeda Enterprise Server, the asset application must be extended to support Agent Embedded API functions. You can link Axeda Agent Embedded capabilities directly into the control processor of your asset or retrofit them to an asset through an external processor attached to the asset. How Agent Embedded operates depends upon what asset information you want to expose to the Enterprise Server, how your asset communicates, and the functionality supported by the operating system of your asset. This section presents the functionality available to your asset application through the use of Agent Embedded.

Web Client Component

To support HTTP communications with proxy servers, the Agent Embedded Toolkit library includes Web Client functionality. The Web Client can support both HTTP and SOCKS proxy servers. The Agent Embedded Toolkit supports Basic and NTLM HTTP authentication schemes, and Username/Password SOCKSv5 authentication method. At this time, it does NOT support Digest HTTP authentication scheme..

XML/SOAP Parser Component

The Agent Embedded Toolkit includes a generic XML parser/formatter and a SOAP RPC message analyzer. The Axeda Enterprise Server uses SOAP-formatted messages to communicate with Agent Embedded and accepts XML-formatted data messages from Agent Embedded.

When it receives data from the asset application, Agent Embedded formats data items into XML elements and places these elements in the data queue. The next time that the asset application calls *AeDRMExecute()*, Agent Embedded assembles the elements in the data queue into eMessage XML documents and queues the eMessages for delivery to the Enterprise Server.

When it receives commands from the Enterprise Server, Agent Embedded tries to convert the incoming octet stream into an XML element tree, which the SOAP RPC message analyzer then processes. The analyzer builds the final SOAP RPC structures, which are ready to be passed to the asset application using the provided callback, if any. The SOAP RPC structure specifies the method, its attributes, and the hierarchy of method parameters. Figure 2-4 on page 2-8 illustrates this process.

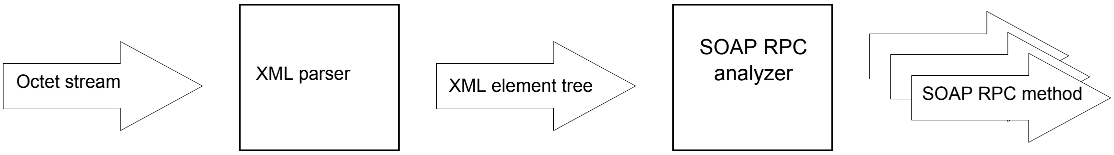

Figure 2-4. XML/SOAP RPC processing

Note: *Agent Embedded transforms SOAP RPC structures for the known SOAP methods to the specific method descriptors (for example, a structure describing the SetTag method) and invokes the corresponding callbacks. If a SOAP method is not known, the corresponding SOAP RPC structure is passed to the generic SOAP method callback, if the asset application provides the callback.*

Queue Manager Component

Before it is actually sent to the Axeda Enterprise Server, asset data is first submitted to the Queue Manager component of Agent Embedded. This component stores the data until it is successfully delivered to the Enterprise Server.

The Queue Manager assigns a priority level to each data item placed in the queue. The order in which data items are delivered to the Platform depends on their associated priority levels, with the highest priority and oldest data delivered first.

To confirm that communications between Agent Embedded and the Enterprise Server are intact, the Platform responds to each message sent from the agent. The Queue Manager stores a data item until the Platform confirms that it has successfully processed a message containing that data item. Upon receiving positive confirmation, the Queue Manager releases all data items sent in the successfully processed message.

Note: *When the queue becomes full, the Queue Manager removes data items from the queue, starting with the oldest and lowest priority data items.*

Encryption Component (optional)

Although the Encryption component is optional, Axeda recommends that you take advantage of this support for Secure Socket Layer/Transport Layer Security (SSL/TLS) encryption and authentication standards for communications with the Axeda Enterprise Server. The SSL/TLS protocols provide security when communicating data over TCP/IP networks. Encryption strength can range from 40-bit to 168-bit keys.

When enabled, SSL/TLS encryption ensures data security in the following ways:

- Before sending data to the Enterprise Server, embedded agents authenticate and confirm the identity of the machine running Enterprise Server.
- All data sent between Agent Embedded and the Platform is encrypted. Both the agent and the Platform encrypt the messages they send and then decrypt the messages they receive. In addition, the messages cannot be altered or tampered with during the communications process.
- Neither Agent Embedded nor the Platform are required to know the encryption key prior to the communication. The keys are securely exchanged by using the public key stored in the SSL certificate of the Platform.

Agent Embedded uses the OpenSSL library (http://www.openssl.org) to provide support for SSL/TLS. To support this encryption, a port of the OpenSSL library should exist for your asset platform. If a version of the OpenSSL library for your asset platform exists, you should define the macro, HAVE_OPENSSL, in your project. Support for SSL/TLS in Agent Embedded is controlled by the macro, ENABLE_SSL.

File Transfer Component (optional)

The File Transfer component of Agent Embedded enables the Axeda Enterprise Server to download files to Agent Embedded, and Agent Embedded to upload files to the server, using the Software Management features of the Enterprise Server. The Enterprise Server initiates file downloads; Agent Embedded or the Enterprise Server can initiate file uploads. The asset application can control which files, if any, it chooses to accept from a download or which files, if any, to upload in response to a request to upload files from the Enterprise Server.

The Agent Embedded Toolkit provides functions and structures for you to use when setting up file transfers. Among the parameters to set are the size of the file and whether to use compression. By default, Agent Embedded compresses files to be uploaded into a *tar.gz* file. Due to a limitation of the tar format, compression is not possible on files larger than 8GB. If you know that the file to be uploaded is already compressed or that it is over 8GB in size, disable compression in Agent Embedded. Note that when you disable compression, you can upload only one file at a time. For more information on setting up file transfers, refer to *"File Transfers"* on page 4-8.

You can choose whether or not to store files downloaded to Agent Embedded in the file system of the application. If files are not stored, Agent Embedded dynamically decompresses any compressed files and passes the data in the downloaded file to the asset application, as requested by the application. The application can then process the file.

The asset application can direct Agent Embedded to handle file transfers automatically or manually. If automatically, Agent Embedded uploads or downloads the files without exposing their content to the application. No callbacks are needed if you direct Agent Embedded to handle file transfers automatically. If manually, the asset application can intercept the file transfers and handle the content of the files as needed. For manual mode downloads, you'll need to use callbacks. In manual mode, the application can direct Agent Embedded to pass portions of the received data to the application for processing, perhaps to filter and store particular files or to apply file content to firmware configuration. For manual mode uploads, the application provides files to Agent Embedded for upload. For automatic mode uploads, Agent Embedded looks in the file system of the application for the files to upload, compresses the files, and streams the compressed content to the server.

The support for the Software Management features of Enterprise Server means that Agent Embedded also supports the Restart instruction for a package in addition to the file transfer instructions. None of the other instructions for a package are supported by Agent Embedded.

Remote Session Component (optional)

Agent Embedded provides support for Remote Sessions. The asset application specifies the remote session interfaces to make available for connection. Agent Embedded provides support for the following types of remote sessions: Desktop, Telnet, Browser, Automatic (Application Bridge server is started automatically on the local host, which is local to the client application), Manual (Application Bridge server is started manually on an arbitrary host; security is not an issue). The Desktop type of remote session supports the Quick Launch feature of the Axeda Applications. Agent Embedded and the Axeda Enterprise Server operate by brokering the communications between the remote server applications and client computers.

On startup, Agent Embedded discovers the version of Axeda Desktop Server running on its asset. When it contacts the Enterprise Server, Agent Embedded registers its list of available remote session interfaces, including the version information for Axeda Desktop Server. When a user of the Axeda Applications selects one of the defined interfaces (through the Axeda Service application), Agent Embedded begins brokering remote session communications between the remote server (as defined in the interface) and the Enterprise Server. In turn, the Platform brokers those communications back to the remote user. For example, if Agent Embedded registers a desktop interface and sends information about the version of Axeda Desktop Server to the Platform, an Applications user with privileges to the related asset can automatically launch the correct version of Axeda Desktop Viewer from the Asset dashboard in the Axeda Service application. When it next contacts the Platform, the Agent begins providing support for desktop communications to the Platform server defined for that interface.

The Agent Embedded Toolkit also provides callbacks that are invoked when a remote session is about to begin or has just embedded. For information about these callbacks, refer to *AeDRMSetOnRemoteSessionEnd()* and *"OnRemoteSessionEndCallback Function"* on page 5-17 and *AeDRMSetOnRemoteSessionStart()* and *"OnRemoteSessionStartCallback Function"* on page 5-18. For information on adding a remote session, refer to *"AeDRMAddRemoteSession()"* on page 5-9.

Program AeDemo1 included in the toolkit installation demonstrates Agent Embedded's Remote Session functionality. Refer to *"Sample Projects"* on page 3-9 for more information about this program.

Agent Embedded APIs

Axeda Agent Embedded APIs are functions that enable Agent Embedded to communicate asset information to the Axeda Enterprise Server, to communicate commands from the Platform back to the asset, to set encryption and proxy settings, and to manage the message queue. These functions are grouped in the following categories: Parameterization, Callback manipulation, Asset data export, Execution control, Asset status control, and Raw HTTP request execution. The rest of this section explains these categories in more detail. Chapter 5, " Function Reference", which presents the details of all functions, groups the functions by these categories.

Parameterization Functions

Parameterization functions enable the asset application to specify parameters for the managed asset, for communications with the Axeda Enterprise Server and proxy server, for Web Client settings for Agent Embedded's own Web Client, and for the data queue and messages.

Managed asset functions create the identifier for the asset (needed for all operations involving the Enterprise Server) and the role of the asset within the Enterprise Server (an Axeda® Gateway, an Axeda® Connector, or an asset managed by a gateway). These functions specify the model and serial numbers for the asset.

Functions for communications with the Enterprise Server specify the type of Enterprise Server(s) with which this Agent will communicate, such as the primary server, backup server(s), and any additional servers. These functions specify the URL address of the Enterprise Server, and set the encryption type and ping rate for contacting the server. Among the set of functions for communicating with the Enterprise Server is a function that allows you to configure Remote Sessions that will be handled by Agent Embedded.

Proxy server functions specify the version of the HTTP proxy server, and the host name, port number, user name, and password for communications through that server. Web Client functions specify settings for the HTTP protocol version, persistent connection usage, SSL parameters, and so on.

Data queue functions provide settings for the maximum queue size. When the queue reaches this size, the Queue Manager begins removing messages from the queue, starting with the oldest and least priority messages. The parameterization functions are listed in Chapter 5, “ Function Reference”, starting on page 5-2.

Callback Registration Functions

Callback registration functions enable Agent Embedded to set and unset callback values to the asset application. Although they are not required, callbacks may be registered to intercept the following events:

- Asset registration with the server.
- Data queue status changes, from empty to not empty or from full to not full.
- Web Client errors.
- SOAP method received from the Axeda Enterprise Server – when it receives SOAP methods from the server, the Agent passes them to the asset application through callbacks. The API supports two types of SOAP method callbacks: *generic* (used to pass a parsed, tree-like SOAP method representation) and *method-specific* (used to pass parameters of pre-defined SOAP methods).
- File transfers between the Enterprise Server and Agent Embedded.
- Remote sessions that are about to be started or that just ended

Callback registration functions are listed in Chapter 5, “ Function Reference”, starting on page 5-14. Each callback function is listed with its corresponding registration function.

Asset Data Export Functions

Asset data export functions enable Agent Embedded to submit asset data for later, rather than immediate, delivery to the Axeda Enterprise Server. Agent Embedded places submitted data into the data queue, based on the specified priority. The following types of asset data are supported for export:

- Snapshots
- Alarms
- Events
- E-mails
- File upload requests

- Opaque messages: asset data that is not formatted by Agent Embedded, but delivered to the Enterprise Server “as-is”. The calling asset application is responsible for the accuracy of opaque messages.

Asset data export functions are listed in Chapter 5, “ Function Reference”, starting on page 5-37.

Execution Control Function

The Execution control function, *AeDRMExecute()* enables Agent Embedded to communicate with the Axeda Enterprise Server. It performs the actions required to deliver queued data and dynamically-generated control messages for configured assets. File transfers and remote sessions are also processed inside *AeDRMExecute()*. This function is available in synchronous and asynchronous versions. Refer to the section, *"Synchronous and Asynchronous Operations"* on page 4-6. This section also describes the operating system-dependent function that you must implement for the default behavior of *AeDRMExecute()* to work, *AeSleep()* . *AeDRMExecute()* calls *AeSleep()* when it completes pending tasks and consumes the unused time.

Execution control functions are listed in Chapter 5, “ Function Reference”, starting on page 5-42.

Asset Status Control

Asset status control functions enable Axeda Agent Embedded to enable or disable communications for previously configured assets. Asset status control functions are listed in Chapter 5, “ Function Reference”, starting on page 5-43.

Raw HTTP Request Execution

These functions enable Agent Embedded to execute raw HTTP requests, enabling it to perform arbitrary HTTP-based transactions with any HTTP server. Raw HTTP request execution functions are listed in Chapter 5, “ Function Reference”, starting on page 5-44.

Chapter 3
Using Agent Embedded

This chapter provides information you need for using the Agent Embedded Toolkit in the following sections:

- ***Contents of the Toolkit Installation*** explains the platform and stack requirements for Axeda Agent Embedded Toolkit. It then presents a list and brief descriptions of the directories of the toolkit installation and a table that lists and briefly describes the Include files of this toolkit.
- ***Building your Device Application*** explains how to configure the library of the Agent Embedded Toolkit. For platforms that are not Win32 or UNIX-derived, it explains how to compile the library. It then explains how to build the library for any platform and how to use the APIs in your device application.
- ***Sample Projects*** lists and briefly describes the sample projects provided in the Agent Embedded Toolkit installation. You can use these samples as starting points for your applications.
- ***Tips and Troubleshooting*** provides information to help you troubleshoot connectivity problems between Agent Embedded and your Axeda Enterprise Server.

Contents of the Toolkit Installation

You can use the Axeda Agent Embedded Toolkit for any platform that has a ‘C’ language compiler and an implementation of TCP/IP stack according to Berkeley Sockets architecture.

The toolkit installation includes the following directories:

- *AgentEmbedded* – contains source files (obfuscated), project files, and a Readme file. Refer to Table 3-2 on page 4 for a listing and brief descriptions of the project files.
- *AgentEmbedded/Demo* – contains five demonstration programs: device data export, command execution, raw HTTP request, remote sessions and file transfer.
- *AgentEmbedded/Libsrc* – contains the following open source libraries used by Agent Embedded:
    - *expat* – XML parser.
    - *libdes* – DES encryption library (required only if OpenSSL is not available).
    - *zlib* – data compression library (required only if support for the File Transfer component is enabled).
- *AgentEmbedded/Include* – contains unobfuscated API Include files. Refer to Table 3-1 on page 3 for more information about the Include files.
- *AgentEmbedded/Sysdeps* – contains some declarations and implementations of functions specific to a particular OS:
    - *Win32* – for Win32 platform.
    - *Unix* – for a UNIX-derived OS.
- *AgentEmbedded/Compat* – source code used to handle differences in the build environments.

Table 3-1 lists and briefly describes the Include files in the toolkit installation.

Table 3-1. Agent Embedded Include Files

File name	Description
Sysdeps/<platform>/AeOSLocal.h	The file in which you can include standard headers and system-dependent declarations
Include/AeContainers.h	Declarations for Collections functions
Include/AeDRMSOAP.h	Declarations for DRMSOAP functions
Include/AeError.h	Declarations for Error functions
Include/AeInterface.h	Declarations for Interface functions (the main API functions)
Include/AeOS.h	System-dependent function declarations. You must implement functions declared in this file for your target platform. The *Sysdeps* directory contains implementations for Win32 and UNIX-derived operating systems.
Include/AeTypes.h	Axeda Agent Embedded types and macros
Include/AeUtil.h	Axeda Agent Embedded utility functions
Include/AeVersion.h	Version macros
Include/AeWebRequest.h	Declarations for HTTP request manipulation
Include/AeXML.h	XML parsers

Table 3-2 lists and describes the project files included in the toolkit installation.

Table 3-2. Project Files

IDE	File	Description
Microsoft Visual Studio 2008	AgentEmbeddedNT_2008.sln	Agent Embedded solution file for MSVC 2008
	AgentEmbeddedNT_2008.vcproj	Agent Embedded project file for MSVC 2008
	AeDemo*(1-5)*NT_2008.vcproj	Project file for the sample programs for MSVC 2008; there are five programs from which to choose.
Microsoft Visual Studio 2005	AgentEmbeddedNT_2005.sln	Agent Embedded solution file for MSVC 2005
	AgentEmbeddedNT_2005.vcproj	Agent Embedded project file for MSVC 2005
	AeDemo*(1-5)*NT_2005.vcproj	Project file for the sample programs for MSVC 2005; there are five programs from which to choose.
Microsoft Visual Studio 6.0	AgentEmbeddedNT.dsw	Agent Embedded workspace file for MSVC 6.0
	AgentEmbeddedNT.dsp	Agent Embedded project file for MSVC 6.0
	AeDemo*(1-5)*NT.dsp	Project file for the sample programs for MSVC 6.0; there are five programs from which to choose.
	Makefile(s)	Makefiles to build Agent Embedded with gmake.

Building your Device Application

First, you need to compile the library of Agent Embedded for your specific platform and application needs. After that task is complete, you can use the API of the Agent Embedded Toolkit in your asset application.

How to configure the library for Agent Embedded

First, define the macros you require for your application using the compiler build tool for your platform. On Windows platforms using Microsoft Visual Studio, define macros in the C/C++ > Preprocessor > Preprocessor Definitions configuration section. On POSIX platforms such as Linux, configure the macros within your Makefile. The following macro definitions determine which characteristics and features are supported by the library:

- AE_BIG_ENDIAN – define this macro when the target hardware platform has the big endian byte order. The default byte order is little endian.
- HAVE_PTHREADS – when this macro is defined, Agent Embedded assumes that a pthread-compliant library is available in the build environment (UNIX-derived platforms only). You may need to adjust project files to specify the location of pthread header files and libraries.
- HAVE_OPENSSL – when this macro is defined, Agent Embedded assumes that OpenSSL libraries are available in the build environment. You may need to adjust project files to specify the location of OpenSSL header files and libraries.
- _REENTRANT – when this macro is defined, support for multi-threaded operations is enabled for UNIX-derived platforms.
- ENABLE_SSL – when this macro is defined, support for secure communications over SSL/TLS is enabled (HAVE_OPENSSL must be defined as well).
- ENABLE_REMOTE_SESSION – when this macro is defined, support for the Remote Session functionality is enabled.
- ENABLE_FILE_TRANSFER – when this macro is defined, support for the File Transfer functionality is enabled.

- ENABLE_LARGEFILE64 – when this macro is defined, support for transfer of files larger than 2GB is enabled. If you know that the files that need to be transferred will grow larger than 2GB, you must define this macro. When this macro is defined, the file size limitations are as follows:
  - Compressed transfers are limited to 8GB (due to a limitation in the tar format).
  - Uncompressed transfers have a theoretical limit of 9223372036854775807 bytes (~9 EB (exabyte)

  Important! *Data type names for the 64-bit integer vary among C compilers, and not all compilers support it, so you need to do the typedef for AeInt64 in the system-dependent part of the library (for example, Sysdeps/Win32/AeOSLocal.h.*

  Your device application should define ENABLE_LARGEFILE64 ***only*** *if your compiler has native support for 64-bit integers. In addition, if the files are transferred to or from a real file system, the file system and Operating System* ***must*** *support 64-bit-based file operations. For more information on file transfers, refer to "File Transfers" on page 4-8.*

How to compile the library

Generally, when you compile the library for a Win32-based or UNIX-derived platform, you do not need to change the source code because the system-dependent code is provided for these platforms in the *Sysdeps* directory. Follow the steps below *only* if you need to port the library for Agent Embedded to a platform *other than* Win32 or UNIX.

To port the Agent Embedded library

1. Build the *expat*, *libdes,* and *zlib* open source libraries found under the *Libsrc* directory. Follow the build instructions included in the package directories. You need to build the *libdes* library only if OpenSSL is not available for your platform or you choose not to use it. You need to build the *zlib* library only if support for the File Transfer component is enabled.
2. Create a directory for your platform files under *Sysdeps*. The rest of these instructions assume the directory name is *MyPlatform*.
3. Provide system-dependent declarations in file *AeOSLocal.h* and copy that file to *Sysdeps/MyPlatform* . Use the existing *AeOSLocal.h* files for Win32 and UNIX as a template.
4. Implement the system-dependent functions declared in *Include/AeOS.h*. Use the existing *AeOS.c* files for Win32 and UNIX as a template.
5. Save the implementations in the file, *AeOS.c* , and copy that file to *Sysdeps/MyPlatform* .
6. Create project file(s) and include the obfuscated code of Agent Embedded library, *Sysdeps/MyPlatform/AeOS.c*, the 'C' source files in the *Compat* directory, and the source code for the demo programs. Be sure to include *Include*, *Sysdeps/MyPlatform*, *Compat,* and the open source directories in the search path of the header file.

   As necessary, link the *expat*, *libdes,* and *zlib* libraries with the demo programs. (In later procedures, you will have to link these libraries to your asset application as well). Use the existing *Makefile* as a template if a *make*-like utility is used to build programs for your platform.

How to build the Agent Embedded library

Now you need to build the Agent Embedded library for your platform. To build this library for Win32 platforms, use Microsoft Visual Studio. To build this library for a UNIX-derived platform, use *gmake/gcc*. Other platforms provide different tools for this task.

Once the library has been built, you can use the APIs of the Agent Embedded Toolkit in your asset application.

To use the agent APIs in your asset application:

1. Configure the basic runtime objects, such as controlled devices and destination Axeda Enterprise Servers (defined in the *AeDRMAddDevice()*, *AeDRMAddServer()* API functions).
2. Configure the server URL as required for the function, using the following syntax:

   `https://<server-name>:<port>/eMessage` (secure communications)

   or

   `http://<server-name>:<port>/eMessage` (non-secure communications)

   where `<server-name>` is the name (or IP address) of the Enterprise Server, and `<port>` is the port number for communicating with the server (use 443 for SSL communications, 80 for non-secure HTTP communications).

   For example, `https://drm.axeda.com:443/eMessage`.
3. Configure any optional parameters as needed. Such parameters may include proxy servers, Web Client, and data queue information (defined in the *AeWebSetXXX()* functions, such as *AeWebSetProxy(),* and *AeDRMSetQueueSize()*). Refer to the 'C' file for the example project you are using for the optional input parameters for each demo program.
4. Implement any callback functionality within your asset application, as appropriate, and register the callbacks using the *Callback Registration Functions and Their Callback Functions* (defined in the *AeDRMSetOnXXX()* functions). Each function may return error codes.

   Note: *Callbacks are **not** required.*
5. Make sure you handle any errors that are returned from the API functions.
6. Call *AeDRMExecute()* within the main control loop of your application. Use synchronous or asynchronous mode as appropriate. Refer to *"Synchronous and Asynchronous Operations"* on page 4-6. When you call *AeDRMExecute()* in asynchronous mode, Agent Embedded saves its state upon exiting from the function so that pending operations may be resumed when the function is called again. The saved state includes any open TCP/IP connections.

   Note: *You should ensure that the intervals between calls to AeDRMExecute() in your application are less than the defined ping rate value.*

Sample Projects

Table 3-1 lists and briefly describes the files needed for building the sample projects included in the Agent Embedded Toolkit installation. You can use the sample projects as a starting point when creating real applications. To compile and run these projects, complete the steps explained in *"How to compile the library"* on page 3-7.

Table 3-1. Sample Projects

File	Description
Demo/AeDemo1.c	Demo program 1: data export example. This program defines a device and configures the primary Axeda Enterprise Server (using URL argument), configures a sample remote session, and then loops and posts simulated data and alarms.
Demo/AeDemo2.c	Demo program 2: command execution example. This program defines a device, configures the primary Enterprise Server (using URL argument), and registers command callbacks. After that, it loops and waits for callback invocation.
Demo/AeDemo3.c	Demo program 3: raw HTTP request example. This program performs *HTTP GET* method on a specified URL and displays the contents.
Demo/AeDemo4.c	Demo program 4: application-controlled file download example. This program intercepts file downloads and uses data in a file with a predefined name (that is, (MC68EZ328-based board) to re-program the device firmware.
Demo/AeDemo5.c	Demo program 5: application-controlled file upload example. This program shows how the device application can intercept file uploads.
AgentEmbeddedNT_2008.sln AgentEmbeddedNT_2005.sln AeDemo(1-5)NT_2008.vcproj AeDemo(1-5)NT_2005. vcproj AeDemo(1-5)NT.dsp	The *vcproj* files are the project files for demo programs for MSVC 2005 and 2008. The demo projects are included in the Agent Embedded Toolkit workspace. Use the project file corresponding to the selected version of Microsoft Visual Studio (AeDemo1 through AeDemo5) and build environment/platform. *Note: Instead of each project file, you can use the solution (sln) files.*
AeDemoCommon.c, AeDemoCommon.h	Utility functions used in the demo programs.

Tips and Troubleshooting

This section provides tips for using the Agent Embedded Toolkit, as well as help with troubleshooting problems you may encounter when developing or running Agent Embedded. For troubleshooting purposes, it is suggested that you set *AeDRMSetDebug()* to true, which will enable debug message logging.

The most common problem that users encounter during a new installation of Axeda Agents and the Axeda Enterprise Server is lack of communication between the Enterprise Server and the Agents. If you follow the steps in the procedure, *"To troubleshoot a communications problem:"* on page 3-10, and collect the information they request, you will help Axeda Technical Support to service your call more quickly. As you work through this procedure, open a file and record your observations or the results of a step. You can also jot down observations on paper. You will need them when talking with Axeda Technical Support.

To troubleshoot a communications problem:

1. Check the following infrastructure issues:
   - SSL — If you have configured the Axeda Agent and Axeda Enterprise Server to use SSL to communicate, verify that the port number is correct (default is 443). Then, check that the SSL strength is configured correctly for both sides.
   - Proxy setup — If you are using a proxy server between the Agent and the Enterprise Server, verify that it is configured correctly.

     Refer to http://help.axeda.com to learn how to obtain the document that lists the proxy server protocols that have been tested and certified with Agent Embedded Web Client connections.
2. Check the Firewall setup:

   a) Is there a firewall that could block either posting data or receiving responses?

   b) Does the firewall block any MIME types? Print the list on the next page and then check all of the MIME types and write down Yes or No to indicate whether the MIME type is blocked by your firewall.

Mime Types

- For EMessage:
  - Agent -> Enterprise (POST)
    Content-Type: application/octet-stream
    Content-Length: 200
    Connection: Keep-Alive
    Expect: 100-continue
  - Enterprise -> Agent (reply)
    HTTP/1.1 100 Continue
    HTTP/1.1 200 OK
    Content-Type: application/octet
    Content-Length: 263
    Connection: close
    ...(data)

- Remote Session (GET)
  - Agent -> Enterprise
    Connection: Keep-Alive
  - Enterprise -> Agent (reply)
    Content-Length: 24
    Content-Type: application/octet-stream
    Connection: Keep-Alive
    Cache-Control: no-cache

  Remote Session (POST)
  - Agent -> Enterprise
    Content-Type application/octet-stream
    Content-Length: 24
    Connection: Keep-Alive
  - Enterprise -> Agent (reply)
    Content-Length: 4
    Connection: Keep-Alive

- File Upload (POST)
  - Agent -> Enterprise
    Content-Type: application/octet-stream
    Content-Length: 467
    Content-MD5: AxaxNGXeCARFwb5udFnQEQ==
    Connection: Keep-Alive
    Expect: 100-Continue
  - Enterprise -> Agent (reply)
    HTTP/1.1 100 Continue
    (data)
    HTTP/1.1 200 OK
    Content-Length: 0

URIs Not Allowed in the Infrastructure

Why is the Agent, Access Remote, or Web browser not correctly communicating with the Axeda Enterprise Server? The following URIs are required between the Axeda Agents, Access, and Web Browser and the Enterprise Server. If the Agent, Access Remote, or Web Browser is not correctly communicating or suddenly stops correctly communicating with the Enterprise Server, please check with your IT department to make sure these URIs are not being blocked anywhere upstream in the network.

Table 3-2. URIs

URI	Used by
/servicelink/*	User
/remote/*	jta20.jar, appbridge.jar and Agent
/rpc/*	Axeda Builder
/eMessage	Axeda Agent
/MtMessage	Axeda Agent
/pserver	Access Remote
/pviewer	Access Viewer
/pgetsession	Access Remote
/paudit	appbridge.jar (for APB Access sessions)
/poptimize	Agent Access (Access Remote & Access Viewer), jta20.jar, and appbridge.jar
/pconnectioninfo	Used for connection info tracking
/psessioninfo	Access Remote & Access Viewer
/pfiletransfer	Access Remote
/upload	Axeda Agent
/download	Axeda Agent
/images/*	Axeda Enterprise Server
/styles/*	Axeda Enterprise Server
/index.html	Axeda Enterprise Server

Refer to http://help.axeda.com if you need more help troubleshooting connections between Agent Embedded and your Axeda Enterprise Server.

Chapter 4
Agent Embedded Operations

This chapter provides details concerning operations that Agent Embedded performs to establish and maintain communications with the Axeda Enterprise Server. This chapter contains the following sections:

- ***Communications Concepts***, provides a brief overview of communications between Agent Embedded and the Axeda Enterprise Server.
- ***Controlling the Communication with the Axeda Enterprise Server*** explains when your application should call *AeDRMExecute()* to ensure the asset is communicating with and responding to requests from the Axeda Enterprise Server.
- ***Initiating Communications*** explains the actions that Agent Embedded takes when the device application calls *AeDRMExecute()*.
- ***Sending XML Messages and Receiving SOAP Messages*** provides specific information about sending eMessages (XML) and receiving and acting on SOAP methods. It also illustrates the relationship of the internal data objects of Agent Embedded. Finally, it explains the two modes for calling *AeDRMExecute()*, synchronous and asynchronous.
- ***Synchronous and Asynchronous Operations*** explains Agent Embedded operations in synchronous and asynchronous modes.
- ***Operating System Abstraction Layer*** explains operating system details for applications using Agent Embedded, including thread safety information.
- ***File Transfers*** lists the functions available for setting up file uploads and downloads and also provides a table of the structures and their parameters that you can use for file uploads and downloads.

Communications Concepts

Like their full-featured counterparts (Axeda Connector and Axeda Gateway), embedded agents provide two-way, Firewall-Friendly monitoring, communications, and control of asset data and events in real time. Included in these services are the abilities to establish and maintain secure communication between the Axeda Enterprise Server and the asset.

Operating as a middleman, Agent Embedded brokers the communications between the asset and the Enterprise Server. The Agent Embedded is integrated with an existing asset application to enable the asset to communicate its data, events, and alarms to the Enterprise Server, and then receive commands back for execution on the asset.

Communications are always initiated by the Agent. Communications between an Agent and the Enterprise Server are based on HTTP. HTTP requests are used for sending Agent messages, and HTTP responses are used for receiving actions from the Enterprise Server.

To maintain the two-directional connectivity between the Platform and the Agent, the Agent needs to contact the server periodically. If outgoing payload data (data items, alarms, events, and so on) is pending on the Agent side, it is sent as soon as possible. If no payload data is pending, the Agent sends a ping message to indicate its active status to the server and also to retrieve any actions that may be pending on the server side.

The period of time between any two messages sent by the Agent should not exceed the ping rate value. For this reason, *AeDRMExecute()* should be called at least as frequently as the ping rate.

Ping Rate

Sometimes referred to as the “heartbeat,” the *ping rate* is the amount of time that the agent waits before sending a ping message when no outgoing data is pending. This interval is defined as a number of seconds. The rate set depends on the particular site and can be adjusted from the Axeda Enterprise Server after the agent has been running.

Controlling the Communication with the Axeda Enterprise Server

An asset application can use Agent Embedded to control communications with the Axeda Enterprise Server. To initiate communications with the Enterprise Server, an application needs to call *AeDRMExecute()*. For example, your application may want to call *AeDRMExecute()* periodically while the asset is operating to make sure the asset is actively connected to and communicating with the Enterprise Server.

Initiating Communications

When the asset application calls *AeDRMExecute()*, Agent Embedded performs the following actions in the order shown:

1. Checks if the agent has already registered with the Axeda Enterprise Server. If not, Agent Embedded generates an appropriate message and schedules it for delivery to the Enterprise Server.
2. Checks if the agent needs to send a status ping message to the Enterprise Server. The agent needs to send a status ping to the server if the time elapsed since the last ping is greater than the value of the ping rate set for the agent. If a ping is needed, Agent Embedded generates an appropriate message and schedules it for delivery to the Enterprise Server.
3. Checks if the agent needs to send a maintenance ping message to the Enterprise Server. The agent needs to send a maintenance ping to the server if the time elapsed since the last maintenance ping is greater than the value of the maintenance ping rate set for the device. If a maintenance ping is needed, Agent Embedded generates an appropriate message and schedules it for delivery to the Enterprise Server.
4. Processes the data queue. All preformatted data items allowed for posting to the server are taken from the data queue and added to a message for delivery.
5. Formats all messages that need to be sent to Enterprise Servers and submits them to the Web Client of Agent Embedded for processing.
6. Processes the Web Client's pending HTTP transactions. Invokes Web Client error callback if there are errors.
7. Processes pending data transfers associated with open remote sessions (if support for remote sessions is enabled).
8. Processes pending data transfers associated with file transfers that are in progress (if support for File Transfers is enabled). Invokes file transfer callback processing as directed by the application.
9. Processes server responses and parses any SOAP methods received from the server for the completed HTTP transactions.
10. Certain SOAP methods cause Agent Embedded to open remote sessions and start file transfers if support for the corresponding functionality is enabled. If remote sessions or file transfers are pending, processes them and if appropriate, invokes any related callback.

11. Invokes a device registration callback if the Enterprise Server indicates the registration was successful.

12. Invokes a data queue status callback if the queue becomes empty during this process.

13. Invokes the SOAP method callback provided for each of the received SOAP methods. The device identifier (deviceId) of the destination device is passed to the callback method.

Sending XML Messages and Receiving SOAP Messages

Certain restrictions are imposed on Agent Embedded when it composes XML messages for delivery. For example, if HTTP transactions are pending for an Axeda Enterprise Server, Agent Embedded cannot generate new messages for that server.

Agent Embedded knows of the following SOAP methods:

- Set tag value (the value of a data item)
- Set time
- Restart device

Agent Embedded uses output parameters (the return values) obtained from a call to a SOAP method callback to generate SOAP command status, which is added to the queue for delivery to the Enterprise Server.

The use of SOAP method callbacks is based on the following rules:

- If Agent Embedded receives a known SOAP method, and you have registered a corresponding callback for your application, it invokes that callback and passes method-specific parameters.
- If a method-specific callback for a received SOAP method was not registered for your application but would be invoked otherwise, the SOAP method is ignored.
- If a generic SOAP method callback was registered for your application, and Agent Embedded receives an unknown SOAP method, it invokes a generic callback. In this case, Agent Embedded passes a tree-like representation of the method parameters to the callback.
- If no callback is invoked, Agent Embedded sends a negative SOAP command status to the Axeda Enterprise Server.

Figure 4-1 illustrates the relationships among the internal data objects of Agent Embedded:

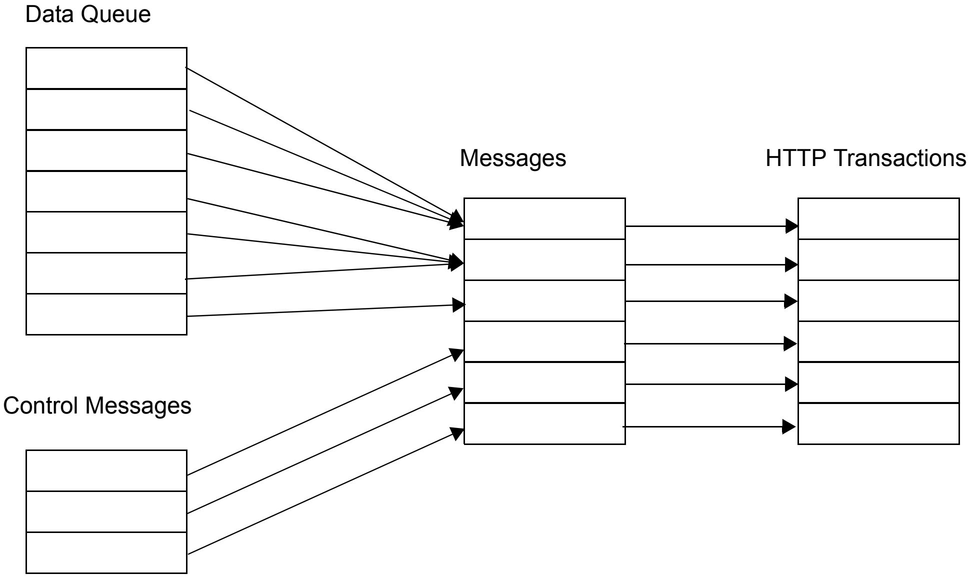

Figure 4-1. Relationships among internal data objects

Synchronous and Asynchronous Operations

For reasons of portability and low resource usage, calls to API functions do not spawn new threads. At the same time, the functionality of Agent Embedded involves I/O operations, which can take a significant amount of time to complete. To achieve flexibility for different use cases, the *AeDRMExecute()* function operates in either asynchronous or synchronous mode, based on the value pointed to by its `pTimeLimit` argument.

The `pTimeLimit` argument is a pointer to the AeTimeValue structure (defined in *AeTypes.h*). When this pointer is NULL, the operation becomes synchronous, or *blocking* (the function blocks until all of the pending tasks are completed). Messages are generated as explained in *"Initiating Communications"* on page 4-3. The calling asset application blocks until all messages are processed (either delivered to the Axeda Enterprise Server or a failure occurs during the delivery). Note that the data queue may remain non-empty after a return from the function call, perhaps because some of the data items could not be posted, the posting failed, or the message exceeded its maximum size.

When the time value pointed to by `pTimeLimit` is zero (0.0 seconds), then the operation becomes asynchronous or *non-blocking* (the function performs pending non-blocking operations and returns immediately). When the time value pointed to by `pTimeLimit` is non-zero, then *AeDRMExecute()* will work in asynchronous or non-blocking mode on pending operations for no longer than the specified time value. If pending network operations remain, they will be processed by subsequent calls to *AeDRMExecute()*. Messages are generated as explained in *"Initiating Communications"* on page 4-3.

The default behavior of *AeDRMExecute()* is asynchronous mode and return when a specified time limit has elapsed. This behavior can be modified by calling *AeDRMSetYieldOnIdle(AeTrue)*, which causes *AeDRMExecute()* to return once no pending network operations remain. When using *AeDRMExecute()* in asynchronous mode, note that the callbacks registered by the asset application are invoked in the context of this function. Since *AeDRMExecute()* has no control over the callback execution, return from the function may be delayed if the callback takes considerable time to finish. As a general rule, the callbacks should be non-blocking.

Operating System Abstraction Layer

During its operations, Agent Embedded makes calls to the operating system of the asset application. The asset application must implement certain OS-specific functions, documented and declared by Agent Embedded, for your specific platform. Agent Embedded is shipped with two sets of OS function implementations: one for Win32 and one for Unix platforms.

The classes of OS-specific functions used by Agent Embedded are defined in *AeOS.h* and are as follows:

- Network-related functions, corresponding to a subset of the BSD Socket layer
- Time query functions
- Thread-safe locking support functions
- File I/O functions

Thread Safety

Although Agent Embedded does not create any new threads itself, your asset application may use multiple threads when calling its API functions. Agent Embedded uses locks to avoid concurrent access to global internal objects. To perform lock acquisition and release, Agent Embedded uses the corresponding functions defined in the abstraction layer of the operating system. The functions that Agent Embedded uses for lock operations are declared in *AeOS.h*.

File Transfers

Your asset application can upload files to the Axeda Enterprise Server through Agent Embedded. Either your application or the Enterprise Server can initiate a file upload. In addition, the Enterprise Server can download files through Agent Embedded to your application. Only the Enterprise Server initiates file downloads. Your application can process and/or store a downloaded file. For example, an asset application can retrieve data from a downloaded file and process the data.

Important! *Agent Embedded supports the transfer of large files, where "large" is defined as files larger than 2GB. To support the transfer of large files, your C compiler must have native support for 64-bit integers. In addition, your operating system and its file system must support 64-bit-based file operations. Since data type names for 64-bit integers vary among C compilers, you must define the AeInt64 typedef in the* ***AeLocalOS.h*** *file for your platform during the configuration step of building the Agent Embedded library (refer to "How to configure the library for Agent Embedded" on page 3-5 for complete information about this step).*

Consider very carefully whether files that Agent Embedded will be transferring will ever grow larger than 2GB. If the answer is yes, then you must define ENABLE_LARGEFILE64, or the results of the transfer will be unpredictable.

When ENABLE_LARGEFILE64 is defined, the file size limitations are as follows: Due to a limitation in tar format, compressed file transfers are limited to 8GB. Uncompressed file transfers have a theoretical limit of 9223372036854775807 bytes (~9EB (exabyte)).

When compression is disabled, only one file can be uploaded at a time. For files that are larger than 2GB but less than 8GB, you can enable compression so that more than one file can be uploaded at a time. Note that when multiple files are transferred, they are not transferred simultaneously.

To support file transfers, Agent Embedded provides several functions and structures. Separate function names exist for file uploads and downloads and for files smaller than 2GB and files larger than 2GB. When you define `ENABLE_LARGEFILE64` (your application will handle files larger than 2GB), Agent Embedded automatically sets the normal function names as synonyms for the *64 functions, so you can use the normal function name in your program and Agent Embedded will manage the rest for you. Without the ENABLE_LARGEFILE64 compiler directive, file sizes and offsets are stored in 32-bit signed integers, with a representation limit of 2^31-1 (2GB).

The file transfer functions are as follows:

- The following Device Data Export functions submit a request for a file upload to the Axeda Enterprise Server on behalf of the asset application. The difference in these two sets of functions is that the *Ex* function allows you to specify additional parameters for the entire upload operation (specified with *AeFileUploadRequestParam()*):
    - *AeDRMPostFileUploadRequest() and AeDRMPostFileUploadRequest64()*
    - *AeDRMPostFileUploadRequestEx() and AeDRMPostFileUploadRequestEx64()*
- Although they are not required, you can register callbacks that are invoked during file transfers. The following Callback Manipulation functions are available when your asset application needs to perform additional processing during or after a file transfer:
    - *AeDRMSetOnFileDownloadBegin()* — Registers the callback invoked when starting the download for a series of one or more files.
    - *AeDRMSetOnFileDownloadData() and AeDRMSetOnFileDownloadData64()* — Registers the callback invoked when Agent Embedded receives a portion of a downloaded file.
    - *AeDRMSetOnFileDownloadEnd()* — Registers the callback invoked when Agent Embedded either encounters an error during the file download, or when all files in the series are received successfully
    - *AeDRMSetOnFileUploadBegin() and AeDRMSetOnFileUploadBegin64()* — Registers the callback invoked when starting the upload for a series of one or more files.
    - *AeDRMSetOnFileUploadBeginEx() and AeDRMSetOnFileUploadBeginEx64()* — Otherwise the same as AeDRMSetOnFileUploadBegin, provides for specifying additional parameters for the entire upload operation (in the argument, `AeFileUploadExecuteParam`, of this function).
    - *AeDRMSetOnFileUploadData()and AeDRMSetOnFileUploadData64()* — Registers the callback invoked when Agent Embedded requires more file data for the upload.
    - *AeDRMSetOnFileUploadEnd()* — Registers the callback invoked when Agent Embedded either encounters an error during the file upload, or when all files are uploaded successfully
    - *AeDRMSetOnFileTransferEvent()* — Registers the callback invoked when Agent Embedded either encounters an error during the file upload, or when all files are uploaded successfully.

Either before or after a file transfer, you can use the following system-dependent Agent Embedded functions to operate on files:

- *AeFileOpen()*
- *AeFileSeek()*
- *AeFileRead()*
- *AeFileWrite()*
- *AeFileClose()*
- *AeFileDelete()*
- *AeFileExist()*
- *AeFileGetSize()and AeFileGetSize64()*
- *AeMakeDir()*

Chapter 5, " Function Reference", explains all the functions in detail; the file transfer functions are defined in *AeInterface.h*. The file manipulation functions are operating-system specific and are defined in *AeOS.h*.

Table 4-1 on page 11 lists and describes the structures that you can use with file transfer functions. These structures are defined in *AeTypes.h*.

Here are some important notes about Agent Embedded and the Axeda Software Management features of the Axeda Enterprise Server:

- Agent Embedded supports only the file upload, file download, and restart instructions for a package. It does not support any other instructions.
- Agent Embedded does NOT support rollback instructions of any kind, so if you send a package with a rollback instruction set that includes a restart, it will not work with Agent Embedded.
- Agent Embedded does support automatic deployment of Software Management packages.
- If a package has a restart instruction and the Agent re-registers without going missing (in other wods, it restarts quickly), then pending package deployments are NOT retried.
- Agent Embedded's built-in file uploader does not support wildcards in the file specification.

Table 4-1. File Transfer Structures

Use this structure	to	Parameters
File attributes `AeFileStat`	Communicate file attributes from Agent Embedded to your asset application during file download or to communicate file attributes from the asset application to Agent Embedded during file upload.	`pName` - a pointer to the name of the file `iType` - The type of file - Unknown, Regular, or Directory (as defined in AeFileType) `iSize` - the size of the file. Depending on the size of the file, the data type of this parameter can be AeInt32 or AeInt 64. Not all operating systems support AeInt64; be sure that your operating system supports this before attempting to upload files greater than 2GB. `iMTime` - the timestamp that the file was last modified.
File upload specification `AeFileUploadSpec`	Request a file upload from the asset application to the Axeda Enterprise Server (one structure for each file). When you are using the structure for this purpose, Agent Embedded ignores the `iPosition` parameter. Agent Embedded also passes this structure to your asset application (one structure per file) when it receives an upload command from the Enterprise Server. In this case, your asset application must start with the file position indicated in the `iPosition` parameter.	`pName` - a pointer to the name of the file `bDelete` - a Boolean indicating whether to delete the file after uploading it `iPosition` - a location in the file from which the asset application should start the upload (this parameter is ignored when the asset application initiates the upload request); the data type of this parameter can be AeInt32 or AeInt 64, depending on the size of the file. Not all operating systems support AeInt64; be sure that your operating system supports this before attempting to upload files greater than 2GB.

Table 4-1. File Transfer Structures *(continued...) (continued...)*

Use this structure	to	Parameters
File upload request parameters AeFileUploadRequest-Param	Communicate additional upload parameters from the asset application to Agent Embedded when the former requests a file upload. The parameters apply to the *entire* upload operation rather than to individual files. When iCompression specifies AeFileCompressionNone, only one file may be requested in the upload. The iPriority value is compared with the priorities of other pending file transfers.	iCompression - whether to use compression, as defined by AeFileCompression (None or TarGzip). pId - a string to pass back to the asset application when the upload command is received from the Axeda Enterprise Server. pHint - a string to associate with the upload for customized processing the server. iPriority - the priority of the file upload, specified as an integer. Larger numbers specify a higher priority. iMask - specifies which parameters were initialized.
File upload execution parameters AeFileUploadExecute-Param	Communicate additional upload parameters from Agent Embedded to the asset application when the upload is executed (that is, the upload command is received from the Axeda Enterprise Server).	pId - the string that has been passed from the asset application to the Axeda Enterprise Server when the upload was requested. pId is not set if the upload was initiated by the Enterprise Server. iMask - specifies which parameters were initialized

Uploading Files without Access to the Entire Content at the Start of the Upload

It is possible to a upload file from Agent Embedded without having access to the entire file content at the beginning of the upload. Here are two ways you can do this:

- If only partial content is available at the beginning of file upload, but the total size of the file is known, simply specify the correct file size in `(*ppFile)->iSize` during the call to the `OnFileUploadData` callback.

  Then, specify the next available file segment using `*ppData` and `*piSize`. Agent Embedded will call `OnFileUploadData` repeatedly until the entire content (as specified with `(*ppFile)->iSize`) has been provided by the callback. This approach works for both compressed and uncompressed uploads.

- If the size of the file is not known during upload, and the upload is uncompressed, use the same approach as before, but specify an estimate of the file size in `(*ppFile)->iSize` during the call to `OnFileUploadData` callback. The estimated file size must be greater than the current position in the upload. To initiate an uncompressed upload, use `AeDRMPostFileUploadRequestEx()` with `pParam->iCompression` set to `AeFileCompressionNone`. This approach works only for uncompressed uploads.

Chapter 5
Function Reference

This chapter lists all Agent Embedded API and library functions, by supported operation. For each function, this reference shows the name and syntax used when calling that function, a description of that function, and the name of the Include file in which that function is defined. This chapter contains the following sections:

Parameterization Functions

For background information on parameterization functions, refer to *"Parameterization Functions"* on page 2-11 of this reference. Use the functions in this section to initialize and shut down Agent Embedded, to configure logging for Agent Embedded, and to specify parameters that you want Agent Embedded to use for communications with your asset (device) and the Axeda Enterprise Server.

AeInitialize()

```
AeError AeInitialize(void)
```

Description: Initializes Axeda Agent Embedded. You must call this function before you can make any other call to Agent Embedded.

Include file: AeInterface.h.

AeShutdown()

```
void AeShutdown(void)
```

Description: Shuts down Axeda Agent Embedded. When the asset application stops using Agent Embedded, call this function to release system resources.

Include file: AeInterface.h.

AeSetLogFunc()

```
void AeSetLogExFunc(void (*pLogFunc)(AeLogRecordType iType, AeChar *pFormat,
...))
```

where:

- `pLogFunc` – pointer to the customer extended log function
- `iType` – indicates the type of log message. As defined in *AeOS.h* (`struct _AeLogRecordType`), the possible types of log message, from least to most verbose, are None, Error, Warning, Info, Debug, and Trace.
- `pFormat` – pointer to the format.

Description: Registers a custom log function (default: AeLog). When `pLogFunc` is NULL, the default log function is set. This function is kept for backwards compatibility only. Agent Embedded does not use this type of log function any more. Instead, use *AeSetLogExFunc()*.

AeSetLogExFunc()

```
void AeSetLogExFunc(void (*pLogExFunc)(AeLogRecordType iType, AeUInt32
iCategory, AeChar *pFormat, ...));
```

where:

- `pLogExFunc` – pointer to the customer extended log function
- `iType` – indicates the type of log message. As defined in *AeOS.h* (`struct _AeLogRecordType`), the possible types of log message, from least to most verbose, are None, Error, Warning, Info, Debug, and Trace.
- `iCategory` – indicates the category of log message. As defined in *AeOS.h* (under `Log message Category Values`), the possible categories for log messages are `NONE`, `NETWORK`, `SERVER_STATUS`, `DATA_QUEUE`, `REMOTE_SESSION`, `FILE_UPLOAD`, and `FILE_DOWNLOAD`
- `pFormat` – pointer to the format.

Description: Registers custom extended log function. When `plogExFunc` is NULL, the default log function (*AeLogEx*) is set.

Include file: AeInterface.h.

AeGetErrorString()

```
AeChar *AeGetErrorString(AeError iError)
```

where:

- `iError` – error code, as defined in *AeError.h*

Description: Returns a description of an error. The function returns a pointer to the static buffer that contains a textual description of the error corresponding to *iError*. The caller must not modify the content of this string. Errors are defined in *AeError.h*.

Include file: AeInterface.h.

AeDRMGetServerStatus

```
AeError AeDRMGetServerStatus(AeInt32 iId, AeBool *pbOnline)
```

where:

- `pbOnline` - the status of the connection to the Axeda Enterprise Server - 1 indicates online, 0 indicates offline
- `iId` - the server id

Description: Returns the status of the connection with the Axeda Enterprise Server.

Include file: AeInterface.h.

AeDRMSetQueueSize()

```
void AeDRMSetQueueSize(AeInt32 iSize)
```

where:

- `iSize` – maximum queue memory size in bytes.

Description: Configures the maximum memory size for the internal data queue. The data queue is used to accumulate asset data submitted through `AeDRMPostXXX()` functions before it is delivered to the specified Axeda Enterprise Server. (*Note: there is no minimum memory size for the internal data queue.*)

Include file: AeInterface.h.

AeDRMSetRetryPeriod()

```
void AeDRMSetRetryPeriod(AeTimeValue *pPeriod)
```

where:

- `pPeriod` – period of time that Agent Embedded will wait before trying to re-establish communication with the Axeda Enterprise Server; the default period is 30 seconds; supports values in seconds and microseconds.

Description: Configures the retry period for communication failures. If Agent Embedded detects a failure during communications with the Axeda Enterprise Server, it stops exchanging data with that server for the specified period of time and then retries.

Include file: AeInterface.h.

AeDRMSetTimeStampMode()

```
void AeDRMSetTimeStampMode(AeBool bServerMode);
```

where:

- `bServerMode` – Boolean value, indicating whether to enable server timestamp mode (pass `AeTrue` to this function)

Description: Enables or disables *server* timestamp mode. By default, Agent Embedded uses the system time of the local machine to generate timestamps for the messages sent to the Axeda Enterprise Server (*local* timestamp mode). You can override this default setting by enabling *server* timestamp mode. In this mode, the Enterprise Server uses its own system time to process the messages.

Include file: AeInterface.h.

AeDRMSetLogLevel()

```
void AeDRMSetLogLevel(AeLogLevel iLevel);
```

where:

- `iLevel` – the maximum log level; the possible levels are:
  - None (*AeLogNone*)
  - Errors only (*AeLogError*)
  - Errors and Warnings (*AeLogWarning*)
  - Information (*AeLogInfo*)
  - Errors, Warnings, and Debug messages (*AeLogDebug*)
  - Errors, Warnings, Debug, and Trace (*AeLogTrace*).

Description: Configures the maximum log level; the default log level is Info (*AeLogInfo*). Agent Embedded logs messages through the calls to *AeLog()*. Only messages whose level is less than or equal to the current logging level are logged. `AeDRMSetLogLevel (AeLogNone)` is equivalent to `AeDRMSetDebug(AeFalse)`. `AeDRMSetLogLevel(AeLogDebug)` is equivalent to `AeDRMSetDebug(AeTrue).`

Include file: AeInterface.h.

AeDRMSetDebug()

```
void AeDRMSetDebug(AeBool bDebug);
```

where:

- `bDebug` – If 1 (true), log messages are enabled (disabled by default); if 0 (false), log messages are disabled

Description: Enables/disables log messages. When log messages are enabled, Agent Embedded uses calls to `AeLog(). AeDRMSetDebug(AeFalse)` is equivalent to `AeDRMSetLogLevel (AeLogNone).AeDRMSetDebug(AeTrue)` is equivalent to `AeDRMSetLogLevel(AeLogDebug).`

Include file: AeInterface.h.

AeDRMSetYieldOnIdle()

```
void AeDRMSetYieldOnIdle(AeBool bYieldOnIdle);
```

where:

- `bYieldOnIdle` – if 1 (true), *AeDRMExecute()* returns once it finds no pending tasks to carry out (idle state).

Description: Specifies whether *AeDRMExecute()* should return once it finds no pending tasks to carry out. The default behavior is to stay inside the function until the specified amount of time elapses. *AeDRMExecute()* calls *AeSleep()* to consume unused time. When you configure *AeDRMExecute()* to return on entering the idle state, you must set up the application to avoid possible tight loops by inserting brief delays between calls to *AeDRMExecute()*.

Include file: AeInterface.h.

AeDRMAddDevice()

```
AeError AeDRMAddDevice(AeDRMDeviceType iType, AeChar *pModelNumber, AeChar
*pSerialNumber, AeInt32 *piId);
```

where:

- `iType` – type of device (master or managed), as defined in *AeDRMDeviceType* in *AeTypes.h*
- `pModelNumber -` model number of the device
- `pSerialNumber -` serial number of the device
- `piId -` assigned device ID; an output parameter

Description: Adds the specified device to the list of devices managed by Agent Embedded. Creates a device entity to which the asset application refers when performing other operations.

Return values for `AeError`:

- `AeOk` - if the device is added successfully
- `AeEExist` - if an attempt to add more than one device of type AeDRMDeviceMaster, or if a device with the specified model/serial number combination was already added.
- `AeEMemory` - the system is out of memory.

Include file: AeInterface.h.

AeDRMAddServer()

```
AeError AeDRMAddServer(AeDRMServerConfigType iType, AeChar *pURL, AeChar
*pOwner, AeTimeValue *pPingRate, AeInt32 *piId);
```

where:

- `iType` – type of server (primary, additional, or backup), as defined in *AeDRMServerConfigType* in *AeTypes.h*.
- `pURL` – address (URL) and port of the server. Axeda recommends that you configure your Axeda Enterprise Server and devices to use secure communications through SSL. To ensure the use of SSL, start the URL with `https://`.

  Use the following syntax for the URL:

  `https://<server-name>:<port>/eMessage` (secure communications) or
  `http://<server-name>:<port>/eMessage` (non-secure communications)

  where `<server-name>` is replaced with the Axeda Enterprise Server name, and <port> is replaced with the port number for the server (use 443 for SSL communications). For example, https://drm.axeda.com:443/eMessage.
- `pOwner` – name of the target database
- `pPingRate` – rate at which this device contacts, or sends ping messages to, the Axeda Enterprise Server; this server supports values defined in seconds and microseconds, for example, 5 seconds and 500000 microseconds, which is 5.5 seconds
- `piId` – assigned server ID; an output parameter

Description: Adds the destination Axeda Enterprise Server to the configuration. You can define multiple servers for a device.

Return values for `AeError`:

- `AeOK` — if the server was added successfully
- `AeEMemory` — out of memory
- `AeEExist` — if the type of server already has been added (only one primary and one backup are allowed)
- `AeBadURL` — if the URL provided is invalid

Include file: AeInterface.h.

AeDRMAddRemoteSession()

```
AeError AeDRMAddRemoteSession(AeInt32 iDeviceId, AeChar *pName, AeChar
*pDescription, AeChar *pType, AeChar *pServer, AeUInt16 iPort);
```

where:

- `iDeviceId` – identifier of the asset (device) for which remote session is configured
- `pName` – name of the remote session (unique among all sessions) for this asset
- `pDescription` – description of the session
- `pType` – type of remote session. The supported types are as follows:
  - `auto` – The Application Bridge is launched automatically by the Axeda Enterprise Server when the user starts the remote session.
  - `manual` – The Application Bridge is expected to be launched by other means (such as by a third-party application).
  - `desktop` – Same as `auto`, but the session is used specifically for desktop sharing.
  - `browser` – Same as `auto`, but the session is used specifically for web browsing.
  - `telnet` – Telnet emulator is launched automatically by the Axeda Enterprise Server when the user starts the remote session (the session is used for telnet).
  - `ssh` – SSH emulator is launched automatically by the Axeda Enterprise Server when the user starts the remote session (the session is used for SSH).
- `pServer` – destination server host (host name or IP address) identifying the remote computer running the session, for example, Telnet server or Web server. Agent Embedded supports both IPv4 (*nnn.nnn.nnn.nnn*) and IPv6 address formats.
- `iPort` – incoming port number on which the remote server is listening, for example, 23 (Telnet) or 80 (HTTP).

Description: Configures Agent Embedded for the specified type of remote session. For a particular remote session, the name must be unique for the specified device. Note that this function does not start a remote session but prepares Agent Embedded to support a particular type of remote session.

Return Values for `AeError`:

- `AeEInternal` — if support for remote sessions has been disabled in your configuration (that is, ENABLE_REMOTE_SESSION is not defined)
- `AeEOK` — if the creation of the remote session was successful
- `AeEMemory` — if the amount of memory available is insufficient to complete the request
- `AeEExist` — if a remote session for the specified device and name already exists

Include file: AeInterface.h.

AeWebSetVersion()

```
AeError AeWebSetVersion(AeInt32 iServerId, AeWebVersion iVersion);
```

where:

- `iServerId` – ID of the Axeda Enterprise Server defined for communications with Web Client
- `iVersion` – HTTP version for communications; valid values are defined in *AeTypes.h*, `AeWebVersion` ("1.0" or "1.1;" 1.1 is the default)*)*

Description: Configures the HTTP version to use for communications with the Axeda Enterprise Server.

Return values for `AeError`:

- `AeEOK` — if the HTTP version was set successfully
- `AeEInvalidArg` — if `iVersion` is not 1.1 or 1.0 or if `iServerId` is invalid

Include file: AeInterface.h.

AeWebSetPersistent()

```
AeError AeWebSetPersistent(AeInt32 iServerId, AeBool bPersistent);
```

where:

- `iServerId` – ID of server defined for communications with Web Client
- `bPersistent` – If true, HTTP/1.1 persistent connection usage is enabled; if false, HTTP/1.1 persistent connection usage is disabled (the default)

Description: Enables/disables persistent connection usage; applies to HTTP/1.1 communications only. The default value is disabled.

Return values for `AeError`:

- `AeEOK` — if the action succeeded (persistent connection was enabled or disabled)
- `AeEInvalidArg` — if `iServerId` is invalid

Include file: AeInterface.h.

AeWebSetTimeout()

```
AeError AeWebSetTimeout(AeInt32 iServerId, AeTimeValue *pTimeOut);
```

where:

- `iServerId` – ID of the Axeda Enterprise Server defined for communications with Web Client
- `pTimeOut` – amount of time after which a connection is timed-out (the default timeout period is 30 seconds); supports values defined in seconds and microseconds

Description: Configures the communication timeout period. When *AeDRMExecute()* does not detect any activity with the specified Axeda Enterprise Server within the timeout period, the connection is considered timed out and the corresponding synchronous error is reported.

Return values for `AeError`:

- `AeEOK` — if all succeeded
- `AeEInvalidArg` — if `iServerId` is invalid

Include file: AeInterface.h.

AeWebSetProxy()

```
AeError AeWebSetProxy(AeWebProxyProtocol iProto, AeChar *pHost, AeUInt16
iPort, AeChar *pUser, AeChar *pPassword);
```

where:

- `iProto` – proxy server for communications; supported value: `HTTP`, as defined in *AeWebProxyProtocol* in *AeTypes.h*
- `pHost` – host name of the proxy server
- `iPort` – port number for communications on the proxy server
- `pUser` – if needed, user name (non-NULL value) for proxy authentication
- `pPassword` – if needed, user's password (non-NULL value) for proxy authentication

Description: Configures proxy server, including user information if needed for authentication.

The Web Client can support both HTTP and SOCKS proxy servers. The Agent Embedded Toolkit supports Basic and NTLM HTTP authentication schemes, and Username/Password SOCKSv5 authentication method. At this time, it does NOT support Digest HTTP authentication scheme.

Return values for `AeError`:

- `AeEOK` — if configuration succeeds
- `AeEInvalidArg` — if `iProto` is not `AeWebProxyProtoHTTP`

Include file: AeInterface.h.

AeWebSetSSL()

```
AeError AeWebSetSSL(AeWebCryptoLevel iLevel, AeBool bServerAuth, AeChar
*pCACertFile);
```

where:

- `iLevel` – encryption level for communications with the Axeda Enterprise Server; valid values are none (no encryption (SSL disabled), low (RSA, RC4 (40 bit), MD5), medium (RSA, RC4 (128 bit), MD5), or high (RSA, 3DES (168 bit), SHA1) as defined in *AeWebCryptoLevel* in AeTypes.h.
- `bServerAuth –` if true, server certificates are validated; if false, server certificates are not validated
- `pCACertFile –` for server certificate validation; specifies a certificate from a trusted CA chain file.

Description: Configures SSL and Agent-Server encryption.

Note: *If this function is defined, then Open SSL and Enable SSL must be defined as well, as* `defined(HAVE_OPENSSL) && defined(ENABLE_SSL).`

Return values for `AeError`:

- `AeEOK` — if configuration succeeds
- `AeESSLGeneral` — if an error occurs, or if SSL support was compiled out

Include file: AeInterface.h.

Callback Registration Functions and Their Callback Functions

For background information on callback registration functions, refer to *"Callback Registration Functions"* on page 2-12 of this reference. Use callback registration functions to enable Axeda Agent Embedded to set and unset callback values to your asset application. This section includes the information for each callback function with each callback registration function.

The prototypes for these functions are in *AeInterface.h*. To locate the prototypes in this file, open it using an editor such as Notepad. Then, search for the function name, without the parentheses. For example, search for `AeDRMSetOnWebError`.

AeDRMSetOnWebError()

```
void AeDRMSetOnWebError(OnWebErrorCallback *pCallback);
```

where:

- `pCallback -` pointer to the callback function to be registered. The format that you need to use for the callback function is described in the section below, *OnWebErrorCallback Function*.

Description: Registers the callback invoked on an asynchronous communication error. When a communication error occurs while Agent Embedded is performing pending tasks in *AeDRMExecute()*, the callback registered by this function is invoked.

Include file: AeInterface.h.

OnWebErrorCallback Function

Use this application-defined callback function with the *AeDRMSetOnWebError()* function, in which *pCallback* defines a pointer to this callback function.

OnWebErrorCallback is a placeholder for the actual name you decide to use in your application. Whatever you decide to name it, the callback function must have the following signature:

```
void OnWebErrorCallback(AeInt32 iServerId, AeError iError);
```

where:

- `iServerId` – server ID of the Axeda Enterprise Server that caused the error
- `iError –` error code, as defined in *AeError.h*

AeDRMSetOnDeviceRegistered()

```
void AeDRMSetOnDeviceRegistered(OnDeviceRegisteredCallback *pCallback);
```

where:

- `pCallback` – pointer to the callback function to be registered. The format that you need to use for the callback function is described in the section below, *OnDeviceRegisteredCallback Function*.

Description: Registers the callback invoked on device registration. The callback registered by this function is invoked when a device is successfully registered while in *AeDRMExecute()*.

Include file: AeInterface.h.

OnDeviceRegisteredCallback Function

Use this application-defined callback function with the *AeDRMSetOnDevice Registered()* function, where *pCallback* defines a pointer to this callback function.

OnDeviceRegisteredCallback is a placeholder for the actual name you decide to use in your application. Whatever you decide to name it, the callback function must have the following signature:

```
void OnDeviceRegisteredCallback(AeInt32 iDeviceId);
```

where:

- `iDeviceId` – ID of registered device

AeDRMSetOnQueueStatus()

```
void AeDRMSetOnQueueStatus(QueueStatusCallback *pCallback);
```

where:

- `pCallback` – pointer to the callback function to be registered. The format that you need to use for the callback function is described in the section below, *OnQueueStatusCallback Function*.

Description: Registers the callback function invoked when the status of the internal data queue changes. Whenever the status of the internal data queue is changed, the callback function registered by this function is invoked.

Include file: AeInterface.h.

OnQueueStatusCallback Function

Use this application-defined callback function with the *AeDRMSetOnQueueStatus()* function, where *pCallback* defines a pointer to this callback function.

OnQueueStatusCallback is a placeholder for the actual name you decide to use in your application. Whatever you decide to name it, the callback function must have the following signature:

```
void QueueStatusCallback(AeDRMQueueStatus iNewStatus);
```

where:

- `iNewStatus` - new queue status. The possible values are Empty, Non-Empty, and Full.

AeDRMSetOnRemoteSessionEnd()

```
void AeDRMSetOnRemoteSessionEnd(void (*pCallback)(AeRemoteInterface
*pInterface));
```

where

- `pCallback` – pointer to the callback function to be registered. The format that you need to use for the callback function is described in the section below, *OnRemoteSessionEndCallback Function*.

Description: Registers the callback function invoked immediately after the end of a remote session. Whenever a remote session ends, this function invokes the callback function.

Include file: AeInterface.h.

OnRemoteSessionEndCallback Function

Use this application-defined callback function with the function, *AeDRMSetOnRemote SessionEnd()*, where `pCallback` defines a pointer to this callback function. This callback accepts the data structure that describes the remote session, which is defined in *AeTypes.h*.

OnRemoteSessionEndCallback is a placeholder for the actual name you decide to use in your application. Whatever you decide to name it, the callback function must have the following signature:

```
void RemoteSessionEndCallback(AeRemoteInterface *pInterface)
```

where:

- `pInterface` - a pointer to the information about the remote session that just ended.

Description: Indicates that the remote session described by pInterface has ended.

Include file: AeInterface.h.

AeDRMSetOnRemoteSessionStart()

```
void AeDRMSetOnRemoteSessionStart(AeBool (*pCallback)(AeRemoteInterface
*pInterface));
```

where

- `pCallback` – pointer to the callback function to be registered. The format that you need to use for the callback function is described in the section below, *OnRemoteSessionStartCallback Function*.

Description: Registers the callback function invoked when Agent Embedded is about to start a remote session. Whenever Agent Embedded is about to start a remote session, the callback function registered by this function is invoked.

Include file: AeInterface.h.

OnRemoteSessionStartCallback Function

Use this application-defined callback function with the function, *AeDRMSetOnRemote SessionStart()*, where `pCallback` defines a pointer to this callback function. This callback accepts the structure that describes the remote session, which is defined in *AeTypes.h*.

OnRemoteSessionStartCallback is a placeholder for the actual name you decide to use in your application. Whatever you decide to name it, the callback function must have the following signature:

```
AeBool RemoteSessionStartCallback(AeRemoteInterface *pInterface)
```

where:

- `pInterface` - a pointer to the information about the remote session that the Agent Embedded was about to start.

Description: Returns a Boolean, indicating if Agent Embedded was allowed to start the remote session (`1` indicates that the remote session was allowed, `0` that the remote session was denied). When a remote session is denied from an asset running Agent Embedded, the message, “Inhibited by agent” appears in the Asset Dashboard, in the Audit Log for the asset.

Include file: AeInterface.h.

AeDRMSetOnSOAPMethod()

```
void AeDRMSetOnSOAPMethod(OnSOAPMethodCallback *pCallback);
```

where:

- `pCallback` – pointer to the callback function to be registered. The format that you need to use for the callback function is described in the section below, *OnSOAPMethodCallback Function*.

Description: Registers the callback function invoked on receiving a generic SOAP method. When a SOAP method is received from the Axeda Enterprise Server and the method is not known (that is, not handled by any of the callback functions registered through *AeDRMSetOnCommandXXX()*), the callback registered by this function is invoked.

The received SOAP method may be processed by the application using the *AeDRMSOAPXXX()* function family.

Include file: AeInterface.h.

OnSOAPMethodCallback Function

Use this application-defined callback function with the *AeDRMSetOnSOAPMethod()* function, where *pCallback* defines a pointer to this callback function.

OnSOAPMethodCallback is a placeholder for the actual name you decide to use in your application. Whatever you decide to name it, the callback function must have the following signature:

```
void OnSOAPMethodCallback(AeInt32 iDeviceId, AeHandle pMethod,
AeDRMSOAPCommandStatus *pStatus);
```

where:

- `iDeviceId` – ID of the target device
- `pMethod` – handle to the SOAP method received from server
- `pStatus` - pointer to the structure for invocation results. `SOAP command status values are defined in` *AeTypes.h* `and include Failed, Invalid Params, Not Implemented, and Deferred.`

AeDRMSetOnSOAPMethodEx()

```
void AeDRMSetOnSOAPMethodEx(OnSOAPMethodExCallback *pCallback);
```

where:

- `pCallback` – pointer to the callback function to be registered. The format that you need to use for the callback function is described in the section below, *OnSOAPMethodExCallback Function*.

Description: Registers the callback invoked on receiving a generic SOAP method.

The application can process the received SOAP method using the *AeDRMSOAPXXX()* family of functions.

Include file: AeInterface.h.

OnSOAPMethodExCallback Function

Use this application-defined callback function with the *AeDRMSetOnSOAP MethodEx()* function, where *pCallback* defines a pointer to this callback function.

OnSOAPMethodExCallback is a placeholder for the actual name you decide to use in your application. Whatever you decide to name it, the callback function must have the following signature:

```
void OnSOAPMethodExCallback(AeInt32 iDeviceId, AeInt32 iServerId, AeHandle
pMethod, AeDRMSOAPCommandId *pSOAPId, AeDRMSOAPCommandStatus *pStatus);
```

where:

- `iDeviceId` – ID of the target device
- `iServerId` – ID of the Axeda Enterprise Server that sent the SOAP method
- `pMethod` – handle to the SOAP method received from server
- `pSOAPId` – handle to the SOAP method ID
- `pStatus` - pointer to the structure for invocation results. SOAP command status values are defined in *AeTypes.h* and include `Failed, Invalid Params, Not Implemented`, and `Deferred.`

Similar to *OnSOAPMethodCallback*, this function accepts two additional parameters, one for the ID of the Axeda Enterprise Server that sent the SOAP method (`iServerId`) and one that points to the ID of the SOAP method (`pSOAPId`). You may want to use these additional parameters to submit invocation results asynchronously (that is, using *AeDRMPostSOAPCommandStatus()*). To indicate that the method is executed asynchronously, the callback must return `AE_DRM_SOAP_COMMAND_STATUS_DEFERRED` in *pStatus->iStatus*.

AeDRMSetOnCommandSetTag()

```
void AeDRMSetOnCommandSetTag(OnCommandSetTagCallback *pCallback);
```

where:

- `pCallback` – pointer to the callback function to be registered. The format that you need to use for the callback function is described in the section below, *OnCommandSetTagCallback Function*.

Description: Registers the callback invoked on receiving the SOAP method, `DynamicData.SetTag`.

Include file: AeInterface.h.

OnCommandSetTagCallback Function

Use this application-defined callback function with the *AeDRMSetOnCommand SetTag()* function, where *pCallback* defines a pointer to this callback function.

OnCommandSetTagCallback is a placeholder for the actual name you decide to use in your application. Whatever you decide to name it, the callback function must have the following signature:

```
void OnCommandSetTagCallback(AeInt32 iDeviceId, AeDRMDataItem *pDataItem,
AeDRMSOAPCommandStatus *pStatus);
```

where:

- `iDeviceId` – ID of the target device
- `pDataItem` – name of the data item and its new value
- `pStatus` - pointer to the structure for invocation results. SOAP command status values are defined in *AeTypes.h* and include `Failed, Invalid Params, Not Implemented`, and `Deferred.`

AeDRMSetOnFileDownloadBegin()

```
void AeDRMSetOnFileDownloadBegin(OnFileDownloadBegin *pCallback);
```

where:

- `pCallback` – callback invoked at the start of a download of series of one or more files; the callback function indicates data processing status by returning *AeTrue* (success) or *AeFalse* (not successful). The format of the callback function is described in the section below, *OnFileDownloadBeginCallback Function*.

Description: Registers the callback invoked when starting the download for a series of one or more files. The callback must return *AeTrue* if the user application intends to perform custom processing on the downloaded files. In this case, Agent Embedded will pass the incoming file data to the application using the callback registered with *AeDRMSetOnFileDownloadData()*. The callback must return *AeFalse*, if Agent Embedded should process downloaded files internally (that is, store them in the local file system).

Include file: AeInterface.h.

OnFileDownloadBeginCallback Function

Use this application-defined callback function with the *AeDRMSetOnFileDownload Begin()* function, where *pCallback* defines a pointer to this callback function.

OnFileDownloadBeginCallback is a placeholder for the actual name you decide to use in your application. Whatever you decide to name it, the callback function must have the following signature:

```
AeBool OnFileDownloadBeginCallback(AeInt32 iDeviceId, AePointer
*ppUserData);
```

where:

- `iDeviceId` – ID of the target device
- `ppUserData` – (optional) output parameter, completed by this callback

This callback may optionally fill the `ppUserData` output parameter with a pointer to the application's structure that is associated with the file download. Agent Embedded will use this pointer in subsequent invocations of callbacks registered with *AeDRMSetOnFileDownloadData()* and *AeDRMSetOnFileDownloadEnd()*.

AeDRMSetOnFileDownloadData() and AeDRMSetOnFileDownloadData64()

```
void AeDRMSetOnFileDownloadData(OnFileDownloadDataCallback *pCallback);
void AeDRMSetOnFileDownloadData64(OnFileDownloadData64Callback
(*pCallback));
```

\where:

- `pCallback` – pointer to a function to be called for each chunk of file data received. The format of the callback is described in the section below, *OnFileDownloadDataCallback and OnFileDownloadData64Callback Functions*.

Description: Registers the callback that is invoked when Agent Embedded receives a portion of a downloaded file. If 64-bit file operations are enabled, *AeDRMSetOnFileDownloadData()* maps automatically to *AeDRmSetOnFileDownloadData64()*.

Include file: AeInterface.h.

OnFileDownloadDataCallback and OnFileDownloadData64Callback Functions

Use this application-defined callback functions with the *AeDRMSetOnFileDownload Data()* and *AeDRMSetOnFileDownloadData64()* functions, in which *pCallback* defines a pointer to the related callback function. *OnFileDownloadDataCallback* and *OnFileDownloadDataExCallback* are placeholders for the actual names you decide to use in your application. Whatever you decide to name them, the callback functions must have the following signatures, respectively:

```
AeBool OnFileDownloadDataCallback(AeInt32 iDeviceId, AeFileStat
*pFile, AeChar *pData, AeInt32 iSize, AePointer pUserData))
AeBool OnFileDownloadData64Callback(AeInt32 iDeviceId, AeFileStat
*pFile, AeChar *pData, AeInt32 iSize, AePointer pUserData)
```

where:

- `iDeviceId` – ID of the target device
- `*pFile` – pointer to the description of the file for download. The structure, *AeFileStat* in *AeTypes.h,*is used to communicate file attributes from Agent Embedded to the application during a file download. The attributes in the structure include
  - `*pName`, which is a pointer to the name of the file
  - `iType`, which is an enumerated list of file types (defined by `enum AeFileType`) and includes `Unknown`, `Regular`, and `Directory`.
  - `iSize`, whose value indicates the size of the file in bytes.

- iMTime, which is the date and time that the file was last modified
- *pData – pointer to the block of received file data
- iSize – pointer to the number of bytes in the block of received file data
- pUserData – pointer previously returned by the application through the callback registered with *AeDRMSetOnFileDownloadBegin()*

When this callback is invoked with a NULL block (*pData* and *iSize* are zero), the last block of the file described by *pFile* has been downloaded. This callback indicates data processing status by returning *AeTrue* in the case of success, or *AeFalse* otherwise. When ENABLE_LARGEFILE64 is defined, Agent Embedded automatically uses the *OnFileDownloadData64Callback()* version of this function.

This callback returns one of the following values:

- AeTrue – if data processing status is successful;
- AeFalse – if data processing status is unsuccessful

If this callback returns AeFalse, the file transfer is halted with a status of AeFailed.

AeDRMSetOnFileDownloadEnd()

```
void AeDRMSetOnFileDownloadEnd(OnFileDownloadEndCallback *pCallback);
```

where:

- `pCallback` – pointer to the callback to be registered. The format that you need to use for the callback function is described in the section below, *OnFileDownloadEndCallback Function*.

Description: Registers the callback that is invoked when Agent Embedded either encounters an error during the file download, or when all files in the series are received successfully. The callback indicates the file download status by returning `AeTrue` in the case of success or `AeFalse` otherwise.

Include file: AeInterface.h.

OnFileDownloadEndCallback Function

Use this application-defined callback function with the *AeDRMSetOnFileDownload End()* function, where *pCallback* defines a pointer to this callback function.

OnFileDownloadEndCallback is a placeholder for the actual name you decide to use in your application. Whatever you decide to name it, the callback function must have the following signature:

```
AeBool OnFileDownloadEndCallback(AeInt32 iDeviceId, AeBool bOK,
AePointer pUserData)
```

where:

- `iDeviceId` – ID of the target device
- `bOK` – identifies if the download failed (*AeFalse*) or was successful (*AeTrue*) (Boolean)
- `pUserData` – pointer previously returned by the application through the callback registered with `AeDRMSetOnFileDownloadBegin()`

AeDRMSetOnFileUploadBegin() and AeDRMSetOnFileUploadBegin64()

```
void AeDRMSetOnFileUploadBegin(OnFileUploadBeginCallback *pCallback);
void AeDRMSetOnFileUploadBegin64(OnFileUploadBegin64Callback *pCallback);
```

where:

- `pCallback` – pointer to the callback function to be registered. The format that you need to use for the callback functions is described in the section below, *OnFileUploadBeginCallback and OnFileUploadBegin64Callback Functions*.

Description: Registers the callback invoked on receiving the `File Upload` SOAP method. The callback must return `AeTrue` if the user application intends to provide data for the uploaded files. In this case, Agent Embedded will request content of the uploaded files from the application through a callback registered with *AeDRMSetOnFileUploadData()*. The callback must return `AeFalse` if Agent Embedded should process uploaded files internally (that is, retrieve them from the local file system). When `ENABLE_LARGEFILE64` is defined, Agent Embedded automatically uses the *AeDRMSetOnFileUploadBegin64()* version of this function.

Include file: AeInterface.h.

OnFileUploadBeginCallback and OnFileUploadBegin64Callback Functions

Use this application-defined callback functions with the *AeDRMSetOnFileUpload Begin()* and *AeDRMSetOnFileUploadBegin64()* functions, respectively, in which *pCallback* defines a pointer to these callback functions.

OnFileUploadBeginCallback and *OnFileUploadBeginExCallback* are placeholders for the actual names you decide to use in your application. Whatever you decide to name them, the callback functions must have the following signatures, respectively:

```
AeBool OnFileUploadBeginCallback(AeInt32 iDeviceId, AeFileUploadSpec
**ppUploads, AePointer *ppUserData)
AeBool OnFileUploadBegin64Callback(AeInt32 iDeviceId, AeFileUploadSpec
**ppUploads, AePointer *ppUserData)
```

where:

- `iDeviceId` – ID of the target device
- `ppUploads` – NULL-terminated array of specifications for the uploaded file
- `ppUserData` – (optional output parameter); pointer to application's structure associated with the file download, as defined in `AeFileUploadSpec` in *AeTypes.h*

When `ENABLE_LARGEFILE64` is defined, Agent Embedded automatically uses the *OnFileUploadBegin64Callback()* version of this function.

This callback may optionally fill the *ppUserData* output parameter with a pointer to the application’s structure associated with the file upload. Agent Embedded will use this pointer in subsequent invocations of callbacks registered with *AeDRMSetOnFileUploadData()*, *AeDRMSetOnFileUploadEnd()*, and *AeDRMSetOnFileTransferEvent()*.

AeDRMSetOnFileUploadBeginEx() and AeDRMSetOnFileUploadBeginEx64()

```
void AeDRMSetOnFileUploadBeginEx(OnFileUploadBeginExCallback *pCallback);
```

```
void AeDRMSetOnFileUploadBeginEx64(OnFileUploadBeginEx64Callback
*pCallback);
```

where:

- `pCallback` – pointer to the callback to be registered. The format that you need to use for the callback function is described in the section below, *OnFileUploadBeginExCallback and OnFileUploadBeginEx64Callback Functions*.

Description: Registers the callback invoked when Agent Embedded receives the `File Upload` SOAP method, like *AeDRMSetOnFileUploadBegin()* and *AeDRMSetOnFileUploadBegin64()*. The difference from those functions lies in the callbacks, which accept additional upload parameters.

If the callback for either of these functions returns `AeTrue`, the user application intends to provide data for uploaded files. If the callback returns `AeFalse`*,* Agent Embedded should process uploaded files internally.

When `ENABLE_LARGEFILE64` is defined, Agent Embedded automatically uses the *AeDRMSetOnFileUploadBeginEx64()* version of this function.

Include file: AeInterface.h.

OnFileUploadBeginExCallback and OnFileUploadBeginEx64Callback Functions

Use this application-defined callback functions with the *AeDRMSetOnFileUpload BeginEx()* and *AeDRMSetOnFileUploadBeginEx64()* functions, in which `pCallback` defines a pointer to these callback functions.

OnFileUploadBeginExCallback and *OnFileUploadBeginEx64Callback* are placeholders for the actual names you decide to use in your application. Whatever you decide to name it, the callback functions must have the following signatures, respectively:

```
AeBool OnFileUploadBeginExCallback(AeInt32 iDeviceId, AeFileUploadSpec
**ppUploads, AeFileUploadExecuteParam *pParam, AePointer *ppUserData)
AeBool OnFileUploadBeginEx64Callback(AeInt32 iDeviceId, AeFileUploadSpec
**ppUploads, AeFileUploadExecuteParam *pParam, AePointer *ppUserData)
```

where:

- `iDeviceId` – ID of the target device
- `ppUploads` – NULL-terminated array of specifications for the uploaded file. `AeFileUploadSpec` is defined in *AeTypes.h*. The application uses this structure to request a file upload (one structure is used for each file). In this situation, the `iPosition` field is ignored by Agent Embedded. When the upload command is received from the Axeda Enterprise Server, `AeFileUploadSpec` is passed from Agent Embedded to the application (one structure per file). In this situation, the application must start with the file position indicated through `iPosition`, if the application intends to provide file data. The setting of `iPosition` depends on the size of the file.
- `pParam` – pointer to additional upload parameters
- `ppUserData` – (optional output parameter); pointer to application's structure associated with the file download, as defined in `AeFileUploadSpec` in *AeTypes.h*

This callback must return `AeTrue` if the user application intends to provide data for the uploaded files. In this case, Agent Embedded will request content of the uploaded files from the application through a callback registered with *AeDRMSetOnFileUpload DataEx()*.

This callback must return `AeFalse` if Agent Embedded should process the uploaded files internally (that is, retrieve them from the local file system).

This callback function may optionally fill the `ppUserData` output parameter with a pointer to the application's structure associated with the file upload. Agent Embedded will use this pointer in subsequent invocations of callbacks registered with *AeDRMSetOnFileUploadData()*, *AeDRMSetOnFileUploadEnd()*, and *AeDRMSetOnFileTransferEvent()*.

AeDRMSetOnFileUploadData()and AeDRMSetOnFileUploadData64()

```
void AeDRMSetOnFileUploadData(OnFileUploadDataCallback *pCallback);
void AeDRMSetOnFileUploadData64(OnFileUploadDataCallback *pCallback);
```

where:

- `pCallback` – pointer to the callback to be registered. The format that you need to use for the callback is described in the section below, *OnFileUploadDataCallback and OnFileUploadData64Callback Functions*.

Description: Registers the callback invoked when Agent Embedded requires more file data for the upload. If the callback function returns *AeTrue,* the next block of file data is available for upload (successful); if it returns *AeFalse*, an error occurred.

When `ENABLE_LARGEFILE64` is defined, Agent Embedded automatically uses the *AeDRMSetOnFileUploadData64()* version of this function.

Include file: AeInterface.h.

OnFileUploadDataCallback and OnFileUploadData64Callback Functions

Use this application-defined callback functions with the *AeDRMSetOnFileUploadData* and *AeDRMSetOnFileUploadData64* functions, in which *pCallback* defines a pointer to the related callback function.

OnFileUploadDataCallback and *OnFileUploadData64Callback* are placeholders for the actual names you decide to use in your application. Whatever you decide to name them, the callback functions must have the following signatures, respectively:

```
AeBool OnFileUploadDataCallback(AeInt32 iDeviceId, AeFileStat **ppFile,
AeChar **ppData, AeInt32 *piSize, AePointer pUserData)
AeBool OnFileUploadData64Callback(AeInt32 iDeviceId, AeFileStat **ppFile,
AeChar **ppData, AeInt32 *piSize, AePointer pUserData))
```

where:

- `iDeviceId` – ID of the target device
- `ppFile` – pointer to an appropriate file descriptor; provided by application
- `ppData` – pointer to a buffer that contains the next block of data to be included in the upload; provided by application
- `piSize` – pointer to the size of the buffer that contains the next block of data to be included in the upload; provided by the application

Note: *If you want to upload a file without having access to the entire file content at the start of the upload (and without knowing the total size), you can do so. For details, refer to the section, "Uploading Files without Access to the Entire Content at the Start of the Upload" on page 4-13, for details.*

- `pUserData` – pointer previously returned by the application through the callback registered with *AeDRMSetOnFileUploadBegin()*

The last block of the file described by `*ppFile` is indicated by a NULL block (`*ppData` and `*piSize` are zero). The end of the upload should be indicated by filling `ppFile` with a NULL pointer. Note that the file descriptor and the buffer returned by the application must remain intact at least until this callback (or the one registered with *AeDRMSetOnFileUploadEnd()* or with *AeDRMSetOnFileTransferEvent()*) is invoked again for this same upload. The application is responsible for managing the memory of these objects.

A NULL block (both `*ppData` and `*piSize` are zero) returned by the application will cause Agent Embedded to defer file transfer processing until the next call to *AeDRMExecute()*, at which time the callback will be invoked again.

When `ENABLE_LARGEFILE64` is defined, Agent Embedded automatically uses the *OnFileUploadData64Callback()* version of this function.

AeDRMSetOnFileUploadEnd()

```
void AeDRMSetOnFileUploadEnd(OnFileUploadEndCallback *pCallback);
```

where:

- `pCallback - pointer to` callback invoked when Agent Embedded either encounters an error during the file upload, or when all files are uploaded successfully. The format of the callback is described in the section below, *OnFileUploadEndCallback Function*.

Description: Registers the callback invoked when Agent Embedded either encounters an error during the file upload, or when all files are uploaded successfully.

Include file: AeInterface.h.

OnFileUploadEndCallback Function

Use this application-defined callback function with the *AeDRMSetOnFileUploadEnd* function, where *pCallback* defines a pointer to this function.

OnFileUploadEndCallback is a placeholder for the actual name you decide to use in your application. Whatever you decide to name it, the callback function must have the following signature:

```
void OnFileUploadEndCallback(AeInt32 iDeviceId, AeBool bOK, AePointer
pUserData)
```

where:

- `iDeviceId` – ID of the target device
- `bOK` – identifies if the upload failed (false) or succeeded (true) (Boolean)
- `ppUserData` – pointer previously returned by the application through the callback registered with *AeDRMSetOnFileUploadBegin()*

AeDRMSetOnFileTransferEvent()

```
void AeDRMSetOnFileTransferEvent(OnFileTransferEventCallback*pCallback);
```

where:

- `pCallback - pointer to` callback invoked when Agent Embedded encounters an event during the file transfer. The format of the callback is described in the section below, *OnFileTransferEventCallback Function*.

Description: Registers the callback invoked when Agent Embedded encounters an event during the file transfer.

Include file: AeInterface.h.

OnFileTransferEventCallback Function

Use this application-defined callback function with the *AeDRMSetOnPingRateUpdate* function, where *pCallback* defines a pointer to this function.

OnFileTransferEventCallback is a placeholder for the actual name you decide to use in your application. Whatever you decide to name it, the callback function must have the following signature:

```
void OnFileTransferEventCallback(AeInt32 iDeviceId, AeFileTransferEvent
iEvent, AePointer pUserData)
```

where:

- `iDeviceId` – ID of the target device
- `iEvent` – identifies the type of event that occurred during the file transfer`.` Types of file transfer events are defined under `_AeFileTransferEvent` in *AeTypes.h* and include `Cancelled`, `Paused`, `Preempted`, and `Reactivated`.
- `ppUserData` – pointer that the application previously returned through the callback registered with *AeDRMSetOnFileUploadBegin()* or *AeDRMSetOnFileDownloadBegin()*

AeDRMSetOnCommandSetTime()

```
void AeDRMSetOnCommandSetTime(OnCommandSetTimeCallback *pCallback);
```

where:

- `pCallback` – pointer to the callback function to be registered. The format that you need to use for the callback is described in the section below, *OnCommandSetTimeCallback Function*.

Description: Registers the callback invoked when Agent Embedded receives the `Set Time` SOAP Method, `EEnterpriseProxy.St.`

Include file: AeInterface.h.

OnCommandSetTimeCallback Function

Use this application-defined callback function with the *AeDRMSetOnCommandSet Time()* function, where *pCallback* defines a pointer to this function.

OnCommandSetTimeCallback is a placeholder for the actual name you decide to use in your application. Whatever you decide to name it, the callback function must have the following signature:

```
void OnCommandSetTimeCallback(AeInt32 iDeviceId, AeTimeValue *pTime,
AeInt32 *piTZOffset, AeDRMSOAPCommandStatus *pStatus);
```

where:

- `iDeviceId` – ID of the target device
- `pTime` – new time value to set on the device; may be NULL if the value is not changed. Supports values defined in seconds and microseconds.
- `piTZOffset` – new time zone offset (minutes from GMT); may be NULL if the value is not changed.
- `pStatus` - pointer to the structure for invocation results. SOAP command status values are defined in *AeTypes.h* and include `Failed, Invalid Params, Not Implemented`, and `Deferred.`

AeDRMSetOnCommandRestart()

```
void AeDRMSetOnCommandRestart(OnCommandRestartCallback *pCallback);
```

where:

- `pCallback` – pointer to the callback function to be registered. The format that you need to use for the callback is described in the section below, *OnCommandRestartCallback Function*.

Description: Registers the callback invoked when Agent Embedded receives the `Restart` SOAP method, `EEnterpriseProxy.Rs.`

Include file: AeInterface.h.

OnCommandRestartCallback Function

Use this application-defined callback function with the *AeDRMSetOnCommand Restart* function, where *pCallback* defines a pointer to this function.

OnCommandRestartCallback is a placeholder for the actual name you decide to use in your application. Whatever you decide to name it, the callback function must have the following signature:

```
void OnCommandRestartCallback(AeInt32 iDeviceId, AeBool bHard,
AeDRMSOAPCommandStatus *pStatus);
```

where:

- `iDeviceId` – ID of the target device
- `bHard` – flag indicating whether hard (1 or true) or soft (0 or false) restart is requested.
- `pStatus` - pointer to the structure for invocation results. SOAP command status values are defined in *AeTypes.h* and include `Failed, Invalid Params, Not Implemented`, and `Deferred.`

Note: *Although the Axeda Gateway does not support hard and soft restarts of managed assets, Agent Embedded DOES support hard and soft restarts for managed assets.*

AeDRMSetOnPingRateUpdate()

```
void AeDRMSetOnPingRateUpdate(OnPingRateUpdateCallback *pCallback);
```

where:

- `pCallback` – pointer to the callback function to be registered. The format that you need to use for the callback is described in the section below, *OnPingRateUpdateCallback Function*.

Description: Registers callback invoked on receiving Set Ping Rate SOAP method, `EEnterpriseProxy.Pu`. The callback is also invoked by Agent Embedded when the ping rate is restored to the original after the time period for a temporary ping rate elapses.

Include file: AeInterface.h.

OnPingRateUpdateCallback Function

Use this application-defined callback function with the *AeDRMSetOnPingRateUpdate* function, where *pCallback* defines a pointer to this function.

OnPingRateUpdateCallback is a placeholder for the actual name you decide to use in your application. Whatever you decide to name it, the callback function must have the following signature:

```
void OnPingRateUpdateCallback(AeInt32 iServerId, AeDRMPingRate
*pPingRate);
```

where:

- `iServerId` – ID of the Axeda Enterprise Server that sent the change to the ping rate
- `pPingRate - a` pointer to the `AeDRMPingRate` structure, which has two fields, `rate` and `pDuration`, The `rate` field specifies the ping rate and `pDuration` specifies the time period during which the new ping rate will be effective. Currently, however, `pDuration` is always `NULL`.

Asset Data Export Functions

For background information on aaset data export functions, refer to *"Asset Data Export Functions"* on page 2-12 of this reference. Each of the asset data export functions accepts a pointer to a structure of a specific type, which describes a particular kind of asset data (such as snapshot or alarm). The specification of the source asset is also provided for each function.

When asset data export functions are called, all strings passed must be of type UTF8-encoded. Then, the data is formatted into one of the eMessage XML elements (unless an opaque message is passed) and placed into the data queue. The data queue status callback is invoked if the queue becomes full.

AeDRMPostAlarm()

```
AeError AeDRMPostAlarm(AeInt32 iDeviceId, AeInt32 iServerId,
AeDRMQueuePriority iPriority, AeDRMAlarm *pAlarm);
```

where:

- `iDeviceId` – identifier for the asset (device) that is the source of the data item
- `iServerId` – identifier of the Axeda Enterprise Server to which the specified data item will be posted
- `iPriority` – priority level for posting the data item to the server; valid values are defined in *AeTypes.h*, `AeDRMQueuePriority` (`low`, `normal`, `high`)
- `pAlarm` – Alarm item to post, as defined in *AeTypes.h*, `AeDRMAlarm`

Description: Submits alarm for delivery to the Axeda Enterprise Server.

Return values for `AeError`:

- `AeEMemory` — if system is out of memory
- `AeEInvalidArg` — if an argument specified an invalid value

Include file: AeInterface.h.

AeDRMPostDataItem()

```
AeError AeDRMPostDataItem(AeInt32 iDeviceId, AeInt32 iServerId,
AeDRMQueuePriority iPriority, AeDRMDataItem *pDataItem);
```

where:

- `iDeviceId` – identifier for the asset (device) that is the source of the data item
- `iServerId` – identifier of the Axeda Enterprise Server to which the specified data item will be posted
- `iPriority` – priority level for posting the data item to the Axeda Enterprise Server; valid values are defined in *AeTypes.h*, `AeDRMQueuePriority` (`low`, `normal`, `high`)
- `pDataItem` - Data item to post

Description: Submits the specified data item for delivery to the Axeda Enterprise Server.

Include file: AeInterface.h.

AeDRMPostDataItemSet()

```
AeError AeDRMPostDataItemSet(AeInt32 iDeviceId, AeInt32 iServerId,
AeDRMQueuePriority iPriority, AeDRMDataItem *pDataItems[], AeInt32 iCount);
```

where:

- `iDeviceId` – identifier for the asset (device) that is the source of the set of data items
- `iServerId` – identifier of the Axeda Enterprise Server to which the specified set of data items will be posted
- `iPriority` – priority level for posting the data item set to the Axeda Enterprise Server; valid values are defined in *AeTypes.h*, `AeDRMQueuePriority` (`low`, `normal`, `high`)
- `pDataItems[]` - Set of data items to post.
- `iCount)` - Count of data items to post

Description: Submits the specified set of data items for delivery The first `iCount` elements of the data item array (`pDataItems`) will be posted to the Axeda Enterprise Server (`iServerId`) based on priority (`iPriority`).

Include file: AeInterface.h.

AeDRMPostEvent()

```
AeError AeDRMPostEvent(AeInt32 iDeviceId, AeInt32 iServerId,
AeDRMQueuePriority iPriority, AeDRMEvent *pEvent);
```

where:

- `iDeviceId` – identifier for the asset (device) that is the source of the event
- `iServerId` – identifier of the Axeda Enterprise Server to which the specified event will be posted
- `iPriority` – priority level for posting the event to the Axeda Enterprise Server; valid values are defined in *AeTypes.h*, `AeDRMQueuePriority` (`low`, `normal`, `high`)
- `pEvent` – Event to post, as defined in *AeTypes.h*, `AeDRMEvent`

Description: Submits the specified event for delivery to the Axeda Enterprise Server.

Include file: AeInterface.h.

AeDRMPostEmail()

```
AeError AeDRMPostEmail(AeInt32 iDeviceId, AeInt32 iServerId,
AeDRMQueuePriority iPriority, AeDRMEmail *pEmail);
```

where:

- `iDeviceId` – identifier for the asset (device) that is the source of the e-mail message
- `iServerId` – identifier of the Axeda Enterprise Server to which the specified e-mail message will be posted
- `iPriority` – priority level for posting the e-mail message to the Axeda Enterprise Server; valid values are defined in *AeTypes.h*, `AeDRMQueuePriority` (`low`, `normal`, `high`)
- `pEmail` – E-mail to post, as defined in `AeDRMEmail` in *AeTypes.h*

Description: Submits e-mail for delivery to the Axeda Enterprise Server.

Include file: AeInterface.h.

AeDRMPostFileUploadRequest() and AeDRMPostFileUploadRequest64()

```
AeError AeDRMPostFileUploadRequest(AeInt32 iDeviceId, AeInt32 iServerId,
AeDRMQueuePriority iPriority, AeFileUploadSpec **ppUploads);
```

```
AeError AeDRMPostFileUploadRequest64(AeInt32 iDeviceId, AeInt32 iServerId,
AeDRMQueuePriority iPriority, AeFileUploadSpec **ppUploads);
```

where:

- `iDeviceId` – identifier for the asset (device) that is the source of the file to upload
- `iServerId` – identifier of the Axeda Enterprise Server to which the file will be uploaded
- `iPriority` – priority level for uploading the file to the Axeda Enterprise Server; valid values are defined in `AeDRMQueuePriority` (`low`, `normal`, `high`), in *AeTypes.h*
- `ppUploads` – NULL-terminated array of file upload specifications, as defined in `AeFileUploadSpec` in *AeTypes.h*

Description: Submits file upload request for delivery to the Axeda Enterprise Server. When `ENABLE_LARGEFILE64` is defined, Agent Embedded automatically uses the *AeDRMPostFileUploadRequest64()* version of this function.

Include file: AeInterface.h.

AeDRMPostFileUploadRequestEx() and AeDRMPostFileUploadRequestEx64()

```
AeError AeDRMPostFileUploadRequestEx(AeInt32 iDeviceId, AeInt32 iServerId,
AeDRMQueuePriority iPriority, AeFileUploadSpec **ppUploads,
AeFileUploadRequestParam *pParam);
```

```
AeError AeDRMPostFileUploadRequestEx64(AeInt32 iDeviceId, AeInt32 iServerId,
AeDRMQueuePriority iPriority, AeFileUploadSpec **ppUploads,
AeFileUploadRequestParam *pParam);
```

where:

- `iDeviceId` – identifier of the asset (device) requesting to upload a file
- `iServerId` – identifier of the Axeda Enterprise Server to which the request will be posted
- `iPriority` – priority level for posting the request to the server; valid values are defined in `AeDRMQueuePriority` (`low`, `normal`, `high`) in *AeTypes.h*
- `ppUploads` – NULL-terminated array of file upload specifications, as defined in `AeFileUploadSpec` in *AeTypes.h*
- `pParam` – pointer to additional parameters for the file upload

Description: Similar to *AeDRMPostFileUploadRequest()*, submits a file upload request for delivery, but specifies additional upload parameters (in `pParam`). When `ENABLE_LARGEFILE64` is defined, Agent Embedded automatically uses the *AeDRMPostFileUploadRequestEx64()* version of this function.

Include file: AeInterface.h.

AeDRMPostOpaque()

```
AeError AeDRMPostOpaque(AeInt32 iDeviceId, AeInt32 iServerId,
AeDRMQueuePriority iPriority, AeChar *pData);
```

where:

- `iDeviceId` – identifier of the asset (device) that is the source of the opaque data
- `iServerId` – identifier of the Axeda Enterprise Server to which the specified opaque data will be posted
- `iPriority` – priority level for posting the opaque data to the server; valid values are defined in `AeDRMQueuePriority` (`low`, `normal`, `high`) in *AeTypes.h*
- `pData` - opaque data; XML must be formatted correctly by the device application

Description: Submits opaque data for delivery. *Opaque data* refers to a pre-formatted eMessage element that Agent Embedded will deliver as is.

Include file: AeInterface.h.

AeDRMPostSOAPCommandStatus()

```
AeError AeDRMPostSOAPCommandStatus(AeInt32 iDeviceId, AeInt32 iServerId,
AeDRMQueuePriority iPriority, AeDRMSOAPCommandId *pSOAPId,
AeDRMSOAPCommandStatus *pStatus);
```

where:

- `iDeviceId` – identifier of the asset (device) that processed the SOAP method
- `iServerId` – identifier of the Axeda Enterprise Server to which the status of the specified SOAP method will be posted
- `iPriority` – priority level for posting the status of the SOAP method to the server; valid values are defined in `AeDRMQueuePriority` (`low`, `normal`, `high`) in *AeTypes.h*
- `pSOAPId` – pointer to the identifier of the SOAP method
- `pStatus` – pointer to the status of the SOAP method. SOAP command status values are defined in AeTypes.h and include `Failed`, `Invalid Params`, `Not Implemented`, and `Deferred.`

Description: Submits the status of the processing for the specified SOAP method for delivery.

Include file: AeInterface.h.

Execution Control Functions

For background information on execution control functions, refer to *"Execution Control Function"* on page 2-13.

AeDRMExecute()

```
AeError AeDRMExecute(AeTimeValue *pTimeLimit);
```

where:

- `pTimeLimit` – the time limit for performing pending tasks; if this is NULL, the operation is synchronous and the function blocks all other processing until all of the pending tasks are completed; if this value is non-NULL, the function returns when `pTimeLimit` elapses. If the tasks complete and there is unused time, *AeDRMExecute()* calls *AeSleep()* to consume unused time.

Description: Performs pending tasks. See also *"Synchronous and Asynchronous Operations"* on page 4-6.

Include file: AeInterface.h.

AeSleep()

`void AeSleep(AeTimeValue *pTime);`

where:

- `Time` – a pointer to the time period to wait before calling *AeDRMExecute()* again

Description: You must implement this function for your operating system to enable *AeDRMExecute()* to work properly in its default behavior. The default behavior is that *AeDRMExecute()* does not return until a specified amount of time elapses; that time is specified in this function. When you use the default behavior, *AeDRMExecute()* calls *AeSleep()* to consume unused time.

> **Note:** *If you set AeDRMSetYieldOnIdle() so that AeDRMExecute() will return as soon as it finds no pending tasks to carry out, then, to avoid tight loops that may occur when AeDRMExecute returns immediately, you need to insert brief delays between calls to AeDRMExecute().*

Include file: AeOS.h

Asset Status Functions

For background information, refer to *"Asset Status Control"* on page 2-13.

AeDRMSetDeviceStatus()

`AeError AeDRMSetDeviceStatus(AeInt32 iDeviceId, AeBool bEnable);`

where:

- `iDeviceId` – identifier of the asset (device) for which processing will be enabled or disabled
- `bEnable` – valid values are `true` (enable) and `false` (disable)

Description: Disables or enables processing for the specified asset at any time. If `bEnable` is set to false, the asset is removed from the configuration of managed assets for the associated Master (or Gateway). This setting is useful when you have a managed asset that is being removed from service permanently and you want to remove it permanently from the associated Master (Gateway).

Include file: AeInterface.h.

AeDRMSetDeviceOnline()

```
AeError AeDRMSetDeviceOnline(AeInt32 iDeviceId, AeBool bOnline)
```

where:

- `iDeviceId` – identifier of the asset whose status you want to set to online or offline
- `bOnline` – valid values are `true` (asset is online) and `false` (asset is offline)

Description: Explicitly sets the status of a managed asset to online or offline. The status reflects the status of the connection between the managed asset and its managing Master (Gateway). This function only changes the attribute that indicates whether the asset is online. Otherwise, processing of data is not affected by this function. On the Axeda Enterprise Server side, a change in the status of a managed asset could be used as a trigger that changes the condition shown for the asset in the Asset Dashboard. Refer to the online help about Asset Conditions for details.

Include file: AeInterface.h.

Raw HTTP Request Execution Functions

For background information, refer to *"Raw HTTP Request Execution"* on page 2-13.

AeWebSyncExecute()

```
AeError AeWebSyncExecute(AeWebRequest *pRequest);
```

where:

- `pRequest` – HTTP request object; defined in `AeWebRequest` in *AeWebRequest.h*,

Description: Synchronously executes a raw HTTP request.

Include file: AeInterface.h.

AeWebAsyncExecute()

```
AeError AeWebAsyncExecute(AeHandle *ppHandle, AeWebRequest *pRequest,
AeTimeValue *pTimeLimit, AeBool *pbComplete);
```

where:

- `ppHandle` – must be NULL on first function call;
- `pRequest` – HTTP request object; defined in `AeWebRequest` in *AeWebRequest.h*
- `pTimeLimit` – number of milliseconds by which time this function call must return. If NULL, this function call is not limited by time; if non-NULL, the function returns when either the time period elapses or the request completes, whichever comes first.
- `pbComplete` – If 1 (true), request is complete; if 0 (false), request is not yet complete.

Description: Asynchronously executes raw HTTP request. The asset application should call this function in a loop until the request is complete (Boolean, pointed by `pbComplete` set to `true`). When the function is called for the first time for a request, the handle pointed by `ppHandle` must be NULL. *AeWebAsyncExecute()* fills out the handle when it returns. To complete the request, subsequent calls should pass this returned handle.

Include file: AeInterface.h.

XML/SOAP Parser Component Functions

For an overview of the XML/SOAP Parser component of Agent Embedded, refer to *"XML/SOAP Parser Component"* on page 2-7. Use the DRMSOAP functions in the *AeDRMSOAP.h* file to create and destroy SOAP communication threads, initiate the DOM parser to parse SOAP commands received, and to make the values in the parsed SOAP commands available to the calling asset application.

AeDRMSOAPNew()

```
AeDRMSOAP *AeDRMSOAPNew(void);
```

Description: Creates a SOAP message object, which may be used for parsing. Returns a pointer to the new object.

Include file: AeDRMSOAP.h

AeDRMSOAPDestroy()

```
void AeDRMSOAPDestroy(AeDRMSOAP *pSOAP);
```

where:

- `pSOAP` – SOAP message object to destroy

Description: Destroys a SOAP message object.

Include file: AeDRMSOAP.h

AeDRMSOAPParse()

```
AeBool AeDRMSOAPParse(AeDRMSOAP *pSOAP, AeChar *pSource, AeInt iLength);
```

where:

- `pSOAP` – SOAP object to which parsed results are saved
- `pSource` – SOAP message to be parsed
- `iLength` - size of SOAP message to be parsed

Description: Parses SOAP message. As result of parsing, the SOAP object (*pSOAP*) is modified. Returns true (1) if parsing is successful, or false (0) otherwise.

Include file: AeDRMSOAP.h

AeDRMSOAPGetFirstMethod()

```
AeHandle AeDRMSOAPGetFirstMethod(AeDRMSOAP *pSOAP);
```

where:

- `pSOAP` – SOAP message object

Description: Returns a handle for the first method in the SOAP message (*pSOAP*), or NULL if the message does not contain methods.

Include file: AeDRMSOAP.h

AeDRMSOAPGetNextMethod()

```
AeHandle AeDRMSOAPGetNextMethod(AeHandle pMethod);
```

where:

- `pMethod` – method, after which a handle for the next method is returned

Description: Returns handle for the method following method `pMethod`, or NULL, if there are no more methods.

Include file: AeDRMSOAP.h

AeDRMSOAPGetMethodByName()

```
AeHandle AeDRMSOAPGetMethodByName(AeDRMSOAP *pSOAP, AeChar *pName);
```

where:

- `pSOAP` – SOAP message object
- `pName` – method in SOAP message for which a handle is retrieved

Description: Returns handle for method named *pName* in the SOAP message object (*pSOAP*), or NULL if not found.

Include file: AeDRMSOAP.h

AeDRMSOAPGetMethodName()

```
AeChar *AeDRMSOAPGetMethodName(AeHandle pMethod);
```

where:

- `pMethod` – handle for the method whose name is retrieved

Description: Returns name for method *pMethod*.

Include file: AeDRMSOAP.h

AeDRMSOAPGetFirstParameter()

```
AeHandle AeDRMSOAPGetFirstParameter(AeHandle pMethod);
```

where:

- `pMethod` – handle to the method from which you want to retrieve a handle to the first parameter

Description: Returns a handle for the first parameter in method `pMethod`, or NULL if the method does not contain parameters.

Include file: AeDRMSOAP.h

AeDRMSOAPGetNextParameter()

```
AeHandle AeDRMSOAPGetNextParameter(AeHandle pParameter);
```

where:

- `pParameter` – handle to the parameter that precedes the parameter for which you want to retrieve a handle

Description: Returns a handle to the parameter following parameter `pParameter`. Returns NULL if there are no more parameters.

Include file: AeDRMSOAP.h

AeDRMSOAPGetParameterByName()

```
AeHandle AeDRMSOAPGetParameterByName(AeHandle pMethod, AeChar *pName);
```

where:

- `pMethod` – method containing parameter for which a handle is retrieved
- `pName` – name of the parameter for which a handle is retrieved

Description: Returns a handle for parameter `pName` in method `pMethod`, or NULL if not found.

Include file: AeDRMSOAP.h

AeDRMSOAPGetParameterFirstChild()

```
AeHandle AeDRMSOAPGetParameterFirstChild(AeHandle pParameter);
```

where:

- `pParameter` – parameter for which a handle is retrieved for the first child

Description: Returns a handle for the first child of parameter `pParameter`, or NULL if there are no children.

Include file: AeDRMSOAP.h

AeDRMSOAPGetParameterName()

```
AeChar *AeDRMSOAPGetParameterName(AeHandle pParameter);
```

where:

- `pParameter` – parameter for which the name is returned

Description: Returns the name for parameter `pParameter`.

Include file: AeDRMSOAP.h

AeDRMSOAPGetParameterValue()

```
AeChar *AeDRMSOAPGetParameterValue(AeHandle pParameter);
```

where:

- `pParameter` – parameter for which the value is returned

Description: Returns the value of parameter `pParameter`. The value of `pParameter` is a string consisting of the first character data portion of the parameter content, minus any leading white space.

Include file: AeDRMSOAP.h

AeDRMSOAPGetParameterValueByName()

```
AeChar *AeDRMSOAPGetParameterValueByName(AeHandle pMethod, AeChar *pName);
```

where:

- `pMethod` – handle to the method containing parameter `pName`
- `pName` – parameter whose value is returned

Description: Returns the value of parameter `pName` in method `pMethod`, or NULL if not found.

Include file: AeDRMSOAP.h

AeDRMSOAPGetFirstAttribute()

```
AeHandle AeDRMSOAPGetFirstAttribute(AeHandle pNode);
```

where:

- `pNode` – handle to the node containing an attribute for which a handle is retrieved

Description: Returns a handle for the first attribute of node `pNode`, or NULL if there are no attributes. A node is either a method or a parameter.

Include file: AeDRMSOAP.h

AeDRMSOAPGetNextAttribute()

```
AeHandle AeDRMSOAPGetNextAttribute(AeHandle pAttribute);
```

where:

- `pAttribute` – attribute, for which a handle to the next attribute is retrieved

Description: Returns a handle for the attribute following attribute `pAttribute`, or NULL if there are no more attributes.

Include file: AeDRMSOAP.h

AeDRMSOAPGetAttributeByName()

```
AeHandle AeDRMSOAPGetAttributeByName(AeHandle pNode, AeChar *pName);
```

where:

- `pNode` – node containing attribute `pName`
- `pName` – attribute for which a handle is retrieved

Description: Returns a handle for attribute `pName` in the node `pNode`, or NULL if not found.

Include file: AeDRMSOAP.h

AeDRMSOAPGetAttributeName()

```
AeChar *AeDRMSOAPGetAttributeName(AeHandle pAttribute);
```

where:

- `pAttribute` – attribute for which the name is returned

Description: Returns the name of attribute `pAttribute`.

Include file: AeDRMSOAP.h

AeDRMSOAPGetAttributeValue()

```
AeChar *AeDRMSOAPGetAttributeValue(AeHandle pAttribute);
```

where:

- `pAttribute` – attribute for which the value is returned

Description: Returns the value of attribute `pAttribute`.

Include file: AeDRMSOAP.h

AeDRMSOAPGetAttributeValueByName()

```
AeChar *AeDRMSOAPGetAttributeValueByName(AeHandle pNode, AeChar *pName);
```

where:

- `pNode` – node containing attribute `pName`
- `pName` – attribute for which the value is retrieved

Description: Returns the value of attribute `pName` in node `pNode`, or NULL if not found.

Include file: AeDRMSOAP.h

System-dependent Functions

Use the system-dependent functions to implement calls between the device application and the Axeda Agent Embedded API. The device application calls are specific to your device.

Initialization of Network Environment

AeNetInitialize()

```
AeError AeNetInitialize(void);
```

Description: Initializes the network environment.

Include file: AeOS.h

Network and Socket Connections

AeNetGetSocket()

`AeError AeNetGetSocket(AeSocket *pSock, AeBool bStream);`

where:

- `pSock` – socket object
- `bStream` – if 1 (true) the socket is connection-oriented; if 0 (false) the socket is not connection-oriented

Description: Creates a socket object.

Include file: AeOS.h

AeNetConnect()

`AeError AeNetConnect(AeSocket *pSock, AeNetAddress *pAddress);`

where:

- `pSock` – socket object
- `pAddress` – network address of the remote service

Description: Connects a socket to the remote service specified by the network address in `pAddress`.

Include file: AeOS.h

AeNetDisconnect()

`AeError AeNetDisconnect(AeSocket *pSock);`

where:

- `pSock` – socket object

Description: Disconnects a connected socket.

Include file: AeOS.h

AeNetSend()

```
AeError AeNetSend(AeSocket *pSock, AePointer pData, AeInt32 iLength, AeInt32
*piSent);
```

where:

- pSock – socket object
- pData – buffer
- iLength – number of bytes
- piSent – actual number of bytes sent over socket

Description: Sends a number of bytes (iLength) from the specified buffer (pData) over the specified socket (pSock). The socket must be connected before calling this function. The actual amount of bytes sent is stored in an integer pointed to by piSent.

Include file: AeOS.h

AeNetReceive()

```
AeError AeNetReceive(AeSocket *pSock, AePointer pData, AeInt32 iLength,
AeInt32 *piReceived);
```

where:

- pSock – socket object
- pData – buffer
- iLength – number of bytes
- piReceived – actual number of bytes received from socket

Description: Receives a number of bytes (up to iLength) into the specified buffer (pData) from the specified socket (pSock). The socket must be connected before calling this function. The actual number of bytes received is stored in integer pointed to by piReceived.

Include file: AeOS.h

AeSelect()

```
AeError AeSelect(AeInt iMaxFD, AeFDArray *pReadFDs, AeFDArray  *pWriteFDs,
AeFDArray *pExceptFDs, AeTimeValue *pTimeOut);
```

where:

- `iMaxFD` – the highest numbered descriptor in any of three sets, plus 1
- `pReadFDs` – set of descriptors
- `pWriteFDs` – set of descriptors
- `pExceptFDs` – set of descriptors
- `pTimeOut` - upper bound on the amount of time elapsed before the function returns; if 0, *AeSelect()* returns immediately; if NULL (no timeout), *AeSelect()* can block indefinitely; supports values defined in seconds and microseconds.

Description: Waits for a number of file descriptors to change status. Three independent sets of descriptors are watched: those listed in `pReadFDs` are watched to see if characters become available for reading (receiving); those listed in `pWriteFDs` are watched to see if a write (send) will not block; those in `pExceptFDs` are watched for exceptions.

On return, the sets are modified in place to indicate which descriptors actually changed status.

Include file: AeOS.h

AeNetSetBlocking()

```
AeError AeNetSetBlocking(AeSocket *pSock, AeBool bBlocking);
```

where:

- `pSock` – socket object
- `bBlocking` – if 1 (true), socket uses blocking mode; if 0 (false), socket uses non-blocking mode

Description: Sets the mode for the specified socket as blocking or non-blocking.

Include file: AeOS.h

AeNetSetNoDelay()

```
AeError AeNetSetNoDelay(AeSocket *pSock, AeBool bNoDelay);
```

where:

- `pSock` – socket object
- `bBlocking` – if `1` (true), Nagle's algorithm is enabled; if `0` (false), disables Nagle's algorithm

Description: Enables or disables TCP_NODELAY (Nagle's algorithm) for the specified socket.

Include file: AeOS.h

AeNetSetSendBufferSize()

```
AeError AeNetSetSendBufferSize(AeSocket *pSock, AeInt32 iSize);
```

where:

- `pSock` – socket object
- `iSize` – size (number of bytes)

Description: Sets the maximum size of the send buffer to `iSize` for the specified socket.

Include file: AeOS.h

AeNetGetPendingError()

```
AeError AeNetGetPendingError(AeSocket *pSock);
```

where:

- `pSock` – socket object

Description: Returns a pending error for socket `pSock`.

Include file: AeOS.h

AeNetResolve()

```
AeError AeNetResolve(AeChar *pHostname, AeUInt32 *piAddress);
```

where:

- `pHostname` – host name to resolve
- `piAddress` – points to resulting 32-bit IP address

Description: Resolves the specified host name to a 32-bit IP address (pointed to by `piAddress`) in the network byte order. Agent Embedded supports both IPv4 (*nnn.nnn.nnn.nnn*) and IPv6 address formats.

Include file: AeOS.h

Machine/Host Information

AeNetHostName()

```
AeError AeNetHostName(AeChar *pHostname, AeInt iLength);
```

where:

- `pHostname` – points to a buffer containing the host name of the local machine
- `iLength` – the size of the buffer

Description: Returns the host name of the local machine and stores the result in the buffer pointed to by `pHostname`.

Include file: AeOS.h

AeNetInetAddr()

```
AeUInt32 AeNetInetAddr(AeChar *pHost);
```

where:

- `pHost` – string containing an IP address in either IPv4 (*nnn.nnn.nnn.nnn*) or IPv6 format

Description: Returns a 32-bit IP address in the network byte order based on a string (`pHost`) containing an IP address in either IPv4 (*nnn.nnn.nnn.nnn*) or IPv6 format.

Include file: AeOS.h

Conversions for Network Communication

AeNetHToNL()

```
AeUInt32 AeNetHToNL(AeUInt32 iNumber);
```

where:

- `iNumber` – 32-bit unsigned integer

Description: Converts a 32-bit unsigned integer to the network byte order.

Include file: AeOS.h

AeNetHToNS()

```
AeUInt16 AeNetHToNS(AeUInt16 iNumber);
```

where:

- `iNumber` – 16-bit unsigned integer

Description: Converts a 16-bit unsigned integer to the network byte order.

Include file: AeOS.h

Mutex Support

AeMutexInitialize()

```
void AeMutexInitialize(AeMutex *pMutex);
```

where:

- `pMutex` – mutex object

Description: Initializes a mutex object.

Include file: AeOS.h

AeMutexDestroy()

```
void AeMutexDestroy(AeMutex *pMutex);
```

where:

- `pMutex` – mutex object

Description: Destroys a mutex object.

Include file: AeOS.h

AeMutexLock()

```
void AeMutexLock(AeMutex *pMutex);
```

where:

- `pMutex` – mutex object

Description: Locks a mutex object.

Include file: AeOS.h

AeMutexUnlock()

```
void AeMutexUnlock(AeMutex *pMutex);
```

where:

- `pMutex` – mutex object

Description: Unlocks a mutex object.

Include file: AeOS.h

Setting Time

AeGetCurrentTime()

```
void AeGetCurrentTime(AeTimeValue *pTime);
```

where:

- `pTime` – current date and time (in microseconds)

Description: Retrieves the current date/time from the system where the application is running (with a microsecond resolution).

Include file: AeOS.h

Logging Support

AeLogOpen()

```
AeError AeLogOpen(void);
```

Description: Opens (initializes) the logging facility.

Include file: AeOS.h

AeLogClose()

```
void AeLogClose(void);
```

Description: Closes (shuts down) the logging facility.

Include file: AeOS.h

AeLog()

```
void AeLog(AeLogRecordType iType, AeChar *pFormat, ...);
```

where:

- `iType` – type of message to log
- `pFormat` – format for the message to log

Description: Logs a message of the specified type.

Include file: AeOS.h

AeLogEx()

```
void AeLogEx(AeLogRecordType iType, AeUInt32 iCategory, AeChar *pFormat, ...);
```

where:

- `iType` – the type of message to log; as defined in *AeOS.h*, the possible types of log messages, from least to most verbose, are `None`, `Error`, `Warning`, `Info`, `Debug`, and `Trace`.
- `iCategory` – the category of message to log; as defined in *AeOS.h*, the possible category values are `NONE, NETWORK, SERVER_STATUS, DATA_QUEUE, REMOTE_SESSION, FILE_UPLOAD`, and `FILE_DOWNLOAD`. These categories can be `OR`'d together.
- `pFormat` – format to use for the message

Description: Default log function. Logs a message of the specified type and category.

Include file: AeOS.h

File Manipulation

For an overview of the File Transfer component of Agent Embedded, refer to *"File Transfer Component (optional)"* on page 2-9.

AeFileOpen()

```
AeFileHandle AeFileOpen(AeChar* name, AeUInt32 openFlags);
```

where:

- `name` - the file to open
- `openFlags` - the mode for opening the file; modes include `Read Only`, `Write Only`, `Read/Write`, `Create`, and `Truncate`, as defined under "`\* file operations support */`" in *AeOS.h*

Description: Opens the named file in the specified mode and returns a handle to the file or, if the file is not found, NULL.

Include file: AeOS.h

AeFileSeek()

```
AeInt32 AeFileOpen(AeFileHandle file, AeInt32 iOffset, AeInt32 iWhere);
```

where:

- `file` - a handle to the file to open
- `iOffset` - the number of bytes from `iWhence`. If `iWhence` is `AE_SEEK_SET`, then `iOffset` is the absolute position in the file. If `AE_SEEK_CUR`, then `iOffset` is a position relative to the current location of the file pointer. If `AE_SEEK_END`, then `iOffset` is a position relative to the end of the file, so `iOffset` must be a negative value.
- `iWhence` - initial position of the file pointer. Possible values are `AE_SEEK_SET`, `AE_SEEK_CUR`, and `AE_SEEK_END`.

Description: Sets the file position.

Include file: AeOS.h

AeFileRead()

```
AeInt32 AeFileRead(AeFileHandle file, void* buf, AeInt32 iSize);
```

where:

- `file` - a handle to the file to read
- `buf` - the buffer used to store the content while reading
- `iSize` - the requested amount of bytes

Description: Reads the content of the specified file and returns the number of bytes read.

Include file: AeOS.h

AeFileWrite()

```
AeInt32 AeFileWrite(AeFileHandle file, void* buf, AeInt32 iSize);
```

where:

- `file` - a handle to the file to write to
- `buf` - the buffer from which the content is taken for writing to the file
- `iSize` - the requested amount of bytes

Description: Writes content to the specified file and returns the number of bytes written.

Include file: AeOS.h

AeFileClose()

```
AeInt32 AeFileClose(AeFileHandle file);
```

where:

- `file` - a handle to the file to close

Description: Closes the specified file. Returns 0 in case of success and -1 in case of failure.

Include file: AeOS.h

AeFileDelete()

```
AeBool AeFileDelete(AeChar name);
```

where:

- `name` - the name of the file to delete

Description: Removes the specified file from the local machine and returns a Boolean to indicate the success (1, true) or failure (0, false) of the operation.

Include file: AeOS.h

AeFileExist()

```
AeBool AeFileExist(AeChar name);
```

where:

- `name` - the name of the file to find (to see if it exists)

Description: Checks for the existence of the specified file and returns a Boolean to indicate the success (1, true) or failure (0, false) of the operation.

Include file: AeOS.h

AeFileGetSize()and AeFileGetSize64()

```
#ifndef ENABLE_LARGEFILE64
AeInt32 AeFileGetSize(AeChar *pName);
#else
AeInt64 AeFileGetSize64(AeChar *pName);
#endif
```

where:

- `pName` - pointer to the name of the file whose size you want to determine

Description: When `ENABLE_LARGEFILE64` is defined, Agent Embedded automatically uses the *AeFileGetSize64* version of this function; otherwise, it uses *AeFileGetSize*. Returns the size of the specified file.

Include file: AeOS.h

AeMakeDir()

```
AeBool AeMakeDir(AeChar *pName);
```

where:

- `Name` - a pointer to the name of the directory you want to create

Description: Returns a Boolean to indicate the success (1, true) or failure (0, false) of the operation. If the directory already exists, the operation fails.

Include file: AeOS.h

HTTP Communications Functions

For an overview of the Web Client component of Agent Embedded, refer to *"Web Client Component"* on page 2-7. Use the HTTP communications functions to support and configure HTTP communications between the Agent Embedded and the Axeda Enterprise Server.

AeWebRequestNew()

```
AeWebRequest *AeWebRequestNew(void);
```

Description: Creates a new HTTP request object.

Include file: AeWebRequest.h

AeWebRequestDestroy()

```
void AeWebRequestDestroy(AeWebRequest *pRequest);
```

where:

- `pRequest` – HTTP request object; defined in `AeWebRequest` in *AeWebRequest.h*

Description: Destroys an HTTP request object.

Include file: AeWebRequest.h

AeWebRequestSetURL()

```
AeBool AeWebRequestSetURL(AeWebRequest *pRequest, AeChar *pString);
```

where:

- `pRequest` – HTTP request object; defined in `AeWebRequest` in *AeWebRequest.h*
- `pString` – string to parse to URL

Description: Parses the specified string into a URL and modifies the corresponding properties of the request (secure, username, password, host name, port, absolute path).

Include file: AeWebRequest.h

AeWebRequestGetURL()

```
AeChar *AeWebRequestGetURL(AeWebRequest *pRequest);
```

where:

- `pRequest` – pointer to the HTTP request object created in **AeWebRequest()*

Description: Composes a URL string, based on the corresponding properties of the request (secure, username, password, host name, port, absolute path).

Include file: AeWebRequest.h

AeWebRequestSetVersion()

```
void AeWebRequestSetVersion(AeWebRequest *pRequest, AeChar *pVersion);
```

where:

- `pRequest` – HTTP request object; defined in `AeWebRequest` in *AeWebRequest.h*
- `pVersion` – version to set for this HTTP request, valid values are “1.0” or “1.1;” the default version is 1.1.

Description: Sets the HTTP version.

Include file: AeWebRequest.h

AeWebRequestSetMethod()

```
void AeWebRequestSetMethod(AeWebRequest *pRequest, AeChar *pMethod);
```

where:

- `pRequest` – HTTP request object; defined in `AeWebRequest` in *AeWebRequest.h*
- `pMethod` – method to set for this HTTP request

Description: Sets the HTTP request method.

Include file: AeWebRequest.h

AeWebRequestSetHost()

```
void AeWebRequestSetHost(AeWebRequest *pRequest, AeChar *pHost);
```

where:

- `pRequest` – HTTP request object; defined in `AeWebRequest` in *AeWebRequest.h*
- `pHost` – host name or IP address

Description: Sets the host name or IP address of the server from which the HTTP request originated. Agent Embedded supports both IPv4 (*nnn.nnn.nnn.nnn*) and IPv6 address formats.

Include file: AeWebRequest.h

AeWebRequestSetPort()

```
#define   AeWebRequestSetPort(r, x) ((r)->iPort = (x))
```

where:

- `r` – HTTP request object
- `x` – port number

Description: Sets the port number of the server from which the HTTP request originated.

Include file: AeWebRequest.h

AeWebRequestSetAbsPath()

```
void AeWebRequestSetAbsPath(AeWebRequest *pRequest, AeChar *pAbsPath);
```

where:

- `pRequest` – HTTP request object; defined in `AeWebRequest` in *AeWebRequest.h*
- `pAbsPath` – Absolute path of server

Description: Sets the absolute path at the server from which the HTTP request originated.

Include file: AeWebRequest.h

AeWebRequestSetUser()

```
void AeWebRequestSetUser(AeWebRequest *pRequest, AeChar *pUser);
```

where:

- `pRequest` – HTTP request object; defined in `AeWebRequest` in *AeWebRequest.h*
- `pUser` – the user name to present (with a password) for authentication

Description: Sets the user name for authentication at the server where the request originated.

Include file: AeWebRequest.h

AeWebRequestSetPassword()

```
void AeWebRequestSetPassword(AeWebRequest *pRequest, AeChar *pPassword);
```

where:

- `pRequest` – HTTP request object; defined in `AeWebRequest` in *AeWebRequest.h*
- `pPassword` – the password associated with the user name set in *AeWebRequestSetUser()*

Description: Sets the password for authentication at the server where the request originated.

Include file: AeWebRequest.h

AeWebRequestSetContentType()

```
void AeWebRequestSetContentType(AeWebRequest *pRequest, AeChar
*pContentType);
```

where:

- `pRequest` – HTTP request object; defined in `AeWebRequest` in *AeWebRequest.h*
- `pContentType` – MIME content type

Description: Sets the MIME content type of the request body.

Include file: AeWebRequest.h

AeWebRequestSetEntityData()

```
#define   AeWebRequestSetEntityData(r, x) ((r)->pEntityData = (x))
```

where:

- `r` – HTTP request object
- `x` – pointer to buffer

Description: Sets a pointer to a buffer for the request body.

Include file: AeWebRequest.h

AeWebRequestSetEntitySize()

```
#define   AeWebRequestSetEntitySize(r, x) ((r)->iEntitySize = (x))
```

where:

- `r` – HTTP request object
- `x` – Size

Description: Sets the size of the request body.

Include file: AeWebRequest.h

AeWebRequestSetPersistent()

```
#define   AeWebRequestSetPersistent(r, x) ((r)->bPersistent = (x))
```

where:

- `r` – HTTP request object
- `x` – value to enable/disable connection

Description: Enables/disables persistent connection.

Include file: AeWebRequest.h

AeWebRequestSetStrict()

```
#define   AeWebRequestSetStrict(r, x) ((r)->bStrict = (x))
```

where:

- `r` – HTTP request object
- `x` – value to enable/disable strict mode

Description: Enables/disables strict mode (in strict mode, the request is processed in strict accordance with the specified HTTP protocol version).

Include file: AeWebRequest.h

AeWebRequestSetSecure()

```
#define   AeWebRequestSetSecure(r, x) ((r)->bSecure = (x))
```

where:

- `r` – HTTP request object
- `x` – value to enable/disable SSL

Description: Enables/disables SSL encryption for the request.

Include file: AeWebRequest.h

AeWebRequestSetTimeOut()

```
#define   AeWebRequestSetTimeOut(r, x) ((r)->timeOut = *(x))
```

where:

- `r` – HTTP request object
- `x` – timeout

Description: Sets a timeout for communications triggered by the request.

Include file: AeWebRequest.h

AeWebRequestSetUserData()

```
#define   AeWebRequestSetUserData(r, x) ((r)->pUserData = (x))
```

where:

- `r` – HTTP request object
- `x` – user data

Description: Sets user data. This may be used in the callbacks.

Include file: AeWebRequest.h

AeWebRequestSetRequestHeader()

```
void AeWebRequestSetRequestHeader(AeWebRequest *pRequest, AeChar *pName,
AeChar *pValue);
```

where:

- `pRequest` – HTTP request object; defined in `AeWebRequest` in *AeWebRequest.h*
- `pName` – name portion of name-value pair
- `pValue` – value portion of name-value pair

Description: Adds custom request header (name - value pair).

Include file: AeWebRequest.h

AeWebRequestSetResponseHeader()

```
void AeWebRequestSetResponseHeader(AeWebRequest *pRequest, AeChar *pName,
AeChar *pValue);
```

where:

- `pRequest` – HTTP request object; defined in `AeWebRequest` in *AeWebRequest.*
- `pName` – name portion of name-value pair
- `pValue` – value portion of name-value pair

Description: Adds custom response header (name - value pair).

Include file: AeWebRequest.h

AeWebRequestGetResponseHeader()

```
AeChar *AeWebRequestGetResponseHeader(AeWebRequest *pRequest, AeChar *pName);
```

where:

- `pRequest` – HTTP request object; defined in `AeWebRequest` in *AeWebRequest.*
- `pName` – name portion of name-value pair

Description: Retrieves a custom response header by name.

Include file: AeWebRequest.h

AeWebRequestGetFirstResponseHeader()

```
AePointer AeWebRequestGetFirstResponseHeader(AeWebRequest *pRequest, AeChar
**ppName, AeChar **ppValue);
```

where:

- `pRequest` – HTTP request object; defined in `AeWebRequest` in *AeWebRequest.*
- `ppName` – name portion of name-value pair
- `ppValue` – value portion of name-value pair

Description: Retrieves the first response header (name - value pair). This function also returns a pointer, which may be used in calls to *AeWebRequestGetNextResponseHeader()*.

Include file: AeWebRequest.h

AeWebRequestGetNextResponseHeader()

```
AePointer AeWebRequestGetNextResponseHeader(AeWebRequest *pRequest, AePointer
pPosition, AeChar **ppName, AeChar **ppValue);
```

where:

- `pRequest` – HTTP request object; defined in `AeWebRequest` in *AeWebRequest.*
- `pPosition` – pointer returned by previous calls to *AeWebRequestGetFirstResponseHeader()* or *AeWebRequestGetNextResponseHeader()*
- `ppName` – name portion of name-value pair
- `ppValue` – value portion of name-value pair

Description: Retrieves the next response header (name - value pair). This function also returns a pointer, which may be used in the subsequent calls.

Include file: AeWebRequest.h

AeWebRequestSetOnError()

```
#define   AeWebRequestSetOnError(r, x) ((r)->pOnError = (x))
```

where:

- `r` – HTTP request object
- `x` – a pointer to the callback function to be registered. The format that you need to use for this callback function is described in the section below, *OnWebRequestErrorCallback Function*.

Description: Registers the callback invoked on an asynchronous communication error.

Include file: AeWebRequest.h

OnWebRequestErrorCallback Function

Use this application-defined callback function with the *AeWebRequestSetOnError()* function, in which *x* defines a pointer to this callback function.

OnWebRequestErrorCallback is a placeholder for the actual name you decide to use in your application. Whatever you decide to name it, the callback function must have the following signature:

```
void (*)(AeWebRequest *pRequest, AeError iError);
```

where:

- `pRequest` – pointer to the HTTP request object; `AeWebRequest` is defined in *AeWebRequest.h*
- `iError` indicates the error code.

AeWebRequestSetOnResponse()

```
#define   AeWebRequestSetOnResponse(r, x) ((r)->pOnResponse = (x))
```

where:

- `r` – HTTP request object
- `x` – a pointer to the callback function to be registered. The format that you need to use for this callback is described in the section below, *OnWebRequestResponseCallback Function*.

Description: Registers the callback function invoked on receipt of the response header.

Include file: AeWebRequest.h

OnWebRequestResponseCallback Function

Use this application-defined callback function with the *AeWebRequestSetOn Response()* function, in which *x* defines a pointer to this callback function.

OnWebRequestResponseCallback is a placeholder for the actual name you decide to use in your application. Whatever you decide to name it, the callback function must have the following signature:

```
AeBool (*)(AeWebRequest *pRequest, AeInt iStatusCode);
```

where:

- `pRequest` – pointer to the HTTP request object; `AeWebRequest` is defined in *AeWebRequest.h*
- `iStatusCode` indicates numeric HTTP status.

The callback function should return `false` if it is required to abort the request. Otherwise, it should return `true`.

AeWebRequestSetOnEntity()

```
#define   AeWebRequestSetOnEntity(r, x) ((r)->pOnEntity = (x))
```

where:

- `r` – HTTP request object
- `x` – a pointer to the callback function to be registered. The format that you need to use for this callback is described in the section below, *OnWebRequestEntityCallback Function*.

Description: Registers the callback invoked on receipt of a chunk of the response body.

Include file: AeWebRequest.h

OnWebRequestEntityCallback Function

Use this application-defined callback function with the *AeWebRequestSetOnEntity()* function, in which *x* defines a pointer to this callback function.

OnWebRequestEntityCallback is a placeholder for the actual name you decide to use in your application. Whatever you decide to name it, the callback function must have the following signature:

```
AeBool (*)(AeWebRequest *pRequest, AeInt32 iDataOffset, AeChar
*pData, AeInt32 iSize);
```

where:

- `pRequest` – pointer to the HTTP request object; `AeWebRequest` is defined in *AeWebRequest.h*
- `pData` – pointer to the chunk buffer
- `iSize` – size of the chunk
- `iDataOffset` – offset of the chunk from the beginning of the response body

This callback function should return `false` if it is required to abort the request. Otherwise, it should return `true`.

AeWebRequestSetOnCompleted()

```
#define   AeWebRequestSetOnCompleted(r, x) ((r)->pOnCompleted = (x))
```

where:

- `r` – HTTP request object
- `x` – a pointer to the callback function to be registered. The format that you need to use for this callback is described in the section below, *OnWebRequestCompletedCallback Function*.

Description: Registers the callback invoked on successful completion of the request.

Include file: AeWebRequest.h

OnWebRequestCompletedCallback Function

Use this application-defined callback function with the *AeWebRequestSetOnCompleted()* function, in which *x* defines a pointer to this callback function.

OnWebRequestCompletedCallback is a placeholder for the actual name you decide to use in your application. Whatever you decide to name it, the callback function must have the following signature:

```
void (*)(AeWebRequest *pRequest);
```

where:

- `pRequest` – pointer to the HTTP request object; `AeWebRequest` is defined in *AeWebRequest.h*

AeWebRequestClearStatus()

```
void AeWebRequestClearStatus(AeWebRequest *pRequest);
```

where:

- `pRequest` – HTTP request object; defined in `AeWebRequest` in *AeWebRequest.h*

Description: Resets the request to the initial state.

Include file: AeWebRequest.h

Index