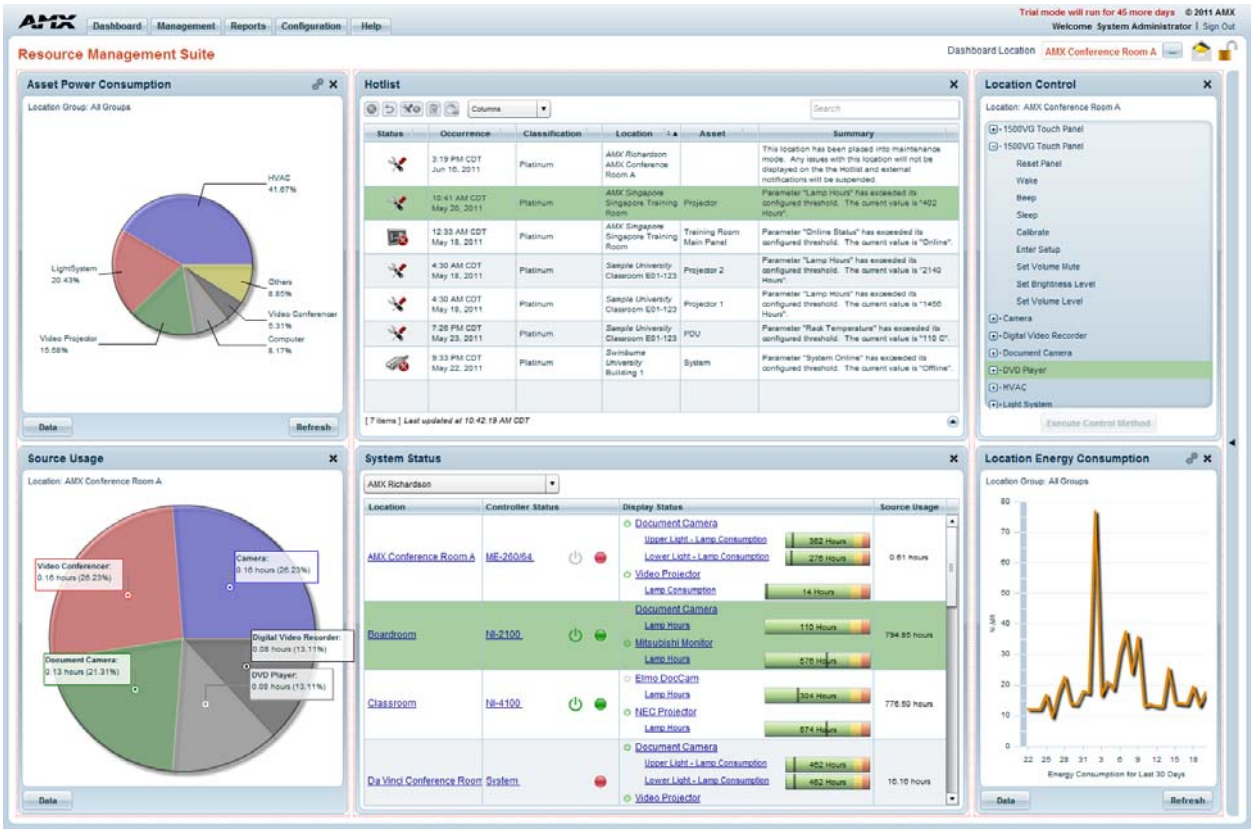
NETLINX PROGRAMMER'S MANUAL

RMS-ENT (SDK V4.3)

RMS ENTERPRISE



AV FOR AN IT WORLD

Table of Contents

RMS Enterprise NetLinx Programmer's Guide (SDK v4.3)

Overview

This document outlines the RMS SDK (v4.3) programming concepts and RMS NetLinx API commands for RMS Enterprise, and describes all channels, levels, and Send Commands available via the RMS NetLinx Virtual Device API.

RMS SDK v4.3 is intended for use with (legacy) NI NetLinx Masters.

System Requirements

The RMS SDK is a set of NetLinx and TPDesign4 files that are included in your control system programs. To utilize this SDK, you will need the following applications installed:

- NetLinx Studio 4.3 (or greater)
- TPDesign4 v3.1 (or greater) for G4 panels
- TPDesign5 for G5 Panels

NOTE: *RMS SDK v4 requires NetLinx Master version 4 firmware; connection instability and Master lockups may result if using version 3 NetLinx Firmware with SDK v4.*

Master & Server Requirements

- NI-Masters with firmware version 4.x (or greater) with at least 32MB allocated to Duet Memory.

  SDK v4.3 was developed primarily for NI Masters and is maintained for legacy (NI-based) systems.

  Always use the latest released version of the v4.3 SDK with legacy NI masters.

  NOTE: *While SDK v4.3 is supported by some versions of NX Master firmware, it is strongly recommended that NX Masters use SDK v4.6. SDK v4.3 is a 'bridge' version (that will run on both NI and NX masters). SDK v4.6 was developed primarily for use with NX Masters and takes advantage of features that are specific to NX Masters. Refer to the RMS Enterprise NetLinx Programmer's Guide (SDK v4.6) for details.*

- RMS Enterprise Server version equal to or greater than the SDK version.

  For example, RMS SDK 4.3 is compatible with RMS Enterprise Server versions 4.3, 4.4, 4.5 and 4.6. However, it is not compatible with RMS Enterprise Server versions 4.1 or 4.2.

Upgrading From RMS v3.3 to RMS Enterprise

RMS Enterprise (SDK4) supports SDK3 legacy client connections without question and without making any changes to the code. In fact, many features will be available without any code changes (such as multi-stage and time delayed notifications and power use based on device type). However, several advanced SDK features (including monitored power through PDU, Duet device native support, system modes, and web configured RMS clients) utilize SDK4 changes that will require registration of the room as new, with a corresponding loss of all history.

- If a user migrates from RMS v3.3 to RMS Enterprise and changes the code from 3.3 SDK to 4.0 SDK they will have all their locations with 3.3 Client Gateways assigned but now OFFLINE and a completely new set of unassigned 4.0 client gateways.
- If you intend to upgrade your system (Server and SDK) completely from RMS 3.3 to RMS Enterprise, it is necessary to delete the 3.3 Client Gateway from the location and assign the corresponding 4.x client gateway.

  Once the 3.3 Client Gateway has been deleted, there will be a loss of all of the history associated with the 3.3 masters and assets.

Terms and Concepts

RMS Enterprise introduces several new concepts and new terminology intended to more accurately convey their roles in the RMS system.

Location Groups

Location groups are logical containers used to organize your physical environment into hierarchical structures in RMS. Location groups may represent buildings, floors, cities, countries, etc. Location groups can contain Locations and/or child location groups (FIG. 1).

Example:

- My Company (*Location Group*)
  - North America (*Location Group*)
    - Unites States (*Location Group*)
      - Colorado (*Location Group*)
      - Nevada (*Location Group*)
      - New York (*Location Group*)
      - Washington (*Location Group*)
      - Texas (*Location Group*)
        - Dallas (*Location Group*)
          - AMX Headquarters (*Location Group*)
            - Einstein (*Location*)
            - Churchill (*Location*)

FIG. 1 Example of Location Groups

Locations

Locations (formerly known as 'Rooms') are logical containers that typically represents a Room or Home, or Equipment Closet where physical devices are present.

A Location must be contained in a Location Group and a location can only contain Assets (devices).

Client Gateway

Client Gateway is a new term in RMS Enterprise that represents the device that is running an instance of the RMS SDK and communicating directly with the RMS server.

In RMS 3.x this is analogous to the "System" entity; however the "System" entity was not only the connection point but also the physical control system master. In RMS Enterprise, the Client Gateway is a more abstract concept, it is the communication bridge between assets (devices) and the RMS system and in most cases it resides on the NetLinx control system platform; however in RMS Enterprise there is a separate asset record that is registered that is intended to represent the physical control system master. The Client Gateway is a communication portal for assets. When first connected to a RMS server, the Client Gateway must be approved by a RMS administrator and assigned to a default location. All assets registered through this client gateway will show up in this default assigned location. Multiple client gateways can be assigned to a single location if needed.

Assets

Assets (formerly known as 'Devices') represent physical devices or equipment. An asset can be a sophisticated electronically monitored device such as a Video Projector or something as simple as a Flip Chart.

Assets can be electronically registered from the RMS SDK on a control system platform or in the case of non-monitored equipment, hand entered via the RMS web based user interface. Assets that are electronically registered via the RMS SDK (and Client Gateway) are automatically assigned to the default location assigned to the client gateway.

However in RMS Enterprise, unlike RMS 3.x, these assets can be moved to other locations in the RMS system simply by dragging the asset record and dropping it into an alternate target location in the RMS web user interface.

Asset Types

Asset Types are a new categorization attribute associated to each registered asset record. Asset types can help group similar assets with like behaviors and purpose together for reporting and searching.

Asset Parameters

Asset Parameters are the attributes or relevant pieces of information about an asset that you want to monitor changes on. For a video projector this may be the power status, the selected video input, the lamp time used, and the unit's operating temperature.

Asset parameters in RMS can be configured to track changes over time and provide historical reporting data. Asset parameters can have monitoring thresholds defined to send alerts or take some action (more on this in the next topic). Asset parameters warehouse information about the device that is expected to change over time and that may need to be tracked and monitored.

Asset Parameter Types

Assets Parameter Types are a new optional categorization attribute associated to each registered asset parameter record. Asset parameters that define a specific asset parameter type are defining that the parameter data provided should participate in some special behaviors and usage in the RMS system. These are primarily used to classify a given parameter with a reporting content type.

Examples of parameter types are "asset.power.consumption" and "asset.transport.runtime". Parameters with these parameter types will have special meaning to RMS and these parameter values will be included in certain reports and other RMS user interface elements.

Asset Parameters Thresholds

Asset Parameter Thresholds are monitoring triggers that can be defined either via the SDK or by the user via the RMS web user interface. For example if the operating temperature of a device were to exceed a manufactured upper limit, the threshold could be tripped and subsequently alert notifications could be sent out to notify the appropriate personnel.

Additionally, control actions can be defined in the RMS web user interface and associated with these thresholds to invoke some action to a tripped parameter event. RMS 3.x imposed a limitation of one parameter threshold per asset parameter. RMS Enterprise allows multiple threshold for each parameter. This enables the ability to perform escalating staged alerts and/or address conditions that may require an upper and lower bounds alert condition.

Asset Metadata Properties

Asset Metadata Properties are a new feature in RMS Enterprise that allow both user defined data and/or programmer defined information about a device to be registered and associated with device records. Typically this type of information is purely informational and relatively static. Asset metadata property changes are not track or monitored. No thresholds can be created for asset metadata properties.

An example of usage may be if a user wants to store a company asset number on each device, or perhaps wants to record a last service data or service contract information. Asset metadata properties also support a Hyperlink data type so user can create links for things like device configuration web pages, device online manuals, etc.

From the SDK perspective, asset metadata properties can be registered along with each asset to store additional informative attributes about the device. For example, if you are programmatically able to interrogate the device and find things like a MAC address or configuration information these can be registered as asset metadata properties and will subsequently be visible and searchable in the RMS web user interface.

Asset Control Methods

Asset Control Methods are also a new convention added in RMS Enterprise. Asset control methods are intended to replace the previous i!-ConnectLinx methodology for externally controlling devices on the RMS system. The previous generation of RMS utilized i!-ConnectLinx because it was an existing convention and framework already established for externally controlling devices on a NetLinx platform. However, the i!-ConnectLinx actions registered in a location were in no way related or associated to each specific asset (device) that was being monitored. This disconnect is what prompted the advent of "control methods" in RMS Enterprise.

Control methods are named functions with an optional list of arguments that can be registered along with each monitored asset. These registered control methods are accessible and can be invoked via the RMS web user interface. RMS Control Macros also utilize these registered asset control methods to perform coordinated sequences of control events across multiple assets and/or locations.

System Power

System Power provides a means to perform a system power ON or OFF on the control system platform. System power is an abstract concept that was automatically registered in the RMS 3.x SDK. System power was composed of a "System Power" parameter for each "System" and an i!-ConnectLinx function to turn the system power ON or OFF.

RMS Enterprise does not make any assumption and register a system power parameter on the control system asset or the asset control methods for controlling system power by default. The control system monitoring module can however, be compiled to include this parameter and control methods if desired.

The RMS Enterprise SDK does provide the system power mechanism and communication infrastructure to route the notifications, but does not provide any concrete implementation behavior for what System Power ON or System Power OFF needs to perform. It is the responsibility of the NetLinx programmer to provide this implementation where needed/desired.

See the *Implementing System Power* section on page 113 for more information.

System Modes

System Mode is also an abstract concept introduced in RMS Enterprise that can facilitate the needs where a system can be changed to a concrete set of operational modes such as "Audio Conferencing", "Video Conferencing", "Presentation", etc.

Like System Power, RMS Enterprise does not make any assumption and register the system mode asset control methods system modes by default. The control system monitoring module can however, be compiled to include this support for control modes if desired.

The RMS Enterprise SDK does provide the system mode mechanism and communication infrastructure to route the notifications, but does not provide any concrete implementation behavior for what System Modes needs to perform. It is the responsibility of the NetLinx programmer to provide this implementation where needed/desired.

See the *Implementing System Modes* section on page 113 for more information.

Source Usage

Source Usage is a tracking mechanism in RMS that can track the time that each source based device is in use. Source devices are typically devices that provide source content for display such as rack computers, laptop inputs, DVD players, document cameras, etc. The RMS 3.x implementation of source usage tracking was highly tied to the i!-ConnectLinx infrastructure and channel based methodology. RMS Enterprise takes a different approach to source usage tracking. Not wanting to make any assumption about how source devices are tracked, the RMS Enterprise SDK requires the NetLinx programmer to be responsible for activating (and in some cases deactivating) source devices in use.

The RMS Enterprise SDK provides more capabilities such as support for both mutually exclusive and non-mutually exclusive sets of source devices as well as multiple source groups. The RMS Enterprise SDK provides a rich and more powerful set of APIs to control the source usage behavior but does require the NetLinx programmer to implement the logic in their program.

See the *Tracking Source Usage* section on page 112 for more information.

Asset Power Monitoring / Energy Management

RMS Enterprise provides power rate (Watts) monitoring for all assets. By default if no specific asset power parameter is provided, then the RMS server will provide estimated power consumption rates based on the assets power status and the asset type associated to the asset.

If a special asset power consumption parameter is registered by the NetLinx programmer, then the server will discontinue any assumed estimation behavior and the responsibility to update the power consumption rate will be that of the NetLinx program.

A special exception to this is AMX power monitoring equipment such as the NXA-PDU-1508-08 (Power Distribution Unit). A NetLinx module is provided that will take ownership of providing power consumption rate information on behalf of other devices that are plugged into the PDU's physical outlets. A device to outlet port mapping is required in the NetLinx program to make these associations.

Please see the *Asset Power / Energy Consumption* section on page 112 for more information on implementing the PDU.

The RMS Enterprise SDK (v4)

Overview

The RMS Enterprise *Software Development Kit* (SDK) provides the necessary runtime framework, programming APIs, convenience API wrappers, device monitoring and management implementations and sample projects to fully integrate a NetLinx control system with RMS.

NOTE: *RMS Enterprise uses SDK version 4. RMS SDK v4 requires NetLinx Master version 4 firmware; connection instability and Master lockups may result if using version 3 NetLinx Firmware with SDK v4.*

The RMS Enterprise SDK consists of the following components:

RMS Enterprise SDK Files

NetLinx Studio Workspace (APW) Files and Support Files

The RMS Enterprise SDK includes two NetLinx Studio Workspace (*.APW) Files:

- **RMS SDK - Duet.apw**: This is a Duet Monitoring Module Workspace (utilizing *Duet* device monitor files - see the *RMS SDK - Duet.apw (Duet Workspace)* section on page 20 for details).
- **RMS SDK - NetLinx.apw:** This is a NetLinx Monitoring Module Workspace (utilizing *NetLinx* device monitor files - see the *RMS SDK - NetLinx.apw (Duet Workspace)* section on page 23 for details).

NetLinx Studio *Workspace* files, as well as the *Source*, *Include* and *Module* files that comprise the workspace can be viewed and edited via the NetLinx Studio application. Refer to NetLinx Studio online help for additional information on working with Workspace files.

By default, both Workspace files are located in the directory:

Program Files > AMX > Resource Management Suite > sdk > netlinx > 4.1.<x>

NOTE: See the *Default File Locations of NetLinx Studio Workspace and Support Files* section on page 25 for a listing of the default locations for the support files described in the following sections.

RMS SDK - Duet.apw (Duet Workspace)

FIG. 2 shows the NetLinx Studio Workspace Bar with the RMS SDK - Duet.apw (Duet Workspace) open:

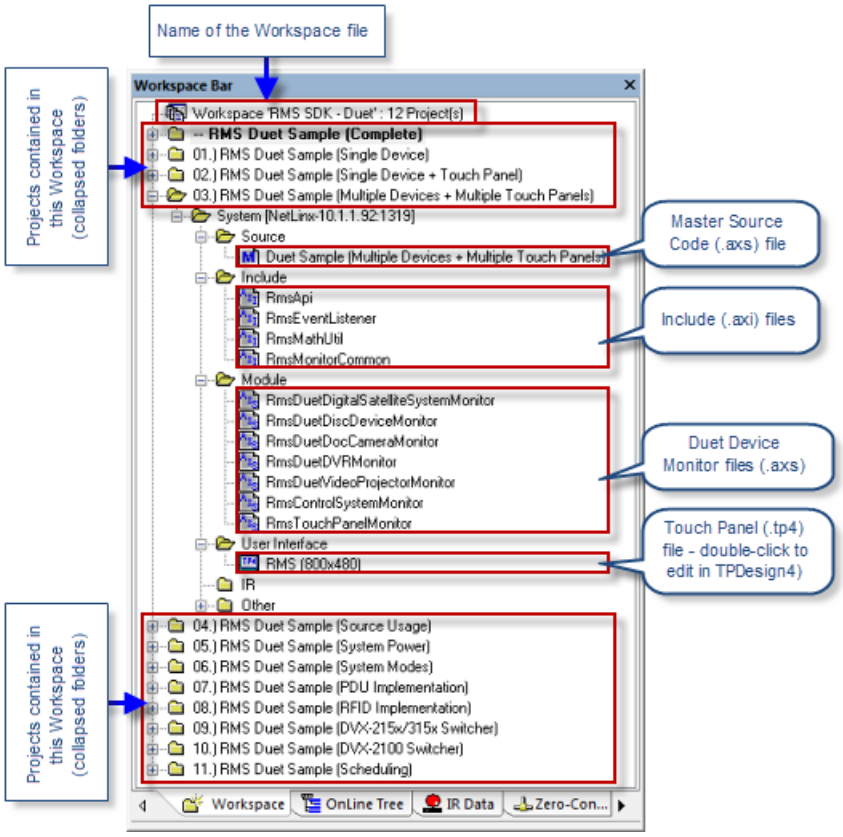


FIG. 2 NetLinx Studio - Workspace Bar (RMS SDK - Duet.apw)

As indicated in the Workspace Bar, the Duet Workspace provided with the RMS Enterprise SDK contains a total of 12 Duet-based projects, each of which presents a starting point for a specific type of NetLinx system.

Each Project folder contains a set of sub-folders that categorize the various Source, Include, Module and Touch Panel files that are included with each project:

Files Included in the Duet Workspace (RMS SDK - Duet.apw)	
File Name	**Description**
Source folder	
<Project Name>.axs.	This is the master Source Code file for the Project/System. The name applied to the Master Source Code file matches the name of it's parent Project.
Include folder	
RmsApi.axi	This include file is provided to implement many of the RMS API channels, levels, and command strings as constants and also to provide many helper functions that can simplify NetLinx integration with the RMS client.
RmsEventListener.axi	This include file is used to listen to the RMS client virtual device events and invoke callback methods when a RMS event occurs. Each event callback method can be exposed to the consuming program by including the respective #DEFINE compiler directive. Please see the list of available event directives in the AXI file. If a #DEFINE event directive in enabled, then the implementation callback method must exists in the consuming code base, else the program node will fail to compile. See the *#DEFINE Compiler Directive* section on page 157 for more information.
RmsGuiApi.axi	This include file provides helper functions that can simplify NetLinx integration with the RMS GUI component.
RmsMathUtil.axi	This include file provides math related helper functions. These functions make certain tasks easier such as the rounding of values and the scaling of levels.
RmsMonitorCommon.axi	This include file is used by each of the RMS Duet device monitoring modules. It acts much like a base class implementation where much of the details and logic for interacting with the RMS client via the RMS virtual device is placed here for reuse and to simplify the implementation in each monitoring module instance. This include listens for RMS specific asset management event notifications from the RMS virtual device and performs callback method calls to each monitoring module as needed.
RmsNlSnapiComponents.axi	This file is used to implement SNAPI listeners for NetLinx devices.
RmsNlTimer.axi	This file is used to implement a timer for NetLinx device monitoring module in RMS.
RmsRfid.axi	This include file contains the NetLinx sample code to implement asset tracking in RMS with AMX Anterus RFID readers and RFID tags. This code was placed in this include file to allow separation from the main RMS implementation code and allow for easy inclusion/exclusion.
RmsSchedulingApi.axi	This include file is provided to implement convenience methods and constants to aid in developing NetLinx code to communicate with RMS scheduling API's.
RmsSchedulingEventListener.axi	This include file is used to listen to the RMS client virtual device events and invoke callback methods when a RMS scheduling event occurs. Each event callback method can be exposed to the consuming program by including the respective #DEFINE compiler directive. See the list of available event directives in the AXI file. If a #DEFINE event directive in enabled, then the implementation callback method must exists in the consuming code base, else the program node will fail to compile.
RmsSourceUsage.axi	This include file contains the NetLinx sample code to implement asset source usage tracking in RMS. This code was placed in this include file to allow separation from the main RMS implementation code and allow for easy inclusion/exclusion. All sources that should have usage tracking must be registered with the *RmsSourceUsageMonitor* module. See the sample code in the AXI file for details.
RmsSystemEventHandler.axi	This include file contains the NetLinx sample code to implement system power ON/OFF and system mode changes advertised by the RMS client via button event and command notifications. This code was placed in this include file to allow separation from the main RMS implementation code and allow for easy inclusion/exclusion. System power and system mode features must be enabled include the *RmsControlSystemMonitor* module (see page 32).

Files Included in the Duet Workspace (RMS SDK - Duet.apw) Cont.	
File Name	**Description**
Module folder	
Note: Each Project in the Duet Workspace uses one or more of the following device monitors. The specific device monitors used depends on the devices defined in each Project.The device monitors listed below are Duet-specific. See the table on page 23 for a listing of NetLinx-specific device monitors. Also see page 25 for a listing of generic device monitors.	
RmsDuetAudioConferencerMonitor.axs	This device monitoring module provides monitoring and control to RMS for Audio Conferencers.
RmsDuetCameraMonitor.axs	This device monitoring module provides monitoring and control to RMS for Cameras.
RmsDuetDigitalSatelliteSystemMonitor	This device monitoring module provides monitoring and control to RMS for Digital Satellite Systems.
RmsDuetDiscDeviceMonitor	This device monitoring module provides monitoring and control to RMS for Disc Devices.
RmsDuetDocCameraMonitor	This device monitoring module provides monitoring and control to RMS for Document Cameras.
RmsDuetDVRMonitor	This device monitoring module provides monitoring and control to RMS for Digital Video Recorders.
RmsDuetLightSystemMonitor	This device monitoring module provides monitoring and control to RMS for Light Systems. *Warning: The light system monitor module will require custom programming to function properly*
RmsDuetMonitorMonitor	This device monitoring module provides monitoring and control to RMS for Monitors.
RmsDuetReceiverMonitor	This device monitoring module provides monitoring and control to RMS for Receivers.
RmsDuetSettopBoxMonitor	This device monitoring module provides monitoring and control to RMS for Set Top Boxes.
RmsDuetSwitcherMonitor	This device monitoring module provides monitoring and control to RMS for Switchers.
RmsDuetTVMonitor	This device monitoring module provides monitoring and control to RMS for TVs.
RmsDuetVideoConferencerMonitor	This device monitoring module provides monitoring and control to RMS for Video Conferencers.
RmsDuetVideoProjectorMonitor	This device monitoring module provides monitoring and control to RMS for Video Projectors.
User Interface folder	
RMS (800x480).TP4	The RMS Enterprise SDK includes a sample touch panel file that includes the application user interface implementation for sending Help and Maintenance requests, and viewing Hotlist Information.

RMS SDK - NetLinx.apw (Duet Workspace)

FIG. 2 shows the NetLinx Studio Workspace Bar with the RMS SDK - NetLinx.apw (NetLinx Workspace) open:

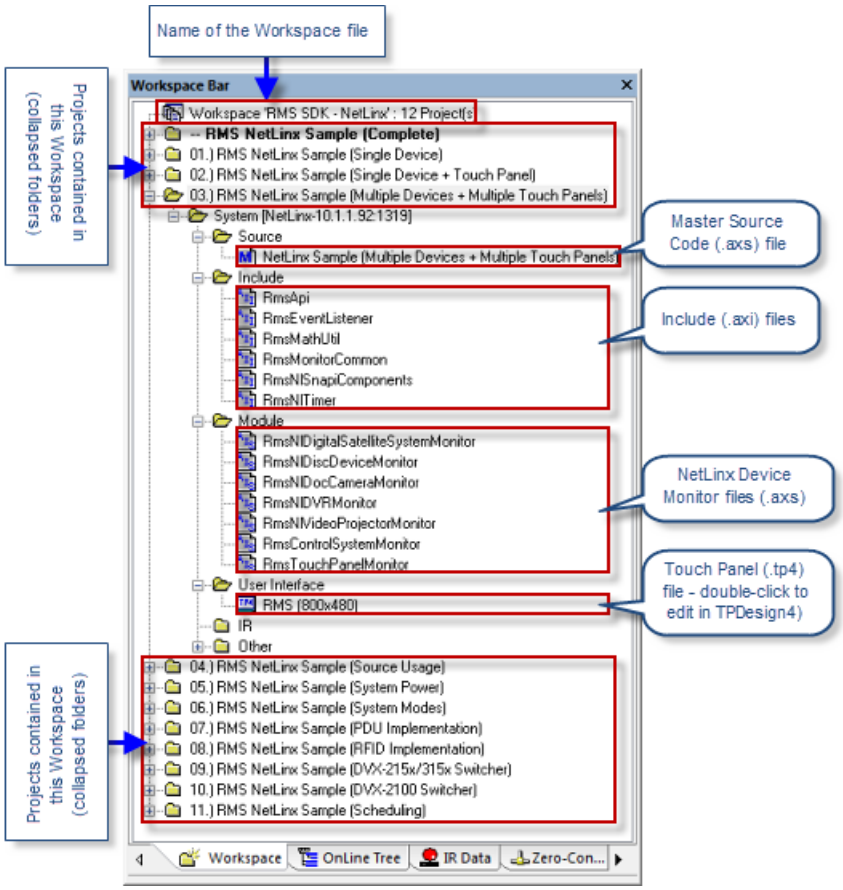


FIG. 3 NetLinx Studio - Workspace Bar (RMS SDK - NetLinx.apw)

As indicated in the Workspace Bar, the NetLinx Workspace provided with the RMS Enterprise SDK contains a total of 12 NetLinx-based projects, each of which presents a starting point for a specific type of NetLinx system.

Each Project folder contains a set of sub-folders that categorize the various Source, Include, Module and Touch Panel files that are included with each project:

Files Included in the NetLinx Workspace (RMS SDK - NetLinx.apw)	
File Name	**Description**
Source folder	
<Project Name>.axs.	This is the master Source Code file for the Project/System. The name applied to the Master Source Code file matches the name of it's parent Project.
Include folder	
RmsApi.axi	This include file is provided to implement many of the RMS API channels, levels, and command strings as constants and also to provide many helper functions that can simplify NetLinx integration with the RMS client.
RmsEventListener.axi	This include file is used to listen to the RMS client virtual device events and invoke callback methods when a RMS event occurs. Each event callback method can be exposed to the consuming program by including the respective #DEFINE compiler directive. Please see the list of available event directives in the AXI file. If a #DEFINE event directive in enabled, then the implementation callback method must exist in the consuming code base, or else the program node will fail to compile.
RmsGuiApi.axi	This include file provides helper functions that can simplify NetLinx integration with the RMS GUI component.
RmsMathUtil.axi	This include file provides math related helper functions. These functions make certain tasks easier such as the rounding of values and the scaling of levels.

Files Included in the NetLinx Workspace (RMS SDK - NetLinx.apw) Cont.	
File Name	**Description**
Include folder (Cont.)	
RmsMonitorCommon.axi	This include file is used by each of the RMS Duet device monitoring modules. It acts much like a base class implementation where much of the details and logic for interacting with the RMS client via the RMS virtual device is placed here for reuse and to simplify the implementation in each monitoring module instance. This include listens for RMS specific asset management event notifications from the RMS virtual device and performs callback method calls to each monitoring module as needed.
RmsNlSnapiComponents.axi	This file is used to implement SNAPI listeners for NetLinx devices.
RmsNlTimer.axi	This file is used to implement a timer for NetLinx device monitoring module in RMS.
RmsRfid.axi	This include file contains the NetLinx sample code to implement asset tracking in RMS with AMX Anterus RFID readers and RFID tags. This code was placed in this include file to allow separation from the main RMS implementation code and allow for easy inclusion/exclusion.
RmsSchedulingApi.axi	This include file is provided to implement convenience methods and constants to aid in developing NetLinx code to communicate with RMS scheduling API's.
RmsSchedulingEventListener.axi	This include file is used to listen to the RMS client virtual device events and invoke callback methods when a RMS scheduling event occurs. Each event callback method can be exposed to the consuming program by including the respective #DEFINE compiler directive. See the list of available event directives in the AXI file. If a #DEFINE event directive in enabled, then the implementation callback method must exists in the consuming code base, else the program node will fail to compile.
RmsSourceUsage.axi	This include file contains the NetLinx sample code to implement asset source usage tracking in RMS. This code was placed in this include file to allow separation from the main RMS implementation code and allow for easy inclusion/exclusion. All sources that should have usage tracking must be registered with the *RmsSourceUsageMonitor* module. See the sample code in the AXI file for details.
RmsSystemEventHandler.axi	This include file contains the NetLinx sample code to implement system power ON/OFF and system mode changes advertised by the RMS client via button event and command notifications. This code was placed in this include file to allow separation from the main RMS implementation code and allow for easy inclusion/exclusion. System power and system mode features must be enabled include the *RmsControlSystemMonitor* module (see page 32).
Module folder	
Note: Each Project in the NetLinx Workspace uses one or more of the following device monitors. The specific device monitors used depends on the devices defined in each Project.The device monitors listed below are NetLinx-specific. See the table on page 20 for a listing of Duet-specific device monitors. Also see page 25 for a listing of generic device monitors.	
RmsNlAudioConferencerMonitor.axs	This device monitoring module provides monitoring and control to RMS for Audio Conferencers.
RmsNlCameraMonitor.axs	This device monitoring module provides monitoring and control to RMS for Cameras.
RmsNlDigitalSatelliteSystemMonitor.axs	This device monitoring module provides monitoring and control to RMS for Digital Satellite Systems.
RmsNlDiscDeviceMonitor.axs	This device monitoring module provides monitoring and control to RMS for Disc Devices.
RmsNlDocCameraMonitor.axs	This device monitoring module provides monitoring and control to RMS for Document Cameras.
RmsNlDVRMonitor.axs	This device monitoring module provides monitoring and control to RMS for Digital Video Recorders.
RmsNlHVACMonitor.axs	This device monitoring module provides monitoring and control to RMS for HVAC systems.
RmsNlLightSystemMonitor.axs	This device monitoring module provides monitoring and control to RMS for Light Systems. *Warning - The light system monitor module will require custom programming to function properly.*
RmsNlMonitorMonitor.axs	This device monitoring module provides monitoring and control to RMS for Monitors.
RmsNlReceiverMonitor.axs	This device monitoring module provides monitoring and control to RMS for Receivers.
RmsNlSecuritySystemMonitor.axs	This device monitoring module provides monitoring and control to RMS for Security Systems.
RmsNlSettopBoxMonitor.axs	This device monitoring module provides monitoring and control to RMS for Set Top Boxes.
RmsNlSwitcherMonitor.axs	This device monitoring module provides monitoring and control to RMS for Switchers.
RmsNlTVMonitor.axs	This device monitoring module provides monitoring and control to RMS for TVs.
RmsNlVideoConferencerMonitor.axs	This device monitoring module provides monitoring and control to RMS for Video Conferencers.
RmsNlVideoProjectorMonitor.axs	This device monitoring module provides monitoring and control to RMS for Video Projectors.

Files Included in the NetLinx Workspace (RMS SDK - NetLinx.apw) Cont.	
File Name	**Description**
User Interface folder	
RMS (800x480).TP4	The RMS Enterprise SDK includes a sample touch panel file that includes the application user interface implementation for sending Help and Maintenance requests, and viewing Hotlist Information.

Monitors - Generic

The RMS Enterprise SDK includes the following default asset monitoring and generic management implementations for supported device types.

RMS Enterprise SDK Components - Generic Device Monitors	
Component	**Description**
RmsControlSystemMonitor.axs	This device monitoring module provides monitoring and control to RMS for the NetLinx master.
RmsDvx2100HDSwitcherMonitor.axs	TBD
RmsDvxSwitcherMonitor.axs	TBD
RmsGenericNetLinxDeviceMonitor.axs	This module provides basic device monitoring to RMS for AMX devices.
RmsPowerDistributionUnitMonitor.axs	This module provides power/energy monitoring and management to RMS for PDU devices.
RmsRfidReaderMonitor.axs	This module implements device monitoring for AMX Anterus RFID reader devices with RMS.
RmsSourceUsageMonitor.axs	This module implements source usage tracking with RMS.
RmsSystemModeMonitor.axs	This module provides monitoring and control for system modes.
RmsSystemPowerMonitor.axs	This module provides basic monitoring and control for system power.
RmsTouchPanelMonitor.axs	This device monitoring module provides monitoring and control to RMS for G3/G4 touch panels.
RmsVirtualDeviceMonitor.axs	This module provides the hooks for monitoring the RMS virtual device.

Default File Locations of NetLinx Studio Workspace and Support Files

File Locations NetLinx Studio Workspace and Support Files	
Workspace Files (*.APW)	Program Files > AMX > Resource Management Suite > sdk > netlinx > 4.1.<x>
Include Files	Program Files > AMX > Resource Management Suite > sdk > netlinx > 4.1.<x> includes
Generic Device Monitors	Program Files > AMX > Resource Management Suite > sdk > netlinx > 4.1.<x> monitors
Duet Device Monitors	Program Files > AMX > Resource Management Suite > sdk > netlinx > 4.1.<x> monitors-duet
NetLinx Device Monitors	Program Files > AMX > Resource Management Suite > sdk > netlinx > 4.1.<x> monitors-netlinx
Touch Panel File	Program Files > AMX > Resource Management Suite > sdk > netlinx > 4.1.<x> touch panels

RMS Enterprise SDK - "Help" directory

The following files are included in the RMS Enterprise SDK *Help* directory:

Program Files > AMX > Resource Management Suite > sdk > netlinx > 4.1.<x> > help

RMS Enterprise SDK Components - Files in the "Help" Directory	
Component	**Type**
RMS NetLinx Programmer's Guide.pdf	This is the PDF file of the *RMS Enterprise NetLinx Programmer's Guide* (this document).
UNWISE.EXE	Double-click this file to un-install the RMS Enterprise SDK.

Other files present in this directory are required to support the installation and un-installation of the RMS SDK are not intended to be viewed or edited.

RMS Enterprise SDK - Support Files

The following files are also included in the RMS Enterprise SDK:

RMS Enterprise SDK Components - Other Files	
Component	**Type**
commons-codec-1.4-AMX-01.jar	This file is required by RMS and is not intended to be modified.
commons-httpclient-3.1-AMX-01.jar	This file is required by RMS and is not intended to be modified.
commons-lang-2.5-AMX-01.jar	This file is required by RMS and is not intended to be modified.
commons-logging-1.1.1-AMX-01.jar	This file is required by RMS and is not intended to be modified.
rmsclient.jar	This file is required by RMS and is not intended to be modified.
RmsClientGui_dr4_0_0.jar	This file is required by RMS and is not intended to be modified.
RmsNetLinxAdapter_dr4_0_0.jar	This file is required by RMS and is not intended to be modified.

RMS Enterprise SDK Components - Other Files (Cont.)	
Component	**Type**
rmsnlplatform.jar	This file is required by RMS and is not intended to be modified.
Readme.html	This is an HTML-formatted version of the readme (release notes) associated with this release of the RMS Enterprise SDK.
Readme.txt	This is an ASCII text version of the readme (release notes) associated with this release of the RMS Enterprise SDK.
rms-sdk-manifest.xml	This file provides a listing of all files included in this release of the RMS SDK.

RMS SDK File Dependencies

FIG. 4 shows the dependencies and relations between the files involved in an RMS Enterprise system.

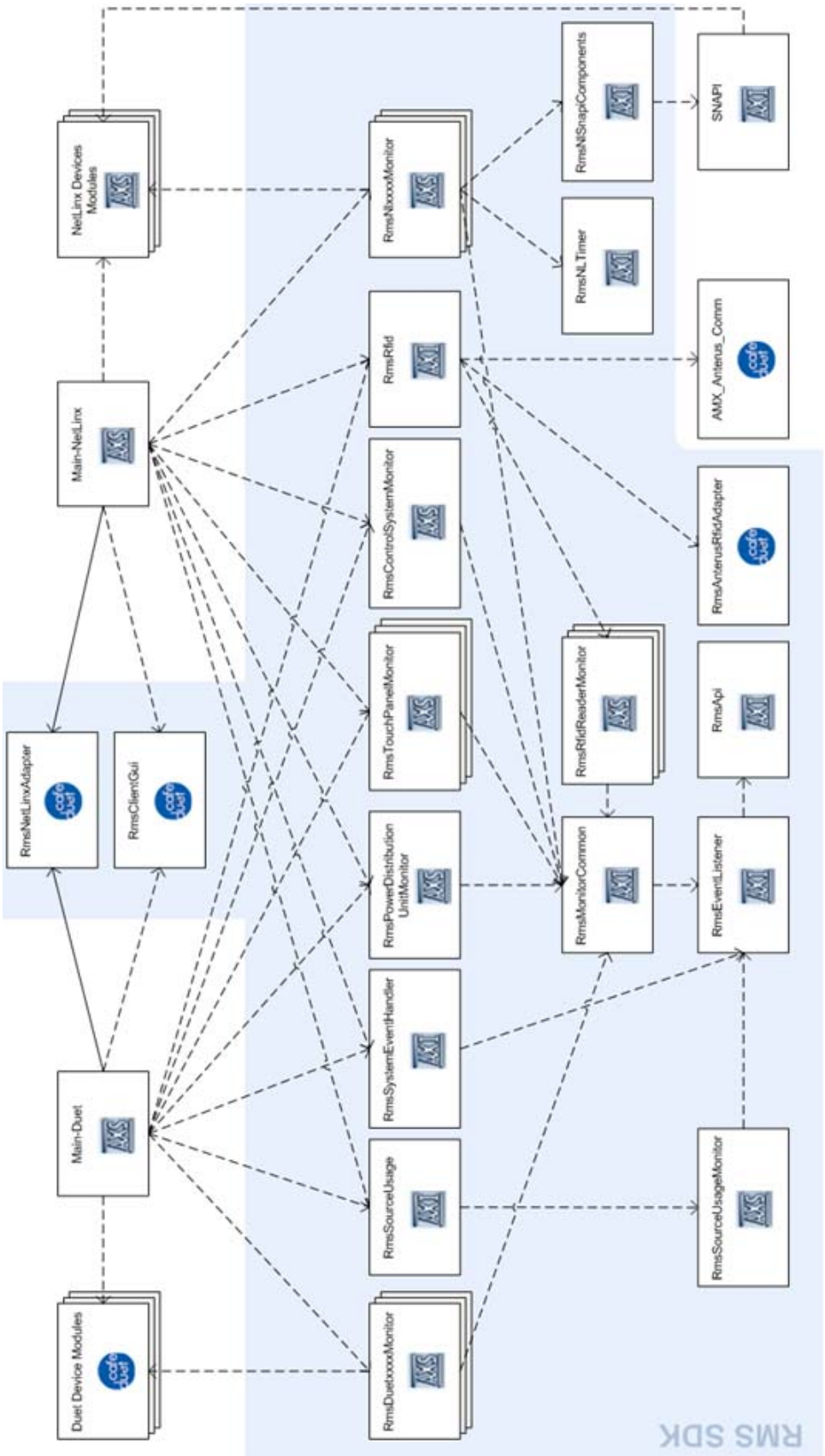


FIG. 4 RMS SDK File Dependencies

RMS Client Overview

FIG. 5 illustrates the RMS Client components deployed to a NetLinx system.

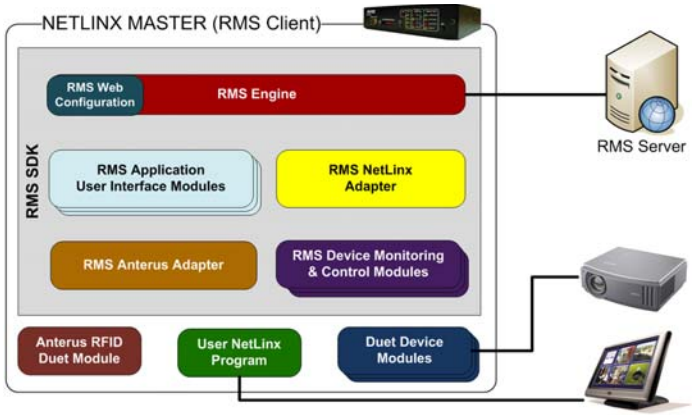


FIG. 5 RMS Client components Deployed to a NetLinx System

RMS Engine

The RMS Engine is deployed to the NetLinx master as an internal Java OSGi bundle and provides the RMS subsystem and communication layers for accessing and communicating with a RMS server. The RMS Engine is automatically deployed on a master when a user includes the RMS NetLinx Adapter module in their NetLinx program.

RMS Client Web Configuration

The RMS Engine provides a web based user interface hosted on the NetLinx master's web server that enables RMS Client configuration and testing the connection to the RMS server.

You can access the RMS web configuration pages via the *Manage Devices* tab under the **System** management section on the NetLinx master's configuration web pages (FIG. 6).

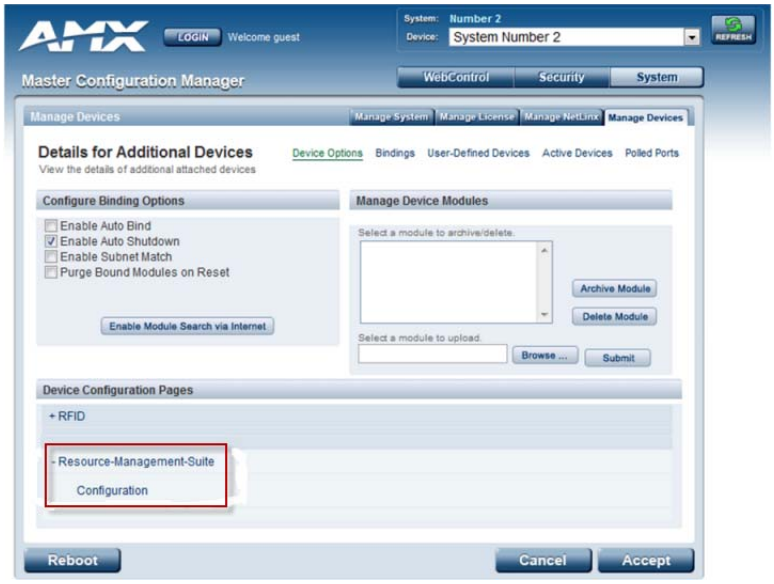


FIG. 6 RMS Client Web Configuration

Once loaded, you can configure the RMS server URL, password, and client gateway name (FIG. 7).

FIG. 7 RMS Client Web Configuration

When setting up or changing the RMS configuration settings, always use the **Test Connection** button to verify that the RMS client can successfully connect and communicate with the RMS server.

RMS NetLinx Adapter Module

The RMS NetLinx Adapter module exposes the RMS NetLinx Virtual Device API for consumption in NetLinx programs. This module is required to interact with RMS from a NetLinx program.

For more information on the RMS NetLinx API, see the *RMS NetLinx Virtual Device API* section on page 120.

RMS Virtual Device (vdvRMS)

This device will be the NetLinx virtual Device ID used to communicate with the RMS subsystem from a NetLinx program (Note this Device ID must exist in the Duet Device ID range).

The following code can be included to load the RMS NetLinx Adapter module.

```
DEFINE_DEVICE

vdvRMS            = 41001:1:0 // RMS Client (Duet Module)

DEFINE_START
// RMS Client - NetLinx Adapter Module
//   This module includes the RMS client module
//   and enables communication via SEND_COMMAND,
//   SEND_STRINGS, CHANNELS, and LEVELS with the
//   RMS Client.
DEFINE_MODULE 'RmsNetLinxAdapter_dr4_0_0' mdlRMSNetLinx(vdvRMS)
```

NOTE: *This module should be defined in the user program before any other modules.*

RMS Client Application User Interface Module

RMS can expose RMS-specific user interface functionality including help & maintenance requests and a location Hotlist view.

To enable this functionality, an array of touch panels must be defined, and then the RMS Client Application User Interface module must be instantiated. RMS Touch Panel files are also provided in the RMS Enterprise SDK. These touch panel files will need to be integrated with any existing TPD file to add the RMS functionality.

By default, all RMS user interface implementation is defined on **Port 7** of the touch panel.

The following code illustrates the inclusion of the RMS Client Application User Interface module.

```
DEFINE_DEVICE

dvTP1               = 10001:1:0 // Touch Panels
dvTP2               = 10002:1:0 // (must be port 1 for base device)
dvTP3               = 10003:1:0
dvTP1_RMS           = 10001:7:0 // RMS Touch Panels
dvTP2_RMS           = 10002:7:0 // (RMS uses port 7 by default)
dvTP3_RMS           = 10003:7:0
DEFINE_VARIABLE

// RMS Touch Panel Array
VOLATILE DEV dvRMSTP[] =
{
   dvTP1_RMS,
   dvTP2_RMS,
   dvTP3_RMS
}
// RMS Touch Panel Array -
// Base Device for System Keyboard handling
```

```
VOLATILE DEV dvRMSTP_Base[] =
{
   dvTP1,
   dvTP2,
   dvTP3
}

DEFINE_START
//   RMS GUI - User Interface Module
//   This module is responsible for all the RMS
//   user interface application logic. This
//   includes help requests, maintenance requests,
//   location hotlist, and server display messages.
DEFINE_MODULE 'RmsClientGui_dr4_0_0' mdlRMSGUI(vdvRMSGui,
                                               dvRMSTP,
                                               dvRMSTP_Base);
```

RMS Touch Panel Monitor Module

The RMS Touch Panel Monitor module is used to register, monitor, and control each physical AMX touch panel connected to the control system master with RMS. Each touch panel configured for use with an instance of the RMS Touch Panel Monitor module will show up in RMS as a separate asset and tracked accordingly.

The diagram in FIG. 8 illustrates the file dependencies for integrating the RMS Touch Panel Monitor module:

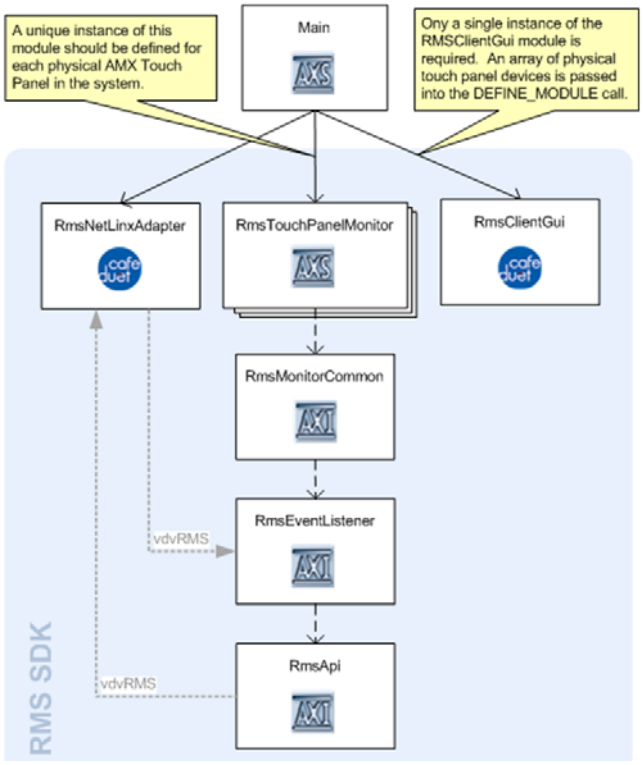


FIG. 8 RMS SDK File Dependencies - RFID Application User Interface Implementation

The following code illustrates the inclusion of the RMS Touch Panel Monitor module.

```
DEFINE_DEVICE

dvTP1               = 10001:1:0 // Touch Panels
dvTP2               = 10002:1:0 // (must be port 1 for base device)
dvTP3               = 10003:1:0
dvTP1_RMS           = 10001:7:0 // RMS Touch Panels
dvTP2_RMS           = 10002:7:0 // (RMS uses port 7 by default)
dvTP3_RMS           = 10003:7:0

//
// AMX Touch Panel Monitor
//
// - include a touch panel monitoring module instance for each
//    touch panel device you wish to monitor.
//
DEFINE_MODULE 'RmsTouchPanelMonitor' mdlRmsTouchPanelMonitorMod_1(vdvRMS,dvTP1);
DEFINE_MODULE 'RmsTouchPanelMonitor' mdlRmsTouchPanelMonitorMod_2(vdvRMS,dvTP2);
DEFINE_MODULE 'RmsTouchPanelMonitor' mdlRmsTouchPanelMonitorMod_3(vdvRMS,dvTP3);
```

The RMS Touch Panel Monitor module supports both G3 and G4 AMX Touch Panels. Depending on which touch panel model and firmware you have, the RMS Touch Panel Monitor module will attempt to monitor a variety of parameters for the touch panel.

The RMS Touch Panel Monitor module is provided as an open source NetLinx module so that you can modify a copy of it to meet any custom requirements.

RMS Touch Panel Files

The RMS Enterprise SDK includes sample touch panel files in the following resolutions:

- 800 x 600
- 800 x 480

The touch panel file include the application user interface implementation for:

- Send Help Requests (see *Sending Help Requests to RMS Server* on page 110)
- Send Maintenance Requests (see *Sending Maintenance Requests to RMS Server* on page 110)
- View Location Hotlist Information

RMS RFID Adapter Module & RMS RFID Monitor (Anterus)

RMS supports RFID asset tracking with the use of an AMX Anterus RFID system. The RMS SDK includes a "*RmsRfid.axi*" Include File that contains all the necessary RFID module declarations and a place to add the RFID read device DPS entries.

The diagram in FIG. 9 illustrates the file dependency chain for RMS & RFID integration:

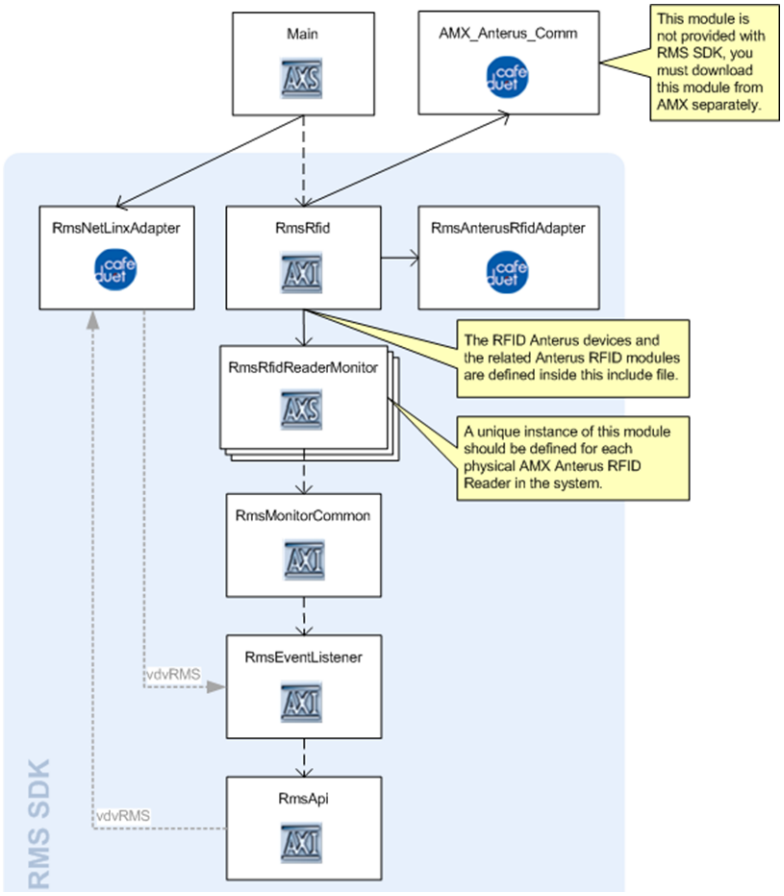


FIG. 9 RMS SDK File Dependencies - RFID Implementation

RmsRfid.axi Include file

The following code illustrates the inclusion of the *RmsRfid.axi* file in the main NetLinx program:

```
DEFINE_START
// RMS RFID Support
#WARN 'Uncomment this line to include RFID support with RMS'
#INCLUDE 'RmsRfid'
```

The Anterus RFID reader devices and Anterus Module should be configured in the *RmsRfid.axi* file.

To have the Anterus RFID reader show up as a monitored asset in the RMS system, you must include a separate instance of the RMS RFID Reader Monitor module for each physical Anterus RFID reader:

```
DEFINE_DEVICE
vdvRMSRfid        = 41003:1:0  // RMS RFID System Adapter    (Duet Module)
vdvAnterusGateway = 41004:1:0  // Duet RFID Virtual Device   (Duet Module)
dvAnterusReader1  =    99:1:0  // AxLink Anterus RFID Reader #1

DEFINE_START

//   RMS RFID - RFID System Monitor
//   This module is responsible for monitoring RFID
//   readers and tags for an AMX Anterus RFID system.
//   This module will relay RFID information to the
//   RMS system.
//  (Note: RMS server must be enabled for RFID support.)
DEFINE_MODULE 'RmsAnterusRFIDAdapter_dr4_0_0' mdlRMSAnterusAdapter(vdvRMSRfid,vdvAnterusGateway)

//   Anterus RFID Duet Module
//   This module enabled RFID management of AMX
//   Anterus RFID readers and tags.
DEFINE_MODULE 'AMX_Anterus_Comm_dr1_0_0' mdlAnterusDuetMod(vdvAnterusGateway,dvAnterusReader1)
//   RMS RFID Reader Device Monitor
//   (include a separate module instance for each physical RFID reader device)
DEFINE_MODULE 'RmsRfidReaderMonitor' mdlRmsRfidReaderMonitorMod_1(vdvRMS,dvAnterusReader1)
DEFINE_MODULE 'RmsRfidReaderMonitor' mdlRmsRfidReaderMonitorMod_2(vdvRMS,dvAnterusReader2)
DEFINE_MODULE 'RmsRfidReaderMonitor' mdlRmsRfidReaderMonitorMod_3(vdvRMS,dvAnterusReader3)
```

NOTE: *The AMX_Anterus_Comm_dr1_0_0.jar module file is not included in the RMS Enterprise SDK. This file must be downloaded separately (a link to download this file is provided on the ANT-RDR catalog page at www.amx.com (Products > Control System Accessories > ANT-RDR).*

RMS Control System Monitor

In RMS Enterprise, the control system master should treated like any other managed and monitored asset in the system. There is a special *RmsControlSystemMonitor* module that should be included in the user program to ensure the control system asset get registered as an asset with the RMS system. Only one instance of this module should be defined on any given NetLinx system.

The diagram in FIG. 10 illustrates the file dependency chain for the *RmsControlSystemMonitor* integration:

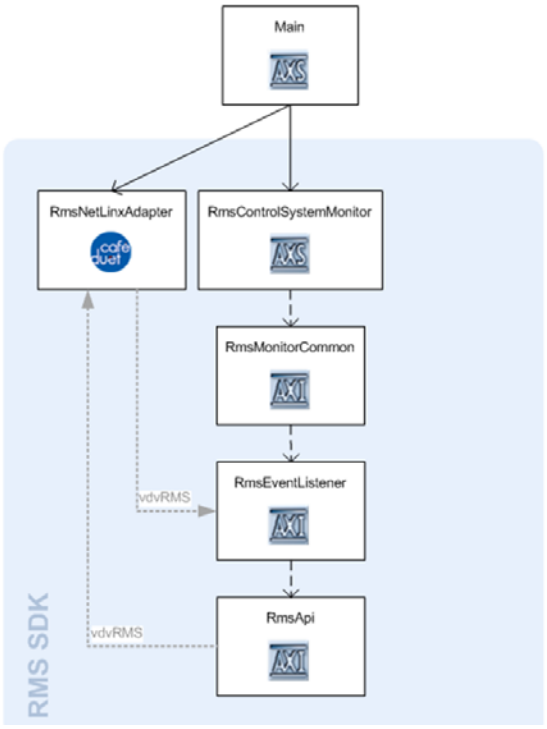


FIG. 10 RMS SDK File Dependencies - Control System Monitor Implementation

The following code illustrates the inclusion of the *RmsControlSystemMonitor* module in the main NetLinx program.

```
//
// AMX Control System Master
//
// - include only one of these control system device monitoring modules.
//   this is intended to serve as an extension point for creating
//   system wide control methods and system level monitoring parameters.
//
DEFINE_MODULE 'RmsControlSystemMonitor' mdlRmsControlSystemMonitorMod(vdvRMS, dvMaster);
```

It is important to note that any location-wide monitoring or control functionality should be implemented as part of the control system asset. This include location-scoped concepts such as System Power and System Modes.

For more details on how to implement System Power or System Modes, please see *Implementing System Power* on page 113 and *Implementing System Modes* on page 113.

RMS System Event Handler

The RMS Enterprise SDK provides an Include File to allow the NetLinx programmer to easily listen for and respond to System generated events such as System Power changes and System Mode changes. This *RmsSystemEventHandler* Include File automatically registers itself as an event listener to the RMS NetLinx virtual device (via the *RmsEventListener* Include File) and subscribes to the system power and system mode event notifications. These event notifications are delivered as callback method invocations to the following functions defined in the *RmsSystemEventHandler* Include File.

- RmsEventSystemPowerChangeRequest (see page 35)
- RmsEventSystemPowerChanged (see page 35)
- RmsEventSystemModeChangeRequest (see page 36)
- RmsEventSystemModeChanged (see page 36)

The diagram in FIG. 11 on page 34 illustrates the file dependency chain for the *RmsSystemEventListener.axi* Include File integration:

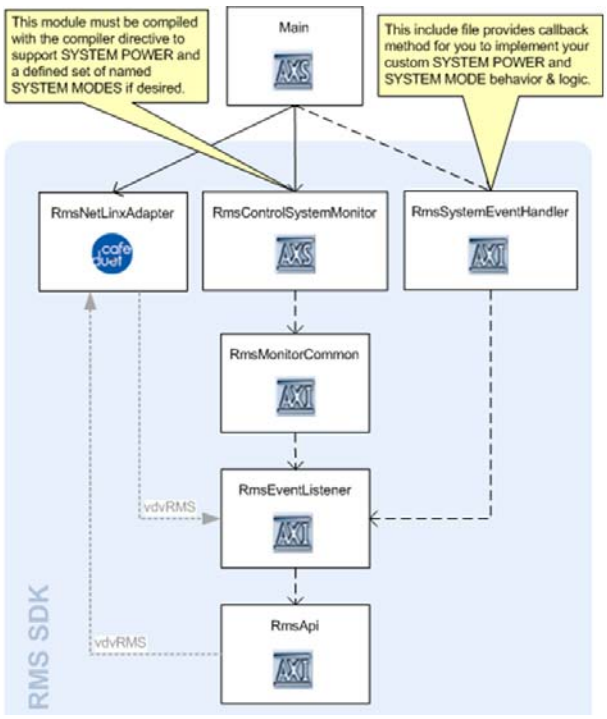


FIG. 11 RMS SDK File Dependencies - System Event Handler

The following code illustrates the inclusion of the *RmsSystemEventHandler.axi* Include File in the main NetLinx program:

```
//
// RMS SYSTEM POWER / SYSTEM MODE EVENT HANDLER
//  system power and system mode monitoring and tracking
//
//  If you do not need or wish to use the system power
//  or system mode concepts, then you may comment out
//  or delete this include file reference.
//
//  If you do intend to implement system power states
//  and/or system mode selections, then please see contents
//  of this include file. You are responsible for
//  implementing the logic for selecting/deselecting
//  sources in your own NetLinx source code.
//
#INCLUDE 'RmsSystemEventHandler';
```

It is important to note that by default RMS does not implement System Power or System Modes. They must be enabled in the *RmsControlSystemMonitor* module and the user program must implement the necessary behavior and logic when these events are triggered. The RMS SDK provides the infrastructure for routing these notifications, but does not provide a default usage implementation.

For more details on how to implement System Power or System Modes, please see the *Implementing System Power* section on page 113 and the *Implementing System Modes* section on page 113.

Functions defined in RmsSystemEventHandler.axi

The following table describes the Functions defined in *RmsSystemEventHandler.axi*:

RmsSystemEventHandler.axi - Functions	

RmsEventSystemPowerChangeRequest	*Description*: RMS system power change request notification.
	• This callback method is invoked by the '*RmsEventListener'* Include File to notify this program when a system power state change has been requested.
	• This method should not be invoked/called by any user implementation code.
	Arguments:
	• *powerOn* - boolean bit TRUE|FALSE for system power state
	Syntax:
	`DEFINE_FUNCTION RmsEventSystemPowerChangeRequest(CHAR powerOn)`
	`{`
	`  //`
	`  // (RMS SYSTEM POWER CHANGE REQUEST NOTIFICATION)`
	`  //`
	`  IF(powerOn)`
	`  {`
	`  // system power ON request received`
	`  }`
	`  ELSE`
	`  {`
	`  // system power OFF request received`
	`  }`
	`}`

RmsEventSystemPowerChanged	*Description*: RMS system power change notification.
	• This callback method is invoked by the '*RmsEventListener.axi*' Include File to notify this program when the system power state has changed.
	• This method should not be invoked/called by any user implementation code.
	Arguments:
	• *powerOn* - boolean bit TRUE|FALSE for system power state
	Syntax:
	`DEFINE_FUNCTION RmsEventSystemPowerChanged(CHAR powerOn)`
	`{`
	`  //`
	`  // (RMS SYSTEM POWER CHANGE NOTIFICATION)`
	`  //`
	`  // upon receiving the system power change notification from`
	`  // the RMS client the user code should implement the necessary`
	`  // code logic to power the 'SYSTEM' ON or OFF`
	`  //`
	`  SEND_STRING 0, '************************************';`
	`  IF(powerOn)`
	`  {`
	`  SEND_STRING 0, ' SYSTEM POWER [ON] ';`
	`  }`
	`  ELSE`
	`  {`
	`  SEND_STRING 0, ' SYSTEM POWER [OFF] ';`
	`  }`
	`  SEND_STRING 0, '************************************';`
	`  #WARN 'Implement your SYSTEM POWER [ON/OFF] logic here!'`
	`}`

RmsSystemEventHandler.axi - Functions (Cont.)	

RmsEventSystemModeChangeRequest	*Description*: RMS system mode change request notification.
	• This callback method is invoked by the '*RmsEventListener.axi*' Include File to notify this program when the system operating mode has requested to be changed.
	• This method should not be invoked/called by any user implementation code.
	Arguments:
	• *modeName* - mode name for the requested system operating mode
	Syntax:
	`DEFINE_FUNCTION RmsEventSystemModeChangeRequest(CHAR newMode[])`
	`{`
	`  //`
	`  // (RMS SYSTEM MODE CHANGE REQUEST NOTIFICATION)`
	`  //`
	`  // upon receiving the system mode change request`
	`  // event notification from the RMS client`
	`  //`
	`  SEND_STRING 0, '*********************************************';`
	`  SEND_STRING 0,"' SYSTEM MODE CHANGE REQUESTED [',newMode,']'";`
	`  SEND_STRING 0, '*********************************************';`
	
	`  // call your own local function to perform the system mode change`
	`  ChangeMySystemMode(newMode);`
	`}`

RmsEventSystemModeChanged	**Description**: RMS system mode change notification.
	• This callback method is invoked by the '*RmsEventListener.axi*' Include File to notify this program when the system operating mode has changed.
	• This method should not be invoked/called by any user implementation code.
	Arguments:
	• *modeName* - mode name for the newly applied system operating mode
	Syntax:
	`DEFINE_FUNCTION RmsEventSystemModeChanged(CHAR modeName[])`
	`{`
	`  //`
	`  // (RMS SYSTEM MODE CHANGE NOTIFICATION)`
	`  //`
	`  // upon receiving the system mode change event`
	`  // notification from the RMS client`
	`  //`
	`  SEND_STRING 0, '************************************';`
	`  SEND_STRING 0,"' SYSTEM MODE CHANGE [',modeName,']'";`
	`  SEND_STRING 0, '************************************';`
	`}`

RMS Power Distribution Unit (PDU) Monitor

Devices such as the NXA-PDU-1508 provide an interesting combination of integration requirements. The PDU unit itself is an asset with many parameters to be registered and monitored in RMS. However, the monitored usage of power for each outlet may truly represent real power consumption on behalf of another asset being registered and monitored in RMS. This scenario presents a use case where an asset is tracking parameter changes on behalf of another asset. The RMS Enterprise SDK provides a default monitoring module for the PDU device to accommodate this cross-asset monitoring requirement.

The diagram in FIG. 12 illustrates the file dependency chain for the *RmsPowerDistributionUnitMonitor* module integration:

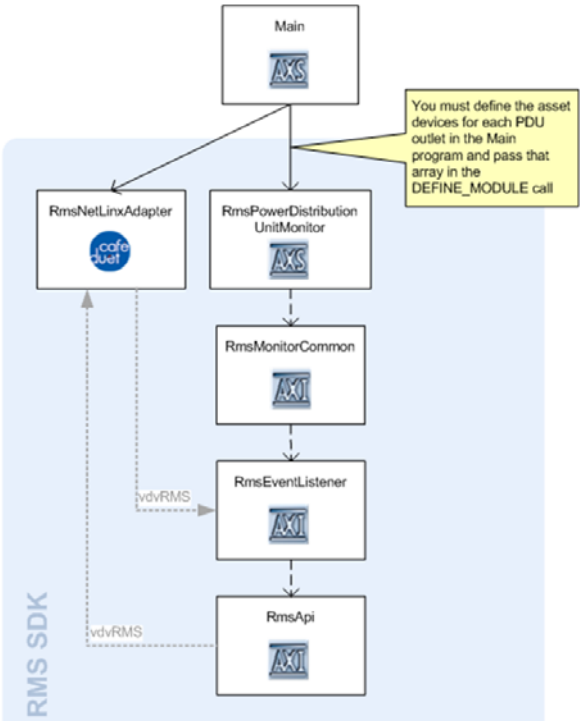


FIG. 12 RMS SDK File Dependencies - Power Distribution Unit Implementation

The code below illustrates the NetLinx code required to integrate the PDU monitor module. The first thing that must be defined is the PDU base device. The NXA-PDU-1508 is a collection of 8 AxLink devices. For the purposes of RMS, only the first (base) address must be defined:

```
DEFINE_DEVICE
dvPDU_Base          =     96:1:0  // AMX PDU (Physical Device (first device))
```

Next, an array of devices must be defined that map asset relationships to each of the outlets on the PDU unit. The code snippet below illustrates this array definition.

```
// PDU Energy./Power Monitored Device Array
//   The PDU can support up to 8 devices, 1 in each
//   of the 8 physical outlets on the unit.

VOLATILE DEV dvPowerMonitoredDevices[8] =
{
        dvDVR,            // Outlet #1
        dvDiscDevice,     // Outlet #2
        dvDocCamera,      // Outlet #3
        dvDSS,            // Outlet #4
        dvVideoProjector, // Outlet #5

        0:0:0,            // Outlet #6 -- null device (nothing plugged in here...)
        0:0:0,            // Outlet #7 -- null device (nothing plugged in here...)
        0:0:0             // Outlet #8 -- null device (nothing plugged in here...)
}
```

Finally, the *RmsPowerDistrubitionUnitMonitor* module must be defined:

```
//
// Power Distribution Unit RMS Monitor
//
// - include one of these for each physical PDU hardware unit.
//   this module will monitor and report power and energy usage
//   information into the RMS system.
//
DEFINE_MODULE 'RmsPowerDistributionUnitMonitor' mdlRmsPowerDistributionUnitMonitorMod(vdvRMS, dvPDU_Base,
dvPowerMonitoredDevices);
```

This *RmsPowerDistributionUnitMonitor* module is provided as a full open source NetLinx module in the RMS SDK. Sample source files are included in the RMS SDK that provide a complete PDU integration example.

RMS Source Usage Monitor Module & Include File

The RMS SDK includes a *RmsSourceUsageMonitor* module and a *RmsSourceUsage.axi* Include File to provide source device usage tracking in the RMS system. Source usage tracking can provide a site some insight into how frequently their installed devices are actively being used and what devices may be in high demand versus devices that are used infrequently.

The source usage tracking in the RMS Enterprise SDK differs from the implementation in prior versions in that it avoids making assumptions about how device usage should be tracked. This means that the NetLinx programmer implementing RMS and source usage tracking has a greater responsibility in controlling when source devices are active and when they have been deactivated. The implementation in RMS Enterprise provides greater capabilities and tools to help the programmer accomplish these goals.

The source usage tracking in the RMS Enterprise SDK works by registering each source asset that you wish to track to a unique index number. Then there are several API calls you can use to control which sources are activated and deactivated. The RMS Enterprise SDK supports both mutually exclusive and non-mutually exclusive source device tracking.

Mutually Exclusive Source Usage Tracking

Mutually exclusive source usage tracking only allows a single source's asset to be active at any given time. When using mutually exclusive source tracking, the implementation programmer is only responsible for 'Activating' the newly selected source. The RMS source usage tracking module will automatically 'Deactivate' any other currently activated mutually exclusive source.

Non-Mutually Exclusive Source Usage Tracking

Non-mutually exclusive source usage tracking allows any combination of sources to be active concurrently at any given time. When using non-mutually exclusive source tracking, the implementation programmer is 100% responsible for coordinating the 'Activated' and 'Deactivated' states for each source index in the source usage tracking monitor.

Non-mutually exclusive source usage tracking is provided to allow you to handle complex or more sophisticated setups where perhaps more than one source can be active at the same time such as cases with split or quad screen setups, or in cases where there are multiple display outputs devices that can accept a variety of different selected sources.

NOTE: *If you combine both mutually exclusive and non-mutually exclusive source usage registrations, then be aware that you cannot reuse the index number. Each source usage asset must be assigned exclusively to a single index number.*

The diagram in FIG. 13 illustrates the file dependency chain for the *RmsSourceUsageMonitor* module and *RmsSourceUsage* Include File integration:

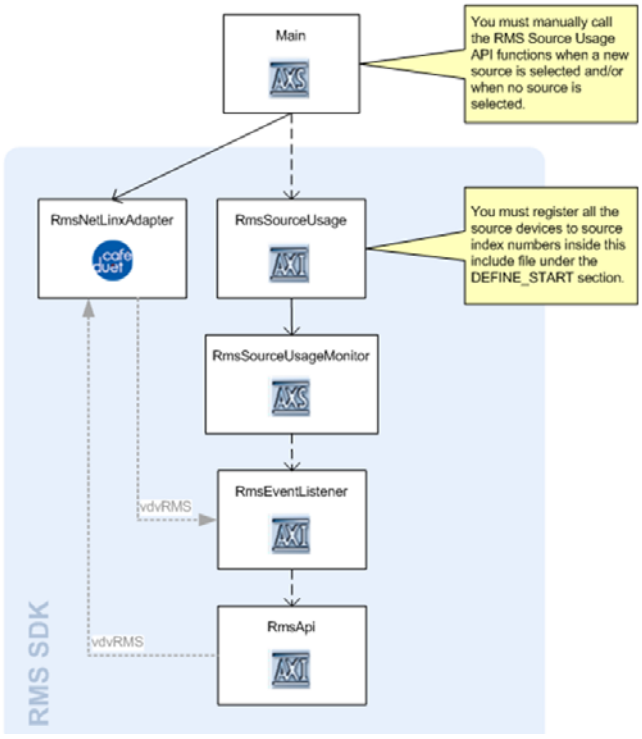


FIG. 13 RMS SDK File Dependencies - Source Usage Implementation

The *RmsSourceUsage.axi* Include File contains the necessary source usage virtual device definition, a location to register each source tracked asset to a source index number, and several convenience functions to help manage the selected (active) source asset states.

The code snippets below illustrates the NetLinx code required to integrate the *RmsSourceUsageMonitor* module. (This is located inside the *RmsSourceUsage.axi*)

First, a source usage NetLinx virtual device must be defined. This virtual device will be used to communicate with the source usage monitor module.

```
DEFINE_DEVICE

// define virtual device for RMS source usage
vdvRMSSourceUsage = 33002:1:0   // RMS Source Usage Monitor
```

The *RmsSourceUsageMonitor* module must be defined, passing in the RMS NetLinx virtual device along with the newly created source usage virtual device.

```
//
// Source Usage Monitoring Module
//
// - include only one of these source usage modules if you wish to support
//   source usage monitoring for assets.
//
DEFINE_MODULE 'RmsSourceUsageMonitor' mdlRmsSourceUsageMonitorMod(vdvRMS,vdvRMSSourceUsage);
```

Shortly after the program starts, each source tracked asset/device needs to be registered to a specific source tracking index number:

```
(***********************************************************)
(*               STARTUP CODE GOES BELOW                   *)
(***********************************************************)
DEFINE_START
// wait 5 seconds before assigning sources

WAIT 50
{
  #WARN 'Assign all assets that should participate in source usage tracking here ...'


  // add all assets to tracked for source usage
  // using the mutually exclusive assignment method means that
  // only a single mutually exclusive source can be active at
  // any given time. If a source asset is activated, then all
  // other sources configured as mutually exclusive will
  // be automatically deactivated.
  // set of sources.
  RmsSourceUsageAssignAssetMutExcl(1, dvDSS);
  RmsSourceUsageAssignAssetMutExcl(2, dvDVR);
  RmsSourceUsageAssignAssetMutExcl(3, dvDiscDevice);
  RmsSourceUsageAssignAssetMutExcl(4, dvDocCamera);
```

Now that all the tracked sources are registered with the *RmsSourceUsageMonitor* module, the NetLinx programmer must implement the tracking code to tell the *RmsSourceUsageMonitor* module when sources are activated (and possibly de-activated).

The *RmsSourceUsageMonitor* module and *RmsSourceUsage* include files are provided as a full open source NetLinx code in the RMS SDK. Sample source files are included in the RMS SDK that provide a complete source usage integration example. For more detailed programming information, see *Tracking Source Usage* on page 112.

RmsSourceUsage.axi - Wrapper Functions

The "*RmsSourceUsage.axi*" Include File contains the following set of wrapper functions to make tracking source usage as simple as possible:

RmsSourceUsage.axi - Wrapper Functions	

RmsSourceUsageReset	*Description*: This function is used to deactivate and reset all source selected asset tracking. Call this method anytime a condition exists where are selected sources should be de-selected and no active sources apply.
	• This reset will also reset any currently pending cached source usage.
	• This function is typically only used on program startup.
	Arguments:
	• none
	Syntax:
	`DEFINE_FUNCTION RmsSourceUsageReset()`
	`{`
	`  // to reset the selected source and all cached source usage`
	`  SEND_COMMAND vdvRMSSourceUsage, 'SOURCE.RESET';`
	`}`

RmsSourceUsage AssignAsset	*Description*: This function is used to assign a RMS asset to be monitored for non-mutually exclusive source usage tracking. This method should be called on startup to assign each source based asset to an index for source usage monitoring.
	Non-mutually exclusive source usage tracking allows any combination of sources to be active concurrently at any given time. When using non-mutually exclusive source tracking, you the implementation programmer are 100% responsible for coordinating the 'Activated' 'Deactivated' states for each source index in the source usage tracking monitor.
	Non-mutually exclusive source usage tracking is provided to allow you to handle complex or more sophisticated setups where perhaps more than one source can be active at the same time such as cases with split or quad screen setups or in cases where there are multiple display outputs devices that can accept a variety of difference selected sources.
	Note: If you combine both mutually exclusive and non-mutually exclusive source usage registrations, then be aware that you cannot reuse index number. Each source usage asset but be assigned exclusively to a single index number.
	Arguments:
	• **index** - source index position for source device
	• **sourceDevice** - device to use as a source device
	Syntax:
	`DEFINE_FUNCTION RmsSourceUsageAssignAsset(INTEGER index, DEV sourceDevice)`
	`{`
	`  STACK_VAR CHAR commandString[RMS_MAX_CMD_LEN];`
	
	`  commandString = RmsPackCmdHeader('SOURCE.ASSIGN');`
	`  commandString = RmsPackCmdParam(commandString,ITOA(index));`
	`  commandString = RmsPackCmdParam(commandString,RmsDevToString`
	`  (sourceDevice));`
	
	`  SEND_COMMAND vdvRMSSourceUsage, commandString;`
	`}`

RmsSourceUsage AssignAssetMutExcl	*Description*: This function is used to assign a RMS asset to be monitored as a mutually exclusive source. This method should be called on startup to assign each source based asset to an index for source usage monitoring.
	Mutually exclusive source usage tracking only allows a single sources asset to be active at any given time. When using mutually exclusive source tracking, you the implementation programmer are only responsible for 'Activating' the newly selected source. The RMS source usage tracking module will automatically 'Deactivate' all other currently activated mutually exclusive sources.
	Note: If you combine both mutually exclusive and non-mutually exclusive source usage registrations, then be aware that you cannot reuse index number. Each source usage asset but be assigned exclusively to a single index number.
	Arguments:
	• **index** - source index position for source device
	• **sourceDevice** - device to use as a source device
	Syntax:
	`DEFINE_FUNCTION RmsSourceUsageAssignAssetMutExcl(INTEGER index, DEV sourceDevice)`
	`{`
	`  STACK_VAR CHAR commandString[RMS_MAX_CMD_LEN];`
	
	`  commandString = RmsPackCmdHeader('SOURCE.ASSIGN');`
	`  commandString = RmsPackCmdParam(commandString,ITOA(index));`
	`  commandString = RmsPackCmdParam(commandString,RmsDevToString(sourceDevice));`
	`  commandString = RmsPackCmdParam(commandString,ITOA(TRUE)); // set mutually exclusive flag`
	
	`  SEND_COMMAND vdvRMSSourceUsage, commandString;`
	`}`

RmsSourceUsage.axi - Wrapper Functions (Cont.)	

RmsSourceUsage ActivateSource	*Description*: This function is used to 'Activate' a source by it's index number. This function will activate either a mutually exclusive or non-mutually exclusive tracked source asset.
	Arguments:
	• **sourceIndex** - index of source to be activated
	Syntax:
	`DEFINE_FUNCTION RmsSourceUsageActivateSource(INTEGER index)`
	`{`
	`  STACK_VAR CHAR commandString[RMS_MAX_CMD_LEN];`
	`  commandString = RmsPackCmdHeader('SOURCE.ACTIVATE');`
	`  commandString = RmsPackCmdParam(commandString,ITOA(index));`
	`  SEND_COMMAND vdvRMSSourceUsage, commandString;`
	`}`

RmsSourceUsage DeactivateSource	*Description*: This function is used to 'Deactivate' a source by it's index number. This function will deactivate either a mutually exclusive or non-mutually exclusive tracked source asset.
	Arguments:
	• **sourceIndex** - index of source to be deactivated
	Syntax:
	`DEFINE_FUNCTION RmsSourceUsageDeactivateSource(INTEGER index)`
	`{`
	`  STACK_VAR CHAR commandString[RMS_MAX_CMD_LEN];`
	`  commandString = RmsPackCmdHeader('SOURCE.DEACTIVATE');`
	`  IF(index > 0)`
	`  {`
	`  commandString = RmsPackCmdParam(commandString,ITOA(index));`
	`  }`
	`  SEND_COMMAND vdvRMSSourceUsage, commandString;`
	`}`

RmsSourceUsage DeactivateAllSources	*Description*: This function is used to 'Deactivate' all currently active source assets. This function deactivates all mutually exclusive and non-mutually exclusive tracked source assets.
	Arguments: none
	Syntax:
	`DEFINE_FUNCTION RmsSourceUsageDeactivateAllSources()`
	`{`
	`  SEND_COMMAND vdvRMSSourceUsage, 'SOURCE.DEACTIVATE';`
	`}`

See the *Tracking Source Usage* section on page 112 for more information.

RmsSourceUsage.axi - Defined Constants

The "*RmsSourceUsage.axi*" Include File includes defined constants for the following:

- RMS API Channels
- RMS API Levels
- Frequently used RMS SEND_COMMAND headers
- RMS Metadata Property Data Types
- RMS Hotlist Record Types
- RMS Display Message Types
- RMS Asset Parameter Threshold Comparison Operators
- RMS Logging Levels
- RMS Asset Control Method Argument Data Types
- RMS Asset Parameter Data Types
- RMS Asset Parameter (Reporting) Types

The "*RmsApi.axi*" Include File includes many helper functions and custom RMS data types for registering assets, registering and updating asset metadata properties, registering asset control methods, registering and updating asset parameters, and building Send Commands for use with the RMS virtual device.

RmsApi.axi - Data Types

The following data types can be found in the *RmsApi.axi* Include File:

```
STRUCTURE RmsAsset
{
  CHAR assetType[50];
  CHAR clientKey[30];
  CHAR globalKey[150];
  CHAR name[50];
  CHAR description[250];
  CHAR manufacturerName[50];
  CHAR manufacturerUrl[250];
  CHAR modelName[50];
  CHAR modelUrl[250];
  CHAR serialNumber[100];
  CHAR firmwareVersion[30];
}

STRUCTURE RmsAssetMetadataProperty
{
  CHAR key[50];
  CHAR name[50];
  CHAR value[50];
  CHAR dataType[30];
  CHAR readOnly;
  CHAR hyperlinkName[50];
  CHAR hyperlinkUrl[100];
}

STRUCTURE RmsAssetControlMethodArgument
{
  INTEGER ordinal;
  CHAR name[50];
  CHAR description[250];
  CHAR dataType[30];
  SLONG defaultValue;
  SLONG minimumValue;
  SLONG maximumValue;
  INTEGER stepValue;
  CHAR enumerationValues[15][30];
}

STRUCTURE RmsAssetParameter
{
  CHAR key[50];
  CHAR name[50];
  CHAR description[250];
  CHAR dataType[30];
  CHAR reportingType[30];
  CHAR initialValue[50];
  CHAR units[50];
  CHAR allowReset;
  CHAR resetValue[50];
  SLONG minimumValue;
  SLONG maximumValue;
  CHAR enumeration[500];
  CHAR trackChanges;
  CHAR bargraphKey[30];
  CHAR stockParam;
}

STRUCTURE RmsAssetParameterThreshold
{
  CHAR name[50];
  CHAR enabled;
  CHAR statusType[30];
  CHAR comparisonOperator[30];
  CHAR value[50];
  CHAR notifyOnTrip;
  CHAR notifyOnRestore;
  CHAR delayInterval;
}
```

RmsApi.axi - General Utility Functions

The following list of General Utility Functions can be found in the *RmsApi.axi* Include File:

RmsApi.axi - General Utility Functions	

RmsDevToString	*Description*: This function is used to convert a device variable of type DEV to a string representation using the <D>:<P>:<S> formatting syntax.
	Arguments:
	• DEV dvDPS
	Returns: DPS string; example "5001:1:0"
	Syntax:
	`DEFINE_FUNCTION CHAR[20] RmsDevToString(DEV dvDPS)`
	`{`
	`  RETURN "ITOA(dvDPS.Number),':',ITOA(dvDPS.Port),':',ITOA(dvDPS.System)"`
	`}`

RmsDeviceIdInList	*Description*: This function is used to determine if a Device ID (number) is included in a list of devices IDs
	Arguments:
	• **INTEGER devID** - Device ID (number) to search for
	• **INTEGER devList[]** - Device ID array to search in
	Returns: Index of the device in array; 0 if not found
	Syntax:
	`DEFINE_FUNCTION INTEGER RmsDeviceIdInList(INTEGER devId, INTEGER devList[])`
	`{`
	`  STACK_VAR INTEGER index;`
	
	`  // iterate over the Device ID listing and search`
	`  // for a matching Device ID, return if found.`
	`  FOR(index = 1; index <= LENGTH_ARRAY(devList); index++)`
	`  {`
	`  IF(devList[index] == devId)`
	`  {`
	`  RETURN index;`
	`  }`
	`  }`
	`  RETURN 0;`
	`}`

RmsDeviceInList	*Description*: This function is used to determine if a device instance is included in an array of devices.
	Arguments:
	• **DEV devTarget** - target device to search for
	• **DEV devList[]** - array of devices to search in
	Returns: index of the device in array; 0 if not found
	Syntax:
	`DEFINE_FUNCTION INTEGER RmsDeviceInList(DEV devTarget, DEV devList[])`
	`{`
	`  STACK_VAR INTEGER index;`
	
	`  // NOTE: this function does not take the device SYSTEM number`
	`  //  into account when evaluating a possible match.`
	`  //  Data/Channel/Button event returned device instances`
	`  //  may return the DPS structure with the real system`
	`  //  number and the hard coded device definition only`
	`  //  specified system '0'.`
	
	`  // iterate over the device listing (array) and search`
	`  // for a matching device instance, return index if found.`
	`  FOR(index = 1; index <= LENGTH_ARRAY(devList); index++)`
	`  {`
	`  IF(devList[index].NUMBER == devTarget.NUMBER &&`
	`  devList[index].PORT == devTarget.PORT)`
	`  {`
	`  RETURN index;`
	`  }`
	`  }`
	`  RETURN 0;`
	`}`

RmsApi.axi Functions (Cont.)	
General Utility Functions (Cont.)	

RmsDeviceStringInList	*Description*: This function is used to determine if a device D:P:S string is found in and array of devices
	Arguments:
	• **CHAR devTarget[]** - target device to search for
	• **DEV devList[]** - array of devices to search in
	Returns: index of the device in array; 0 if not found
	Syntax:
	`DEFINE_FUNCTION INTEGER RmsDeviceStringInList(CHAR devTarget[], DEV devList[])`
	`{`
	`  STACK_VAR INTEGER index;`
	
	`  // iterate over the device listing (array) and search`
	`  // for a matching device instance, return index if found.`
	`  FOR(index = 1; index <= LENGTH_ARRAY(devList); index++)`
	`  {`
	`  IF(RmsDevToString(devList[index]) == devTarget)`
	`  {`
	`  RETURN index;`
	`  }`
	`  }`
	`  RETURN index;`
	`}`

RmsBooleanValue	*Description*: This function is used to parse a string and return a char (bit) for boolean state.
	Arguments:
	• **CHAR value[]** - string to convert to char (bit)
	Returns: 0 if string results in FALSE; 1 if string results in TRUE
	Syntax:
	`DEFINE_FUNCTION CHAR RmsBooleanValue(CHAR value[])`
	`{`
	`  IF(LOWER_STRING(value) == 'true' || value=='1' || LOWER_STRING(value) == 'on' )`
	`  {`
	`  RETURN TRUE;`
	`  }`
	`  ELSE`
	`  {`
	`  RETURN FALSE;`
	`  }`
	`}`

RmsBooleanString	*Description*: This function is used to convert a char (bit) to a RMS boolean string of 'true' or 'false'.
	Arguments:
	• **CHAR value** - convert bit to boolean string
	Returns: 'false' if char value is 0; 'true' if char value is >0
	Syntax:
	`DEFINE_FUNCTION CHAR[5] RmsBooleanString(CHAR value)`
	`{`
	`  IF(value)`
	`  {`
	`  RETURN 'true';`
	`  }`
	`  ELSE`
	`  {`
	`  RETURN 'false';`
	`  }`
	`}`

RmsApi.axi - Miscellaneous Functions

The following list of other Miscellaneous Functions can be found in the *RmsApi.axi* Include File:

RmsApi.axi - Miscellaneous Functions	

RmsGetVersionInfo	*Description*: This function is used to query the RMS client to display version information via the master's telnet console.
	Arguments: *none*
	Returns: 1 if call was successful; 0 if call was unsuccessful
	Syntax:
	`DEFINE_FUNCTION CHAR RmsGetVersionInfo()`
	`{`
	`  // send the version request to RMS`
	`  SEND_COMMAND vdvRMS, RMS_COMMAND_VERSION_REQUEST;`
	
	`  RETURN TRUE;`
	`}`

RmsReinitialize	*Description*: This reinitialize request will reset the RMS connection to the server and force all the asset parameter, control methods and metadata to be resent/registered with RMS server.
	Arguments: *none*
	Returns: *n/a*
	Syntax:
	`DEFINE_FUNCTION RmsReinitialize()`
	`{`
	`  // send the reinitialization request to RMS`
	`  SEND_COMMAND vdvRMS, RMS_COMMAND_REINITIALIZE;`
	`}`

RmsSystemPowerOn	*Description*: This function will set the RMS system power to the ON state. System Power event notifications will be sent out if this request causes a state change .
	Arguments: *none*
	Returns: *n/a*
	Syntax:
	`DEFINE_FUNCTION RmsSystemPowerOn()`
	`{`
	`  // set the system power to the ON state`
	`  SEND_COMMAND vdvRMS,RMS_COMMAND_SYSTEM_POWER_ON;`
	`}`

RmsSystemPowerOff	*Description*: This function will set the RMS system power to the OFF state. System Power event notifications will be sent out if this request causes a state change .
	Arguments: *none*
	Returns: *n/a*
	Syntax:
	`DEFINE_FUNCTION RmsSystemPowerOff()`
	`{`
	`  // set the system power to the OFF state`
	`  SEND_COMMAND vdvRMS,RMS_COMMAND_SYSTEM_POWER_OFF;`
	`}`

RmsSystemSetMode	*Description*: This function will apply a new System Mode by name. System Mode event notifications will be sent out if this request causes a state change
	Arguments: modeName (System Mode name - see *Implementing System Modes* on page 113)
	Returns: *n/a*
	Syntax:
	`DEFINE_FUNCTION RmsSystemSetMode(CHAR modeName[])`
	`{`
	`  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];`
	
	`  // set the system mode in RMS`
	`  rmsCommand = RmsPackCmdHeader(RMS_COMMAND_SYSTEM_MODE);`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,modeName);`
	`  SEND_COMMAND vdvRMS, rmsCommand;`
	`}`

For detailed information, please review the source code and code comments inside the *RmsApi.axi* Include File. Additionally, please see the *NetLinx Programming* sections in this document where the usage of many of these functions in the RmsApi are demonstrated:

- See *Programming - RMS Required Modules* on page 55.
- See *Programming - Asset Management* on page 57.
- See *Programming - Client Messaging* on page 109.
- See *Programming - Advanced Topics* on page 111.

RMS Event Listener

The RMS Event listener is a common Include File that is consumed by many components of the RMS Enterprise SDK. In a typical user program you will most likely never need to interact with the RMS Event listener directly, because the RMS Enterprise SDK wraps most of the necessary functionality into more convenient and abstract implementations.

However if you do require lower level access and integration with RMS, you can consume this Include File directly to listen for RMS event notifications. The RMS Event listener listens for command data events directly on the RMS NetLinx virtual device API and parses the event command data and finally invokes callback method calls to any subscribers. By default no RMS events are monitored, your user code must subscribe for each particular event of interest that you want the RMS Event Listener to invoke a callback method on.

The diagram in FIG. 14 illustrates the file dependency chain for the *RmsEventListener* Include File integration:

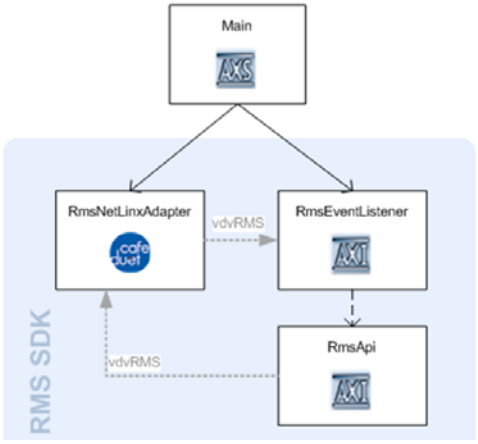


FIG. 14 RMS SDK File Dependencies - Event Listener Implementation

The code snippet below demonstrates how to subscribe for the System Power notification event.

```
// subscribe to system event notification callback methods
#DEFINE INCLUDE_RMS_EVENT_SYSTEM_POWER_CALLBACK;
// include RmsEventListener (which also includes RMS API)
#INCLUDE 'RmsEventListener';
```

If an event subscription compiler directive is included, then the user code must implement the required callback method so the RMS Event Listener can perform the callback invocation. The following code snippet illustrates the required method that must be included in the user code.

```
// include callback method for RmsEventListener to call
DEFINE_FUNCTION RmsEventSystemPowerChanged(CHAR powerOn)
{
}
```

A complete listing of eligible RMS notification events and their respective callback method signatures are included in the comments inside the RMSEventListener.axi file.

NOTE: *If you are already using the RmsSystemEventHandler Include File, it already defines the include definition for the RmsEventListener Include File and it already subscribes to both the system power and system mode RMS event notification callback method. It is not possible to register two callback methods in a single compiled NetLinx program, except for inside NetLinx modules which are independently compiled as separate programs outside the scope of the main NetLinx program.*

RMS Duet Device Monitoring Modules

The RMS Enterprise SDK includes default asset monitoring and management implementations for the following Duet device types:

RMS Duet Device Monitoring Modules	
• Audio Conferencer	• Monitor
• Camera	• Receiver
• Digital Satellite System	• Settop Box
• Document Camera	• Switcher
• Digital Video Recorder	• TV
• Disc Device (DVD, CD)	• Video Conferencer
• Light System	• Video Projector

Each Duet device that is intended to be used with RMS should include both the Duet device module and a RMS Duet device monitoring module. The Duet device monitoring modules are lightweight wrapper NetLinx-based modules that facilitate device registration and provide extension points for NetLinx programmers to add any necessary custom asset parameters, control methods, or metadata properties.

The diagram in FIG. 15 illustrates the file dependency chain for the *RmsDuetXxxxxMonitor* modules integration:

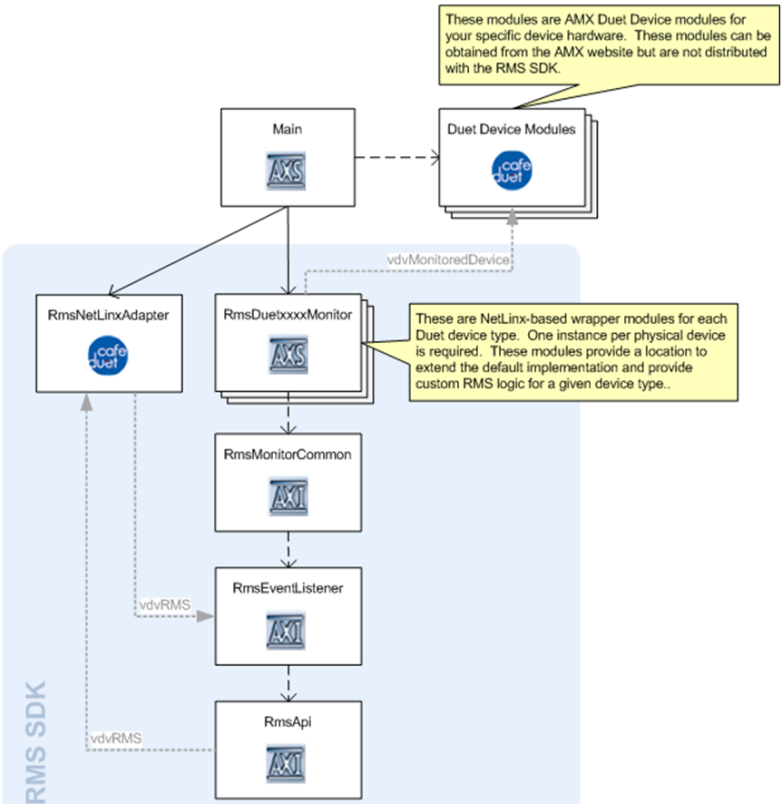


FIG. 15 RMS SDK File Dependencies - Duet Device Implementation

Each Duet device monitoring module requires the following three arguments (a) the RMS Virtual Device, (b) the device for the real device, and (c) the Duet module virtual device. These RMS Duet device monitor modules are provided as open source NetLinx code.

(a) RMS NetLinx Virtual Device - This virtual device must be exposed using the RMS NetLinx Adapter module.

(b) Duet Module Virtual Device - This device will be the NetLinx virtual Device ID used to communicate with the RMS subsystem from NetLinx programming. (Note this Device ID must exists in the Duet Device ID range.)

(c) Real Device - This is the physical device I/O port and is used by RMS to associate a unique client key to the asset using the D:P:S identification string.

The following code snippet illustrates a defined Duet module and its corresponding RMS Duet device monitoring module.

```
DEFINE_DEVICE

dvVideoProjector  =  5001:23:0   // Video Projector (Physical Device)
vdvVideoProjector =  41116:1:0   // Duet Video Projector Device

DEFINE_START
// Video Projector Duet Device Module
DEFINE_MODULE 'BarcoVideoProjector_dr1_0_0' mdlVideoProjector(vdvVideoProjector,dvVideoProjector)
// RMS Video Projector Device Monitor
DEFINE_MODULE 'RmsDuetVideoProjectorMonitor' mdlRmsVideoProjectorMon(vdvRMS,vdvVideoProjector,
                                                                     dvVideoProjector)
```

The sample code above is all that is needed to implement the default device monitoring and management with RMS.

Implementing Driver Design (*.XDD) Modules

The following code snippet illustrates a defined XDD module and its corresponding RMS XDD device monitoring module.

```
DEFINE_DEVICE
dvVideoProjector  = 5001:3:0   // Video Projector (Physical Device)
vdvVideoProjector = 41003:1:1  // Duet Video Projector Device

DEFINE_VARIABLE
CHAR myProjectorXDDFile[] = 'VideoProjector_xdd_name.xdd'

DEFINE_START
// Video Projector Duet Device Module
DEFINE_MODULE 'DeviceDriverEngine' mdlVideoProjectorXDD(vdvVideoProjector,dvVideoProjector,myProjectorXDDFile)
// RMS Video Projector Device Monitor
DEFINE_MODULE 'RmsDuetVideoProjectorMonitor' mdlRmsVideoProjectorMon(vdvRMS,vdvVideoProjector,vVideoProjector)
```

The sample code above is all that is needed to implement the default device monitoring and management with RMS.

For more detailed information, review the source code provided for each *RmsDuetXxxxxxMonitor* module. Additionally, see the *Programming - Asset Management* section on page 57 for more details on programming for asset management.

RMS NetLinx Device Monitoring Modules

The RMS Enterprise SDK includes default asset monitoring and management NetLinx code based implementations for the following device types:

RMS NetLinx Device Monitoring Modules	
• Audio Conferencer	• Receiver
• Camera	• Security System
• Digital Satellite System	• Settop Box
• Document Camera	• Switcher
• Digital Video Recorder	• TV
• Disc Device (DVD, CD)	• Video Conferencer
• HVAC	• Video Projector
• Monitor	

Unlike the lightweight wrapper RMS Duet device monitor modules, these NetLinx device monitor modules provide the full implementation of the device monitoring and management logic in NetLinx code. These modules can be used, extended, and/or modified when integrating a control system solution that do not use Duet based device modules.

These RMS NetLinx device monitoring modules do provide the default implementation for monitoring device activity and reporting the updates to RMS; however, these modules require some form of input from the user's device implementation code to trigger the updated. These RMS NetLinx device monitoring modules use the SNAPI (Standard NetLinx API) protocols on the NetLinx device virtual devices to monitor for changes.

This does not mean that you have to implement Duet-based devices, however the NetLinx programmer is responsible for emulating the SNAPI channel/level/command API for a particular device. For example, to allow the RMS NetLinx monitoring module to track a device port, the NetLinx programmer must implement the SNAPI status channel (#255) for device power.

When the physical device power is turned on, the NetLinx programmer should set channel #255 to the ON state. The RMS NetLinx device monitor module will listen for the channel change and update RMS accordingly.

In addition to the SNAPI emulation required by the NetLinx programmer, it should also be noted that each RMS NetLinx Device Monitor module includes a set of #DEFINE HAS_XXXXX compiler directives.

These "HAS" definitional allow you to compile the device monitor modules with specific sets of functionality included or excluded depending on if your physical device supports the features. See the *RMS NetLinx Monitoring Module HAS_PROPERTY Compiler Directives* section on page 119 for details.

The diagram in FIG. 16 on page 49 illustrates the file dependency chain for the *RmsNlXxxxxMonitor* modules integration:

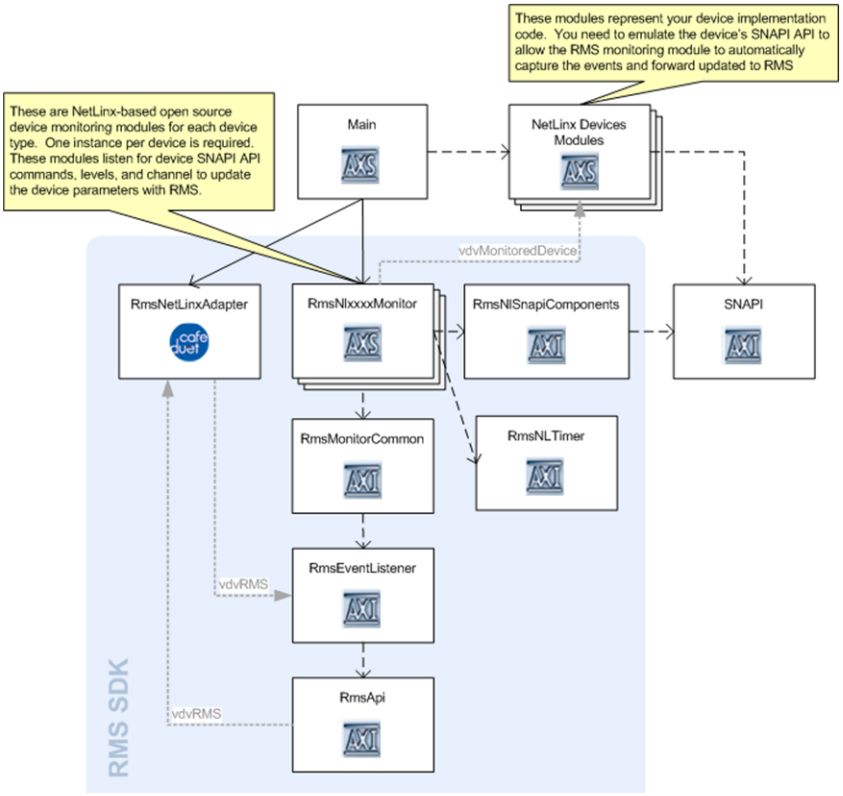


FIG. 16 RMS SDK File Dependencies - NetLinx Device Implementation

Each NetLinx device monitoring module requires the following three arguments (a) the RMS NetLinx Virtual Device, (b) a NetLinx virtual device dedicated to this module, and (c) the device for the real device. These RMS NetLinx device monitor modules are provided as open source NetLinx code.

(a) RMS NetLinx Virtual Device - This virtual device must be exposed using the RMS NetLinx Adapter module.

(b) NetLinx Module Virtual Device - This device will be the NetLinx virtual Device ID used to communicate with the RMS subsystem from NetLinx programming. Note this Device ID must be virtual and must not exist in the Duet Device ID range.

(c) Real Device - This is the physical device I/O port and is used by RMS to associate a unique client key to the asset using the D:P:S identification string.

The following code snippet illustrates a RMS NetLinx device monitoring module:

```
DEFINE_DEVICE

dvVideoProjector    =  5001:23:0  // Video Projector (Physical Device)
vdvVideoProjector   =  34001:1:0  // Video Projector Virtual Device
DEFINE_START
// RMS Video Projector Device Monitor
DEFINE_MODULE 'RmsNlVideoProjectorMonitor' mdlRmsVideoProjectorMon(vdvRMS,
                                                                   vdvVideoProjector,
                                                                   dvVideoProjector)
```

For more detailed information, review the source code provided for each *RmsNlXxxxxxMonitor* module. Additionally, see the *Programming - Asset Management* section on page 57 for more details on programming for asset management.

Sample Source & Workspace Files

The RMS Enterprise Software Development Kit provides a number of source code examples both for the Duet-based implementation and the NetLinx-based implementation.

These examples can be found in the SDK installation directory and are also included in each of the two sample workspaces.

A "Main-Duet" and "Main-NetLinx" sample files are also provided that are fully inclusive of all of the implementation samples fully integrated into a single source project.

Getting Started

Source Code Samples

The RMS Enterprise Software Development Kit (SDK) provides a number of source code examples both for the Duet-based implementation and the NetLinx-based implementation. These examples can be found in the SDK installation directory and are also included in each of the two sample workspaces. These samples include the following concepts:

- Single Monitored Device Implementation
- Single Monitored Device + Single Touch Panel & Client GUI Implementation
- Multiple Monitored Devices + Multiple Touch Panels & Client GUI Implementation
- Power Distribution Unit Implementation
- Source Usage Tracking Implementation
- System Power + System Mode Implementation

"Main-Duet" and "Main-NetLinx" sample files are also provided that are fully inclusive of all of the implementation samples fully integrated into a single source project.

NetLinx or Duet Path?

When starting a RMS Enterprise project, one of the first things a control system programmer must determine is if they are going to use Duet based device modules or if they are going to completely code all the device implementation in NetLinx.

Duet Device Implementation

If using Duet device modules, the RMS SDK implementation can be much simpler in that it can automatically communicate with the Duet modules to determine device capabilities, device state, and listen for device change events. No custom NetLinx programming is required to integrate the default device monitoring and control behavior with RMS. All that is needed is a lightweight wrapper module defined alongside each Duet device definition.

Choosing the Duet route in RMS does not mean that the device monitoring and control logic is 100% predetermined and fixed. A NetLinx programmer can override and/or extend the default implementation logic directly in the RMS Duet Monitor wrapper module. There are callback methods included in the wrapper modules that provide the opportunity to register additional parameters, metadata properties, and/or control methods.

Refer to the *RMS Duet Device Monitoring Modules* section on page 46 section for more information.

NetLinx Device Implementation

If not using Duet device modules, but rather implementing device functionality directly in NetLinx code or in NetLinx modules, then the RMS SDK offers the RMS NetLinx Device Monitoring modules which provide the default implementation logic for device monitoring and control with RMS. These monitoring modules are written completely in NetLinx code and are provided as open source modules with the RMS SDK.

Implementing the RMS NetLinx Device Monitoring modules requires some additional programming effort on the part of the NetLinx programmer. The RMS NetLinx Device Monitoring modules are programmed to provide a default implementation of device monitoring and control but require some input to know when parameters or state has changed on a physical device.

The RMS NetLinx Device Monitoring modules use SNAPI as a convention for listening for these type of notifications. Therefore, the NetLinx programmer must emulate the SNAPI communication for a particular device type to notify the RMS NetLinx Device Monitoring module of any change on the physical device. Since these are fully open source, a programmer can make a custom copy of the module and implement their own custom logic or extended behaviors as needed. There are callback methods included in the monitor modules that provide the opportunity to register additional parameters, metadata properties, and/or control methods.

Refer to the *RMS Duet Device Monitoring Modules* section on page 46 section for more information.

Understanding the RMS Client Connection Lifecycle

When the RMS client is loaded on the NetLinx platform, it follows a strict connection lifecycle. It may be helpful as a NetLinx programmer to understand the connection states and how the RMS client transitions through the connection lifecycle. The RMS Client contains the following connection states (FIG. 17):

- INIT
- DISABLED
- OFFLINE
- CONNECT-SERVER
- CONNECT-FAIL
- CONNECT-CLIENT
- CONNECT-LOCATION
- ONLINE-UNREGISTERED
- ONLINE
- REINITIALIZE

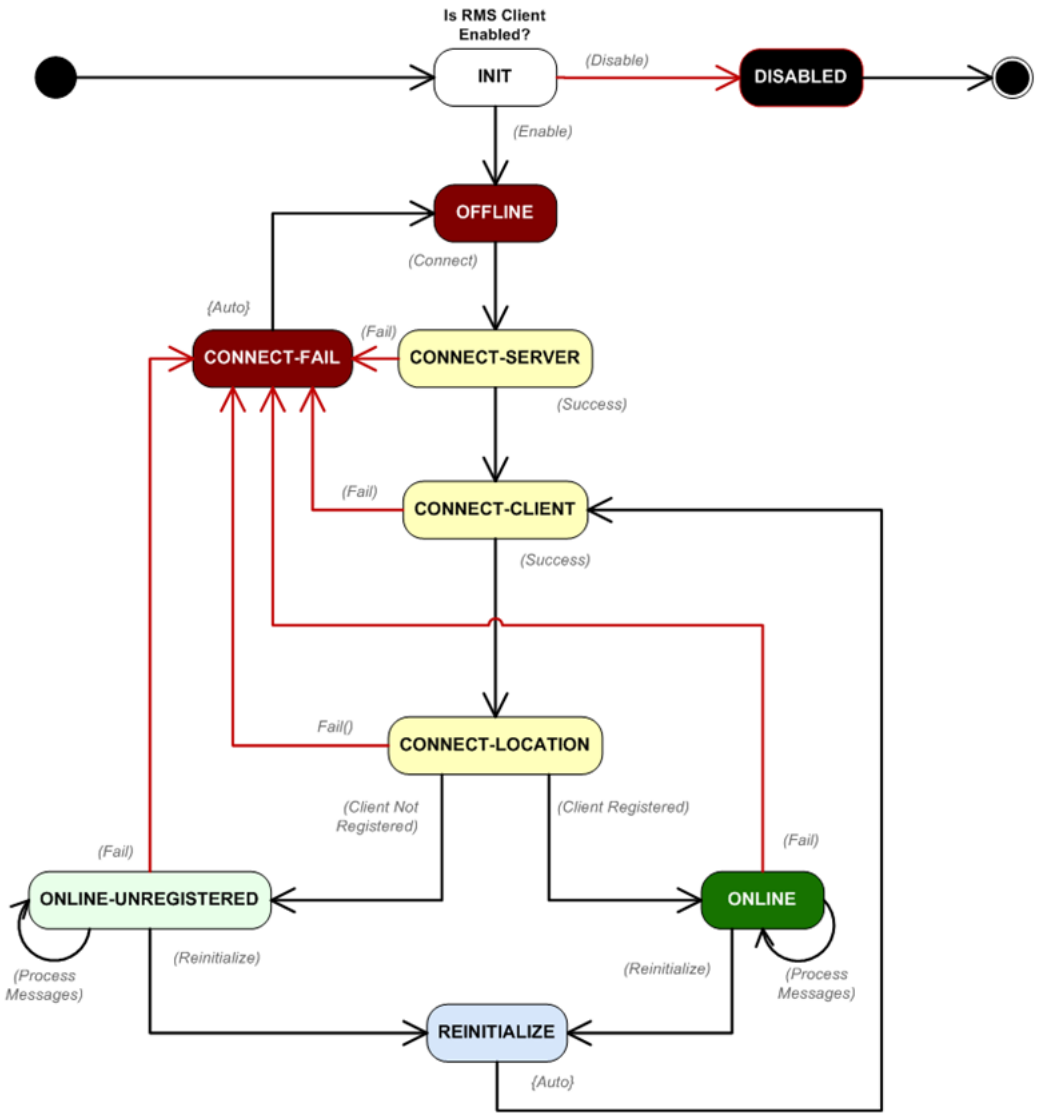


FIG. 17 Client Connection State Machine

INIT

When the NetLinx system starts and loads all modules and the user program, the RMS Client starts in the INIT state. Upon entering this state, the RMS Client looks at the RMS configuration settings to determine if the RMS Client is enabled and contains valid RMS server path settings.

- If the RMS Client configuration is disabled or if any of the RMS server path settings are missing, the RMS Client will transition to the DISABLED state.
- If the RMS Client configuration is enabled and valid RMS server path settings are present, then the RMS Client will transition into the OFFLINE state.
- When the RMS client configuration settings are modified and saved, the RMS client will re-enter the INIT connection state.

DISABLED

Upon entering this state, the RMS Client will remain idle and not perform any connection attempts to the RMS server.

If the RMS client configuration settings are modified and saved, the RMS client will re-enter the INIT connection state.

OFFLINE

After the RMS Client enters the OFFLINE state, it will start a connection delay timer. After the elapsed connection delay time (default is 30 seconds), the RMS Client will transition into the CONNECT-SERVER connection state to attempt a connection with the RMS server.

If the connection attempt fails, the RMS Client connection state will return to this OFFLINE state and wait for the next connection delay time interval to re-attempt connection to the RMS server. There are two connection delay time intervals that are used in the OFFLINE state.

The first delay time is known as GREEDY and the second is known as SETBACK. By default the OFFLINE state will use the GREEDY time delay which is by default 30 seconds to attempt connections to the RMS server. If the RMS client is not able to successfully

connect to the RMS server after a configured number of retry attempts (10 attempts by default) then the OFFLINE state will switch to using the SETBACK delay time interval (5 minutes by default) between connection attempts to the RMS server.

The GREEDY versus SETBACK delay times provide a balance between client re-connection attempt for some short interruption versus some longer term outage.

CONNECT-SERVER

When the RMS Client enters the CONNECT-SERVER state, it will attempt a HTTP GET REST call to the RMS server to query for the RMS server information data.

If the connection is successful, the RMS Client will transition into the CONNECT-CLIENT state. Else, if the connection attempt fails to connect or fails to successfully get the RMS server information data, the RMS Client will transition into the CONNECT-FAIL state.

CONNECT-FAIL

When the RMS Client enters the CONNECT-FAIL state, it will increment an internal tracking count of the number of successive connection failure attempts and transition to the OFFLINE state to wait out the appropriate connection attempt delay interval and then retry the connection again.

CONNECT-CLIENT

When the RMS Client enters the CONNECT-CLIENT state, it will attempt a HTTP PUT REST request to the RMS server to register the RMS Client Gateway information with the RMS system.

If the request is successful, the RMS Client will transition into the CONNECT-LOCATION state. Else, if the request attempt fails, the RMS Client will transition into the CONNECT-FAIL state.

CONNECT-LOCATION

When the RMS Client enters the CONNECT-LOCATION state, it will attempt a HTTP GET REST request to the RMS server to obtain the location identity associated with this RMS Client Gateway in the RMS system.

If the request successfully returns location information, the RMS Client will transition into the ONLINE state.

If the request results in a server response of 404, meaning that no location is defined for this RMS Client Gateway, the RMS Client will transition into the ONLINE-UNREGISTERED state. Else, if the request attempt fails for some other reason, the RMS Client will transition into the CONNECT-FAIL state.

ONLINE-UNREGISTERED

When the RMS Client enters the ONLINE-UNREGISTERED state, it means that the RMS Client Gateway has successfully been registered with the RMS system, but the Client Gateway has not been approved/accepted by a RMS administrator and assigned to a location in the RMS system. In this state, the RMS Client will not perform any asset management or any other functions. It will only poll the RMS server for client pending messages and will only take action on a REINITIALIZE message received from the RMS server.

Upon receiving a REINITIALIZE message, the RMS Client will transition to the REINITIALIZE state. A REINITIALIZE message is automatically queued for the RMS client when a RMS administrator approves and assigns the Client Gateway to a location in the RMS system.

Refer to the *Accepting RMS Client Gateway in RMS Web User Interface* section on page 53 topic for details on how to approve and assign a new RMS Client Gateway. If at any point in the client message polling with the RMS server, if a connection cannot be established with the RMS server, then the RMS Client will transition to the CONNECT-FAIL state.

REINITIALIZE

When the RMS Client enters the REINITIALIZE state, it will restart an immediate connection attempt to the RMS server by transitioning directly to the CONNECT-CLIENT state.

ONLINE

When the RMS Client enters the ONLINE state, it means that the RMS Client Gateway has successfully been registered with the RMS system, has been licensed and approved by a RMS administrator, and assigned to a location in the RMS system. At this point the RMS Client will begin asset management and monitoring registration and synchronization and begin polling the RMS server on regular intervals for any client pending messages from the RMS server. In this connection state, the RMS Client is functional.

Upon receiving a REINITIALIZE message, the RMS Client will transition to the REINITIALIZE state. If at any point in the client message polling with the RMS server, if a connection cannot be established with the RMS server, then the RMS Client will transition to the CONNECT-FAIL state.

Programming the RMS Client

A RMS integrated program must be compiled and deployed to a NetLinx master before the system can participate in the RMS system.

Configure the RMS Client

Before the RMS Client can connect to a RMS server, the RMS Client must be enabled and configured with the server path URL and access credentials.

The easiest way to configure the RMS Client is to access the NetLinx system's web configuration pages and use the RMS Client Web Configuration page to enable RMS and apply the proper settings. See the *RMS Client Web Configuration* section on page 28 for more information.

An alternate method to establish the RMS configuration is to programmatically define the connection settings in NetLinx code. See the *Programmatically Setting the Client Configuration* section on page 114 for more information.

Accepting RMS Client Gateway in RMS Web User Interface

After the RMS Client has been programmed and configured to access the RMS server, a RMS administrator must approve the RMS Client gateway's participation in the RMS system and assign the Client Gateway to a location in the RMS system. This process takes place in the RMS Application Web User Interface.

All newly registered Client Gateway's will appear on the RMS Hotlist (FIG. 18).

FIG. 18 RMS Hotlist - Newly Registered Client Gateway

Unregistered Client Gateways can also be accessed under Manage / Client Gateways from the main menu. Unregistered clients will appear in the “Unassigned Client Gateways” group container (FIG. 19).

FIG. 19 RMS Client Gateway Management page - Unregistered Client Gateways

From the Hotlist or from the Client Gateway management page, an administrator can “Assign” the client gateway to an existing or new location (FIG. 20).

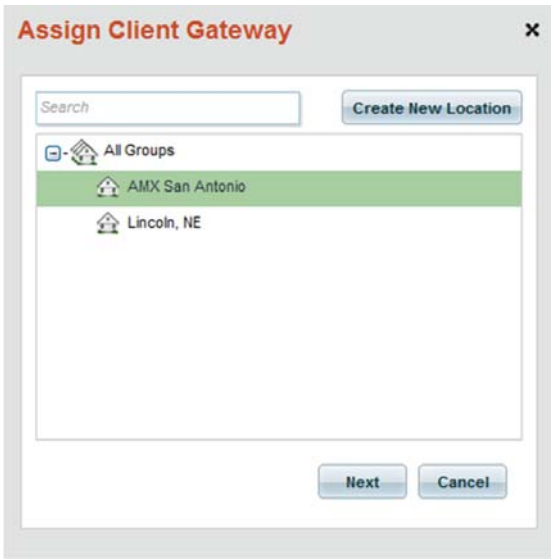


FIG. 20 Assign Client Gateway

- If assigning to an existing location you will also be prompted with the opportunity to replace an existing client gateway if this new client were a replacement for a defective control system master.
- Otherwise, select *Assign Client Gateway* to simply add this as a new client in this location (FIG. 21).

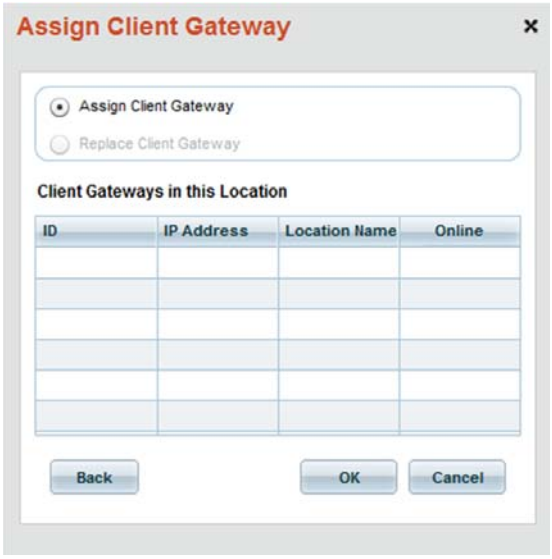


FIG. 21 Assign Client Gateway

If you select *Create a New Location*, you will be prompted with a location entry dialog to enter the new location information and licensing.

After completing your Client Gateway assignment, the RMS Client will be re-initialized. It may take a few minutes for it to come back online and begin registering assets with the RMS system.

Programming - RMS Required Modules

Overview

To implement the RMS Client there are a couple of modules that are required to be implemented in the NetLinx user program. Example snippets are listed below but it is suggested that you read each related help topic for each RMS SDK component in full detail to get a greater understanding of the purpose and usage for each component.

The RMS Client NetLinx Adapter Module is required to load the RMS Client into the NetLinx master and provide a NetLinx virtual device to communicate with the RMS Client. The following code snippets can be included to load the RMS NetLinx Adapter module.

This device will be the NetLinx virtual Device ID used to communicate with the RMS subsystem from NetLinx programming. (Note this Device ID must exists in the Duet Device ID range.)

```
DEFINE_DEVICE

vdvRMS            =  41001:1:0      // RMS Client (Duet Module)
vdvRMSGUI         =  41002:1:0      // RMS Client GUI (Duet Module)

DEFINE_START
// RMS Client - NetLinx Adapter Module
//   This module includes the RMS client module
//   and enables communication via SEND_COMMAND,
//   SEND_STRINGS, CHANNELS, and LEVELS with the
//   RMS Client.
DEFINE_MODULE 'RmsNetLinxAdapter_dr4_0_0' mdlRMSNetLinx(vdvRMS)
```

If you are incorporating the RMS Touch Panel user interfaces into your system, you will also need to include the RMS Client Application User Interface module and additionally if you want the Touch Panels to show up as assets in the RMS system, you will need an instance of the RMS Touch Panel Monitor module for each touch panel device. The following code snippets can be included to load the client user interface and related touch panel monitoring modules.

```
DEFINE_DEVICE

dvTP1               =  10001:1:0   // Touch Panels
dvTP2               =  10002:1:0   //  (must be port 1 for base device)
dvTP3               =  10003:1:0
dvTP1_RMS           =  10001:7:0   // RMS Touch Panels
dvTP2_RMS           =  10002:7:0   //  (RMS uses port 7 by default)
dvTP3_RMS           =  10003:7:0

DEFINE_VARIABLE
// RMS Touch Panel Array
VOLATILE DEV dvRMSTP[] =
{
   dvTP1_RMS,
   dvTP2_RMS,
   dvTP3_RMS
}

// RMS Touch Panel Array -
//  Base Device for System Keyboard handling
VOLATILE DEV dvRMSTP_Base[] =
{
   dvTP1,
   dvTP2,
   dvTP3
}
DEFINE_START
// RMS GUI - User Interface Module
//   This module is responsible for all the RMS
//   user interface application logic. This
//   includes help requests, maintenance requests,
//   location hotlist, and server display messages.

DEFINE_MODULE 'RmsClientGui_dr4_0_0' mdlRMSGUI(vdvRMSGui,dvRMSTP,dvRMSTP_Base);
//
// AMX Touch Panel Monitor
//
//  - include a touch panel monitoring module instance for each
//    touch panel device you wish to monitor.
//
DEFINE_MODULE 'RmsTouchPanelMonitor' mdlRmsTouchPanelMonitorMod_1(vdvRMS,dvTP1);
DEFINE_MODULE 'RmsTouchPanelMonitor' mdlRmsTouchPanelMonitorMod_2(vdvRMS,dvTP2);
DEFINE_MODULE 'RmsTouchPanelMonitor' mdlRmsTouchPanelMonitorMod_3(vdvRMS,dvTP3);
```

It is recommended to include a single instance of the RMS Control System Monitor module to register the NetLinx control system platform as an asset with RMS as well. The following code snippets can be included to load the client user interface and related touch panel monitoring modules.

```
//
// AMX Control System Master
//
// - include only one of these control system device monitoring modules.
//   this is intended to serve as an extension point for creating
//   system wide control methods and system level monitoring parameters.
//
DEFINE_MODULE 'RmsControlSystemMonitor' mdlRmsControlSystemMonitorMod(vdvRMS,dvMaster);
```

Finally you will need to integrate either Duet device RMS monitoring modules or NetLinx device RMS monitoring modules or a combination of both. There are many additional features and components that you can optionally integrate with RMS such as source usage, RFID asset tracking, etc.

See the *The RMS Enterprise SDK (v4)* section on page 20 for more information.

Programming - Asset Management

Asset Monitoring Modules

All RMS asset monitoring modules in the RMS SDK implement the *RmsMonitorCommon.axi* Include File.

The *RmsMonitorCommon.axi* Include File implements the underlying basic requirements and workflow required for asset management. Usage of this Include File simplifies the implementation requirements inside each asset monitoring module and abstracts most of the more complicated RMS logic allowing the monitoring module to focus solely on its own asset monitoring and management implementation without knowledge of other assets or other RMS events.

The *RmsMonitorCommon.axi* Include File provides the appropriate callback methods on the asset monitoring modules at the appropriate times when information is needed by RMS or when asset related information is evented from RMS.

RmsMonitorCommon.axi will invoke these following callback methods that must be implemented in each asset monitoring module:

- **RegisterAssets** - see *Registering Assets* on page 57
- **RegisterAssetParameters** - see *Registering Asset Parameters* on page 63
- **SynchronizeAssetParameters** - see *Synchronizing Asset Parameter Values* on page 81
- ResetAssetParameterValue
- **RegisterAssetMetadata** - see *Registering Asset Metadata Properties* on page 82
- **SynchronizeAssetMetadata** - see *Synchronizing Asset Metadata Properties* on page 89
- **RegisterAssetControlMethods** - see *Registering Asset Control Methods* on page 93
- **ExecuteAssetControlMethod** - see *Executing Asset Control Functions* on page 106
- SystemPowerChanged
- SystemModeChanged

Registering Assets

The following callback method will be invoked in each asset monitoring module when it is time to perform the Asset registration with RMS:

```
(*********************************************************)
(* Name:  RegisterAsset                                  *)
(* Args:  RmsAsset asset data object to be registered .  *)
(*                                                       *)
(* Desc:  This is a callback method that is invoked by   *)
(*        RMS to notify this module that it is time to   *)
(*        register this asset.                           *)
(*                                                       *)
(*        This method should not be invoked/called       *)
(*        by any user implementation code.               *)
(*********************************************************)
DEFINE_FUNCTION RegisterAsset(RmsAsset asset)
{
}
```

The *RmsApi* Include File (which is already included by each asset monitoring module) provides a few asset registration methods to use:

- **RmsAssetRegister** (see page 59)
- **RmsAssetRegisterAmxDevice** (see page 60)
- **RmsAssetRegisterDuetDevice** (see page 61)
- **RmsAssetRegistrationSubmit** (see page 62)

Asset Registration - Required & Optional Information

Each asset registration uses the following information.

Asset Registration - Required & Optional Information	
Asset Type (String) [REQUIRED]	This is a unique identifier for an asset type that categorizes this asset as a specific asset type. • All the default RMS provided asset types are defined in the *RmsApi.axi* Include File as constants. • RMS Enterprise also supports custom user defined asset types.
Client Key (String) [REQUIRED] << AUTOPOPULATED BY RMS SDK	This is a unique identifier for this asset. This id string must be uniquely scoped for this asset on the control system master. This asset client key is typically the D:P:S string for the physical device. • The RMS SDK will auto populate this field when using one of the registration methods exposed by the RmsApi.axi Include File. • All asset parameters, metadata properties, and control methods will use this asset key for registration and notification callbacks.

Asset Registration - Required & Optional Information (Cont.)	
Global Key (String) [OPTIONAL]	This is a globally unique identifier for this asset. This ID string must be globally unique across the entire RMS system. • This field is optional and typically left empty unless some address such as a MAC address can guarantee global uniqueness. • This global key may be used in scenarios to help identify when asset have been relocated to new controls systems.
Name (String) [REQUIRED]	Friendly name for this asset/device.
Description (String) [OPTIONAL]	Descriptive text for this asset/device.
Manufacturer Name (String) [OPTIONAL]	Descriptive text identifying the manufacturer for this asset/device. If the manufacturer name is not already defined in the RMS database, it will be added.
Manufacturer URL (String) [OPTIONAL]	URL address for the manufacturer for this asset/device.
Model Name (String) [OPTIONAL]	Descriptive text identifying the model name/id for this asset/device. If the model name is not already defined in the RMS database, it will be added and associated to the manufacturer.
Model URL (String) [OPTIONAL]	URL address for the specific model of this asset/device.
Serial Number (String) [OPTIONAL]	Serial number string for this asset/device.
Firmware Version (String) [OPTIONAL]	Firmware version information for this asset/device.

NOTE: *Duet device asset registrations will interrogate the Duet module for most of this information so there is no need to provide these details for Duet asset registrations. You can optionally override these values for Duet devices by setting the data in the asset registration call or by applying specific RMS Duet Module Properties to the Duet device module instances.*

Asset Types

The default RMS asset types are defined as constants in the *RmsApi.axi* Include File. Users can create new custom asset types in the RMS Web User Interface and custom assign programming keys.

```
// RMS Asset Types
CHAR RMS_ASSET_TYPE_DEVICE_CONTROLLER[]               = 'DeviceController'
CHAR RMS_ASSET_TYPE_TOUCH_PANEL[]                     = 'TouchPanel'
CHAR RMS_ASSET_TYPE_CONTROL_SYSTEM[]                  = 'ControlSystem'
CHAR RMS_ASSET_TYPE_SOURCE_USAGE[]                    = 'SourceUsage'
CHAR RMS_ASSET_TYPE_UNKNOWN[]                         = 'Unknown'
CHAR RMS_ASSET_TYPE_AUDIO_CONFERENCER[]               = 'AudioConferencer'
CHAR RMS_ASSET_TYPE_AUDIO_MIXER[]                     = 'AudioMixer'
CHAR RMS_ASSET_TYPE_AUDIO_PROCESSOR[]                 = 'AudioProcessor'
CHAR RMS_ASSET_TYPE_AUDIO_TAPE[]                      = 'AudioTape'
CHAR RMS_ASSET_TYPE_AUDIO_TUNER_DEVICE[]              = 'AudioTunerDevice'
CHAR RMS_ASSET_TYPE_CAMERA[]                          = 'Camera'
CHAR RMS_ASSET_TYPE_DIGITAL_MEDIA_DECODER[]           = 'DigitalMediaDecoder'
CHAR RMS_ASSET_TYPE_DIGITAL_MEDIA_ENCODER[]           = 'DigitalMediaEncoder'
CHAR RMS_ASSET_TYPE_DIGITAL_MEDIA_SERVER[]            = 'DigitalMediaServer'
CHAR RMS_ASSET_TYPE_DIGITAL_SATELLITE_SYSTEM[]        = 'DigitalSatelliteSystem'
CHAR RMS_ASSET_TYPE_DIGITAL_VIDEO_RECORDER[]          = 'DigitalVideoRecorder'
CHAR RMS_ASSET_TYPE_DISC_DEVICE[]                     = 'DiscDevice'
CHAR RMS_ASSET_TYPE_DOCUMENT_CAMERA[]                 = 'DocumentCamera'
CHAR RMS_ASSET_TYPE_HVAC[]                            = 'HVAC'
CHAR RMS_ASSET_TYPE_KEYPAD[]                          = 'Keypad'
CHAR RMS_ASSET_TYPE_LIGHT[]                           = 'Light'
CHAR RMS_ASSET_TYPE_MONITOR[]                         = 'Monitor'
CHAR RMS_ASSET_TYPE_MOTOR[]                           = 'Motor'
CHAR RMS_ASSET_TYPE_MULTI_WINDOW[]                    = 'MultiWindow'
CHAR RMS_ASSET_TYPE_POOL_SPA[]                        = 'PoolSpa'
CHAR RMS_ASSET_TYPE_PRE_AMP_SURROUND_SOUND_PROCESSOR[]='PreAmpSurroundSoundProcessor'
CHAR RMS_ASSET_TYPE_RECEIVER[]                        = 'Receiver'
CHAR RMS_ASSET_TYPE_SECURITY_SYSTEM[]                 = 'SecuritySystem'
CHAR RMS_ASSET_TYPE_SENSOR_DEVICE[]                   = 'SensorDevice'
CHAR RMS_ASSET_TYPE_SETTOP_BOX[]                      = 'SettopBox'
CHAR RMS_ASSET_TYPE_SLIDE_PROJECTOR[]                 = 'SlideProjector'
CHAR RMS_ASSET_TYPE_SWITCHER[]                        = 'Switcher'
CHAR RMS_ASSET_TYPE_TEXT_KEYPAD[]                     = 'TextKeypad'
CHAR RMS_ASSET_TYPE_TV[]                              = 'TV'
CHAR RMS_ASSET_TYPE_UTILITY[]                         = 'Utility'
```

```
CHAR RMS_ASSET_TYPE_VCR[]                           = 'VCR'
CHAR RMS_ASSET_TYPE_VIDEO_CONFERENCER[]             = 'VideoConferencer'
CHAR RMS_ASSET_TYPE_VIDEO_PROCESSOR[]               = 'VideoProcessor'
CHAR RMS_ASSET_TYPE_VIDEO_PROJECTOR[]               = 'VideoProjector'
CHAR RMS_ASSET_TYPE_VIDEO_WALL[]                    = 'VideoWall'
CHAR RMS_ASSET_TYPE_VOLUME_CONTROLLER[]             = 'VolumeController'
CHAR RMS_ASSET_TYPE_WEATHER[]                       = 'Weather'
CHAR RMS_ASSET_TYPE_AMPLIFIER[]                     = 'Amplifier'
CHAR RMS_ASSET_TYPE_IO_DEVICE[]                     = 'IODevice'
CHAR RMS_ASSET_TYPE_RELAY_DEVICE[]                  = 'RelayDevice'
CHAR RMS_ASSET_TYPE_UPS[]                           = 'UPS'
CHAR RMS_ASSET_TYPE_LIGHTSYSTEM[]                   = 'LightSystem'
CHAR RMS_ASSET_TYPE_RFIDSYSTEM[]                    = 'RFIDSystem'
CHAR RMS_ASSET_TYPE_DISPLAY[]                       = 'Display';
```

Asset Registration Functions

The *Asset Registration* functions in the *RmsApi.axi* Include File are described in the following table:

Asset Registration Functions	

RmsAssetRegister	*Description*: This function is used to register a new asset with the RMS system. This method is the general use asset registration method. This method should be used for custom implemented NetLinx controlled devices and any asset that is not AMX hardware.
	Arguments:
	• **DEV device** - asset physical device instance
	• **RmsAsset asset** - asset configuration to register
	Returns: 1 if asset registration call was successful; 0 if asset registration call was unsuccessful
	Syntax:
	`DEFINE_FUNCTION CHAR RmsAssetRegister(DEV device, RmsAsset asset)`
	`{`
	`  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];`
	
	`  // ensure RMS is ONLINE, REGISTERED, and ready for ASSET registration`
	`  IF(![vdvRMS,RMS_CHANNEL_ASSETS_REGISTER])`
	`  {`
	`  SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetRegister> :: RMS is not ready to accept asset management registrations.';`
	`  RETURN FALSE;`
	`  }`
	
	`  // the asset client key is the DPS string for NetLinx-based devices`
	`  IF(asset.clientKey == '')`
	`  {`
	`  asset.clientKey = RmsDevToString(device);`
	`  }`
	
	`  // create asset registration send command`
	`  rmsCommand = RmsPackCmdHeader('ASSET.REGISTER.DEV');`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,RmsDevToString(device));`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,asset.name);`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,asset.clientKey);`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,asset.assetType);`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,asset.globalKey);`
	
	`  // add device to registration queue`
	`  SEND_COMMAND vdvRMS, rmsCommand;`
	
	`  // now that the asset has been created in the asset registration queue, finalize and submit the asset`
	`  RETURN RmsAssetRegistrationSubmit(asset);`
	`}`

Asset Registration Functions (Cont.)

RmsAssetRegisterAmxDevice

Description: This method is the used exclusively to perform asset registration for AMX hardware based devices. This method should only be used when registering a AMX hardware based device. The RMS Client will interrogate the NetLinx device information data structure from the NetLinx device manager and include the NetLinx device information with the asset registration.

Arguments:

- **DEV device** - asset physical device instance
- **RmsAsset asset** - asset configuration to register

Returns:

1 if asset registration call was successful; 0 if asset registration call was unsuccessful

Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetRegisterAmxDevice(DEV device, RmsAsset asset)
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure RMS is ONLINE, REGISTERED, and ready for ASSET registration
  IF(![vdvRMS,RMS_CHANNEL_ASSETS_REGISTER])
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetRegisterAmxDevice> :: RMS is not ready to accept asset
    management registrations.';
    RETURN FALSE;
  }

  // the asset client key is the DPS string for NetLinx-based devices
  IF(asset.clientKey == '')
  {
    asset.clientKey = RmsDevToString(device);
  }

  // create asset registration send command
  rmsCommand = RmsPackCmdHeader('ASSET.REGISTER.AMX.DEV');
  rmsCommand = RmsPackCmdParam(rmsCommand,RmsDevToString(device));
  rmsCommand = RmsPackCmdParam(rmsCommand,asset.name);
  rmsCommand = RmsPackCmdParam(rmsCommand,asset.clientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,asset.assetType);
  rmsCommand = RmsPackCmdParam(rmsCommand,asset.globalKey);

  // add device to registration queue
  SEND_COMMAND vdvRMS, rmsCommand;

  // now that the asset has been created in the asset registration queue, finalize and submit the asset
  RETURN RmsAssetRegistrationSubmit(asset);
}
```

Asset Registration Functions (Cont.)	
RmsAssetRegister DuetDevice	*Description*: This method is used exclusively to perform asset registration for Duet based devices. This method should only be used when registering a device that is using a Duet device module. The RMS Client will interrogate the Duet device module to obtain the default device information, parameters, metadata properties, and control methods. *Arguments*: • **DEV device** - asset physical device instance • **DEV duetDevice** - Duet virtual device instance • **RmsAsset asset** - asset configuration to register *Returns*: 1 if asset registration call was successful; 0 if asset registration call was unsuccessful *Syntax*:

```
DEFINE_FUNCTION CHAR RmsAssetRegisterDuetDevice(DEV realDevice, DEV duetDevice, RmsAsset asset)
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure RMS is ONLINE, REGISTERED, and ready for ASSET registration
  IF(![vdvRMS,RMS_CHANNEL_ASSETS_REGISTER])
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetRegisterDuetDevice> :: RMS is not ready to accept asset
    management registrations.';
    RETURN FALSE;
  }

  // the asset client key is the DPS string for NetLinx-based devices
  IF(asset.clientKey == '')
  {
    asset.clientKey = RmsDevToString(realDevice);
  }

  // create asset registration send command
  rmsCommand = RmsPackCmdHeader('ASSET.REGISTER.DUET.DEV');
  rmsCommand = RmsPackCmdParam(rmsCommand,RmsDevToString(realDevice));
  rmsCommand = RmsPackCmdParam(rmsCommand,RmsDevToString(duetDevice));
  rmsCommand = RmsPackCmdParam(rmsCommand,asset.name);
  rmsCommand = RmsPackCmdParam(rmsCommand,asset.clientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,asset.assetType);
  rmsCommand = RmsPackCmdParam(rmsCommand,asset.globalKey);

  // add device to registration queue
  SEND_COMMAND vdvRMS, rmsCommand;

  // now that the asset has been created in the asset registration queue, finalize and submit the asset
  RETURN RmsAssetRegistrationSubmit(asset);
}
```

Asset Registration Functions (Cont.)

RmsAssetRegistration Submit

Description: This function is used to perform the registration on a fully populated RmsAsset data object.
Note: This function is typically not accessed directly, but rather called by a another asset registration wrapper function.
Arguments:

- **RmsAsset asset** - asset configuration to register

Returns:
1 if asset registration call was successful; 0 if asset registration call was unsuccessful
Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetRegistrationSubmit(RmsAsset asset)
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure RMS is ONLINE, REGISTERED, and ready for ASSET registration
  IF(![vdvRMS,RMS_CHANNEL_ASSETS_REGISTER])
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetRegistrationSubmit> :: RMS is not ready to accept asset
    management registrations.';
    RETURN FALSE;
  }

  // add asset description
  IF(asset.description != '')
  {
    rmsCommand = RmsPackCmdHeader('ASSET.DESCRIPTION');
    rmsCommand = RmsPackCmdParam(rmsCommand,asset.clientKey);
    rmsCommand = RmsPackCmdParam(rmsCommand,asset.description);
    SEND_COMMAND vdvRMS, rmsCommand;
  }

  // add asset serial number
  IF(asset.serialNumber != '')
  {
    rmsCommand = RmsPackCmdHeader('ASSET.SERIAL');
    rmsCommand = RmsPackCmdParam(rmsCommand,asset.clientKey);
    rmsCommand = RmsPackCmdParam(rmsCommand,asset.serialNumber);
    SEND_COMMAND vdvRMS, rmsCommand;
  }

  // add asset firmware version
  IF(asset.firmwareVersion != '')
  {
    rmsCommand = RmsPackCmdHeader('ASSET.FIRMWARE');
    rmsCommand = RmsPackCmdParam(rmsCommand,asset.clientKey);
    rmsCommand = RmsPackCmdParam(rmsCommand,asset.firmwareVersion);
    SEND_COMMAND vdvRMS, rmsCommand;
  }

  // add asset manufacturer information
  IF(asset.manufacturerName != '')
  {
    rmsCommand = RmsPackCmdHeader('ASSET.MANUFACTURER');
    rmsCommand = RmsPackCmdParam(rmsCommand,asset.clientKey);
    rmsCommand = RmsPackCmdParam(rmsCommand,asset.manufacturerName);
    rmsCommand = RmsPackCmdParam(rmsCommand,asset.manufacturerUrl);
    SEND_COMMAND vdvRMS, rmsCommand;
  }

  // add asset manufacturer information
  IF(asset.modelName != '')
  {
    rmsCommand = RmsPackCmdHeader('ASSET.MODEL');
    rmsCommand = RmsPackCmdParam(rmsCommand,asset.clientKey);
    rmsCommand = RmsPackCmdParam(rmsCommand,asset.modelName);
    rmsCommand = RmsPackCmdParam(rmsCommand,asset.modelUrl);
    SEND_COMMAND vdvRMS, rmsCommand;
  }
  // submit the asset registration now
  rmsCommand = RmsPackCmdHeader('ASSET.SUBMIT');
  rmsCommand = RmsPackCmdParam(rmsCommand,asset.clientKey);
  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

Asset Registration Functions (Cont.)	
RmsAssetExclude	*Description*: This function is used to define an exclusion an any asset attempting to register using the provided asset client key string identifier. *Arguments*: • **CHAR assetClientKey[]** - asset key to exclude *Returns*: 1 if call was successful; 0 if call was unsuccessful *Syntax*:

```
DEFINE_FUNCTION CHAR RmsAssetExclude(CHAR assetClientKey[])
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetExclude> :: missing asset client key';
    RETURN FALSE;
  }

  // submit the asset exclusion now
  rmsCommand = RmsPackCmdHeader('ASSET.EXCLUDE');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,'true'); // force exclusion
  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

Registering Asset Parameters

The following callback method will be invoked in each asset monitoring module when it is time to register asset parameters with RMS:

```
(***********************************************************)
(* Name:  RegisterAssetParameters                          *)
(* Args:  -none-                                           *)
(*                                                         *)
(* Desc:  This is a callback method that is invoked by     *)
(*        RMS to notify this module that it is time to     *)
(*        register this asset's parameters to be monitored *)
(*        by RMS.                                          *)
(*                                                         *)
(*        This method should not be invoked/called         *)
(*        by any user implementation code.                 *)
(***********************************************************)
DEFINE_FUNCTION RegisterAssetParameters()
{
}
```

The *RmsApi* Include File (which is already included by each asset monitoring module) provides the following wrapper functions for registering asset parameters with RMS:

- **RmsAssetParameterEnqueueEnumeration** (see page 66)
- **RmsAssetParameterEnqueueDecimal** (see page 66)
- **RmsAssetParameterEnqueueDecimalWithBargraph** (see page 67)
- **RmsAssetParameterEnqueueLevel** (see page 67)
- **RmsAssetParameterEnqueueNumber** (see page 68)
- **RmsAssetParameterEnqueueNumberWithBargraph** (see page 69)
- **RmsAssetParameterEnqueueString** (see page 70)
- **RmsAssetParameterEnqueueBoolean** (see page 70)
- **RmsAssetParameterEnqueue** (see page 71)
- **RmsAssetParameterSubmit** (see page 73)
- **RmsAssetParameterDelete** (see page 73)

Asset Parameters - Required & Optional Information

Each asset parameter requires the following information:

Asset Parameters - Required & Optional Information	
Key (String) [REQUIRED]	This is a unique identifier for this asset parameter. This ID string must be uniquely scoped for this asset. Parameter value updates and reset notifications will use this key identifier when operating on this parameter.
Name (String) [REQUIRED]	Friendly name for this asset parameter.

Asset Parameters - Required Information (Cont.)	
Description (String) [OPTIONAL]	Descriptive text for this asset parameter.

Data Type (String)
[REQUIRED]
	Data type for this asset parameter's value.
	The following parameter data types are supported for RMS asset parameters:
	• Enumeration - list of choices
	• Decimal - floating point number
	• Level - number with fixed upper and lower bounds
	• Number
	• String
	• Boolean
	A set of constants are defined for the available data types in the *RmsApi.axi* Include File.
	`// RMS Asset Parameter Data Types`
	`RMS_ASSET_PARAM_DATA_TYPE_NUMBER = 'NUMBER';`
	`RMS_ASSET_PARAM_DATA_TYPE_STRING = 'STRING';`
	`RMS_ASSET_PARAM_DATA_TYPE_ENUMERATION = 'ENUMERATION';`
	`RMS_ASSET_PARAM_DATA_TYPE_LEVEL = 'LEVEL';`
	`RMS_ASSET_PARAM_DATA_TYPE_BOOLEAN = 'BOOLEAN';`
	`RMS_ASSET_PARAM_DATA_TYPE_DECIMAL = 'DECIMAL';`

Reporting Type (String)
[OPTIONAL]
	Reporting type for this asset parameter's value. This type is only applied to parameters that have a special meaning in the RMS reporting system.
	A set of constants are defined for the available reporting types in the *RmsApi.axi* Include File:
	`// RMS Asset Parameter (Reporting) Types`
	`RMS_ASSET_PARAM_TYPE_NONE = 'NONE';`
	`RMS_ASSET_PARAM_TYPE_ASSET_ONLINE = 'ASSET_ONLINE';`
	`RMS_ASSET_PARAM_TYPE_ASSET_POWER = 'ASSET_POWER';`
	`RMS_ASSET_PARAM_TYPE_POWER_CONSUMPTION = 'POWER_CONSUMPTION';`
	`RMS_ASSET_PARAM_TYPE_ENVIONMENTAL_EMISSIONS = 'EMISSIONS';`
	`RMS_ASSET_PARAM_TYPE_LAMP_USAGE = 'LAMP_USAGE';`
	`RMS_ASSET_PARAM_TYPE_BATTERY_LEVEL = 'BATTERY_LEVEL';`
	`RMS_ASSET_PARAM_TYPE_BATTERY_CHARGING_STATE = 'BATTERY_CHARGING_STATE';`
	`RMS_ASSET_PARAM_TYPE_RFID_LOCATION = 'RFID_LOCATION';`
	`RMS_ASSET_PARAM_TYPE_SOURCE_USAGE = 'SOURCE_USAGE';`
	`RMS_ASSET_PARAM_TYPE_SOURCE_STATE = 'SOURCE_STATE'`
	`RMS_ASSET_PARAM_TYPE_SYSTEM_ONLINE = 'SYSTEM_ONLINE';`
	`RMS_ASSET_PARAM_TYPE_SYSTEM_POWER = 'SYSTEM_POWER';`
	`RMS_ASSET_PARAM_TYPE_TRANSPORT_STATE = 'TRANSPORT_STATE';`
	`RMS_ASSET_PARAM_TYPE_TRANSPORT_USAGE = 'TRANSPORT_USAGE';`
	`RMS_ASSET_PARAM_TYPE_DISPLAY_USAGE = 'DISPLAY_USAGE';`
	`RMS_ASSET_PARAM_TYPE_TEMPERATURE = 'TEMPERATURE';`
	`RMS_ASSET_PARAM_TYPE_SECURITY_STATE = 'SECURITY_STATE';`
	`RMS_ASSET_PARAM_TYPE_LIGHT_LEVEL = 'LIGHT_LEVEL';`
	`RMS_ASSET_PARAM_TYPE_SIGNAL_STRENGTH = 'SIGNAL_STRENGTH';`
	`RMS_ASSET_PARAM_TYPE_HVAC_STATE = 'HVAC_STATE';`
	`RMS_ASSET_PARAM_TYPE_DIALER_STATE = 'DIALER_STATE';`
	`RMS_ASSET_PARAM_TYPE_DOCKING_STATE = 'DOCKING_STATE';`

Initial Value (String) [OPTIONAL]	The current parameter value (if known) should be set at the initial value when the parameter gets register in the RMS system.
Units (String) [OPTIONAL]	If the parameter value should have a unit descriptor following the value in the user interface, then provide the unit descriptor here.
Allow Reset (Boolean) [REQUIRED]	If this parameter attribute is set to TRUE, then the asset parameter RESET option will be available in the RMS web user interface. Users can reset the value from the RMS web pages and a notification will be delivered to the asset monitoring module to notify it of the reset value.
Reset Value (String) [OPTIONAL]	If Allow Reset is enabled, the a default reset value should be provided. When users that perform a RESET from the RMS web user interface this reset value will be applied as the current value for the asset parameter.
Minimum Value (Signed Long) [OPTIONAL]	This parameter attribute only applies to parameters of a numeric data type: Number, Decimal, Level. It is optional for data types Number and Decimal and required for data type Level. This attribute specifies the lowest possible value that this parameter value can reach.
Maximum Value (Signed Long) [OPTIONAL]	This parameter attribute only applies to parameters of a numeric data type: Number, Decimal, Level. It is optional for data types Number and Decimal and required for data type Level. This attribute specifies the highest possible value that this parameter value can reach.
Enumeration (String) [OPTIONAL]	This parameter attribute only applies to and is required for parameters of data type: *Enumeration*. This attribute should contain a pipe delimited list of string values.
Track Changes (Boolean) [REQUIRED]	This parameter attribute determines if the value changes that this parameter reports to the RMS server are tracked for historical and reporting purposes. Only asset parameter values that have usefulness to reporting or historical reference should be tracked.

Asset Parameters - Required Information (Cont.)	

Bargraph Key (String)
[OPTIONAL]
	Asset parameters of data type *Number*, *Level*, or *Decimal* can have an associated bargraph applied in the RMS web user interface.
	Each bargraph has an associated key identifier.
	A set of the default asset parameter bargraphs are listed as constants in the *RmsApi.axi* Include File.
	`// RMS Asset Bargraph Keys`
	`RMS_ASSET_PARAM_BARGRAPH_GENERAL_PURPOSE = 'general';`
	`RMS_ASSET_PARAM_BARGRAPH_BATTERY_LEVEL = 'battery.level';`
	`RMS_ASSET_PARAM_BARGRAPH_VOLUME_LEVEL = 'volume.level';`
	`RMS_ASSET_PARAM_BARGRAPH_SIGNAL_STRENGTH = 'signal.strength';`
	`RMS_ASSET_PARAM_BARGRAPH_LIGHT_LEVEL = 'light.level';`
	`RMS_ASSET_PARAM_BARGRAPH_LAMP_CONSUMPTION = 'lamp.consumption';`
	`RMS_ASSET_PARAM_BARGRAPH_TEMPERATURE = 'temperature';`

Stock Parameters (Boolean) [REQUIRED]	

The following code snippets provide an example of registering two asset parameters with RMS:

```
// volume level
RmsAssetParameterEnqueueLevel(assetClientKey,            << asset key >
                        'touch.panel.volume.level',      << param key >>
                        'Volume Level',                  << name >>
                        'Master volume level',           << description >>
                         RMS_ASSET_PARAM_TYPE_NONE,      << report type >>
                             panelInfo.volumeLevel,      << init value >>
                             0,                          << minimum >>
                             100,                        << maximum >>
                             '%',                        << units >>
                             RMS_ALLOW_RESET_NO,         << allow reset >>
                             0,                          << reset value >>
                             RMS_TRACK_CHANGES_NO,       << track change>>
                    RMS_ASSET_PARAM_BARGRAPH_VOLUME_LEVEL); << bargraph >>
// volume mute
RmsAssetParameterEnqueueBoolean(assetClientKey,
                                'touch.panel.volume.mute',
                                'Volume Mute',
                                'Last reported volume mute state',
                                RMS_ASSET_PARAM_TYPE_NONE,
                                panelInfo.volumeMute,
                                RMS_ALLOW_RESET_NO,
                                FALSE,
                                RMS_TRACK_CHANGES_NO);
```

After all asset parameter registrations have been queued, you must perform a submit call to send all the registrations to the RMS server.

```
// submit all parameter registrations
RmsAssetParameterSubmit(assetClientKey);
```

Asset Parameters Registration and Update Functions

The *Asset Parameters Registration* and *Update* functions in the *RmsApi.axi* Include File are described in the following table:

Asset Parameters Registration and Update Functions

RmsAssetParameter EnqueueEnumeration

Description: This function is used to place an asset parameter registration in queue for a specified asset client key. The asset parameter being registered is of asset parameter data type: ENUMERATION. The enumeration values should be provided as a pipe "|" delimited list of string values.

Arguments: see method signature below

Returns: 1 if call was successful; 0 if call was unsuccessful

Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetParameterEnqueueEnumeration(CHAR assetClientKey[],
                                                         CHAR parameterKey[],
                                                         CHAR parameterName[],
                                                         CHAR parameterDescription[],
                                                         CHAR reportingType[],
                                                         CHAR initialValue[],
                                                         CHAR enumeration[],
                                                         CHAR allowReset,
                                                         CHAR resetValue[],
                                                         CHAR trackChanges)
{
  STACK_VAR RmsAssetParameter parameter

  // set all parameter properties for number param
  parameter.dataType = RMS_ASSET_PARAM_DATA_TYPE_ENUMERATION;
  parameter.key = parameterKey;
  parameter.name = parameterName;
  parameter.description = parameterDescription;
  parameter.reportingType = reportingType;
  parameter.initialValue = initialValue;
  parameter.allowReset = allowReset;
  parameter.resetValue = resetValue;
  parameter.trackChanges = trackChanges;
  parameter.enumeration = enumeration;

  RETURN RmsAssetParameterEnqueue(assetClientKey, parameter);
}
```

RmsAssetParameter EnqueueDecimal

Description: This function is used to place an asset parameter registration in queue for a specified asset client key. The asset parameter being registered is of asset parameter data type: DECIMAL

Arguments: see method signature below

Returns: 1 if call was successful; 0 if call was unsuccessful

Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetParameterEnqueueDecimal(CHAR assetClientKey[],
                                                     CHAR parameterKey[],
                                                     CHAR parameterName[],
                                                     CHAR parameterDescription[],
                                                     CHAR reportingType[],
                                                     DOUBLE initialValue,
                                                     SLONG minimumValue,
                                                     SLONG maximumValue,
                                                     CHAR units[],
                                                     CHAR allowReset,
                                                     DOUBLE resetValue,
                                                     CHAR trackChanges)
{
  STACK_VAR RmsAssetParameter parameter

  // set all parameter properties for number param
  parameter.dataType = RMS_ASSET_PARAM_DATA_TYPE_DECIMAL;
  parameter.key = parameterKey;
  parameter.name = parameterName;
  parameter.description = parameterDescription;
  parameter.reportingType = reportingType;
  parameter.initialValue = FTOA(initialValue);
  parameter.units = units;
  parameter.allowReset = allowReset;
  parameter.resetValue = FTOA(resetValue);
  parameter.trackChanges = trackChanges;
  parameter.minimumValue = minimumValue;
  parameter.maximumValue = maximumValue;

  RETURN RmsAssetParameterEnqueue(assetClientKey, parameter);
}
```

Asset Parameters Registration and Update Functions (Cont.)

RmsAssetParameter EnqueueDecimal WithBargraph

Description: This function is used to place an asset parameter registration in queue for a specified asset client key. The asset parameter being registered is of asset parameter data type: DECIMAL

Arguments: see method signature below

Returns: 1 if call was successful; 0 if call was unsuccessful

Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetParameterEnqueueDecimalWithBargraph
(CHAR assetClientKey[],
                                                    CHAR parameterKey[],
                                                    CHAR parameterName[],
                                                    CHAR parameterDescription[],
                                                    CHAR reportingType[],
                                                    DOUBLE initialValue,
                                                    SLONG minimumValue,
                                                    SLONG maximumValue,
                                                    CHAR units[],
                                                    CHAR allowReset,
                                                    DOUBLE resetValue,
                                                    CHAR trackChanges,
                                                    CHAR bargraphKey[])
{
  STACK_VAR RmsAssetParameter parameter

  // set all parameter properties for number param
  parameter.dataType = RMS_ASSET_PARAM_DATA_TYPE_DECIMAL;
  parameter.key = parameterKey;
  parameter.name = parameterName;
  parameter.description = parameterDescription;
  parameter.reportingType = reportingType;
  parameter.initialValue = FTOA(initialValue);
  parameter.units = units;
  parameter.allowReset = allowReset;
  parameter.resetValue = FTOA(resetValue);
  parameter.trackChanges = trackChanges;
  parameter.minimumValue = minimumValue;
  parameter.maximumValue = maximumValue;
  parameter.bargraphKey = bargraphKey;

  RETURN RmsAssetParameterEnqueue(assetClientKey, parameter);
}
```

RmsAssetParameter EnqueueLevel

Description: This function is used to place an asset parameter registration in queue for a specified asset client key. The asset parameter being registered is of asset parameter data type: LEVEL

Arguments: see method signature below

Returns: 1 if call was successful; 0 if call was unsuccessful

Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetParameterEnqueueLevel (CHAR assetClientKey[],
                                                    CHAR parameterKey[],
                                                    CHAR parameterName[],
                                                    CHAR parameterDescription[],
                                                    CHAR reportingType[],
                                                    SLONG initialValue,
                                                    SLONG minimumValue,
                                                    SLONG maximumValue,
                                                    CHAR units[],
                                                    CHAR allowReset,
                                                    SLONG resetValue,
                                                    CHAR trackChanges,
                                                    CHAR bargraphKey[])
{
  STACK_VAR RmsAssetParameter parameter

  // set all parameter properties for number param
  parameter.dataType = RMS_ASSET_PARAM_DATA_TYPE_LEVEL;
  parameter.key = parameterKey;
  parameter.name = parameterName;
  parameter.description = parameterDescription;
  parameter.reportingType = reportingType;
  parameter.initialValue = ITOA(initialValue);
  parameter.units = units;
  parameter.allowReset = allowReset;
  parameter.resetValue = ITOA(resetValue);
  parameter.trackChanges = trackChanges;
  parameter.minimumValue = minimumValue;
  parameter.maximumValue = maximumValue;
  parameter.bargraphKey = bargraphKey;

  RETURN RmsAssetParameterEnqueue(assetClientKey, parameter);
}
```

Asset Parameters Registration and Update Functions (Cont.)	

RmsAssetParameter EnqueueNumber	*Description*: This function is used to place an asset parameter registration in queue for a specified asset client key. The asset parameter being registered is of asset parameter data type: NUMBER
	Arguments: see method signature below
	Returns: 1 if call was successful; 0 if call was unsuccessful
	Syntax:
	`DEFINE_FUNCTION CHAR RmsAssetParameterEnqueueNumber(CHAR assetClientKey[], CHAR parameterKey[], CHAR parameterName[], CHAR parameterDescription[], CHAR reportingType[], SLONG initialValue, SLONG minimumValue, SLONG maximumValue, CHAR units[], CHAR allowReset, SLONG resetValue, CHAR trackChanges)`
	`{`
	`  STACK_VAR RmsAssetParameter parameter`
	
	`  // set all parameter properties for number param`
	`  parameter.dataType = RMS_ASSET_PARAM_DATA_TYPE_NUMBER;`
	`  parameter.key = parameterKey;`
	`  parameter.name = parameterName;`
	`  parameter.description = parameterDescription;`
	`  parameter.reportingType = reportingType;`
	`  parameter.initialValue = ITOA(initialValue);`
	`  parameter.units = units;`
	`  parameter.allowReset = allowReset;`
	`  parameter.resetValue = ITOA(resetValue);`
	`  parameter.trackChanges = trackChanges;`
	`  parameter.minimumValue = minimumValue;`
	`  parameter.maximumValue = maximumValue;`
	
	`  RETURN RmsAssetParameterEnqueue(assetClientKey, parameter);`
	`}`

Asset Parameters Registration and Update Functions (Cont.)	
RmsAssetParameter EnqueueNumber WithBargraph	*Description*: This function is used to place an asset parameter registration in queue for a specified asset client key. The asset parameter being registered is of asset parameter data type: NUMBER *Arguments*: see method signature below *Returns*: 1 if call was successful; 0 if call was unsuccessful *Syntax*:

```
DEFINE_FUNCTION CHAR RmsAssetParameterEnqueueNumberWithBargraph(CHAR assetClientKey[],
                                                                CHAR parameterKey[],
                                                                CHAR parameterName[],
                                                                CHAR parameterDescription[],
                                                                CHAR reportingType[],
                                                                SLONG initialValue,
                                                                SLONG minimumValue,
                                                                SLONG maximumValue,
                                                                CHAR units[],
                                                                CHAR allowReset,
                                                                SLONG resetValue,
                                                                CHAR trackChanges,
                                                                CHAR bargraphKey[])
{
  STACK_VAR RmsAssetParameter parameter

  // set all parameter properties for number param
  parameter.dataType = RMS_ASSET_PARAM_DATA_TYPE_NUMBER;
  parameter.key = parameterKey;
  parameter.name = parameterName;
  parameter.description = parameterDescription;
  parameter.reportingType = reportingType;
  parameter.initialValue = ITOA(initialValue);
  parameter.units = units;
  parameter.allowReset = allowReset;
  parameter.resetValue = ITOA(resetValue);
  parameter.trackChanges = trackChanges;
  parameter.minimumValue = minimumValue;
  parameter.maximumValue = maximumValue;
  parameter.bargraphKey = bargraphKey;

  RETURN RmsAssetParameterEnqueue(assetClientKey, parameter);
}

  // ensure a threshold value has been provided
  IF(threshold.value == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetParameterThresholdEnqueueEx> :: missing threshold value';
    RETURN FALSE;
  }

  // if a status type was not provided, then apply the NOT_ASSIGNED status type
  IF(threshold.statusType == '')
  {
    threshold.statusType = RMS_STATUS_TYPE_NOT_ASSIGNED;
  }

  // submit the asset parameter update now
  rmsCommand = RmsPackCmdHeader('ASSET.PARAM.THRESHOLD');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,parameterKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,threshold.name);
  rmsCommand = RmsPackCmdParam(rmsCommand,RmsBooleanString(threshold.enabled));
  rmsCommand = RmsPackCmdParam(rmsCommand,threshold.statusType);
  rmsCommand = RmsPackCmdParam(rmsCommand,threshold.comparisonOperator);
  rmsCommand = RmsPackCmdParam(rmsCommand,threshold.value);
  rmsCommand = RmsPackCmdParam(rmsCommand,RmsBooleanString(threshold.notifyOnTrip));
  rmsCommand = RmsPackCmdParam(rmsCommand,RmsBooleanString(threshold.notifyOnRestore));
  IF(threshold.delayInterval > 0)
  {
    rmsCommand = RmsPackCmdParam(rmsCommand,'true'); // delayed = TRUE
  }
  ELSE
  {
    rmsCommand = RmsPackCmdParam(rmsCommand,'false');  // delayed = FALSE
  }
  rmsCommand = RmsPackCmdParam(rmsCommand,ITOA(threshold.delayInterval));

  SEND_COMMAND vdvRMS, rmsCommand;
}
```

Asset Parameters Registration and Update Functions (Cont.)	

RmsAssetParameter EnqueueString	*Description*: This function is used to place an asset parameter registration in queue for a specified asset client key. The asset parameter being registered is of asset parameter data type: STRING
	Arguments: see method signature below
	Returns: 1 if call was successful; 0 if call was unsuccessful
	Syntax:
	`DEFINE_FUNCTION CHAR RmsAssetParameterEnqueueString(CHAR assetClientKey[], CHAR parameterKey[], CHAR parameterName[], CHAR parameterDescription[], CHAR reportingType[], CHAR initialValue[], CHAR units[], CHAR allowReset, CHAR resetValue[], CHAR trackChanges)`
	`{`
	`  STACK_VAR RmsAssetParameter parameter`
	`  // set all parameter properties for string param`
	`  parameter.dataType = RMS_ASSET_PARAM_DATA_TYPE_STRING;`
	`  parameter.key = parameterKey;`
	`  parameter.name = parameterName;`
	`  parameter.description = parameterDescription;`
	`  parameter.reportingType = reportingType;`
	`  parameter.initialValue = initialValue;`
	`  parameter.units = units;`
	`  parameter.allowReset = allowReset;`
	`  parameter.resetValue = resetValue;`
	`  parameter.trackChanges = trackChanges;`
	`  RETURN RmsAssetParameterEnqueue(assetClientKey, parameter);`
	`}`

RmsAssetParameter EnqueueBoolean	*Description*: This function is used to place an asset parameter registration in queue for a specified asset client key. The asset parameter being registered is of asset parameter data type: BOOLEAN
	Arguments: see method signature below
	Returns: 1 if call was successful; 0 if call was unsuccessful
	Syntax:
	`DEFINE_FUNCTION CHAR RmsAssetParameterEnqueueBoolean(CHAR assetClientKey[], CHAR parameterKey[], CHAR parameterName[], CHAR parameterDescription[], CHAR reportingType[], CHAR initialValue, CHAR allowReset, CHAR resetValue, CHAR trackChanges)`
	`{`
	`  STACK_VAR RmsAssetParameter parameter`
	`  // set all parameter properties for boolean param`
	`  parameter.dataType = RMS_ASSET_PARAM_DATA_TYPE_BOOLEAN;`
	`  parameter.key = parameterKey;`
	`  parameter.name = parameterName;`
	`  parameter.description = parameterDescription;`
	`  parameter.reportingType = reportingType;`
	`  parameter.initialValue = RmsBooleanString(initialValue);`
	`  parameter.allowReset = allowReset;`
	`  parameter.resetValue = RmsBooleanString(resetValue);`
	`  parameter.trackChanges = trackChanges;`
	`  RETURN RmsAssetParameterEnqueue(assetClientKey, parameter);`
	`}`

Asset Parameters Registration and Update Functions (Cont.)

RmsAssetParameter Enqueue

Description: This function is used to place an asset parameter registration in queue for a specified asset client key. This method accepts the data structure *RmsAssetParameter* as an argument

Arguments: see method signature below

Returns: 1 if call was successful; 0 if call was unsuccessful

Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetParameterEnqueue(CHAR assetClientKey[],
                                              RmsAssetParameter parameter)
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetParameterEnqueue> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a parameter key has been provided
  IF(parameter.key == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetParameterEnqueue> :: missing parameter key';
    RETURN FALSE;
  }

  // ensure a parameter data type is assigned
  IF(parameter.dataType == '')
    parameter.dataType = RMS_ASSET_PARAM_DATA_TYPE_STRING;

  // ensure a parameter reporting type is assigned
  IF(parameter.reportingType == '')
    parameter.reportingType = RMS_ASSET_PARAM_TYPE_NONE;

  // enqueue asset control method for registration
  rmsCommand = RmsPackCmdHeader('ASSET.PARAM');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,parameter.key);
  rmsCommand = RmsPackCmdParam(rmsCommand,parameter.name);
  rmsCommand = RmsPackCmdParam(rmsCommand,parameter.description);
  rmsCommand = RmsPackCmdParam(rmsCommand,parameter.dataType);
  rmsCommand = RmsPackCmdParam(rmsCommand,parameter.reportingType);
  rmsCommand = RmsPackCmdParam(rmsCommand,parameter.initialValue);
  rmsCommand = RmsPackCmdParam(rmsCommand,parameter.units);
  rmsCommand = RmsPackCmdParam(rmsCommand,RmsBooleanString(parameter.allowReset));
  rmsCommand = RmsPackCmdParam(rmsCommand,parameter.resetValue);
  rmsCommand = RmsPackCmdParam(rmsCommand,ITOA(parameter.minimumValue));
  IF(parameter.maximumValue > parameter.minimumValue)
  {
    // only register a max value if it is greater than the minimum value
    rmsCommand = RmsPackCmdParam(rmsCommand,ITOA(parameter.maximumValue));
  }
  ELSE
  {
    // undefined max value
    rmsCommand = RmsPackCmdParam(rmsCommand,'');
  }
 rmsCommand = RmsPackCmdParam(rmsCommand,parameter.enumeration);
 rmsCommand = RmsPackCmdParam(rmsCommand,RmsBooleanString(parameter.trackChanges));
 rmsCommand = RmsPackCmdParam(rmsCommand,parameter.bargraphKey);
 rmsCommand = RmsPackCmdParam(rmsCommand,RmsBooleanString(parameter.stockParam));

  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

Asset Parameters Registration and Update Functions (Cont.)

RmsAssetOnlineParameter Enqueue

Description: This function is used to queue an asset online parameter for a specified asset key.

Arguments:

- CHAR assetClientKey[]
- INTEGER nOnline

Returns: n/a

Syntax:

```
DEFINE_FUNCTION RmsAssetOnlineParameterEnqueue(CHAR assetClientKey[], INTEGER nOnline)
{
  STACK_VAR RmsAssetParameter parameter;
  STACK_VAR RmsAssetParameterThreshold threshold;

  // device device online parameter
  parameter.key = RMS_KEY_ASSET_ONLINE;
  parameter.name = 'Online Status';
  parameter.description = 'Current asset online or offline state';
  parameter.dataType = RMS_ASSET_PARAM_DATA_TYPE_ENUMERATION;
  parameter.enumeration = 'Offline|Online';
  parameter.allowReset = RMS_ALLOW_RESET_NO;
  parameter.stockParam = TRUE;
  parameter.reportingType = RMS_ASSET_PARAM_TYPE_ASSET_ONLINE;
  parameter.trackChanges = RMS_TRACK_CHANGES_YES;
  parameter.resetValue = 'Offline';
  IF(nOnline)
  {
    parameter.initialValue = 'Online';
  }
  ELSE
  {
    parameter.initialValue = 'Offline';
  }

  // enqueue parameter
  RmsAssetParameterEnqueue(assetClientKey, parameter);

  // populate default threshold settings
  threshold.name = 'Offline';
  threshold.comparisonOperator = RMS_ASSET_PARAM_THRESHOLD_COMPARISON_EQUAL;
  threshold.value = 'Offline';
  threshold.statusType = RMS_STATUS_TYPE_MAINTENANCE;
  threshold.enabled = TRUE;
  threshold.notifyOnRestore = TRUE;
  threshold.notifyOnTrip = TRUE;

  // add a default threshold for the device online/offline parameter
  RmsAssetParameterThresholdEnqueueEx(assetClientKey,
                                      parameter.key,
                                      threshold)
}
```

RmsAssetOnlineParameter Update

Description: This function is used to update an asset online parameter for a specified asset key.

Arguments:

- CHAR assetClientKey[]
- INTEGER nOnline

Returns: n/a

Syntax:

```
DEFINE_FUNCTION RmsAssetOnlineParameterUpdate(CHAR assetClientKey[], INTEGER nOnline)
{
  // update RMS parameter value for the device online parameter
  IF(nOnline)
  {
    RmsAssetParameterSetValue(assetClientKey,RMS_KEY_ASSET_ONLINE,'Online');
  }
  ELSE
  {
    RmsAssetParameterSetValue(assetClientKey,RMS_KEY_ASSET_ONLINE,'Offline');
  }
}
```

Asset Parameters Registration and Update Functions (Cont.)	

RmsAssetParameterSubmit	*Description*: This function is used to submit any pending asset monitored parameter that are currently in queue waiting to be registered with RMS.
	Arguments:
	• **CHAR assetClientKey[]** - asset client key
	Returns: 1 if call was successful; 0 if call was unsuccessful
	Syntax:
	`DEFINE_FUNCTION CHAR RmsAssetParameterSubmit(CHAR assetClientKey[])`
	`{`
	`  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];`
	
	`  // ensure RMS is ONLINE, REGISTERED, and ready for ASSET registration`
	`  IF(![vdvRMS,RMS_CHANNEL_ASSETS_REGISTER])`
	`  {`
	`  SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetParameterSubmit> :: RMS is not ready to accept asset parameters changes.';`
	`  RETURN FALSE;`
	`  }`
	
	`  // ensure an asset client key has been provided`
	`  IF(assetClientKey == '')`
	`  {`
	`  SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetParameterSubmit> :: missing asset client key';`
	`  RETURN FALSE;`
	`  }`
	
	`  // submit the asset parameter registration now`
	`  rmsCommand = RmsPackCmdHeader('ASSET.PARAM.SUBMIT');`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);`
	`  SEND_COMMAND vdvRMS, rmsCommand;`
	
	`  RETURN TRUE;`
	`}`

RmsAssetParameterDelete	*Description*: This function is used to delete an existing asset monitored parameter from the RMS server.
	Arguments:
	• **CHAR assetClientKey[]** - asset client key
	• **CHAR parameterKey[]** - monitored parameter key
	Returns: 1 if call was successful; 0 if call was unsuccessful
	Syntax:
	`DEFINE_FUNCTION CHAR RmsAssetParameterDelete(CHAR assetClientKey[],`
	`  CHAR parameterKey[])`
	`{`
	`  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];`
	
	`  // ensure RMS is ONLINE, REGISTERED, and ready for ASSET registration`
	`  IF(![vdvRMS,RMS_CHANNEL_ASSETS_REGISTER])`
	`  {`
	`  SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetParameterDelete> :: RMS is not ready to accept asset parameters changes.';`
	`  RETURN FALSE;`
	`  }`
	
	`  // ensure an asset client key has been provided`
	`  IF(assetClientKey == '')`
	`  {`
	`  SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetParameterDelete> :: missing asset client key';`
	`  RETURN FALSE;`
	`  }`
	
	`  // ensure a parameter key has been provided`
	`  IF(parameterKey == '')`
	`  {`
	`  SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetParameterDelete> :: missing parameter key';`
	`  RETURN FALSE;`
	`  }`
	
	`  // submit the asset parameter delete command`
	`  rmsCommand = RmsPackCmdHeader('ASSET.PARAM.DELETE');`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,parameterKey);`
	`  SEND_COMMAND vdvRMS, rmsCommand;`
	
	`  RETURN TRUE;`
	`}`

Registering Asset Parameters Thresholds

During asset parameter registration you can also register one or more asset parameter thresholds. Asset parameter thresholds are used in RMS as alerting triggers for individual parameter values if they exceed some established threshold value.

The *RmsApi* Include File (which is already included by each asset monitoring module) provides the following wrapper function for registering asset parameter thresholds with RMS.

- **RmsAssetParameterThresholdEnqueue** (see page 75)
- **RmsAssetParameterThresholdEnqueueEx** (see page 75)

Each asset parameter threshold requires the following information:

Asset Parameter Thresholds - Required & Optional Information	
Name (String) [REQUIRED]	Friendly name for this asset parameter threshold. The name must be unique for this asset parameter.
Enabled (Boolean) [REQUIRED]	Defines if this asset parameter threshold is enabled by default.

Status Type (String)
[OPTIONAL]
	Defines the RMS status type (alert type) if this asset parameter threshold is tripped.
	A set of the default status types are listed as constants in the *RmsApi.axi* Include File.
	`// RMS Status Types`
	`RMS_STATUS_TYPE_NOT_ASSIGNED = 'NOT_ASSIGNED';`
	`RMS_STATUS_TYPE_HELP_REQUEST = 'HELP_REQUEST';`
	`RMS_STATUS_TYPE_ROOM_COMMUNICATION_ERROR = 'ROOM_COMMUNICATION_ERROR';`
	`RMS_STATUS_TYPE_CONTROL_SYSTEM_ERROR = 'CONTROL_SYSTEM_ERROR';`
	`RMS_STATUS_TYPE_MAINTENANCE = 'MAINTENANCE';`
	`RMS_STATUS_TYPE_EQUIPMENT_USAGE = 'EQUIPMENT_USAGE';`
	`RMS_STATUS_TYPE_NETWORK = 'NETWORK';`
	`RMS_STATUS_TYPE_SECURITY = 'SECURITY';`
	RMS Enterprise also supports user defined status types. Users can create new custom status types in the RMS Web User Interface and custom assign programming keys.

Comparison Operator (String)
[OPTIONAL]
	Defines the comparison type to use when evaluating the current parameter value against the threshold value.
	A set of the default comparison threshold operator types are listed as constants in the *RmsApi.axi* Include File:
	`// RMS Asset Parameter Threshold Comparison Operators`
	`RMS_ASSET_PARAM_THRESHOLD_COMPARISON_NONE = 'NONE';`
	`RMS_ASSET_PARAM_THRESHOLD_COMPARISON_LESS_THAN = 'LESS_THAN';`
	`RMS_ASSET_PARAM_THRESHOLD_COMPARISON_LESS_THAN_EQUAL = 'LESS_THAN_EQUAL_TO';`
	`RMS_ASSET_PARAM_THRESHOLD_COMPARISON_GREATER_THAN = 'GREATER_THAN';`
	`RMS_ASSET_PARAM_THRESHOLD_COMPARISON_GREATER_EQUAL = 'GREATER_THAN_EQUAL_TO';`
	`RMS_ASSET_PARAM_THRESHOLD_COMPARISON_EQUAL = 'EQUAL_TO';`
	`RMS_ASSET_PARAM_THRESHOLD_COMPARISON_NOT_EQUAL = 'NOT_EQUAL_TO';`
	`RMS_ASSET_PARAM_THRESHOLD_COMPARISON_CONTAINS = 'CONTAINS';`
	`RMS_ASSET_PARAM_THRESHOLD_COMPARISON_DOES_NOT_CONTAIN = 'DOES_NOT_CONTAIN';`

Value (String) [OPTIONAL]	Defines the threshold value that will be compared against the comparison type to use when evaluating asset parameter value changes.
Notify On Trip (Boolean) [OPTIONAL]	If set to TRUE, then when this asset parameter threshold is tripped email notifications will be sent out to registered recipients of the status type and the item will be placed in the RMS Hotlist.
Notify On Restore (Boolean) [OPTIONAL]	If set to TRUE, then when this asset parameter threshold is restored email notifications will be sent out to registered recipients of the status type.
Delay Interval (Integer) [OPTIONAL]	If set to a value greater than zero, then the asset parameter value must remain at a value that exceeds the defined threshold value for this delay interval the specified number of seconds before RMS will trip the asset parameter threshold and cause notifications and hotlist records to get generated.

The following code snippet provides an example of registering an asset parameter and accompanying asset parameter threshold:

```
// register asset parameter for PDU unit overcurrent alarm
RmsAssetParameterEnqueueBoolean(assetClientKey,
                        'pdu.overcurrent.alarm',
                        'Chassis Overcurrent Alarm',
                        'Last reported input voltage into the PDU.',
                        RMS_ASSET_PARAM_TYPE_NONE,
                        pduCache.overcurrentAlarm,
                        RMS_ALLOW_RESET_YES,
                        0,
                        RMS_TRACK_CHANGES_YES);
// create a threshold for when the OVERCURRENT alarm occurs
RmsAssetParameterThresholdEnqueue(assetClientKey,                   << asset key >>
                        'pdu.overcurrent.alarm',                    << param key >>
                        'Overcurrent',                              << threshold name >>
                         RMS_STATUS_TYPE_MAINTENANCE,               << status type >>
                         RMS_ASSET_PARAM_THRESHOLD_COMPARISON_EQUAL, << compare op >>
                                  'true');                          << threshold value >>
```

Asset Parameter Thresholds Functions

The *Asset Parameter Thresholds* functions in the *RmsApi.axi* Include File are described in the following table:

Asset Parameter Thresholds Functions

RmsAssetParameter ThresholdEnqueue

Description: This function is used to add an asset parameter threshold on the asset parameter currently pending in the asset parameter registration queue.

Arguments: see method signature below

Returns: 1 if call was successful; 0 if call was unsuccessful

Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetParameterThresholdEnqueue(CHAR assetClientKey[],
                                                       CHAR parameterKey[],
                                                       CHAR thresholdName[],
                                                       CHAR thresholdStatusType[],
                                                       CHAR thresholdComparisonOperator[],
                                                       CHAR thresholdValue[])
{
  STACK_VAR RmsAssetParameterThreshold threshold;

  // setup threshold data structure
  threshold.name = thresholdName;
  threshold.statusType = thresholdStatusType;
  threshold.comparisonOperator = thresholdComparisonOperator;
  threshold.value = thresholdValue;
  threshold.delayInterval = 0;
  threshold.notifyOnTrip = TRUE;
  threshold.notifyOnRestore = FALSE;
  threshold.enabled = TRUE;

  RETURN RmsAssetParameterThresholdEnqueueEx(assetClientKey, parameterKey, threshold);
}
```

RmsAssetParameter ThresholdEnqueueEx

Description: This function is used to add an asset parameter threshold on the asset parameter currently pending in the asset parameter registration queue. This EXTENDED function accepts the data structure *RmsAssetParameterThreshold* as an argument.

Arguments: see method signature below

Returns: 1 if call was successful; 0 if call was unsuccessful

Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetParameterThresholdEnqueueEx(CHAR assetClientKey[],CHAR parameterKey[],
                                                         RmsAssetParameterThreshold threshold)
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetParameterThresholdEnqueueEx> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a parameter key has been provided
  IF(parameterKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetParameterThresholdEnqueueEx> :: missing parameter key';
    RETURN FALSE;
  }

  // ensure a threshold name has been provided
  IF(threshold.name == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetParameterThresholdEnqueueEx> :: missing threshold name';
    RETURN FALSE;
  }
  // ensure a threshold comparison operator has been provided
  IF(threshold.comparisonOperator == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetParameterThresholdEnqueueEx> :: missing threshold
    comparison operator';
    RETURN FALSE;
  }
```

Updating Asset Parameter Values

Asset parameter value changes will need to be sent to RMS as they change on the NetLinx system. Parameters that have a high frequency of changes should be throttled in a way that parameter updates are sent to RMS on a limited basis; possible limiting by imposing logic where only changes that reach at certain statistical significance are sent or logic where changes are only updated on a set time delay.

The *RmsApi.axi* Include File (which is already included by each asset monitoring module) provides the following wrapper functions for updating asset parameter values with RMS.

These functions forward an immediate parameter value change to the RMS server:

- **RmsAssetParameterSetValueBoolean** (see page 77)
- **RmsAssetParameterSetValueNumber** (see page 77)
- **RmsAssetParameterSetValueDecimal** (see page 77)
- **RmsAssetParameterSetValueLevel** (see page 77)
- **RmsAssetParameterSetValue** (see page 78)
- **RmsAssetParameterUpdateValue** (see page 78)

These functions queue parameter value changes and only forward them to the RMS server when the Submit method is called. This queuing technique should be used when there are more than 2 parameter updates that occur back-to-back.

- **RmsAssetParameterEnqueueSetValueBoolean** (see page 79)
- **RmsAssetParameterEnqueueSetValueNumber** (see page 79)
- **RmsAssetParameterEnqueueSetValueDecimal** (see page 79)
- **RmsAssetParameterEnqueueSetValueLevel** (see page 79)
- **RmsAssetParameterEnqueueSetValue** (see page 80)
- **RmsAssetParameterEnqueueUpdateValue** (see page 80)
- **RmsAssetParameterUpdatesSubmit** (see page 81)

Asset Parameter Value Updates - Required Information

Each asset parameter value update requires the following information:

Asset Parameter Value Updates - Required Information	
Asset Key (String) [REQUIRED]	
Parameter Key (String) [REQUIRED]	

Parameter Operation (String)
[REQUIRED]
	This is the operation to perform against the current parameter value with the new value provided in the update call.
	A set of the available update operations are listed as constants in the *RmsApi.axi* Include File:
	`// RMS Asset Parameter Update Operation`
	`RMS_ASSET_PARAM_UPDATE_OPERATION_SET = 'SET_VALUE';`
	`RMS_ASSET_PARAM_UPDATE_OPERATION_INCREMENT = 'INCREMENT_VALUE';`
	`RMS_ASSET_PARAM_UPDATE_OPERATION_DECREMENT = 'DECREMENT_VALUE';`
	`RMS_ASSET_PARAM_UPDATE_OPERATION_DIVIDE = 'DIVIDE_VALUE';`
	`RMS_ASSET_PARAM_UPDATE_OPERATION_RESET = 'RESET_VALUE';`

Parameter Value (String) [REQUIRED]	This is the value that will be applied to the parameter value using the provided operation.

- Each of the *RmsAssetParameterSetValueXXX* and *RmsAssetParameterEnqueueSetValueXXX* functions impose an update operation of SET_VALUE.
- The *RmsAssetParameterUpdateValueXXX* and *RmsAssetParameterEnqueueUpdateValueXXX* functions allow the caller to define which operation to perform.

The following code snippet provides an example of immediately updating an asset parameter value.

```
// send docked status update to RMS
RmsAssetParameterSetValue(assetClientKey,'touch.panel.docked','true');
```

Asset Parameter Set Value Functions for Immediate Value Change

The *Asset Parameter Set Value* functions (for immediate value change) in the *RmsApi.axi* Include File are described in the following table:

Asset Parameter Set Value Functions	

RmsAssetParameter SetValueBoolean	*Description*: This function is used to set a new asset parameter value to the RMS server immediately.
	This function will set an asset parameter of data type: BOOLEAN
	Arguments:
	• **CHAR assetClientKey[]** - asset client key
	• **CHAR parameterKey[]** - monitored parameter key
	• **CHAR parameterValue** - monitored parameter value
	Returns:1 if call was successful; 0 if call was unsuccessful
	Syntax:
	`DEFINE_FUNCTION CHAR RmsAssetParameterSetValueBoolean(CHAR assetClientKey[],CHAR parameterKey[], CHAR parameterValue)`
	`{`
	`  // submit the asset parameter update now`
	`  RETURN RmsAssetParameterSetValue(assetClientKey,parameterKey,RmsBooleanString(parameterValue));`
	`}`

RmsAssetParameter SetValueNumber	*Description*: This function is used to set a new asset parameter value to the RMS server immediately.
	This function will set an asset parameter of data type: NUMBER
	Arguments:
	• **CHAR assetClientKey[]** - asset client key
	• **CHAR parameterKey[]** - monitored parameter key
	• **SLONG parameterValue** - monitored parameter value
	Returns: 1 if call was successful; 0 if call was unsuccessful
	Syntax:
	`DEFINE_FUNCTION CHAR RmsAssetParameterSetValueNumber(CHAR assetClientKey[],CHAR parameterKey[], SLONG parameterValue)`
	`{`
	`  RETURN RmsAssetParameterSetValue(assetClientKey,parameterKey,ITOA(parameterValue));`
	`}`

RmsAssetParameter SetValueDecimal	*Description*: This function is used to set a new asset parameter value to the RMS server immediately.
	This function will set an asset parameter of data type: DECIMAL
	Arguments:
	• **CHAR assetClientKey[]** - asset client key
	• **CHAR parameterKey[]** - monitored parameter key
	• **DOUBLE parameterValue** - monitored parameter value
	Returns: 1 if call was successful; 0 if call was unsuccessful
	Syntax:
	`DEFINE_FUNCTION CHAR RmsAssetParameterSetValueDecimal(CHAR assetClientKey[],CHAR parameterKey[], DOUBLE parameterValue)`
	`{`
	`  RETURN RmsAssetParameterSetValue(assetClientKey,parameterKey,FTOA(parameterValue));`
	`}`

RmsAssetParameter SetValueLevel	*Description*: This function is used to set a new asset parameter value to the RMS server immediately.
	This function will set an asset parameter of data type: LEVEL
	Arguments:
	• **CHAR assetClientKey[]** - asset client key
	• **CHAR parameterKey[]** - monitored parameter key
	• **SLONG parameterValue** - monitored parameter value
	Returns: 1 if call was successful; 0 if call was unsuccessful
	Syntax:
	`DEFINE_FUNCTION CHAR RmsAssetParameterSetValueLevel (CHAR assetClientKey[],CHAR parameterKey[], SLONG parameterValue)`
	`{`
	`  RETURN RmsAssetParameterSetValue(assetClientKey,parameterKey,ITOA(parameterValue));`
	`}`

Asset Parameter Set Value Functions (Cont.)

RmsAssetParameterSetValue

Description: This function is used to set a new asset parameter value to the RMS server immediately.
This function will set an asset parameter of data type: STRING
Arguments:

- **CHAR assetClientKey[]** - asset client key
- **CHAR parameterKey[]** - monitored parameter key
- **CHAR parameterValue[]** - monitored parameter value

Returns: 1 if call was successful; 0 if call was unsuccessful
Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetParameterSetValue(CHAR assetClientKey[],CHAR parameterKey[],CHAR parameterValue[])
{
  // submit the asset parameter update now
  RETURN RmsAssetParameterUpdateValue(assetClientKey,
                                      parameterKey,
                                      RMS_ASSET_PARAM_UPDATE_OPERATION_SET,
                                      parameterValue);
}
```

RmsAssetParameterUpdateValue

Description: This function is used to update an asset parameter value to the RMS server immediately.
Arguments:

- **CHAR assetClientKey[]** - asset client key
- **CHAR parameterKey[]** - monitored parameter key
- **CHAR parameterOperation[]** - update operation
- **CHAR parameterValue[]** - update parameter value

Returns: 1 if call was successful; 0 if call was unsuccessful
Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetParameterUpdateValue(CHAR assetClientKey[],CHAR parameterKey[],
                                                  CHAR parameterOperation[], CHAR parameterValue[])
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure RMS is ONLINE, REGISTERED, and ready for ASSET registration
  IF(![vdvRMS,RMS_CHANNEL_ASSETS_REGISTER])
  {
    RETURN FALSE;
  }

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, "'>>>> RMS API ERROR <RmsAssetParameterUpdateValue> :: missing asset client key for
    parameter: ',parameterKey";
    RETURN FALSE;
  }

  // ensure a parameter key has been provided
  IF(parameterKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetParameterUpdateValue> :: missing parameter key';
    RETURN FALSE;
  }

  // if an operation was not provided, then apply the SET operation
  IF(parameterOperation == '')
  {
    parameterOperation = RMS_ASSET_PARAM_UPDATE_OPERATION_SET;
  }

  // submit the asset parameter update now
  rmsCommand = RmsPackCmdHeader('ASSET.PARAM.UPDATE');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,parameterKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,parameterOperation);
  rmsCommand = RmsPackCmdParam(rmsCommand,parameterValue);
  rmsCommand = RmsPackCmdParam(rmsCommand,'true');  // SUBMIT-NOW = TRUE
  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

Asset Parameter Set Value Functions via Update Queue

The *Asset Parameter Set Value* functions (via Update Queue) in the *RmsApi.axi* Include File are described in the following table:

Asset Parameter Set Value Functions via Update Queue	

RmsAssetParameter EnqueueSetValueBoolean	*Description*: This function is used to set a new asset parameter value to the RMS server. The update is not sent immediately, but rather placed in an update queue waiting for a submission call.
	This function will set an asset parameter of data type: BOOLEAN
	Arguments:
	• **CHAR assetClientKey[]** - asset client key
	• **CHAR parameterKey[]** - monitored parameter key
	• **CHAR parameterValue** - monitored parameter value
	Returns: 1 if call was successful; 0 if call was unsuccessful
	Syntax:
	`DEFINE_FUNCTION CHAR RmsAssetParameterEnqueueSetValueBoolean(CHAR assetClientKey[],CHAR parameterKey[], CHAR parameterValue)`
	`{`
	`  // enqueue the asset parameter update`
	`  RETURN RmsAssetParameterEnqueueSetValue(assetClientKey,parameterKey,RmsBooleanString(parameterValue));`
	`}`

RmsAssetParameter EnqueueSetValueNumber	*Description*: This function is used to set a new asset parameter value to the RMS server. The update is not sent immediately, but rather placed in an update queue waiting for a submission call.
	This function will set an asset parameter of data type: NUMBER
	Arguments:
	• **CHAR assetClientKey[]** - asset client key
	• **CHAR parameterKey[]** - monitored parameter key
	• **SLONG parameterValue** - monitored parameter value
	Returns: 1 if call was successful; 0 if call was unsuccessful
	Syntax:
	`DEFINE_FUNCTION CHAR RmsAssetParameterEnqueueSetValueNumber(CHAR assetClientKey[],CHAR parameterKey[], SLONG parameterValue)`
	`{`
	`  // enqueue the asset parameter update`
	`  RETURN RmsAssetParameterEnqueueSetValue(assetClientKey,parameterKey,ITOA(parameterValue));`
	`}`

RmsAssetParameter EnqueueSetValueDecimal	*Description*: This function is used to set a new asset parameter value to the RMS server. The update is not sent immediately, but rather placed in an update queue waiting for a submission call.
	This function will set an asset parameter of data type: DECIMAL
	Arguments:
	• **CHAR assetClientKey[]** - asset client key
	• **CHAR parameterKey[]** - monitored parameter key
	• **DOUBLE parameterValue** - monitored parameter value
	Returns: 1 if call was successful; 0 if call was unsuccessful
	Syntax:
	`DEFINE_FUNCTION CHAR RmsAssetParameterEnqueueSetValueDecimal(CHAR assetClientKey[],CHAR parameterKey[], DOUBLE parameterValue)`
	`{`
	`  // enqueue the asset parameter update`
	`  RETURN RmsAssetParameterEnqueueSetValue(assetClientKey,parameterKey,FTOA(parameterValue));`
	`}`

RmsAssetParameter EnqueueSetValueLevel	*Description*: This function is used to set a new asset parameter value to the RMS server. The update is not sent immediately, but rather placed in an update queue waiting for a submission call.
	This function will set an asset parameter of data type: LEVEL
	Arguments:
	• **CHAR assetClientKey[]** - asset client key
	• **CHAR parameterKey[]** - monitored parameter key
	• **SLONG parameterValue** - monitored parameter value
	Returns: 1 if call was successful; 0 if call was unsuccessful
	Syntax:
	`DEFINE_FUNCTION CHAR RmsAssetParameterEnqueueSetValueLevel(CHAR assetClientKey[],CHAR parameterKey[], SLONG parameterValue)`
	`{`
	`  // enqueue the asset parameter update`
	`  RETURN RmsAssetParameterEnqueueSetValue(assetClientKey,parameterKey,ITOA(parameterValue));`
	`}`

Asset Parameter Set Value Functions via Update Queue (Cont.)

RmsAssetParameter EnqueueSetValue

Description: This function is used to set a new asset parameter value to the RMS server. The update is not sent immediately, but rather placed in an update queue waiting for a submission call.
This function will set an asset parameter of data type: STRING
Arguments:

- **CHAR assetClientKey[]** - asset client key
- **CHAR parameterKey[]** - monitored parameter key
- **CHAR parameterValue[]** - monitored parameter value

Returns: 1 if call was successful; 0 if call was unsuccessful
Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetParameterEnqueueSetValue(CHAR assetClientKey[],CHAR parameterKey[],
                                                      CHAR parameterValue[])
{
  // enqueue the asset parameter update
  RETURN RmsAssetParameterEnqueueUpdateValue(assetClientKey,parameterKey,
                                             RMS_ASSET_PARAM_UPDATE_OPERATION_SET,parameterValue);
}
```

RmsAssetParameter EnqueueUpdateValue

Description: This function is used to update an asset parameter value on the RMS server. The update is not sent immediately, but rather placed in an update queue waiting for a submission call.
Arguments:

- **CHAR assetClientKey[]** - asset client key
- **CHAR parameterKey[]** - monitored parameter key
- **CHAR parameterOperation[]** - update operation
- **CHAR parameterValue[]** - update parameter value

Returns: 1 if call was successful; 0 if call was unsuccessful
Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetParameterEnqueueUpdateValue(CHAR assetClientKey[],CHAR parameterKey[],
                                                         CHAR parameterOperation[],CHAR parameterValue[])
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure RMS is ONLINE, REGISTERED, and ready for ASSET registration
  IF(![vdvRMS,RMS_CHANNEL_ASSETS_REGISTER])
  {
    RETURN FALSE;
  }

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetParameterEnqueueUpdateValue> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a parameter key has been provided
  IF(parameterKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetParameterEnqueueUpdateValue> :: missing parameter key';
    RETURN FALSE;
  }

  // if an operation was not provided, then apply the SET operation
  IF(parameterOperation == '')
  {
    parameterOperation = RMS_ASSET_PARAM_UPDATE_OPERATION_SET;
  }

  // enqueue the asset parameter update
  rmsCommand = RmsPackCmdHeader('ASSET.PARAM.UPDATE');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,parameterKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,parameterOperation);
  rmsCommand = RmsPackCmdParam(rmsCommand,parameterValue);
  rmsCommand = RmsPackCmdParam(rmsCommand,'false');  // SUBMIT-NOW = FALSE
  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

Asset Parameter Set Value Functions via Update Queue (Cont.)	

RmsAssetParameter UpdatesSubmit	*Description*: This function is used to submit all pending asset parameter value updated to the RMS server.
	Arguments:
	• **CHAR assetClientKey[]** - asset client key
	Returns: 1 if call was successful; 0 if call was unsuccessful
	Syntax:
	`DEFINE_FUNCTION CHAR RmsAssetParameterUpdatesSubmit(CHAR assetClientKey[])`
	`{`
	`  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];`
	`  // ensure RMS is ONLINE, REGISTERED, and ready for ASSET registration`
	`  IF(![vdvRMS,RMS_CHANNEL_ASSETS_REGISTER])`
	`  {`
	`  SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetParameterUpdatesSubmit> :: RMS is not ready to accept asset parameters changes.';`
	`  RETURN FALSE;`
	`  }`
	`  // ensure an asset client key has been provided`
	`  IF(assetClientKey == '')`
	`  {`
	`  SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetParameterUpdatesSubmit> :: missing asset client key';`
	`  RETURN FALSE;`
	`  }`
	`  // submit the pending asset parameter updates now`
	`  rmsCommand = RmsPackCmdHeader('ASSET.PARAM.UPDATE.SUBMIT');`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);`
	`  SEND_COMMAND vdvRMS, rmsCommand;`
	`  RETURN TRUE;`
	`}`

Synchronizing Asset Parameter Values

The following callback method will be invoked in each asset monitoring module when it is time to perform the Asset parameter value synchronization with RMS.

```
(***********************************************************)
(* Name:  SynchronizeAssetParameters                       *)
(* Args:  -none-                                           *)
(*                                                         *)
(* Desc:  This is a callback method that is invoked by     *)
(*        RMS to notify this module that it is time to     *)
(*        update/synchronize this asset parameter values   *)
(*        with RMS.                                        *)
(*                                                         *)
(*        This method should not be invoked/called         *)
(*        by any user implementation code.                 *)
(***********************************************************)
DEFINE_FUNCTION SynchronizeAssetParameters(RmsAsset asset)
{
}
```

The *RmsApi.axi* Include File (which is already included by each asset monitoring module) provides the following wrapper function for updating asset parameter values with RMS. These functions queue parameter value changes and only forward them to the RMS server when the Submit method is called. This queuing technique should be used when there are more than 2 parameter updates that occur back-to-back.

- **RmsAssetParameterEnqueueSetValueBoolean** (see page 79)
- **RmsAssetParameterEnqueueSetValueNumber** (see page 79)
- **RmsAssetParameterEnqueueSetValueDecimal** (see page 79)
- **RmsAssetParameterEnqueueSetValueLevel** (see page 79)
- **RmsAssetParameterEnqueueSetValue** (see page 80)
- **RmsAssetParameterEnqueueUpdateValue** (see page 80)
- **RmsAssetParameterUpdatesSubmit** (see page 81)

The following code snippet provides an example of queuing asset parameter value updates and sending them in a final submit call.

```
// battery level
RmsAssetParameterEnqueueSetValueLevel(assetClientKey,'touch.panel.battery.level',panelInfo.batteryLevel);
// battery charging
RmsAssetParameterEnqueueSetValueBoolean(assetClientKey,'touch.panel.battery.charging',panelInfo.charging);
// submit all the pending parameter updates now
RmsAssetParameterUpdatesSubmit(assetClientKey);
```

Registering Asset Metadata Properties

The following callback method will be invoked in each asset monitoring module when it is time to perform the Asset metadata registration with RMS.

```
(***********************************************************)
(* Name:  RegisterAssetMetadata                            *)
(* Args:  -none-                                           *)
(*                                                         *)
(* Desc:  This is a callback method that is invoked by     *)
(*        RMS to notify this module that it is time to     *)
(*        register this asset's metadata properties with   *)
(*        RMS.                                             *)
(*                                                         *)
(*        This method should not be invoked/called         *)
(*        by any user implementation code.                 *)
(***********************************************************)
DEFINE_FUNCTION RegisterAssetMetadata()
{
}
```

The *RmsApi.axi* Include File (which is already included by each asset monitoring module) provides the following wrapper function for registering asset metadata properties with RMS.

- **RmsAssetMetadataEnqueueString** (see page 83)
- **RmsAssetMetadataEnqueueBoolean** (see page 84)
- **RmsAssetMetadataEnqueueNumber** (see page 85)
- **RmsAssetMetadataEnqueueDecimal** (see page 86)
- **RmsAssetMetadataEnqueueHyperlink** (see page 87)
- **RmsAssetMetadataEnqueue** (see page 88)
- **RmsAssetMetadataSubmit** (see page 88)

Each asset parameter metadata property requires the following information:

Asset Metadata Properties - Required Information	
Key (String)	This is a unique identifier for this asset metadata property. This ID string must be uniquely scoped for this asset. Metadata property value updates will use this key identifier when operating on this metadata property.
Name (String)	Friendly name for this asset metadata property.
Value (String)	Value field for this asset metadata property. This is required for all metadata property types except HYPERLINK.

Data Type (String)	Data type for this asset metadata property value. The following set of constants are defined for the available data types in the *RmsApi.axi* Include File.
	`// RMS Metadata Property Data Types`
	`RMS_METADATA_TYPE_STRING = 'STRING';`
	`RMS_METADATA_TYPE_MEMO = 'MEMO';`
	`RMS_METADATA_TYPE_BOOLEAN = 'BOOLEAN';`
	`RMS_METADATA_TYPE_NUMBER = 'NUMBER';`
	`RMS_METADATA_TYPE_DECIMAL = 'DECIMAL';`
	`RMS_METADATA_TYPE_DATE = 'DATE';`
	`RMS_METADATA_TYPE_TIME = 'TIME';`
	`RMS_METADATA_TYPE_HYPERLINK = 'HYPERLINK';`
	`RMS_METADATA_TYPE_DATETIME = 'DATETIME';`

Read Only (Boolean)	Defines if this asset parameter threshold is enabled by default.
Hyperlink Name (String)	This metadata property attribute is only required when defining a metadata property of data type: Hyperlink. This field should contain the hyperlink display name/text.
Hyperlink URL (String)	This metadata property attribute is only required when defining a metadata property of data type: Hyperlink. This field should contain the hyperlink URL address.

The following code snippet provides an example of registering asset metadata properties. A final submission method call is required to submit the queued asset metadata properties to the RMS server.

```
// this is a new asset registration, register all asset metadata properties now.
RmsAssetMetadataEnqueueBoolean(assetClientKey,'touch.panel.g4','G4 Enabled',panelInfo.isG4Panel);

RmsAssetMetadataEnqueueBoolean(assetClientKey,'touch.panel.headless','Headless',panelInfo.isHeadlessPanel);

RmsAssetMetadataEnqueueBoolean(assetClientKey,'touch.panel.dockable','Dockable',panelInfo.hasDock);

RmsAssetMetadataEnqueueBoolean(assetClientKey,'touch.panel.battery','Battery',panelInfo.hasBattery);

// submit metadata for registration now
RmsAssetMetadataSubmit(assetClientKey);
```

Asset Metadata Registration Functions

The *Asset Metadata Registration* functions in the *RmsApi.axi* Include File are described in the following table:

Asset Metadata Registration Functions	
RmsAssetMetadata EnqueueString	*Description*: This function is used to place an asset metadata property registration in queue. This function registers a metadata property of type: STRING *Arguments*: • **CHAR assetClientKey[]** - asset client key • **CHAR metadataKey[]** - metadata property key • **CHAR metadataName[]** - metadata property name • **CHAR metadataValue[]** - metadata property value *Returns*: 1 if call was successful; 0 if call was unsuccessful *Syntax*:

```
DEFINE_FUNCTION CHAR RmsAssetMetadataEnqueueString(CHAR assetClientKey[],CHAR metadataKey[],CHAR metadataName[],
CHAR metadataValue[])
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataEnqueueString> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a metadata key has been provided
  IF(metadataKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataEnqueueString> :: missing metadata key';
    RETURN FALSE;
  }

  // ensure a metadata name has been provided
  IF(metadataName == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataEnqueueString> :: missing metadata name';
    RETURN FALSE;
  }

  // submit the asset metadata registration now
  rmsCommand = RmsPackCmdHeader('ASSET.METADATA');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataName);
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataValue);
  rmsCommand = RmsPackCmdParam(rmsCommand,RMS_METADATA_TYPE_STRING);
  rmsCommand = RmsPackCmdParam(rmsCommand,'true'); // read-only
  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

Asset Metadata Registration Functions (Cont.)	
RmsAssetMetadata EnqueueBoolean	*Description*: This function is used to place an asset metadata property registration in queue. This function registers a metadata property of type: BOOLEAN *Arguments*: • **CHAR assetClientKey[]** - asset client key • **CHAR metadataKey[]** - metadata property key • **CHAR metadataName[]** - metadata property name • **CHAR metadataValue** - metadata property value *Returns*: 1 if call was successful; 0 if call was unsuccessful *Syntax*:

```
DEFINE_FUNCTION CHAR RmsAssetMetadataEnqueueBoolean(CHAR assetClientKey[],CHAR metadataKey[],CHAR metadataName[],
                                                    CHAR metadataValue)
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataEnqueueBoolean> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a metadata key has been provided
  IF(metadataKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataEnqueueBoolean> :: missing metadata key';
    RETURN FALSE;
  }

  // ensure a metadata name has been provided
  IF(metadataName == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataEnqueueBoolean> :: missing metadata name';
    RETURN FALSE;
  }

  // submit the asset metadata registration now
  rmsCommand = RmsPackCmdHeader('ASSET.METADATA');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataName);
  rmsCommand = RmsPackCmdParam(rmsCommand,RmsBooleanString(metadataValue));
  rmsCommand = RmsPackCmdParam(rmsCommand,RMS_METADATA_TYPE_BOOLEAN);
  rmsCommand = RmsPackCmdParam(rmsCommand,'true'); // read-only
  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

Asset Metadata Registration Functions (Cont.)	
RmsAssetMetadata EnqueueNumber	*Description*: This function is used to place an asset metadata property registration in queue. This function registers a metadata property of type: NUMBER *Arguments*: • **CHAR assetClientKey[]** - asset client key • **CHAR metadataKey[]** - metadata property key • **CHAR metadataName[]** - metadata property name • **SLONG metadataValue** - metadata property value *Returns*: 1 if call was successful; 0 if call was unsuccessful *Syntax*:

```
DEFINE_FUNCTION CHAR RmsAssetMetadataEnqueueNumber(CHAR assetClientKey[],CHAR metadataKey[],CHAR metadataName[],
                                                   SLONG metadataValue)
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataEnqueueNumber> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a metadata key has been provided
  IF(metadataKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataEnqueueNumber> :: missing metadata key';
    RETURN FALSE;
  }

  // ensure a metadata name has been provided
  IF(metadataName == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataEnqueueNumber> :: missing metadata name';
    RETURN FALSE;
  }

  // submit the asset metadata registration now
  rmsCommand = RmsPackCmdHeader('ASSET.METADATA');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataName);
  rmsCommand = RmsPackCmdParam(rmsCommand,ITOA(metadataValue));
  rmsCommand = RmsPackCmdParam(rmsCommand,RMS_METADATA_TYPE_NUMBER);
  rmsCommand = RmsPackCmdParam(rmsCommand,'true'); // read-only
  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

Asset Metadata Registration Functions (Cont.)

RmsAssetMetadataEnqueueDecimal

Description: This function is used to place an asset metadata property registration in queue.
This function registers a metadata property of type: DECIMAL
Arguments:

- **CHAR assetClientKey[]** - asset client key
- **CHAR metadataKey[]** - metadata property key
- **CHAR metadataName[]** - metadata property name
- **DOUBLE metadataValue** - metadata property value

Returns: 1 if call was successful; 0 if call was unsuccessful
Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetMetadataEnqueueDecimal(CHAR assetClientKey[],CHAR metadataKey[],CHAR metadataName[],
                                                    DOUBLE metadataValue)
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataEnqueueDecimal> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a metadata key has been provided
  IF(metadataKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataEnqueueDecimal> :: missing metadata key';
    RETURN FALSE;
  }

  // ensure a metadata name has been provided
  IF(metadataName == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataEnqueueDecimal> :: missing metadata name';
    RETURN FALSE;
  }

  // submit the asset metadata registration now
  rmsCommand = RmsPackCmdHeader('ASSET.METADATA');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataName);
  rmsCommand = RmsPackCmdParam(rmsCommand,FTOA(metadataValue));
  rmsCommand = RmsPackCmdParam(rmsCommand,RMS_METADATA_TYPE_DECIMAL);
  rmsCommand = RmsPackCmdParam(rmsCommand,'true'); // read-only
  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

Asset Metadata Registration Functions (Cont.)

RmsAssetMetadataEnqueueHyperlink

Description: This function is used to place an asset metadata property registration in queue.
This function registers a metadata property of type: HYPERLINK
Arguments:

- **CHAR assetClientKey[]** - asset client key
- **CHAR metadataKey[]** - metadata property key
- **CHAR metadataName[]** - metadata property name
- **CHAR hyperlinkName[]** - metadata hyperlink name
- **CHAR hyperlinUrl[]** - metadata hyperlink address

Returns: 1 if call was successful; 0 if call was unsuccessful
Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetMetadataEnqueueHyperlink(CHAR assetClientKey[],CHAR metadataKey[],CHAR metadataName[],
                                                      CHAR metadataHyperlinkName[],CHAR metadataHyperlinkUrl[])
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataEnqueueHyperlink> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a metadata key has been provided
  IF(metadataKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataEnqueueHyperlink> :: missing metadata key';
    RETURN FALSE;
  }

  // ensure a metadata name has been provided
  IF(metadataName == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataEnqueueHyperlink> :: missing metadata name';
    RETURN FALSE;
  }

  // submit the asset metadata registration now
  rmsCommand = RmsPackCmdHeader('ASSET.METADATA');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataName);
  rmsCommand = RmsPackCmdParam(rmsCommand,''); // no value data (hyperlink)
  rmsCommand = RmsPackCmdParam(rmsCommand,RMS_METADATA_TYPE_HYPERLINK);
  rmsCommand = RmsPackCmdParam(rmsCommand,'true'); // read-only
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataHyperlinkName);
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataHyperlinkUrl);
  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

Asset Metadata Registration Functions (Cont.)

RmsAssetMetadata Enqueue

Description: This function is used to place an asset metadata property registration in queue.
This function registers an asset metadata property defined by the *RmsAssetMetadataProperty* data structure argument.
Arguments:

- **CHAR assetClientKey[]** - asset client key
- **RmsAssetMetadataProperty** - metadata Property

Returns: 1 if call was successful; 0 if call was unsuccessful
Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetMetadataEnqueue(CHAR assetClientKey[],RmsAssetMetadataProperty
                                             metadataProperty)
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataEnqueue> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a metadata key has been provided
  IF(metadataProperty.key == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataEnqueue> :: missing metadata key';
    RETURN FALSE;
  }

  // ensure a metadata name has been provided
  IF(metadataProperty.name == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataEnqueue> :: missing metadata name';
    RETURN FALSE;
  }

  // ensure a metadata property data type is assigned
  IF(metadataProperty.dataType == '')
    metadataProperty.dataType = RMS_METADATA_TYPE_STRING;

  // submit the asset metadata registration now
  rmsCommand = RmsPackCmdHeader('ASSET.METADATA');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataProperty.key);
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataProperty.name);
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataProperty.value);
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataProperty.dataType);
  rmsCommand = RmsPackCmdParam(rmsCommand,RmsBooleanString(metadataProperty.readOnly));
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataProperty.hyperlinkName);
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataProperty.hyperlinkUrl);
  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

RmsAssetMetadata Submit

Description: This function is used to submit any pending asset metadata properties that are currently in queue waiting to be registered with RMS.
Arguments:

- **CHAR assetClientKey[]** - asset client key

Returns: 1 if call was successful; 0 if call was unsuccessful
Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetMetadataSubmit(CHAR assetClientKey[])
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure RMS is ONLINE, REGISTERED, and ready for ASSET registration
  IF(![vdvRMS,RMS_CHANNEL_ASSETS_REGISTER])
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataSubmit> :: RMS is not ready to accept asset metadata
    changes.';
    RETURN FALSE;
  }

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataSubmit> :: missing asset client key';
    RETURN FALSE;
  }

  // submit the asset registration now
  rmsCommand = RmsPackCmdHeader('ASSET.METADATA.SUBMIT');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

Asset Metadata Registration Functions (Cont.)

RmsAssetMetadataDelete

Description: This function is used to delete an existing asset metadata property from the RMS server.

Arguments:

- **CHAR assetClientKey[]** - asset client key
- **CHAR metadataKey[]** - metadata property key

Returns:

1 if call was successful; 0 if call was unsuccessful

Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetMetadataDelete(CHAR assetClientKey[], CHAR metadataKey[])
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure RMS is ONLINE, REGISTERED, and ready for ASSET registration
  IF(![vdvRMS,RMS_CHANNEL_ASSETS_REGISTER])
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataDelete> :: RMS is not ready to accept asset metadata
    changes.';
    RETURN FALSE;
  }

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataDelete> :: missing asset client key';
    RETURN FALSE;
  }

  // submit the asset registration now
  rmsCommand = RmsPackCmdHeader('ASSET.METADATA.DELETE');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataKey);
  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

Synchronizing Asset Metadata Properties

The following callback method will be invoked in each asset monitoring module when it is time to perform the Asset metadata synchronization with RMS.

```
(***********************************************************)
(* Name:  SynchronizeAssetMetadata                         *)
(* Args:  -none-                                           *)
(*                                                         *)
(* Desc:  This is a callback method that is invoked by     *)
(*        RMS to notify this module that it is time to     *)
(*        update/synchronize this asset metadata properties*)
(*        with RMS if needed.                              *)
(*                                                         *)
(*        This method should not be invoked/called         *)
(*        by any user implementation code.                 *)
(***********************************************************)
DEFINE_FUNCTION SynchronizeAssetMetadata ()
{
}
```

Traditionally, asset metadata properties are relatively static information and thus do not require any synchronization of values after registration. However, this callback method does provide the opportunity to perform any necessary metadata updates if your implementation does include any dynamic metadata property values.

The *RmsApi.axi* Include File (which is already included by each asset monitoring module) provides the following wrapper function for updating asset metadata property values with RMS.

- **RmsAssetMetadataUpdateString** (see page 90)
- **RmsAssetMetadataUpdateBoolean** (see page 90)
- **RmsAssetMetadataUpdateNumber** (see page 91)
- **RmsAssetMetadataUpdateDecimal** (see page 91)
- **RmsAssetMetadataUpdateHyperlink** (see page 92)
- **RmsAssetMetadataUpdateValue** (see page 93)

The following code snippet provides an example of updating asset metadata properties.

```
// this is a new asset registration, register all
RmsAssetMetadataUpdateNumber(assetClientKey,'touch.panel.display.timeout',panelInfo.displayTimeout);

RmsAssetMetadataUpdateNumber(assetClientKey,'touch.panel.shutdown.timeout',panelInfo.shutdownTimeout);

RmsAssetMetadataUpdateString(assetClientKey,'touch.panel.file.system',panelInfo.fileSystem);
```

Asset Metadata Update Functions

The *Asset Metadata Update* functions in the *RmsApi.axi* Include File are described in the following table:

Asset Metadata Update Functions

RmsAssetMetadata UpdateString

Description: This function is used to update and existing asset metadata property value.
This function updates a metadata property of type: STRING
Arguments:

- **CHAR assetClientKey[]** - asset client key
- **CHAR metadataKey[]** - metadata property key
- **CHAR metadataValue[]** - metadata property value

Returns: 1 if call was successful; 0 if call was unsuccessful
Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetMetadataUpdateString(CHAR assetClientKey[],CHAR metadataKey[],CHAR metadataValue[])
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure RMS is ONLINE, REGISTERED, and ready for ASSET registration
  IF(![vdvRMS,RMS_CHANNEL_ASSETS_REGISTER])
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataUpdateString> :: RMS is not ready to accept asset
    metadata changes.';
    RETURN FALSE;
  }

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataUpdateString> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a metadata key has been provided
  IF(metadataKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataUpdateString> :: missing metadata key';
    RETURN FALSE;
  }

  // submit the asset metadata update now
  RETURN RmsAssetMetadataUpdateValue(assetClientKey,metadataKey,metadataValue);
}
```

RmsAssetMetadata UpdateBoolean

Description: This function is used to update and existing asset metadata property value.
This function updates a metadata property of type: BOOLEAN
Arguments:

- **CHAR assetClientKey[]** - asset client key
- **CHAR metadataKey[]** - metadata property key
- **CHAR metadataValue** - metadata property value

Returns: 1 if call was successful; 0 if call was unsuccessful
Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetMetadataUpdateBoolean(CHAR assetClientKey[],CHAR metadataKey[],CHAR metadataValue)
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure RMS is ONLINE, REGISTERED, and ready for ASSET registration
  IF(![vdvRMS,RMS_CHANNEL_ASSETS_REGISTER])
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataUpdateBoolean> :: RMS is not ready to accept asset metadata
changes.';
    RETURN FALSE;
  }

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataUpdateBoolean> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a metadata key has been provided
  IF(metadataKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataUpdateBoolean> :: missing metadata key';
    RETURN FALSE;
  }

  // submit the asset metadata update now
  RETURN RmsAssetMetadataUpdateValue(assetClientKey,metadataKey,RmsBooleanString(metadataValue));
}
```

Asset Metadata Update Functions (Cont.)

RmsAssetMetadata UpdateNumber

Description: This function is used to update and existing asset metadata property value.
This function updates a metadata property of type: NUMBER
Arguments:

- **CHAR assetClientKey[]** - asset client key
- **CHAR metadataKey[]** - metadata property key
- **SLONG metadataValue** - metadata property value

Returns: 1 if call was successful; 0 if call was unsuccessful
Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetMetadataUpdateNumber(CHAR assetClientKey[],CHAR metadataKey[],SLONG metadataValue)
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure RMS is ONLINE, REGISTERED, and ready for ASSET registration
  IF(![vdvRMS,RMS_CHANNEL_ASSETS_REGISTER])
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataUpdateNumber> :: RMS is not ready to accept asset metadata
changes.';
    RETURN FALSE;
  }

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataUpdateNumber> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a metadata key has been provided
  IF(metadataKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataUpdateNumber> :: missing metadata key';
    RETURN FALSE;
  }

  // submit the asset metadata update now
  RETURN RmsAssetMetadataUpdateValue(assetClientKey,metadataKey,
  ITOA (metadataValue));
}
```

RmsAssetMetadata UpdateDecimal

Description: This function is used to update and existing asset metadata property value.
This function updates a metadata property of type: DECIMAL
Arguments:

- **CHAR assetClientKey[]** - asset client key
- **CHAR metadataKey[]** - metadata property key
- **DOUBLE metadataValue -** metadata property value

Returns: 1 if call was successful; 0 if call was unsuccessful
Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetMetadataUpdateDecimal(CHAR assetClientKey[],CHAR metadataKey[],DOUBLE metadataValue)
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure RMS is ONLINE, REGISTERED, and ready for ASSET registration
  IF(![vdvRMS,RMS_CHANNEL_ASSETS_REGISTER])
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataUpdateDecimal> :: RMS is not ready to accept asset metadata
changes.';
    RETURN FALSE;
  }

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataUpdateDecimal> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a metadata key has been provided
  IF(metadataKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataUpdateDecimal> :: missing metadata key';
    RETURN FALSE;
  }

  // submit the asset metadata update now
  RETURN RmsAssetMetadataUpdateValue(assetClientKey,metadataKey,FTOA(metadataValue));
}
```

Asset Metadata Update Functions (Cont.)

RmsAssetMetadata UpdateHyperlink

Description: This function is used to update and existing asset metadata property value.
This function updates a metadata property of type: HYPERLINK
Arguments:

- **CHAR assetClientKey[]** - asset client key
- **CHAR metadataKey[]** - metadata property key
- **CHAR hyperlinkName[]** - metadata hyperlink name
- **CHAR hyperlinUrl[]** - metadata hyperlink address

Returns: 1 if call was successful; 0 if call was unsuccessful
Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetMetadataUpdateHyperlink(CHAR assetClientKey[],CHAR metadataKey[],
                                                     CHAR metadataHyperlinkName[],CHAR metadataHyperlinkUrl[])
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure RMS is ONLINE, REGISTERED, and ready for ASSET registration
  IF(![vdvRMS,RMS_CHANNEL_ASSETS_REGISTER])
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataUpdateHyperlink> :: RMS is not ready to accept asset
    metadata changes.';
    RETURN FALSE;
  }

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataUpdateHyperlink> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a metadata key has been provided
  IF(metadataKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataUpdateHyperlink> :: missing metadata key';
    RETURN FALSE;
  }

  // submit the asset metadata update now
  rmsCommand = RmsPackCmdHeader('ASSET.METADATA.UPDATE');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,''); // empty placeholder for value field for other data types
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataHyperlinkName);
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataHyperlinkUrl);
  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

Asset Metadata Update Functions (Cont.)

RmsAssetMetadata UpdateValue

Description: This function is used to update and existing asset metadata property value
This function updates a metadata property of type: HYPERLINK
Arguments:

- **CHAR assetClientKey[]** - asset client key
- **CHAR metadataKey[]** - metadata property key
- **CHAR metadataValue[] -** metadata property value

Returns: 1 if call was successful; 0 if call was unsuccessful
Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetMetadataUpdateValue(CHAR assetClientKey[],CHAR metadataKey[],CHAR metadataValue[])
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure RMS is ONLINE, REGISTERED, and ready for ASSET registration
  IF(![vdvRMS,RMS_CHANNEL_ASSETS_REGISTER])
  {
    //SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataUpdateValue> :: RMS is not ready to accept asset
    metadata changes.';
    RETURN FALSE;
  }

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataUpdateValue> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a metadata key has been provided
  IF(metadataKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataUpdateValue> :: missing metadata key';
    RETURN FALSE;
  }

  // submit the asset metadata update now
  rmsCommand = RmsPackCmdHeader('ASSET.METADATA.UPDATE');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataValue);
  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

Registering Asset Control Methods

The following callback method will be invoked in each asset monitoring module when it is time to perform the asset control method registration with RMS.

```
(***********************************************************)
(* Name:  RegisterAssetControlMethods                      *)
(* Args:  -none-                                           *)
(*                                                         *)
(* Desc:  This is a callback method that is invoked by     *)
(*        RMS to notify this module that it is time to     *)
(*        register this asset's control methods with RMS.  *)
(*                                                         *)
(*        This method should not be invoked/called         *)
(*        by any user implementation code.                 *)
(***********************************************************)
DEFINE_FUNCTION RegisterAssetControlMethods ()
{
}
```

The *RmsApi.axi* Include File (which is already included by each asset monitoring module) provides the following wrapper functions for registering asset control methods and asset control method arguments with RMS:

- **RmsAssetControlMethodEnqueue** (see page 95)
- **RmsAssetControlMethodArgumentString** (see page 96)
- **RmsAssetControlMethodArgumentBoolean** (see page 97)
- **RmsAssetControlMethodArgumentNumber** (see page 98)
- **RmsAssetControlMethodArgumentNumberEx** (see page 99)
- **RmsAssetControlMethodArgumentDecimal** (see page 100)
- **RmsAssetControlMethodArgumentLevel** (see page 101)
- **RmsAssetControlMethodArgumentEnum** (see page 102)
- **RmsAssetControlMethodArgumentEnumEx** (see page 103)
- **RmsAssetControlMethodArgumentEnqueue** (see page 104)

- **RmsAssetControlMethodsSubmit** (see page 105)
- **RmsAssetControlMethodDelete** (see page 105)

Each asset control method requires the following information:

Asset Control Methods - Required Information	
Key (String)	This is a unique identifier for this asset control method. This ID string must be uniquely scoped for this asset. Control method execution request will use this key identifier when operating on this control method.
Name (String)	Friendly name for this asset control method.
Description (String)	Descriptive help text for this asset control method.

Asset Control Method Arguments - Required Information

Each asset control method argument requires the following information:

Asset Control Method Arguments - Required Information	
Ordinal (Integer)	Argument order position for this asset control method argument in the control method. When control methods are executed, the arguments returned will be order by this ordinal index.
Name (String)	Friendly name for this asset control method argument.
Description (String)	Descriptive help text for this asset control method argument.

Data Type (String)	The following control method argument data types are supported for RMS asset control methods:
	String, Boolean, Number, Decimal, Level, Enumeration.
	A set of constants are defined for the available data types in the *RmsApi.axi* Include File.
	`// RMS Control Method Argument Data Types`
	`RMS_METHOD_ARGUMENT_TYPE_NUMBER = 'NUMBER';`
	`RMS_METHOD_ARGUMENT_TYPE_STRING = 'STRING';`
	`RMS_METHOD_ARGUMENT_TYPE_ENUMERATION = 'ENUMERATION';`
	`RMS_METHOD_ARGUMENT_TYPE_LEVEL = 'LEVEL';`
	`RMS_METHOD_ARGUMENT_TYPE_BOOLEAN = 'BOOLEAN';`
	`RMS_METHOD_ARGUMENT_TYPE_DECIMAL = 'DECIMAL';`

Default Value (String) [OPTIONAL]	Provide a default argument value that is preselected for this asset control method argument.
Minimum Value (Signed Long) [OPTIONAL]	This attribute only applies to control method arguments of a numeric data type: *Number*, *Decimal*, *Level*. • It is optional for data types Number and Decimal and required for data type Level. • This attribute specifies the lowest possible value that this argument value can reach.
Maximum Value (Signed Long) [OPTIONAL]	This attribute only applies to control method arguments of a numeric data type: *Number*, *Decimal*, *Level*. • It is optional for data types Number and Decimal and required for data type Level. • This attribute specifies the lowest possible value that this argument value can reach.
Step Value (Integer) [OPTIONAL]	This attribute only applies to control method arguments of a numeric data type: *Number* and *Level*. This attribute specifies the incremental value steps that the RMS UI will enforce when a user clicks UP or DOWN on the numeric value stepper control.
Enumeration (String) [OPTIONAL]	This parameter attribute only applies to and is required for asset control method arguments of data type: *Enumeration*. This attribute should contain a pipe delimited list of string values.

The following code snippet provides an example of registering asset control methods and asset control method arguments:

```
// SETUP
RmsAssetControlMethodEnqueue(assetClientKey,'touch.panel.setup','Enter Setup',
                             'Enter setup configuration pages on touch panel user interface');
// SLEEP
RmsAssetControlMethodEnqueue(assetClientKey,'touch.panel.sleep','Sleep',
                             'Put the touch panel user interface into sleep mode');

// BRIGHTNESS LEVEL
RmsAssetControlMethodEnqueue(assetClientKey,'touch.panel.brightness','Set Brightness Level',
                             'Set the display brightness level on the touch panel user interface');

// BRIGHTNESS LEVEL ARGUMENT
RmsAssetControlMethodArgumentLevel(assetClientKey,'touch.panel.brightness',0,'Brightness Level',
                                   'Available range 1-100', 70, 1, 100, 1);

// MUTE ON/OFF
RmsAssetControlMethodEnqueue(assetClientKey,'touch.panel.volume.mute','Set Volume Mute',
                             'Set the audio mute status on the touch panel user interface');

// MUTE ON/OFF ARGUMENT
RmsAssetControlMethodArgumentBoolean(assetClientKey,'touch.panel.volume.mute',0,'Mute On','Mute ON/OFF',
                                     FALSE);
// when done queuing all asset control methods and arguments for this asset, we just need to submit
// them to finalize and register them with the RMS server
  RmsAssetControlMethodsSubmit(assetClientKey);
```

Asset Control Methods Registration Functions

The *Asset Control Methods Registration* functions in the *RmsApi.axi* Include File are described in the following table:

Asset Control Methods Registration Functions

RmsAssetControlMethod Enqueue

Description: This function is used to place an asset control method registration in queue in the RMS client.

Arguments:

- **CHAR assetClientKey[]** - asset client key
- **CHAR methodKey[]** - control method key
- **CHAR methodName[]** - control method name
- **CHAR methodDescription[]** - control method description

Returns: 1 if call was successful; 0 if call was unsuccessful

Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetControlMethodEnqueue(CHAR assetClientKey[],CHAR methodKey[],CHAR methodName[],
                                                  CHAR methodDescription[])
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodEnqueue> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a control method key has been provided
  IF(methodKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodEnqueue> :: missing control method key';
    RETURN FALSE;
  }

  // ensure a control method key has been provided
  IF(methodName == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodEnqueue> :: missing control method name';
    RETURN FALSE;
  }

  // enqueue asset control method for registration
  rmsCommand = RmsPackCmdHeader('ASSET.METHOD');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,methodKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,methodName);
  rmsCommand = RmsPackCmdParam(rmsCommand,methodDescription);
  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

Asset Control Methods Registration Functions (Cont.)	
RmsAssetControlMethod ArgumentString	*Description*: This function is used to add an asset control method argument to an asset control method registration that is currently in queue and has not yet been submitted to the RMS server. The asset control method argument being added is of type: STRING *Arguments*: see method signature below *Returns*: 1 if call was successful; 0 if call was unsuccessful *Syntax*:

```
DEFINE_FUNCTION CHAR RmsAssetControlMethodArgumentString(CHAR assetClientKey[],CHAR methodKey[],
                                                         INTEGER argumentOrdinal,CHAR argumentName[],
                                                         CHAR argumentDescription[],
                                                         CHAR argumentDefaultValue[])
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentString> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a control method key has been provided
  IF(methodKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentString> :: missing control method key';
    RETURN FALSE;
  }

  // ensure a control method argument has been provided
  IF(argumentName == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentString> :: missing control method
                    argument name';
    RETURN FALSE;
  }

  // submit the asset registration now
  rmsCommand = RmsPackCmdHeader('ASSET.METHOD.ARGUMENT.STRING');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,methodKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,ITOA(argumentOrdinal));
  rmsCommand = RmsPackCmdParam(rmsCommand,argumentName);
  rmsCommand = RmsPackCmdParam(rmsCommand,argumentDescription);
  rmsCommand = RmsPackCmdParam(rmsCommand,argumentDefaultValue);
  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

Asset Control Methods Registration Functions (Cont.)

RmsAssetControlMethod ArgumentBoolean

Description: This function is used to add an asset control method argument to an asset control method registration that is currently in queue and has not yet been submitted to the RMS server.
The asset control method argument being added is of type: BOOLEAN
Arguments: see method signature below
Returns: 1 if call was successful; 0 if call was unsuccessful
Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetControlMethodArgumentBoolean(CHAR assetClientKey[],CHAR methodKey[],
                                                          INTEGER argumentOrdinal,CHAR argumentName[],
                                                          CHAR argumentDescription[],
                                                          CHAR argumentDefaultValue)
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentBoolean> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a control method key has been provided
  IF(methodKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentBoolean> :: missing control method key';
    RETURN FALSE;
  }

  // ensure a control method argument has been provided
  IF(argumentName == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentBoolean> :: missing control method
                    argument name';
    RETURN FALSE;
  }

  // submit the asset registration now
  rmsCommand = RmsPackCmdHeader('ASSET.METHOD.ARGUMENT.BOOLEAN');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,methodKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,ITOA(argumentOrdinal));
  rmsCommand = RmsPackCmdParam(rmsCommand,argumentName);
  rmsCommand = RmsPackCmdParam(rmsCommand,argumentDescription);
  rmsCommand = RmsPackCmdParam(rmsCommand,RmsBooleanString(argumentDefaultValue));
  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

Asset Control Methods Registration Functions (Cont.)	
RmsAssetControlMethod ArgumentNumber	*Description*: This function is used to add an asset control method argument to an asset control method registration that is currently in queue and has not yet been submitted to the RMS server. The asset control method argument being added is of type: NUMBER *Arguments*: see method signature below *Returns*: 1 if call was successful; 0 if call was unsuccessful *Syntax*:

```
DEFINE_FUNCTION CHAR RmsAssetControlMethodArgumentNumber( CHAR assetClientKey[],CHAR methodKey[],
                                                          INTEGER argumentOrdinal, CHAR argumentName[],
                                                          CHAR argumentDescription[],SLONG argumentDefaultValue)
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentNumber> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a control method key has been provided
  IF(methodKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentNumber> :: missing control method key';
    RETURN FALSE;
  }

  // ensure a control method argument has been provided
  IF(argumentName == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentNumber> :: missing control method
                    argument name';
    RETURN FALSE;
  }

  // submit the asset registration now
  rmsCommand = RmsPackCmdHeader('ASSET.METHOD.ARGUMENT.NUMBER');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,methodKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,ITOA(argumentOrdinal));
  rmsCommand = RmsPackCmdParam(rmsCommand,argumentName);
  rmsCommand = RmsPackCmdParam(rmsCommand,argumentDescription);
  rmsCommand = RmsPackCmdParam(rmsCommand,ITOA(argumentDefaultValue));
  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

Asset Control Methods Registration Functions (Cont.)	

RmsAssetControlMethod ArgumentNumberEx	*Description*: This function is used to add an asset control method argument to an asset control method registration that is currently in queue and has not yet been submitted to the RMS server. This EXTENDED function provides the additional arguments to provide min, max, and step values.
	The asset control method argument being added is of type: NUMBER
	Arguments: see method signature below
	Returns: 1 if call was successful; 0 if call was unsuccessful
	Syntax:
	`DEFINE_FUNCTION CHAR RmsAssetControlMethodArgumentNumberEx( CHAR assetClientKey[],CHAR methodKey[], INTEGER argumentOrdinal,CHAR argumentName[], CHAR argumentDescription[],SLONG argumentDefaultValue, SLONG argumentMinimumValue,SLONG argumentMaximumValue, INTEGER argumentStepValue)`
	`{`
	`  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];`
	
	`  // ensure an asset client key has been provided`
	`  IF(assetClientKey == '')`
	`  {`
	`    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentNumberEx> :: missing asset client key';`
	`    RETURN FALSE;`
	`  }`
	
	`  // ensure a control method key has been provided`
	`  IF(methodKey == '')`
	`  {`
	`    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentNumberEx> :: missing control method key';`
	`    RETURN FALSE;`
	`  }`
	
	`  // ensure a control method argument has been provided`
	`  IF(argumentName == '')`
	`  {`
	`    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentNumberEx> :: missing control method argument name';`
	`    RETURN FALSE;`
	`  }`
	
	`  // submit the asset registration now`
	`  rmsCommand = RmsPackCmdHeader('ASSET.METHOD.ARGUMENT.NUMBER');`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,methodKey);`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,ITOA(argumentOrdinal));`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,argumentName);`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,argumentDescription);`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,ITOA(argumentDefaultValue));`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,ITOA(argumentMinimumValue));`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,ITOA(argumentMaximumValue));`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,ITOA(argumentStepValue));`
	`  SEND_COMMAND vdvRMS, rmsCommand;`
	
	`  RETURN TRUE;`
	`}`

Asset Control Methods Registration Functions (Cont.)	

RmsAssetControlMethod ArgumentDecimal	*Description*: This function is used to add an asset control method argument to an asset control method registration that is currently in queue and has not yet been submitted to the RMS server.
	The asset control method argument being added is of type: DECIMAL
	Arguments: see method signature below
	Returns: 1 if call was successful; 0 if call was unsuccessful
	Syntax:
	`DEFINE_FUNCTION CHAR RmsAssetControlMethodArgumentDecimal(CHAR assetClientKey[],CHAR methodKey[], INTEGER argumentOrdinal,CHAR argumentName[], CHAR argumentDescription[], DOUBLE argumentDefaultValue)`
	`{`
	`  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];`
	
	`  // ensure an asset client key has been provided`
	`  IF(assetClientKey == '')`
	`  {`
	`  SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentDecimal> :: missing asset client key';`
	`  RETURN FALSE;`
	`  }`
	
	`  // ensure a control method key has been provided`
	`  IF(methodKey == '')`
	`  {`
	`  SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentDecimal> :: missing control method key';`
	`  RETURN FALSE;`
	`  }`
	
	`  // ensure a control method argument has been provided`
	`  IF(argumentName == '')`
	`  {`
	`  SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentDecimal> :: missing control method argument name';`
	`  RETURN FALSE;`
	`  }`
	
	`  // submit the asset registration now`
	`  rmsCommand = RmsPackCmdHeader('ASSET.METHOD.ARGUMENT.DECIMAL');`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,methodKey);`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,ITOA(argumentOrdinal));`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,argumentName);`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,argumentDescription);`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,FTOA(argumentDefaultValue));`
	`  SEND_COMMAND vdvRMS, rmsCommand;`
	
	`  RETURN TRUE;`
	`}`

Asset Control Methods Registration Functions (Cont.)	
RmsAssetControlMethod ArgumentLevel	*Description*: This function is used to add an asset control method argument to an asset control method registration that is currently in queue and has not yet been submitted to the RMS server. The asset control method argument being added is of type: LEVEL. *Arguments*: see method signature below *Returns*: 1 if call was successful; 0 if call was unsuccessful *Syntax*:

```
DEFINE_FUNCTION CHAR RmsAssetControlMethodArgumentLevel(CHAR assetClientKey[],CHAR methodKey[],
                                                        INTEGER argumentOrdinal,CHAR argumentName[],
                                                        CHAR argumentDescription[],
                                                        SLONG argumentDefaultValue,
                                                        SLONG argumentMinimumValue,
                                                        SLONG argumentMaximumValue,
                                                        INTEGER argumentStepValue)
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];


  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentLevel> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a control method key has been provided
  IF(methodKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentLevel> :: missing control method key';
    RETURN FALSE;
  }

  // ensure a control method argument has been provided
  IF(argumentName == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentLevel> :: missing control method
                    argument name';
    RETURN FALSE;
  }

  // submit the asset registration now
  rmsCommand = RmsPackCmdHeader('ASSET.METHOD.ARGUMENT.LEVEL');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,methodKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,ITOA(argumentOrdinal));
  rmsCommand = RmsPackCmdParam(rmsCommand,argumentName);
  rmsCommand = RmsPackCmdParam(rmsCommand,argumentDescription);
  rmsCommand = RmsPackCmdParam(rmsCommand,ITOA(argumentDefaultValue));
  rmsCommand = RmsPackCmdParam(rmsCommand,ITOA(argumentMinimumValue));
  rmsCommand = RmsPackCmdParam(rmsCommand,ITOA(argumentMaximumValue));
  rmsCommand = RmsPackCmdParam(rmsCommand,ITOA(argumentStepValue));
  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

Asset Control Methods Registration Functions (Cont.)

RmsAssetControlMethod ArgumentEnum

Description: This function is used to add an asset control method argument to an asset control method registration that is currently in queue and has not yet been submitted to the RMS server. The asset control method argument being added is of type: ENUMERATION. Enumeration is provided as a pipe '|' separated list of strings.

Arguments: see method signature below

Returns: 1 if call was successful; 0 if call was unsuccessful

Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetControlMethodArgumentEnum(CHAR assetClientKey[],CHAR methodKey[],
                                                       INTEGER argumentOrdinal,CHAR argumentName[],
                                                       CHAR argumentDescription[],
                                                       CHAR argumentDefaultValue[],
                                                       CHAR argumentEnumerationValues[])
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];
            INTEGER index;

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentEnum> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a control method key has been provided
  IF(methodKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentEnum> :: missing control method key';
    RETURN FALSE;
  }

  // ensure a control method argument has been provided
  IF(argumentName == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentEnum> :: missing control method
                    argument name';
    RETURN FALSE;
  }

  // submit the asset registration now
  rmsCommand = RmsPackCmdHeader('ASSET.METHOD.ARGUMENT.ENUM');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,methodKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,ITOA(argumentOrdinal));
  rmsCommand = RmsPackCmdParam(rmsCommand,argumentName);
  rmsCommand = RmsPackCmdParam(rmsCommand,argumentDescription);
  rmsCommand = RmsPackCmdParam(rmsCommand,argumentDefaultValue);
  rmsCommand = RmsPackCmdParam(rmsCommand,argumentEnumerationValues);

  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

Asset Control Methods Registration Functions (Cont.)	

RmsAssetControlMethod ArgumentEnumEx	*Description*: This function is used to add an asset control method argument to an asset control method registration that is currently in queue and has not yet been submitted to the RMS server. The asset control method argument being added is of type: ENUMERATION. This EXTENDED function support enumeration values provided as a multi-dimensional array.
	Arguments: see method signature below
	Returns: 1 if call was successful; 0 if call was unsuccessful
	Syntax:
	`DEFINE_FUNCTION CHAR RmsAssetControlMethodArgumentEnumEx(CHAR assetClientKey[],CHAR methodKey[], INTEGER argumentOrdinal,CHAR argumentName[], CHAR argumentDescription[], CHAR argumentDefaultValue[], CHAR argumentEnumerationValues[][])`
	`{`
	`  STACK_VAR CHAR rmsEnumValues[RMS_MAX_CMD_LEN];`
	`  INTEGER index;`
	
	`  // ensure an asset client key has been provided`
	`  IF(assetClientKey == '')`
	`  {`
	`  SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentEnumEx> :: missing asset client key';`
	`  RETURN FALSE;`
	`  }`
	
	`  // ensure a control method key has been provided`
	`  IF(methodKey == '')`
	`  {`
	`  SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentEnumEx> :: missing control method key';`
	`  RETURN FALSE;`
	`  }`
	
	`  // ensure a control method argument has been provided`
	`  IF(argumentName == '')`
	`  {`
	`  SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentEnumEx> :: missing control method argument name';`
	`  RETURN FALSE;`
	`  }`
	
	`  // convert array of enumeration values into pipe separated string`
	`  FOR(index = 1; index <= LENGTH_ARRAY(argumentEnumerationValues); index++)`
	`  {`
	`  rmsEnumValues = "rmsEnumValues,argumentEnumerationValues[index],'|'";`
	`  }`
	
	`  IF(LENGTH_STRING(rmsEnumValues))`
	`  SET_LENGTH_STRING(rmsEnumValues,LENGTH_STRING(rmsEnumValues)-1)`
	
	`// Add pipe parsing and call up`
	`  RmsAssetControlMethodArgumentEnum(assetClientKey, methodKey, argumentOrdinal, argumentName, argumentDescription, argumentDefaultValue, rmsEnumValues)`
	
	`  RETURN TRUE;`
	`}`

Asset Control Methods Registration Functions (Cont.)	
RmsAssetControlMethod ArgumentEnqueue	*Description*: This function is used to add an asset control method argument to an asset control method registration that is currently in queue and has not yet been submitted to the RMS server. This method accepts a *RmsAssetControlMethodArgument* data type argument. *Arguments*: • **CHAR assetClientKey[]** - asset client key • **CHAR methodKey[]** - control method key • **RmsAssetControlMethodArgument** argument *Returns*: 1 if call was successful; 0 if call was unsuccessful *Syntax*:

```
DEFINE_FUNCTION CHAR RmsAssetControlMethodArgumentEnqueue(CHAR assetClientKey[],CHAR methodKey[],
                                                          RmsAssetControlMethodArgument argument)
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];
            INTEGER index;

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentEnqueue> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a control method key has been provided
  IF(methodKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentEnqueue> :: missing control method key';
    RETURN FALSE;
  }

  // ensure a control method argument has been provided
  IF(argument.name == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodArgumentEnqueue> :: missing control method
                    argument name';
    RETURN FALSE;
  }

  // ensure a control method argument data type is assigned
  IF(argument.dataType == '')
    argument.dataType = RMS_METHOD_ARGUMENT_TYPE_STRING;

  // submit the asset registration now
  rmsCommand = RmsPackCmdHeader('ASSET.METHOD.ARGUMENT');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,methodKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,ITOA(argument.ordinal));
  rmsCommand = RmsPackCmdParam(rmsCommand,argument.name);
  rmsCommand = RmsPackCmdParam(rmsCommand,argument.description);
  rmsCommand = RmsPackCmdParam(rmsCommand,ITOA(argument.dataType));
  rmsCommand = RmsPackCmdParam(rmsCommand,argument.defaultValue);
  rmsCommand = RmsPackCmdParam(rmsCommand,ITOA(argument.minimumValue));
  rmsCommand = RmsPackCmdParam(rmsCommand,ITOA(argument.maximumValue));
  rmsCommand = RmsPackCmdParam(rmsCommand,ITOA(argument.stepValue));

  FOR(index = 1; index <= LENGTH_ARRAY(argument.enumerationValues); index++)
  {
    rmsCommand = RmsPackCmdParam(rmsCommand,argument.enumerationValues[index]);
  }

  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

Asset Control Methods Registration Functions (Cont.)

RmsAssetControlMethods Submit

Description:
This function is used to submit any pending asset control methods that are currently in queue waiting to be registered with RMS.
Arguments:
- **CHAR assetClientKey[]** - asset client key

Returns: 1 if call was successful; 0 if call was unsuccessful
Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetControlMethodsSubmit(CHAR assetClientKey[])
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure RMS is ONLINE, REGISTERED, and ready for ASSET registration
  IF(![vdvRMS,RMS_CHANNEL_ASSETS_REGISTER])
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodsSubmit> :: RMS is not ready to accept
    asset control method changes.';
    RETURN FALSE;
  }

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodsSubmit> :: missing asset client key';
    RETURN FALSE;
  }

  // submit the pended queued asset registrations now
  rmsCommand = RmsPackCmdHeader('ASSET.METHOD.SUBMIT');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

RmsAssetControlMethod Delete

Description: This function is used to delete an existing asset control method from the RMS server.
Arguments:
- **CHAR assetClientKey[]** - asset client key
- **CHAR methodKey[]** - control method key

Returns: 1 if call was successful; 0 if call was unsuccessful
Syntax:

```
DEFINE_FUNCTION CHAR RmsAssetControlMethodDelete(CHAR assetClientKey[],CHAR methodKey[])
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure RMS is ONLINE, REGISTERED, and ready for ASSET registration
  IF(![vdvRMS,RMS_CHANNEL_ASSETS_REGISTER])
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodDelete> :: RMS is not ready to accept asset
                           control method changes.';
    RETURN FALSE;
  }

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodDelete> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a control method key has been provided
  IF(methodKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodDelete> :: missing control method key';
    RETURN FALSE;
  }

  // delete existing registered asset control method
  rmsCommand = RmsPackCmdHeader('ASSET.METHOD.DELETE');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,methodKey);
  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

Executing Asset Control Functions

The following callback method will be invoked in each asset monitoring module when it is time to execute an asset control method. The NetLinx programmer can then take specific action based on the provided method key and parse any method arguments from the provided arguments character array.

Arguments will remain packed (encoded) in the character array, and use the "*RmsParseCmdParam*" function to extract each method argument.

The method arguments will be ordered in the packed array by their ordinal index with which they were registered.

```
(***********************************************************)
(* Name:  ExecuteAssetControlMethod                        *)
(* Args:  methodKey - unique method key that was executed  *)
(*        arguments - array of argument values invoked     *)
(*                    with the execution of this method.   *)
(*                                                         *)
(* Desc:  This is a callback method that is invoked by     *)
(*        RMS to notify this module that it should         *)
(*        fulfill the execution of one of this asset's     *)
(*        control methods.                                 *)
(*                                                         *)
(*        This method should not be invoked/called         *)
(*        by any user implementation code.                 *)
(***********************************************************)
DEFINE_FUNCTION ExecuteAssetControlMethod (CHAR methodKey[], CHAR arguments[])

{
  SELECT
  {
    // SLEEP
    ACTIVE(methodKey == 'touch.panel.sleep'):
    {
      SEND_COMMAND dvMonitoredDevice,'SLEEP'
    }
    // SETUP
    ACTIVE(methodKey == 'touch.panel.setup'):
    {
      SEND_COMMAND dvMonitoredDevice,'SETUP'
    }
    // BRIGHTNESS LEVEL 1-100
    ACTIVE(methodKey == 'touch.panel.brightness'):
    {
      STACK_VAR CHAR brightnessArgument[5];
      brightnessArgument = RmsParseCmdParam(arguments);
      SEND_COMMAND dvMonitoredDevice,"'BRIT-',brightnessArgument"
    }
    // MUTE ON/OFF
    ACTIVE(methodKey == 'touch.panel.volume.mute'):
    {
      STACK_VAR CHAR muteArgument[5];
      muteArgument = RmsParseCmdParam(arguments);
      SEND_COMMAND dvMonitoredDevice,"'^MUT-',ITOA(RmsBooleanValue(muteArgument))"
    }
  }
}
```

Please review the NetLinx code samples and *RmsApi.axi* code comments for more details on executing asset control methods.

Excluding Default Asset Parameters

If the RMS Client and SDK register an asset parameter by default that you don't want exposed in your system, you can define an exclusion by calling the following method before the asset registration takes place. The suggested place to put this exclude call would be immediately after the vdvRMS virtual device comes online.

RmsAssetParameterExclude	
RmsAssetParameter Exclude	*Description*: This function is used to exclude a specific asset monitored parameter from being registered to the specified asset client key. *Arguments*: • **CHAR assetClientKey[]** - asset client key • **CHAR parameterKey[]** - monitored parameter key *Returns*: 1 if call was successful; 0 if call was unsuccessful *Syntax*:

```
DEFINE_FUNCTION CHAR RmsAssetParameterExclude(CHAR assetClientKey[],CHAR parameterKey[])
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetParameterExclude> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a parameter key has been provided
  IF(parameterKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetParameterExclude> :: missing parameter key';
    RETURN FALSE;
  }

  // submit the asset exclusion now
  rmsCommand = RmsPackCmdHeader('ASSET.PARAM.EXCLUDE');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,parameterKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,'true'); // force exclusion
  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

Excluding Default Asset Metadata Properties

If the RMS Client and SDK register an asset metadata property by default that you don't want exposed in your system, you can define an exclusion by calling the following method before the asset registration takes place. The suggested place to put this exclude call would be immediately after the vdvRMS virtual device comes online.

RmsAssetMetadataExclude	
RmsAssetMetadata Exclude	*Description*: This function is used to exclude a specific asset metadata property from being registered to the specified asset client key. *Arguments*: • **CHAR assetClientKey[]** - asset client key • **CHAR metadataKey[]** - metadata property key *Returns*: 1 if call was successful; 0 if call was unsuccessful *Syntax*:

```
DEFINE_FUNCTION CHAR RmsAssetMetadataExclude(CHAR assetClientKey[], CHAR metadataKey[])
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataExclude> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a metadata key has been provided
  IF(metadataKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetMetadataExclude> :: missing metadata key';
    RETURN FALSE;
  }

  // submit the asset exclusion now
  rmsCommand = RmsPackCmdHeader('ASSET.METADATA.EXCLUDE');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,metadataKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,'true'); // force exclusion
  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

Excluding Default Asset Control Methods

If the RMS Client and SDK register an asset control method by default that you don't want exposed in your system, you can define an exclusion by calling the following method before the asset registration takes place. The suggested place to put this exclude call would be immediately after the vdvRMS virtual device comes online.

RmsAssetControlMethodExclude	
RmsAssetControl MethodExclude	*Description*: This function is used to exclude a specific asset control method from being registered to the specified asset client key. *Arguments*: • **CHAR assetClientKey[]** - asset client key • **CHAR methodKey[]** - control method key *Returns*: 1 if call was successful; 0 if call was unsuccessful *Syntax*:

```
DEFINE_FUNCTION CHAR RmsAssetControlMethodExclude(CHAR assetClientKey[], CHAR methodKey[])
{
  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

  // ensure an asset client key has been provided
  IF(assetClientKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodExclude> :: missing asset client key';
    RETURN FALSE;
  }

  // ensure a method key has been provided
  IF(methodKey == '')
  {
    SEND_STRING 0, '>>>> RMS API ERROR <RmsAssetControlMethodExclude> :: missing method key';
    RETURN FALSE;
  }

  // submit the asset method exclusion now
  rmsCommand = RmsPackCmdHeader('ASSET.METHOD.EXCLUDE');
  rmsCommand = RmsPackCmdParam(rmsCommand,assetClientKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,methodKey);
  rmsCommand = RmsPackCmdParam(rmsCommand,'true'); // force exclusion
  SEND_COMMAND vdvRMS, rmsCommand;

  RETURN TRUE;
}
```

Programming - Client Messaging

Listening for Client Display Messages

If you have implemented the RMS Application GUI modules and integrated the RMS touch panel pages, then client display messages from RMS are automatically handled and displayed to the touch panels.

However, if you need to display the message content to some other device, or if you are implementing the logic on your own, then obtaining a notification when the RMS system has delivered a message intended for display is quite simple.

The following example will use the *RmsEventListener.axi* Include File, and subscribe to an event callback to get the display message notification. Refer to the documentation for *RmsEventListener.axi* and see the *Listening for RMS Notification Events* section on page 111 for more details on the inner workings of the notification callback in RMS.

First, you must subscribe to the event notification. This is done using the following compiler directive. Make sure this is defined in your user NetLinx program.

```
// SUBSCRIBE TO DISPLAY MESSAGE CALLBACK NOTIFICATIONS
#DEFINE INCLUDE_RMS_EVENT_DISPLAY_MESSAGE_CALLBACK
```

Next, you must define the following callback method in your user NetLinx program. The method will automatically be invoked by the RMS SDK anytime a display message is received. You should implement your custom display logic inside this callback method.

```
// CALLBACK METHOD FOR DISPLAY MESSAGE EVENT NOTIFICATIONS
DEFINE_FUNCTION RmsEventDisplayMessage(CHAR type[],
                                       CHAR title[],
                                       CHAR body[],
                                       INTEGER timeoutSeconds,
                                       CHAR modal),
                                       CHAR responseMessage,)
                                       LONG locationId,
                                       CHAR isDefaultLocation)
{
  // IMPLEMENT MESSAGE DISPLAY HERE
}
```

responseMessage

This parameter indicates whether the received message is a response message to a service provider request. For example, if a Help Request is sent from the Touch Panel to the RMS Web Application and is displayed in the RMS Hotlist, an RMS Administrator can respond to the request by sending a reply message.

- If this administrator takes this action on the service provider request item, then the message received on the NetLinx system will include the "*responseMessage*" parameter set to TRUE.
- If the RMS Administrator sends an unsolicited or generic message to the location, then this "*responseMessage*" parameter will be set to FALSE.

locationId

This parameter indicates the targeted location destination for the display message. There are cases where the system may receive more than one display message assigned to different locations. One such case is when the system contains a Touch Panel asset that has been reassigned in the RMS web user interface to another location.

- If a message is destined for this relocated Touch Panel asset, then a display message will be delivered with the alternate location ID. The "*isDefaultLocation*" parameter indicates if the location is the default location for this client gateway of if the message is destined for an alternate location.
- The client gateway will only receive display messages for alternate locations if a touch panel that is physically connected to this client gateway asset has been reassigned to an alternate location in the RMS web interface.

Sending Help Requests to RMS Server

Sending a help request to the RMS system is very simple. A convenience method exposed in the *RmsApi.axi* Include File wraps the raw Send Command to the vdvRMS virtual device.

```
RmsSendHelpRequest ('Please send help, video projector
                     won't turn on');
```

The following table provides a detailed description of the *RmsSendHelpRequest* function:

RmsSendHelpRequest	

RmsSendHelpRequest	*Description*: This function is used to submit a help request message to the RMS server.
	Arguments:
	• **CHAR requestMessage[]** - message body
	Returns: 1 if call was successful; 0 if call was unsuccessful
	Syntax:
	`DEFINE_FUNCTION CHAR RmsSendHelpRequest(CHAR requestMessage[])`
	`{`
	`  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];`
	`  // ensure a request message has been provided`
	`  IF(requestMessage == '')`
	`  {`
	`  SEND_STRING 0, '>>>> RMS API ERROR <RmsSendHelpRequest> :: missing request message';`
	`  RETURN FALSE;`
	`  }`
	`  // send the request message to RMS`
	`  rmsCommand = RmsPackCmdHeader(RMS_COMMAND_HELP_REQUEST);`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,requestMessage);`
	`  SEND_COMMAND vdvRMS, rmsCommand;`
	`  RETURN TRUE;`
	`}`

Sending Maintenance Requests to RMS Server

Sending a maintenance request to the RMS system is very simple. A convenience method exposed in the *RmsApi.axi* Include File wraps the raw Send Command to the vdvRMS virtual device.

```
RmsSendMaintenanceRequest ('Please fix the screen, it won't open');
```

The following table provides a detailed description of the *RmsSendMaintenanceRequest* function:

RmsSendMaintenanceRequest	

RmsSendMaintenanceRequest	*Description*: This function is used to submit a maintenance request message to the RMS server.
	Arguments:
	• **CHAR requestMessage[]** - message body
	Returns: 1 if call was successful; 0 if call was unsuccessful
	Syntax:
	`DEFINE_FUNCTION CHAR RmsSendMaintenanceRequest(CHAR requestMessage[])`
	`{`
	`  STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];`
	`  // ensure a request message has been provided`
	`  IF(requestMessage == '')`
	`  {`
	`  SEND_STRING 0, '>>>> RMS API ERROR <RmsSendMaintenanceRequest> :: missing request message';`
	`  RETURN FALSE;`
	`  }`
	`  // send the request message to RMS`
	`  rmsCommand = RmsPackCmdHeader(RMS_COMMAND_MAINTENANCE_REQUEST);`
	`  rmsCommand = RmsPackCmdParam(rmsCommand,requestMessage);`
	`  SEND_COMMAND vdvRMS, rmsCommand;`
	`  RETURN TRUE;`
	`}`

Programming - Advanced Topics

Listening for RMS Notification Events

RMS Enterprise introduces a SUBSCRIBER / EVENT LISTENER pattern for handling event notifications emitted from the RMS Client via the RMS NetLinx Adapter as command data events over the (vdvRMS) RMS NetLinx virtual device interface. This pattern is implemented via the inclusion of the *RmsEventListener.axi* Include File via the use of compiler directives and callback methods.

To subscribe to an RMS event, you must include the *RmsEventListener.axi* Include File in your parent program (if it is not already included).

Next include a #DEFINE INCLUDE_RMS_EVENT_XXXXX statement for each event that you want to handle. The list of available event and #DEFINE directives can be found in the source code comments inside the *RmsEventListener.axi* Include File.

In this example, we will subscribe to two events: *System Power Request* and *Client Online*.

```
#DEFINE INCLUDE_RMS_EVENT_SYSTEM_POWER_REQUEST_CALLBACK
#DEFINE INCLUDE_RMS_EVENT_CLIENT_ONLINE_CALLBACK
```

This will effectively enable the callback method invocation to our main program anytime the RMS client reached the ONLINE state, and when a SYSTEM POWER state change request has been made.

Enabling these two compiler directives will expose code in the *RmsEventListener.axi* Include File to perform calls to the following method signatures (these are known as callback methods). These methods must be defined in your main program.

Internally, the *RmsEventListener.axi* Include File includes a NetLinx DATA_EVENT listener on the RMS NetLinx Virtual Device.

Upon receiving a COMMAND DATA_EVENT, this implementation code will parse the command header and any necessary event arguments. It will then perform the appropriate the callback method.

```
// THIS CALLBACK METHOD IS INVOKED AUTOMATICALLY WHEN
// THE SYSTEM POWER STATE CHANGES
DEFINE_FUNCTION RmsEventSystemPowerChangeRequest(CHAR powerOn)
{
}

// THIS CALLBACK METHOD IS INVOKED AUTOMATICALLY WHEN
// RMS HAS REQUESTED THAT SYSTEM POWER BE CHANGEDT
DEFINE_FUNCTION RmsEventClientOnline()
{
}
```

The method signatures that correspond to each event are also listed in the source code comments in the *RmsEventListener.axi* Include File.

You should place your custom implementation logic in the body of each of these callback methods.

Listening for RMS Exceptions

RMS Exceptions (Errors) encountered by the RMS Client (RMS Engine) are delivered to the RMS NetLinx Virtual Device via COMMAND DATA EVENTS. The *RmsEventListener.axi* Include File provides an event callback to easily handle these events. Refer to the documentation for *RmsEventListener.axi* and see the *Listening for RMS Notification Events* section on page 111 for more details on the inner workings of the notification callbacks in RMS.

First, you must subscribe to the event notification. This is done using the following compiler directive. Make sure this is defined in your user NetLinx program.

```
// SUBSCRIBE TO RMS EXCEPTION CALLBACK NOTIFICATIONS
#DEFINE INCLUDE_RMS_EVENT_EXCEPTION_CALLBACK
```

Next, you must define the following callback method in your user NetLinx program. The method will automatically be invoked by the RMS SDK anytime a RMS Exception event is received. You should implement your custom handling logic inside this callback method.

The event will include an exception textual message and the command header of the offending command sent to the RMS NetLinx virtual device if one is present:

```
// CALLBACK METHOD FOR RMS CLIENT EXCEPTION EVENT NOTIFICATIONS
DEFINE_FUNCTION RmsEventException (CHAR exceptionMessage [],
                                   CHAR commandHeader [])
{
  // IMPLEMENT CUSTOM EXCEPTION HANDLING HERE
}
```

Asset Power / Energy Consumption

RMS Enterprise will automatically assume power consumption rates (Watts) based on the asset type an asset is defined with and based on the power state of the device.

- When the asset power is ON, the RMS server will apply the RUNTIME power rate (Watts) assigned to either the asset model if configured, or asset type.
- When the asset power is OFF, the RMS server will apply the STANDBY power rate (Watts) assigned to the asset model or asset type.

This default automatic handling of asset power consumption (Watts) by the RMS server can be overridden if the asset registers a parameter of type "POWER_CONSUMPTION". If the server detects this asset parameter registered to the asset by the RMS client, then it will suspend all automatic handling of asset power consumption on behalf of this asset.

Registering an asset parameter of type "POWER_CONSUMPTION" means that the NetLinx programmer will be fully responsible for providing the power consumption value (in Watts) to RMS as they change. This is necessary if the device can provide information on its power consumption or if you have some power monitoring equipment that can monitor power consumption on behalf of this asset.

NOTE: *If using the AMX NXA-PDU-1508-08 power management unit to monitor asset power on behalf of other assets on the RMS client, then this asset parameter implementation work has already been done inside the RmsPowerDistributionUnitMonitor module.*

If you wish to implement your own custom asset power consumption parameter for monitoring asset power, please review the NetLinx source code in the *RmsDuetLightSystemMonitor* and *RmsPowerDistributionUnitMonitor* modules for concrete examples on how to implement targeted asset power consumption rates for assets.

Tracking Source Usage

The RMS SDK includes a *RmsSourceUsageMonitor* module and a *RmsSourceUsage* Include File to provide source device usage tracking in the RMS system.

Please review the *RMS Source Usage Monitor Module & Include File* section on page 38 for more information on source usage tracking with the new RMS source usage monitoring module.

Enabling support for source usage tracking in your program is as simple as referencing the *RmsSourceUsage.axi* Include File and defining a virtual device to represent the source usage module. This is illustrated in the code below:

```
// Define your source usage virtual device under the
// DEFINE_DEVICE section in your program
vdvRMSSourceUsage = 33002:1:0

// Include the RMS Source Usage wrapper functions and module
#INCLUDE 'RmsSourceUsage';
```

After you've declared that your program will be using Source Usage, the next step is to assign the devices that you'll be tracking to a unique index. Each device is *required* to have a unique index. For example:

```
// Assign Mutually Exclusive Devices
RmsSourceUsageAssignAssetMutExcl(1, dvDSS);
RmsSourceUsageAssignAssetMutExcl(2, dvDVR);
RmsSourceUsageAssignAssetMutExcl(3, dvDiscDevice);
RmsSourceUsageAssignAssetMutExcl(4, dvDocCamera);
// Assign Mutually Exclusive Virtual Devices
RmsSourceUsageAssignAssetMutExcl(5, vdvAuxInput);
RmsSourceUsageAssignAssetMutExcl(6, vdvLaptop);
```

You'll notice above that a mixture of virtual and real devices are being assigned to have their source usage tracked. Non-controlled devices should have their source usage tracked by passing in the virtual device. All controlled devices should be tracked by passing in the actual device.

To activate source usage tracking on the Disc Device defined above, the following code would be used:

```
// Activate source usage tracking on dvDiscDevice and
// disable source usage tracking on all other devices
RmsSourceUsageActivateSource(3);
```

The above devices are having their source usage tracked in a mutually exclusive manner. In other words, when tracking of source usage is activated for a device, such as the Disc Device in the above example, the previous selected device will have its source usage tracking deactivated automatically. For example, in the below code, tracking of source usage will be activated for the Document Camera device, and subsequently disabled for the Disc Device:

```
// Activate source usage tracking on dvDocCamera and
// disable source usage tracking on dvDiscDevice
RmsSourceUsageActivateSource(4);
```

The code below illustrates how you would manually deactivate source usage tracking for a device:

```
// Deactivate source usage tracking on dvDocCamera
RmsSourceUsageDeactivateSource(4);
```

Please review the *RmsSourceUsage.axi* code comments for more details on tracking source usage in RMS.

Implementing System Power

Because RMS Enterprise offers greater flexibility and support for multiple control system masters that can coexists in a single location, RMS SDK 4.0 does not provide any concrete implementation of System Power. The RMS SDK provides the communications infrastructure and state management for System Power, but the NetLinx programmer must provide the implementation in the user program.

First, if you wish to provide System Power on the target NetLinx controller, you must first enable the compiler directive in the *RmsControlSystemMonitor* module.

```
// comment out the HAS_SYSTEM_POWER precompiler variable
// if you do not wish to register a 'System Power' parameter
// and control methods for this control system. Some
// implementations may not warrant/desire the concept of
// a single System Power convention.
#WARN 'Define if you want a SYSTEM POWER parameter & control methods registered for this control system ...'
// to include the system power parameter and system power control methods
// this module definition is required
DEFINE_MODULE 'RmsSystemPowerMonitor' mdlRmsSystemPowerMonitorMod(vdvRMS,dvMaster);
```

Enabling this compiler directive will cause the System asset to register System Power asset parameter and control methods.

Next, you must implement the logic for what actions to take when the System Power status changes. You can use the event subscription model to get notified when the system power state changes via callback methods invoked by the *RmsEventListener.axi* Include File.

NOTE: *This event subscriber and listener callback method are automatically provided for you if you include the RmsSystemEventHandler.axi Include File.*

```
#DEFINE INCLUDE_RMS_EVENT_SYSTEM_POWER_REQUEST_CALLBACK
// THIS CALLBACK METHOD IS INVOKED AUTOMATICALLY WHEN
// RMS HAS REQUESTED THAT SYSTEM POWER BE CHANGED
DEFINE_FUNCTION RmsEventSystemPowerChanged(CHAR powerOn)
{
  // IMPLEMENT LOGIC FOR SYSTEM POWER ON/OFF
}
```

If you need to implement some logic based on System Power state on a specific asset/device, then there is already a callback method stub implemented inside each device monitor module. Just add your implementation logic to the body of this existing callback method:

```
(***********************************************************)
(* Name:  SystemPowerChanged                               *)
(* Args:  powerOn - boolean value representing ON/OFF      *)
(*                                                         *)
(* Desc:  This is a callback method that is invoked by     *)
(*        RMS to notify this module that the SYSTEM POWER  *)
(*        state has changed states.                        *)
(*                                                         *)
(*        This method should not be invoked/called         *)
(*        by any user implementation code.                 *)
(***********************************************************)
DEFINE_FUNCTION SystemPowerChanged (CHAR powerOn)
{
  // optionally implement logic based on
  // system power state.
}
```

Implementing System Modes

RMS SDK 4.0 introduces a new feature concept of *System Modes*. System Modes can be used if your location supports the concept of configuring the control system and user experience by choosing a named startup mode of behavior. Modes such as Presentation, Video Conference, Audio Conference, etc are common system modes of behavior.

The RMS SDK does not provide any concrete implementation of System Mode, but does provide the communications infrastructure for System Mode changes and event notifications. NetLinx programmers must provide any concrete implementation of System Modes in the user program.

First, if you wish to provide System Modes on the target NetLinx controller, you must first enable the compiler directive in the *RmsControlSystemMonitor* module and define a pipe-delimited series of available named modes.

```
// comment out the HAS_SYSTEM_MODES compiler directive
// if you do not wish to register a 'System Mode' control
// method for this control system. Some implementations
// may not warrant/desire the concept of system mode convention.
#WARN 'Define if you want a SYSTEM MODES such as Presentation, Video Conf, etc ...'
// to include the system mode parameter and system mode control method
// this module definition is required
DEFINE_MODULE 'RmsSystemModeMonitor' mdlRmsSystemModeMonitorMod(vdvRMS,dvMaster,SYSTEM_MODES);
```

Enabling this compiler directive will cause the System asset to register System Modes asset control methods.

Next, you must implement the logic for what actions to take when the System Mode changes. You can use the event subscription model to get notified when the system mode changes via callback methods invoked by the *RmsEventListener.axi* Include File.

NOTE: *This event subscriber and listener callback method are automatically provided if you include the RmsSystemEventHandler.axi Include File.*

```
#DEFINE INCLUDE_RMS_EVENT_SYSTEM_MODE_REQUEST_CALLBACK

// THIS CALLBACK METHOD IS INVOKED AUTOMATICALLY WHEN
// RMS HAS REQUESTED THAT THE SYSTEM MODE BE CHANGED
DEFINE_FUNCTION RmsEventSystemModeChangeRequest (CHAR newMode[])
{
  // IMPLEMENT LOGIC FOR SYSTEM MODE CHANGE
}
```

If you need to implement some logic based on System Mode changes on a specific asset/device then there is already a callback method stub implemented inside each device monitor module. Just add your implementation logic to the body of this existing callback method:

```
(***********************************************************)
(* Name:  SystemModeChanged                                *)
(* Args:  modeName - string value representing mode change *)
(*                                                         *)
(* Desc:  This is a callback method that is invoked by     *)
(*        RMS to notify this module that the SYSTEM MODE   *)
(*        state has changed states.                       *)
(*                                                         *)
(*        This method should not be invoked/called         *)
(*        by any user implementation code.                *)
(***********************************************************)
DEFINE_FUNCTION SystemModeChanged (CHAR newMode[])
{
  // optionally implement logic based on
  // newly selected system mode name.
}
```

Programmatically Setting the Client Configuration

The default method to enable and configure a NetLinx system for use with RMS is to use the *RMS Client Web Configuration* page exposed by the NetLinx master's web-based configuration tools.

Alternatively, the RMS SDK also supports a SEND_COMMAND API for settings RMS client configuration. See the *RMS Client Settings & Configuration - Command API* section on page 132 for a complete listing of the available API options and commands.

Below is an example of a RMS configuration applied via the command API.

```
// SET RMS CLIENT CONFIGRUATION
SEND_COMMAND vdvRMS, 'CONFIG.CLIENT.NAME-MyNetLinxMaster'
SEND_COMMAND vdvRMS, 'CONFIG.SERVER.URL-http://myserver.mycompany.com/rms'
SEND_COMMAND vdvRMS, 'CONFIG.SERVER.PASSWORD-password'

// ENABLE THE RMS CLIENT AS THE LAST STEP
SEND_COMMAND vdvRMS, 'CONFIG.CLIENT.ENABLED-true'
```

Listening for RMS Client Connections

If your NetLinx program needs to listen for RMS client connection status, there are a couple of options you can use. The simplest method is to monitor channel 250 for client connectivity and channel 251 for client registered status on the RMS NetLinx virtual device interface. Alternatively, the following event notifications are provided via the *RmsEventListener.axi* Include File that provide details on RMS client connectivity.

This event subscription and event notification callback method indicates when the RMS client becomes connected to the RMS server:

```
#DEFINE INCLUDE_RMS_EVENT_CLIENT_ONLINE_CALLBACK
DEFINE_FUNCTION RmsEventClientOnline()
{
}
```

This event subscription and event notification callback method indicates when the RMS client becomes disconnected from the RMS server:

```
#DEFINE INCLUDE_RMS_EVENT_CLIENT_OFFLINE_CALLBACK
DEFINE_FUNCTION RmsEventClientOffline()
{
}
```

This event subscription and event notification callback method indicates the RMS client is connected and registered on the RMS system:

```
#DEFINE INCLUDE_RMS_EVENT_CLIENT_REGISTERED_CALLBACK
DEFINE_FUNCTION RmsEventClientRegistered()
{
}
```

This event subscription and event notification callback method indicates when the RMS client connection state changes:

```
#DEFINE INCLUDE_RMS_EVENT_CLIENT_STATE_CHANGE_CALLBACK
DEFINE_FUNCTION RmsEventClientStateChanged(CHAR oldState[],
                                           CHAR newState[])
{
}
```

Proxy Custom Commands Through RMS

The RMS NetLinx virtual device interface supports a special command syntax for NetLinx programmers to implement their own custom messaging. The *vdvRms* device is already defined in all RMS asset monitoring modules and the main program. If you find a need to be able to send custom messages throughout the system, you can do this using the RMS command proxy syntax.

Basically, any Send Command that starts with the '**@**' character is considered a proxy command, and the RMS virtual device will relay this command back out on the virtual device interface.

```
// PROXY CUSTOM COMMAND THRU RMS VIRUTAL DEVICE
SEND_COMMAND vdvRMS, '@MYCUSTOMHEADER-MyCustomField1,MyCustomField2'
```

The *RmsApi.axi* Include File also contains a wrapper function to facilitate these custom commands. If using this function, there is no need to include the '@' starting character, the function will automatically affix this to the start of the header command.

```
// PROXY CUSTOM COMMAND THRU RMS VIRUTAL DEVICE
RmsProxyCustomCommand ('MYCUSTOMHEADER','MyCustomData');
```

This custom command can then be intercepted/received by any DATA_EVENT listener of the *vdvRMS* virtual device.

```
// LISTEN FOR CUSTOM PROXIED MESSAGE ON THE RMS VIRUTAL DEVICE
DATA_EVENT[vdvRMS]
{
  COMMAND:
  {
     STACK_VAR CHAR rmsHeader[RMS_MAX_HDR_LEN];
     IF(LEFT_STRING(rmsHeader,1) == '@')
      {
         // IMPLEMENT CUSTOM MESSAGE HANDLER HERE
      }
  }
}
```

Custom proxy command handling is also supported by the RMS Event Notification system implemented in the *RmsEventListener.axi* Include File.

Include the following compiler directive to subscribe for the event notification callback and implement the callback method to handle the custom command:

```
// SUBSCRIBE TO CUSTOM USER PROXY COMMANDS
#DEFINE INCLUDE_RMS_EVENT_CUSTOM_COMMAND_CALLBACK

// CALLBACK METHOD FOR USER PROXY COMMANDS
DEFINE_FUNCTION RmsEventCustomCommand(CHAR header[], CHAR data[])
{
   // IMPLEMENT CUSTOM MESSAGE HANDLING LOGIC HERE
}
```

RMS Duet Module Properties

Overview

When a Duet device is registered with RMS, RMS can look at RMS specific Duet module properties for overriding device specific information. These properties allow a programmer to override the default implementation and specify device specific unique information.

If not provided, RMS will use a fallback mechanism to determine the most appropriate values. These fallback properties are also listed below. Use the PROPERTY command for each Duet module to set any of these desired properties.

NOTE: *If you provide specific concrete information such as asset model and manufacturer in the initial asset registration call, then the RMS SDK will use that information and not look at the corresponding Duet module properties.*

After device registration RMS will update all these properties with the current applied values so you can query any of these named properties to obtain their current values.

RMS Duet Module Properties	
RMS-Type (*Required*)	This property is required for RMS to register the Duet device with the RMS server. The only acceptable value for this property at this time is 'Asset'. `SEND_COMMAND vdvDuetModule, "'PROPERTY-RMS-Type,Asset'"` There is no fallback for this property; it is required for use with RMS. (This property is automatically set by each RMS monitoring module.)
RMS-Asset-Name	This property will define the asset's friendly name used when registering this asset with the RMS server. This property will not override an asset name that was explicitly provided in a RMS Duet Asset registration call. If not provided, the fallback default Duet module property of '`Device-Name`' will be used to determine the asset name.
RMS-Asset-Description	This property will define the asset's descriptive text used when registering this asset with the RMS server. This property will not override an asset description that was explicitly provided in a RMS Duet Asset registration call. If not provided, the fallback default Duet module property of '`Device-Description`' will be used to determine the asset description.
RMS-Asset-Client-Key	This property will define the asset's system unique client key identifier string. This identifier string must be unique in scope on the NetLinx master. If not provided, the fallback value is constructed using the D:P:S address of the physical (real) device. Example: "5001:16:0" **It is not recommended to override this property** unless you know specifically what you are doing in some advanced programming scenario.
RMS-Asset-Global-Key	This property will define the asset's globally unique key identifier string. This identifier string much be unique in scope of the entire RMS system including all locations, and all device manufacturers. Values such as MAC addresses may be used for this property, but values like serial numbers may not be unique across all manufacturer devices. If not provided, no fallback value will be used. This property is optional and is not required. **It is not recommended to override this property** unless you know specifically what you are doing in some advanced programming scenario.
RMS-Asset-Make	This property will define the asset's manufacturer name used when registering this asset with the RMS server. This property will not override an asset manufacturer name that was explicitly provided in a RMS Duet Asset registration call. If not provided, the fallback default Duet module property of '`Device-Make`' will be used to determine the asset manufacturer name.
RMS-Asset-Make-Url	This property will define the asset's manufacturer website URL used when registering this asset with the RMS server. This property will not override an asset manufacturer URL that was explicitly provided in a RMS Duet Asset registration call. If not provided, the fallback Duet module property of '`Device-Make-Url`' will be used to determine the asset manufacturer website URL.
RMS-Asset-Model	This property will define the asset's model name used when registering this asset with the RMS server. This property will not override an asset model name that was explicitly provided in a RMS Duet Asset registration call. If not provided, the fallback default Duet module property of '`Device-Model`' will be used to determine the asset model name.
RMS-Asset-Model-Url	This property will define the asset's model website URL used when registering this asset with the RMS server. This property will not override an asset model URL that was explicitly provided in a RMS Duet Asset registration call. If not provided, the fallback Duet module property of '`Device-Model-Url`' will be used to determine the asset model website URL.

RMS Duet Module Properties (Cont.)	
RMS-Asset-Serial	This property will define the asset's serial number used when registering this asset with the RMS server. This property will not override an asset serial number that was explicitly provided in a RMS Duet Asset registration call. If not provided, the fallback default Duet module property of '`Device-Serial`' will be used to determine the asset serial number.
RMS-Asset-Type	This property will define the asset's type identifier used when registering this asset with the RMS server. The asset type helps RMS classify and categorize the device. This property will not override an asset type string that was explicitly provided in a RMS Duet Asset registration call. If not provided, the fallback default Duet module property of 'Device-SDKClass' will be used to determine the asset's device type. In this fallback case, RMS will only use the last portion of the SDK class string that specifies the class name. **It is not recommended to override this property** unless you know specifically what you are doing in some advanced programming scenario.
RMS-Asset-Module-Name	This property cannot be set, but can be queried to determine the RMS client module responsible for registering the asset.
RMS-Asset-Module-Version	This property cannot be set, but can be queried to determine the RMS client module version responsible for registering the asset.

HAS-PROPERTIES

Duet Module HAS-PROPERTIES

The RMS 4.0 SDK observes the pre-defined set of HAS-PROPERTIES defined by the Duet Device Module to help identify which features that a module supports.

If the HAS-PROPERTY is missing or set to "false" for each of the defined HAS-PROPERTIES below, the RMS Duet Module Monitor will exclude the respective asset metadata properties, monitoring parameters, and control methods based on the feature.

Duet Module HAS-PROPERTIES			
Duet Device Type	**RMS Supported HAS-PROPERTIES**		
Audio Conferencer	• Has-Power • Has-Volume • Has-Auto-Answer	• Has-Dialer • Has-Privacy	• Has-Hook • Has-Phonebook
Camera	• Has-Power • Has-Camera-Preset • Has-Auto-Focus • Has-Auto-Iris	• Has-Pan-Tilt • Has-Focus • Has-Iris • Has-Zoom	• Has-Pan-Tilt-Speed • Has-Iris-Speed • Has-Focus-Speed • Has-Zoom-Speed
Digital Satellite System	• Has-Power • Has-Station	• Has-Station-Preset	• Has-Tuner-Band
Digital Video Recorder	• Has-Power • Has-Input-Select	• Has-Station • Has-Station-Preset	• Has-Tuner-Band • Has-Disc-Transport
Disc Device	• Has-Power	• Has-Disc-Transport	• Has-Disc-Select
Document Camera	• Has-Power • Has-Input-Select • Has-Upper-Light • Has-Lower-Light	• Has-Auto-Focus • Has-Auto-Iris • Has-Focus • Has-Iris	• Has-Zoom • Has-Focus-Speed • Has-Iris-Speed • Has-Zoom-Speed
HVAC	• Has-CoolSetpoint • Has-Hold	• Has-Lock • Has-Outdoor-Temperature	• Has-Temperature-Scale
Note: The RMS SDK includes support for "HVAC" as a RMS NetLinx Monitoring module but does not include a RMS Duet Monitoring Module for "HVAC".			
Light System	• Has-Light-Level		
Monitor	• Has-Power • Has-Volume	• Has-Aspect-Ratio	• Has-Input-Select
Receiver	• Has-Power • Has-Volume • Has-Balance • Has-Loudness	• Has-Treble • Has-Bass • Has-Input-Select	• Has-Station • Has-Station-Preset • Has-Tuner-Band
Security System	• Has-Power	• Has-OK-To-Arm	
Note: The RMS SDK includes support for "Security System" as a RMS NetLinx Monitoring module but does not include a RMS Duet Monitoring Module for "Security System".			
Settop Box	• Has-Power • Has-Volume	• Has-Station • Has-Station-Preset	• Has-Tuner-Band
Switcher	• Has-Power • Has-Volume	• Has-Gain	• Has-Switcher-Preset
TV	• Has-Power • Has-Volume • Has-Aspect-Ratio	• Has-Input-Select • Has-Station	• Has-Station-Preset • Has-Tuner-Band
Video Conferencer	• Has-Power • Has-Volume • Has-Auto-Answer • Has-Input-Select • Has-Privacy • Has-Audible-Ring	• Has-Dialer • Has-Hook • Has-Camera-Preset • Has-Auto-Focus • Has-Auto-Iris • Has-Pan-Tilt	• Has-Focus • Has-Zoom • Has-Pan-Tilt-Speed • Has-Focus-Speed • Has-Zoom-Speed • Has-Phonebook
Video Projector	• Has-Lamp • Has-Volume	• Has-Aspect-Ratio	• Has-Input-Select

Please note that older Duet Device Modules may not have the applicable set of HAS-PROPERTIES defined. In this case, you may need to manually define the HAS-PROPERTIES for the Duet Module. Typically if the Duet Module is denoted as Meeting Room Compatible on the AMX InConcert online database, then it already includes the necessary HAS-PROPERTIES.

To manually define the features supported and add the necessary HAS-PROPERTIES, you will need to use the "PROPERTY-" send command to declare each HAS-PROPERTY on the Duet Module's virtual device.

These HAS-PROPERTY definitions should be applied after the virtual device comes online.

```
// Manually define the set of HAS-PROPERTIES for this Duet
// Device Module based on the devices actual feature set
DATA_EVENT[vdvRECV1]
{
  ONLINE:
  {
    SEND_COMMAND vdvRECV1,'PROPERTY-Has-Power,true';
    SEND_COMMAND vdvRECV1,'PROPERTY-Has-Volume,true';
    SEND_COMMAND vdvRECV1,'PROPERTY-Has-Balance,true';
    SEND_COMMAND vdvRECV1,'PROPERTY-Has-Loudness,true';
    SEND_COMMAND vdvRECV1,'PROPERTY-Has-Treble,true';
    SEND_COMMAND vdvRECV1,'PROPERTY-Has-Bass,true';
    SEND_COMMAND vdvRECV1,'PROPERTY-Has-Input-Select,true';
    SEND_COMMAND vdvRECV1,'PROPERTY-Has-Station,true';
    SEND_COMMAND vdvRECV1,'PROPERTY-Has-Station-Preset,true';
    SEND_COMMAND vdvRECV1,'PROPERTY-Has-Tuner-Band,true';
  }
}
```

RMS NetLinx Monitoring Module HAS_PROPERTY Compiler Directives

The RMS NetLinx device monitoring modules include a similar convention to the Duet Module HAS-PROPERTIES to limit the RMS registered asset metadata properties, monitoring parameters, and control methods based on the features supported by the device (see the *Duet Module HAS-PROPERTIES* section on page 118 for more information).

Instead of HAS-PROPERTIES that are assigned using a SEND_COMMAND to the Duet virtual device, each RMS NetLinx Monitoring module includes the supported HAS-PROPERTIES as compiler directives that are declared at the top of each source file. The HAS_PROPERTY compiler directives should be commented out or uncommented based on the features of the physical device.

Example:

```
//
// Has-Properties
//
#DEFINE HAS_POWER
#DEFINE HAS_VOLUME
#DEFINE HAS_SOURCE_SELECT
#DEFINE HAS_TUNER
#DEFINE HAS_PREAMP
```

RMS NetLinx Virtual Device API

Overview

The RMS NetLinx virtual device API provides the lowest level API and direct access to the RMS Engine running on the NetLinx platform. The RMS NetLinx Adapter module must be defined in the user program code to define a NetLinx virtual device and connect it to the RMS Engine.

This section defines all the channels, levels, and Send Commands exposed by the RMS Engine via the RMS NetLinx Adapter.

API Conventions

The RMS NetLinx Virtual Device API is based on the same conventions used by Duet device modules:

- **SEND_COMMANDS**: are used to send query requests or instructions to RMS.
- **DATA EVENTS (Commands)**: are used to listen to query response or notification events from RMS.
- **CHANNELS**: are used to provide simple feedback on RMS client state/status.
- **LEVELS**: are used to provide simple feedback on RMS client state/status.

Send commands that start with the "?" character are queries that interrogate the RMS client subsystem for information.

Both query responses and notification events that are generated from the RMS NetLinx Adapter module are emitted as COMMAND DATA_EVENTS on the RMS virtual device.

Character & String Escaping

Strings and commands sent and received from the RMS NetLinx adapter virtual device (*vdvRMS*) have special characters (*comma*, *single quote* and *double quote*) that must be properly escaped when required in the data string.

The RMS API uses the Duet string escaping convention. More information on the character escaping can be found in the Standard NetLinx API Help guide in NetLinx Studio under the topic of Command and Escape Characters.

The *RmsApi* Include File includes convenience function wrappers that will automatically perform the string escaping and un-escaping:

- **RmsParseDPSFromString** (see page 121)
- **RmsPackCmdHeader** (see page 121)
- **RmsPackCmdParam** (see page 121)
- **RmsPackCmdParamArray** (see page 122)
- **RmsParseCmdHeader** (see page 122)
- **RmsDuetParseCmdParam** (see page 122)
- **RmsParseCmdParamEx** (see page 123)

The RMS SDK includes convenience function wrappers in the *RMSAPI.axi* Include File that will automatically perform the string escaping and un-escaping.

The following code snippet provides an example of escaping a command destined for the RMS NetLinx virtual device API:

```
STACK_VAR CHAR rmsCommand[RMS_MAX_CMD_LEN];

rmsCommand = RmsPackCmdHeader('ASSET.SERIAL');
rmsCommand = RmsPackCmdParam(rmsCommand,'128:1:0');
rmsCommand = RmsPackCmdParam(rmsCommand,'01235X34GS2');

SEND_COMMAND vdvRMS, rmsCommand;
```

This code snippet provides an example of un-escaping a command received from the RMS NetLinx virtual device API.

```
STACK_VAR CHAR rmsHeader[RMS_MAX_HDR_LEN];
STACK_VAR CHAR rmsParam1[RMS_MAX_PARAM_LEN];
STACK_VAR CHAR rmsParam2[RMS_MAX_PARAM_LEN];

rmsHeader = RmsParseCmdHeader(DATA.TEXT);
rmsParam1 = RmsParseCmdParam(DATA.TEXT);
rmsParam2 = RmsParseCmdParam(DATA.TEXT);
```

RMS Command Escaping Functions

The Command Escaping Functions in the *RmsApi.axi* Include File are described in the following table:

RMS Command Escaping Functions	

RmsParseDPS FromString	*Description*: This function is used to parse a string and extract a D:P:S device instance.
	Arguments:
	• **CHAR cCmd[] (in)** - target string to search in
	• **DEV dvDPS (out)** - device to return
	Returns: -nothing-
	Syntax:
	`DEFINE_FUNCTION RmsParseDPSFromString(CHAR cCmd[], DEV dvDPS)`
	`STACK_VAR`
	`INTEGER nPos`
	`{`
	`  dvDPS.Number = ATOI(cCmd)`
	`  dvDPS.Port = 1`
	`  dvDPS.System = 0`
	`  nPos = FIND_STRING(cCmd,':',1)`
	`  IF (nPos)`
	`  {`
	`  nPos++`
	`  dvDPS.Port = ATOI(MID_STRING(cCmd,nPos,LENGTH_STRING(cCmd)-nPos+1))`
	`  nPos = FIND_STRING(cCmd,':',nPos)`
	`  IF (nPos)`
	`  {`
	`  nPos++`
	`  dvDPS.System = ATOI(MID_STRING(cCmd,nPos,LENGTH_STRING(cCmd)-nPos+1))`
	`  }`
	`  }`

RmsPackCmdHeader	*Description*:
	Adds the command header to the string and adds the command if missing.
	This function assumes the standard Duet command separator '-'.
	Parameters:
	• **(1) IN** - sndcmd/str header
	Returns: Packed header with command separator added if missing
	Syntax:
	`DEFINE_FUNCTION CHAR[RMS_MAX_HDR_LEN] RmsPackCmdHeader(CHAR cHdr[])`
	`{`
	`  STACK_VAR CHAR cSep[1]`
	`  cSep = '-'`
	
	`  IF (RIGHT_STRING(cHdr,LENGTH_STRING(cSep)) != cSep)`
	`  RETURN "cHdr,cSep";`
	
	`  RETURN cHdr;`
	`}`

RmsPackCmdParam	*Description*: Use this function to package parameter for module SEND_COMMAND or SEND_STRING. Wraps the parameter in double-quotes if it contains the separator.
	This function assumes the standard Duet parameter separator ','.
	Parameters:
	• **(1) IN** - sndcmd/str to which parameter will be added
	• **(2) IN** - sndcmd/str parameter
	Returns: Packed parameter wrapped in double-quotes if needed, added to the command.
	Syntax:
	`DEFINE_FUNCTION CHAR[RMS_MAX_CMD_LEN] RmsPackCmdParam(CHAR cCmd[], CHAR cParam[])`
	`{`
	`  STACK_VAR CHAR cTemp[RMS_MAX_CMD_LEN]`
	`  STACK_VAR CHAR cTempParam[RMS_MAX_CMD_LEN]`
	`  STACK_VAR CHAR cCmdSep[1]`
	`  STACK_VAR CHAR cParamSep[1]`
	`  STACK_VAR INTEGER nLoop`
	`  cCmdSep = '-'`
	`  cParamSep = ','`
	
	`  // Not the first param? Add the param separator`
	`  cTemp = cCmd`
	`  IF (FIND_STRING(cCmd,cCmdSep,1) != (LENGTH_STRING(cCmd)-LENGTH_STRING(cCmdSep)+1))`
	`  cTemp = "cTemp,cParamSep"`
	
	`  // Escape any quotes`
	`  FOR (nLoop = 1; nLoop <= LENGTH_ARRAY(cParam); nLoop++)`
	`  {`
	`  IF (cParam[nLoop] == '"')`
	`  cTempParam = "cTempParam,'"'"`
	`  cTempParam = "cTempParam,cParam[nLoop]"`
	`  }`

RMS Command Escaping Functions (Cont.)

RmsPackCmdParam Array

Description: Use this function to package parameters for module SEND_COMMAND or SEND_STRING. Wraps the parameter in double-quotes if it contains the separator and separates them using the separator sequence. This function assumes the standard Duet parameter separator ','

Parameters:

- **(1) IN** - sndcmd/str to which parameter will be added
- **(2) IN** - sndcmd/str parameter array

Returns: Packed parameters wrapped in double-quotes if needed

Syntax:

```
DEFINE_FUNCTION CHAR[RMS_MAX_CMD_LEN] RmsPackCmdParamArray(CHAR cCmd[], CHAR cParams[][])
{
  STACK_VAR CHAR    cTemp[RMS_MAX_CMD_LEN]
  STACK_VAR INTEGER nLoop
  STACK_VAR INTEGER nMax
  STACK_VAR CHAR cCmdSep[1]
  STACK_VAR CHAR cParamSep[1]
  cCmdSep = '-'
  cParamSep = ','

  nMax = LENGTH_ARRAY(cParams)
  IF (nMax == 0)
    nMax = MAX_LENGTH_ARRAY(cParams)

  cTemp = cCmd
  FOR (nLoop = 1; nLoop <= nMax; nLoop++)
    cTemp = RmsPackCmdParam(cTemp,cParams[nLoop])

  RETURN cTemp;
}
```

RmsParseCmdHeader

Description: Use this function to parse out parameters from module SEND_COMMAND or SEND_STRING. Parses the strings sent to or from modules extracting the command header.
Command separating character assumed to be '-', Duet standard

Parameters:

- **(1) IN/OUT** - sndcmd/str data

Returns: Parsed property/method name, still includes the leading '?' if present.

Syntax:

```
DEFINE_FUNCTION CHAR[RMS_MAX_HDR_LEN] RmsParseCmdHeader(CHAR cCmd[])
{
  STACK_VAR CHAR cTemp[RMS_MAX_HDR_LEN]
  STACK_VAR CHAR cSep[1]
  cSep = '-'

  // Assume the argument to be the command
  cTemp = cCmd

  // If we find the separator, remove it from the command
  IF (FIND_STRING(cCmd,cSep,1) > 0)
  {
    cTemp = REMOVE_STRING(cCmd,cSep,1)
    IF (LENGTH_STRING(cTemp))
      cTemp = LEFT_STRING(cTemp,LENGTH_STRING(cTemp)-LENGTH_STRING(cSep))
  }

  // Did not find separator, argument is the command (like ?SOMETHING)
  ELSE
    cCmd = ""

  RETURN cTemp;
}
```

RmsDuetParseCmd Param

Description: Use this function to parse out parameters from module SEND_COMMAND or SEND_STRING. Parses the strings sent to or from modules extracting the parameters.

- A single param is picked of the cmd string and removed, through the separator.
- The separator is NOT returned from the function.
- If the first character of the param is a double quote, the function will remove up to (and including) the next double-quote and the separator without spaces.
- The double quotes will then be stripped from the parameter before it is returned.
- If the double-quote/separator sequence is not found, the function will remove up to (and including) the separator character and the leading double quote will NOT be removed.
- If the separator is not found, the entire remained of the command is removed.
- Command separating character assumed to be ',', Duet standard

Parameters:

- **(1) IN/OUT** - sndcmd/str data

Returns: Parse parameter from the front of the string not including the separator.

Syntax:

```
DEFINE_FUNCTION CHAR[RMS_MAX_PARAM_LEN] RmsParseCmdParam(CHAR cCmd[])
{
  RETURN RmsParseCmdParamEx(cCmd, ',');
}
```

RMS Command Escaping Functions (Cont.)	
RmsParseCmdParamEx	*Description*: Use this function to parse out parameters from module SEND_COMMAND or SEND_STRING. Parses the strings sent to or from modules extracting the parameters. • A single param is picked of the cmd string and removed, through the separator. • The separator is NOT returned from the function. • If the first character of the param is a double quote, the function will remove up to (and including) the next double-quote and the separator without spaces. • The double quotes will then be stripped from the parameter before it is returned. • If the double-quote/separator sequence is not found, the function will remove up to (and including) the separator character and the leading double quote will NOT be removed. • If the separator is not found, the entire remained of the command is removed. *Parameters*: • **(1) IN/OUT** - sndcmd/str data • **(2) SEPARATOR** - delimiting character *Returns*: Parse parameter from the front of the string not including the separator. *Syntax*:

```
DEFINE_FUNCTION CHAR[RMS_MAX_PARAM_LEN] RmsParseCmdParamEx(CHAR cCmd[], CHAR separator)
{
  STACK_VAR CHAR cTemp[RMS_MAX_PARAM_LEN]
  STACK_VAR CHAR cSep[1]
  STACK_VAR CHAR chC
  STACK_VAR INTEGER nLoop
  STACK_VAR INTEGER nState
  STACK_VAR CHAR bInquotes
  STACK_VAR CHAR bDone
  cSep[1] = separator;

  // Reset state
  nState = 1; //ST_START
  bInquotes = FALSE;
  bDone = FALSE;

  // Loop the command and escape it
  FOR (nLoop = 1; nLoop <= LENGTH_ARRAY(cCmd); nLoop++)
  {
    // Grab characters and process it based on state machine
    chC = cCmd[nLoop];
    Switch (nState)
    {
      // Start or string: end of string bails us out
      CASE 1: //ST_START
      {
        // Starts with a quote?
        // If so, skip it, set flag and move to collect.
        IF (chC == '"')
        {
          nState = 2; //ST_COLLECT
          bInquotes = TRUE;
        }

        // Starts with a separator?  Empty param
        ELSE IF (chC == cSep)
        {
          // I am done
          bDone = TRUE;
        }

        // Not a quote or a comma?  Add it to the string and move to collection
        Else
        {
          cTemp = "cTemp, chC"
          nState = 2; //ST_COLLECT
        }
        BREAK;
      }

      // Collect string.
 CASE 2: //ST_COLLECT
      {
        // If in quotes, just grab the characters
        IF (bInquotes)
        {
          // Ah...found a quote, jump to end quote state
          IF (chC == '"' )
          {
            nState = 3; //ST_END_QUOTE
            BREAK;
          }
        }
```

RMS Command Escaping Functions (Cont.)

RmsParseCmdParam Ex (Cont.)

Syntax (Cont.):

```
        // Not in quotes, look for separator
        ELSE IF (chC == cSep)
        {
          // I am done
          bDone = TRUE;
          BREAK;
        }
        // Not in quotes, look for quotes (this would be wrong)
        // But instead of barfing, I will just add the quote (below)
        ELSE IF (chC == '"' )
        {
          // I will check to see if it should be escaped
          IF (nLoop < LENGTH_ARRAY(cCmd))
          {
            // If this is 2 quotes back to back, just include the one
            IF (cCmd[nLoop+1] = '"')
              nLoop++;
          }
        }

        // Add character to collection
        cTemp = "cTemp,chC"
        BREAK;
      }

      // End Quote
      CASE 3: //ST_END_QUOTE
      {
        // Hit a separator
        IF (chC == cSep)
        {
          // I am done
          bDone = TRUE;
        }

        // OK, found a quote right after another quote.  So this is escaped.
        ELSE IF (chC == '"')
        {
          cTemp = "cTemp,chC"
          nState = 2; //ST_COLLECT
        }
        BREAK;
      }
    }

    // OK, if end of string or done, process and exit
    IF (bDone == TRUE || nLoop >= LENGTH_ARRAY(cCmd))
    {
      // remove cTemp from cCmd
      cCmd = MID_STRING(cCmd, nLoop + 1, LENGTH_STRING(cCmd) - nLoop)

      // cTemp is done
      RETURN cTemp;
    }
  }

  // Well...we should never hit this
  RETURN "";
}
```

Unicode Strings/Commands

RMS Enterprise supports Unicode characters for international installations. The RMS NetLinx Virtual Device API supports all Send Commands sent to the RMS virtual device as NetLinx WIDECHAR character arrays. The entire command string including the command header must be sent as a WIDECHAR array.

Example:

```
// this example includes a UNICODE string.
// to send UNICODE instructions to RMS, you must
// send a WIDECHAR data type to the SEND_COMMAND
SEND_COMMAND vdvRMS, _WC('SERVICE.HELP.REQUEST- 请把帮助 ')
```

Alternatively if Unicode is not needed, the Send Command can be sent as traditional ASCII character arrays (Strings).

Example:

```
// this example includes an ASCII string.
SEND_COMMAND vdvRMS, 'SERVICE.HELP.REQUEST-Please Send Help!')
```

Channel API

RMS Client Channels

RMS Client Channels	
Channel	**Description**
250 - CLIENT ONLINE (*Feedback*)	This channel state represents the connectivity state of the RMS client with the RMS server. The channel is set to ON if the client is currently connected and online with the RMS server. *Note: The client may be connected to the RMS server and thus be online, but may not yet have been registered with the RMS server. Only after the client is registered is it allowed to participate in RMS features and functions.* This channel cannot be set, it is supplied as a feedback (read only) property.
251 - CLIENT REGISTERED (*Feedback*)	This channel state represents the registration state of the RMS client with the RMS server. The channel is set to ON if the client is currently online and registered with the RMS server. This channel cannot be set, it is supplied as a feedback (*read only*) property.
249 - ASSET INITIALIZE / REGISTRATION (*Feedback*)	This channel state represents the state of the RMS client when it is ready to accept asset registrations. This channel will become active (set to the ON state) after the RMS client establishes a connection to the RMS server and determines that the client is registered to a location. *Note: This channel will be deactivated (set to the OFF state) if the RMS client looses connectivity with the RMS server.* Assets that need to be registered and asset changes should only be communicated with the RMS client after receiving this channel event or the notification event command: "ASSETS.REGISTER". This channel cannot be set, it is supplied as a feedback (read only) property
248 - RFID INITIALIZE/REGISTRATION (Feedback)	This channel state represents the state of the RMS client when it is ready to accept RFID reader & tag registrations and tag synchronization. This channel will become active (set to the ON state) after the RMS client establishes a connection to the RMS server, determines that the client is registered to a location, and determines that the RMS system has been enabled for RFID capabilities. *Note: This channel will be deactivated (set to the OFF state) if the RMS client looses connectivity with the RMS server.* RFID readers and tags should only attempt to register or communicate with the RMS client calls after receiving this channel event or the notification event command "RFID.INITIALIZE". This channel cannot be set, it is supplied as a feedback (read only) property
247 - VERSION REQUEST	This channel receives a BUTTON PUSH event anytime the "?VERSIONS" query is submitted to the RMS client. All RMS related modules should respond to this request by printing their name and version to the master's TELNET console.
240 - CLIENT GATEWAY ASSET LICENSED (Feedback)	This channel state represents the asset licensed state of the RMS client with the RMS server. The channel is set to ON if the client currently has an asset license assigned on the RMS server. This channel cannot be set, it is supplied as a feedback (read only) property.

255 - CLIENT - SYSTEM POWER (Feedback)	This channel state represents the POWER state of the "System".
	This channel cannot be set, it is supplied as a feedback (*read only*) property.
	The following event commands are also issued for system power changes:
	`SYSTEM.POWER.ON`
	`SYSTEM.POWER.OFF`

9 - CLIENT - TOGGLE SYSTEM POWER (Momentary Function)	When this channel is activated the POWER state of the "System" will be cycled (toggled) between ON and OFF states.
27 - CLIENT - SET SYSTEM POWER ON REQUEST	This channel is represented by a PUSH BUTTON_EVENT. This event indicates that a request has been made by RMS to turn system power ON.
28 - CLIENT - SET SYSTEM POWER OFF REQUEST	This channel is represented by a PUSH BUTTON_EVENT. This event indicates that a request has been made by RMS to turn system power OFF.

Level API

RMS Hotlist Levels

RMS Hotlist Levels	
Level	**Description**
1	This level value represents the number of hotlist records available for the default client location. This level cannot be set, it is supplied as a feedback (read only) property.

RMS Client Logging - Command API

All RMS logging information is emitted in the NetLinx master's telnet console.

RMS Client Logging Instruction Commands & Queries

RMS Client Logging Instruction Commands & Queries	
Command	**Description**
LOG.LEVEL-<1|2|3|4> **LOG.LEVEL-ERROR** **LOG.LEVEL-WARNING** **LOG.LEVEL-INFO** **LOG.LEVEL-DEBUG**	This command will set the runtime logging level for the RMS client. Please note that a production system should only operate in the ERROR (1) level. Lower logging levels will decrease the performance of the NetLinx system. This setting is not persisted across reboots. To define a persistent logging level, see the CONFIG.LOG.LEVEL.DEFAULT command in the Configuration commands section. Available levels: • 1 - ERROR • 2 - WARNING • 3 - INFO • 4 - DEBUG
LOG.DEBUG	This command sets the runtime logging level for the RMS client to the DEBUG level. *Note: A production system should only operating in the ERROR (4) level.* Lower logging levels will decrease the performance of the NetLinx system.
LOG.INFO	This command sets the runtime logging level for the RMS client to the INFO level. *Note: A production system should only operating in the ERROR (4) level.* Lower logging levels will decrease the performance of the NetLinx system.
LOG.WARNING	This command sets the runtime logging level for the RMS client to the WARNING level. *Note: A production system should only operating in the ERROR (4) level.* Lower logging levels will decrease the performance of the NetLinx system.
LOG.ERROR	This command sets the runtime logging level for the RMS client to the ERROR level. *Note: A production system should only operating in the ERROR (4) level.* Lower logging levels will decrease the performance of the NetLinx system.
LOG.SHOW.LEVEL- <true|false>	This command will instruct the RMS client to set the runtime logging output to include the current logger level in the display string. After applying the updated settings, the acknowledgment response command will be returned in the following format: `LOG.SHOW.LEVEL-<true|false>` (This option is enabled by default.)
LOG.SHOW.LOGGER- <true|false>	This command will instruct the RMS client to set the runtime logging output to include the current logger package name in the display string. After applying the updated settings, the acknowledgment response command will be returned in the following format: `LOG.SHOW.LOGGER-<true|false>` (This option is enabled by default.)
LOG.SHOW.THREAD- <true|false>	This command will instruct the RMS client to set the runtime logging output to include the current thread context name in the display string. After applying the updated settings, the acknowledgment response command will be returned in the following format: `LOG.SHOW.THREAD-<true|false>` (This option is disabled by default.)
LOG.ENABLE.ALL	This command will instruct the RMS client to enable the runtime logging output for all RMS logging packages
LOG.DISABLE.ALL	This command will instruct the RMS client to disable the runtime logging output for all RMS logging packages

LOG.LOGGER.ENABLED-
<logger-name>,
<logger-enabled>
	This command will instruct the RMS client to enable the runtime logging output for a single specified RMS logging package.
	`<logger-name>`
	This is the name of the logger package that is writing RMS log information.
	`<logger-enabled>`
	This current enabled or disabled state to set the RMS logger package. The value should be set to "true" or "false".

RMS Client Logging Query Commands

All RMS logging information is emitted in the NetLinx master's telnet console.

RMS Client Logging Query Commands	
Command	**Description**
?LOG.LEVEL	This command will query the RMS client to determine the current runtime logging level. The query response command will be returned in the following format: `LOG.LEVEL-<1|2|3|4>` The numerical representation of the current logging level will be returned.
?LOG.SHOW.LEVEL	This command will query the RMS client to determine if the runtime logging output will include the current level in the display string. The query response command will be returned in the following format: `LOG.SHOW.LEVEL-<true|false>` This option is *enabled* by default.
?LOG.SHOW.LOGGER	This command will query the RMS client to determine if the runtime logging output will include the current logger name in the display string. The query response command will be returned in the following format: `LOG.SHOW.LOGGER-<true|false>` This option is *enabled* by default.
?LOG.SHOW.THREAD	This command will query the RMS client to determine if the runtime logging output will include the current thread context name in the display string. The query response command will be returned in the following format: `LOG.SHOW.THREAD-<true|false>` This option is *disabled* by default.
?LOG.LOGGERS	This command will query the RMS client to return a listing of all the known RMS loggers and their current state. The query response command will be return a series of response strings for each logger in the following format: `LOG.LOGGER-<logger-name>,<logger-enabled>` This option is *disabled* by default.

?LOG.LOGGER.ENABLED-<logger-name>	This command will query the RMS client to determine if a specified logger package name is enabled or disabled.
	The query response command will be returned in the following format:
	`LOG.LOGGER-<logger-name>,<logger-enabled>`
	`<logger-name>`
	This is the name of the logger package that is writing RMS log information.
	`<logger-enabled>`
	This displays the current enabled or disabled state of the RMS logger package.
	The value is returned as "true" or "false"

RMS Client Exceptions / Errors - Command API

RMS Client Error/Exception Notification Event Command

RMS Client Error/Exception Notification Event Command	
Command	**Description**

EXCEPTION-
<exception-message>
(,<command-header>)
	This command response will be emitted whenever a command instruction sent to the RMS NetLinx adapter virtual device fails. If a command header is available, it will be included as a second argument on the event string.
	Example:
	`EXCEPTION-no hotlist records in buffer,`
	`?HOTLIST.RECORD.ID`
	The "`?HOTLIST.RECORD.ID`" was the command instruction issued that failed.
	In this example, the command failed because the "`?HOTLIST.RECORDS`" command was never issued to fetch the hotlist records from the RMS server.

RMS Client Management - Command API

RMS Client Query Commands

RMS Client Query Commands	
Query Command	**Description / Response Commands**

?VERSIONS	This command will query all the components in the RMS client to display each component's version information.
	The results of this query are not emitted as a command events, but rather are only displayed in the console output of the NetLinx master's telnet console.
	The output may look something like this:
	`(0060927393) ... querying all RMS components to display their versions ...`
	`(0060927398) RMS Server : 4.0.0`
	`(0060927401) RMS Database : 4.0.0`
	`(0060927405) RMS Client : 4.0.0`

?SERVER.VERSION	This command will query the RMS client to return the RMS server's version. The response command will be returned in the following format: `SERVER.VERSION-<version-string>`
?SERVER.TIMESYNC.ENABLED	This command will query the RMS client to determine if the RMS system is configured to provide time synchronization to the RMS clients. The response command will be returned in the following format: `SERVER.TIMESYNC.ENABLED-<true|false>` *Note: If Time Synchronization is not enabled on the RMS server, then it is the responsibility of the system administrator to ensure that the Location in which the controller resides is configured to use the same TimeZone as the controller.* If Location TimeZones don't match the server's Timezone, scheduling date and times will be calculated incorrectly.
?DATABASE.VERSION	This command will query the RMS client to return the RMS database's version. The response command will be returned in the following format: `DATABASE.VERSION-<version-string>`
?CLIENT.VERSION	This command will query the RMS client to return the RMS client module version. The response command will be returned in the following format: `CLIENT.VERSION-<version-string>`

?CLIENT	This command will query the RMS client to return the RMS client gateway's properties. The response command will be returned in the following format:
	`CLIENT-<unique-identifier>,`
	`<name>,`
	`<mac-address>,`
	`<hostname>,`
	`<ip-address>,`
	`<subnet-mask>,`
	`<gateway-address>`

?CLIENT.UID	This command will query the RMS client to return the RMS client gateway's unique identifier. The response command will be returned in the following format: `CLIENT.UID-<unique-identifier>`
?CLIENT.NAME	This command will query the RMS client to return the RMS client gateway's name property. The response command will be returned in the following format: `CLIENT.NAME-<name>`
?CLIENT.GATEWAY	This command will query the RMS client to return the RMS client gateway's TCP/IP gateway address. The response command will be returned in the following format: `CLIENT.GATEWAY-<gateway-address>`
?CLIENT.HOSTNAME	This command will query the RMS client to return the RMS client gateway's TCP/IP hostname. The response command will be returned in the following format: `CLIENT.HOSTNAME-<hostname>`
?CLIENT.IP	This command will query the RMS client to return the RMS client gateway's TCP/IP address. The response command will be returned in the following format: `CLIENT.IP-<ip-address>`
?CLIENT.MAC	This command will query the RMS client to return the RMS client gateway's network interface MAC address. The response command will be returned in the following format: `CLIENT.MAC-<mac-address>`
?CLIENT.SUBNET	This command will query the RMS client to return the RMS client gateway's TCP/IP subnet address. The response command will be returned in the following format: `CLIENT.SUBNET-<subnet-mask>`

RMS Client Query Commands (Cont.)	
Query Command	**Description / Response Commands**
?CLIENT.ONLINE	This command will query the RMS client to determine if the client is currently connected and online with the RMS server. The response command will be returned in the following format: `CLIENT.ONLINE-<true|false>` Please note that the client may be connected to the RMS server and thus be online, but may not yet have been registered with the RMS server. Only after the client is registered is it allowed to participate in RMS features and functions.
?CLIENT.OFFLINE	This command will query the RMS client to determine if the client is currently disconnected and offline with the RMS server. The response command will be returned in the following format: `CLIENT.OFFLINE-<true|false>`
?CLIENT.REGISTERED	This command will query the RMS client to determine if the client has been accepted and registered on the RMS server. The response command will be returned in the following format: `CLIENT.REGISTERED-<true|false>`
?CLIENT.CONNECTION.STATE	This command will query the RMS client to determine the current connection state. The response command will be returned in the following format: `CLIENT.CONNECTION.STATE-<state-string>` The following connections states are available: • INIT - The client is in this state when the system first boots up. This state is prior to the RMS client reading it's configuration settings. • DISABLED - The client enters this state after the client configuration has been loaded and determines that the client is either disabled or the necessary configuration settings required to connect to a RMS server are missing. • OFFLINE - The client enters this state after the client configuration has been loaded and determines that the client is enabled. In this state the client will routinely attempt to connect to the configured RMS server. The client may also enter this state if the connection with the RMS server has been lost. • CONNECT-SERVER - The client enters this state when attempting to connect to the RMS server. If the connection is successful, the client transitions to the CONNECT-CLIENT state, otherwise to the CONNECT-FAIL state. • CONNECT-CLIENT - The client enters this state when after successfully connecting to the RMS server in the CONNECT-SERVER state. During this state the client will attempt to register the client gateway information with the RMS server. If the connection and client gateway registration is successful, the client transitions to the CONNECT-LOCATION state, otherwise to the CONNECT-FAIL state. • CONNECT-LOCATION - The client enters this state when after successfully connecting to the RMS server and registering the client gateway information in the CONNECT-CLIENT state. During this state the client will attempt to obtain the default client configured RMS location from the RMS server. If a valid location is returned is successfully, the client transitions to the ONLINE state, otherwise if a location is not returned, meaning that this client gateway has not been registered to a location in the RMS system, the state will transition to the ONLINE-UNREGISTERED state. • CONNECT-FAIL - The client enters this state if at any point the connection to the RMS server cannot be established or is lost. After entering this state, the client will automatically transition back to the OFFLINE state and wait for a re-connection attempt. • ONLINE - The client enters this state after successfully connecting to the RMS server, registering the client gateway information, and obtaining a valid RMS location from the server. The client will remain in this state and periodically request any pending client destined messages. If a connection attempt or request fails to the RMS server, the client will transition to the CONNECT-FAIL state. • ONLINE-UNREGISTERED - The client enters this state when after successfully connecting to the RMS server, registering the client gateway information, and failing to obtain a valid RMS location from the server. The client will remain in this state and periodically request any pending client destined messages. If a connection attempt or request fails to the RMS server, the client will transition to the CONNECT-FAIL state. While in this state and processing received client messages, all messages will be ignored with the exception of the REINITIALIZE request message. If a REINITIALIZE request is received, then the client will transition to the REINITIALIZE state to begin a client re-initialization. A REINITIALIZE request is sent by the server when a client gateway is registered to a location in the RMS system. • REINITIALIZE - The client enters this state anytime a REINITIALIZE request is received from the RMS server. A REINITIALIZE request is sent by the server when a client gateway is registered to a location in the RMS system. After entering this state, the client will automatically transition to the CONNECT-CLIENT state to renegotiate online status with the RMS server. Note that a connection state diagram is provided in Appendix A (see FIG. 17 on page 51).

RMS Client Instruction Commands

RMS Client Instruction Commands	
Command	**Description**
CLIENT.CONNECT	This command will force the RMS client system to immediately connect to the RMS server. This command will only function if (a) not already connected to a RMS server and (b) the RMS client is enabled in the RMS client configuration.
CLIENT.DISCONNECT	This command will force the RMS client system to immediately disconnect from the RMS server. This command will only function if the client is already connected to a RMS server.
CLIENT.REINIT	This command will force the RMS client system to immediately disconnect from the RMS server and then reestablish a connection to the RMS server and re-initialize all asset registrations. This command will only function if the client is already connected to a RMS server.
CLIENT.MESSAGES.RETRIEVE	This command will force the RMS client system to immediately retrieve and process all pending client-destined messages from the RMS server. This command will only function if the client is already connected to a RMS server.
CLIENT.TIMESYNC	This command will invoke a time synchronization to take place between the client system and RMS server. When complete the client system will be set to the time of the server relative to the client assigned location time zone offset.

RMS Client Event Notification Commands

RMS Client Event Notification Commands	
Command	**Description**
CLIENT.ONLINE	This event notification command will be sent when the RMS client connects to the RMS server and enters the ONLINE or ONLINE-UNREGISTERED state.
CLIENT.REGISTERED	This event notification command will be sent when the RMS client connects to the RMS server and enters the ONLINE state and is registered on the RMS server.
CLIENT.OFFLINE	This event notification command will be sent when the RMS client enters the OFFLINE state from the INIT state or if the RMS client is not able to communicate with the RMS server and transitions to the CONNECT-FAIL and then OFFLINE states.
CLIENT.CONNECTION.STATE.TRANSITION- <old-state-string>, <new-state-string>	This event notification command will be sent when the RMS client transitions to each connection state. The states are listed in the CLIENT.CONNECTION.STATE Send Command. Note that a connection state diagram is provided in Appendix A (see FIG. 17 on page 51).
VERSIONS	This event notification command is issued when the '?VERSIONS' request is issued in the RMS client. All RMS modules should respond to this request by printing their module name and version to the master's telnet console.
SYSTEM.POWER.ON	This event notification command is issued when the system power is turned on. The user implementation code should implement the logic for this event notification to perform the necessary system startup procedure. This event notification is also communicated via channel 255.
SYSTEM.POWER.OFF	This event notification command is issued when the system power is turned off. The user implementation code should implement the logic for this event notification to perform the necessary system shutdown procedure. This event notification is also communicated via channel 255.
SERVER.INFO- <rms-app-version>, <rms-database-version>, <is-timesync-enabled>, <is-smtp-enabled>, <is-rfid-tracking-enabled>, <min-poll-time>, <max-poll-time>	This event notification command is issued on initial connect to the server where server information is automatically retrieved and any subsequent server information updates received from the server when a server setting is altered from the RMS web application.

RMS Location Information - Command API

RMS Default Location Queries	
Command	**Description**

?CLIENT.LOCATION	This command will query the RMS client to determine the default client assigned location information.
	The response command will be returned in the following format:
	`CLIENT.LOCATION-<location-id>,`
	`<location-name>,`
	`<location-owner>,`
	`<location-phone-number>,`
	`<location-occupancy>,`
	`<location-prestige-name>,`
	`<location-timezone>,`
	`<location-asset-licensed>`
	Please see the location properties below for a description of each.

?CLIENT.LOCATION.NAME	This command will query the RMS client to determine the default client assigned location name property. The response command will be returned in the following format: `CLIENT.LOCATION.NAME-<name-string>`
?CLIENT.LOCATION.ID	This command will query the RMS client to determine the default client assigned location ID property. The response command will be returned in the following format: `CLIENT.LOCATION.ID-<id-number>`
?CLIENT.LOCATION.OWNER	This command will query the RMS client to determine the default client assigned location owner property. The response command will be returned in the following format: `CLIENT.LOCATION.OWNER-<owner>`
?CLIENT.LOCATION.PHONE.NUMBER	This command will query the RMS client to determine the default client assigned location phone number property. The response command will be returned in the following format: `CLIENT.LOCATION.PHONE.NUMBER-<phone-number>`
?CLIENT.LOCATION.TIMEZONE	This command will query the RMS client to determine the default client assigned location timezone property. The response command will be returned in the following format: `CLIENT.LOCATION.TIMEZONE-<timezone-string>` The timezone string is provided in the Java timezone ID string format. (http://mindprod.com/applet/tz.html). *Note: If Time Synchronization is not enabled on the RMS server, then it is the responsibility of the system administrator to ensure that the Location in which the controller resides is configured to use the same TimeZone as the controller. If Location TimeZones don't match the server's Timezone, scheduling date and times will be calculated incorrectly.*
?CLIENT.LOCATION.OCCUPANCY	This command will query the RMS client to determine the default client assigned location occupancy property. The response command will be returned in the following format: `CLIENT.LOCATION.OCCUPANCY-<occupancy-number>`
?CLIENT.LOCATION.PRESTIGE	This command will query the RMS client to determine the default client assigned location prestige name property. The response command will be returned in the following format: `CLIENT.LOCATION.PRESTIGE-<prestige-name-string>`
?CLIENT.LOCATION.ASSET.LICENSED	This command will query the RMS client to determine the default client assigned location asset license status. The response command will be returned in the following format: `CLIENT.LOCATION.ASSET.LICENSED-<true|false>`

RMS Default Location Event Notification Commands

RMS Default Location Event Notification Commands	
String	**Description**
LOCATION- <is-client-default-location>, <location-id>, <location-name>, <location-owner>, <location-phone-number>, <location-occupancy>, <location-prestige-name>, <location-timezone>, <location-asset-licensed>	During the RMS client connection and registration initialization sequence, if a default client gateway location is detected, this event notification command will be advertised providing listeners with the location information. • If no location is available, the event notification command will be sent, but no location parameters will be defined. • This event notification command will also be issued anytime the client default location is modified via the RMS web based user interface. • This event notification command may also be issued for changes to alternate locations (locations that are not the default client gateway's assigned location). This will occur if the client gateway is supporting assets that have been relocated to other locations in the RMS system. • The first argument of <is-client-default-location> is key to determining if this location notification is for the default client gateway location or not. • The first argument <is-client-default-location> identifies if this notification command contains the location information for the default location assigned to this client gateway.

RMS Client Settings & Configuration - Command API

Please note that the required configuration settings needed to get the client system communicating with the RMS server are now also provided via a web based configuration page hosted on the NetLinx systems web server.

RMS Client Configuration Instruction Commands

RMS Client Configuration Instruction Commands	
Command	**Description**
CONFIG.CLIENT.ENABLED- <true|false> **CONFIG.CLIENT.ENABLED-** <1|0>	This command sets the enabled state of the RMS client system.
CONFIG.SERVER.URL- <url>	This command will set the URL used to access the RMS server. Include the port number if using a non-standard IP port to serve the RMS server application. Example: `SEND_COMMAND vdvRMS, 'CONFIG.SERVER.URL-http:\\server\rms'`
CONFIG.SERVER.PASSWORD- <password>	This command will set the client access password credentials used when communicating with the RMS server.
CONFIG.SERVER.RETRY.INTERVAL- <#> (# : number of seconds)	This command will set the number of seconds to wait between connection attempts to the RMS server. This settings is used during the *GREEDY* connection attempt period. • The default retry interval is 30 seconds. • The minimum retry interval allowed is 30 seconds.
CONFIG.SERVER.RETRY.COUNT- <#> (# : number of attempts)	This command will set the number of attempts made to establish a connection to the RMS server before the connection state switched to the *SETBACK* connection attempt period. • The default retry attempt count is 10. • The minimum retry attempt count allowed is 5
CONFIG.SERVER.RETRY.SETBACK.INTERVAL- <#> (# : number of seconds)	This command will set the number of seconds to wait between connection attempts to the RMS server while in the *SETBACK* connection attempt period. This should be set to a higher delay time than the *GREEDY* connection attempt interval. • The default retry interval is 300 seconds (*5 minutes*). • The minimum retry interval allowed is 30 seconds.
CONFIG.CLIENT.NAME- <name>	This command will set the default system name displayed in RMS when this system comes online.
CONFIG.CLIENT.HEARTBEAT- <#> (# : number of seconds)	This command will set the number of seconds to wait between client pending message retrieval requests issued to the RMS server. These requests are also used to let the server know that the system is still alive and online. Please be aware that the higher the frequency of these requests the greater the load placed on the RMS server. This frequency can be adjusted higher to optimize server performance. *Note: The default heartbeat (and the minimum heartbeat interval allowed) interval is 15 seconds.*
CONFIG.SERVER.TEST	This command can be issued to perform a test connection to the RMS server. This test can help identify if the configured server URL and access credentials are configured property. *Note: This is the same function as using the TEST button on the RMS web configuration page.*
CONFIG.LOG.LEVEL.DEFAULT-<1|2|3|4> **CONFIG.LOG.LEVEL.DEFAULT-ERROR** **CONFIG.LOG.LEVEL.DEFAULT-WARNING** **CONFIG.LOG.LEVEL.DEFAULT-INFO** **CONFIG.LOG.LEVEL.DEFAULT-DEBUG**	This command will set the default logging level for the RMS client. *Note: A production system should only operate in the ERROR (4) level. Lower logging levels will decrease the performance of the NetLinx system.* This setting is persisted across reboots and is used to set the default runtime logging level for RMS at system startup. Available levels: • 1 - ERROR • 2 - WARNING • 3 - INFO • 4 - DEBUG

RMS Client Configuration Query Commands

RMS Client Configuration Query Commands	
Query Command	**Description / Response Command**
?CONFIG.CLIENT.ENABLED- <true|false>	This query command determines the enabled state of the RMS client system. Response command format: `CONFIG.CLIENT.ENABLED-<true|false>`
?CONFIG.SERVER.URL	This query command determines the URL used to access the RMS server. Response command format: `CONFIG.SERVER.URL-<url>`

?CONFIG.SERVER.
RETRY.INTERVAL
	This query command determines the number of seconds to wait between connection attempts to the RMS server. This settings is used during the *GREEDY* connection attempt period.
	Response command format:
	`CONFIG.SERVER.RETRY.INTERVAL-<#>`
	`(# : number of seconds)`

?CONFIG.SERVER.
RETRY.COUNT
	This query command determines the number of attempts made to establish a connection to the RMS server before the connection state switched to the *SETBACK* connection attempt period.
	Response command format:
	`CONFIG.SERVER.RETRY.COUNT-<#>`
	`(# : number of attempts)`

?CONFIG.SERVER.SETBACK.
RETRY.INTERVAL
	This query command determines the number of seconds to wait between connection attempts to the RMS server while in the *SETBACK* connection attempt period.
	This should be set to a higher delay time than the *GREEDY* connection attempt interval.
	Response command format:
	`CONFIG.SERVER.SETBACK.RETRY.INTERVAL-<#>`
	`(# : number of seconds)`

?CONFIG.CLIENT.NAME	This query command determines the default system name displayed in RMS when this system comes online. Response command format: `CONFIG.CLIENT.NAME-<name>`

?CONFIG.CLIENT.HEARTBEAT	This query command determines the number of seconds to wait between client pending message retrieval requests issued to the RMS server.
	Response command format:
	`CONFIG.CLIENT.HEARTBEAT-<#>`
	`(# : number of seconds).`

?CONFIG.LOG.LEVEL.DEFAULT	This query command will determine the default logging level for the RMS client. Please note that a production system should only operating in the ERROR (4) level. Lower logging levels will decrease the performance of the NetLinx system. Response command format: `CONFIG.LOG.LEVEL.DEFAULT-<1|2|3|4>` Available levels: 1 - ERROR 2 - WARNING 3 - INFO 4 - DEBUG

RMS Client Configuration Event Notification Commands

RMS Client Configuration Event Notification Commands	
Command	**Description**
CONFIG.CHANGE- <property-name>, <property-value>	This event notification command will be sent when a RMS client configuration setting is changed.

RMS Status Type Management - Command API

RMS Status Type Registration Commands

RMS Status Type Registration Commands	
Command	**Description**
STATUS.TYPE- <status-type-key>, <name> (,<description>) (,<appear-on-hotlist>) (,<hotlist-priority-level>)	This command is used to register a new status type on the RMS system. • This command will create a new status type record and make the new status type available in the RMS User Interface and available for asset registration. • This command will not update the status type if an existing status type record by this key already exists in the RMS system. Arguments: • **<status-type-key>:** The globally unique string to identify the new status type. This key can later be used to register assets parameter thresholds to this new status type. • **<name>**: The name of the new status type. • **(<description>)**: This argument is *optional* . Descriptive text for the new status type to help users understand the meaning and context of what it is intended to convey. • **(<appear-on-hotlist>)**: This argument is *optional* . This argument can be set to "true" or "false" depending on if tripped parameter thresholds should be listed in the RMS hotlist. • **(<hotlist-priority-level>)**: This argument is *optional* . This argument is an integer representing the priority level of this system type. Priority levels are used in the RMS user interface when sorting the hotlist records by status type. **Note**: Status types can be removed from the RMS User Interface, but cannot be removed by the SDK.

RMS Status Types Query Commands

RMS Status Types Query Commands	
Query Command	**Description / Response Commands**

?STATUS.TYPE	This command is used to query the RMS system for a listing of all status types.
	The response command will be returned in the following format:
	First a total count is returned.
	`STATUS.TYPE.RECORD-<record-count>,`
	If the record count is greater than zero, then a series of the following command responses will be sent:
	`STATUS.TYPE.RECORD-<record-index>,`
	`<record-count>,`
	`<status-type-key>,`
	`<name>,`
	`<description>,`
	`<appear-on-hotlist>,`
	`<hotlist-priority-level>,`
	`<system-defined>`

?STATUS.TYPE-
<status-type-key>
	This command is used to query the RMS system for a specific status type record by status type key.
	`<status-type-key>`
	The status type key of the status type record you wish to query must be included.
	This status type key is the unique programming identifier for a given status type.
	`STATUS.TYPE-<status-type-key>,`
	`<name>,`
	`<description>,`
	`<appear-on-hotlist>,`
	`<hotlist-priority-level>,`
	`<system-defined>`

RMS Asset Registration - Command API

RMS Asset Registration Event Notification Commands

RMS Asset Registration Event Notification Commands	
String	**Description**
ASSETS.REGISTER	This event notification command is sent by the RMS client when it is ready for assets to begin registration with the RMS client. Upon receiving this event notification command, any asset wishing to participate in the RMS system should perform asset registration.
ASSET.REGISTERED- <asset-client-key>, <asset-id>, <new-registration>	This event notification command is sent by the RMS client when an asset registration has been completed. Upon receiving this event string, any asset can begin registering/updating metadata properties, control functions, or parameters.
ASSET.LOCATION.CHANGE- <asset-client-key>, <asset-id>, <new-location-id>	This event notification command is sent by the RMS client when an asset location change has been made on the RMS server.

RMS Asset Registration Commands

RMS Asset Registration Commands	
Command	**Description**
ASSET.REGISTER.DEV-<D:P:S>, <asset-name>, (,<asset-client-key>) (,<asset-type-key>) (,<asset-global-key>)	This command is used to register a NetLinx-based device asset with the RMS system. If the device uses a NetLinx DPS, then this call should be used to register the asset. This command will queue the asset for registration. Additional asset properties can be set using the asset command set provided in this section. When ready to submit the asset registration, use the ASSET.SUBMIT command. Arguments: • **<D:P:S>**: The DPS string should be provided here in the format: *Device:Port:System*. • **<asset-name>**: This argument should contain a friendly name for the asset. This name can be later changed by the user in the RMS system. • **<asset-client-key>**: This argument is *optional* and typically not set for a NetLinx-device based asset registration. This key represents an asset identifier that is unique on this client system (*does not have to be globally unique*). If not provided, the DPS string is used as the client key. This asset client key is typically the D:P:S string in NetLinx systems. • **(<asset-type-key>)**: This argument is *optional* but it is recommended to set this to an appropriate asset type key. The asset types are listed in the RMS web user interface under the Settings menu. Please note, this field must contain the asset type key, not the asset type name. (For Duet devices, the asset type key is the same as the device interface.) • **<asset-global-key>**: This argument is *optional* and should only be set if the asset contains some unique identifier that can be assured to be globally unique across all assets in all RMS locations. Properties such as a MAC address which are globally unique could be used. Please note that an asset serial number by itself is not guaranteed to be unique across multiple vendors. Thus if attempting to use a serial number it may need to be prefixed with some additional information such as manufacturer & model. If an asset is registered with a global key, the RMS system can detect if the asset gets relocated to a different location anywhere in the RMS system.
ASSET.REGISTER.AMX.DEV-<D:P:S>, (,<asset-name>) (,<asset-client-key>) (,<asset-type-key>) (,<asset-global-key>)	This command is used to register an AMX manufactured NetLinx-based device asset with the RMS system. If the device uses a NetLinx DPS and is manufactured by AMX, then this call should be used to register the asset. This command will queue the asset for registration. This command will pre-populate all available asset fields with information obtained from the NetLinx device info query. Additional asset properties can be set using the asset command set provided in this section. When ready to submit the asset registration, use the ASSET.SUBMIT command. Arguments: • **<D:P:S>:** The DPS string should be provided here in the format: Device:Port:System. • **<asset-name>**: This argument is *optional* but it is recommended to provide an explicit identifying friendly name appropriate for the asset. This argument should contain a friendly name for the asset. This name can be later changed by the user in the RMS system. • **<asset-client-key>**: This argument is *optional* and typically not set for a NetLinx-device based asset registration. This key represents an asset identifier that is unique on this client system (*does not have to be globally unique*). If not provided, the DPS string is used as the client key. This asset client key is typically the D:P:S string in NetLinx systems. • **(<asset-type-key>)**: This argument is *optional* but it is recommended to set this to an appropriate asset type key. The asset types are listed in the RMS web user interface under the Settings menu. Please note, this field must contain the asset type key, not the asset type name. (For Duet devices, the asset type key is the same as the device interface.) • **<asset-global-key>**: This argument is *optional* and should only be set if the asset contains some unique identifier that can be assured to be globally unique across all assets in all RMS locations. Properties such as a MAC address which are globally unique could be used. Please note that an asset serial number by itself is not guaranteed to be unique across multiple vendors. Thus if attempting to use a serial number it may need to be prefixed with some additional scoping information such as manufacturer & model. If an asset is registered with a global key, the RMS system can detect if the asset gets relocated to a different location anywhere in the RMS system.

RMS Asset Registration Commands (Cont.)	
Command	**Description**
ASSET.REGISTER- <asset-name>, <asset-client-key> (,<asset-type-key>) (,<asset-global-key>)	This command is used to register a non-NetLinx-based asset with the RMS system. If a device or asset does not have an associated NetLinx DPS, then this call should be used to register the asset. This command will queue the asset for registration. Additional asset properties can be set using the asset command set provided in this section. When ready to submit the asset registration, use the ASSET.SUBMIT command. Arguments: • **<asset-name>**: This argument should contain a friendly name for the asset. This name can be later changed by the user in the RMS system. • **<asset-client-key>**: This argument is *required*. The value for this field represents an asset identifier that is unique on this client system (*does not have to be globally unique*). Any naming or numbering system can be used as long as the identifier is unique for each asset on the client system and is persisted across system/device reboots & firmware upgrades. • **<asset-type-key>**: This argument is *optional* but it is recommended to set this to an appropriate asset type key. The asset types are listed in the RMS web user interface under the Settings menu. Please note, this field must contain the asset type key, not the asset type name. (For Duet devices, the asset type key is the same as the device interface.) • **<asset-global-key>**: This argument is *optional* and should only set if the asset contains some unique identifier that can be assured to be globally unique across all assets in all RMS locations. Properties such as a MAC address which are globally unique could be used. Please note that an asset serial number by itself is not guaranteed to be unique across multiple vendors. Thus if attempting to use a serial number it may need to be prefixed with some additional scoping information such as manufacturer & model. If an asset is registered with a global key, the RMS system can detect if the asset gets relocated to a different location anywhere in the RMS system.
ASSET.DESCRIPTION- <asset-client-key>, <description>	This command sets the asset description property on an asset that has been queued for registration. The asset client key must be provided to perform the set on the specific asset. If the ASSET.REGISTER.DEV command was used to register a NetLinx-based device and no explicit asset client key was specified, then the asset client key will be the DPS string. Arguments: • **<asset-client-key>**: This argument is *required*. The value for this field represents an asset identifier that is unique on this client system (*does not have to be globally unique*). Any naming or numbering system can be used as long as the identifier is unique for each asset on the client system and is persisted across system/device reboots & firmware upgrades.
ASSET.NAME- <asset-client-key>, <asset-name>	This command sets the asset name property on an asset that has been queued for registration. The asset client key must be provided to perform the set on the specific asset. If the ASSET.REGISTER.DEV command was used to register a NetLinx-based device and no explicit asset client key was specified, then the asset client key will be the DPS string. Arguments: • **<asset-client-key>**: This argument is *required*. The value for this field represents an asset identifier that is unique on this client system (*does not have to be globally unique*). Any naming or numbering system can be used as long as the identifier is unique for each asset on the client system and is persisted across system/device reboots & firmware upgrades. • **<asset-name>**: This argument should contain a friendly name for the asset. This name can be later changed by the user in the RMS system.

RMS Asset Registration Commands (Cont.)	
Command	**Description**
ASSET.GLOBAL.KEY- <asset-client-key>, <asset-global-key>	This command sets the asset global key property on an asset that has been queued for registration. The asset client key must be provided to perform the set on the specific asset. If the ASSET.REGISTER.DEV command was used to register a NetLinx-based device and no explicit asset client key was specified, then the asset client key will be the DPS string. This property is *optional* and should only be set if the asset contains some unique identifier that can be assured to be globally unique across all assets in all RMS locations. Properties such as a MAC address which are globally unique could be used. Please note that an asset serial number by itself is not guaranteed to be unique across multiple vendors. Thus if attempting to use a serial number it may need to be prefixed with some additional scoping information such as manufacturer & model. If an asset is registered with a global key, the RMS system can detect if the asset gets relocated to a different location anywhere in the RMS system. Arguments: • **<asset-client-key>**: This argument is *required*. The value for this field represents an asset identifier that is unique on this client system (*does not have to be globally unique*). Any naming or numbering system can be used as long as the identifier is unique for each asset on the client system and is persisted across system/device reboots & firmware upgrades. • **<asset-global-key>**: This argument is *optional* and should only set if the asset contains some unique identifier that can be assured to be globally unique across all assets in all RMS locations. Properties such as a MAC address which are globally unique could be used. Please note that an asset serial number by itself is not guaranteed to be unique across multiple vendors. Thus if attempting to use a serial number it may need to be prefixed with some additional scoping information such as manufacturer & model. If an asset is registered with a global key, the RMS system can detect if the asset gets relocated to a different location anywhere in the RMS system.
ASSET.TYPE- <asset-client-key>, <asset-type-key>	This command sets the asset type key property on an asset that has been queued for registration. The asset client key must be provided to perform the set on the specific asset. If the ASSET.REGISTER.DEV command was used to register a NetLinx-based device and no explicit asset client key was specified, then the asset client key will be the DPS string. This property is *optional* but it is recommended to set this to an appropriate asset type key. The asset types are listed in the RMS web user interface under the Settings menu. Please note, this field must contain the asset type key, not the asset type name. (For Duet devices, the asset type key is the same as the device interface.) Arguments: • **<asset-client-key>**: This argument is *required*. The value for this field represents an asset identifier that is unique on this client system (*does not have to be globally unique*). Any naming or numbering system can be used as long as the identifier is unique for each asset on the client system and is persisted across system/device reboots & firmware upgrades. • **<asset-type-key>**: This argument is *optional* but it is recommended to set this to an appropriate asset type key. The asset types are listed in the RMS web user interface under the Settings menu. Please note, this field must contain the asset type key, not the asset type name. (For Duet devices, the asset type key is the same as the device interface.)
ASSET.SERIAL- <asset-client-key>, <serial-number>	This command sets the asset serial number property on an asset that has been queued for registration. The asset client key must be provided to perform the set on the specific asset. If the ASSET.REGISTER.DEV command was used to register a NetLinx-based device and no explicit asset client key was specified, then the asset client key will be the DPS string. Arguments: • **<asset-client-key>**: This argument is *required*. The value for this field represents an asset identifier that is unique on this client system (*does not have to be globally unique*). Any naming or numbering system can be used as long as the identifier is unique for each asset on the client system and is persisted across system/device reboots & firmware upgrades.
ASSET.MANUFACTURER- <asset-client-key>, <asset-manufacturer-name> (,<asset-manufacturer-url>)	This command sets the asset manufacturer name and URL properties on an asset that has been queued for registration. The asset client key must be provided to perform the set on the specific asset. If the ASSET.REGISTER.DEV command was used to register a NetLinx-based device and no explicit asset client key was specified, then the asset client key will be the DPS string. Arguments: • **<asset-client-key>**: This argument is *required*. The value for this field represents an asset identifier that is unique on this client system (*does not have to be globally unique*). Any naming or numbering system can be used as long as the identifier is unique for each asset on the client system and is persisted across system/device reboots & firmware upgrades. The manufacturer URL property is *optional*.

RMS Asset Registration Commands (Cont.)	
Command	**Description**
ASSET.MODEL- <asset-client-key>, <asset-model-name> (,<asset-model-url>)	This command sets the asset model name and URL properties on an asset that has been queued for registration. The asset client key must be provided to perform the set on the specific asset. If the ASSET.REGISTER.DEV command was used to register a NetLinx-based device and no explicit asset client key was specified, then the asset client key will be the DPS string. Arguments: • **<asset-client-key>**: This argument is *required*. The value for this field represents an asset identifier that is unique on this client system (*does not have to be globally unique*). Any naming or numbering system can be used as long as the identifier is unique for each asset on the client system and is persisted across system/device reboots & firmware upgrades. The model URL property is optional.
ASSET.FIRMWARE- <asset-client-key>, <firmware-version>	This command sets the asset firmware version property on an asset that has been queued for registration. The asset client key must be provided to perform the set on the specific asset. If the ASSET.REGISTER.DEV command was used to register a NetLinx-based device and no explicit asset client key was specified, then the asset client key will be the DPS string. Arguments: • **<asset-client-key>**: This argument is *required*. The value for this field represents an asset identifier that is unique on this client system (*does not have to be globally unique*). Any naming or numbering system can be used as long as the identifier is unique for each asset on the client system and is persisted across system/device reboots & firmware upgrades.
ASSET.SUBMIT- <asset-client-key>	This command submits a queued asset registration to the RMS client. This command should follow either the ASSET.REGISTER.DEV command or the ASSET.REGISTER command and all asset property set commands. This command will finalize the asset registration and remove the asset details from the registration queue. Arguments: • **<asset-client-key>**: This argument is *required*. The value for this field represents an asset identifier that is unique on this client system (*does not have to be globally unique*). Any naming or numbering system can be used as long as the identifier is unique for each asset on the client system and is persisted across system/device reboots & firmware upgrades.
ASSET.EXCLUDE- <asset-client-key>(,<exclude>)	This command will add the asset to the asset registration exclusion list. If an asset registration attempt is made after the asset has been added to the exclusion list, the asset registration will be ignored. This command must be issued to RMS prior to the asset registration attempt. Arguments: • **<asset-client-key>**: This argument is *required*. The value for this field represents an asset identifier that is unique on this client system (*does not have to be globally unique*). Any naming or numbering system can be used as long as the identifier is unique for each asset on the client system and is persisted across system/device reboots & firmware upgrades. • **<exclude>**: This parameter is optional and if not included the default value assumed is TRUE. If set to TRUE, then the asset will be added to the asset registration exclusion list. If set to FALSE the asset will be removed from the exclusion list.

RMS Asset Location - Command API

RMS Asset Location Query Commands

RMS Asset Location Query Commands	
Query Command	**Description / Response Commands**

?ASSET.LOCATION-
<asset-client-key>
	This command is used to query the RMS server for asset location information for a specific asset.
	<asset-client-key>: The asset client key of the asset you wish to query must be included. This asset client key is typically the D:P:S string in NetLinx systems.
	The following response command will be returned in response to this query.
	`ASSET.LOCATION-<asset-client-key>,`
	`<location-id>,`
	`<location-name>,`
	`<location-owner>,`
	`<location-phone-number>,`
	`<location-occupancy>,`
	`<location-prestige-name>,`
	`<location-timezone>,`
	`<location-asset-licensed>`
	If no location can be found, then only the asset client key parameter will be returned:
	`ASSET.LOCATION-<asset-client-key>`

RMS Asset Metadata Properties - Command API

RMS Asset Metadata Properties Query Commands

RMS Asset Metadata Properties Query Commands	
Query Command	**Description / Response Commands**

?ASSET.METADATA
<asset-client-key>
	This command is used to query for all asset metadata records for a specific asset from the RMS server.
	Arguments:
	<asset-client-key>: The asset client key of the asset you wish to query must be included. This asset client key is typically the D:P:S string in NetLinx systems.
	An ASSET.METADATA.RECORD.COUNT command string followed by a series of ASSET.METADATA.RECORD commands will be returned in response to this query.
	First a total count is returned.
	`ASSET.METADATA.RECORD.COUNT-<asset-client-key>,<metadata-record-count>`
	If the record count is greater than zero, then a series of the following command responses will be sent:
	`ASSET.METADATA.RECORD-`
	`<asset-client-key>,`
	`<record-index>,`
	`<record-count>,`
	`<metadata-key>,`
	`<metadata-name>,`
	`<metadata-value>,`
	`<metadata-data-type>,`
	`<metadata-read-only>`
	`(,<metadata-hyperlink-name>)`
	`(,<metadata-hyperlink-url>)`

?ASSET.METADATA-
<asset-client-key>,
<metadata-key>
	This command is used to query for a specific asset metadata property from the RMS server.
	Arguments:
	• **<asset-client-key>**: The asset client key of the asset you wish to query must be included. This asset client key is typically the D:P:S string in NetLinx systems.
	• **<metadata-key>**: This field is the asset metadata property key string used to uniquely identify the metadata property scoped to this asset.
	The asset metadata property record will be returned in the following response command format:
	`ASSET.METADATA-<asset-client-key>,`
	`<metadata-key>,`
	`<metadata-name>,`
	`<metadata-value>,`
	`<metadata-data-type>`
	`<metadata-read-only>,`
	`(,<metadata-hyperlink-name>)`
	`(,<metadata-hyperlink-url>)`

RMS Asset Metadata Properties Registration Commands

RMS Asset Metadata Properties Registration Commands	
Command	**Description**

ASSET.METADATA-
<asset-client-key>,
<metadata-key>,
<metadata-name>,
<metadata-value>
(,<metadata-data-type>)
(,<metadata-read-only>)
(,<metadata-hyperlink-name>)
(,<metadata-hyperlink-url>)
	This command is used to enqueue an asset metadata property for a specific asset in RMS.
	If the asset metadata property does not exists on the RMS server, a new record is created, if the property already exists, then it is updated with the details provided.
	Arguments:
	• **<asset-client-key>**: The asset client key of the asset you wish to create or update the metadata property. This asset client key is typically the D:P:S string in NetLinx systems.
	• **<metadata-key>**: This field is the asset metadata property key string used to uniquely identify the metadata property scoped to this asset.
	• **<metadata-name>**: This field is used to provide a friendly name for the asset metadata property in the RMS user interface.
	• **<metadata-value>**: This field is used to specify the string value for the asset metadata property.
	This field is not used when defining a metadata property of type HYPERLINK.
	For the metadata data type "DATE", use the format: YYYY-MM-DD
	For the metadata data type "DATETIME", use the format: YYYY-MM-DDThh:mm:ss
	• **<metadata-data-type>**: This field is *optional*. This field can determine which data type the metadata property should be configured as. If not included, by default, this field is set to 'STRING' which represents a data type of string. The following metadata data types are available.
	`"STRING"`
	`"MEMO"`
	`"BOOLEAN"`
	`"NUMBER"`
	`"DECIMAL"`
	`"DATE"`
	`"TIME"`
	`"HYPERLINK"`
	`"DATETIME"`
	• **<metadata-read-only>**: This property is *optional*. This field can be set to TRUE or FALSE depending on whether the metadata property should be allowed to be edited by users in the RMS user interface. By default, this property is set to TRUE.
	• **(<metadata-hyperlink-name>)**: This field is *optional* and should only be included if the metadata property data type is set to HYPERLINK. This field is used to provide a friendly URL link name for the metadata property.
	• **(<metadata-hyperlink-url>)**: This field is *optional* and should only be included if the metadata property data type is set to HYPERLINK. This field is used to provide the link URL for the metadata property.

ASSET.METADATA.SUBMIT- <asset-client-key>	This command will submit all pending queued asset metadata records for the specified asset that are currently staged for registration. Asset metadata records are en-queued and staged using the "ASSET.METADATA" command. Arguments: • **<asset-client-key>**: The asset client key of the asset you wish to create or update the metadata property. This asset client key is typically the D:P:S string in NetLinx systems.
ASSET.METADATA.CLEAR- <asset-client-key>	This command will remove all pending queued asset metadata records for the specified asset that are currently staged for registration. Arguments: • **<asset-client-key>**: The asset client key of the asset you wish to create or update the metadata property. This asset client key is typically the D:P:S string in NetLinx systems.
ASSET.METADATA.REMOVE- <asset-key>,<metadata-key>	This command will remove a single pending queued asset metadata record for the specified asset that is currently staged for registration. Arguments: • **<asset-client-key>**: The asset client key of the asset you wish to create or update the metadata property. This asset client key is typically the D:P:S string in NetLinx systems. • **<metadata-key>**: This field is the asset metadata property key string used to uniquely identify the metadata property scoped to this asset.

RMS Asset Metadata Properties Registration Commands (Cont.)	
Command	**Description**
ASSET.METADATA.UPDATE- <asset-client-key>, <metadata-key>, <metadata-value> (,<metadata-hyperlink-name>) (,<metadata-hyperlink-url>)	This command is used to perform an update on a given metadata property value on a single asset metadata property already defined/registered on the RMS server. If this command is sent for a metadata property that has not yet been registered on the RMS server, an error will occur and no further action will be taken. **Note**: All metadata data type with the exception of Hyperlink should provide the updated value (as a string) in the "<metadata-value>" argument. If you need to update a metadata property of type Hyperlink, then specify an empty string in the "<metadata-value>" argument and provide the hyperlink name and url in the optional command arguments: "<metadata-hyperlink-name>" and "<metadata-hyperlink-url>". Arguments: • **<asset-client-key>**: The asset client key of the asset you wish to create or update the metadata property. This asset client key is typically the D:P:S string in NetLinx systems. • **<metadata-key>**: This field is the asset metadata property key string used to uniquely identify the metadata property scoped to this asset. • **<metadata-value>**: This field is used to specify the string value for the asset metadata property. This field is not used when defining a metadata property of type HYPERLINK. For the metadata data type "DATE", use the format: YYYY-MM-DD For the metadata data type "DATETIME", use the format: YYYY-MM-DDThh:mm:ss • **(<metadata-hyperlink-name>)**: This field is *optional* and should only be included if the metadata property data type is set to HYPERLINK. This field is used to provide a friendly URL link name for the metadata property. • **(<metadata-hyperlink-url>)**: This field is *optional* and should only be included if the metadata property data type is set to HYPERLINK. This field is used to provide the link URL for the metadata property.

ASSET.METADATA.DELETE-
<asset-client-key>,
<metadata-key>
	This command will delete an asset metadata property from the RMS server.
	Arguments:
	• **<asset-client-key>**: The asset client key of the asset you wish to create or update the metadata property. This asset client key is typically the D:P:S string in NetLinx systems.
	• **<metadata-key>**: This field is the asset metadata property key string used to uniquely identify the metadata property scoped to this asset.
	The following response command will be returned based on the success of the delete operation:
	`ASSET.METADATA.DELETE-<asset-client-key>,`
	`<metadata-key>,`
	`<success>`
	• **<success>:** This argument will be a true|false value indicating the success of the delete operation.

ASSET.METADATA.EXCLUDE- <asset-client-key>, <metadata-key>(,<exclude>)	This command will add the asset metadata property to the asset metadata property registration exclusion list. If an asset metadata property registration attempt is made after the asset metadata property has been added to the exclusion list, the asset metadata property registration will be ignored. This command must be issued to RMS prior to the asset metadata property registration attempt. Arguments: • **<asset-client-key>**: The asset client key of the asset you wish to create or update the metadata property. This asset client key is typically the D:P:S string in NetLinx systems. • **<metadata-key>**: This field is the asset metadata property key string used to uniquely identify the metadata property scoped to this asset. • **(<exclude>)**: This parameter is optional and if not included the default value assumed is TRUE. If set to TRUE, then the metadata property will be added to the asset metadata property exclusion list. If set to FALSE the asset will be removed from the exclusion list.

RMS Asset Parameters - Command API

RMS Asset Parameters Event Notification Commands

RMS Asset Parameters Event Notification Commands	
String	**Description**
ASSET.PARAM.UPDATE- <asset-client-key>, <parameter-key>, <change-operator>, <change-value>	This notification command event is sent from the RMS client virtual device when an asset parameter receives an update value operation.
ASSET.PARAM.VALUE- <asset-client-key>, <parameter-key>, <parameter-name>, <parameter-value>	This notification command event is sent from the RMS client virtual device after an asset parameter has been updated on the server and a new current value has been determined.
ASSET.PARAM.RESET- <asset-client-key>, <parameter-key>, <parameter-name>, <parameter-value>	This notification command event is sent from the RMS client virtual device after an asset parameter has been reset. The new current value is provided in the event command arguments.

RMS Asset Parameters Query Commands

RMS Asset Parameters Query Commands	
Query Command	**Description / Response Commands**

?ASSET.PARAM-
<asset-client-key>
	This command is used to query for all asset parameter records for a specific asset from the RMS server.
	Arguments:
	<asset-client-key>: The asset client key of the asset you wish to query must be included. This asset client key is typically the D:P:S string in NetLinx systems.
	An ASSET.PARAM.RECORD.COUNT command string followed by a series of ASSET.PARAM.RECORD commands will be returned in response to this query.
	First a total count is returned.
	`ASSET.PARAM.RECORD.COUNT-<record-count>`
	If the record count is greater than zero, then a series of the following command responses will be sent:
	`ASSET.PARAM.RECORD-<asset-client-key>,`
	`<record-index>,`
	`<record-count>,`
	`<parameter-key>,`
	`<method-name>,`
	`<method-description>`
	If the parameter contains any defined thresholds, then a series of threshold commands will be additionally sent:
	`ASSET.PARAM.THRESHOLD-<asset-client-key>,`
	`<parameter-key>,`
	`<threshold-record-index>,`
	`<threshold-record-count>,`
	`<threshold-name>,`
	`<enabled>,`
	`<status-type-key>,`
	`<comparison-operator>,`
	`<threshold-value>,`
	`<notify-on-trip>,`
	`<notify-on-restore>,`
	`<delayed>,`
	`<delay-interval>`
	If the parameter does not contain any defined thresholds, then a single threshold command with a record index and total of '0' is sent:
	`ASSET.PARAM.THRESHOLD-<asset-client-key>,`
	`<parameter-key>,`
	`0,`
	`0`

RMS Asset Parameters Query Commands (Cont.)	
Command	**Description**

?ASSET.PARAM-
<asset-client-key,
<parameter-key>
	This command is used to query for a specific asset parameter from the RMS server.
	Arguments:
	<asset-client-key>: The asset client key of the asset you wish to query must be included. This asset client key is typically the D:P:S string in NetLinx systems.
	<parameter-key>: This field is the asset parameter key string used to uniquely identify the asset parameter scoped to this asset.
	`ASSET.PARAM-<asset-client-key>,`
	`<parameter-key>,`
	`<parameter-name>,`
	`<parameter-description>,`
	`<parameter-data-type>,`
	`<parameter-type>,`
	`<parameter-value>,`
	`<parameter-units>,`
	`<parameter-allow-reset>,`
	`<parameter-reset-value>,`
	`<parameter-min-value>,`
	`<parameter-max-value>,`
	`<parameter-enumeration>,`
	`<parameter-track-changes>,`
	`<parameter-is-stock>`
	For a detailed listing of the parameter arguments, see the ASSET.PARAM registration command (below).
	If the parameter contains any defined thresholds, then a series of threshold commands will be additionally sent:
	`ASSET.PARAM.THRESHOLD-<asset-client-key>,`
	`<parameter-key>,`
	`<threshold-record-index>,`
	`<threshold-record-count>,`
	`<threshold-name>,`
	`<enabled>,`
	`<status-type-key>,`
	`<comparison-operator>,`
	`<threshold-value>,`
	`<notify-on-trip>,`
	`<notify-on-restore>,`
	`<delayed>,`
	`<delay-interval>`

RMS Asset Parameters Registration & Update Commands

RMS Asset Parameters Registration & Update Commands	
Command	**Description**
ASSET.PARAM-<asset-client-key>, <parameter-key>, <parameter-name> (,<parameter-description>) (,<parameter-data-type>) (,<parameter-type>) (,<parameter-value>) (,<parameter-units>) (,<parameter-allow-reset>) (,<parameter-reset-value>) (,<parameter-min-value>) (,<parameter-max-value>) (,<parameter-enumeration>) (,<parameter-track-changes>) (,<parameter-bargraph-key>) (,<parameter-is-stock>)	This command is used to en-queue an asset parameter for a specific asset in RMS. If the asset parameter does not exist on the RMS server, a new record is created, if the parameter already exists, then no changes are applied. Arguments: • **<asset-client-key>**: The asset client key of the asset you wish to create the asset parameter. This asset client key is typically the D:P:S string in NetLinx systems. • **<parameter-key>**: This field is the asset parameter key string used to uniquely identify the parameter scoped to this asset. • **<parameter-name>**: This field is the asset parameter displayed name. • **<parameter-description>**: This property is *optional* and allows the program to provide an asset parameter textual description. • **<parameter-data-type>**: This property is *optional* and specifies an explicit asset parameter data type. If not provided, the data type is assumed to be STRING. The available parameter data type values include the following: NUMBER STRING ENUMERATION LEVEL BOOLEAN DECIMAL • **<parameter-type>**: This property is *optional* and specifies an explicit asset parameter type. If not provided, the parameter type is assumed to be NONE. The parameter type is used to designate the parameter to a specific parameter profile and behavior in RMS. For example, if the parameter should be included in RMS reporting and views for Lamp Hours, then a parameter type of LAMP_USAGE is applied.

RMS Asset Parameters Registration & Update Commands (Cont.)	
Command	**Description**

ASSET.PARAM- (Cont.)	The available parameter types include the following values:
	NONE
	ASSET_ONLINE
	ASSET_POWER
	POWER_CONSUMPTION
	EMISSIONS
	LAMP_USAGE
	BATTERY_LEVEL
	BATTERY_CHARGING_STATE
	RFID_LOCATION
	SOURCE_USAGE
	SOURCE_STATE
	SYSTEM_ONLINE
	SYSTEM_POWER
	TRANSPORT_STATE
	TRANSPORT_USAGE
	DISPLAY_USAGE
	TEMPERATURE
	SECURITY_STATE
	LIGHT_LEVEL
	SIGNAL_STRENGTH
	HVAC_STATE
	DIALER_STATE
	DOCKING_STATE
	• **<parameter-value>**: This property is *optional* and specifies an explicit asset parameter initial value to register with. If not provided, the parameter value remains null.
	• **<parameter-units>**: This property is *optional* and specifies a units decorator for the asset parameter. This is used as a suffix string when the RMS user interface displays the parameter value.
	• **<parameter-allow-reset>**: This property is *optional* and specifies if the asset parameter value can be RESET at runtime in the RMS user interface. If not specified, this property is assumed to be FALSE.
	• **<parameter-reset-value>**: This property is *optional* and specifies the asset parameter reset value applied to the parameter when a user RESETS the parameter at runtime in the RMS user interface. This argument should be defined if <parameter-allow-reset> is set to "true".
	• **<parameter-min-value>**: This property is *optional* and if included specifies a minimum acceptable parameter value. This property should only be used with numeric data types.
	• **<parameter-max-value>**: This property is *optional* and if included specifies a maximum acceptable parameter value. This property should only be used with numeric data types.
	• **<parameter-enumeration>**: This property is *optional* and specifies a pipe delimited series of enumeration values. This property should only be used with the enumeration data type.
	• **<parameter-track-changes>**: This property is *optional* and specifies if the RMS server should maintain a historical record of parameter value changes. If not defined, this property has a default value of 'false'.
	For a detailed listing if the parameter arguments, please see the ASSET.PARAM registration command.
	If the parameter contains any defined thresholds, then a series of threshold commands will be additionally sent:
	`ASSET.PARAM.THRESHOLD-<asset-client-key>,`
	`<parameter-key>,`
	`<threshold-record-index>,`
	`<threshold-record-count>,`
	`<threshold-name>,`
	`<enabled>,`
	`<status-type-key>,`
	`<comparison-operator>,`
	`<threshold-value>,`
	`<notify-on-trip>,`
	`<notify-on-restore>,`
	`<delayed>,`
	`<delay-interval>`

RMS Asset Parameters Registration & Update Commands (Cont.)	
Command	**Description**
ASSET.PARAM.THRESHOLD- <asset-client-key>, <parameter-key>, <threshold-name>, <enabled>,<status-type-key>, <comparison-operator>, <threshold-value>, <notify-on-trip>, <notify-on-restore>, <delayed>,<delay-interval>	This command will add a named asset parameter threshold for an asset parameter currently in the pending queued asset parameter records. Asset parameter records are en-queued and staged using the "ASSET.PARAM" command.
ASSET.PARAM.SUBMIT- <asset-client-key>	This command will submit all pending queued asset parameter records for the specified asset that are currently staged for registration. Asset parameter records are en-queued and staged using the "ASSET.PARAM" command.
ASSET.PARAM.CLEAR- <asset-client-key>	This command will remove all pending queued asset parameter records for the specified asset that are currently staged for registration.
ASSET.PARAM.REMOVE- <asset-client-key>, <parameter-key>	This command will remove a single pending queued asset parameter record for the specified asset that is currently staged for registration.

ASSET.PARAM.DELETE-
<asset-client-key>,
<parameter-key>
	This command will delete an asset parameter from the RMS server.
	The following response command will be returned based on the success of the delete operation:
	`ASSET.PARAMETER.DELETE-<asset-client-key>,`
	`<parameter-key>,`
	`<success>`
	<success>: This argument will be a true|false value indicating the success of the delete operation.

ASSET.PARAM.UPDATE- <asset-client-key>, <parameter-key>, <parameter-operation>, <parameter-value> (,<submit-now>)	This command will perform an update of the asset parameter value with the RMS server. The asset parameter should already be registered prior to issuing this command. Arguments: • **<parameter-operation>**: The parameter update operation argument specifies how the RMS server will apply the <parameter-value>. The following list of operations are available: SET_VALUE INCREMENT_VALUE DECREMENT_VALUE MULTIPLY_VALUE DIVIDE_VALUE RESET_VALUE • **(<submit-now>)**: This optional argument can be set to a value of true|false indicating if this single parameter value update should be sent to the RMS server immediately. If not provided, the argument is assumed to be a value of 'false'. If this argument is omitted or set to a value of 'false', then the parameter update will be en-queued waiting for a final 'ASSET.PARAM.UPDATE.SUBMIT' command to submit all of the pending updates at once. If this argument is set to a value of 'true', then this single parameter update will be immediately forwarded to the RMS server. A value of 'true' will not submit any pending parameter updates that are presently in queue.
ASSET.PARAM.UPDATE.SUBMIT- <asset-client-key>	This command will submit all pending queued asset parameter update records for the specified asset that are currently staged for updating. Asset parameter update records are enqueued and staged using the "ASSET.PARAM.UPDATE" command.
ASSET.PARAM.UPDATE.CLEAR- <asset-client-key>	This command will remove all pending queued asset parameter update records for the specified asset that are currently staged for updating.
ASSET.PARAM.UPDATE.REMOVE- <asset-client-key>,<parameter-key>	This command will remove a single pending queued asset parameter update record for the specified asset that is currently staged for update.
ASSET.PARAM.EXCLUDE- <asset-client-key>, <parameter-key>(,<exclude>)	This command will add the asset parameter to the asset parameter registration exclusion list. If an asset parameter registration attempt is made after the asset parameter has been added to the exclusion list, the asset parameter registration will be ignored. This command must be issued to RMS prior to the asset parameter registration attempt. Arguments: • **<exclude>**: This parameter is optional and if not included the default value assumed is TRUE. If set to TRUE, then the asset parameter will be added to the asset parameter method exclusion list. If set to FALSE the asset will be removed from the exclusion list.

RMS Asset Control Methods - Command API

RMS Asset Control Methods Event Notification Commands

RMS Asset Control Methods Event Notification Commands	
String	**Description**
ASSET.METHOD.EXECUTE- <asset-client-key>, <method-key>, **[,<method-argument-value>]***	This event notification command is issued by the RMS when an instruction is received from the RMS server to execute a control method.

RMS Asset Control Methods Query Commands

RMS Asset Control Methods Query Commands	
Command	**Description / Response Commands**

?ASSET.METHOD-
<asset-client-key>
	This command is used to query for all asset control method records for a specific asset from the RMS server.
	• <asset-client-key>: The asset client key of the asset you wish to query must be included. This asset client key is typically the D:P:S string in NetLinx systems.
	An ASSET.METHOD.RECORD.COUNT command string followed by a series of ASSET.METHOD.RECORD commands will be returned in response to this query.
	First a total count is returned.
	`ASSET.METHOD.RECORD.COUNT-<record-count>`
	If the record count is greater than zero, then a series of the following command responses will be sent:
	`ASSET.METHOD.RECORD-<asset-client-key>,`
	`<record-index>,`
	`<record-count>,`
	`<method-key>,`
	`<method-name>,`
	`<method-description>`

?ASSET.METHOD-
<asset-client-key,
<method-key>
	This command is used to query for a specific asset control method from the RMS server.
	• <asset-client-key>: The asset client key of the asset you wish to query must be included. This asset client key is typically the D:P:S string in NetLinx systems.
	• <method-key>: This field is the asset control method key string used to uniquely identify the control method scoped to this asset.
	`ASSET.METHOD-<asset-client-key>,`
	`<method-key>,`
	`<method-name>,`
	`<method-description>`

RMS Asset Control Method Registration Commands

RMS Asset Control Method Registration Commands	
Command	**Description**
ASSET.METHOD-<asset-client-key>, <method-key>,<method-name>, (,<method-description>)	This command is used to en-queue an asset control method for a specific asset in RMS. If the asset control method does not exist on the RMS server, a new record is created, if the control method already exists, then no changes are applied. • <asset-client-key>: The asset client key of the asset you wish to create the asset control method. This asset client key is typically the D:P:S string in NetLinx systems. • <method-key>: This field is the asset control method key string used to uniquely identify the control method scoped to this asset. • <method-name>: This field is the asset control method displayed name. • <method-description>: This property is OPTIONAL and allows the program to provide an asset control method textual description.

RMS Asset Control Method Registration Commands (Cont.)	
Command	**Description**
ASSET.METHOD.ARGUMENT- <asset-client-key>,<method-key>, <argument-ordinal>,<argument-name>, (,<argument-description>) (,<argument-data-type>) (,<argument-default-value>) (,<argument-minimum-value>) (,<argument-maximum-value>) (,<argument-step-value>) (,<argument-enumeration-value>)	This command will add an asset control method argument to a control method currently staged in the control method registration queue. • <argument-ordinal>: This property identifies the argument numerical position in the set of arguments. This ordinal value should start at 1. • <argument-data-type>: This property defines the data type for the control method argument. The acceptable values are: NUMBER STRING ENUMERATION LEVEL BOOLEAN DECIMAL • <argument-default-value>: This property is OPTIONAL and defines the default value for the argument when prompted in the RMS UI. This default value must adhere to the data type specified. • <argument-minimum-value>: This property is OPTIONAL and defines a minimum possible value to allow for this control method argument. This minimum value only applies to arguments of data type: NUMBER, DECIMAL, and LEVEL. • <argument-maximum-value>: This property is OPTIONAL and defines a maximum possible value to allow for this control method argument. This maximum value only applies to arguments of data type: NUMBER, DECIMAL, and LEVEL. • <argument-step-value>: This property is OPTIONAL and defines a step to increase or decrease the value by when using the UP and DOWN controls in the RMS UI. This step value only applies to arguments of data type: NUMBER, DECIMAL, and LEVEL. • <argument-enumeration-value>: This optional argument that defines a set of enumeration values. This is a pipe "|" delimited string where each enumeration value is separated by pipe "|" characters.
ASSET.METHOD.ARGUMENT.BOOLEAN- <asset-client-key>,<method-key>, <argument-ordinal>,<argument-name>, (,<argument-description>) (,<argument-default-value>)	This command will add an asset control method argument of type "BOOLEAN" to a control method currently staged in the control method registration queue. • <argument-default-value>): This is an optional argument and its default value is 'false'. Acceptable values are 'true" and 'false'.
ASSET.METHOD.ARGUMENT.STRING- <asset-client-key>,<method-key>, <argument-ordinal>,<argument-name>, (,<argument-description>) (,<argument-default-value>)	This command will add an asset control method argument of type "STRING" to a control method currently staged in the control method registration queue.
ASSET.METHOD.ARGUMENT.NUMBER- <asset-client-key>,<method-key>, <argument-ordinal>,<argument-name>, (,<argument-description>) (,<argument-default-value>) (,<argument-minimum-value>) (,<argument-maximum-value>) (,<argument-step-value>)	This command will add an asset control method argument of type "NUMBER" to a control method currently staged in the control method registration queue.
ASSET.METHOD.ARGUMENT.DECIMAL- <asset-client-key>,<method-key>, <argument-ordinal>,<argument-name>, (,<argument-description>) (,<argument-default-value>) (,<argument-minimum-value>) (,<argument-maximum-value>) (,<argument-step-value>)	This command will add an asset control method argument of type "DECIMAL" to a control method currently staged in the control method registration queue.
ASSET.METHOD.ARGUMENT.LEVEL- <asset-client-key>,<method-key>, <argument-ordinal>,<argument-name>, (,<argument-description>) (,<argument-default-value>) (,<argument-minimum-value>) (,<argument-maximum-value>) (,<argument-step-value>)	This command will add an asset control method argument of type "LEVEL" to a control method currently staged in the control method registration queue.

RMS Asset Control Method Registration Commands (Cont.)	
Command	**Description**
ASSET.METHOD.ARGUMENT.ENUM- <asset-client-key>,<method-key>, <argument-ordinal>,<argument-name>, (,<argument-description>) (,<argument-default-value>) (,<argument-enumeration-value>)	This command will add an asset control method argument of type "ENUM" to a control method currently staged in the control method registration queue. • <argument-enumeration-value>: This optional argument defines a set of enumeration values. This is a pipe "|" delimited string where each enumeration value is separated by pipe "|" characters.
ASSET.METHOD.UPDATE- <asset-client-key>,<method-key>, <method-name>, (,<method-description>)	This command can update an asset's control method name and description properties on the RMS server.
ASSET.METHOD.SUBMIT- <asset-client-key>	This command will submit all pending queued asset control method records for the specified asset that are currently staged for registration. Asset control method records are en-queued and staged using the "ASSET.METHOD" command.
ASSET.METHOD.CLEAR- <asset-client-key>	This command will remove all pending queued asset control method records for the specified asset that are currently staged for registration.
ASSET.METHOD.REMOVE- <asset-client-key>,<method-key>	This command will remove a single pending queued asset control method record for the specified asset that is currently staged for registration.

ASSET.METHOD.DELETE-
<asset-client-key>,<method-key>
	This command will delete an asset control method from the RMS server.
	The following response command will be returned based on the success of the delete operation:
	`ASSET.METHOD.DELETE-<asset-client-key>,`
	`<method-key>,`
	`<success>`
	• <success>: This argument will be a true|false value indicating the success of the delete operation.

ASSET.METHOD.EXECUTE- <asset-client-key>,<method-key> [,<argument-value>]*	This command is used to manually execute a RMS asset control method. A listing of required argument values should be included in this command instruction.
ASSET.METHOD.EXCLUDE- <asset-client-key>,<method-key> (,<exclude>)	This command will add the asset control method to the asset control method registration exclusion list. If an asset control method registration attempt is made after the asset control method has been added to the exclusion list, the asset control methods registration will be ignored. This command must be issued to RMS prior to the asset control method registration attempt. • <exclude>: This parameter is optional and if not included the default value assumed is TRUE. If set to TRUE, then the control methods will be added to the asset control method exclusion list. If set to FALSE the asset will be removed from the exclusion list.

RMS Hotlist - Command API

RMS Hotlist Event Notification Commands

RMS Hotlist Event Notification Commands	
String	**Description**
HOTLIST.COUNT- <is-client-default-location>, <location-id>, <record-count-number>	This event notification command will be sent when the RMS client detects a change in the number of hotlist records available for the default client location.

RMS Hotlist Query Commands

RMS Hotlist Query Commands	
Query Command	**Description / Response Command**
?HOTLIST.COUNT- (<location-id>)	This command will query the RMS client to determine the number of active hotlist records for this location already known by the client. The following command response will be returned from this query: `HOTLIST.COUNT-<location-id>,<record-count-number>` This command query does not perform a query against the RMS server, but rather returns the number of hotlist records last known by the RMS client. The RMS client and server communicate the hotlist count on a regular basis. The <location-id> argument is *optional*. If included the hotlist query will perform a lookup on the specified location. If omitted, the hotlist query will perform a lookup on the client gateway's default location.
?HOTLIST.RECORDS- (<location-id>)	This command will query the RMS server and retrieve hotlist items for this location. This command will perform a query directly to the RMS server and when the records have been delivered the following count response command will be returned by the RMS client. `HOTLIST.RECORD.COUNT-<location-id>,<number-of-total-records>` The queried hotlist records are cached in the RMS client and the following HOTLIST.RECORD query commands can be used to interrogate the cached data records. The <location-id> argument is *optional*. If included the hotlist query will perform a lookup on the specified location. If omitted, the hotlist query will perform a lookup on the client gateway's default location.

RMS Hotlist Query Commands (Cont.)	
Query Command	**Description / Response Command**

?HOTLIST.RECORD-
<start>(,<length>)
(,<location-id>)
	This command will query the RMS client hotlist record cache and return the requested hotlist record(s).
	• The <start> argument can be either a '*' character or a number representing which index position to start at. If the '*' character is assigned to the start parameter the query will return all hotlist records. The collection is '1' based and should start at index 1 (not '0').
	• The <length> argument is optional and can be a number representing how many records to return. If this argument is not specified, then a length of 1 is assumed.
	• The <location-id> argument is *optional*. If included the hotlist query will perform a lookup on the specified location. If omitted, the hotlist query will perform a lookup on the client gateway's default location.
	Examples:
	Return all records:
	`SEND_COMMAND vdvRMS, '?HOTLIST.RECORD-*'`
	Return only first record:
	`SEND_COMMAND vdvRMS, '?HOTLIST.RECORD-1'`
	Return first 10 records:
	`SEND_COMMAND vdvRMS, '?HOTLIST.RECORD-1,10'`
	Return records 5-10 records:
	`SEND_COMMAND vdvRMS, '?HOTLIST.RECORD-5,5'`
	The hotlist record will be returned in the following command response format:
	`HOTLIST.RECORD-<location-id>,`
	`<record-index>,`
	`<record-count>,`
	`<hotlist-record-id>,`
	`<hotlist-type-id>,`
	`<hotlist-type-name>,`
	`<status-type-id>,`
	`<status-type-name>,`
	`<priority-level>,`
	`<description>,`
	`<relative-date-string>,`
	`<relative-time-string>,`
	`<asset-id>,`
	`<asset-name>,`
	`<asset-parameter-id>,`
	`<asset-parameter-name>,`
	`<asset-parameter-threshold-id>,`
	`<asset-parameter-threshold-name>,`
	`<asset-parameter-threshold-comparison-operator>,`
	`<asset-parameter-threshold-value>`

?HOTLIST.RECORD.ID-
<start>(,<length>)
(,<location-id>)
	This command will query the RMS client hotlist record cache and return the requested hotlist record id(s).
	• The <start> argument can be either a '*' character or a number representing which index position to start at. If the '*' character is assigned to the start parameter the query will return all hotlist records. The collection is '1' based and should start at index 1 (not '0').
	• The <length> argument is optional and can be a number representing how many records to return. If this argument is not specified, then a length of 1 is assumed.
	• The <location-id> argument is *optional*. If included the hotlist query will perform a lookup on the specified location. If omitted, the hotlist query will perform a lookup on the client gateway's default location.
	The hotlist record(s) will be returned in the following command response format:
	`HOTLIST.RECORD.ID-<location-id>,`
	`<record-index>,`
	`<record-count>,`
	`<hotlist-record-id>`

RMS Hotlist Query Commands (Cont.)	
Query Command	**Description / Response Command**

?HOTLIST.RECORD.CREATED-
<start>(,<length>)
(,<location-id>)
	This command will query the RMS client hotlist record cache and return the requested hotlist record created date(s).
	• The <start> argument can be either a '*' character or a number representing which index position to start at. If the '*' character is assigned to the start parameter the query will return all hotlist records. The collection is '1' based and should start at index 1 (not '0').
	• The <length> argument is optional and can be a number representing how many records to return. If this argument is not specified, then a length of 1 is assumed.
	• The <location-id> argument is *optional*. If included the hotlist query will perform a lookup on the specified location. If omitted, the hotlist query will perform a lookup on the client gateway's default location.
	The hotlist record(s) will be returned in the following command response format:
	`HOTLIST.RECORD.CREATED-<location-id>,`
	`<record-index>,`
	`<record-count>,`
	`<created-date>`
	The date string will be provided in the international ISO-8601 standard format:
	YYYY-MM-DDThh:mm:ss
	http://www.iso.org/iso/date_and_time_format

?HOTLIST.RECORD.MODIFIED-
<start>(,<length>)
(,<location-id>)
	This command will query the RMS client hotlist record cache and return the requested hotlist record last modified date(s).
	The <start> argument can be either a '*' character or a number representing which index position to start at. If the '*' character is assigned to the start parameter the query will return all hotlist records. The collection is '1' based and should start at index 1 (not '0').
	The <length> argument is optional and can be a number representing how many records to return. If this argument is not specified, then a length of 1 is assumed.
	The <location-id> argument is *optional*. If included the hotlist query will perform a lookup on the specified location. If omitted, the hotlist query will perform a lookup on the client gateway's default location.
	The hotlist record(s) will be returned in the following command response format:
	`HOTLIST.RECORD.MODIFIED-<location-id>,`
	`<record-index>,`
	`<record-count>,`
	`<modified-date>`
	The date string will be provided in the international ISO-8601 standard format:
	YYYY-MM-DDThh:mm:ss
	http://www.iso.org/iso/date_and_time_format

?HOTLIST.RECORD.TYPE-
<start>(,<length>)
(,<location-id>)
	This command will query the RMS client hotlist record cache and return the requested hotlist record type (s).
	The <start> argument can be either a '*' character or a number representing which index position to start at. If the '*' character is assigned to the start parameter the query will return all hotlist records. The collection is '1' based and should start at index 1 (not '0').
	The <length> argument is optional and can be a number representing how many records to return. If this argument is not specified, then a length of 1 is assumed.
	The <location-id> argument is *optional*. If included the hotlist query will perform a lookup on the specified location. If omitted, the hotlist query will perform a lookup on the client gateway's default location.
	The hotlist record(s) will be returned in the following command response format:
	`HOTLIST.RECORD.TYPE-<location-id>,`
	`<record-index>,`
	`<record-count>,`
	`<hotlist-type-id>,`
	`<hotlist-type-name>`
	The hotlist types available are listed below:
	0 - CLIENT_GATEWAY
	1 - PARAMETER_TRIP
	2 - SERVICE_PROVIDER
	3 - ASSET_MAINTENANCE
	4 - LOCATION_MAINTENANCE
	5 - USER
	6 - NOTIFICATION_PROVIDER
	7 - LOG_PROVIDER
	8 - SCHEDULING
	9 - MESSAGE

RMS Hotlist Query Commands (Cont.)	
Query Command	**Description / Response Command**

?HOTLIST.RECORD.STATUS.TYPE-
<start>(,<length>)
(,<location-id>)
	This command will query the RMS client hotlist record cache and return the requested hotlist record status type (s).
	The <start> argument can be either a '*' character or a number representing which index position to start at. If the '*' character is assigned to the start parameter the query will return all hotlist records. The collection is '1' based and should start at index 1 (not '0').
	The <length> argument is optional and can be a number representing how many records to return. If this argument is not specified, then a length of 1 is assumed.
	The <location-id> argument is *optional*. If included the hotlist query will perform a lookup on the specified location. If omitted, the hotlist query will perform a lookup on the client gateway's default location.
	The hotlist record(s) will be returned in the following command response format:
	`HOTLIST.RECORD.STATUS.TYPE-<location-id>,`
	`<record-index>,`
	`<record-count>,`
	`<status-type-id>,`
	`<status-type-name>`
	There are system defined status types as well as user defined status types. The available status types can be viewed/configured in the RMS web settings pages.

?HOTLIST.RECORD.PRIORITY-
<start>(,<length>)
(,<location-id>)
	This command will query the RMS client hotlist record cache and return the requested hotlist record priority level(s).
	The <start> argument can be either a '*' character or a number representing which index position to start at. If the '*' character is assigned to the start parameter the query will return all hotlist records. The collection is '1' based and should start at index 1 (not '0').
	The <length> argument is optional and can be a number representing how many records to return. If this argument is not specified, then a length of 1 is assumed.
	The <location-id> argument is *optional*. If included the hotlist query will perform a lookup on the specified location. If omitted, the hotlist query will perform a lookup on the client gateway's default location.
	The hotlist record(s) will be returned in the following command response format:
	`HOTLIST.RECORD.PRIORITY-<location-id>,`
	`<record-index>,`
	`<record-count>,`
	`<priority-level>`

?HOTLIST.RECORD.DESCRIPTION-
<start>(,<length>)
(,<location-id>)
	This command will query the RMS client hotlist record cache and return the requested hotlist record description(s).
	The <start> argument can be either a '*' character or a number representing which index position to start at. If the '*' character is assigned to the start parameter the query will return all hotlist records. The collection is '1' based and should start at index 1 (not '0').
	The <length> argument is optional and can be a number representing how many records to return. If this argument is not specified, then a length of 1 is assumed.
	The <location-id> argument is *optional*. If included the hotlist query will perform a lookup on the specified location. If omitted, the hotlist query will perform a lookup on the client gateway's default location.
	The hotlist record(s) will be returned in the following command response format:
	`HOTLIST.RECORD.DESCRIPTION-<location-id>,`
	`<record-index>,`
	`<record-count>,`
	`<description>`

?HOTLIST.RECORD.TIMESTAMP-
<start>(,<length>)
(,<location-id>)
	This command will query the RMS client hotlist record cache and return the requested hotlist record relative date and time(s).
	The <start> argument can be either a '*' character or a number representing which index position to start at. If the '*' character is assigned to the start parameter the query will return all hotlist records. The collection is '1' based and should start at index 1 (not '0').
	The <length> argument is optional and can be a number representing how many records to return. If this argument is not specified, then a length of 1 is assumed.
	The <location-id> argument is *optional*. If included the hotlist query will perform a lookup on the specified location. If omitted, the hotlist query will perform a lookup on the client gateway's default location.
	The hotlist record(s) will be returned in the following command response format:
	`HOTLIST.RECORD.TIMESTAMP-<location-id>,`
	`<record-index>,`
	`<record-count>,`
	`<relative-date-string>,`
	`<relative-time-string>`
	The relative date and time strings are formatted for the locale of the default location that the client gateway is assigned to. These string are intended for display on the user interface, if you need a programmatic date/time string with a fixed format, please see the HOTLIST.RECORD.CREATED and HOTLIST.RECORD.MODIFIED commands.

RMS Hotlist Query Commands (Cont.)	
Query Command	**Description / Response Command**

?HOTLIST.RECORD.ASSET-
<start>(,<length>)
(,<location-id>)
	This command will query the RMS client hotlist record cache and return the requested hotlist record asset id and name(s).
	The <start> argument can be either a '*' character or a number representing which index position to start at. If the '*' character is assigned to the start parameter the query will return all hotlist records. The collection is '1' based and should start at index 1 (not '0').
	The <length> argument is optional and can be a number representing how many records to return. If this argument is not specified, then a length of 1 is assumed.
	The <location-id> argument is *optional*. If included the hotlist query will perform a lookup on the specified location. If omitted, the hotlist query will perform a lookup on the client gateway's default location.
	The hotlist record (s) will be returned in the following command response format:
	`HOTLIST.RECORD.ASSET-<location-id>,`
	`<record-index>,`
	`<record-count>,`
	`<asset-id>,`
	`<asset-name>`
	Please note that the asset id may be 0 and the asset name may be empty if no asset is associated with a given hotlist record.

?HOTLIST.RECORD.PARAMETER-
<start>(,<length>)
(,<location-id>)
	This command will query the RMS client hotlist record cache and return the requested hotlist record asset parameter id and name(s).
	• The <start> argument can be either a '*' character or a number representing which index position to start at. If the '*' character is assigned to the start parameter the query will return all hotlist records. The collection is '1' based and should start at index 1 (not '0').
	• The <length> argument is optional and can be a number representing how many records to return. If this argument is not specified, then a length of 1 is assumed.
	• The <location-id> argument is *optional*. If included the hotlist query will perform a lookup on the specified location. If omitted, the hotlist query will perform a lookup on the client gateway's default location.
	The hotlist record (s) will be returned in the following command response format:
	`HOTLIST.RECORD.PARAMETER-<location-id>,`
	`<record-index>,`
	`<record-count>,`
	`<asset-parameter-id>,`
	`<asset-parameter-name>`
	Please note that the asset parameter id may be 0 and the asset name may be empty if no asset parameter is associated with a given hotlist record.

?HOTLIST.RECORD.THRESHOLD-
<start>(,<length>)
(,<location-id>)
	This command will query the RMS client hotlist record cache and return the requested hotlist record asset parameter threshold information.
	• The <start> argument can be either a '*' character or a number representing which index position to start at. If the '*' character is assigned to the start parameter the query will return all hotlist records. The collection is '1' based and should start at index 1 (not '0').
	• The <length> argument is optional and can be a number representing how many records to return. If this argument is not specified, then a length of 1 is assumed.
	• The <location-id> argument is *optional*. If included the hotlist query will perform a lookup on the specified location. If omitted, the hotlist query will perform a lookup on the client gateway's default location.
	The hotlist record (s) will be returned in the following command response format:
	`HOTLIST.RECORD.THRESHOLD-<location-id>,`
	`<record-index>,`
	`<record-count>,`
	`<asset-parameter-threshold-id>,`
	`<asset-parameter-threshold-name>,`
	`<asset-parameter-threshold-comparison-operator>,`
	`<asset-parameter-threshold-value>`
	Please note that the asset parameter threshold id may be 0 and the other threshold field may be empty if no asset parameter threshold is associated with a given hotlist record.
	The asset parameter threshold comparison operators are listed below:
	0 - None
	1 - Less Than
	2 - Less Than or Equal To
	3 - Greater Than
	4 - Greater Than or Equal To
	5 - Equal To
	6 - Not Equal To
	7 - Contains
	8 - Does not contain

RMS Messaging - Command API

RMS Messaging Event Notification Commands

RMS Messaging Event Notification Commands	
String	**Description**
MESSAGE.DISPLAY- <message-type>, <message-title>, <message-body>, <message-timeout-seconds>, <message-modal-boolean>, <is-response-message>, (,<location-id>,) <is-default-location>	This event notification command will be sent when the RMS client should display a requested message on all available RMS users interfaces in the location. • Message Type (information|warning|security|critical|question) • Message Title String • Message Body Message • Message Timeout (seconds) • Message Modal (true|false) • Response Message (true|false) • Location ID The <is-response-message> parameter indicates if this message was sent as a response to a service provider request (true) or delivered as a generic message (false) from the RMS server. • If the user of the RMS web application clicks the Reply/Respond on the hotlist when selecting a Help or Maintenance Request hotlist item, the message will be tagged as a response message. • If the user of the RMS web application select the 'Send Message To Location' function, then the message will not be tagged as a response message. The <location-id> parameter is included to scope the message to a specific location. There are special edge cases where the message's location id can differ from that of the client gateway's assigned location. For example, if a touch panel asset has been relocated in the RMS web user interface, it should only show display messages for the new assigned location, even if that is not the location where the client gateway resides. The <is-default-location> argument helps identify if the location ID that the messages is destined for is the default location that the client gateway is assigned to. If the location ID is for an alternate location, this argument will return a value of false.

RMS Messaging Query Commands

RMS Messaging Query Commands	
Query Command	**Description / Response Command**
?MESSAGE.EMAIL.ENABLED	This command will query the RMS server to determine if the server's SMTP mail function has been enabled. The following command response will be returned: `MESSAGE.EMAIL.ENABLED-<true|false>`

RMS Messaging Instruction Commands

RMS Messaging Instruction Commands		
Command	**Description**	
MESSAGE.DISPLAY-	This command will send a RMS display message to all RMS touch panel user interfaces on this client system.	
<message-type>,	(required)	Message Type (information|warning|security|critical|question)
<message-title>,	(required)	Message Title String
<message-body>	(required)	Message Body Message
(,<message-timeout-seconds>)	(optional)	Message Timeout (seconds)
(,<message-modal-boolean>)	(optional)	Message Modal (true|false)
(,<is-response-message>)	(optional)	Response Message (true|false)
(,<location-id>)	(optional)	Location ID
	If a location id is provided, then the message will be scoped for delivery to a specific location. If this location id argument is not provided or left empty, then the message will be delivered to all display message capable devices on this client gateway. *Note: This message will only deliver local display messages on the client gateway, this command does not send an instruction to the RMS server.*	

RMS Messaging Instruction Commands		
Command	**Description**	
MESSAGE.EMAIL-	This command will construct and send an email message to the RMS server for delivery.	
<to>,	(required)	To Address List
<subject>,	(required)	Message Subject
<message-body>	(required)	Message Body Message
(,<cc>)	(optional)	CC Address List
(,<bcc>)	(optional)	BCC Address List
	In the address list fields, you can include multiple address recipients using a comma or semicolon delimiter. Note that if you use the comma delimiter, the comma character must be properly escaped for the Send Command to work. This email message will be sent from the RMS server. The from address and reply to address are those that are configured for the RMS server's SMTP settings.	

RMS RFID Management - Command API

For more information on the RFID reader and RFID tag properties, please see the AMX Anterus product documentation and programming guides.

RMS RFID System Event Notification Commands

RMS RFID System Event Notification Commands	
String	**Description**
RFID.INITIALIZE	If the RMS system is configured with the RFID management feature, this event notification command will be sent when the RMS client is ready to accept RFID reader and tag registrations. This notification command event may also be issued in response to the RFID.INITIALIZE Send Command request.

RMS RFID System Query Commands

RMS RFID System Query Commands	
Query Command	**Description / Response Command**
?RFID.ENABLED	This command will query the RMS server to determine if the server's RFID features have been enabled. The follow response command will be returned: `RFID.ENABLED-<true|false>`

RMS RFID System Management Commands

RMS RFID System Management Commands	
Command	**Description**
RFID.INITIALIZE	This command will send an instruction to the RMS client to re-initialize the RFID system, readers, and tag information with the RMS server.
RFID.READER.REGISTER- <rfid-reader-address> (,<rfid-reader-address>..)	This command should be sent to the RMS client to register all RFID Readers. This command should be sent in response to the RFID.INITIALIZE event. Each reader address should be included, this command supports 1 or more reader addresses.
RFID.READER.STATUS- <rfid-reader-address>, <rfid-reader-status>, <error-code>	This command should be sent to the RMS client on any RFID reader status change. This command only needs to be sent after the RFID system has been initialized.
RFID.TAG.REGISTER- <rfid-reader-address>, <rfid-tag-id>, <rfid-tag-name>, <rfid-tag-info>, <rfid-tag-timestamp>, <rfid-tag-signal-strength>, <rfid-tag-battery-level>	This command should be sent to the RMS client immediately after registering the RFID readers using the RFID.READER.REGISTER command. This is typically in response to the RFID.INITIALIZE event and allows the RMS client to synchronize all RFID tag information with the RMS server. This command buffers all the tag registrations in preparation for submission to the RMS server.
RFID.READER.SUBMIT.REGISTERED.TAGS- <rfid-reader-address>	This command should be sent to the RMS client after all the RFID tag registration commands are complete. This command submits all the buffered tag registration information to the RMS server.

RMS RFID System Management Commands (Cont.)	
Command	**Description**
RFID.TAG.ACQUIRED- <rfid-reader-address>, <rfid-tag-id>, <rfid-tag-name>, <rfid-tag-info>, <rfid-tag-timestamp>, <rfid-tag-signal-strength>, <rfid-tag-battery-level>	After all the RFID tags have been registered, any subsequent RFID tag changes must be communicated to the RMS server. If a RFID tag is newly acquired, this command should be sent to the RMS client to update the RFID tag with the RMS server. RFID tag changes are temporarily buffered inside the RMS client and are forwarded to the RMS server in batch to optimize the data communications with the server.
RFID.TAG.LOST- <rfid-reader-address>, <rfid-tag-id>, <rfid-tag-name>, <rfid-tag-info>, <rfid-tag-timestamp>	After all the RFID tags have been registered, any subsequent RFID tag changes must be communicated to the RMS server. If a RFID tag is lost by a reader, this command should be sent to the RMS client to update the RFID tag with the RMS server. RFID tag changes are temporarily buffered inside the RMS client and are forwarded to the RMS server in batch to optimize the data communications with the server.
RFID.TAG.UPDATE- <rfid-reader-address>, <rfid-tag-id>, <rfid-tag-name>, <rfid-tag-info>, <rfid-tag-timestamp>, <rfid-tag-signal-strength>, <rfid-tag-battery-level>	After all the RFID tags have been registered, any subsequent RFID tag changes must be communicated to the RMS server. If a RFID tag has been updated, this command should be sent to the RMS client to update the RFID tag with the RMS server. RFID tag changes are temporarily buffered inside the RMS client and are forwarded to the RMS server in batch to optimize the data communications with the server.

RMS Service Provider - Command API

RMS Service Provider Commands

RMS Service Provider Commands	
Command	**Description**
SERVICE.HELP.REQUEST- <message-body> (,<location-id>)	This command can be sent to the RMS client to issue a help request to the RMS server. This help request will show in the RMS hotlist and the appropriate service provider notifications will be issued. The <location-id> parameter is optional. • If provided, it scopes the service provider request to a specific location. • If not provided, the location is scoped to that of the default location that the client gateway is assigned to.
SERVICE.MAINTENANCE.REQUEST- <message-body> (,<location-id>)	This command can be sent to the RMS client to issue a maintenance request to the RMS server. This maintenance request will show in the RMS hotlist and the appropriate service provider notifications will be issued. The <location-id> parameter is optional. • If provided, it scopes the service provider request to a specific location. • If not provided, the location is scoped to that of the default location that the client gateway is assigned to.

RMS NetLinx Scheduling Client API

Overview

NetLinx programmers can make use of RMS scheduling information in the following ways:

1. Query RMS for a variety of information about scheduling events. Programmers are provided with both send commands and functions for their queries.
2. Execute commands to perform such tasks as creating or ending events. As with queries, both send commands or function calls are provided.
3. Monitor RMS scheduling change events such as changes to monthly summary.

The response to a query or request for the execution of a task will be in the form of either a data event or the execution of a callback function. In some cases a single command may result in more than one event type or more than one call to the same event as in the case of reporting all the booking events for the day.

Date and Time Format Information

Unless otherwise stated, dates consist of two components, a date in the form of 'mm/dd/yyyy' and a time in the form of 'hh:mm:ss'.

The time is in 24 hour format (hours are in the range 0 through 23).

Use the *RmsSetGuiDatePattern* and *RmsSetGuiDatePattern* functions to format the Client time and date format. See *Setting The Client Time and Date Format* on page 167 for details.

Location Information

Locations or location ID's in the scheduling API's are numeric values greater than zero. When using send commands, many allow the input of a location ID in numeric format as the last argument. In most cases the location is optional and if not specified, the default location for the client will be assumed.

In the notes for the syntax for SEND_COMMANDs, an optional location argument is indicated by parentheses.

Many commands return as part of their response an indication as to whether or not the information relates to the default location. The response will always be the CHAR constant for 'TRUE' or 'FALSE'.

Callbacks

The RMS SDK provides a wide range of functions generally known as callbacks which are executed as part of an event communication process. The scheduling API's follow this same approach and thus requires the programmer to determine the callbacks of interest by including #DEFINE statements in their code. In addition, a programmer must create an implementation of the callback function.

- For specifics on the callbacks and their related #DEFINE statements, check the "*RmsSchedulingApi.axi*" and "*RmsSchedulingEventListener.axi*" NetLinx include files.
- For each of the callbacks documented below, the #DEFINE which will enable it is listed with it.

#DEFINE Compiler Directive

This directive defines a symbol to be used only by #IF_DEFINED and #IF_NOT_DEFINED compiler directives.

```
#DEFINE <symbol>
```

The name of the symbol must be unique among all other identifiers in the program. The symbol can be defined anywhere in the program file but cannot be used in any statement that appears before it is defined.

Example:

```
// Specify the INCLUDE_TOGGLE_VIDEO_PROJECTOR_POWER_FUNCTION compiler
// directive if there is a video projected connected to the controller
#DEFINE INCLUDE_TOGGLE_VIDEO_PROJECTOR_POWER_FUNCTION

#IF_DEFINED INCLUDE_TOGGLE_VIDEO_PROJECTOR_POWER_FUNCTION
DEFINE_FUNCTION toggleVideoProjectorPower()
{
  // code to toggle video projector power goes here
}
#END_IF
```

Notice in the above sample that the #DEFINE compiler directive is specified before the #IF_DEFINED and #END_IF compiler directives. When #IF_DEFINED and #IF_NOT_DEFINED compiler directives are specified in include files, the include file statements (defined by the #INCLUDE compiler directive) need to be declared after the #DEFINE compiler directive statements.

For more information on Compiler Directives, refer to the *NetLinx Programming Language Reference Guide*.

Scheduling Structures

In many cases additional information about a scheduling booking event is provided in the form of a NetLinx data structure. The three main structure types are "**RmsEventBookingResponse**", "**RmsEventBookingMonthlySummary**" and "**RmsEventBookingDailyCount**".

In the case of "*RmsEventBookingResponse*", many of the fields provide general booking event information common to any response. Included in the same structure are fields which are only meaningful in specific situations.

- Fields such as "*isSuccessful*" or "*failureDescription*" only relate to callbacks which are the response to a command. These specific fields are not meaningful in callbacks reporting changes such as the start of an event.
- Fields like "*elapseMinutes*" only related to active booking events, while "*minutesUntilStart*" only relate to the next active booking event.
- In the specific case where an "*RmsEventBookingResponse*" structure is the response to a failed command, most of the fields except "*failureDescription*" may not be meaningful. When possible, the relevant fields will be filled in with data to help better understand the circumstances that lead to the error. As an example, some fields will have "*N/A*" to help in this determination.

Additional information can be found within the "*RmsSchedulingApi.axi*" Include File.

Function Status and Logging

As a general rule, most commands return the CHAR constant for TRUE if successful or the CHAR constant for FALSE if an error is detected. A common error example might be incorrect or missing arguments. Errors are logged using the normal NetLinx and RMS SDK logging API's.

Query Functions

There are several scheduling *query* functions defined in the *RmsSchedulingApi.axi* Include File:

- **RmsBookingsRequest** (see page 159)
- **RmsBookingRequest** (see page 160)
- **RmsBookingsSummariesDailyRequest** (see page 160)
- **RmsBookingsSummaryDailyRequest** (see page 161)
- **RmsBookingActiveRequest** (see page 161)
- **RmsBookingNextActiveRequest** (see page 162)

Create/Extend/End Functions

The following *create*, *extend* and *end* functions defined in the *RmsSchedulingApi.axi* Include File that alter booking events:

- **RmsBookingCreate** (see page 162)
- **RmsBookingExtend** (see page 163)
- **RmsBookingEnd** (see page 163)

NOTE: *The RmsBookingCreate function is only valid for the current day; the RmsBookingExtend and RmsBookingEnd functions are only valid for the active meeting.*

Server Initiated Events

There are several Server Initiated Events defined in the *RmsSchedulingApi.axi* Include File that report when there are changes in booking events:

- **RmsEventSchedulingActiveUpdated** (see page 164)
- **RmsEventSchedulingNextActiveUpdated** (see page 164)
- **RmsEventSchedulingEventEnded** (see page 164)
- **RmsEventSchedulingEventStarted** (see page 164)
- **RmsEventSchedulingEventUpdated** (see page 164)
- **RmsEventSchedulingDailyCount** (see page 165)
- **RmsEventSchedulingMonthlySummaryUpdated** (see page 165)

The tables in this section provide detailed descriptions of each scheduling query function. Each row in the tables describes various aspects of each scheduling query function. Each row is defined below:

- **Description**: This section provides a description of the basic functionality provided by each query function.
- **Compiler Directive**: This section provides the compiler directive required in order to implement each query function. If a function compiler directive is included, then the user code must implement the required callback method so the RMS Event Listener can invoke the callback.

  The following code snippet illustrates the required method that must be included in the user code.

  ```
  // include callback method for RmsBookingsRequest to call
  #DEFINE INCLUDE_SCHEDULING_BOOKINGS_RECORD_RESPONSE_CALLBACK
  ```

- **Function Syntax**: This section provides the required syntax for each query function, as well as definitions for all variables.
- **Returns**: This section provides the expected return data for each query function.

- **Callback Function Syntax**: This section provides the syntax of the callback function associated with each query function, including definitions of all included parameters.
- **SEND_COMMAND Syntax**: This section provides the required syntax for the NetLinx SEND_COMMAND associated with each query function, including any required variables. NetLinx SEND_COMMANDS represent an alternative method of performing the same thing as the Function (described in the *Function Syntax* row).

RmsBookingsRequest

RmsBookingsRequest	
Description:	Queries the event booking records for a location and specific date.
Compiler Directive:	`#DEFINE INCLUDE_SCHEDULING_BOOKINGS_RECORD_RESPONSE_CALLBACK`

Function Syntax:	`RmsBookingsRequest(CHAR startDate[], LONG locationId)`
	where:
	• **startDate** - A date in the form of 'mm/dd/yyyy'. Note, no time is needed.
	• **locationid** - LONG location ID
	Example:
	`RmsBookingsRequest('9/10/2012',3)`

Returns:	FALSE is returned if startDate is empty. If the request is successful, the response will consist of two components: 1) A summary data event of the form: `SCHEDULING.BOOKINGS.COUNT-<location-id>, <is-client-default-location>, <booking-record-count>` • **location-id** - LONG which is the location ID • **is-client-default-location** - CHAR constant TRUE or FALSE • **booking-record-count** - INTEGER 2) A callback function will be executed for each record. Each record will include the total number of records in the response as well as the record number or index of the specific response in that response group. In the case where there are no records, **recordCount** and **recordIndex** will both be 0 (zero) and most of the other structure information will be meaningless.
Callback Function Syntax:	`RmsEventSchedulingBookingsRecordResponse CHAR isDefaultLocation, INTEGER recordIndex, INTEGER recordCount, CHAR bookingId[], RmsEventBookingResponse eventBookingResponse)` where: • **isDefaultLocation** - boolean, TRUE if th location in the response is the default location • **recordIndex** - INTEGER value which indicates the record number for this response • **recordCount** - INTEGER which reports the total number of records for this response. • **bookingId** - CHAR array which is the booking ID of the booking event this record is for. • **eventBookingResponse** - This structure provides additional information about each booking event.

SEND_COMMAND Syntax:	`?SCHEDULING.BOOKINGS-<start-date>(,<location-id>)`
	• **start-date** - NetLinx LDATE format string
	• **location-id** - (optional) LONG location ID
	Example - To request the bookings for location number 3 on September 9th:
	`SEND_COMMAND vdvRMS,'?SCHEDULING.BOOKINGS-9/10/2012,3'`

RmsBookingRequest

RmsBookingRequest	
Description:	Request information about a specific booking by booking ID. If location ID is less than 1, the default location will be used.
Compiler Directive:	`#DEFINE INCLUDE_SCHEDULING_BOOKING_RESPONSE_CALLBACK`
Function Syntax:	`RmsBookingRequest(CHAR bookingId[], LONG locationId)` • **bookingId** - unique ID of the requested booking event • **locationid** - LONG location ID
Returns:	FALSE is returned if booking ID is empty. Example: `RmsBookingRequest('31--1999997729',4)`
Callback Function Syntax:	If the request is successful, the following callback function will be executed: `RmsEventSchedulingBookingResponse (CHAR isDefaultLocation, CHAR bookingId[], RmsEventBookingResponse eventBookingResponse)` • **isDefaultLocation** - boolean, TRUE if th location in the response is the default location • **bookingId** - CHAR array which is the booking ID of the booking event that was requested • **eventBookingResponse** - This structure provides additional information about the booking event

SEND_COMMAND Syntax:	`?SCHEDULING.BOOKING-<booking-id>(,<location-id>)`
	• **booking-id** - unique ID of the requested booking event
	• **location-id** - (optional) LONG location ID
	Example - To request information about booking event ID '31-1999997729' at location ID '4':
	`SEND_COMMAND vdvRMS,'?SCHEDULING.BOOKING-31--1999997729,4'`

RmsBookingsSummariesDailyRequest

RmsBookingsSummariesDailyRequest	
Description:	Query Monthly Booking Summary For Specified Month of the Year and Location. If location ID is less than 1, the default location will be used.
Compiler Directive:	`#DEFINE INCLUDE_SCHEDULING_BOOKING_SUMMARIES_DAILY_RESPONSE_CALLBACK`

Function Syntax:	`RmsBookingsSummariesDailyRequest(SINTEGER dayOfTheMonth, LONG locationId)`
	• **dayOfTheMonth** - SINTEGER which is the day of the current month for the requested summary.
	• **locationId** - LONG which is the location ID of the meeting
	Example:
	`RmsBookingsSummariesDailyRequest(10,6)`

Returns:	FALSE is returned if month is less than 1 or greater than 12. If the request is successful, the response will consist of two components: 1) A summary data event of the form: `SCHEDULING.BOOKINGS.SUMMARIES.DAILY.COUNT-<location-id>, <is-client-default-location>, <booking-summary-count>` • **location-id** - LONG which is the location ID of the meeting • **is-client-default-location** - boolean, TRUE if the location in the response is the default location • **booking-summary-count** - INTEGER reflecting the total number of summary records 2) If the record count is greater than zero, a callback function will be executed for each record. Each record will include the total number of records in the response as well as the record number or index of the specific response in that response group (see "Callback Function Syntax" below).
Callback Function Syntax:	`RmsEventSchedulingSummariesDailyResponse(CHAR isDefaultLocation, RmsEventBookingDailyCount dailyCount` • **isDefaultLocation** - boolean, TRUE if th location in the response is the default location • **dailyCount** - A structure with information about a specific date

SEND_COMMAND Syntax:	`?SCHEDULING.BOOKINGS.SUMMARIES.DAILY-<month-date>(,<location-id>)`
	where:
	• **month** - INTEGER which is the month of the current year for the requested summary.
	• **location-id** - (optional) LONG location ID
	Example - To request information for the 10th month of the year (October) and location ID '6':
	`SEND_COMMAND vdvRMS,'?SCHEDULING.BOOKINGS.SUMMARIES.DAILY-10,6`

RmsBookingsSummaryDailyRequest

RmsBookingsSummaryDailyRequest	
Description:	Query a single daily event summary record by date and location. If location ID is less than 1, the default location will be used.
Compiler Directive:	`#DEFINE INCLUDE_SCHEDULING_BOOKING_SUMMARY_DAILY_RESPONSE_CALLBACK`

Function Syntax:	`RmsBookingsSummaryDailyRequest(CHAR summaryDate[], LONG locationId)`
	• **summaryDate** - date in the form of 'mm/dd/yyyy'. Note that no time is needed
	• **locationid** - LONG which is the location ID of the meeting
	Example:
	`RmsBookingsSummaryDailyRequest('10/01/2012',3)`

Returns:	FALSE is returned if summaryDate is empty. If the request is successful, the following callback function will be executed (see *Callback Function Syntax* below).
Callback Function Syntax:	`RmsEventSchedulingSummaryDailyResponse(CHAR isDefaultLocation, RmsEventBookingDailyCount dailyCount)` • **isDefaultLocation** - boolean, TRUE if th location in the response is the default location • **dailyCount** - A structure with information about a specific date

SEND_COMMAND Syntax:	`?SCHEDULING.BOOKINGS.SUMMARY.DAILY-<summary-date>(,<location-id>)`
	• **summary-date** - date in the form of 'mm/dd/yyyy'. Note that no time is needed
	• **location-id** - (optional) LONG location ID
	Example - To request information for the 10/1/2012 , location ID '3':
	`SEND_COMMAND vdvRMS,'?SCHEDULING.BOOKINGS.SUMMARY.DAILY-10/01/2012,3'`

RmsBookingActiveRequest

RmsBookingActiveRequest	
Description:	Query the current active booking for a given location. If location ID is less than 1, the default location will be used.
Compiler Directive:	`#DEFINE INCLUDE_SCHEDULING_ACTIVE_RESPONSE_CALLBACK`

Function Syntax:	`RmsBookingActiveRequest(LONG locationId)`
	• **locationid** - LONG which is the location id of the meeting
	Example:
	`RmsBookingActiveRequest(3)`

Returns:	If the request is successful, the response will consist of two components: 1) A summary data event of the form: `SCHEDULING.BOOKING.ACTIVE.COUNT-<is-client-default-location>, <booking-record-count>` • **is-client-default-location** - CHAR constant TRUE or FALSE • **booking-record-count** - INTEGER indicating the next active booking record count 2) If the record count is greater than zero, a callback function will be executed for each record. Each record will include the total number of records in the response as well as the record number or index of the specific response in that response group (see *Callback Function Syntax* below).

Callback Function Syntax:	`RmsEventSchedulingActiveResponse(CHAR isDefaultLocation, INTEGER recordIndex, INTEGER recordCount,`
	`CHAR bookingId[], RmsEventBookingResponse eventBookingResponse)`
	• **isDefaultLocation** - boolean, TRUE if th location in the response is the default location
	• **recordIndex** - INTEGER value which indicates the record number for this response
	• **recordCount** - INTEGER which reports the total number of records for this response.
	• **bookingId** - CHAR array which is the booking ID of the booking event this record is for.
	• **eventBookingResponse** - This structure provides additional information about each booking event.

SEND_COMMAND Syntax:	`?SCHEDULING.BOOKING.ACTIVE-(<location-id>)`
	• **location-id** - (optional) LONG location ID
	Example - To request information regarding the current active booking for location ID '3':
	`SEND_COMMAND vdvRMS,'?SCHEDULING.BOOKING.ACTIVE-3'`

RmsBookingNextActiveRequest

RmsBookingNextActiveRequest	
Description:	Query the next active booking for a given location. If location ID is less than 1, the default location will be used.
Compiler Directive:	`#DEFINE INCLUDE_SCHEDULING_NEXT_ACTIVE_RESPONSE_CALLBACK`

Function Syntax:	`RmsBookingNextActiveRequest(LONG locationId)`
	• **locationid** - LONG which is the location id of the meeting
	Example:
	`RmsBookingNextActiveRequest(3)`

Returns:	If the request is successful, the response will consist of two components: 1) A summary data event of the form: `SCHEDULING.BOOKING.NEXT.ACTIVE.COUNT-<is-client-default-location>, <booking-record-count>` • **is-client-default-location** - CHAR constant TRUE or FALSE • **booking-record-count** - INTEGER indicating the next active booking record count 2) If the record count is greater than zero, a callback function will be executed for each record. Each record will include the total number of records in the response as well as the record number or index of the specific response in that response group (see "Callback Function Syntax" below).

Callback Function Syntax:	`RmsEventSchedulingNextActiveResponse(CHAR isDefaultLocation, INTEGER recordIndex, INTEGER recordCount,`
	`CHAR bookingId[], RmsEventBookingResponse eventBookingResponse)`
	• **isDefaultLocation** - boolean, TRUE if the location in the response is the default location
	• **recordIndex** - INTEGER value which indicates the record number for this response
	• **recordCount** - INTEGER which reports the total number of records for this response.
	• **bookingId** - CHAR array which is the booking ID of the booking event this record is for.
	• **eventBookingResponse** - This structure provides additional information about each booking event

SEND_COMMAND Syntax:	`?SCHEDULING.BOOKING.NEXT.ACTIVE-(<location-id>)`
	• **location-id** - (optional) LONG location ID
	Example - To request information regarding the next active booking for location ID '3':
	`SEND_COMMAND vdvRMS,'?SCHEDULING.BOOKING.NEXT.ACTIVE-3'`

RmsBookingCreate

RmsBookingCreate	
Description:	Create an ad hoc booking event. If location ID is less than 1, the default location will be used. **Note**: "RmsBookingCreate" is only valid for the current day.
Compiler Directive:	`#DEFINE INCLUDE_SCHEDULING_CREATE_RESPONSE_CALLBACK`

Function Syntax:	`RmsBookingCreate(CHAR startDate[], CHAR startTime[], INTEGER durationMinutes, CHAR subject[], CHAR messageBody[], LONG locationId)`
	• **startDate** - NetLinx LDATE format CHAR array
	• **startTime** - NetLinx TIME format CHAR array
	• **durationMinutes** - INTEGER for the duration of the meeting in minutes
	• **subject** - Subject of the meeting
	• **messageBody** - Additional information about the meeting
	• **locationid** - LONG which is the location id of the meeting
	Example:
	`RmsBookingCreate('08/24/2012','07:00:00',15,'Status Meeting','Discuss current project status',3)`

Returns:	If the request is successful, the response will consist of a single callback:
Callback Function Syntax:	`RmsEventSchedulingCreateResponse (CHAR isDefaultLocation, CHAR responseText[], RmsEventBookingResponse eventBookingResponse)` • **isDefaultLocation** - boolean, TRUE if the location in the response is the default location • **responseText** - If successful this will contain the event booking ID. On an error, this will contain information about the error . • **eventBookingResponse** - This structure provides additional information about the booking event

SEND_COMMAND Syntax:	`SCHEDULING.BOOKING.CREATE-<start-date>,<start-time>, <duration-minutes>,<subject>, <message-body>,(<location-id>)`
	• **start-date** - NetLinx LDATE format CHAR array
	• **start-time** - NetLinx TIME format CHAR array
	• **duration-minutes** - INTEGER for the duration of the meeting in minutes
	• **subject** - Subject of the meeting
	• **message-body** - Additional information about the meeting
	• **location-id** - (optional) LONG location ID
	Example - Create a 15 minute meeting on 8/24/2012 at 7 AM for location ID '3':
	`SEND_COMMAND 41001:1:0,'SCHEDULING.BOOKING.CREATE-08/24/2012,07:00:00,15,Status Meeting, Discuss current project status,3'`

RmsBookingExtend

RmsBookingExtend	
Description:	Extend a booking event. If location ID is less than 1, the default location will be used. **Note**: "RmsBookingExtend "is only valid for the active meeting.
Compiler Directive:	`#DEFINE INCLUDE_SCHEDULING_EXTEND_RESPONSE_CALLBACK]`
Function Syntax:	`RmsBookingExtend(CHAR bookingId[], LONG extendDurationMinutes, LONG locationId)` • **bookingId** - CHAR array which is the booking ID of the booking event this record is for • **extendDurationMinutes** - LONG which is the number of minutes to extend the booking event • **locationid** - LONG which is the location id of the meeting
Returns:	If the request is successful, the response will consist of a single callback: Example: `RmsBookingExtend('29--1999997952',30)`

Callback Function Syntax:	`SCHEDULING.BOOKING.EXTEND-<booking-id>,<extend-duration-minutes>,(<location-id>)`
	• **bookingId** - CHAR array which is the booking ID of the booking event this record is for
	• **extendDurationMinutes** - LONG which is the number of minutes to extend the booking event
	• **location-id** - (optional) LONG location ID
	Example - Extend booking ID '29-1999997952' for 30 minutes:
	`SEND_COMMAND 41001:1:0,'SCHEDULING.BOOKING.EXTEND-29--1999997952,30'`

SEND_COMMAND Syntax:	`RmsEventSchedulingExtendResponse (CHAR isDefaultLocation, CHAR responseText[], RmsEventBookingResponse eventBookingResponse)` • **isDefaultLocation** - boolean, TRUE if the location in the response is the default location • **responseText** - If the request is successful this will contain the event booking ID. On an error, this will contain information about the error • **eventBookingResponse** - This structure provides additional information about the booking event

RmsBookingEnd

RmsBookingEnd	
Description:	End a booking event. If location ID is less than 1, the default location will be used. **Note**: "RmsBookingEnd" is only valid for the active meeting.
Compiler Directive:	`#DEFINE INCLUDE_SCHEDULING_END_RESPONSE_CALLBACK`

Function Syntax:	`RmsBookingEnd(CHAR bookingId[], LONG locationId)`
	• **bookingId** - CHAR array which is the booking ID of the booking event this record is for
	• **locationid** - LONG which is the location id of the meeting
	Example:
	`RmsBookingEnd('29-1999997939', 30)`

Returns:	If the request is successful, the response will consist of a single callback:
Callback Function Syntax:	`RmsEventSchedulingEndResponse (CHAR isDefaultLocation, CHAR responseText[], RmsEventBookingResponse eventBookingResponse)` • **isDefaultLocation** - boolean, TRUE if the location in the response is the default location • **responseText** - If the request is successful this will contain the event booking ID. On an error, this will contain information about the error • **eventBookingResponse** - This structure provides additional information about the booking event

SEND_COMMAND Syntax:	`SCHEDULING.BOOKING.END-<booking-id>,(<location-id>)`
	• **booking-id** - CHAR array which is the booking ID of the booking event this record is for
	• **location-id** - (optional) LONG location ID
	Example - End booking ID '29-1999997939' :
	`SEND_COMMAND 41001:1:0,'SCHEDULING.BOOKING.END-29-1999997939'`

Server Initiated Events

RMS provides information regarding the status of booking events. As with other callbacks, any programmer interested in receiving the events must provide a '#DEFINE' for the specific include directive, as well as a function which implements any necessary code to process the event.

As an example, to receive events when an event starts, a programmer must provide the following line of code:

```
#DEFINE INCLUDE_SCHEDULING_EVENT_STARTED_CALLBACK
```

With the '#DEFINE' compiler directive in place, there must be a function definition which implements any logic to process the event:

```
DEFINE_FUNCTION RmsEventSchedulingEventStarted(CHAR bookingId[], RmsEventBookingResponse
eventBookingResponse){// Do something to process the event}
```

RmsEventSchedulingActiveUpdated

RmsEventSchedulingActiveUpdated	
Description:	This callback is executed when RMS wants to indicate the current active booking event has updated/changed. **Note**: Active may be more than 1.
Compiler Directive:	`#DEFINE INCLUDE_SCHEDULING_ACTIVE_UPDATED_CALLBACK`
Syntax:	`RmsEventSchedulingActiveUpdated(CHAR bookingId[], RmsEventBookingResponse eventBookingResponse)` • **booking-id** - CHAR array which is the booking ID of the booking event this record is for • **eventBookingResponse** - This structure provides additional information about the booking event

RmsEventSchedulingNextActiveUpdated

RmsEventSchedulingNextActiveUpdated	
Description:	This callback is executed when RMS wants to indicate the next active booking event has updated/changed. **Note**: Active may be more than 1.
Compiler Directive:	`#DEFINE INCLUDE_SCHEDULING_NEXT_ACTIVE_UPDATED_CALLBACK`
Syntax:	`RmsEventSchedulingNextActiveUpdated(CHAR bookingId[], RmsEventBookingResponse eventBookingResponse)` • **booking-id** - CHAR array which is the booking ID of the booking event this record is for • **eventBookingResponse** - This structure provides additional information about the booking event

RmsEventSchedulingEventEnded

RmsEventSchedulingEventEnded	
Description:	This callback is executed when RMS wants to indicate a booking event has ended.
Compiler Directive:	`#DEFINE INCLUDE_SCHEDULING_EVENT_ENDED_CALLBACK`
Syntax:	`RmsEventSchedulingEventEnded(CHAR bookingId[], RmsEventBookingResponse eventBookingResponse)` • **booking-id** - CHAR array which is the booking ID of the booking event this record is for • **eventBookingResponse** - This structure provides additional information about the booking event

RmsEventSchedulingEventStarted

RmsEventSchedulingEventStarted	
Description:	This callback is executed when RMS wants to indicate a booking event has started.
Compiler Directive:	`#DEFINE INCLUDE_SCHEDULING_EVENT_STARTED_CALLBACK`
Syntax:	`RmsEventSchedulingEventStarted(CHAR bookingId[], RmsEventBookingResponse eventBookingResponse)` • **booking-id** - CHAR array which is the booking ID of the booking event this record is for • **eventBookingResponse** - This structure provides additional information about the booking event

RmsEventSchedulingEventUpdated

RmsEventSchedulingEventUpdated	
Description:	This callback is executed when RMS wants to indicate the a booking event has updated/changed.
Compiler Directive:	`#DEFINE INCLUDE_SCHEDULING_EVENT_UPDATED_CALLBACK`
Syntax:	`RmsEventSchedulingEventUpdated(CHAR bookingId[], RmsEventBookingResponse eventBookingResponse)` • **booking-id** - CHAR array which is the booking ID of the booking event this record is for • **eventBookingResponse** - This structure provides additional information about the booking event

RmsEventSchedulingDailyCount

RmsEventSchedulingDailyCount	
Description:	This callback is executed when RMS provides daily count information such as in when there is a monthly summary update.
Compiler Directive:	`#DEFINE INCLUDE_SCHEDULING_DAILY_COUNT_CALLBACK`
Syntax:	`RmsEventSchedulingDailyCount(CHAR isDefaultLocation, RmsEventBookingDailyCount dailyCount)` • **isDefaultLocation** - boolean, TRUE if the location in the response is the default location • **dailyCount** - A structure with information about a specific date

RmsEventSchedulingMonthlySummaryUpdated

RmsEventSchedulingMonthlySummaryUpdated	
Description:	This callback is executed when RMS wants to indicate the a monthly summary has updated/changed.
Compiler Directive:	`#DEFINE INCLUDE_SCHEDULING_MONTHLY_SUMMARY_UPDATED_CALLBACK`
Syntax:	`RmsEventSchedulingMonthlySummaryUpdated(INTEGER dailyCountsTotal, RmsEventBookingMonthlySummary monthlySummary)`

RmsGuiApi.axi File

The convenience Functions and Send Commands contained in the *RmsGuiApi.axi* file provide the ability to designate a panel as being for either external or internal use, as well as to define several default settings for the panel's scheduling interface, as described below.

Internal and External Scheduling Panel Designation

RMS Scheduling supports the concept of *internal* and *external* scheduling panels. This is to differentiate between panels that are located inside a room versus panels located on the outside, regarding *Doorbell* and *Do Not Disturb* messages as well as *End / Extend Meeting* messages. The same set of scheduling pages are used for both *internal* and *external* panels.

Designating a panel as being for either external or internal use is as simple as including the *RmsGuiApi.axi* file and invoking one of the following convenience Functions or Send Commands:

RmsSetInternalPanel

RmsSetInternalPanel	
Description:	Designates the touch panel for internal use. Internal Panels: • Will display pop-ups with messages from the RMS administrator. • Have no "Ring Doorbell" button displayed (all Internal panels will display a pop-up message and play the doorbell sound when the button is pushed on an external panel) • Have "Do Not Disturb" button displayed • Have "End Meeting" button displayed • Have "Extend Meeting" button displayed
Function Syntax:	`DEFINE_FUNCTION CHAR RmsSetInternalPanel(DEV baseTouchPanelDps, DEV rmsTouchPanelDps)`
SEND_COMMAND Syntax:	`SEND_COMMAND vdvRMSGui, 'SET_INTERNAL_PANEL-<baseTouchPanelDps>,<rmsTouchPanelDps>'`

RmsSetExternalPanel

RmsSetExternalPanel	
Description:	Designates the touch panel for external use. External Panels: • Will not display pop-ups with messages from the RMS administrator. • Has "Ring Doorbell" button displayed • Has no "Do Not Disturb" button displayed (all panels designated as external will show a "Do Not Disturb" message overlaying other controls when the button is pushed on an internal panel) • Has no "End Meeting" button displayed • Has no "Extend Meeting" button displayed
Function Syntax:	`DEFINE_FUNCTION CHAR RmsSetExternalPanel(DEV baseTouchPanelDps, DEV rmsTouchPanelDps)`
SEND_COMMAND Syntax:	`SEND_COMMAND vdvRMSGui, 'SET_EXTERNAL_PANEL-<baseTouchPanelDps>,<rmsTouchPanelDps>'`

Setting Defaults for the Scheduling Panel's UI

The following functions provide the ability to set various defaults on the touch panel's scheduling interface, including default text for meeting *Subject*, *Details*, the default time for meeting *Duration*, as well as removing the keyboard display for bookings:

RmsSetDefaultEventBookingSubject

RmsSetDefaultEventBookingSubject	
Description:	Change the text string for the default display of the meeting subject. Max length: RMS_MAX_PARAM_LEN (defined in *RmsApi.axi*)
Function Syntax:	`DEFINE_FUNCTION CHAR RmsSetDefaultEventBookingSubject(CHAR subject[RMS_MAX_PARAM_LEN]);`
Example:	`RmsSetDefaultEventBookingSubject('Ad Hoc Meeting');` Pre-populates the *Subject* field with the text "*Ad Hoc Meeting*".

RmsSetDefaultEventBookingBody

RmsSetDefaultEventBookingBody	
Description:	Change the text string for the default display of the meeting details. Max length: RMS_MAX_PARAM_LEN (defined in *RmsApi.axi*)
Function Syntax:	`DEFINE_FUNCTION CHAR RmsSetDefaultEventBookingBody(CHAR subject[RMS_MAX_PARAM_LEN]);`
Example:	`RmsSetDefaultEventBookingBody('Weekly Status Update');` Pre-populates the Details field with the text "Weekly Status Update".

RmsSetDefaultEventBookingDuration

RmsSetDefaultEventBookingDuration	
Description:	Enter value for default meeting duration in minutes (default = 60). Max value: varies by scheduling provider and configuration.
Function Syntax:	`DEFINE_FUNCTION CHAR RmsSetDefaultEventBookingDuration(INTEGER duration);`
Example:	`RmsSetDefaultEventBookingDuration(30);` Pre-sets the Duration field to 30 (minutes).

RmsEnableEventBookingAutoKeyboard

RmsEnableEventBookingAutoKeyboard	
Description:	Disable this feature to remove the keyboard for quick bookings that do not require subject or details text entry (*G5 touch panels only)*. Default = True (enabled)
Function Syntax:	`DEFINE_FUNCTION CHAR RmsEnableEventBookingAutoKeyboard(CHAR enableAutoKeyboard);`
Example:	`RmsEnableEventBookingAutoKeyboard(FALSE);` Disables the keyboard display.

Enabling and Disabling the Touch Panel's LEDs

Use the following functions to enable/disable the touch panel's LEDs. By default, these LEDs are enabled, and they indicate whether an event (meeting) is currently in progress:

- Green = No event in progress
- Red = Event in progress

RmsEnableLedSupport

RmsEnableLedSupport	
Description:	Enable/disable the touch panel's LEDs.
Function Syntax:	`DEFINE_FUNCTION CHAR RmsEnableLedSupport(CHAR enableLedSupport);`

Example:	`RmsEnableLedSupport(TRUE);`
	Enables the panel LEDs.
	`RmsEnableLedSupport(FALSE);`
	Disables the panel LEDs.

Setting The Client Time and Date Format

NOTE: *The date and time pattern used by the following functions pattern follows the Java 1.4 SimpleDateFormat conventions.*

Use the following functions to format the Client time and date format:

RmsSetGuiDatePattern

RmsSetGuiDatePattern	
Description:	This function is used to manually set a date pattern which will be used for formatting dates on all touch panels.
Function Syntax:	`DEFINE_FUNCTION CHAR RmsSetGuiDatePattern(CHAR datePattern);`
Arguments:	datePattern: • M/d/yy • yyyy-MM-dd Sample patterns for the date *October 7th, 2013*: • M/d/yy = 10/7/13 • yyyy-MM-dd = 2013-10-07
Example:	`RmsSetGuiDatePattern("yyyy-MM-dd")` Sets the date pattern to *yyyy-MM-dd*

RmsSetGuiTimePattern

RmsSetGuiTimePattern	
Description:	This function is used to manually set a time pattern which will be used for formatting times on all touch panels.
Function Syntax:	`DEFINE_FUNCTION CHAR RmsSetGuiTimePattern(CHAR timePattern);`
Arguments:	timePattern: • h:mm a • HH:mm Sample patterns for the time *5:38 PM*: • h:mm = 5:38 PM • HH:mm = 17:38
Example:	`RmsSetGuiTimePattern("HH:mm")` Sets the date pattern to *HH:mm*

Duet Device API Implementation

Overview

The tables in this section provide detailed descriptions of the standard control functions and associated parameters implemented in the RMS 4.0 Duet Device API, organized by Device Type.

The Device Types included in the RMS 4.0 Duet Device API are:

NOTE: *All Control Method Keys must have a unique argument signature comprising of the list of arguments, the argument names, and the argument data types.*

AUDIO CONFERENCER

AUDIO CONFERENCER		Key	Register?	Data Type	Units	Bargraph Key	Default Value	Reset Value	Param Type	History Track Changes	Status Type
Control Methods	Set Power (ON|OFF)	asset.power	Has-Power	Enum			OFF				
	Set Volume Mute (ON|OFF)	volume.mute	Has-Volume	Boolean			OFF				
	Set Volume Level (0-255)	volume.level	Has-Volume	Level			N/A				
	Set Auto Answer (ON|OFF)	dialer.auto.answer	Has-Auto-Answer	Boolean			N/A				
	Dial Speed Dial Preset	dialer.dial.preset	Has-Dialer	Number			N/A				
	Dial Telephone Number (String)	dialer.dial.number	Has-Dialer	String			N/A				
	Redial	dialer.redial	Has-Dialer	No Args			N/A				
	Set Privacy (ON|OFF)	conferencer.privacy	Has-Privacy	Boolean			ON				
	Set Off Hook (ON|OFF)	dialer.hook	Has-Hook	Boolean			OFF				
Monitored Parameters	Online (ONLINE|OFFLINE)	asset.online	{Always Register}	Enum	N/A	N/A		NO_RESET	ASSET_ONLINE	Yes	MAINTENANCE
	Power Consumption (*This parameter is registered and managed by the RMS server.*)	asset.power.consumption	{Optional Registration}	Decimal	watts	N/A		NO_RESET	POWER_CONSUMPTION	Yes	N/A
	Device Data Initialized (TRUE|FALSE)	asset.data.initialized	{Always Register}	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Module Debug State (ERROR|WARNING|DEBUG|INFO)	asset.debug.level	{Always Register}	Enum	N/A	N/A		NO_RESET	NONE	Yes	N/A
	Power Status (ON|OFF)	asset.power	Has-Power	Enum	N/A	N/A		NO_RESET	ASSET_POWER	Yes	N/A
	Volume Mute (ON|OFF)	volume.mute	Has-Volume	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Volume Level (0-255)	volume.level	Has-Volume	Level	N/A	volume.level		NO_RESET	NONE	No	N/A
	Off Hook (ON|OFF)	dialer.hook	Has-Hook	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Auto Answer (ON|OFF)	dialer.auto.answer	Has-Auto-Answer	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Dialer Status (Enum: *DialerStatus*)	dialer.status	Has-Dialer	Enum	N/A	N/A		NO_RESET	DIALER_STATE	No	N/A
	Privacy (ON|OFF)	conferencer.privacy	Has-Privacy	Boolean	N/A	N/A		NO_RESET			
	Last Incoming Call (String) - not getter fn, only processIncomingCall event	dialer.incoming.call	Has-Dialer	String	N/A	N/A		NO_RESET	NONE	Yes	N/A
Metadata Properties	Duet Device Category ** *DUET MODULES ONLY*	duet.device.category	{Always Register}	String							
	Duet Device Channels ** *DUET MODULES ONLY*	duet.device.channels	{Always Register}	Number							
	Duet Device Levels ** *DUET MODULES ONLY*	duet.device.levels	{Always Register}	Number							
	Duet Device Revision ** *DUET MODULES ONLY*	duet.device.revision	{Always Register}	String							
	Duet Module ** *DUET MODULES ONLY*	duet.module.name	{Always Register}	String							
	Duet Module Version ** *DUET MODULES ONLY*	duet.module.version	{Always Register}	String							
	Physical Device DPS ** *DUET MODULES ONLY*	duet.physical.dps	{Always Register}	String							
	Phonebook Capacity	phonebook.capacity	Has-Phonebook	Number							

CAMERA

CAMERA		Key	Register?	Data Type	Units	Bargraph Key	Default Value	Reset Value	Param Type	History Track Changes	Status Type
Control Methods	Set Power (ON|OFF)	asset.power	Has-Power	Enum			OFF				
	Select Camera Preset (Number: *1 to MAX CAMERA PRESETS*)	camera.preset	Has-Camera-Preset	Number			N/A				
Monitored Parameters	Online (ONLINE|OFFLINE)	asset.online	{Always Register}	Enum	N/A	N/A		NO_RESET	ASSET_ONLINE	Yes	MAINTENANCE
	Power Consumption (*This parameter is registered and managed by the RMS server.*)	asset.power.consumption	{Optional Registration}	Decimal	watts	N/A		NO_RESET	POWER_CONSUMPTION	Yes	N/A
	Device Data Initialized (TRUE|FALSE)	asset.data.initialized	{Always Register}	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Module Debug State (ERROR|WARNING|DEBUG|INFO)	asset.debug.level	{Always Register}	Enum	N/A	N/A		NO_RESET	NONE	Yes	N/A
	Power Status (ON|OFF)	asset.power	Has-Power	Enum	N/A	N/A		NO_RESET	ASSET_POWER	Yes	N/A
	Auto Focus (ON|OFF)	camera.focus.auto	Has-Auto-Focus	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Auto Iris (ON|OFF)	camera.iris.auto	Has-Auto-Iris	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Pan Position (0-255)	camera.pan.position	Has-Pan-Tilt	Level	N/A	N/A		NO_RESET	NONE	No	N/A
	Tilt Position (0-255)	camera.tilt.position	Has-Pan-Tilt	Level	N/A	N/A		NO_RESET	NONE	No	N/A
	Last Selected Camera Preset (Number)	camera.preset	Has-Camera-Preset	Number	N/A	N/A		NO_RESET	NONE	No	N/A
	Focus Level (0-255)	camera.focus.level	Has-Focus	Level	N/A	general		NO_RESET	NONE	No	N/A
	Iris Level (0-255)	camera.iris.level	Has-Iris	Level	N/A	general		NO_RESET	NONE	No	N/A
	Zoom Level (0-255)	camera.zoom.level	Has-Zoom	Level	N/A	general		NO_RESET	NONE	No	N/A
	Pan Speed (0-255)	camera.pan.speed	Has-Pan-Tilt-Speed	Level	N/A	general		NO_RESET	NONE	No	N/A
	Tilt Speed (0-255)	camera.tilt.speed	Has-Pan-Tilt-Speed	Level	N/A	general		NO_RESET	NONE	No	N/A
	Iris Speed (0-255)	camera.iris.speed	Has-Iris-Speed	Level	N/A	general		NO_RESET	NONE	No	N/A
	Focus Speed (0-255)	camera.focus.speed	Has-Focus-Speed	Level	N/A	general		NO_RESET	NONE	No	N/A
	Zoom Speed (0-255)	camera.zoom.speed	Has-Zoom-Speed	Level	N/A	general		NO_RESET	NONE	No	N/A
Metadata Properties	Duet Device Category ** *DUET MODULES ONLY*	duet.device.category	{Always Register}	String							
	Duet Device Channels ** *DUET MODULES ONLY*	duet.device.channels	{Always Register}	Number							
	Duet Device Levels ** *DUET MODULES ONLY*	duet.device.levels	{Always Register}	Number							
	Duet Device Revision ** *DUET MODULES ONLY*	duet.device.revision	{Always Register}	String							
	Duet Module ** *DUET MODULES ONLY*	duet.module.name	{Always Register}	String							
	Duet Module Version ** *DUET MODULES ONLY*	duet.module.version	{Always Register}	String							
	Physical Device DPS ** *DUET MODULES ONLY*	duet.physical.dps	{Always Register}	String							
	Camera Preset Count	camera.preset.count	Has-Camera-Preset	Number							

DIGITAL SATELLITE SYSTEM

DIGITAL SATELLITE SYSTEM		Key	Register?	Data Type	Units	Bargraph Key	Default Value	Reset Value	Param Type	History Track Changes	Status Type
Control Methods	Set Power (ON|OFF)	asset.power	Has-Power	Enum			OFF				
	Set Station (String)	tuner.station	Has-Station	String			N/A				
	Select Station Preset (1 to MAX STATION PRESETS)	tuner.station.preset	Has-Station-Preset	Number			N/A				
	Select Tuner Band (Enum: *Band*)	tuner.band	Has-Tuner-Band	Enum			N/A				
Monitored Parameters	Online (ONLINE|OFFLINE)	asset.online	{Always Register}	Enum	N/A	N/A		NO_RESET	ASSET_ONLINE	Yes	MAINTENANCE
	Power Consumption (*This parameter is registered and managed by the RMS server.*)	asset.power.consumption	{Optional Registration}	Decimal	watts	N/A		NO_RESET	POWER_CONSUMPTION	Yes	N/A
	Device Data Initialized (TRUE|FALSE)	asset.data.initialized	{Always Register}	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Module Debug State (ERROR|WARNING|DEBUG|INFO)	asset.debug.level	{Always Register}	Enum	N/A	N/A		NO_RESET	NONE	Yes	N/A
	Power Status (ON|OFF)	asset.power	Has-Power	Enum	N/A	N/A		NO_RESET	ASSET_POWER	Yes	N/A
	Tuner Band (Enum: *Band*)	tuner.band	Has-Tuner-Band	Enum	N/A	N/A		NO_RESET	NONE	No	N/A
	Station (String)	tuner.station	Has-Station	String	N/A	N/A		NO_RESET	NONE	No	N/A
Metadata Properties	Duet Device Category ** *DUET MODULES ONLY*	duet.device.category	{Always Register}	String							
	Duet Device Channels ** *DUET MODULES ONLY*	duet.device.channels	{Always Register}	Number							
	Duet Device Levels ** *DUET MODULES ONLY*	duet.device.levels	{Always Register}	Number							
	Duet Device Revision ** *DUET MODULES ONLY*	duet.device.revision	{Always Register}	String							
	Duet Module ** *DUET MODULES ONLY*	duet.module.name	{Always Register}	String							
	Duet Module Version ** *DUET MODULES ONLY*	duet.module.version	{Always Register}	String							
	Physical Device DPS ** *DUET MODULES ONLY*	duet.physical.dps	{Always Register}	String							
	Station Preset Count	tuner.station.count	Has-Station-Preset	Number							
	Tuner Band Count	tuner.band.count	Has-Tuner-Band	Number							

DIGITAL VIDEO RECORDER

DIGITAL VIDEO RECORDER		Key	Register?	Data Type	Units	Bargraph Key	Default Value	Reset Value	Param Type	History Track Changes	Status Type
Control Methods	Set Power (ON|OFF)	asset.power	Has-Power	Enum			OFF				
	Play	transport.play	{Always Register}	None							
	Stop	transport.stop	{Always Register}	None							
	Pause	transport.pause	{Always Register}	None							
	Next	transport.next	{Always Register}	None							
	Previous	transport.previous	{Always Register}	None							
	Record	transport.record	{Always Register}	None							
	Select Input Source (Enum: *Source*)	source.input	Has-Input-Select	Number			N/A				
	Set Station (String)	tuner.station	Has-Station	String			N/A				
	Select Station Preset (1 to MAX TUNER PRESETS)	tuner.station.preset	Has-Station-Preset	Number			N/A				
	Select Tuner Band (Enum: *Band*)	tuner.band	Has-Tuner-Band	Enum							
Monitored Parameters	Online (ONLINE|OFFLINE)	asset.online	{Always Register}	Enum	N/A	N/A		NO_RESET	ASSET_ONLINE	Yes	MAINTENANCE
	Power Consumption (*This parameter is registered and managed by the RMS server.*)	asset.power.consumption	{Optional Registration}	Decimal	watts	N/A		NO_RESET	POWER_CONSUMPTION	Yes	N/A
	Device Data Initialized (TRUE|FALSE)	asset.data.initialized	{Always Register}	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Module Debug State (ERROR|WARNING|DEBUG|INFO)	asset.debug.level	{Always Register}	Enum	N/A	N/A		NO_RESET	NONE	Yes	N/A
	Power Status (ON|OFF)	asset.power	Has-Power	Enum	N/A	N/A		NO_RESET	ASSET_POWER	Yes	N/A
	Transport State (Enum: *Transport*)	transport.state	Has-Disc-Transport	Enum	N/A	N/A		NO_RESET	TRANSPORT_STATE	Yes	N/A
	Input Source (Enum: *Source*)	source.input	Has-Input-Select	Enum	N/A	N/A		NO_RESET	SOURCE_STATE	Yes	N/A
	Tuner Band (Enum: *Band*)	tuner.band	Has-Tuner-Band	Enum	N/A	N/A		NO_RESET	NONE	No	N/A
	Run Time (Based on established dependent parameter states) in Hours *Special Notes*: This parameter will listen for transport state events and device power events and track the number of hours (fractional) that the device is in a state other than STOP.	transport.runtime	Has-Disc-Transport	Decimal	Hours	N/A		0	TRANSPORT_USAGE	Yes	N/A
	Station (String)	tuner.station	Has-Station	String	N/A	N/A		NO_RESET	NONE	No	N/A
Metadata Properties	Duet Device Category ** *DUET MODULES ONLY*	duet.device.category	{Always Register}	String							
	Duet Device Channels ** *DUET MODULES ONLY*	duet.device.channels	{Always Register}	Number							
	Duet Device Levels ** *DUET MODULES ONLY*	duet.device.levels	{Always Register}	Number							
	Duet Device Revision ** *DUET MODULES ONLY*	duet.device.revision	{Always Register}	String							
	Duet Module ** *DUET MODULES ONLY*	duet.module.name	{Always Register}	String							
	Duet Module Version ** *DUET MODULES ONLY*	duet.module.version	{Always Register}	String							
	Physical Device DPS ** *DUET MODULES ONLY*	duet.physical.dps	{Always Register}	String							
	Station Preset Count	tuner.station.count	Has-Station-Preset	Number							
	Tuner Band Count	tuner.band.count	Has-Tuner-Band	Number							
	Input Source Count	source.input.count	Has-Input-Select	Number							

DISC DEVICE

DISC DEVICE		Key	Register?	Data Type	Units	Bargraph Key	Default Value	Reset Value	Param Type	History Track Changes	Status Type
Control Methods	Set Power (ON|OFF)	asset.power	Has-Power	Enum			OFF				
	Play	transport.play	Has-Disc-Transport	None							
	Stop	transport.stop	Has-Disc-Transport	None							
	Pause	transport.pause	Has-Disc-Transport	None							
	Select Disc (Number: *1 to Disc Capacity*)	disc.select	Has-Disc-Select	Number			1				
	Next Track/Chapter	transport.next	Has-Disc-Transport	None							
	Previous Track/Chapter	transport.previous	Has-Disc-Transport	None							
	Select Track/Chapter (Number: *1 to Max Chapters*)	disc.track.select	{Always Register}	Number			N/A				
Monitored Parameters	Online (ONLINE|OFFLINE)	asset.online	{Always Register}	Enum	N/A	N/A		NO_RESET	ASSET_ONLINE	Yes	MAINTENANCE
	Power Consumption (*This parameter is registered and managed by the RMS server.*)	asset.power.consumption	{Optional Registration}	Decimal	watts	N/A		NO_RESET	POWER_CONSUMPTION	Yes	N/A
	Device Data Initialized (TRUE|FALSE)	asset.data.initialized	{Always Register}	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Module Debug State (ERROR|WARNING|DEBUG|INFO)	asset.debug.level	{Always Register}	Enum	N/A	N/A		NO_RESET	NONE	Yes	N/A
	Power Status (ON|OFF)	asset.power	Has-Power	Enum	N/A	N/A		NO_RESET	ASSET_POWER	Yes	N/A
	Transport State (Enum: *Transport*)	transport.state	Has-Disc-Transport	Enum	N/A	N/A		NO_RESET	TRANSPORT_STATE	Yes	N/A
	Disc Number	disc.selected	Has-Disc-Select	Number	N/A	N/A		NO_RESET	NONE	No	N/A
	Disc Duration	disc.duration	{Always Register}	String	N/A	N/A		NO_RESET	NONE	No	N/A
	Number of Tracks	disc.track.count	{Always Register}	Number	N/A	N/A		NO_RESET	NONE	No	N/A
	Disc Type	disc.type	{Always Register}	Enum	N/A	N/A		NO_RESET	NONE	No	N/A
	Run Time (Based on established dependent parameter states) in Hours *Special Notes*: This parameter will listen for transport state events and device power events and track the number of hours (fractional) that the device is in a state other than STOP.	transport.runtime	Has-Disc-Transport	Decimal	Hours	N/A		0	TRANSPORT_USAGE	Yes	N/A
	Track/Chapter Number	disc.track.selected	{Always Register}	Number	N/A	N/A		NO_RESET	NONE	No	N/A
	Track/Chapter Duration	disc.track.duration	{Always Register}	String	N/A	N/A		NO_RESET	NONE	No	N/A
Metadata Properties	Duet Device Category ** *DUET MODULES ONLY*	duet.device.category	{Always Register}	String							
	Duet Device Channels ** *DUET MODULES ONLY*	duet.device.channels	{Always Register}	Number							
	Duet Device Levels ** *DUET MODULES ONLY*	duet.device.levels	{Always Register}	Number							
	Duet Device Revision ** *DUET MODULES ONLY*	duet.device.revision	{Always Register}	String							
	Duet Module ** *DUET MODULES ONLY*	duet.module.name	{Always Register}	String							
	Duet Module Version ** *DUET MODULES ONLY*	duet.module.version	{Always Register}	String							
	Physical Device DPS ** *DUET MODULES ONLY*	duet.physical.dps	{Always Register}	String							
	Disc Capacity (Number)	disc.capacity	{Always Register}	Number							

DOCUMENT CAMERA

DOCUMENT CAMERA		Key	Register?	Data Type	Units	Bargraph Key	Default Value	Reset Value	Param Type	History Track Changes	Status Type
Control Methods	Set Power (ON|OFF)	asset.power	Has-Power	Enum			OFF				
	Select Input Source (Enum: *Source*)	source.input	Has-Input-Select	Number			N/A				
	Upper Light (ON|OFF)	document.camera.light.upper.power	Has-Upper-Light	Boolean			ON				
	Lower Light (ON|OFF)	document.camera.light.lower.power	Has-Lower-Light	Boolean			ON				
Monitored Parameters	Online (ONLINE|OFFLINE)	asset.online	{Always Register}	Enum	N/A	N/A		NO_RESET	ASSET_ONLINE	Yes	MAINTENANCE
	Power Consumption (*This parameter is registered and managed by the RMS server.*)	asset.power.consumption	{Optional Registration}	Decimal	watts	N/A		NO_RESET	POWER_CONSUMPTION	Yes	N/A
	Device Data Initialized (TRUE|FALSE)	asset.data.initialized	{Always Register}	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Module Debug State (ERROR|WARNING|DEBUG|INFO)	asset.debug.level	{Always Register}	Enum	N/A	N/A		NO_RESET	NONE	Yes	N/A
	Power Status (ON|OFF)	asset.power	Has-Power	Enum	N/A	N/A		NO_RESET	ASSET_POWER	Yes	N/A
	Input Source (Enum: *Source*)	source.input	Has-Input-Select	Enum	N/A	N/A		NO_RESET	SOURCE_STATE	Yes	N/A
	Auto Focus (ON|OFF)	camera.focus.auto	Has-Auto-Focus	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Auto Iris (ON|OFF)	camera.iris.auto	Has-Auto-Iris	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Focus Level (0-255)	camera.focus.level	Has-Focus	Focus	N/A	general		NO_RESET	NONE	No	N/A
	Iris Level (0-255)	camera.iris.level	Has-Iris	Level	N/A	general		NO_RESET	NONE	No	N/A
	Zoom Level (0-255)	camera.zoom.level	Has-Zoom	Level	N/A	general		NO_RESET	NONE	No	N/A
	Focus Speed (0-255)	camera.focus.speed	Has-Focus-Speed	Level	N/A	general		NO_RESET	NONE	No	N/A
	Iris Speed (0-255)	camera.iris.speed	Has-Iris-Speed	Level	N/A	general		NO_RESET	NONE	No	N/A
	Zoom Speed (0-255)	camera.zoom.speed	Has-Zoom-Speed	Level	N/A	general		NO_RESET	NONE	No	N/A
	Upper Light (ON|OFF)	document.camera.light.upper.power	Has-Upper-Light	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Lower Light (ON|OFF)	document.camera.light.lower.power	Has-Lower-Light	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Upper Light Hours (Based on established dependent parameter states)	lamp.consumption.upper	Has-Upper-Light	Number	Hours	lamp.consumption		0	LAMP_USAGE	Yes	MAINTENANCE
	Lower Light Hours (Based on established dependent parameter states)	lamp.consumption.lower	Has-Lower-Light	Number	Hours	lamp.consumption		0	LAMP_USAGE	Yes	MAINTENANCE
Metadata Properties	Duet Device Category ** *DUET MODULES ONLY*	duet.device.category	{Always Register}	String							
	Duet Device Channels ** *DUET MODULES ONLY*	duet.device.channels	{Always Register}	Number							
	Duet Device Levels ** *DUET MODULES ONLY*	duet.device.levels	{Always Register}	Number							
	Duet Device Revision ** *DUET MODULES ONLY*	duet.device.revision	{Always Register}	String							
	Duet Module ** *DUET MODULES ONLY*	duet.module.name	{Always Register}	String							
	Duet Module Version ** *DUET MODULES ONLY*	duet.module.version	{Always Register}	String							
	Physical Device DPS ** *DUET MODULES ONLY*	duet.physical.dps	{Always Register}	String							
	Input Source Count	source.input.count	Has-Input-Select	Number							

HVAC

HVAC		Key	Register?	Data Type	Units	Bargraph Key	Default Value	Reset Value	Param Type	History Track Changes	Status Type
Control Methods	Set Cool Setpoint (Level)	hvac.cool.setpoint	Has-CoolSetpoint	Number			N/A				
	Set Heat Setpoint (Level)	hvac.heat.setpoint	{Always Register}	Number			N/A				
	Set Fan (Enum: *FanState*)	hvac.fan.state	{Always Register}	Enum			N/A				
	Set HVAC State (Enum: *HVACState*)	hvac.state	{Always Register}	Enum			N/A				
	Set Thermostat Hold (ON|OFF)	hvac.thermostat.hold	Has-Hold	Boolean			FALSE				
	Set Thermostat Lock (ON|OFF)	hvac.thermostat.lock	Has-Lock	Boolean			FALSE				
Monitored Parameters	Online (ONLINE|OFFLINE)	asset.online	{Always Register}	Enum	N/A	N/A		NO_RESET	ASSET_ONLINE	Yes	MAINTENANCE
	Power Consumption (*This parameter is registered and managed by the RMS server.*)	asset.power.consumption	{Optional Registration}	Decimal	watts	N/A		NO_RESET	POWER_CONSUMPTION	Yes	N/A
	Device Data Initialized (TRUE|FALSE)	asset.data.initialized	{Always Register}	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Module Debug State (ERROR|WARNING|DEBUG|INFO)	asset.debug.level	{Always Register}	Enum	N/A	N/A		NO_RESET	NONE	Yes	N/A
	Cool Setpoint	hvac.cool.setpoint	Has-CoolSetpoint	Number	°	N/A		NO_RESET	NONE	No	N/A
	Heat Setpoint	hvac.heat.setpoint	{Always Register}	Number	°	N/A		NO_RESET	NONE	No	N/A
	Power Status (ON|OFF) *** TRACK WITH FAN POWER STATUS EVENT*	asset.power	{Always Register}	Enum	N/A	N/A		NO_RESET	ASSET_POWER	Yes	N/A
	Fan Status (Enum: *PowerState*)	hvac.fan.status	{Always Register}	Enum	N/A	N/A		NO_RESET	NONE	No	N/A
	Thermostat Hold (ON|OFF)	hvac.thermostat.hold	Has-Hold	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Thermostat Lock (ON|OFF)	hvac.thermostat.lock	Has-Lock	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	HVAC State (Enum: *HVACState*)	hvac.state	{Always Register}	Enum	N/A	N/A		NO_RESET	HVAC_STATE	No	N/A
	Indoor Temperature	temperature.indoor	{Always Register}	Number	°	temperature		NO_RESET	TEMPERATURE	Yes	N/A
	Outdoor Temperature	temerature.outboor	Has-Outdoor-Temperature	Number	°	temperature		NO_RESET	TEMPERATURE	Yes	N/A
Metadata Properties	Duet Device Category *** DUET MODULES ONLY*	duet.device.category	{Always Register}	String							
	Duet Device Channels *** DUET MODULES ONLY*	duet.device.channels	{Always Register}	Number							
	Duet Device Levels *** DUET MODULES ONLY*	duet.device.levels	{Always Register}	Number							
	Duet Device Revision *** DUET MODULES ONLY*	duet.device.revision	{Always Register}	String							
	Duet Module *** DUET MODULES ONLY*	duet.module.name	{Always Register}	String							
	Duet Module Version *** DUET MODULES ONLY*	duet.module.version	{Always Register}	String							
	Physical Device DPS *** DUET MODULES ONLY*	duet.physical.dps	{Always Register}	String							
	Cool Setpoint Range: Hi	hvac.cool.setpoint.hi	Has-CoolSetpoint	Number							
	Cool Setpoint Range: Low	hvac.cool.setpoint.low	Has-CoolSetpoint	Number							
	Heat Setpoint Range: Hi	hvac.heat.setpoint.hi	{Always Register}	Number							
	Heat Setpoint Range: Low	hvac.heat.setpoint.low	{Always Register}	Number							
	Temperature Scale	temperature.scale	Has-Temperature-Scale	String							

***Note**: The RMS SDK includes support for "HVAC" as a RMS NetLinx Monitoring module but does not include a RMS Duet Monitoring Module for "HVAC".*

LIGHT SYSTEM

LIGHT SYSTEM		Key	Register?	Data Type	Units	Bargraph Key	Default Value	Reset Value	Param Type	History Track Changes	Status Type
Control Methods	Set Light On (Zone Name)	light.power.on	{Always Register}	Enum			N/A				
	Set Light Off (Zone Name)	light.power.off	{Always Register}	Enum			N/A				
	Recall Lighting Scene	light.scene	{Always Register}	Enum			N/A				
	Set Light Level (Zone Name, Level 0-255, Time)	light.level	Has-Light-Level	Enum Level Level			N/A				
Monitored Parameters	Online (ONLINE|OFFLINE)	asset.online	{Always Register}	Enum	N/A	N/A		NO_RESET	ASSET_ONLINE	Yes	MAINTENANCE
	Device Data Initialized (TRUE|FALSE)	asset.data.initialized	{Always Register}	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Module Debug State (ERROR|WARNING|DEBUG|INFO)	asset.debug.level	{Always Register}	Enum	N/A	N/A		NO_RESET	NONE	Yes	N/A
	Light 1..<N> Power (ON|OFF)	light.power.<N>	{Always Register}	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Light 1..<N> Level (Number)	light.level.<N>	Has-Light-Level	Level	N/A	light.level		NO_RESET	LIGHT_LEVEL	No	N/A
	Light 1..<N> Power Consumption (Decimal : Watts)	light.power.consumption.<N>	{Always Register}	Decimal	watts	N/A		NO_RESET	POWER_CONSUMPTION	Yes	N/A
Metadata Properties	Duet Device Category ** *DUET MODULES ONLY*	duet.device.category	{Always Register}	String							
	Duet Device Channels ** *DUET MODULES ONLY*	duet.device.channels	{Always Register}	Number							
	Duet Device Levels ** *DUET MODULES ONLY*	duet.device.levels	{Always Register}	Number							
	Duet Device Revision ** *DUET MODULES ONLY*	duet.device.revision	{Always Register}	String							
	Duet Module ** *DUET MODULES ONLY*	duet.module.name	{Always Register}	String							
	Duet Module Version ** *DUET MODULES ONLY*	duet.module.version	{Always Register}	String							
	Physical Device DPS ** *DUET MODULES ONLY*	duet.physical.dps	{Always Register}	String							
	Lights Count	light.count	{Always Register}	Number							

MONITOR

MONITOR		Key	Register?	Data Type	Units	Bargraph Key	Default Value	Reset Value	Param Type	History Track Changes	Status Type
Control Methods	Set Power (ON|OFF)	asset.power	Has-Power	Enum			OFF				
	Set Volume Mute (ON|OFF)	volume.mute	Duet: Has-Volume XDD: Has-Volume-Mute-Cycle	Boolean			OFF				
	Set Volume Level (0-255)	volume.level	Has-Volume	Level			N/A				
	Set Aspect Ratio (Enum: *Aspect Ratio*)	display.aspect.ratio	Has-Aspect-Ratio	Enum			N/A				
	Select Input Source (Enum: *Source*)	source.input	Has-Input-Select	Enum			N/A				
Monitored Parameters	Online (ONLINE|OFFLINE)	asset.online	{Always Register}	Enum	N/A	N/A		NO_RESET	ASSET_ONLINE	Yes	MAINTENANCE
	Power Consumption (*This parameter is registered and managed by the RMS server.*)	asset.power.consumption	{Optional Registration}	Decimal	watts	N/A		NO_RESET	POWER_CONSUMPTION	Yes	N/A
	Device Data Initialized (TRUE|FALSE)	asset.data.initialized	{Always Register}	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Module Debug State (ERROR|WARNING|DEBUG|INFO)	asset.debug.level	{Always Register}	Enum	N/A	N/A		NO_RESET	NONE	Yes	N/A
	Power Status (ON|OFF)	asset.power	Has-Power	Enum	N/A	N/A		NO_RESET	ASSET_POWER	Yes	N/A
	Volume Mute (ON|OFF)	volume.mute	Has-Volume	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Volume Level (0-255)	volume.level	Has-Volume	Level	N/A	volume.level		NO_RESET	NONE	No	N/A
	Display Usage (time consumption based on PowerState)	display.usage	{Always Register}	Decimal	Hours	display.usage		0	DISPLAY_USAGE	Yes	MAINTENANCE
	Aspect Ratio (Enum: *Aspect Ratio*)	display.aspect.ratio	Has-Aspect-Ratio	Enum	N/A	N/A		NO_RESET	NONE	No	N/A
	Active Window (LEFT|RIGHT|MAIN|SUB)	display.window.active	Duet: {Always Register} XDD: not registered	Enum	N/A	N/A		NO_RESET	NONE	No	N/A
	Video Type (Enum: *VideoType*)	display.video.type	{Always Register}	Enum	N/A	N/A		NO_RESET	NONE	No	N/A
	Input Source (Enum: *Source*)	source.input	Has-Input-Select	Enum	N/A	N/A		NO_RESET	SOURCE_STATE	Yes	N/A
Metadata Properties	Duet Device Category ** *DUET MODULES ONLY*	duet.device.category	{Always Register}	String							
	Duet Device Channels ** *DUET MODULES ONLY*	duet.device.channels	{Always Register}	Number							
	Duet Device Levels ** *DUET MODULES ONLY*	duet.device.levels	{Always Register}	Number							
	Duet Device Revision ** *DUET MODULES ONLY*	duet.device.revision	{Always Register}	String							
	Duet Module ** *DUET MODULES ONLY*	duet.module.name	{Always Register}	String							
	Duet Module Version ** *DUET MODULES ONLY*	duet.module.version	{Always Register}	String							
	Physical Device DPS ** *DUET MODULES ONLY*	duet.physical.dps	{Always Register}	String							
	Input Source Count	source.input.count	Has-Input-Select	Number							
	Shutting Down Time ** *XDD MODULES ONLY*	shutting.down.time									
	Start Up Time ** *XDD MODULES ONLY*	start.up.time									

RECEIVER

RECEIVER		Key	Register?	Data Type	Units	Bargraph Key	Default Value	Reset Value	Param Type	History Track Changes	Status Type
Control Methods	Set Power (ON|OFF)	asset.power	Has-Power	Enum			OFF				
	Set Volume Mute (ON|OFF)	volume.mute	Has-Volume	Boolean			OFF				
	Set Volume Level (0-255)	volume.level	Has-Volume	Level			N/A				
	Set Balance (-128 to 128)	preamp.balance	Has-Balance	Level			OFF				
	Set Loudness State (ON|OFF)	preamp.loudness	Has-Loudness	Boolean			N/A				
	Set Treble Level (0-255)	preamp.treble	Has-Treble	Level			OFF				
	Set Bass Level (0-255)	preamp.bass	Has-Bass	Level			N/A				
	Select Input Source (Enum: *Source*)	source.input	Has-Input-Select	Number			OFF				
	Set Station (String)	tuner.station	Has-Station	String			N/A				
	Select Station Preset (1 to MAX STATION PRESETS)	tuner.station.preset	Has-Station-Preset	Number			OFF				
	Select Tuner Band (Enum: *Band*)	tuner.band	Has-Tuner-Band	Enum			N/A				
Monitored Parameters	Online (ONLINE|OFFLINE)	asset.online	{Always Register}	Enum	N/A	N/A		NO_RESET	ASSET_ONLINE	Yes	MAINTENANCE
	Power Consumption (*This parameter is registered and managed by the RMS server.*)	asset.power.consumption	{Optional Registration}	Decimal	watts	N/A		NO_RESET	POWER_CONSUMPTION	Yes	N/A
	Device Data Initialized (TRUE|FALSE)	asset.data.initialized	{Always Register}	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Module Debug State (ERROR|WARNING|DEBUG|INFO)	asset.debug.level	{Always Register}	Enum	N/A	N/A		NO_RESET	NONE	Yes	N/A
	Power Status (ON|OFF)	asset.power	Has-Power	Enum	N/A	N/A		NO_RESET	ASSET_POWER	Yes	N/A
	Volume Mute (ON|OFF)	volume.mute	Has-Volume	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Volume Level (0-255)	volume.level	Has-Volume	Level	N/A	volume.level		NO_RESET	NONE	No	N/A
	Balance (-128 to 128)	preamp.balance	Has-Balance	Level	N/A	general		NO_RESET	NONE	No	N/A
	Loudness State (ON|OFF)	preamp.loudness	Has-Loudness	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Treble Level (0-255)	preamp.treble	Has-Treble	Level	N/A	general		NO_RESET	NONE	No	N/A
	Bass Level (0-255)	preamp.bass	Has-Bass	Level	N/A	general		NO_RESET	NONE	No	N/A
	Input Source (Enum: *Source*)	source.input	Has-Input-Select	Enum	N/A	N/A		NO_RESET	SOURCE_STATE	Yes	N/A
	Tuner Band (Enum: *Band*)	tuner.band	Has-Tuner-Band	Enum	N/A	N/A		NO_RESET	NONE	No	N/A
	Station (Channel)	tuner.station	Has-Station	String	N/A	N/A		NO_RESET	NONE	No	N/A
Metadata Properties	Duet Device Category ** *DUET MODULES ONLY*	duet.device.category	{Always Register}	String							
	Duet Device Channels ** *DUET MODULES ONLY*	duet.device.channels	{Always Register}	Number							
	Duet Device Levels ** *DUET MODULES ONLY*	duet.device.levels	{Always Register}	Number							
	Duet Device Revision ** *DUET MODULES ONLY*	duet.device.revision	{Always Register}	String							
	Duet Module ** *DUET MODULES ONLY*	duet.module.name	{Always Register}	String							
	Duet Module Version ** *DUET MODULES ONLY*	duet.module.version	{Always Register}	String							
	Physical Device DPS ** *DUET MODULES ONLY*	duet.physical.dps	{Always Register}	String							
	Input Source Count	source.input.count	Has-Input-Select	Number							
	Station Preset Count	tuner.station.count	Has-Station-Preset	Number							
	Tuner Band Count	tuner.band.count	Has-Tuner-Band	Number							

SECURITY SYSTEM

SECURITY SYSTEM		Key	Register?	Data Type	Units	Bargraph Key	Default Value	Reset Value	Param Type	History Track Changes	Status Type
Control Methods	Set Security State (Enum: *SecurityState*, *Password*)	security.system.state	{Always Register}	Enum String			N/A				
Monitored Parameters	Online (ONLINE|OFFLINE)	asset.online	{Always Register}	Enum	N/A	N/A		NO_RESET	ASSET_ONLINE	Yes	MAINTENANCE
	Power Consumption (*This parameter is registered and managed by the RMS server.*)	asset.power.consumption	{Optional Registration}	Decimal	watts	N/A		NO_RESET	POWER_CONSUMPTION	Yes	N/A
	Device Data Initialized (TRUE|FALSE)	asset.data.initialized	{Always Register}	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Module Debug State (ERROR|WARNING|DEBUG|INFO)	asset.debug.level	{Always Register}	Enum	N/A	N/A		NO_RESET	NONE	Yes	N/A
	Power Status (ON|OFF) *** FIXED POWER STATE = 'ON'	asset.power	Has-Power	Enum	N/A	N/A		NO_RESET	ASSET_POWER	Yes	N/A
	Security Status (Enum: *SecurityStatus*)	security.system.status	{Always Register}	Enum	N/A	N/A		NO_RESET	SECURITY_STATE	No	N/A
	OK to Arm (TRUE|FALSE)	security.system.oktoarm	Has-OK-To-Arm	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
Metadata Properties	Duet Device Category ** *DUET MODULES ONLY*	duet.device.category	{Always Register}	String							
	Duet Device Channels ** *DUET MODULES ONLY*	duet.device.channels	{Always Register}	Number							
	Duet Device Levels ** *DUET MODULES ONLY*	duet.device.levels	{Always Register}	Number							
	Duet Device Revision ** *DUET MODULES ONLY*	duet.device.revision	{Always Register}	String							
	Duet Module ** *DUET MODULES ONLY*	duet.module.name	{Always Register}	String							
	Duet Module Version ** *DUET MODULES ONLY*	duet.module.version	{Always Register}	String							
	Physical Device DPS ** *DUET MODULES ONLY*	duet.physical.dps	{Always Register}	String							

Note: *The RMS SDK includes support for "Security System" as a RMS NetLinx Monitoring module but does not include a RMS Duet Monitoring Module for "Security System".14*

SETTOP BOX

SETTOP BOX		Key	Register?	Data Type	Units	Bargraph Key	Default Value	Reset Value	Param Type	History Track Changes	Status Type
Control Methods	Set Power (ON|OFF)	asset.power	Has-Power	Enum			OFF				
	Set Station (String)	tuner.station	Has-Station	String			N/A				
	Select Station Preset (1 to MAX STATION PRESETS)	tuner.station.preset	Has-Station-Preset	Number			N/A				
	Select Tuner Band (Enum: *Band*)	tuner.band	Has-Tuner-Band	Enum			N/A				
	Set Volume Mute (ON|OFF)	volume.mute	Has-Volume	Boolean			OFF				
	Set Volume Level (0-255)	volume.level	Has-Volume	Level			N/A				
Monitored Parameters	Online (ONLINE|OFFLINE)	asset.online	{Always Register}	Enum	N/A	N/A		NO_RESET	ASSET_ONLINE	Yes	MAINTENANCE
	Power Consumption (*This parameter is registered and managed by the RMS server.*)	asset.power.consumption	{Optional Registration}	Decimal	watts	N/A		NO_RESET	POWER_CONSUMPTION	Yes	N/A
	Device Data Initialized (TRUE|FALSE)	asset.data.initialized	{Always Register}	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Module Debug State (ERROR|WARNING|DEBUG|INFO)	asset.debug.level	{Always Register}	Enum	N/A	N/A		NO_RESET	NONE	Yes	N/A
	Power Status (ON|OFF)	asset.power	Has-Power	Enum	N/A	N/A		NO_RESET	ASSET_POWER	Yes	N/A
	Tuner Band (Enum: *Band*)	tuner.band	Has-Tuner-Band	Enum	N/A	N/A		NO_RESET	NONE	No	N/A
	Station (String)	tuner.station	Has-Station	String	N/A	N/A		NO_RESET	NONE	No	N/A
	Volume Mute (ON|OFF)	volume.mute	Has-Volume	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Volume Level (0-255)	volume.level	Has-Volume	Level	N/A	volume.level		NO_RESET	NONE	No	N/A
Metadata Properties	Duet Device Category ** *DUET MODULES ONLY*	duet.device.category	{Always Register}	String							
	Duet Device Channels ** *DUET MODULES ONLY*	duet.device.channels	{Always Register}	Number							
	Duet Device Levels ** *DUET MODULES ONLY*	duet.device.levels	{Always Register}	Number							
	Duet Device Revision ** *DUET MODULES ONLY*	duet.device.revision	{Always Register}	String							
	Duet Module ** *DUET MODULES ONLY*	duet.module.name	{Always Register}	String							
	Duet Module Version ** *DUET MODULES ONLY*	duet.module.version	{Always Register}	String							
	Physical Device DPS ** *DUET MODULES ONLY*	duet.physical.dps	{Always Register}	String							
	Station Preset Count	tuner.station.count	Has-Station-Preset	Number							
	Tuner Band Count	tuner.band.count	Has-Tuner-Band	Number							

SWITCHER

SWITCHER		Key	Register?	Data Type	Units	Bargraph Key	Default Value	Reset Value	Param Type	History Track Changes	Status Type
Control Methods	Set Volume Mute (ON|OFF)	volume.mute	Has-Volume	Boolean			OFF				
	Set Volume Level (0-255)	volume.level	Has-Volume	Level			N/A				
	Set Gain Mute (ON|OFF)	gain.mute	Has-Gain	Boolean			N/A				
	Set Gain Level (0-255)	gain.level	Has-Gain	Level			N/A				
	Select Switcher Preset (1 to MAX SWITCHER PRESET)	switcher.preset	Has-Switcher-Preset	Number			N/A				
	Switch (ALL|AUDIO|VIDEO, Input, Output)	switcher.switch	{Always Register}	Enum Number Number			ALL				
Monitored Parameters	Online (ONLINE|OFFLINE)	asset.online	{Always Register}	Enum	N/A	N/A		NO_RESET	ASSET_ONLINE	Yes	MAINTENANCE
	Power Consumption (*This parameter is registered and managed by the RMS server.*)	asset.power.consumption	{Optional Registration}	Decimal	watts	N/A		NO_RESET	POWER_CONSUMPTION	Yes	N/A
	Device Data Initialized (TRUE|FALSE)	asset.data.initialized	{Always Register}	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Module Debug State (ERROR|WARNING|DEBUG|INFO)	asset.debug.level	{Always Register}	Enum	N/A	N/A		NO_RESET	NONE	Yes	N/A
	Power Status (ON|OFF) *** FIXED POWER STATE = 'ON'	asset.power	Has-Power	Enum	N/A	N/A		NO_RESET	ASSET_POWER	Yes	N/A
	Volume Mute (ON|OFF)	volume.mute	Has-Volume	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Volume Level (0-255)	volume.level	Has-Volume	Level	N/A	volume.level		NO_RESET	NONE	No	N/A
	Gain Mute (ON|OFF)	gain.mute	Has-Gain	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Gain Level (0-255)	gain.level	Has-Gain	Level	N/A	general		NO_RESET	NONE	No	N/A
Metadata Properties	Duet Device Category ** *DUET MODULES ONLY*	duet.device.category	{Always Register}	String							
	Duet Device Channels ** *DUET MODULES ONLY*	duet.device.channels	{Always Register}	Number							
	Duet Device Levels ** *DUET MODULES ONLY*	duet.device.levels	{Always Register}	Number							
	Duet Device Revision ** *DUET MODULES ONLY*	duet.device.revision	{Always Register}	String							
	Duet Module ** *DUET MODULES ONLY*	duet.module.name	{Always Register}	String							
	Duet Module Version ** *DUET MODULES ONLY*	duet.module.version	{Always Register}	String							
	Physical Device DPS ** *DUET MODULES ONLY*	duet.physical.dps	{Always Register}	String							
	Switcher Input Count	switcher.input.count	{Always Register}	Number							
	Switcher Output Count	switcher.output.count	{Always Register}	Number							
	Switcher Preset Count	switcher.preset.count	Has-Switcher-Preset	Number							

TV

TV		Key	Register?	Data Type	Units	Bargraph Key	Default Value	Reset Value	Param Type	History Track Changes	Status Type
Control Methods	Set Power (ON|OFF)	asset.power	Has-Power	Enum			OFF				
	Set Volume Mute (ON|OFF)	volume.mute	Duet: Has-Volume XDD: Has-Volume-Mute-Cycle	Boolean			OFF				
	Set Volume Level (0-255)	volume.level	Has-Volume	Level			N/A				
	Set Aspect Ratio (Enum: *Aspect Ratio*)	display.aspect.ratio	Has-Aspect-Ratio	Enum			N/A				
	Select Input Source (Enum: *Source*)	source.input	Has-Input-Select	Number			N/A				
	Set Station (String)	tuner.station	Has-Station	String			N/A				
	Select Station Preset (1 to MAX STATION PRESETS)	tuner.station.preset	Has-Station-Preset	Number			N/A				
	Select Tuner Band (Enum: *Band*)	tuner.band	Has-Tuner-Band	Enum			N/A				
Monitored Parameters	Online (ONLINE|OFFLINE)	asset.online	{Always Register}	Enum	N/A	N/A		NO_RESET	ASSET_ONLINE	Yes	MAINTENANCE
	Power Consumption (*This parameter is registered and managed by the RMS server*)	asset.power.consumption	{Optional Registration}	Decimal	watts	N/A		NO_RESET	POWER_CONSUMPTION	Yes	N/A
	Device Data Initialized (TRUE|FALSE)	asset.data.initialized	{Always Register}	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Module Debug State (ERROR|WARNING|DEBUG|INFO)	asset.debug.level	{Always Register}	Enum	N/A	N/A		NO_RESET	NONE	Yes	N/A
	Power Status (ON|OFF)	asset.power	Has-Power	Enum	N/A	N/A		NO_RESET	ASSET_POWER	Yes	N/A
	Volume Mute (ON|OFF)	volume.mute	Has-Volume	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Volume Level (0-255)	volume.level	Has-Volume	Level	N/A	volume.level		NO_RESET	NONE	No	N/A
	Display Usage (time consumption based on PowerState)	display.usage	{Always Register}	Decimal	Hours	display.usage		0	DISPLAY_USAGE	Yes	MAINTENANCE
	Aspect Ratio (Enum: *Aspect Ratio*)	display.aspect.ratio	Has-Aspect-Ratio	Enum	N/A	N/A		NO_RESET	NONE	No	N/A
	Tuner Band (Enum: *Band*)	tuner.band	Has-Tuner-Band	Enum	N/A	N/A		NO_RESET	NONE	No	N/A
	Station (Channel)	tuner.station	Has-Station	String	N/A	N/A		NO_RESET	NONE	No	N/A
	Input Source (Enum: *Source*)	source.input	Has-Input-Select	Enum	N/A	N/A		NO_RESET	SOURCE_STATE	Yes	N/A
Metadata Properties	Duet Device Category ** *DUET MODULES ONLY*	duet.device.category	{Always Register}	String							
	Duet Device Channels ** *DUET MODULES ONLY*	duet.device.channels	{Always Register}	Number							
	Duet Device Levels ** *DUET MODULES ONLY*	duet.device.levels	{Always Register}	Number							
	Duet Device Revision ** *DUET MODULES ONLY*	duet.device.revision	{Always Register}	String							
	Duet Module ** *DUET MODULES ONLY*	duet.module.name	{Always Register}	String							
	Duet Module Version ** *DUET MODULES ONLY*	duet.module.version	{Always Register}	String							
	Physical Device DPS ** *DUET MODULES ONLY*	duet.physical.dps	{Always Register}	String							
	Input Source Count	source.input.count	Has-Input-Select	Number							
	Station Preset Count	tuner.station.count	Has-Station-Preset	Number							
	Tuner Band Count	tuner.band.count	Has-Tuner-Band	Number							
	Shutting Down Time ** *XDD MODULES ONLY*	shutting.down.time	Has-Power	Number							
	Starting Up Time ** *XDD MODULES ONLY*	starting.up.time	Has-Power	Number							

VIDEO CONFERENCER

VIDEO CONFERENCER		Key	Register?	Data Type	Units	Bargraph Key	Default Value	Reset Value	Param Type	History Track Changes	Status Type
Control Methods	Set Power (ON|OFF)	asset.power	Has-Power	Enum			OFF				
	Set Volume Mute (ON|OFF)	volume.mute	Has-Volume	Boolean			OFF				
	Set Auto Answer (ON|OFF)	dialer.auto.answer	Has-Auto-Answer	Boolean			N/A				
	Set Volume Level (0-255)	volume.level	Has-Volume	Level			N/A				
	Select Input Source (Enum: *Source*)	source.input	Has-Input-Select	Number			N/A				
	Set Privacy (ON|OFF)	conferencer.privacy	Has-Privacy	Boolean			ON				
	Set Audible Ring (ON|OFF)	dialer.ring.audible	Has-Audible-Ring	Boolean			ON				
	Dial Speed Dial Preset	dialer.dial.preset	Has-Dialer	Number			N/A				
	Dial Telephone Number (String)	dialer.dial.number	Has-Dialer	String			N/A				
	Redial	dialer.redial	Has-Dialer	No Args			N/A				
	Set Off Hook (ON|OFF)	dialer.hook	Has-Hook	Boolean			ON				
	Select Camera Preset (Number: *1 to MAX CAMERA PRESETS*)	camera.preset	Has-Camera-Preset	Number			N/A				
Monitored Parameters	Online (ONLINE|OFFLINE)	asset.online	{Always Register}	Enum	N/A	N/A		NO_RESET	ASSET_ONLINE	Yes	MAINTENANCE
	Power Consumption (*This parameter is registered and managed by the RMS server.*)	asset.power.consumption	{Optional Registration}	Decimal	watts	N/A		NO_RESET	POWER_CONSUMPTION	Yes	N/A
	Device Data Initialized (TRUE|FALSE)	asset.data.initialized	{Always Register}	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Module Debug State (ERROR|WARNING|DEBUG|INFO)	asset.debug.level	{Always Register}	Enum	N/A	N/A		NO_RESET	NONE	Yes	N/A
	Power Status (ON|OFF)	asset.power	Has-Power	Enum	N/A	N/A		NO_RESET	ASSET_POWER	Yes	N/A
	Volume Mute (ON|OFF)	volume.mute	Has-Volume	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Volume Level (0-255)	volume.level	Has-Volume	Level	N/A	volume.level		NO_RESET	NONE	No	N/A
	Input Source (Enum: *Source*)	source.input	Has-Input-Select	Enum	N/A	N/A		NO_RESET	SOURCE_STATE	Yes	N/A
	Privacy (ON|OFF)	conferencer.privacy	Has-Privacy	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Audible Ring (ON|OFF)	dialer.ring.audible	Has-Audible-Ring	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Off Hook (ON|OFF)	dialer.hook	Has-Hook	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Auto Answer (ON|OFF)	dialer.auto.answer	Has-Auto-Answer	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Dialer Status (Enum: *DialerState*)	dialer.status	Has-Dialer	Enum	N/A	N/A		NO_RESET	DIALER_STATE	No	N/A
	Last Incoming Call (String) - not getter fn, only processIncomingCall event	dialer.incoming.call	Has-Dialer	String	N/A	N/A		NO_RESET	NONE	No	N/A
	Auto Focus (ON|OFF)	camera.focus.auto	Has-Auto-Focus	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Auto Iris (ON|OFF)	camera.iris.auto	Has-Auto-Iris	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Pan Position (0-255)	camera.pan.position	Has-Pan-Tilt	Level	N/A	general		NO_RESET	NONE	No	N/A
	Tilt Position (0-255)	camera.tilt.position	Has-Pan-Tilt	Level	N/A	general		NO_RESET	NONE	No	N/A
	Focus Level (0-255)	camera.focus.level	Has-Focus	Level	N/A	general		NO_RESET	NONE	No	N/A
	Zoom Level (0-255)	camera.zoom.level	Has-Zoom	Level	N/A	general		NO_RESET	NONE	No	N/A
	Last Selected Camera Preset (Number)	camera.preset	Has-Camera-Preset	Number	N/A	N/A		NO_RESET	NONE	No	N/A
	Pan Speed (0-255)	camera.pan.speed	Has-Pan-Tilt-Speed	Level	N/A	general		NO_RESET	NONE	No	N/A
	Tilt Speed (0-255)	camera.tilt.speed	Has-Pan-Tilt-Speed	Level	N/A	general		NO_RESET	NONE	No	N/A
	Focus Speed (0-255)	camera.focus.speed	Has-Focus-Speed	Level	N/A	general		NO_RESET	NONE	No	N/A
	Zoom Speed (0-255)	camera.zoom.speed	Has-Zoom-Speed	Level	N/A	general		NO_RESET	NONE	No	N/A
Metadata Properties	Duet Device Category ** *DUET MODULES ONLY*	duet.device.category	{Always Register}	String							
	Duet Device Channels ** *DUET MODULES ONLY*	duet.device.channels	{Always Register}	Number							
	Duet Device Levels ** *DUET MODULES ONLY*	duet.device.levels	{Always Register}	Number							
	Duet Device Revision ** *DUET MODULES ONLY*	duet.device.revision	{Always Register}	String							
	Duet Module ** *DUET MODULES ONLY*	duet.module.name	{Always Register}	String							
	Duet Module Version ** *DUET MODULES ONLY*	duet.module.version	{Always Register}	String							
	Physical Device DPS ** *DUET MODULES ONLY*	duet.physical.dps	{Always Register}	String							
	Input Source Count	source.input.count	Has-Input-Select	Number							
	Phonebook Capacity	phonebook.capacity	Has-Phonebook	Number							
	Camera Preset Count	camera.preset.count	Has-Camera-Preset	Number							

VIDEO PROJECTOR

VIDEO PROJECTOR		Key	Register?	Data Type	Units	Bargraph Key	Default Value	Reset Value	Param Type	History Track Changes	Status Type
Control Methods	Set Power (ON|OFF)	projector.lamp.power	Duet: Has-Lamp XDD: Set Power	Enum			ON				
	Set Volume Mute (ON|OFF)	volume.mute	Duet: Has-Volume XDD: Has-Volume-Mute-Cycle	Boolean			OFF				
	Set Volume Level (0-255)	volume.level	Has-Volume	Level			N/A				
	Set Aspect Ratio (Enum: *Aspect Ratio*)	display.aspect.ratio	Has-Aspect-Ratio	Enum			N/A				
	Select Input Source (Enum: *Source*)	source.input	Has-Input-Select	Number			N/A				
Monitored Parameters	Online (ONLINE|OFFLINE)	asset.online	{Always Register}	Enum	N/A	N/A		NO_RESET	ASSET_ONLINE	Yes	MAINTENANCE
	Power Consumption (*This parameter is registered and managed by the RMS server.*)	asset.power.consumption	{Optional Registration}	Decimal	watts	N/A		NO_RESET	POWER_CONSUMPTION	Yes	N/A
	Device Data Initialized (TRUE|FALSE)	asset.data.initialized	{Always Register}	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Module Debug State (ERROR|WARNING|DEBUG|INFO)	asset.debug.level	{Always Register}	Enum	N/A	N/A		NO_RESET	NONE	Yes	N/A
	Power Status (ON|OFF) - *XDD MODULES* Lamp Power - *DUET MODULES*	projector.power.status projector.lamp.power	Has-Power	Enum	N/A	N/A		NO_RESET	ASSET_POWER	No	N/A
	Lamp Consumption (*Hours*) *Special Notes*: This consumption time is tallied artificially unless the Duet module returns a lamp hours value of '0' or greater while the lamp power state is 'ON'. Any time the Duet module sends a lamp time event, the artificial tracking will cease.	lamp.consumption	Has-Lamp	Number	Hours	lamp.consumption		0	LAMP_USAGE	Yes	MAINTENANCE
	Volume Mute (ON|OFF)	volume.mute	Has-Volume	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Volume Level (0-255)	volume.level	Has-Volume	Level	N/A	volume.level		NO_RESET	NONE	No	N/A
	Aspect Ratio (Enum: *Aspect Ratio*)	display.aspect.ratio	Has-Aspect-Ratio	Enum	N/A	N/A		NO_RESET	NONE	No	N/A
	Input Source (Enum: *Source*)	source.input	Has-Input-Select	Enum	N/A	N/A		NO_RESET	SOURCE_STATE	Yes	N/A
Metadata Properties	Duet Device Category ** *DUET MODULES ONLY*	duet.device.category	{Always Register}	String							
	Duet Device Channels ** *DUET MODULES ONLY*	duet.device.channels	{Always Register}	Number							
	Duet Device Levels ** *DUET MODULES ONLY*	duet.device.levels	{Always Register}	Number							
	Duet Device Revision ** *DUET MODULES ONLY*	duet.device.revision	{Always Register}	String							
	Duet Module ** *DUET MODULES ONLY*	duet.module.name	{Always Register}	String							
	Duet Module Version ** *DUET MODULES ONLY*	duet.module.version	{Always Register}	String							
	Physical Device DPS ** *DUET MODULES ONLY*	duet.physical.dps	{Always Register}	String							
	Input Source Count	source.input.count	Has-Input-Select	Number							
	Lamp Cooling Down ** *DUET MODULES ONLY*	projector.lamp.cooldown.time	Has-Lamp	Number							
	Lamp Warming Up ** *DUET MODULES ONLY*	projector.lamp.warmup.time	Has-Lamp	Number							
	Shutting Down Time (*Seconds*) ** *XDD MODULES ONLY*	projector.lamp.shuttingdown.time	Has-Power	Number							
	Starting Up Time (*Seconds*) ** *XDD MODULES ONLY*	projector.lamp.startingup.time	Has-Power	Number							

AMX TOUCH PANEL

AMX TOUCH PANEL		Key	Register?	Data Type	Units	Bargraph Key	Default Value	Reset Value	Param Type	History Track Changes	Status Type
Control Methods	Enter Setup Pages	touch.panel.setup	{Always Register}	N/A			N/A				
	Sleep	touch.panel.sleep	{Always Register}	N/A			N/A				
	Wake	touch.panel.wake	{Always Register}	N/A			N/A				
	Calibrate	touch.panel.calibrate	{Always Register}	N/A			N/A				
	Beep (force beep, double beep)	touch.panel.beep	{Always Register}	Boolean Boolean			N/A				
	Set Brightness Level (1-100) (G4 only)	touch.panel.brightness	G4 Only	Level			N/A				
	Set Volume Mute (ON|OFF) (G4 only)	touch.panel.volume.mute	G4 Only	Boolean			N/A				
	Set Volume Level (1-100) (G4 only)	touch.panel.volume.level	G3 Only	Level			N/A				
	Set Brightness Level (1-8) (G3 Only)	touch.panel.brightness	G3 Only	Level			N/A				
	Reset Panel (G3 Only)	touch.panel.reset	G3 Only	N/A			N/A				
	Shutdown (G4 Wireless Panels Only)	touch.panel.shutdown	G4 & Wireless Only	N/A			N/A				
Monitored Parameters	Online (ONLINE|OFFLINE)	asset.online	{Always Register}	Enum	N/A	N/A		NO_RESET	ASSET_ONLINE	Yes	MAINTENANCE
	Power Consumption (*This parameter is registered and managed by the RMS server.*)	asset.power.consumption	{Always Register}	Decimal	watts	N/A		NO_RESET	POWER_CONSUMPTION	Yes	N/A
	Docked (TRUE|FALSE) (G4 Docking Panels Only)	touch.panel.docked	G4 & Docking Only	Boolean	N/A	N/A		NO_RESET	DOCKING_STATE	Yes	N/A
	Wireless Channel (G4 Wireless Panels Only)	touch.panel.wireless.channel	G4 & Wireless Only	Number	N/A	N/A		NO_RESET	NONE	No	N/A
	Battery Level (G4 Wireless Panels Only)	touch.panel.battery.level	G4 & Wireless Only	Level	%	battery.level		NO_RESET	BATTERY_LEVEL	Yes	MAINTENANCE
	Battery Charging (G4 Wireless Panels Only)	touch.panel.battery.charging	G4 & Wireless Only	Boolean	N/A	N/A		NO_RESET	BATTERY_CHARGING_STATE	No	N/A
	Wireless Signal Strength (G4 Wireless Panels Only) Note: This will be updated on a periodic basis as not to flood RMS with constant changes.	touch.panel.wireless.signal.strength	G4 & Wireless Only	Level	dB	signal.strength		NO_RESET	SIGNAL_STRENGTH	No	N/A
	Volume Level (0-100) (G4 only)	touch.panel.volume.level	G4 Only	Level	%	volume.level		NO_RESET	NONE	No	N/A
	Volume Mute (ON|OFF) (G4 only)	touch.panel.volume.mute	G4 Only	Boolean	N/A	N/A		NO_RESET	NONE	No	N/A
	Brightness Level (0-100) (G4 only)	touch.panel.brightness	G4 Only	Level	%	general		NO_RESET	NONE	No	N/A
Metadata Properties	Wireless Panel (TRUE|FALSE)	touch.panel.wireless	{Always Register}	Boolean							
	Dockable Panel (TRUE|FALSE)	touch.panel.dockable	{Always Register}	Boolean							

Appendix A: RmsApi - Structures

Overview

This section provides a consolidated listing of the Structures used in the *RmsApi.axi* file:

RMS Asset Data Structure

STRUCTURE RmsAsset

RmsAsset			
Data Type	**Parameter**	**Length**	**Comment**
CHAR	assetType	50	
CHAR	clientKey	30	
CHAR	globalKey	150	
CHAR	name	50	
CHAR	description	250	
CHAR	manufacturerName	50	
CHAR	manufacturerUrl	250	
CHAR	modelName	50	
CHAR	modelUrl	250	
CHAR	serialNumber	100	
CHAR	firmwareVersion	30	

RMS Asset Metadata Property Data Structure

STRUCTURE RmsAssetMetadataProperty

RmsAssetMetadataProperty			
Data Type	**Parameter**	**Length**	**Comment**
CHAR	key	50	
CHAR	name	50	
CHAR	value	50	
CHAR	dataType	30	
CHAR	readOnly		
CHAR	hyperlinkName	50	
CHAR	hyperlinkUrl	100	

RMS Asset Control Method Data Structure

STRUCTURE RmsAssetControlMethodArgument

RmsAssetControlMethodArgument			
Data Type	**Parameter**	**Length**	**Comment**
INTEGER	ordinal		
CHAR	name	50	
CHAR	description	250	
CHAR	dataType	30	
CHAR	defaultValue	30	
SLONG	minimumValue		
SLONG	maximumValue		
INTEGER	stepValue		
CHAR	enumerationValues	15, 30	

RMS Asset Parameter Data Structure

STRUCTURE RmsAssetParameter

RmsAssetParameter			
Data Type	**Parameter**	**Length**	**Comment**
CHAR	key	50	
CHAR	name	50	
CHAR	description	250	
CHAR	dataType	30	
CHAR	reportingType	30	
CHAR	initialValue	50	
CHAR	units	50	
CHAR	allowReset		
CHAR	resetValue	50	
SLONG	minimumValue		
SLONG	maximumValue		
CHAR	enumeration	500	
CHAR	trackChanges		
CHAR	bargraphKey	30	
CHAR	stockParam		

RMS Asset Parameter Threshold Data Structure

STRUCTURE RmsAssetParameterThreshold

RmsAssetParameterThreshold			
Data Type	**Parameter**	**Length**	**Comment**
CHAR	name	50	
CHAR	enabled		
CHAR	statusType	30	
CHAR	comparisonOperator	30	
CHAR	value	50	
CHAR	notifyOnTrip		
CHAR	notifyOnRestore		
INTEGER	delayInterval		

RMS Client Gateway Data Structure

STRUCTURE RmsClientGateway

RmsClientGateway			
Data Type	**Parameter**	**Length**	**Comment**
CHAR	uid	100	
CHAR	name	100	
CHAR	hostname	100	
CHAR	ipAddress	50	
CHAR	ipPort	20	
CHAR	gateway	100	
CHAR	subnetMask	100	
CHAR	macAddress	100	
CHAR	sdkVersion	20	
INTEGER	communicationProtocol		
INTEGER	communicationProtocolVersion		

RMS Location Data Structure

STRUCTURE RmsLocation

RmsLocation			
Data Type	**Parameter**	**Length**	**Comment**
INTEGER	id		
CHAR	name	100	
CHAR	timezone	100	
INTEGER	occupancy		
CHAR	prestigeName	100	
CHAR	owner	100	
CHAR	phoneNumber	50	
CHAR	assetLicensed		

RMS Display Message Data Structure

STRUCTURE RmsDisplayMessage

RmsDisplayMessage			
Data Type	**Parameter**	**Length**	**Comment**
CHAR	type	100	
CHAR	title	250	
CHAR	message	2000	
LONG	timeout		
CHAR	isModal		
CHAR	isResponse		
LONG	locationId		

RMS Asset Control Method Data Structure

STRUCTURE RmsAssetControlMethod

RmsAssetControlMethod			
Data Type	**Parameter**	**Length**	**Comment**
CHAR	assetClientKey	50	
CHAR	methodKey	50	
CHAR	argumentValues	20, 250	Max 20 arguments

Appendix B: RmsSchedulingApi - Structures

Overview

This section provides a consolidated listing of the Structures used in the *RmsSchedulingApi.axi* file:

RMS Event Booking Response Data Structure

This structure represents two types of data:

1. A response to a NetLinx command
2. Information initiated by the server

NOTE: *In different events some fields may not be meaningful. As an example, for adhoc event creation, attendees will not be populated.*

STRUCTURE RmsEventBookingResponse

RmsEventBookingResponse		
Data Type	**Parameter**	**Comment**
CHAR		
LONG		
CHAR		
CHAR		
CHAR		
CHAR		
CHAR		
CHAR		
CHAR		
CHAR		
CHAR		
CHAR		
CHAR		
LONG		Only used for active booking event
LONG		Only used for next active booking events
LONG		Only used for active booking events
CHAR		
CHAR		Not used in some contexts such as adhoc creation
CHAR		
CHAR		Not used if result is from a successful event

RMS Event Booking Monthly Summary Data Structure

This structure represents a RMS Booking Monthly Summary

NOTE: *Daily counts associated with a monthly summary are represented as a RmsEventBookingDailyCount structure, only a total of the number of entries is provided by dailyCountsTotal*

STRUCTURE RmsEventBookingMonthlySummary

RmsEventBookingMonthlySummary		
Data Type	**Parameter**	**Comment**
LONG		
CHAR		
CHAR		
CHAR		
CHAR		
INTEGER		

RMS Event Booking Daily Count Data Structure

This structure represents a RMS Booking Daily Count

NOTE: *recordCount and recordNumber are only meaningful in the context of a monthly summary. For a monthly summary, these items provide information about the total number of records in the monthly summary and the record number of this specific entry.*

STRUCTURE RmsEventBookingDailyCount

RmsEventBookingDailyCount		
Data Type	**Parameter**	**Comment**
LONG	location	
INTEGER	dayOfMonth	
INTEGER	bookingCount	
INTEGER	recordCount	
INTEGER	recordNumber	

3000 RESEARCH DRIVE, RICHARDSON, TX 75082 AMX.com | 800.222.0193 | 469.624.8000 | +1.469.624.7400 | fax 469.624.7153

AMX (UK) LTD, AMX by HARMAN - Unit C, Auster Road, Clifton Moor, York, YO30 4GD United Kingdom • +44 1904-343-100 • www.amx.com/eu/

Last Revised:
6/12/2017