NetLinx Programmer's Guide

RMS

Resource Management Suite®

(v3.3 or higher)

Enterprise Applications

Last Revised: 4/13/2010

Table of Contents

Overview

The Resource Management Suite® products are PC server applications designed to manage rooms and equipment. The RMS server also monitors equipment in the rooms and sends notifications for room problems and help requests. The RMS server allows for the logging of room and device use, errors that occur, and offline events. The RMS server offers a variety of build-in reports for historical and statistical analysis, as well as device monitoring through a user extensible framework. This framework allows you to customize what devices should be monitored, the conditions that indicates a problem or fault, and what type of problem or fault this condition represents. The RMS server generates notifications and routes them to different personnel when a fault condition occurs, routing such notifications to the appropriate personnel as determined by the notification configuration.

The RMS Software Development Kit (SDK) is composed of a series of modules that allow users to monitor equipment errors and usage, view appointments, display welcome images and messages, and view current appointment details from any NetLinx compatible touch panel. Users can create presets to be executed when a meeting starts from the actions available through i!-ConnectLinx.

i!-ConnectLinx provides the mechanism to expose actions to the RMS server and to manage action execution on the NetLinx system. In the RMS web pages users can create control functions which are essentially macro sequences of i!-ConnectLinx actions. These control function macros can be directly executed or scheduled from the RMS web pages. i!-ConnectLinx handles these requests and presents it to the NetLinx program for execution. See the i!-ConnectLinx help file for details on programming i!-ConnectLinx.

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 2.5 (or later)
- TPDesign4 v2.6 (or later) for G4 panels

Concepts

Network Configuration

The RMS application is a client/server application where the NetLinx system acts as the client and the RMS application server listens for connections from NetLinx systems. NetLinx and the RMS application server communicate using TCP/IP sockets. In order to establish communication, each NetLinx system must be able to resolve and connect to the RMS application server. This can be accomplished with a variety of Network configurations including local area networks (LAN), wide area networks (WAN), and the Internet.

In order to communicate with RMS, a NetLinx system must have the RMS modules added to its programming. The *RMSEngineMod* module includes the core API and communication stack that allows NetLinx to communicate with the RMS server.

Since each NetLinx system acts as the client, it must be configured to communicate to the RMS server using the '**SERVER-**' command in NetLinx programming. NetLinx can accept either an IP address or a HostName for the server. NetLinx supports DNS so if you are using a HostName, the HostName must be registered with the DNS server that NetLinx has been configured to use. The DNS server configuration will be picked up automatically through DHCP if the DNS servers are registered with the DHCP server. For more information on configuring DNS servers in NetLinx, see the NetLinx master’s instruction manual.

Optionally, the server IP or host name can be placed in a file called **ServerInfo.txt** and placed in the RMS directory of the NetLinx master's file system. If this file is present, the RMS communication module ignores the SERVER- command and uses the address supplied in the file. Enter the IP address or hostname on a single line using a text editor and FTP the file to the NetLinx master. If the RMS directory does not exist, you can create it and place the file in the directory.

By default, NetLinx and the RMS server will communicate using TCP/IP port **3839**.

Port 3839 is registered to AMX Resource Management Suite with IANA (http://www.iana.org/assignments/port-numbers). This can be changed to suit your particular facility but it must be changed in both the RMS server software and each NetLinx system. In the RMS server, this is accomplished through the Configuration Wizard. In NetLinx, this is accomplished through the 'SERVER-' command in NetLinx programming.

If using the ServerInfo.txt file, append a ":" and the port number to the server IP address or host name.

MeetingManager 1.0 used port 9090 for communications. If you are upgrading from MeetingManager 1.0, you may wish to continue to use port 9090. During the upgrade process, you are prompted to change to port 3839 or continue to use port 9090. If you change to port 3839, you need to upgrade all NetLinx systems to use the modules from the RMS 2.0 SDK. You can use port 9090 with both MeetingManager 1.0 and 2.0 NetLinx systems.

Once a NetLinx system has been programmed with the RMS modules and the server's IP address or HostName, the NetLinx system automatically connects to the RMS server.

Install Checklist	
Is the RMS server's host name registered with your DNS server?	
Yes	• Configure each NetLinx system to point the correct DNS server and supply the HostName to the NetLinx programmer to use in the 'SERVER-' command. The DNS server configuration will be picked up automatically through DHCP if the DNS servers are registered with the DHCP server.
No	• Determine the IP address of the RMS server and supply this to the NetLinx programmer to use in the 'SERVER-' command.
Do you want to use 3839 as the TCP/IP port for communications between Netlinx and the RMS server?	
Yes	• No changes need to be made in either the RMS server or NetLinx.
No	• Configure the TCP/IP in the RMS server using the Configuration Wizard and supply the new port to the NetLinx programmer to use in the 'SERVER-' command.

Device Monitoring Framework

RMS provides device monitoring through a user extensible framework. This framework allows you to customize what devices are monitored, the conditions that indicate a problem or fault, and what type of problem or fault this condition represents. RMS generates notifications when a fault condition occurs, as determined by the notification configuration.

Each room has one or more monitored devices. Each device can be a physical device, such as a video projector, or a logical device, like the RMS software. However, each monitored device must be associated with a NetLinx-connected device. In the case of a video projector, this device would be the IR card, Serial Card or IP Socket used to communicate with the projector. The RMS software is associated with the NetLinx master itself.

Each monitored device has one or more device parameters that represent monitored items. For instance, monitoring lamp hours of a video projector is accomplished through a "Lamp Hours" parameter that belongs to the "Video Projector" device. All parameters must be associated with a device.

In order to monitor a device, the NetLinx system must register the device and one or more parameters with RMS. For instance, monitoring of lamp hours of the video projector is only available if the NetLinx system has added the appropriate code. In many cases, this is as simple as adding a RMS support module.

Device Values

Each monitored device has a set of values used in its description. These values are supplied when the device is registered and consist of the following:

Device Values	
• Device Number	This is the device number of the device, as defined in the NetLinx program. Devices are tracked by Device ID so this value must be unique within the devices of a given room. For instance, you can have multiple "1:1:0" devices as long as there is only one device with a Device ID of "1:1:0" in the room.
• Name	This is the name of device. This name is displayed on the administrators console and readily identifies the device.
• Manufacturer	This is the manufacturer of the device. If this value is not supplied during registration, the manufacturer of the NetLinx-connected device will be used.
• Model	This is the model number of the device. If this value is not supplied during registration, the model name of the NetLinx-connected device will be used.
• Device Type	This is the device type of the NetLinx-connected device. This might be "NI-2000" or "NXP-TPI/4 Touch Panel". This is available for Axcess and NetLinx devices. This information is registered automatically by the RMS server.
• Serial Number	This is the serial number of the NetLinx-connected Device. This is only available for NetLinx devices. This information is registered automatically by the RMS server.
• Firmware Version	This is the firmware version of the NetLinx-connected device. This is only available for NetLinx devices. This information is registered automatically be the RMS server.
• Address and Address Type	This is the physical address and address type for the Netlinx-connected device. This information describes how the device is connected to the NetLinx master. • A device connected via ICSNet will display "ICSNet" for the address type and the hardware's network address for the address. • A device connected via IP will display "TCP/IP" for the address type and the IP address for the address. • Axcess devices will display "AXLink" for both values. This information may be useful for diagnosing device connectivity problems. This information is registered automatically by the RMS server.

Parameter Values

Each parameter has a set of values used to determine what conditions indicate a problem and what type of problem this condition represents. These values are supplied when the parameter is registered and consist of the following:

Parameter Values	
• Name	This is the name of parameter. This name is displayed on the RMS server console and readily identifies the parameter. Parameters are tracked by name so this name must be unique within the parameters of a given device. For instance, you can have multiple "Lamp Hours" parameters as long as there is only one "Lamp Hours" parameter per monitored device.
• Parameter Type	This value indicates if this value is a number or a string. This information is used to determine how to perform certain operation, such as addition and comparisons between the new and threshold values. For instance, comparing "10" and "2" as strings results in "10" less than "2" but comparing them as numbers results in "2" less than "10".

Parameter Values (Cont.)	
• Value and Units	This is the current value of the parameter. Units are appended to the value when displayed in the web console.
• Threshold Value and Comparison Operator	The threshold value is the value for which this parameter is considered to indicate a problem or fault. The comparison operator is used to detect when the value changes from the un-faulted to the faulted condition. • The comparison operators "Less Than", "Less Than or Equal To", "Greater Than", "Greater Than or Equal To", "Equal To", and "Not Equal To" can be used for string and number parameters. • The comparison operators "Contains" and "Does Not Contain" are primarily used for string parameters. For example, "Lamp Hours" might have a threshold value of 1000 and any value over this would require maintenance. The comparison operator would then be "Greater Than". • When this parameter changes from a value that is not greater than 1000 to a value that is greater than 1000, the fault status is set. • When the value changes from a value greater than 1000 to a value not greater than 1000, the fault status is cleared. These value are supplied during registration but can be modified by the administrator from the RMS server console.
• Status Type	The status represents the type of problem a faulted condition represents. Status Types include "Help Request", "Maintenance Request", "Room Communication Error", "Control System Error", "Network Error", "Security", and "Equipment Usage." For example, when "Lamp Hours" changes from an un-faulted (not greater than 1000) to a faulted (greater than 1000), this change represents a "Maintenance Request" status that requires an AV technician to repair the equipment. If the "Device Online" parameter changes from "Online" to "Offline", this change could represent a "Security" or "Control System Error" status. These value are supplied during registration but can be modified by the administrator from the RMS server console.
• Reset Flag and Reset Value	These values determine if and how the parameter can be reset from the RMS server console. If the Reset Flag is set, then the administrator can reset the value remotely. When the administrator selects "Reset" from the console, the Reset Value is copied to the Value and the faulted condition is cleared. These values are useful for parameters such as VCR "Run Time" which would be manually reset when the VCR is cleaned.
• Minimum and Maximum Values	These values are used to restrict the range of the threshold and reset values that the administrator can enter on the RMS server console. These values would be used when the parameter represents a value with a bounded range, such as a Volume Level.
• Enumeration List	This value is used to restrict the range of the threshold and reset values that the administrator can enter on the RMS server console. This value would be used when the parameter represents a value with a bounded list, such as a list containing the values Power On and Power Off.

All parameters must be registered by the NetLinx system. The administrator cannot add parameters from the RMS console. The administrator can modify Threshold Value, Comparison Operator, and Status Type for any parameter. This provides the administrator with the ability to set their own thresholds and re-classify messages based on their facility.

For instance, an administrator can set the Video projector's "Lamp Hours" threshold to the expected lamp life of a newly replaced lamp or change the "Device Communicating" parameter from a "Control System Error" to a "Security" status if the projector is in danger of being stolen.

Status Types

RMS supports the following status types for device monitoring: "Help Request", "Maintenance Request", "Room Communication Error", "Control System Error", "Network Error", "Security", and "Equipment Usage."

While there are no firm rules for what these status types mean and how they are used, AMX provides the following description of each status type and recommends that your usage is consistent with these descriptions.

Status Types	
• Help Request	A user generated help request such as a help button on the touch panel.
• Maintenance Request	A user or monitored equipment generated maintenance request. Maintenance issues would include items that require a technician to visit the room.
• Room Communication Error	A loss of communication between the room and the RMS server.
• Control System Error	Any error that represents a control system error, such as an offline device or loss of communication with a device.
• Network Error	Any network related error. These would most commonly be associated with loss of communication with devices that communicate via IP.
• Security	Any security related issue. It might be appropriate to classify issues that might normally be classified as Control System or Network errors as Security issues instead. This might include a touch panel going offline or loss of communication with a projector depending on the physical security of these devices.
• Equipment Usage	Any issue that does not require repair or maintenance and that is mainly used for status.

Notification Process

As NetLinx sends parameter updates to the RMS server, the RMS server checks to see if the parameter's threshold value has been reached. This comparison is made by checking the previous value of the parameter against the threshold and by checking the new version of the parameter against the threshold using the threshold comparison operator. If the comparison for the old value is False and the comparison for the new value is True, then the parameter triggers an Alert message. If the comparison for the old value is True and the comparison for the new value is False, then the parameter triggers an Advise message. Therefore an Alert message is generated when a parameter reaches its threshold, and an Advise message is generated when a parameter returns to its normal operating range.

Alert Messages

When an Alert message occurs, the RMS server first checks to see if message should be logged to the various log services. A message is created for each log service using the Log Text of the parameter's Alert template, or the default template if a custom template has not been assigned. Next, the RMS server checks for any notifications in the Notification List matching the group, room, and status type for the parameter and dispatch any messages via SMTP or SNPP as needed using the appropriate text from the template assigned to the parameter.

Advise Messages

When an Advise message occurs, the RMS server first checks to see if the parameter is configured for sending Advise messages. If not, no messages are sent and no Log entries are created. If the parameter has been configured for Advise messages, the message is logged and dispatched via SMTP an SNPP as described above. However, the Advise template assigned to the parameter, or the default Advise template if no template has been assigned to the parameter, is used to generate the text for the log entries and messages.

For instance, if the previous value for Projector Lamp Hours is 999 and the new value is 1001 and the threshold is set to 1000 and the threshold operator is set as "Greater Than", the RMS server checks to see if the previous value compared to the threshold, i.e. 999 is Greater than 1000 is False, has a different result than the new value compared to the threshold, i.e. 1001 is Greater than 1000 is True. This change results in an Alert message being logged using the RMS logging settings. Also, a message is sent to all users registers for a notification matching the parameters group, room and status type.

If the Lamp Hours changes from 1001 to 999, the RMS server triggers an Advise message. If the parameter is configured to send Advise messages, the message is sent to the log and to all users registered for a notification matching the parameters group, room, and status type.

RFID Device Tracking

RMS supports device location tracking using radio frequency identification (RFID) technology. For more information on how RFID works with RMS, please see the RFID section in the RMS Administrator's Guide.

Implementing RFID tracking with the RMS SDK is very simple. RMS makes use of the Anterus RFID hardware and the Anterus RFID Duet Module to enable RFID functionality in RMS.

If you are using RMSCodeCrafter, then it already supports the options for RFID and prompts you for the proper input criteria. However, if you prefer to manually implement the RFID tracking code, then the steps required to include RFID tracking are as follows:

1. Download and install the Anterus Duet Module on the same computer where you are compiling the RMS NetLinx project.
2. Define the Anterus RFID reader devices in the DEFINE_DEVICE section.
3. Define and include the Anterus Duet Device module using DEFINE_MODULE
4. Define and include the *RMSRFIDTracking* NetLinx module using DEFINE_MODULE
5. Send the 'PROPERTY-Identifiers' command to the Anterus Duet module virtual device and include all the RFID readers the module should monitor. (see the Anterus documentation for more information on the 'PROPERTY-Identifiers' command.)

The Anterus RFID Duet Module monitors and manages all the physical RFID readers and RFID tags.

You can configure reader thresholds and sensitivity using the web based Anterus configuration pages hosted on the NetLinx master. The purpose of the RMSRFIDTracking NetLinx module is to coordinate RFID tag status, listen for all RFID tag changes, and communicate this information up to the RMS server.

The sample code included in the RMS SDK includes all the necessary code to implement the Anterus Duet module and RFID tracking.

In the RMSMain.AXS file, search and find the #DEFINE RMS_RFID_ENABLED statement.

If this statement is un-commented, then the sample code will compile including all the necessary RFID implementation code.

The implementation code can be found in the RMSMain.AXS and RMSMain.AXI files.

Getting Started

Overview

In order to monitor devices from an RMS system, you will need to add programming to your NetLinx project. Only the devices and parameters that you register from NetLinx can be monitored; the administrator cannot add parameters from the RMS console.

While all of the device monitoring programming can be done manually, RMS CodeCrafter can generate code for your project. From this program, you can enter the information for the device monitoring and then generate an Include (AXI) file. The Include (AXI) file contains the necessary code to register monitored devices. Once the Include file is created, you need to include this file in your main program with an #INCLUDE statement and make sure the RMS device is defined. Also, you need to add code to set the values of any custom parameters.

Using RMS CodeCrafter

To use RMSCodeCrafter, create a new RMSCodeCrafter project by opening the program from the **AMX Resource Management Suite > RMSCodeCrafter Program Folder**.

For details on operating the program, see the RMSCodeCrafter help file.

The RMS SDK consists of a series of modules to simplify device monitoring programming. Device monitoring modules handle the registration of devices and parameters, and keeping track of lamp hours and transport run time. In most cases, adding device monitoring is achieved by selecting the appropriate device monitoring module and adding code to inform the module of important device changes.

The RMS support modules register and monitor the following parameters:

Basic Device (RMSBasicDeviceMod):

Device Online/Offline, Power, Communication Status for Serial devices, Control Failure (Optional), IP Address of Socket-based devices.

Projector (RMSProjectorMod):

Device Online/Offline, Power, Lamp Hours, Communication Status for Serial devices, Control Failure (Optional), IP Address of Socket-based devices.

Transport (RMSTransportMod):

Device Online/Offline, Power, Run Time, Communication Status for Serial devices, Control Failure (Optional), IP Address of Socket-based devices.

Slide Projector (RMSSldProjMod):

Device Online/Offline, Power, Lamp Hours

RMS NetLinx Code Architecture

FIG. 1 provides a visual description of the architecture of the RMSMain.axi and the RMS support modules:

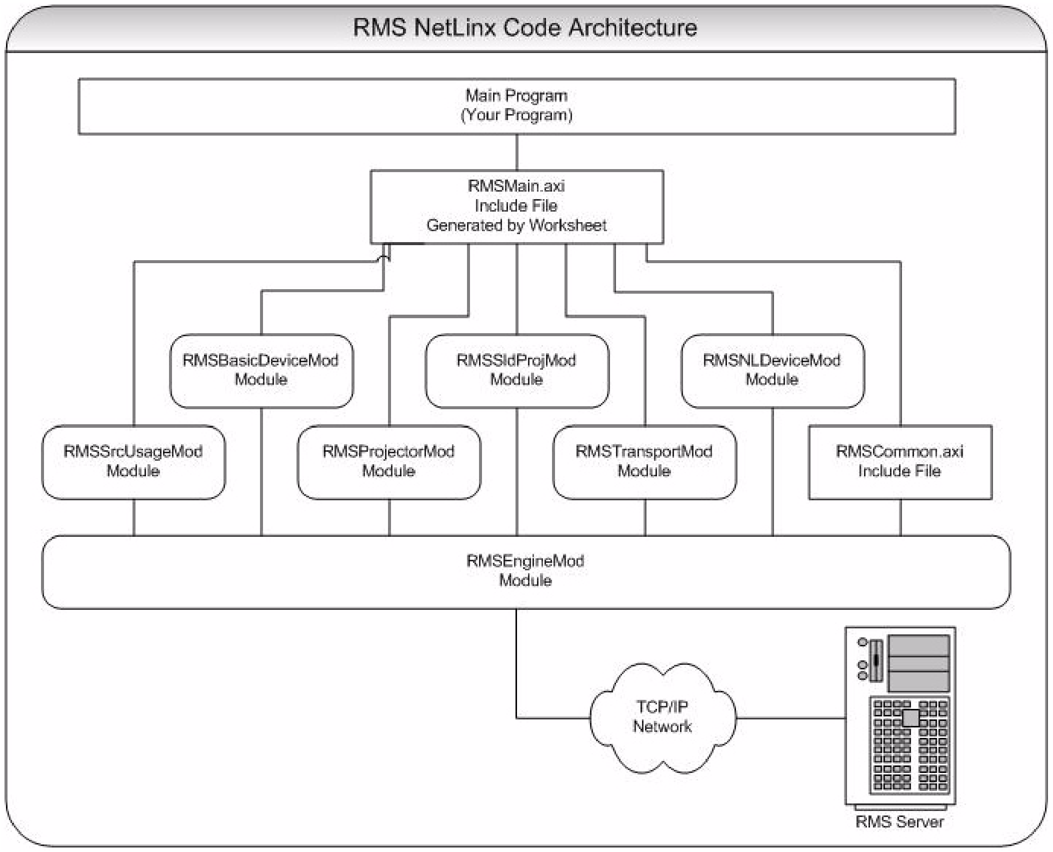

FIG. 1 Architecture of The RMSMain.axi And The RMS Support Modules

Interfacing With the RMS SDK

Once you have used RMS CodeCrafter to generate the device monitoring code for your system, you will need to communicate device status to the RMS support modules.

First, you will need to notify RMS when the system power is turned on and off.

To notify RMS when the system power is ON, call this function:

```
RMSSetSystemPower(TRUE)
```

To notify RMS when the system power is turned off, call this function:

```
RMSSetSystemPower(FALSE)
```

Next, you will need to notify RMS when device power is turned on and off.

- If you are using an AMX Comm module to communicate to your device, the RMS support modules will automatically communicate with the Comm module to determine power status.
- If you are using a power sensing device to monitor power and the power sending status appears on channel 255 of the real device, the RMS support modules will automatically detect power status.

To notify RMS when the device power is ON, call this function:

```
RMSSetDevicePower(DeviceIdentifier,TRUE)
```

Where *DeviceIdentifier* is the identifier for the real device, such as dvProj.

To notify RMS when the device power is OFF, call this function:

```
RMSSetDevicePower(DeviceIdentifier,FALSE)
```

Where *DeviceIdentifier* is the identifier for the real device, such as dvProj.

For **projectors**, you will need to notify RMS when the lamp hours changes If your projector does not support a lamp hours command, you need to make sure you notify RMS of the projector power using RMSSetDevicePower(). The RMSProjectorMod module will estimate lamp hours using projector power, if you are using an AMX Comm module to communicate to your device, the RMSProjectorMod will communicate with the Comm module to determine lamp hours automatically.

If your projector supports a lamp hours command, it is recommended you add code to poll and parse lamp hours. Once you have obtained lamp hours from your projector, notify RMS by calling this function:

```
RMSSetLampHours(DeviceIdentifier,Value)
```

Where *DeviceIdentifier* is the identifier for the real device, such as dvProj, and Value is the lamp hour's value.

For **transport devices**, you will need to notify RMS when the transport state changes. If you are using an AMX Comm module to communicate to your device, the RMSTransportMod will communicate with the Comm module to determine transport state automatically. If you are using an AMX system call with no feedback offset, i.e. FIRST is 0, RMSTransportMod will communicate with the SYSTEM_CALL to determine transport state automatically.

To notify RMS of the transport state for a custom transport implementation, use this function:

```
RMSSetTransportState(DeviceIdentifier,State)
```

Where *DeviceIdentifier* is the identifier for the real device, such as dvVCR, and State is the transport state: Play=1, stop-2, Pause=3, FFwd=4, Rew=5, SrchFwd=6, SrchRev=7, Record=8.

For **serial devices**, including, RS232 and IP controlled devices, you need to add some polling commands to these device in order to allow the RMS support module to properly report the Device Communicating parameter. The RMSBasicDeviceMod, RMSProjectorMod and RMSTransportMod expect to receive a string from the device every 30 seconds. If a string is not received within the timeout period, a loss of device communication is reported to the RMS server.

The default value for the Device Communicating timeout is 30 seconds. If this value works for your device, all you need to do is add the polling for the device.

If you want to change the timeout, set the Device Communicating timeout for a monitored device, which will in turn call RMSSetCommunicationTimeout() to change the default timeout.

- The timeout time is specified in 1/10 seconds.
- If you want to disable the Device Communicating parameter, set the timeout to 0.

Service Mode

RMS supports a Service mode where no errors will be reported from monitored devices.

Service mode is designed to allow a technician to work on a room without causing error reports. For instance, if a projector needs to be replaced or serviced, RMS would report *Device Not Communicating* when the technician disconnected the power cable or communication cable. To prevent this error from being reported to RMS, put RMS in service mode using the 'SERVICE-ON' command. When the work is completed, exit service mode using the 'SERVICE-OFF' command.

NOTE

Service Mode does not prevent notifications from being sent if disconnection occurs from the NetLinx Master.

Since service mode bypasses error reporting, it represents a security problem. For instance, in service mode no error is reported when the projector stops communicating even if it is being disconnected by unauthorized personnel. Therefore, service mode does not appear as a button on a touch panel.

Service mode should be implemented in an appropriate method for the facility. These methods may include:

- A button on a protected touch panel page
- A button on a protected web-based touch panel page
- A switched connected to an IO port on a NetLinx system accessible only by technicians.
- A key-activated switch connected to an IO port on a NetLinx system.

Device Parameter Persistence

Monitored devices and parameters are registered with RMS the first time a NetLinx system connects to the RMS server. These devices and parameters are stored internally in RMS.

When NetLinx connects and sends device and parameter registration, any devices and parameters that already exists in RMS are not overwritten. This allows the administrator to change a value, such as the lamp life threshold of a projector, and the value will not be lost, even if the NetLinx systems disconnects and reconnects. As a result, changes to device monitoring NetLinx code will not take affect if the changes are made to devices or parameters that already exists in RMS.

For instance, if you change the threshold value for a parameter or delete a device or parameter and reload the NetLinx code, the new threshold will not be used and any deleted device or parameters will still appear in RMS.

To clear out all monitored devices and parameters, delete the room and then add the room back.

- Deleting a room from RMS deletes all associated monitored devices and parameters from the RMS server.
- Optionally, you can delete a device or a parameter from the RMS console provided the device is not the "System" device and the parameter is not one of its parameters.

Custom Device Monitoring Programming

Overview

The RMS SDK is made up of a series of modules and include files. See FIG. 1 on page 8 for a visual description of the architecture of the RMSMain.axi and the RMS support modules.

RMSCommon.axi

RMSCommon.axi is an included file designed to help perform many device monitoring tasks. This file provides device-monitoring constants, functions that generate device monitoring SEND_COMMANDs to RMS, as well as providing a "callback" function for important device monitoring RMS events.

In order to use this include file, your program will need to define the RMS device and a couple of functions. The include file sends commands to and creates an event for the RMS device, vdvRMSEngine. You must create this device in your program. In your code, the device definition needs to be defined as:

```
DEFINE_DEVICE
vdvRMSEngine = 33001:1:0
```

The virtual device number needs to be unique and not conflict with any other virtual device defined in your program.

RMS will notify your program when it is time to register devices and parameters and when the administrator resets a parameter from the RMS console. RMS sends these events to your program as a string from vdvRMSEngine. The event processing section in this include file will process these strings, parse the parameters and call a function in your program to notify you of the event.

These functions need to be defined in your program whether you use them or not, otherwise the compiler will generate an error since it cannot find these functions.

The two functions you need to include are:

RMSDevMonRegisterCallback()

This function is called when RMS engine module connects to the RMS server. Since the RMS engine module does not store any information about monitored devices and their parameters, this information must be sent to the RMS only when the module is connected to the server. If you want to add any custom device monitoring code, you can register your device and parameters in this function.

In your code, the function needs to be defined as:

```
DEFINE_FUNCTION RMSDevMonRegisterCallback()
{
}
```

RMSDevMonSetParamCallback()

The function is called when the RMS administrator chooses "Reset" for a parameter that can be reset on the RMS console. You can determine which parameter was reset by checking the value of dvDPS and cName. All parameters values are sent as a string so you will need to convert it appropriately.

In your code, the function needs to be defined as:

```
DEFINE_FUNCTION RMSDevMonSetParamCallback(DEV dvDPS, CHAR cName[], CHAR cValue[])
{
}
```

RMS Engine Module

The RMS Engine Module automatically registers the "System" device that is associated with the NetLinx Master, Device ID 0:1:0.

RMS automatically registers four parameters for this device: *System Power*, *Help Request*, *Maintenance Request*, and *Service Mode*.

In addition, the system will monitor room communication status for each room. These parameters require no programming on your part for registration. However, you will need to add support for system power. RMS registers and manages this parameter for you but you need to notify RMS when the system power is turned On or Off. You can do this in one of four ways:

Turn System On:

```
(1) RMSSetSystemPower(TRUE)
(2) SEND_STRING vdvRMSEngine,'POWER=1'
(3) PULSE[vdvRMSEngine,27]
(4) ON[vdvCLActions,1001]
```

Turn System Off:

```
(1) RMSSetSystemPower(FALSE)
(2) SEND_STRING vdvRMSEngine,'POWER=0'
(3) PULSE[vdvRMSEngine,28]
(4) ON[vdvCLActions,1002]
```

The last way to inform RMS utilizes the *i!-ConnectLinx* device.

If you add the programming to your system to allow i-ConnectLinx to control power using the standard power channels and provide feedback to i!-ConnectLinx for system power, this information will automatically be read by RMS.

See the RMS Engine Module definition for details about the module and its parameters.

Example:

```
//This registers the dvRelay device under the name "Rack Power" - called in the online event
for dvRelay RMSRegisterDevice(dvRELAY,'Rack Power','AMX','NI-3000 Relay')//

//This sets the parameters for the registered device - called in the online event for dvRelay//

  RMSRegisterDeviceIndexParam(dvRELAY,'Rack Power',
      1,RMS_COMP_LESS_THAN,RMS_STAT_MAINTENANCE,
      FALSE,0,
      RMS_PARAM_SET,nRMSRackPowerRackPower,
      'OFF|ON')

//This function is called from CHANNEL_EVENT [dvRELAY,0] (Relay on or off)//
  DEFINE_FUNCTION RMSSetRackPowerRackPower(INTEGER nValue)
  LOCAL_VAR
  CHAR bInit
  {
    IF (nRMSRackPowerRackPower <> nValue || bInit = FALSE)
      RMSChangeIndexParam(dvRELAY,'Rack Power',nValue)
    nRMSRackPowerRackPower = nValue
    bInit = TRUE
  }
  DATA_EVENT [dvRELAY]
  {
    ONLINE:
    {
    RMSRegisterDevice(dvRELAY,'Rack Power','AMX','NI-3000 Relay')
    RMSRegisterDeviceIndexParam(dvRELAY,'Rack Power',
        1,RMS_COMP_LESS_THAN,RMS_STAT_MAINTENANCE,
        FALSE,0,
        RMS_PARAM_SET,nRMSRackPowerRackPower,
        'OFF|ON')
    }
    OFFLINE:
      RMSNetLinxDeviceOffline(dvRELAY)
  }
```

```
CHANNEL_EVENT [dvRELAY,0]
{
// Channel On
  ON:
  {
    SWITCH (CHANNEL.CHANNEL)
    {
      CASE 1:
        RMSSetRackPowerRackPower(1)
        break
    }
  }
// Channel Off
  OFF:
  {
    SWITCH (CHANNEL.CHANNEL)
    {
    CASE 1:
      RMSSetRackPowerRackPower(0)
      break
    }
  }
}
```

RMS Device Monitoring Support Modules

Next, you will want to consider adding RMS device monitoring support modules for monitoring basic devices. Adding these support modules will handle most of the monitoring requirements for these devices. RMS offers the following support modules:

RMSBasicDeviceMod

This module monitors basic devices. For each device, this module will register and monitor online/offline status, communication status, control failure, and power.

- Communication status is registered only if the device is a two-way device. This includes serial devices and IP sockets.
- Control failure is registered only if enabled via a SEND_COMMAND, and is based on the ability to control power.
- If the device is an IP-based device, the IP address NetLinx is communicating with is also registered with RMS.

FIG. 2 on page 13 provides a visual description of the architecture of the **RMSBasicDeviceMod** module:

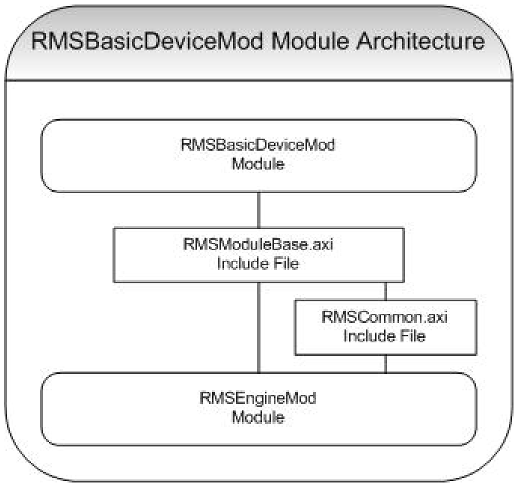
RMSBasicDeviceMod Module Architecture
RMSBasicDeviceMod Module
RMSModuleBase.axi Include File
RMSCommon.axi Include File
RMSEngineMod Module

FIG. 2 Architecture of The RMSProjectorMod Module

RMSProjectorMod

This module monitors projectors. For each projector, this module will register and monitor online/offline status, communication status, control failure, power, and lamp hours.

- Communication status is registered only if the device is a two-way device. This includes serial devices and IP sockets.
- Control failure is registered only if enabled via a SEND_COMMAND, and is based on the ability to control power.
- If the device is an IP-based device, the IP address NetLinx is communicating with is also registered with RMS.
- Lamp hours are determined by counting the time that the device's power is On. However, this module can also accept the value of lamp hours as a SEND_COMMAND when you have a projector that can provide lamp hours. Since this module registers lamp hours, it is recommended only for use with devices that have lamps that need to be replaced.

FIG. 3 provides a visual description of the architecture of the **RMSProjectorMod** module:

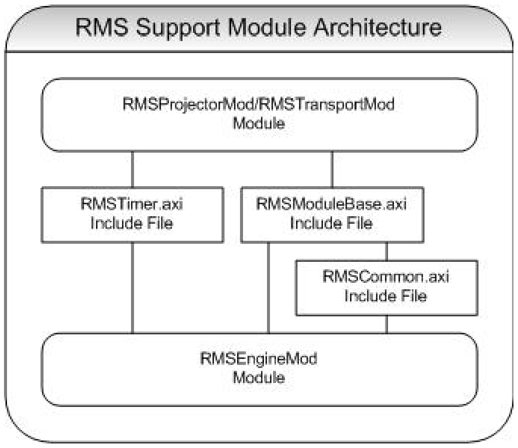
RMS Support Module Architecture
RMSProjectorMod/RMSTransportMod Module
RMSTimer.axi Include File
RMSModuleBase.axi Include File
RMSCommon.axi Include File
RMSEngineMod Module

FIG. 3 Architecture of The RMSTransportMod Module

RMSTransportMod

This module monitors transport devices. For each transport device, this module will register and monitor online/offline status, communication status, power, and run time.

- Communication status is registered only if the device is a two-way device. This includes serial devices and IP sockets.
- Control failure is register only if enabled via a SEND_COMMAND, and is based on the ability to control power.
- If the device is an IP-based device, the IP address NetLinx is communicating with is also registered with RMS.
- Run time is determined by counting the time that the device is in a transport state other than stop while the power is on.

FIG. 4 provides a visual description of the architecture of the **RMSTransportMod** module:

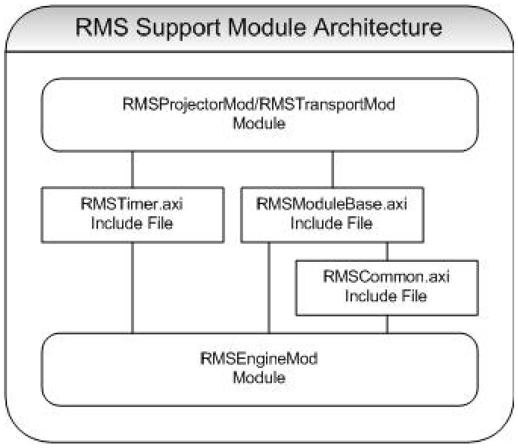
RMS Support Module Architecture
RMSProjectorMod/RMSTransportMod Module
RMSTimer.axi Include File
RMSModuleBase.axi Include File
RMSCommon.axi Include File
RMSEngineMod Module

FIG. 4 Architecture of The RMSSldProjMod Module

RMSSldProjMod

This module monitors slide projectors. For each projector, this module will register and monitor online/offline status, power, and lamp hours.

Lamp hours are determined by counting the time that the device's power is On.

FIG. 5 provides a visual description of the architecture of the **RMSSldProjMod** module:

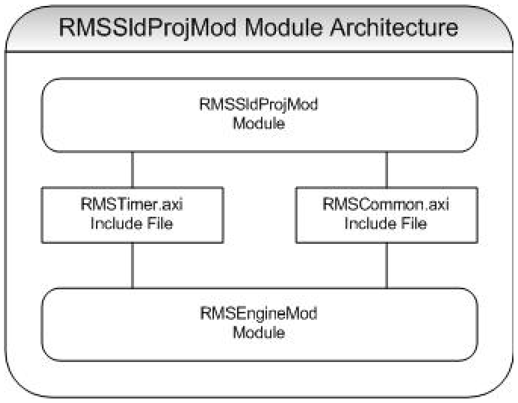
RMSSldProjMod Module Architecture
RMSSldProjMod Module
RMSTimer.axi Include File
RMSCommon.axi Include File
RMSEngineMod Module

FIG. 5 Architecture of The RMSBasicDeviceMod Module

Programming

These modules require a virtual device, the real device of the device to be monitored, and the RMS Engine module's device. If you are using an AMX module for communicating with a device, the virtual device used for the Comm module can be passed to the device monitoring support module.

Since the support modules are written to listen for the messages for the particular device types they support, no additional programming is needed. Simply define the devices, add the module, and pass the device numbers as module parameters.

Example:

```
DEFINE_DEVICE
dvSlide = 96:1:0
dvVPROJ = 5001:1:0
dvVCR = 5001:2:0
dvSWT = 5001:3:0
vdvVPROJ = 33001:1:0
vdvVCR = 33002:1:0
vdvSWT = 33003:1:0
vdvRMSEngine = 33003:1:0
```

```
// Projector Monitoring Code
DEFINE_MODULE 'RMSProjectorMod' mdlProj1(vdvVPROJ,
                                         dvVPROJ,
                                         vdvRMSEngine)
DEFINE_MODULE 'COMM_XXXXX' COMM(dvVPROJ, vdvVPROJ)

// VCR Monitoring Code
DEFINE_MODULE RMSTransportMod' mdlVCR1 (vdvVCR,
                                        dvVCR,
                                        vdvRMSEngine)
DEFINE_MODULE 'COMM_XXXXX' COMM(dvVCR, vdvVCR)
```

If you are not using an AMX module for communicating with a device, you will need to add programming to notify the module of changes in the device state.

For the Basic Device and Projector module, you will need to notify the module when the power is turned On or Off. Optionally, if you have polled for projector lamp hours, you can provide this value directly.

For the transport module, you will need to notify the module when the power is turned On or Off and when the transport state changes.

Notify Modules	

• Turn Power On:	`RMSSetDevicePower(dvProj,TRUE)`
	`SEND_STRING vdvVPROJ,'POWER=1'`
	`PULSE[vdvVPROJ,27]`
	`ON[vdvVPROJ,255]`

• Turn Power Off:	`RMSSetDevicePower(dvProj,FALSE)`
	`SEND_STRING vdvVPROJ,'POWER=0'`
	`PULSE[vdvVPROJ,28]`
	`OFF[vdvVPROJ,255]`

• Set Lamp Hours	`RMSSetLampHours(dvProj,Value)`
	`SEND_STRING vdvVProj,'LAMPTIME=Value'`

• Set Transport State (1=Play, 2=Stop, etc...):	`RMSSetTransportState(dvVCR,State)`
	`SEND_STRING vdvVCR, 'TRANSPORT=State'`
	`PULSE[vdvVCR,State+240]`

Transport States	
1	Play
2	Stop
3	Pause
4	Fast Forward
5	Rewind
6	Search forward
7	Search Reverse
8	Record

Note that for power state, you can PULSE channel 27 or 28 to set the state.

Since most IR files store the power functions on these channels, no additional programming is needed to send power state to the module when using these channels to control power.

Also, power status is monitored on channel 255 which is often linked to a power sensing device connected to an IO device. If you are using an IO device to monitor power, the IO status should be set to report on channel 255 of the real device.

In some cases, this requires a 'SET IO LINK' command to be sent to the real device. In these cases, simply pass the real device as both the virtual and real device of the support module. However, in this case, you cannot use SEND_STRING for notifying the module of transport state.

Note that for transport state, you can pulse a channel between 241-248 to set the transport state.

Since AMX SYSTEM_CALLs use those channels to store transport state, no additional programming is needed to send transport state to the module when using a SYSTEM_CALL.

In this case, simply pass the real device as both the virtual and real device of RMSTransportMod. However, in this case, you cannot use SEND_STRING for notifying the module of transport state.

Control Failure

When the device is IR, power status is monitored using channel **255**.

Axcent3's, NXC_IRS4 cards, NXI's and NI series controllers can all provide an IO link that enables an IO status to appear on channel 255 of the device.

These modules will watch for power attempts using channel 9, 27 or 28 and report a control failure if the power of the device does not respond properly. Additionally, the module will monitor channel 254, used as a power fail channel when using the 'PON' commands, and report control failure conditions when this channel is on.

This functionality must be enabled via the RMSEnablePowerFailure() function, defined in the RMSCommon.axi include file.

Example:

```
DATA_EVENT[vdvRMSEngine]
{
  ONLINE:
  {
    RMSEnablePowerFailure(dvProj)
  }
}
```

Device Information

You can define the name, manufacturer, and model using **RMSSetDeviceInfo().**

Device information is usually sent in a device registration message and can only be sent when the RMS engine module connects to the RMS server. However, if the device is monitored by a support module, the device info message can be sent at any time.

Example:

```
DATA_EVENT[vdvRMSEngine]
{
  ONLINE:
  {
    RMSSetDeviceInfo(dvProj,'Name','Manufacturer','Model Number')
    RMSSetDeviceInfo(dvVCR,'Name','Manufacturer','Model Number')
  }
}
```

The *RMSSetDeviceInfo()* is defined in the RMSCommon.axi include file.

Monitoring Source Usage

RMS can monitor source usage by using the RMSSrcUsageMod module. RMSSrcUsageMod will track the amount of time, in minutes, a given source is selected and logs this information to RMS when a new source is selected. This information can be used to generate reports to determine the actual usage of a device in a room.

Source Select

RMSSrcUsageMod monitors the selected source through i!-ConnectLinx.

i!-ConnectLinx includes 20 source selects in the standard function list. Any standard source selected registered with i!-ConnectLinx will automatically register in RMS by RMSSrcUsageMod.

As your programming sets the selected source on the i!-ConnectLinx device, RMSSrcUsageMod will track the usage of the source and report it to RMS.

To enable usage monitoring of a standard i!-ConnectLinx source, simply register the source with i!-ConnectLinx and add programming for the source select as if you were programming a button from a touch panel.

Example:

```
DEFINE_EVENT
BUTTON_EVENT[vdvCLActions,1011] // VCR Select
{
  PUSH:
  {
           // Switch the projector and switcher to select the VCR
     PULSE[vdvCLActions,1011]
  }
}
DATA_EVENT[vdvCLActions]
{
  ONLINE:
  {
         // VCR Select
    SEND_COMMAND vdvCLActions,"'ADD STD-1011'"
}
```

Additionally, you can add custom source to i!-ConnectLinx as custom actions.

Any custom action registered with i!-ConnectLinx that is named "**Select …**" will be registered as a custom source.

For instance, a custom action called "Select Slide To Video" will register a source called "Slide To Video."

Example:

```
DEFINE_EVENT
BUTTON_EVENT[vdvCLActions,1]// Custom Source
{
  PUSH:
  {
         // Switch the projector & switcher to select the Source
     PULSE[vdvCLActions, 1]
  }
}

DATA_EVENT[vdvCLActions]
{
  ONLINE:
  {
         // VCR Select
    SEND_COMMAND vdvCLActions,"'ADD ACTION-1,Select Custom Source'"
}
```

To notify i!-ConnectLinx and RMSSrcUsageMod when no source is active, set the i!-ConnectLinx status for Power Off using the standard Power Off action:

```
PULSE[vdvCLActions, 1002]
```

By default, RMS monitors a single source at a time. If a new source is selected, the previous selected source's usage is tracked and the new source is selected. However, if you have more that one destination in your system, such as two projectors, this operation is not desirable.

RMS can monitor each source independently based on the status of the source select channel. To enable this mode in RMSSrcUsageMod, call RMSSetMultiSource() with a parameter of true.

Example:

```
DATA_EVENT[vdvRMSEngine]
{
  ONLINE:
  {
    RMSSetMultiSource(TRUE)
  }
}
```

FIG. 6 provides a visual description of the architecture of the **RMSSrcUsageMod** module:

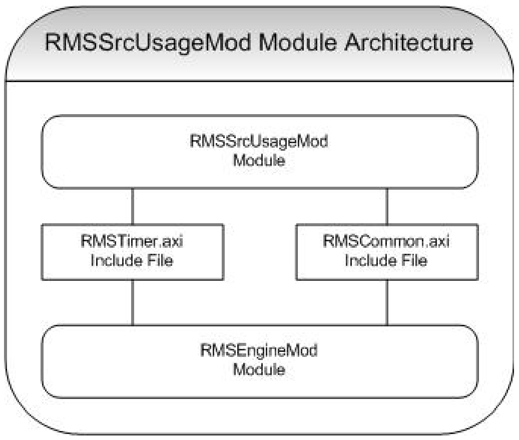
RMSSrcUsageMod Module Architecture
RMSSrcUsageMod Module
RMSTimer.axi Include File
RMSCommon.axi Include File
RMSEngineMod Module

FIG. 6 Architecture of The RMSSrcUsageMod Module

Monitoring Many NetLinx-Connected Devices

RMSNLDeviceMod

This module monitors one or more NetLinx-connected devices. For each device, the module will register and monitor the online/offline status.

This module provides a very simple way to monitor NetLinx-connected devices. However, it does not allow the naming of these devices. All devices registered with this module will display their device definition for their name, for example 128:1:0, and the manufacturer and model will be determined by the device.

This module is most useful for monitoring a large quantity of NetLinx devices where the logical name of the device is not important, such as a bank of Input or Relay cards.

To use this module, create a device array with the NetLinx connected devices you want monitored. Then pass this device array to the module:

```
DEFINE_DEVICE
dvDev1 = 5002:1:0
dvDev2 = 5002:2:0
dvDev3 = 5002:3:0
vdvRMSEngine = 33001:1:0
```

```
DEFINE_VARIABLE

// RMS NetLinx Device to Monitor
VOLATILE DEV dvMonitoredDevices[]= {dvDev1, dvDev2, dvDev3}
DEFINE_MODULE 'RMSNLDeviceMod' mdlNLD(dvMonitoredDevices, vdvRMSEngine)
```

FIG. 7 provides a visual description of the architecture of the **RMSNLDeviceMod** module:

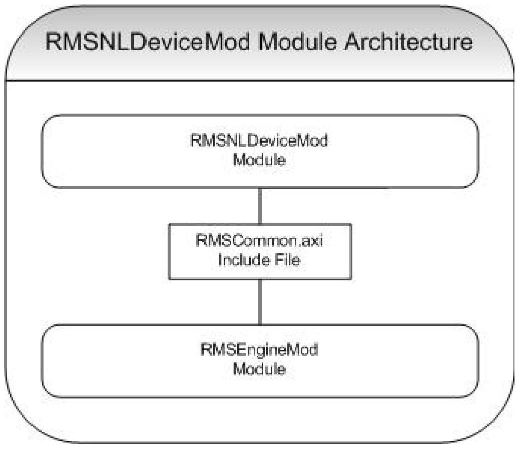
RMSNLDeviceMod Module Architecture
RMSNLDeviceMod Module
RMSCommon.axi Include File
RMSEngineMod Module

FIG. 7 Architecture of The RMSNLDeviceMod Module

Monitoring A Single NetLinx-Connected Device

The RMSCommon.axi include file provides two functions that help to monitor the Online/Offline status of a NetLinx connected device. You can use these functions to monitor a device like a touch panel or bus box. These two functions are:

- **RMSNetLinxDeviceOnline(dvDPS, cName)**
- **RMSNetLinxDeviceOffline(dvDPS)**

RMSNetLinxDeviceOnline() will register the device and the online/offline parameter as well as set the parameter to online. This function will need to be called in two places:

- Call *RMSNetLinxDeviceOnline()* in the RMSDevMonRegisterCallback() function to make sure it is registered when the RMS engine module connects to the RMS server.
- Also, call *RMSNetLinxDeviceOnline()* when the NetLinx-connected device reports online.

RMSNetLinxDeviceOffline() updates the online/offline parameter to offline. It only needs to be called when the NetLinx-connected device reports offline.

Example:

```
DEFINE_DEVICE
dvTP          = 10000:1:0
DEFINE_FUNCTION RMSDevMonRegisterCallback()
{
  RMSNetLinxDeviceOnline(dvTP,'Touch Panel 1')
}
DEFINE_EVENT
DATA_EVENT[dvTP]
{
  ONLINE:
    RMSNetLinxDeviceOnline(dvTP,'Touch Panel 1')
  OFFLINE:
    RMSNetLinxDeviceOffline(dvTP)
}
```

Registering Devices

The *RMSCommon.axi* include file provides some simple functions for registering devices. The functions can be used in the *RMSDevMonRegisterCallback()* function, called when RMS engine module connects to the RMS server.

These functions generate SEND_COMMANDs, which you can generate manually. However, using these functions may help eliminate syntax issues.

To register a device, call this function:

```
RMSRegisterDevice(dvDPS, cName, cManufacturer, cModel)
```

This function needs to be called in two places:

- Call **RMSRegisterDevice ()** in the *RMSDevMonRegisterCallback()* function to make sure it is registered when the RMS engine module connects to the RMS server.
- Also, call **RMSRegisterDevice ()** when the NetLinx-connected device reports online. This function will automatically register the Online/Offline parameter and set this value to Online.

The **RMSRegisterDevice()** function and the corresponding RMS SEND_COMMAND that it generates will only work for devices that are currently online. This is because RMS tracks information such as firmware version and serial number that are only available when the device is online.

Registering Parameters

Before registering a parameter, the device with which the parameter is associated must have been previously registered. However, if a support module RMS has registered the device already, you do not need to re-register it. For instance, you may want to add a parameter to the "System" device, 0:1:0. In this case, simply register the parameter for device 0:1:0.

The combination of Number and String parameters types and enumeration lists provide four unique kinds of parameters to the NetLinx program. These are:

Registering Parameters	
• Number Parameter	Number parameters are parameters of type number with no enumeration list. These are commonly used for values that are programmatically available and displayed in numeric form. Examples of number parameters are projector lamp hours and VCR run time.
• String Parameter	String parameters are parameters of type string with no enumeration list. These are commonly used for values that are programmatically available and displayed in text form. Examples of string parameters are help or maintenance request.
• Enum Parameter	Enum parameters are parameters of type string with an enumeration list. These are commonly used for values that are programmatically available and displayed in text form where the text is expected to be limited to a list. The value NetLinx sends for an enumeration parameter needs to exist in the enumeration list. However, the administrator will only be allowed to pick a threshold or reset value from the enumeration list. An example of an enum parameter is the currently selected source. The "|" character is used to separate values in the enumeration list.
• Index Parameter	Index parameters are parameters of type number with an enumeration list and are similar to the Enum parameter. However, these are commonly used for values that are programmatically available numerically but displayed in text form where the text is expected to be limited to a list. The value NetLinx sends for an enumeration parameter must exist in the enumeration list. However, the value sent from NetLinx represents the index into the enumerated list instead of the actual value. An example of Enum parameters is power. The value for power is often available programmatically as a zero or a one but should be displayed as "Off" or "On." This is accomplished by sending a value of zero or one to RMS and providing an enumeration list of "Off|On" where the "|" character is used to separate values in the enumeration list.

The include file provides four functions for registering these parameters:

```
RMSRegisterDeviceNumberParam(dvDPS,
                             cName,
                             slThreshold,
                             nThresholdCompare,
                             nThresholdStatus,
                             bCanReset,
                             slResetValue,
                             nInitialOp,
                             slInitial,
                             slMin,
                             slMax)

RMSRegisterDeviceIndexParam  dvDPS,
                             cName,
                             nThreshold,
                             nThresholdCompare,
                             nThresholdStatus,
                             bCanReset,
                             nResetValue,
                             nInitialOp,
                             nInitial,
                             cEnumList)

RMSRegisterDeviceStringParam dvDPS,
                             cName,
                             cThreshold,
                             nThresholdCompare,
                             nThresholdStatus,
                             bCanReset,
                             cResetValue,
                             nInitialOp,
                             cInitial)

RMSRegisterDeviceEnumParam   dvDPS,
                             cName,
                             cThreshold,
                             nThresholdCompare,
                             nThresholdStatus,
                             bCanReset,
                             cResetValue,
                             nInitialOp,
                             cInitial,
                             cEnumList)
```

Optionally, you can register a parameter with a *Unit* field. The Unit field will be displayed next to the parameter value and threshold. Consider adding a "V" if you or monitoring voltage or a "%" if you are monitoring percentage.

Two additional registration functions allow for the units field and can be used in place of the above functions:

```
RMSRegisterDeviceNumberParamWithUnits(dvDPS,
                                       cName,
                                       cUnits,
                                       slThreshold,
                                       nThresholdCompare,
                                       nThresholdStatus,
                                       bCanReset,
                                       slResetValue,
                                       nInitialOp,
                                       slInitial,
                                       slMin,
                                       slMax)

RMSRegisterDeviceStringParamWithUnits(dvDPS,
                                       cName,
                                       cUnits,
                                       cThreshold,
                                       nThresholdCompare,
                                       nThresholdStatus,
                                       bCanReset,
                                       cResetValue,
                                       nInitialOp,
                                       cInitial)
```

This function needs to be called in two places:

- Call **RMSRegisterDevicexxxParam ()** in the *RMSDevMonRegisterCallback()* function to make sure it is registered when the RMS engine module connects to the RMS server.
- Also, call **RMSRegisterDevicexxxParam ()** when the NetLinx-connected device reports online.

Parameters

- **dvDPS** is the device number of the device this parameter is associated with,
- **cName** is the name of the parameter to register.
- **nThresholdCompare** can be any of the following values:

  ```
  RMS_COMP_NONE (0),
  RMS_COMP_LESS_THAN (1),
  RMS_COMP_LESS_THAN_EQ_TO (2),
  RMS_COMP_GREATER_THAN (3),
  RMS_COMP_GREATER_THAN_EQ_TO (4),
  RMS_COMP_EQUAL_TO (5),
  RMS_COMP_NOT_EQUAL_TO (6),
  RMS_COMP_CONTAINS (7),
  RMS_COMP_DOES_NOT_CONTAIN (8).
  ```

  This value, along with **slThreshold**, **nThreshold**, or **cThreshold** is used to test to see when the parameter indicates a fault, as determined by the threshold comparison changing from false to true (i.e. Value > 10).

- The **nThresholdStatus** can be any of the following:

  ```
  RMS_STAT_NOT_ASSIGNED (1),
  RMS_STAT_HELP_REQUEST (2),
  RMS_STAT_ROOM_COMM_ERR (3),
  RMS_STAT_CONTROLSYSTEM_ERR (4),
  RMS_STAT_MAINTENANCE (5),
  RMS_STAT_EQUIPMENT_USAGE (6),
  RMS_STAT_NETWORK_ERR (7),
  RMS_STAT_SECURITY_ERR (8).
  ```

  One of these values classifies a fault with this parameter as a Help Request, Room Communication Error, Control System Error, Maintenance, Equipment usage, Network Error, or Security issue.
- **bReset** and **slResetValue**, **nResetValue** or **cResetValue** are used to allow the administrator to manually reset the value.

  If **bReset** if *False*, then **slResetValue**, **nResetValue** and **cResetValue** are ignored.
- **nInitialOp** and **slInitial**, **nInitial** and **cInitial** are used to set the value of the parameter at the time it is registered.

  nInitialOp can be one any of the following values:

  ```
  RMS_PARAM_SET (0),
  RMS_PARAM_INC (1),
  RMS_PARAM_DEC (2),
  RMS_PARAM_MULTIPLY (3),
  RMS_PARAM_DIVIDE (4),
  RMS_PARAM_UNCHANGED (6).
  ```

  This eliminates the need to send a separate set parameter messages after the parameter is registered. These constants allow you to control whether the supplied value is set, added to, subtracted from, multiply with, divided by the number in the database or if you simply want the value in the database to remain the same.
- **slInitial**, **nInitial** and **cInitial** are the value with which the operation will be performed.
- **slMin**, **slMax** and **eEnumList** are used to limit the administrator's selection of the threshold and reset values.
  - The "|" character is used to separate values in the enumeration list.
  - For *Number* parameters, **slMin** and **slMax** define the range the value is expected to have.
  - For *Index* and *Enum*, **cEnumList** contains the allowed values of the parameter.

Setting Parameter Values

You can set a parameter value any time after the RMS engine module has connected to the RMS server. Before setting the value of a parameter, the parameter must be registered.

When registering parameters, you can supply the initial value. Therefore, you will not need to set the parameter explicitly when it is registered, only subsequent changes.

The include file provides four functions for setting parameter values. They are:

```
RMSChangeNumberParam(dvDPS, cName, nOp, slValue)
RMSChangeIndexParam(dvDPS, cName, nValue)
RMSChangeStringParam(dvDPS, cName, nOp, cValue)
RMSChangeEnumParam(dvDPS, cName, cValue)
```

- **dvDPS** is the device number of the device with which this parameter is associated.
- **cName** is the name of the parameter to set.
- **nOp** can be one of the following values:

  ```
  RMS_PARAM_SET (0),
  RMS_PARAM_INC (1),
  RMS_PARAM_DEC (2),
  RMS_PARAM_MULTIPLY (3),
  RMS_PARAM_DIVIDE (4),
  RMS_PARAM_RESET (5).
  ```

These constants allow you to control whether the supplied value is set, added to, subtracted from, multiply with, divided by the number in the database or if you simply want the value in the database reset to the values supplied during parameter registration.

- **slValue**, **nValue** and **cValue** are the current value with which the operation will be performed.

  Note that *Index* and *Enum* parameters so do not support the **nOp** argument. The results of mathematical operation on an Index or Enum parameter are undefined.

  Note that **nValue** for *Index* parameters are 0 based. The first element in **cEnum**, supplied during registration, is index 0, the second element is index 1, etc...

Most commonly, you will use the **RMS_PARAM_SET (0)** operation. However, there may be instances where you want to simply allow the database to keep track of the value and notify it of changes only. In these cases **RMS_PARAM_INC (1)** and **RMS_PARAM_DEC (2)** are useful for adding and subtracting a value from the current value in the database.

These operations can be used with both *Number* and *String* parameters.

- **RMS_PARAM_INC (1)** appends the string value to the current string
- **RMS_PARAM_DEC (2)** removes the string value from the current string.

The results of multiple and divide on a String parameter are undefined.

Note that values for **slMin**, **slMax** and **cEnumList** supplied during parameter registration are not used to validate the value set using these functions.

- If a *Number* parameter value falls outside the range of **slMin** or **slMax**, the value is stored in the database and displayed.
- If an *Enum* parameter value does not appear in the **cEnum** list, the value is stored in the database and displayed.
- If an *Index* parameter value does not have an associated index in the **Enum** List, the value is stored in the database and displayed as an empty value.

Custom "Scheduling Only" Programming

Overview

If you wish to only utilize the scheduling features of RMS, the RMS SDK can be optimized by only including the necessary RMS modules in your NetLinx program.

NOTE

RMS is capable of supporting multiple (up to 12) instances of Scheduling on a single NetLinx Master. If you intend to run multiple instances of Scheduling on a Master, then that Master should be dedicated solely to RMS Scheduling only.

Review the following listing of RMS modules to determine which you will need to include in your program.

Custom "Scheduling Only" Programming RMS modules

RMS Module	**Required for Scheduling**	**Description**
RMSEngineMod	Yes	The RMSEngineMod is required to facilitate basic communication between the RMS server and the NetLinx master.
i!-ConnectLinxEngineMod	Yes / Dependency	The i!-ConnectLinxEngineMod is a required dependency of the RMSEngineMod and must be included in the NetLinx program.
RMSUIMod	Optional	The RMSUIMod is required if a touch panel is located inside the room to display the room's schedule and meeting information. Additionally, this module supports an external display located outside the room.
RMSWelcomeOnlyUIMod	Optional	The RMSWelcomeOnlyUIMod is required only if a touch panel is located outside a room for an external display of the room's schedule and meeting information, and no touch panel is located inside the room. If the RMSUIMod is included in your program, then the RMSWelcomeOnlyUIMod will not be needed.
RMSHelpUIMod	No	This module is not required for RMS scheduling.
RMSNLDeviceMod	No	This module is not required for RMS scheduling.
RMSProjectorMod	No	This module is not required for RMS scheduling.
RMSTransportMod	No	This module is not required for RMS scheduling.
RMSBasicDeviceMod	No	This module is not required for RMS scheduling.
RMSSldProjMod	No	This module is not required for RMS scheduling.
RMSSrcUsageMod	No	This module is not required for RMS scheduling.

NetLinx Modules

RMSEngineMod Module

- Commands
- Strings
- Channels
- Levels
- Module Definition

Commands

RMSEngineMod listens for the following commands from the *vdvRMSEngine* device:

RMSEngineMod - Commands and Descriptions	
`'SERVER-[IP/Hostname]'`	Set the address of the RMS server. A port can be specified by adding a colon (:) and a port number to the end of the IP address or host name. Example: `'192.168.1.1:10501'` (sets the port to 10501)
`'?SERVER'`	Request the current server settings from the module.
`'TELNET PORT-[PortNumber]'`	Sets the TELNET port the RMS Engine will use to communicate with the NetLinx master. • This command is only needed if the TELNET port on the master has been changed from the default port 24. • This command should be issued in the online event handler for the vdvRMSEngine virtual device to ensure the change instruction is communicated prior to the RMS Engine's initialization and connection to the RMS server. Example: `'TELNET PORT-1023'`
`'?TELNET PORT'`	Queries the RMS Engine for the currently configured TELNET port used by the RMS Engine. • After issuing this command the RMS Engine will respond with 'TELNET PORT-23', where '23' is the value of the current TELNET port configuration. • The response is sent as a STRING on the vdvRMSEngine virtual device and as a STRING on device 0, the master's console output. Example: `'?TELNET PORT'`
`'HTTP PORT-[PortNumber]'`	Sets the HTTP port the RMS Engine will use to communicate with the NetLinx master. • This command is only needed if the HTTP port on the master has been changed from the default port 80. • This command should be issued in the online event handler for the vdvRMSEngine virtual device to ensure the change instruction is communicated prior to the RMS Engine's initialization and connection to the RMS server. Example: `'HTTP PORT-8080'`

RMSEngineMod - Commands and Descriptions (Cont.)	
`'?HTP PORT'`	Queries the RMS Engine for the currently configured HTTP port used by the RMS Engine. • After issuing this command the RMS Engine will respond with 'HTTP PORT-80', where '80' is the value of the current HTTP port configuration. • The response is sent as a STRING on the vdvRMSEngine virtual device and as a STRING on device 0, the master's console output. Example: `'?HTTP PORT'`
`'CONNECT'`	If the NetLinx master's RMS IP socket is not connected to the RMS server, this command will force an immediate connection attempt.
`'DISCONNECT'`	If the NetLinx master's RMS IP socket is currently connected to a RMS server, this command will force an immediate disconnect.
`'RECONNECT'`	If the NetLinx master's RMS IP socket is currently connected to a RMS server, this command will force an immediate disconnect and re-connect. If the NetLinx master's RMS IP socket is not currently connected to a RMS server, this command will force an immediate connection attempt.
`'CLROOT-[Folder]'`	Set the root folder of i!-ConnectLinx for this room. This command can be used to limit the portion of the i!-ConnectLinx tree available to users when configuring room presets. This command is used when a single NetLinx master is running more than one instance of RMS.
`'ADD DEV-[DPS],[Name],[Man],[Model]`	Set device info for monitored device. DPS must be in string form (ex: `'5001:1:0'`). ***Note**: Name, Manufacturer and Model are optional.*

RMSEngineMod - Commands and Descriptions (Cont.)	
`'ADD NPARAM-[DPS],[Name],[Threshold],[Status],[Reset],[Initial Value],[Min],[Max]'`	Add a Number parameter. DPS must be in string form (ex: `'5001:1:0'`).
Threshold can be any of the following:	• >=Value - Greater Than or Equal to Value • <=Value - Less Than or Equal to Value • ==Value -Equal to Value • !=Value - Not Equal to Value • <>Value - Not Equal to Value • >Value - Greater Than Value • <Value - Less Than Value • (Value) - Contains Value •)Value(- Does Not Contain Value
Status can be any of the following:	• NONE or 1 - Not assigned • HELP or 2 - Help Request • ROOM or 3 - Room Communication Error • CONT or 4 - Control System Error • MAIN or 5 - Maintenance Error • USAG or 6 - Equipment Usage • NETW or 7 - Network Error • SECU or 8 - Security Error
Reset can be any reset value or 'NO RESET' for no reset.	
Initial Value can be one of the following:	• Value - Set Value • +=Value - Increment by Value • -=Value - Decrement by Value • *=Value - Multiply by Value • \=Value - Divide by Value • /=Value - Divide by Value • RESET - reset the value • NOCHANGE - do not change the value.
slMin and *slMax* can be any number from -2147483647 to 2147483647. *slMax* must be greater than *slMin*. **Note**: *Set slMax and slMin to zero or omit the values to not impose a range.*	
`'ADD SPARAM-[DPS],[Name],[Threshold],[Status],[Reset],[Initial Value]'`	Add a String parameter. See above in the 'ADD NPARAM' definition for a description of all remaining arguments.
`'ADD IPARAM-[DPS],[Name],[Threshold],[Status],[Reset],[Initial Value],[Enum List]'`	Add an Index parameter. Enum List is a list of possible values separated by the '|' character. See above in the 'ADD NPARAM' definition for a description of all remaining arguments.
`'ADD EPARAM-[DPS],[Name],[Threshold],[Status],[Reset],[Initial Value],[Enum List]'`	Add an Enum parameter. Enum List is a list of possible values separated by the '|' character. See above in the 'ADD NPARAM' definition for a description of all remaining arguments.
`'ADD NPARAM2-[DPS],[Name],[Units],[Track Changes],[Threshold],[Status],[Reset],[Initial Value],[Min],[Max]'`	Add a Number parameter with unit of measurement and/or history tracking. Units is a text string of the units to append to the parameter value when displayed. Track Changes can be a value of '1' to enable or a value of '0' to disable historical parameter value tracking. See above in the 'ADD NPARAM' definition for a description of all remaining arguments.

RMSEngineMod - Commands and Descriptions (Cont.)	
`'ADD SPARAM2-[DPS],[Name],[Units],[TrackChanges],[Threshold],[Status],[Reset],[Initial Value]'`	Add a String parameter with unit of measurement and/or history tracking. Units is a text string of the units to append to the parameter value when displayed. Track Changes can be a value of '1' to enable or a value of '0' to disable historical parameter value tracking. See above in the 'ADD NPARAM' definition for a description of all remaining arguments.
`'ADD IPARAM2-[DPS],[Name],[Units],[Track Changes],[Threshold],[Status],[Reset],[Initial Value],[Enum List]'`	Add an Index parameter with unit of measurement and/or history tracking. Units is a text string of the units to append to the parameter value when displayed. Track Changes can be a value of '1' to enable or a value of '0' to disable historical parameter value tracking. Enum List is a list of possible values separated by the '|' character. See above in the 'ADD NPARAM' definition for a description of all remaining arguments.
`'ADD EPARAM2-[DPS],[Name],[Units],[Track Changes],[Threshold],[Status],[Reset],[Initial Value],[Enum List]'`	Add an Enum parameter with unit of measurement and/or history tracking. Units is a text string of the units to append to the parameter value when displayed. Track Changes can be a value of '1' to enable or a value of '0' to disable historical parameter value tracking. Enum List is a list of possible values separated by the '|' character. See above in the 'ADD NPARAM' definition for a description of all remaining arguments.
`'SET PARAM-[DPS],[Name],[Value]'` *Value* can be one of the following:	Set a parameter value. DPS must be in string form (ex: '5001:1:0'). • Value - Set Value • +=Value - Increment by Value • -=Value - Decrement by Value • *=Value - Multiply by Value • \=Value - Divide by Value • /=Value - Divide by Value • RESET - reset the value • NOCHANGE - do not change the value.
`'HELP-[Help Message]'`	Send a help request.
`'MAINT-[Maintenance Message]'`	Send a maintenance request.
`'GET APPTS-[Date]'`	Get the appointment list for Date.
`'GET APPT COUNT-[Month],[Year]'`	Get the appointment count for Month and Year.
`'DFORMAT-DAY/MONTH'`	Set Date format European format: Day/Month/Year
`'DFORMAT-MONTH/DAY'`	Set Date format US format: Month/Day/Year
`'TFORMAT-12 HOUR'`	Set Time format to 12 hour format: [01-12]:[00-59] [AM,PM]
`'TFORMAT-24 HOUR'`	Set Time format to 24-hour (military) format: [00-23]:[00-59]
`'DEBUG-CONNECT'`	Turn on debug. (including messages related to connection to RMS server)
`'DEBUG-ON’`	Turn on general debug. (no connection messages)
`'DEBUG-OFF’`	Turn off debug. (Default)
`'VERSION’`	Send version information to master debug port (master messaging)
`'SERVICE-ON’`	Turn on service mode. While in service mode, no errors will be reported.
`'SERVICE-OFF’`	Turn off service mode. (Default)

RMSEngineMod - Commands and Descriptions (Cont.)	
'ANSWER-QuestionID,Flags,Answer'	Provide an answer to a question. QuestionID and Flags should be the values supplied from the QUESTION- string. The Answer is a text string.
'IGNORE SERVER TIME UPDATE'	Normally, NetLinx receives a time update from the RMS server. This command tells the module to ignore these time updates.
‘RESERVE-[StartDate],[StartTime],[Duration],[Subject],[<MessageBody>]	Requests the RMS server to add an ac-hoc appointment on behalf of the room. • [StartDate] should be formatted as ‘mm/dd/yyyy’ • [StartTime] should be formatted as ‘hh:mm’ • [Duration] is the number of minutes the meeting should be scheduled. • [Subject] provides the meeting subject text. • [<MessageBody>] is an optional parameter and it provides the meeting message body text. Upon receipt, the RMS Server will process this request and return a confirmation or error ‘RESERVE-‘ string. Please see the Strings API page for further information of the returned server response. The ‘RESERVE’ command is supported when using the RMS internal scheduling system or when using Microsoft Exchange. The ‘RESERVE’ command request will fail if the requested appointment/meeting will conflict with another scheduled appointment/meeting.
‘EXTEND-[Minutes]’	Extends the current appointment/meeting by the number of minutes provided. Upon receipt, the RMS Server will process this request and return a confirmation or error ‘EXTEND-‘ string. Please see the Strings API page for further information of the returned server response. The ‘EXTEND’ command is supported when using the RMS internal scheduling system or when using Microsoft Exchange. The ‘EXTEND’ command request will fail if the appointment/meeting extension will conflict with another scheduled appointment/meeting.
‘ENDNOW’	End the current appointment/meeting immediately. Upon receipt, the RMS Server will process this request and return a confirmation or error ‘ENDNOW-‘ string. Please see the Strings API page for further information of the returned server response. The ‘ENDNOW’ command is supported when using the RMS internal scheduling system or when using Microsoft Exchange.

RMSEngineMod - Commands and Descriptions (Cont.)	

`'?SERVER COMM'`	This command will query the RMS server to obtain the server outbound communication status for each of the out-bound communication paths.
	Possible string responses:
	`// server outbound communication status response (1=online)`
	`SERVER COMM-SMTP=1,SYSLOG=1,SNPP=1,SNMP=1`
	
	`// server outbound communication status response (0=offline)`
	`SERVER COMM-SMTP=0,SYSLOG=0,SNPP=0,SNMP=0`
	
	`// master is not connected to RMS server`
	`SERVER COMM-UNKNOWN`

`'?LICENSE'`	This command will query the RMS server to obtain the server and room licensing information.
	Possible string responses:
	`// licensing response (1=licensed)`
	`LICENSE-SERVER=1,ASSET=1,SCHEDULE=1`
	
	`// licensing response (0=unlicensed)`
	`LICENSE-SERVER=0,ASSET=0,SCHEDULE=0`
	
	`// unknown licensing state response`
	`// (master is not connected to RMS server)`
	`LICENSE-UNKNKOWN`
	These responses are also automatically sent out (unsolicited) each time the master establishes a connection to the RMS server.

`'RFID INITIALIZE'`	This command is automatically sent from the RMSEngine on the RMSEngine virtual device if the RMS server supports RFID tracking. This command is sent out on each connection to the RMS server.
`READERS-<address>[,<address>]*`	Reports all RFID reader addresses to the RMS Server. <address>: user assigned RFID reader address This command is sent upon receipt of 'RFID INITIALIZE' from RMS. `READERS-96:1:1,97:1:1,SomeAddress` ***Note:*** *This command syntax is an exact match to the output command feedback from the Anterus Duet module.* *These commands send out on the Anterus virtual device are simply relayed to the RMSEngine virtual device and on to the RMSServer.* *For more detailed information on the command syntax, please consult the Anterus Duet module documentation.*

RMSEngineMod - Commands and Descriptions (Cont.)	
`TAGSBYREADER-<readerAddress>,<tagCount>,<tagIndex>[,<tagId>,<tagName>,<tagInfo>,<tagSignalStrength>,<tagPercentPower>]*`	Reports all RFID tags acquired by a RFID reader to the RMS Server. • <readerAddress>: user assigned reader address • <tagCount>: total # of tags acquired. • <tagIndex>: index of the first tag in this event. • <tagId>: tag id • <tagName>: tag Name • <tagInfo>: tag Info • <tagSignalStrength>: tag RSSI level 0...255 • <tagPercentPower>: 0...100 (100 means full battery) `TAGSBYREADER-97,2,1,1111076,MyTag1, tag info1,250,100,1111048,MyTag2,tag info2,250,90` ***Note:*** *This command syntax is an exact match to the output command feedback from the Anterus Duet module.* *These commands send out on the Anterus virtual device are simply relayed to the RMSEngine virtual device and on to the RMSServer.* *For more detailed information on the command syntax, please consult the Anterus Duet module documentation.*
`TAGACQUIRED-<readerAddress>,<tagID>,<tagName>,<tagInfo>,<timestamp>,<tagSignalStrength>,<tagPercentPower>`	Reports the acquisition of a RFID tag to the RMS Server. • <readerAddress>: user assigned reader address • <tagID>: tag id • <tagName>: tag Name • <tagInfo>: tag Info • <timestamp>: YYYYMMDDhhmmssnnn string • <tagSignalStrength>: 1...255 • <tagPercentPower>: 0...100 (100 means full battery) `TAGACQUIRED-97,1111002,Mytag, Tag info ,20071107133720449,60,50` ***Note:*** *This command syntax is an exact match to the output command feedback from the Anterus Duet module.* *These commands send out on the Anterus virtual device are simply relayed to the RMSEngine virtual device and on to the RMSServer.* *For more detailed information on the command syntax, please consult the Anterus Duet module documentation.*
`TAGLOST-<readerAddress>,<tagID>,<tagName>,<tagInfo>,<lastTimeStamp>`	Reports the loss of a RF tag to the RMS Server. • <readerAddress>: user assigned reader address • <tagID>: tag id • <tagName>: tag Name • <tagInfo>: tag Info • <timestamp>: YYYYMMDDhhmmssnnn string. Time of last valid tag signal level read. `TAGLOST-97,1111002,MyTag, Tag info,20071107133720449` ***Note:*** *This command syntax is an exact match to the output command feedback from the Anterus Duet module.* *These commands send out on the Anterus virtual device are simply relayed to the RMSEngine virtual device and on to the RMSServer.* *For more detailed information on the command syntax, please consult the Anterus Duet module documentation.*

RMSEngineMod - Commands and Descriptions (Cont.)	
`TAGSIGNALSTRENGTH-<readerAddress>,<tagID>,<tagName>,<tagInfo>,<timestamp>,<tagSignalStrength>,<tagPercentPower>`	Reports a change in the signal strength of a RF tag to the RMS Server. • <readerAddress>: user assigned reader address • <tagID>: tag id • <tagName>: tag Name • <tagInfo>: tag Info • <timestamp>: YYYYMMDDhhmmssnnn string • <tagSignalStrength>: 1...255 • <tagPercentPower>: 0...100 (100 means full battery) TAGSIGNALSTRENGTH-97,1111002,Mytag,Tag info,20071107133830449,65,50 ***Note:*** *This command syntax is an exact match to the output command feedback from the Anterus Duet module.* *These commands send out on the Anterus virtual device are simply relayed to the RMSEngine virtual device and on to the RMSServer.* *For more detailed information on the command syntax, please consult the Anterus Duet module documentation.*
`READERSTATUS-<address>,<status>,<errCnt>`	Reports a reader's status information to the RMS Server. • <address>: user assigned reader address • <status>: ONLINE, OFFLINE • <errCnt>: error count = # of invalid tag reads. READERSTATUS-96,ONLINE,0 ***Note:*** *This command syntax is an exact match to the output command feedback from the Anterus Duet module.* *These commands send out on the Anterus virtual device are simply relayed to the RMSEngine virtual device and on to the RMSServer.* *For more detailed information on the command syntax, please consult the Anterus Duet module documentation.*

Strings

RMSEngineMod listens for the following string from the vdvRMSEngine device:

RMSEngineMod - Strings and Descriptions	
`'POWER=[Power State]'`	Set the system power state. [PowerState] should be 0 for off and 1 for on. **`Example:`** `'POWER=1'`

RMSEngineMod sends the following strings to the vdvRMSEngine device:

RMSEngineMod - Strings and Descriptions	
`'REGISTER'`	Signifies that the RMS Engine module has connected to the RMS server. Upon receiving this string, it is safe to register devices and parameters.
`'SET PARAM-[DPS],[Name],[Value]'`	Set a parameter value in NetLinx. DPS will be in string form (ex: '5001:1:0'). This string is sent when the administrator manually resets a parameter from the RMS console.
`'ROOM NAME-[Name]`	The name of the room as defined in the RMS server.
`'ROOM LOCATION-[Location]'`	The location of the room as defined in the RMS server.
`'ROOM OWNER-[Owner]'`	The owner of the room as defined in the RMS server.

RMSEngineMod - Strings and Descriptions (Cont.)	
`'WEB SERVER-[HostPort]'`	The host/port of the web server where RMS is running.
`'WEB ROOT-[Directory]'`	The root directory of RMS on the web server where RMS is running.
`'CHANGE-[Date1,Date2, Date3,...]'`	A message indicating that appointments have been changed for the following dates. Dates are comma separated and in the format MM/DD/YYYY.
`"'APPT COUNT-[Month], [Year],'[Binary Array]"`	The appointment count for Month and Year. Binary Array is a character array that contains the counts for each day as binary value where the first element in the array is for the first day of Month.
`"'APPT LIST-[Date], [Total Appt], '[Binary Appt List]"`	The appointment list summary for Date. Total Appt provides the total number of appointments for Date. Binary Appt list is a character array with 48 entries, 1 for each ½ hour time slot for Date. The value in index 1 is the appointment index of the appointment that occupies the first ½ hour time slot (12:00am - 12:30 am).
`"'APPT-Index, '[Binary Data]"`	The appointment data for an individual appointment. Index is a value between 1 and Total Appt form the 'APPT LIST" string. The Binary Data is a VARIABLE_TO_STRING encoded copy of an appointment structure. The appointment structure is defined as _sRMSAppointment in the include file. This binary data can be passed to the *RMSDecodeAppointmentString()* function, defined in the include file, to convert the data to an appointment structure.
`'MESG-[Flags],[Message]'`	A message from the RMS console. Flags are the message flag and Message is the text of the message. Currently, there are no Flags defined.
`'EXTEND-No'`	A message indicating the current appointment could not be extended due to a scheduling conflict.
`'SERVER-[Address]:[Port]'`	The current RMS server address used by the module. This is sent in response to a '/SERVER' command.
`'PRODUCT-[ID],[Name], [Version]'`	The product ID, Product name, and version of the RMS server this module is connected to. This string is sent upon connection to the RMS server.
`'QUESTION-[ID],[Flags], [Questions],[Answer1], [Answer2],[Answer3], [Answer4]'`	A question sent from the RMS server. The ID and Flags are used to send the response. A question and up to 4 answers may be included. All answers are optional.
`‘RESERVE-YES,[StartDate], [StartTime],[Duration], [Subject]`	A “RESERVE’ appointment/meeting request command succeeded. • [StartDate] is formatted as ‘mm/dd/yyyy’ • [StartTime] is formatted as ‘hh:mm’ • [Duration] is the number of minutes the meeting is scheduled for. • [Subject] meeting subject text.
`‘RESERVE-NO,[StartDate], [StartTime],[Duration], [Subject],[ErrorMessage]`	A “RESERVE’ appointment/meeting request command failed or was rejected by the server. • [StartDate] is formatted as ‘mm/dd/yyyy’ • [StartTime] is formatted as ‘hh:mm’ • [Duration] is the number of minutes the meeting is scheduled for. • [Subject] meeting subject text. • [MessageBody] meeting message body text. • [ErrorMessage] An error message is provided describing the reason for the failure

RMSEngineMod - Strings and Descriptions (Cont.)	
`‘EXTEND-YES’`	An “EXTEND’ current appointment/meeting request command succeeded.
`‘EXTEND-NO, [ErrorMessage]’`	An “EXTEND’ current appointment/meeting request command failed or was rejected by the server. An error message is provided describing the reason for the failure.
`‘ENDNOW-YES’`	An “ENDNOW’ request command succeeded.
`‘ENDNOW-NO, [ErrorMessage]’`	An “ENDNOW’ request command failed or was rejected by the server. An error message is provided describing the reason for the failure.

`'SERVER COMM-SMTP=<status>,SYSLOG=<status>, SNPP=<status>,SNMP=<status>'`	This is the response string format returned for a '?SERVER COM' query command. Each outbound communication channel reports is status with a '1' for online or a '0' for offline. The <status> in the string format represents a '0' or '1' depending on the status of each communication channel.
	Example string responses:
	`// server outbound communication status response (1=online)`
	`SERVER COMM-SMTP=1,SYSLOG=1,SNPP=1,SNMP=1`
	
	`// server outbound communication status response (0=offline)`
	`SERVER COMM-SMTP=0,SYSLOG=0,SNPP=0,SNMP=0`

`'SERVER COMM-UNKNOWN'`	This is the response string format returned for a '?SERVER COM' query command when the master is not connected to the RMS Server and the actual outbound communication status cannot be determined.

`'LICENSE-SERVER=<status>, ASSET=<status>, SCHEDULE=<status>'`	This is the response string format returned for a '?LICENSE' query command.
	This return string will return the status for the server license and the room's asset and scheduling license.
	The <status> in the string format represents a '0' for not licensed or '1' for licensed depending on the status of each licensed component.
	Example string responses:
	`// licensing response (1=licensed)`
	`LICENSE-SERVER=1,ASSET=1,SCHEDULE=1`
	
	`// licensing response (0=unlicensed)`
	`LICENSE-SERVER=0,ASSET=0,SCHEDULE=0`
	This status string is also automatically sent out (unsolicited) each time the master establishes a connection to the RMS server.

`'LICENSE-UNKNKOWN'`	This is the response string format returned for a '?LICENSE' query command when the master is not connected to the RMS server and the actual license status cannot be determined.

Channels

RMSEngineMod - Channels	
9	Toggle System Power State
27	Set system power to ON
28	Set system power to OFF
100	Run Preset for Current Appointment
248	RMS Server Socket Connected This channel reflects the state of the actual TCP/IP connection between the master and the RMS Server.
249	RMS Database Online This channel indicates the status of the RMS Database connection on the RMS server.
250	RMS Server Online This channel is a composite channel that indicates that the RMS server is online or offline. Both the RMS server TCP/IP socket must be connected and the RMS Database must be Online for this channel to be set to the ON state. If one of these conditions is not met, this channel will be set to the OFF state.
251	Dynamic Images Enabled

Levels

RMSEngineMod - Levels	
1	Current Appointment Index
2	Minutes Remaining In Appointment
3	Next Appointment Index
4	Minutes Until Next Appointment
5	First Appointment Index
6	Last Appointment Index
7	Number of Appointment Remaining today

Module Definition

```
DEFINE_MODULE 'RMSEngineMod' mdlRMSEng(vdvRMSEngine,
                                        dvRMSSocket,
                                        vdvCLActions).
```

Where `mdlRMSEng` is a unique module name.

- `vdvRMSEngine:` A virtual device for communicating to the RMSEngineMod module
- `dvRMSSocket:` A socket device for communicating to the RMS server.
- `vdvCLActions:` A i!-ConnectLinx device number for executing pre-meeting presets. If you are not using i!-ConnectLinx, pass in a value of 0:0:0.

Touch Panel Pages

This module requires no pages.

RMSRFIDTrackingMod Module

- Commands
- Module Definition

Commands

RMSRFIDTrackingMod listens for the following commands from the vdvRMSEngine device:

RMSRFIDTrackingMod - Commands and Descriptions	
`'RFID INITIALIZE'`	This command is automatically sent from the RMSEngine on the RMSEngine virtual device if the RMS server supports RFID tracking. This command is sent out on each connection to the RMS server.
`'DEBUG ON'`	Enable RMS debugging messages.
`'DEBUG OFF'`	Disable RMS debugging messages.
`'VERSION'`	Send version information to master debug port (master messaging)

Module Definition

The purpose of this module is to synchronize RFID tag changes received from the Anterus RFID Module with the RMS Server. This module is only needed if using RFID tracking and the Anterus RFID Duet Module.

NOTE

If you are deploying multiple instance of RMS to a single NetLinx master, then use the module "RMSRFIDTrackingMod-Multi" instead of this module. More information can be found describing the step required to support RFID device tracking with multiple instances of RMS in the Multi-Instancing RFID Device Tracking in RMS section on page 75.

```
DEFINE_MODULE 'RMSRFIDTrackingMod'
mdlRMSRFIDTracking(vdvAnterusGateway,vdvRMSEngine)
```

Where `mdlRMSRFIDTracking` is a unique module name.

• `vdvAnterusGateway`	A virtual device for communicating to the Anterus RFID Module
• `vdvRMSEngine`	A virtual device for communicating to RMSEngineMod module.

Touch Panel Pages

This module requires no pages.

RMSRFIDTrackingMod-Multi Module

- Commands
- Module Definition

Commands

RMSRFIDTrackingMod-Multi listens for the following commands from the vdvRMSEngine device:

RMSRFIDTrackingMod-Multi - Commands and Descriptions	
`'RFID INITIALIZE'`	This command is automatically sent from the RMSEngine on the RMSEngine virtual device if the RMS server supports RFID tracking. This command is sent out on each connection to the RMS server.
`'DEBUG ON'`	Enable RMS debugging messages.
`'DEBUG OFF'`	Disable RMS debugging messages.
`'VERSION'`	Send version information to master debug port (master messaging)

Module Definition

The purpose of this module is to synchronize RFID tag changes received from the Anterus RFID Module with a single instance of RMS in a multi-instance RMS NetLinx program. This module is only needed if using RFID tracking and the Anterus RFID Duet Module in a multi-instance RMS configuration.

If not deploying multiple instance of RMS to a single NetLinx master, then use the module "RMSRFIDTrackingMod" instead of this module.

```
DEFINE_MODULE 'RMSRFIDTrackingMod-Multi' mdlRMSRFIDTracking
(vdvAnterusGateway,vdvRMSEngine,cRFIDReaderAddresses[][])
```

Where `mdlRMSRFIDTracking` is a unique module name.

• `vdvAnterusGateway:`	A virtual device for communicating to the Anterus RFID Module
• `vdvRMSEngine:`	A virtual device for communicating to an instance of RMSEngineMod module.
• `cRFIDReaderAddresses[][]:`	This parameter should contain an array of strings (a two-dimensional array) used to identify which Anterus RFID readers that this module instance is responsible for monitoring and relaying RFID tag information to the RMS server. Only RFID tags from RFID readers that are defined in this parameter will be sent to the RMS server. All others will be filtered out. Each RFID reader configured in Anterus supports a user-defined reader address label. It is this reader address label that should be used and included in the string array for this module parameter. For more information on the RFID reader address label, please refer to the Anterus module documentation.

Touch Panel Pages

This module requires no pages.

RMSUIMod Module

- Commands
- Module Definition
- Touch Panel Pages
- Constants

Commands

RMSUIMod listens for the following commands from the vdvRMSEngine device:

RMSUIMod - Commands and Descriptions	
`'DFORMAT-DAY/MONTH'`	Set Date format European format: Day/Month/Year
`'DFORMAT-MONTH/DAY'`	Set Date format US format: Month/Day/Year
`'TFORMAT-12 HOUR'`	Set Time format to 12 hour format: [01-12]:[00-59] [AM,PM]
`'TFORMAT-24 HOUR'`	Set Time format to 24-hour (military) format: [00-23]:[00-59]
`'DEBUG-ON'`	Turn on debug.
`'DEBUG-OFF'`	Turn off debug. (Default)
`'VERSION'`	Send version information to master debug port (master messaging)

Module Definition

```
DEFINE_MODULE 'RMSUIMod' mdlRMSUI(vdvRMSEngine,
                                  dvTp,
                                  dvTP_Base,
                                  dvTPWelcome,
                                  dvTPWelcome_Base,
                                  RMS_MEETING_DEFAULT_SUBJECT,
                                  RMS_MEETING_DEFAULT_MESSAGE)
```

• `vdvRMSEngine:`	A virtual device for communicating to RMSEngineMod RMSEngineMod module
• `dvTP:`	An array of main touch panel devices implementing RMSUIMod.
• `dvTP_Base:`	An array of main touch panel base devices addresses that correspond with the main touch panel devices defined in the dvTP device array. This base device array is used to capture keyboard string data events from the main touch panels. • If the RMS pages and buttons are defined on the base device address (Port 1) then the same device array used for the dvTP parameter can also be used for this parameter. • If the RMS pages and buttons are defined on a panel port that is not the base device address, then this parameter will need an array of the base panel device addresses.
• `dvTPWelcome:`	An array of welcome touch panel devices implementing RMSUIMod.
• `dvTPWelcome_Base:`	An array of welcome touch panel base devices addresses that correspond with the welcome touch panel devices defined in the dvTPWelcome device array. This base device array is used to capture keyboard string data events from the welcome touch panels. • If the RMS welcome pages and buttons are defined on the base device address (Port 1) then the same device array used for the dvTPWelcome parameter can also be used for this parameter. • If the RMS welcome pages and buttons are defined on a panel port that is not the base device address, then this parameter will need an array of the base panel device addresses.

• `RMS_MEETING_ DEFAULT_SUBJECT:`	A character array/string that defined the default subject text for meeting appointments scheduled form the touch panel.
• `RMS_MEETING_ DEFAULT_MESSAGE:`	A character array/string that defined the default message body text for meeting appointments scheduled form the touch panel.

NOTE

All channel and variable text numbers are defined inside the module. If you need to change them, edit the module and re-compile the module and your program.

Touch Panel Pages

RMSUIMod requires the following touch panel pages for **dvTP**:

- **Main** - user defined room control page.
- **rmsSplashPage** - panel start-up page
- **rmsRoomPage** - RMS inside room control panel

RMSUIMod requires the following touch panel pages for dvTPWelcome:

- **rmsSplashPage** - panel start-up page
- **rmsWelcomePage** - RMS outside room welcome panel (electronic signage panel)

RMSUIMod - Touch Panel Pop-Up Pages	
Required for dvTp	
Popup Page	**Popup Group**
`rmsAbout`	NA
`rmsCalendar`	NA
`rmsDoNotDisturb`	NA
`rmsServerOffline`	NA
`rmsHelpQuestion`	rmsDialogs
`rmsHelpRequest`	rmsDialogs
`rmsHelpResponse`	rmsDialogs
`rmsMeetingDoesNotExist`	rmsDialogs
`rmsMeetingEndConfirmation`	rmsDialogs
`rmsMeetingEndWarning`	rmsDialogs
`rmsMeetingExtendWarning`	rmsDialogs
`rmsMeetingInfo`	rmsDialogs
`rmsMeetingRequest`	rmsDialogs
`rmsMeetingRequestFailed`	rmsDialogs
`rmsMessage`	rmsDialogs
`rmsServiceRequest`	rmsDialogs
`rmsDoorbell`	rmsDoorbell
`rmsMonthSelect`	rmsDropDowns
`rmsYearSelect`	rmsDropDowns
`rmsViewSchedule1`	rmsViewSchedule
`rmsViewSchedule3`	rmsViewSchedule
`rmsViewSchedule4`	rmsViewSchedule

RMSUIMod - Touch Panel Pop-Up Pages (Cont.)	
Required for dvTPWelcome	
Popup Page	**Popup Group**
`rmsAbout`	NA
`rmsDoNotDisturb`	NA
`rmsServerOffline`	NA
`rmsMeetingDetails`	rmsDialogs
`rmsMeetingDoesNotExist`	rmsDialogs
`rmsMeetingEndConfirmation`	rmsDialogs
`rmsMeetingInfo`	rmsDialogs
`rmsMeetingRequest`	rmsDialogs
`rmsMeetingRequestFailed`	rmsDialogs
`rmsMessage`	rmsDialogs
`rmsViewSchedule1`	rmsViewSchedule
`rmsViewSchedule3`	rmsViewSchedule
`rmsViewSchedule4`	rmsViewSchedule

Constants

The following constants are defined in the RMSUIMod.axs file:

RMSUIMod - Constants	
Constant	**Description**
`__RMS_UI_NAME__`	RMS module name. This constant should not be modified.
`__RMS_UI_VERSION__`	RMS module version. This constant should not be modified.
`RMS_BEEP_DOORBELL_REQUEST`	Panel Beep on Doorbell button push from outside welcome touch panel. 1 = ENABLED / 0 = DISABLED
`RMS_BEEP_DOORBELL_RECEIPT`	Panel Beep on Doorbell signal on in room touch panel. 1 = ENABLED / 0 = DISABLED
`RMS_BEEP_PRESET_RUN`	Panel Beep on Meeting Preset Execute button press from in-room touch panel. 1 = ENABLED / 0 = DISABLED
`RMS_WELCOME_PREP_TIME_MIN`	Numbers of minutes before meeting start time to prepare for next meeting.
`RMS_WARN_TIME_MIN`	Number of minutes before meeting ends to display a warning dialog on the touch panel.
`RMS_EXTEND_TIME_MIN`	Number of minutes to extend meeting end time when the EXTEND meeting button is pressed from the touch panel.
`RMS_G4_USER_SCHEDULE_COLOR_0`	Color 1 of 3 Alternating meeting block colors for daily schedule.
`RMS_G4_USER_SCHEDULE_COLOR_1`	Color 2 of 3 Alternating meeting block colors for daily schedule.
`RMS_G4_USER_SCHEDULE_COLOR_2`	Color 3 of 3 Alternating meeting block colors for daily schedule.

RMSWelcomeOnlyUIMod Module

- Commands
- Module Definition
- Touch Panel Pages
- Constants

Commands

RMSWelcomeOnlyUIMod listens for the following commands from the vdvRMSEngine device.

RMSWelcomeOnlyUIMod - Commands and Descriptions	
`'DFORMAT-DAY/MONTH'`	Set Date format European format: Day/Month/Year.
`'DFORMAT-MONTH/DAY'`	Set Date format US format: Month/Day/Year.
`'TFORMAT-12 HOUR'`	Set Time format to 12 hour format: [01-12]:[00-59] [AM,PM].
`'TFORMAT-24 HOUR'`	Set Time format to 24-hour (military) format: [00-23]:[00-59].
`'DEBUG-ON'`	Turn on debug.
`'DEBUG-OFF'`	Turn off debug. (Default).
`'VERSION'`	Send version information to master debug port (master messaging).

Module Definition

```
DEFINE_MODULE ' RMSWelcomeOnlyUIMod' mdlRMSUI(vdvRMSEngine,
                                        dvTPWelcome,
                                        dvTPWelcome_Base,
                                        RMS_MEETING_DEFAULT_SUBJECT,
                                        RMS_MEETING_DEFAULT_MESSAGE)
```

- `vdvRMSEngine:` A virtual device for communicating to the RMSEngineMod module.
- `dvTPWelcome:` An array of welcome touch panel devices implementing RMSUIMod.
- `dvTPWelcome_Base:` An array of welcome touch panel base devices addresses that correspond with the welcome touch panel devices defined in the dvTPWelcome device array.

  This base device array is used to capture keyboard string data events from the welcome touch panels.

  - If the RMS welcome pages and buttons are defined on the base device address (Port 1) then the same device array used for the dvTPWelcome parameter can also be used for this parameter.
  - If the RMS welcome pages and buttons are defined on a panel port that is not the base device address, then this parameter will need an array of the base panel device addresses.
- `vdvRMSEngine:` A virtual device for communicating to RMSEngineMod module.
- `RMS_MEETING_DEFAULT_SUBJECT:` A character array/string that defined the default subject text for meeting appointments scheduled form the touch panel.
- `RMS_MEETING_DEFAULT_MESSAGE:` A character array/string that defined the default message body text for meeting appointments scheduled form the touch panel.

NOTE

All channel and variable text numbers are defined inside the module. If you need to change them, edit the module and re-compile the module and your program.

Touch Panel Pages

RMSWelcomeOnlyUIMod requires the following touch panel pages for dvTP:

- *rmsSplashPage* - panel start-up page
- *rmsWelcomePage* - RMS outside room welcome panel (electronic signage panel)

RMSWelcomeOnlyUIMod requires the following touch panel pop-up pages for *dvTPWelcome*:

RMSWelcomeOnlyUIMod - Touch Panel Pop-up Pages	
Popup Page	**Popup Group**
rmsAbout	NA
rmsDoNotDisturb	NA
rmsServerOffline	NA
rmsMeetingDetails	rmsDialogs
rmsMeetingDoesNotExist	rmsDialogs
rmsMeetingEndConfirmation	rmsDialogs
rmsMeetingInfo	rmsDialogs
rmsMeetingRequest	rmsDialogs
rmsMeetingRequestFailed	rmsDialogs
rmsMessage	rmsDialogs
rmsViewSchedule1	rmsViewSchedule
rmsViewSchedule3	rmsViewSchedule
rmsViewSchedule4	rmsViewSchedule

Constants

The following constants are defined in the RMSWelcomeOnlyUIMod.axs file:

RMSWelcomeOnlyUIMod - Constants	
Constant	**Description**
__RMS_UI_NAME__	RMS module name. This constant should not be modified.
__RMS_UI_VERSION__	RMS module version. This constant should not be modified.
RMS_BEEP_DOORBELL_REQUEST	Panel Beep on Doorbell button push from outside welcome touch panel. 1 = ENABLED / 0 = DISABLED
RMS_BEEP_DOORBELL_RECEIPT	Panel Beep on Doorbell signal on in room touch panel. 1 = ENABLED / 0 = DISABLED
RMS_BEEP_PRESET_RUN	Panel Beep on Meeting Preset Execute button press from in-room touch panel. 1 = ENABLED / 0 = DISABLED
RMS_WELCOME_PREP_TIME_MIN	Numbers of minutes before meeting start time to prepare for next meeting.
RMS_WARN_TIME_MIN	Number of minutes before meeting ends to display a warning dialog on the touch panel.
RMS_EXTEND_TIME_MIN	Number of minutes to extend meeting end time when the EXTEND meeting button is pressed from the touch panel.
RMS_G4_USER_SCHEDULE_COLOR_0	Color 1 of 3 Alternating meeting block colors for daily schedule.
RMS_G4_USER_SCHEDULE_COLOR_1	Color 2 of 3 Alternating meeting block colors for daily schedule.
RMS_G4_USER_SCHEDULE_COLOR_2	Color 3 of 3 Alternating meeting block colors for daily schedule.

RMSHelpUIMod Module

- Commands
- Module Definition
- Touch Panel Pages

Commands

RMSHelpUIMod listens for the following commands from the vdvRMSEngine device.

RMSHelpUIMod - Commands and Descriptions	
`'DFORMAT-DAY/ MONTH'`	Set Date format European format: Day/Month/Year
`'DFORMAT-MONTH/ DAY'`	Set Date format US format: Month/Day/Year
`'TFORMAT-12 HOUR'` `[AM,PM]`	Set Time format to 12 hour format: [01-12]:[00-59]
`'TFORMAT-24 HOUR'` `[00-23]:[00-59]`	Set Time format to 24-hour (military) format:
`'DEBUG-ON'`	Turn on debug.
`'DEBUG-OFF'`	Turn off debug. (Default)
`'VERSION'`	Send version information to master debug port (mastermessaging)

Module Definition

```
DEFINE_MODULE 'RMSHelpUIMod' mdlRMSHelpUI(vdvRMSEngine,
                                          dvTp,
                                          dvTP_Base)
```

- `vdvRMSEngine:` A virtual device for communicating to RMSEngineMod.
- `dvTP:` An array of main touch panel devices implementing RMSHelpUIMod.
- `dvTP_Base:` An array of main touch panel base devices addresses that correspond with the main touch panel devices defined in the dvTP device array.

  This base device array is used to capture keyboard string data events from the main touch panels.

  - If the RMS pages and buttons are defined on the base device address (Port 1) then the same device array used for the dvTP parameter can also be used for this parameter.
  - If the RMS pages and buttons are defined on a panel port that is not the base device address, then this parameter will need an array of the base panel device addresses.

NOTE

All channel and variable text numbers are defined inside the module. If you need to change them, edit the module and re-compile the module and your program.

Touch Panel Pages

RMSHelpUIMod requires the following touch panel pages for dvTP:

- *Main* - user defined room control page.
- *rmsSplashPage* - panel start-up page
- *rmsRoomPage* - RMS inside room control panel

RMSHelpUIMod requires the following touch panel pop-up pages for **dvTp**:

RMSHelpUIMod - Touch Panel Pop-up Pages	
Popup Page	**Popup Group**
`rmsAbout`	NA
`rmsServerOffline`	NA
`rmsHelpQuestion`	rmsDialogs
`rmsHelpRequest`	rmsDialogs
`rmsHelpResponse`	rmsDialogs
`rmsServiceRequest`	rmsDialogs
`rmsMessage`	rmsDialogs
`rmsDoorbell`	rmsDialogs

RMSNLDeviceMod Module

- Commands
- Module Definition

Commands

RMSNLDeviceMod listens for the following commands from the vdvRMSEngine device.

RMSNLDeviceMod - Commands and Descriptions	
`'VERSION'`	Send version information to master debug port (master messaging)

Module Definition

```
DEFINE_MODULE 'RMSNLDeviceMod' mdlNLD(dvMonitoredDevices,
                                      vdvRMSEngine)
```

Where `mdlNLD` is a unique module name.

• `dvMonitoredDevices:`	An array of NetLinx-connected devices to monitor.
• `vdvRMSEngine:`	A virtual device for communicating to the RMSEngineMod module.

Touch Panel Pages

This module requires no pages.

RMSProjectorMod Module

- Commands
- Strings
- Channels
- Module Definition

Commands

RMSProjectorMod listens for the following commands from the *vdvRMSEngine* device.

RMSProjectorMod - Commands and Descriptions (from vdvRMSEngine device)	
`'DEV INFO-[DPS],[Name],[Man],[Model]'`	Set device information for device monitoring. DPS must be in string form (ex: '5001:1:0').
`'DEV NAME-[DPS],[Name]'`	Set device name for device monitoring. DPS must be in string form (ex: '5001:1:0').
`'COMM TO-[DPS],[Timeout]'`	Set the device communication timeout to Timeout. Timeout is in 1/10-second increments. Sending a value of zero will disable the device communication timeout and the registration of the device communicating parameter.
`'DEV PWR-[DPS],[State]'`	Set the device power state. DPS must be in string form (ex: '5001:1:0'). State must be 1 or 0.
`'LAMP HOURS-[DPS],[Value]'`	Set the projector lamp hours. DPS must be in string form (ex: '5001:1:0').
`'POWER FAIL ON-[DPS]'`	Set power failure detection on. DPS must be in string form (ex: '5001:1:0').
`'POWER FAIL OFF-[DPS]'`	Set power failure detection off. DPS must be in string form (ex: '5001:1:0').
`'VERSION'`	Send version information to master debug port (master messaging)

RMSProjectorMod listens for the following commands from the *vdvProjModule* device.

This set of commands is intended to provide monitoring for Duet-based modules.

RMSProjectorMod - Commands and Descriptions (from vdvProjModule device)	
`'LAMPTIME-[Value]'`	Sets the projector lamp hours device information for device monitoring. Example: 'LAMPTIME-200'

Strings

RMSProjectorMod listens for the following strings from the *vdvProjModule* device.

RMSProjectorModStrings and Descriptions	
`'POWER=[Power State]'`	Set the system power state. [PowerState] should be 0 for off and 1 for on. **Example:** `'POWER=1'`
`'LAMPTIME=[Value]'`	Set the projector lamp hours. **Example:** `'LAMPTIME=200'`
`'MODEL=[Model]'`	Set the model number.
`'MANUFACTURER=[Manufacturer]'`	Set the manufacturer name.

Any string received from the physical device (dvProj) is an indication that the device is communicating.

As long as a string is received within the period set by the communication timeout command, the module will notify RMS that the device is communicating. See the Communication Timeout command for more details.

NOTE

If using a Duet module, strings from the physical device are captured inside the Duet module are not bubbled up to the STRING data event.
For Duet devices, instead of monitoring string traffic on physical device, Channel 252 (DATA_INITIALIZED) is used to monitor device communication status.

Channels

RMSProjectorMod watches for these channels on the **vdvProjModule** device:

RMSProjectorMod - Channels	
9	Toggle System Power State
27	Set system power to ON
28	Set system power to OFF
252	Device Communication/Initialized Status (Duet Modules)
254	Power failure
255	Power Status

Module Definition

```
DEFINE_MODULE 'RMSProjectorMod' mdlProj1(vdvProjModule,
                                         dvProj,
                                         vdvRMSEngine)
```

Where `mdlProj1is` a unique module name.

• `vdvProjModule:`	A virtual device for communicating to RMSProjectorMod. This can be the same virtual device used to communicate with an InConcert module or a physical projector. If controlling the projector or display using an IR device, pass the physical device to this parameter.
• `dvProj:`	A physical projector or socket device to which the projector is connected.
• `vdvRMSEngine:`	A virtual device for communicating to RMSEngineMod module.

Touch Panel Pages

This module requires no pages.

Reporting for Multiple Bulb Projectors (Limited Support)

The following section describes how to register multiple lamp parameters, in order to allow reporting on multiple-bulb projectors. Support for this feature requires some custom coding, as described below:

RMSMain.axi File Changes

The following changes must be made to the **RMSMain.axi** file, to register multiple lamp parameters:

```
RMSRegisterStockNumberParam(DVPROJ, 'Lamp Hours A1', RMS_UNIT_HOURS, RMS_TRACK_CHANGES,
     1800, RMS_COMP_GREATER_THAN, RMS_STAT_MAINTENANCE,
     RMS_PARAM_CANNOT_RESET, 0, RMS_PARAM_SET, slRMSPROJECTORLampHoursA1, 0, 0)
RMSRegisterStockNumberParam(DVPROJ, 'Lamp Hours A2', RMS_UNIT_HOURS, RMS_TRACK_CHANGES,
     1800, RMS_COMP_GREATER_THAN, RMS_STAT_MAINTENANCE,
     RMS_PARAM_CANNOT_RESET, 0, RMS_PARAM_SET, slRMSPROJECTORLampHoursA2, 0, 0)
```

The following code shows how Code Crafter originally created the lamp hour parameter:

```
//  RMSRegisterDeviceNumberParam(DVPROJ,'Lamp Hours A1',
//      1800,RMS_COMP_GREATER_THAN,RMS_STAT_MAINTENANCE,
//      FALSE,0,
//      RMS_PARAM_SET,slRMSPROJECTORLampHoursA1,0,0)
//  RMSRegisterDeviceNumberParam(DVPROJ,'Lamp Hours A2',
//      1800,RMS_COMP_GREATER_THAN,RMS_STAT_MAINTENANCE,
//      FALSE,0,
//      RMS_PARAM_SET,slRMSPROJECTORLampHoursA2,0,0)
```

RMSProjectorMod Module Changes

The following changes must be made to the RMSProjectorMod module.

There are two functions that must be changed.

The first runs from line 231 to line 235:

```
(****************************************)
(* Call Name: RMSDevMonSetLampPower    *)
(* Function:  Set Lamp Power           *)
(* Param:     1=On, 0=Off              *)
(* Return:    None                     *)
(****************************************)
DEFINE_FUNCTION RMSDevMonSetLampPower(CHAR bState)
{
     // Are we counting? if not, who cares? This is for lamp counting only
     // DeviceVase.axi takes care of power
     IF (bCountLampHours == FALSE)
          RETURN;
          // Did we register yet? If not, do it now
          // IF (bIgnoreLampHours)                  //comment this out
          // {   bIgnoreLampHours = FALSE
          //     RMSDevMonRegisterCallback()
          //     send_string 0,'did not'
          // }
     // Did the state change
     IF (bState > 0)
     {
     // Start Timer & offset timer to think it started lMinuteRemainder minutes before now
          RMSTimerStart()
          RMSTimerResetStartTime(0,TYPE_CAST(0-lMinuteRemainder),0)
     }
     ELSE
     {
     // Remember minute mark and stop timer
     IF (RMSTimerIsRunning())
          {
               lMinuteRemainder = RMSTimerGetMinutes()
               lMinuteRemainder = lMinuteRemainder % 60
          }
          RMSTimerStop();
     }
}
```

The second runs from line 270 to line 274:

```
(***************************************)
(* Call Name: RMSDevMonSetLampHours    *)
(* Function:  Set Lamp Hours           *)
(* Param:     # of Hours               *)
(* Return:    None                     *)
(***************************************)
DEFINE_FUNCTION RMSDevMonSetLampHours(SLONG slHours)
{
     // Let the comm module handle lamp hours
     // Also, kill timer, we will not be using it anymore...
     bCountLampHours = FALSE
     RMSTimerStop();

     // Did we register yet? If not, do it now
     // IF(bIgnoreLampHours)                          //comment this out
     // {
     //   bIgnoreLampHours = FALSE
     //   RMSDevMonRegisterCallback()
     //   send_string 0,'did it'
     // }
     // bounds check the lamp hours value
     // note if erroneous lamp time is received from Duet module, it will be a Duet min
     // int value (-2147483648)
     IF((slHours >= RMS_LAMP_TIME_LOWER_BOUNDS) && (slHours <= RMS_LAMP_TIME_UPPER_BOUNDS))
     {
          // Is this a change?
          IF (slHours <> slLampHours)
          {
               slLampHours = slHours
               RMSChangeNumberParam(dvRealDevice, RMS_DEVICE_LAMP, RMS_PARAM_SET, slLampHours)
          }
     }
}
```

Remember, when using the modified module, you must register lamp hours in the above manner.

It may be a good principle to keep the modified module with a different name, and then include that named module when using multiple lamp projectors in the DEFINE_MODULE section of the code.

RMSTransportMod Module

- Commands
- Strings
- Channels
- Module Definition

Commands

RMSTransportMod listens for the following commands from the **vdvRMSEngine** device.

RMSTransportMod - Commands and Descriptions	
`'DEV INFO-[DPS],[Name],[Man],[Model]'`	Set device information for device monitoring. DPS must be in string form (ex: `'5001:1:0'`).
`'DEV NAME-[DPS],[Name]'`	Set device name for device monitoring. DPS must be in string form (ex: `'5001:1:0'`).
`'COMM TO-[DPS],[Timeout]'`	Set the device communication timeout to Timeout. Timeout is in 1/10-second increments. Sending a value of zero will disable the device communication timeout and the registration of the device communicating parameter.
`'DEV PWR-[DPS],[State]'`	Set the device power state. DPS must be in string form (ex: '5001:1:0'). State must be 1 or 0.
`'XPORT STATE-[DPS],[State]'`	Set the transport state. DPS must be in string form (ex: ''`5001:1:0'`). State should be a value 1-8, Play=1, Stop=2, etc…
`'POWER FAIL ON-[DPS]'`	Set power failure detection on. DPS must be in string form (ex: `'5001:1:0'`).
`'POWER FAIL OFF-[DPS]'`	Set power failure detection off. DPS must be in string form (ex: `'5001:1:0'`).
`'VERSION'`	Send version information to master debug port (master messaging)

Strings

RMSTransportMod listens for the following strings from the **vdvVCRModule** device.

RMSTransportMod - Strings and Descriptions	
`'POWER=[Power State]'`	Set the system power state. [PowerState] should be 0 for off and 1 for on. Example: 'POWER=1'
`'TRANSPORT=[Transport State]'`	Set the system power state. Transport State can be 2 for Stop or any other value for a non-stopped condition. Any value other than 2 will enable the run-time counter.
`'MODEL=[Model]'`	Set the model number.
`'MANUFACTURER=[Manufacturer]'`	Set the manufacturer name.

Any string received from the physical device (dvProj) is an indication that the device is communicating.

As long as a string is received within the period set by the communication timeout command, the module will notify RMS that the device is communicating. See the Communication Timeout command for more details.

NOTE

If using a Duet module, strings from the physical device are captured inside the Duet module are not bubbled up to the STRING data event.
For Duet devices, instead of monitoring string traffic on physical device, Channel 252 (DATA_INITIALIZED) is used to monitor device communication status.

Channels

RMSTransportMod watches for these channels on the physical device, and for Duet modules on the virtual device:

RMSTransportMod - Channels	
9	toggle System Power State
27	Set system power to ON
28	Set system power to OFF
241	Set transport state to play.
242	Set transport state to stop.
243	Set transport state to pause.
244	Set transport state to fast forward.
245	Set transport state to rewind.
246	Set transport state to search forward.
247	Set transport state to search reverse.
248	Set transport state to record.
252	Device Communication/Initialized Status (Duet Modules)
254	Power failure
255	Power Status

Module Definition

```
DEFINE_MODULE 'RMSTransportMod' mdlVCR1(vdvVCRModule,
                                        dvVCR,
                                        vdvRMSEngine)
```

Where `mdlVCR1` is a unique module name.

• `vdvVCRModule:`	A virtual device for communicating to RMSTransportMod. This can be the same virtual device used to communicate with an InConcert module or the physical device. If using a SYSTEM_CALL for this device, pass the physical device to this parameter.
• `dvVCR:`	A physical device or socket device to which the transport device is connected.
• `vdvRMSEngine:`	A virtual device for communicating to the RMSEngineMod module.

Touch Panel Pages

This module requires no pages.

RMSBasicDeviceMod Module

- Commands
- Strings
- Channels
- Module Definition

Commands

RMSBasicDeviceMod listens for the following commands from the **vdvRMSEngine** device.

RMSBasicDeviceMod - Commands and Descriptions	
`'DEV INFO-[DPS],[Name],[Man],[Model]'`	Set device information for device monitoring. DPS must be in string form (ex: '5001:1:0').
`'DEV NAME-[DPS],[Name]'`	Set device name for device monitoring. DPS must be in string form (ex: '5001:1:0').
`'COMM TO-[DPS],[Timeout]'`	Set the device communication timeout to Timeout. Timeout is in 1/10-second increments. Sending a value of zero will disable the device communication timeout and the registration of the device communicating parameter.
`'DEV PWR-[DPS],[State]'`	Set the device power state. DPS must be in string form (ex: '5001:1:0'). State must be 1 or 0.
`'POWER FAIL ON-[DPS]'`	Set power failure detection on. DPS must be in string form (ex: '5001:1:0').
`'POWER FAIL OFF-[DPS]'`	Set power failure detection off. DPS must be in string form (ex: '5001:1:0').
`'VERSION'`	Send version information to master debug port (master messaging)

Strings

RMSBasicDeviceMod listens for the following strings from the **vdvModule** device.

RMSBasicDeviceMod - Strings and Descriptions	
`'POWER=[Power State]'`	Set the system power state. [PowerState] should be 0 for off and 1 for on. **Example:** `'POWER=1'`
`'MODEL=[Model]'`	Set the model number.
`'MANUFACTURER=[Manufacturer]'`	Set the manufacturer name.

Any string received from the physical device (dvProj) is an indication that the device is communicating.

As long as a string is received within the period set by the communication timeout command, the module will notify RMS that the device is communicating. See the Communication Timeout command for more details.

NOTE

If using a Duet module, strings from the physical device are captured inside the Duet module are not bubbled up to the STRING data event.
For Duet devices, instead of monitoring string traffic on physical device, Channel 252 (DATA_INITIALIZED) is used to monitor device communication status.

Channels

RMSBasicDeviceMod watches for these channels on the **vdvModule** device.

RMSBasicDeviceMod - Channels	
9	toggle System Power State
27	Set system power to ON
28	Set system power to OFF
252	Device Communication/Initialized Status (Duet Modules)
254	Power failure
255	Power Status

Module Definition

```
DEFINE_MODULE 'RMSBasicdeviceMod' mdBasicDev1(vdvModule,
                                              dvDevice,
                                              vdvRMSEngine)
```

Where `mdlBasicDev1` is a unique module name.

• `vdvModule:`	A virtual device for communicating to RMSBasicDeviceMod. This can be the same virtual device used to communicate with an InConcert module or a physical device. If controlling the device using an IR device, pass the physical device to this parameter.
• `dvDevice:`	A physical device or socket device to which the virtual device is connected.
• `vdvRMSEngine:`	A virtual device for communicating to RMSEngineMod module.

Touch Panel Pages

This module requires no pages.

RMSSldProjMod Module

- Commands
- Channels
- Module Definition

Commands

RMSSldProjMod listens for the following commands from the vdvRMSEngine device.

RMSSldProjMod - Commands and Descriptions	
`'DEV INFO-[DPS],[Name],[Man],[Model]'`	Set device information for device monitoring. DPS must be in string form (ex: '5001:1:0').
`'DEV NAME-[DPS],[Name]'`	Set device name for device monitoring. DPS must be in string form (ex: '5001:1:0').
`'VERSION'`	Send version information to master debug port (master messaging).

Channels

RMSSldProjMod watches for these channels on the **dvSldProj** device.

RMSSldProjMod - Channels	
5	Slide Projector Power State

Module Definition

```
DEFINE_MODULE 'RMSSldProjMod' mdlSldProj1(dvSldProj,
                                          vdvRMSEngine)
```

Where `mdlSldProj1` is a unique module name.

- `dvSldProj:` A physical device to which the slide projector is connected.
- `vdvRMSEngine:` A virtual device for communicating to RMSEngineMod module.

Touch Panel Pages

This module requires no pages.

RMSSrcUsageMod Module

- Commands
- Channels
- Module Definition

Commands

RMSSrcUsageMod listens for the following commands from the vdvRMSEngine device.

RMSSrcUsageMod - Commands and Descriptions	
`'MULTISOURCE-[State]'`	Set the multi-source tracking state. State can be ON or OFF. The default is OFF.
`'VERSION'`	Send version information to master debug port (master messaging)

Channels

RMSSrcUsageMod watches for these channels on the vdvConnectLinx device.

RMSSrcUsageMod - Channels			
1001	Power On	**1021**	Rack Computer Select
1002	Power Off	**1022**	Aux PC Select
1011	VCR Select	**1023**	Aux Video Select
1012	HI8 VCR Select	**1024**	Slide Select
1013	Umatic VCR Select	**1025**	Digital Media Player Select
1014	DVD Select	**1041**	Music Select
1015	Laser Disc Select	**1042**	CD Select
1016	TV/DVD/Cable Select	**1043**	Cassette Select
1017	Video Conference Select	**1044**	DAT Select
1018	Document Camera Select	**1045**	Mini Disc Select
1019	Room Camera Select	**1046**	Aux Audio Select
1020	Whiteboard Select	**1047**	Digital Audio Player Select

RMSSrcUsageMod will also listen for i!-ConnectLinx registration of custom actions and attempt to determine if they represent source selects.

Any custom action registered with a name that starts with "Select " is assumed to be a custom source. RMSSrcUsageMod will register this source name with the RMS server and treat the associated channel as a source select channel.

For more information on i!-ConnectLinx programming, refer to the i!-ConnectLinx help file installed with i!-ConnectLinx as part of the RMS SDK.

Module Definition

```
DEFINE_MODULE 'RMSSrcUsageMod' mdlSrcUsage(vdvRMSEngine,
                                           vdvCLActions)
```

Where `mdlSrcUsage` is a unique module name.

- `vdvRMSEngine:` A virtual device for communicating to RMSEngineMod module.
- `vdvCLActions:` A i!-ConnectLinx device number for monitoring source selection.

Touch Panel Pages

This module requires no pages.

Anterus Duet Module

The purpose of this module is to monitor and manage Anterus RFID Tag readers and Anterus RFID Tags. If using RFID tracking in conjunction with RMS, this module is required along with the RMSRFIDTracking NetLinx module.

The Anterus Duet module is not included in the RMS SDK distribution but may be acquired directly from the AMX website.

For programming details on the Anterus Duet module, please consult the documentation provided with the module.

NOTE

Only one instance of the Anterus Duet Module should be defined in your NetLinx program.
This module is intended to operate as a single instance and does not support multiple instances on a single NetLinx master.
Multiple instances of RMS can communicate with a single instance of the Anterus Duet Module using the RMSRFIDTrackingMod-Multi module.
For more information see the RMSRFIDTrackingMod-Multi Module section on page 41

Module Definition

```
DEFINE_MODULE 'AMX_Anterus_Comm_dr1_0_0' mdlAnterusDuetMod(vdvAnterusGateway,
dvAnterusReader1)
```

- `vdvAnterusGateway:` A virtual device for communicating to the Anterus RFID Module
- `dvAnterusReader1:` The Anterus RFID reader device for the first reader instance.

NOTE

Additional RFID readers must be registered to the Anterus Duet module using the 'PROPERTY-Identifiers' command. Please consult the Anterus Duet module documentation for more information.

i!-ConnectLinx

Overview

i!-ConnectLinx™ is a framework that allows you to expose NetLinx™ actions that can be utilized by other user interfaces or processes outside the NetLinx Control System. For instance, i!-ConnectLinx can be programmed to expose source select functions and i!-ConnectLinx compatible technologies, such as RMS, can use this information to allow the source selects to be executed as a scheduled event.

i!-ConnectLinx also provides a mechanism to request actions to be executed on the NetLinx Control System. Once a process outside the NetLinx Control System has obtained the action list, the process can then make a request to i!-ConnectLinx to execute that action. i!-ConnectLinx handles this request and makes this request available to the NetLinx program for execution.

i!-ConnectLinxEngineMod, is the main i!-ConnectLinx module that handles exposing and executing action requests, see the *Module* section on page 66. To support i!-ConnectLinx, you simply include this module in your program, define your actions and write programming to support those actions.

The i!-ConnectLinxEngineMod module makes the list of actions available to other processes, executes their requests, and provides your program with a push when an action needs to be executed.

Using i!-ConnectLinx

Little work is required to add i!-ConnectLinx to your existing NetLinx code. i!-ConnectLinx is implemented as a NetLinx module. Adding the module definition and all its parameters to your code is all that is required.

In order to use i!-ConnectLinx, you need to program and define a series of actions in the NetLinx Control System. The key to the i!-ConnectLinx engine is the virtual device, `vdvCLActions`. For additional information reference the *Module* section on page 66. Support the actions you want executed remotely using this virtual device.

Think of the virtual device, `vdvCLActions`, as a touch panel. Normally, you write your NetLinx program to respond to certain push channels from a touch panel; i!-ConnectLinx is exactly the same. Let’s say you want to provide the user with the ability to play and stop a VCR. Imagine you have two touch panel buttons that do these functions; write code that responds to the pushes:

```
BUTTON_EVENT[TP,1]                    (* VCR Play *)
{
  PUSH:
  {
    PULSE[VCR,1]
  }
}
BUTTON_EVENT[TP,2]                    (* VCR Stop *)
{
  PUSH:
  {
    PULSE{VCR,2]
  }
}
```

To expose these actions using i!-ConnectLinx, write the same code substituting the touch panel device for your i!-ConnectLinx virtual device:

```
BUTTON_EVENT[vdvCLActions,1]          (* VCR Play *)
{
  PUSH:
  {
    PULSE[VCR,1]
  }
}
BUTTON_EVENT[vdvCLActions,2]          (* VCR Stop *)
{
  PUSH:
  {
    PULSE{VCR,2]
  }
}
```

When the i!-ConnectLinx engine gets a request to play the VCR, i!-ConnectLinx will "push" the button of the virtual device just like a user pushes a button on a touch panel. There is now only one thing left to do: Tell the user which actions are which.

In order to expose an action for execution via i!-ConnectLinx, you need to support the programming for the action, as we have just seen, and you need to tell i!-ConnectLinx what that action is.

To specify the name of an action, send a command to the i!-ConnectLinx virtual device describing the name of a given channel code. To specify the names of the actions in the above example, you would add some code like this:

```
DATA_EVENT[vdvCLActions]
{
  ONLINE:
  {
    (* Setup actions *)
    (* VCR Play *)
    SEND_COMMAND vdvCLActions,"'ADD ACTION-1,VCR Play"

    (* VCR Stop *)
    SEND_COMMAND vdvCLActions,"'ADD ACTION-2,VCR Stop' "
}
```

Once i!-ConnectLinx receives these commands, it stores this information in an XML file that can be used by i!-ConnectLinx compatible technologies to browse available actions.

In addition to specifying the name of an action, you can also supply a help string and a folder name. The help string helps a user understand the intent of the action more clearly. The folder name allows you to organize the actions in a tree view so that actions are more easily browsed.

Standard Actions

So far, i!-ConnectLinx has handled custom actions where each action is likely to be different from system to system. In the Using i!-ConnectLinx example on page 59, action 1 played the VCR. However, in another system, it is very unlikely that action 1 plays the VCR.

i!-ConnectLinx uniquely identifies each action list. Once an i!-ConnectLinx compatible technology programs itself to execute an action on a system, it also stores a copy of the system identifier from the action list. This identifier is sent to i!-ConnectLinx along with this action execution request.

If the action identifier does not match the i!-ConnectLinx system that received the request, the action is not executed. This eliminates any ambiguity that may exist, since each system's action 1 may be different.

i!-ConnectLinx supports standard actions. Standard actions are actions defined by AMX and supported natively by i!-ConnectLinx.

When adding actions to i!-ConnectLinx, it is best to use the standard action if it is available. That way, the action can be executed regardless of which system the i!-ConnectLinx compatible technology was programmed to control.

The list of standard actions are listed in the **i!-ConnectLinxStdFunctionList.xls** file. The standard actions ID are the same as the channel number used to execute the action.

For instance, VCR Select has an ID of **1011** so the programming to support this standard action is:

```
BUTTON_EVENT[vdvCLActions,1011]        (* VCR Select *)
{
  PUSH:
  {
    // Switch the projector and switcher to select the VCR
  }
}
```

To add a standard action, look up the action ID in the Standard Function List file, and send that in a send command to i!-ConnectLinx to tell it you want to support that action.

To change the above example to standard action:

1. Lookup **VCR Play** and **VCR Stop** in the Standard Function List.
2. Find their IDs. VCR Play is **1131**, and VCR Stop is **1132**.
3. Send the IDs to i!-ConnectLinx:

```
DATA_EVENT[vdvCLActions]
{
  ONLINE:
  {
    (* Setup actions *)
    (* VCR Play *)
    SEND_COMMAND vdvCLActions,"'ADD STD-1131'"

    (* VCR Stop *)
    SEND_COMMAND vdvCLActions,"'ADD STD-1132'"
}
```

Additionally, change the two `BUTTON_EVENT`s to trigger for channels **1131** and **1132** instead of 1 and 2.

There are other syntax's of the add standard action command that allow you add multiple actions at a time. The '`&`' character can be used to signify "AND" and the '`-`' character can be used to signify "through".

Since many of the standard actions are related, they can also be added by macros. A macro is a list of one or more standard actions. In the case of a VCR, the full set of transports are needed, not just Play and Stop. Also, if the VCR exists in the system then there is likely a way to select the VCR as the active source. Therefore, the "`vcr`" macro includes the VCR source select and the standard transports. To load a set of actions by macro, simply send a command to i!-ConnectLinx with the macro you want added.

Example:

```
DATA_EVENT[vdvCLActions]
{
  ONLINE:
  {
    (* Setup actions *)
    (* VCR Select and Play-Record *)
    SEND_COMMAND vdvCLActions,"'ADD MACRO-vcr'"
}
```

For a complete list of macros, see the **i!-ConnectLinxStdFunctionList.xls** file.

A common method for programming i!-ConnectLinx is to simply register standard actions and respond to the actions by "DO_PUSH"ing an existing button on the touch panel.

Example:

```
BUTTON_EVENT[vdvCLActions,1011](* VCR Select *)
{
  PUSH:
    DO_PUSH(dvTP,11) (* Button 11 on dvTP selects VCR *)
}
```

To make programming i!-ConnectLinx easier, the i!-ConnectLinxStdFunctionList.xls file includes an i!-ConnectLinx Code Generator page.

On this page, you can enter the i!-ConnectLinx device, the Touch Panel device and the Touch Panel buttons for each standard action.

The code generator will create an Include (AXI) file that contains the necessary code to register and respond to the selected actions. Optionally, the code generator can include the DEFINE_MODULE statement for i!-ConnectLinx.

Once the Include file is created, you will need to include this file in your main program with an #INCLUDE statement and make sure the i!-ConnectLinx and Touch Panel devices are defined.

See the *i!-ConnectLinxStdFunctionList.xls* file for more details.

Action Arguments

i!-ConnectLinx supports action arguments to supply additional information with each action. For instance, if you wanted to support an action to set the program volume, the user needs to supply a volume level. This is accomplished using arguments.

Each action can support zero or more arguments. Each argument can be one of the following types:

- **Number** – A single number from **–32767** to **32767**. You can define the minimum value, maximum value, desired step, and a default value. The user is presented with a text box in which to enter this number.
- **Level** – Similar to a Number argument, only the user is presented with a slider to enter the level.
- **String** – A string. You can define the minimum length, maximum length, and default value. The user is presented with a text box to enter this string.
- **Enumeration** – A list or enumeration of values from which the user may choose. The user is presented with a drop down list to choose and value from.

Each argument is numbered in the order they are added. Arguments are added by using the '`ADD NARG`', '`ADD LARG`', '`ADD SARG`', and '`ADD EARG`' commands.

When an i!-ConnectLinx compatible technology requests an action with arguments to be executed, the argument values are passed to i!-ConnectLinx.

i!-ConnectLinx then posts the argument values as levels for number and level arguments, and strings for string and enumeration arguments. These values can be retrieved by using `LEVEL_EVENT`s and `DATA_EVENT`s in your program and must be saved. Then, when an action request is triggered via a `BUTTON_EVENT`, you can retrieve these argument values and use them (as appropriate) for the action to be executed.

Each argument is provided an ID at the time it is added. The ID's start at one and are numbered sequentially to each argument as they are added. When
i!-ConnectLinx posts the argument value, it supplies the ID number as well. For numbers and levels, this ID is the level number to which the argument is posted. For strings and enumerations, this ID is included in the string that posts the argument value.

For an example, see the **i!-ConnectLinxStandardFunctionShell.axi** file.

Action Persistence and Distribution

i!-ConnectLinx stores the supported actions in a XML file called **i!-ConnectLinx.xml** located in the **doc:\user\connectlinx directory**. All action information is stored in this file. i!-ConnectLinx compatible technologies retrieve this file directly from the NetLinx Master.

It may not always be practical to keep all the i!-ConnectLinx action list files on the NetLinx Master. For instance, in a corporate environment with 20 NetLinx Masters in various conference rooms, a user outside the company needs to have direct access to each NetLinx Master through the firewall in order to download the files. Additionally, each NetLinx Master needs it's own DNS entry, so users do not have to remember an IP Address.

To simplify action list management, i!-ConnectLinx compatibly technologies support an action list index file format. This index file lists the names of various files and a URL where the file can be retrieved. This allows you to move all the action list files from the NetLinx Masters to a web server for easy retrieval. Place this index file in a directory called **connectlinx** off the root directory of the web server and name it **i!-ConnectLinx.xml**. However, it can contain links to any URL with any file name in any folder.

In the above example, the IT department might collect all the action list files and place them in the connectlinx directory of the company's web server. Each file should be renamed to reflect the room that the action list is for. Then a web developer should edit the supplied i!-ConnectLinxList.xml file to reflect the names and URL's of each of these files and rename it to i!-ConnectLinx. Now anyone can retrieve an action list for the company's system by pointing to the company's main web address and selecting a room file from the list.

If desired, the action list index file can be viewed in an HTML browser by using an eXtensible Style Language file. A web developer can make any adjustments to the XSL file so the index file has the look of the company's web site when viewed in an HTML browser. A sample XSL file, **i!-ConnectLinxList.xsl**, is supplied with i!-ConnectLinx and should be placed in the same directory on the web server as the index file.

The URL contained in the index file can point to an additional index file to allow for tree style navigation. For instance, the main file might list cities where the company has offices, which point to an index file for each city. Each city index file might contain a list of buildings and point to building index files. Then each building index file contains the list of rooms in that building and points to the actual action list for each room.

International Issues / Localization

Localization is the process by which an application is adapted to a locale, and describes a user's environment or geographical location.

i!-ConnectLinx provides the standard action name, help string, and folder names for all the standard actions. This information is built directly into the i!-ConnectLinx module. If English is not the primary language for the room, the standard action text can be changed.

The standard action text can be stored in a file called **i!-ConnectLinxStdText.xml** located in the **doc:\user\connectlinx** directory.

When a standard action is added, the text from this file is used for the action name, the help string and folder names.

The **i!-ConnectLinxStdText.xml** can be created in two ways. The **i!-ConnectLinxStdTextTemplate.xml** file can be altered directly and saved as **i!-ConnectLinxStdText.xml** in the **doc:\user\connectlinx** directory. However, this file is difficult to edit in a standard text editor so an XML file editor is recommended.

Alternatively, the **i!-ConnectLinxStdText.xml** file can be created using the **i!-ConnectLinxEngineStdTextWriter.axs** file.

To change the language:

1. Open this file in NetLinx Studio2
2. Alter the text to support the language you choose.
3. Compile and download this file to a NetLinx Master.

The **i!-ConnectLinxStdText.xml** is written out to the **doc:\user\connectlinx** directory.

Once this file has been created once, it can be FTP'd to the NetLinx Master and placed in the doc:\user\connectlinx directory. When i!-ConnectLinx starts up, the text is read from this file and used for all standard actions.

Programming

i!-ConnectLinx appears on the NetLinx bus as a NetLinx device. This device has 1 port with channels, levels, commands and strings like most other devices.

Channels

i!-ConnectLinx supports the following channels:

i!-ConnectLinx Channels	
All	Action to be executed for this action ID.

Levels

i!-ConnectLinx supports the following levels:

i!-ConnectLinx Levels	
All	Action number and level arguments

Commands

i!-ConnectLinx supports the following out-bound commands (Master to device).

The commands are sent in the standard Send_Command format:

```
SEND_COMMAND dvMP, "'SET ROOM NAME-Tesla'"
SEND_COMMAND dvMP, "'ADD MACRO-vcr'"
```

i!-ConnectLinx Commands	
`'SET ROOM INFO-[Room Name],[Room Location],[Room Owner]'`	Sets the room name, room location, and room owner to be displayed in the action list file.
`'SET ROOM NAME-[Room Name]'`	Sets the room name to be displayed in the action list file.
`'SET ROOM LOCATION-[Room Location]'`	Sets the room location to be displayed in the action list file.
`'SET ROOM OWNER-[Owner Name]'`	Sets the room owner to be displayed in the action list file.
`'ADD MACRO-[Macro Name]'`	Adds a group of standard actions. See **i!-ConnectLinxStdFunction-List.xls** for a complete list of macro names.
`'ADD STD-[ID]-[ID]&[ID]'`	Adds one or more standard actions by ID. The '&' is used for "AND" and '-' is used for "THROUGH".
`'ADD FOLDER-[Folder Name],[Parent]'`	Adds a folder to the action list. The parent specifies which parent folder the new folder is added to. If parent is not supplied, this folder is added to the root of the action list.

i!-ConnectLinx Commands (Cont.)	
`'ADD ACTION-[ID],[Action],[Help String],[Folder]'`	Adds an action. The ID and Action are required. The `Help String` appears in the action list file to help the user determine this action's function more clearly. The `Folder` is the folder in which this action is added to, and must have been previously created. If the folder is not supplied, the action is added to the root of the action list.
`'ADD NARG-[Action],[Arg Name],[Min],[Max],[Step],[Default]'`	Adds a number argument to `Action`. The `Arg Name` (Argument Name) is required. The `Min` and `Max` define the limits for this argument in the range **–32767** to **32767**. The `Step` defines the minimum step between increments/decrements. The `Default` value defines the initial value this argument is set to when the user edits this argument.
`'ADD LARG-[Action],[Arg Name],[Min],[Max],[Step],[Default]'`	Adds a level argument to `Action`. The `Arg Name` (Argument Name) is required. The `Min` and `Max` define the limits for this argument in the range **–32767** to **32767**. The `Step` defines the minimum step between increments/decrements. The `Default` value defines the initial value this argument is set to when the user edits this argument.
`'ADD SARG-[Action],[Arg Name],[Min],[Max],[Default]'`	Adds a string argument to `Action`. The `Arg Name` (Argument Name) is required. The `Min` and `Max` define the min and max length of the string. The `Default` value defines the initial value this argument is set to when the user edits this argument.
`'ADD EARG-[Action],[Arg Name],[Default],[Enum1],[Enum2]...'`	Adds an enum argument to `Action`. The `Arg Name` (Argument Name) is required. The `Default` value defines the initial value this argument is set to when the user edits this argument. `Enum1`, `Enum2`,... define the available choices for the argument.
`'GET NODE-[Name],[Start],[End]'`	Get the node description for Name including children from Start to End. Returns a PARENT string and multiple CHILD strings.
`'GET ACTION-[Name]'`	Get the action description for Name. Returns ACTION string and multiple argument strings (NARG, LARG, SARG and EARG).
`'GET UUID'`	Get the UUID for this i!-ConnectLinx. Returns a UUID string.
`'GET ROOM INFO'`	Get the room info for this i!-ConnectLinx. Returns ROOM NAME, ROOM LOCATION and, ROOM OWNER strings.
`'GET ROOM NAME'`	Get the room info for this i!-ConnectLinx. Returns a ROOM NAME string.
`'GET ROOM LOCATION'`	Get the room info for this i!-ConnectLinx. Returns a ROOM LOCATION string.
`'GET ROOM OWNER'`	Get the room info for this i!-ConnectLinx. Returns a ROOM OWNER string.
`'DEBUGON'`	Turns on debug.
`'DEBUGOF'`	Turns off debug. (Default)
`'RESET'`	Resets the action list.
`'VERSION'`	Sends version information to Master debug port (Master messaging).

Strings

i!-ConnectLinx supports the following in-bound string (device to Master).

i!-ConnectLinx Strings	
String	**Description**
`"'ARG[Argument ID]-[Argument String]'"`	Argument String for string and enum arguments for an action executed soon.
`'PARENT-[Name],[Child Count],[Parent]'`	Describes a parent node. Returned by GET NODE command.
`'CHILD[Child#]-[Name],[ChildCount],[Parent]'`	Describes a child of a node. Returned by GET NODE command.
`'ACTION-[ID],[Action],[Help String],[Folder]'`	Describes an action. Returned by GET ACTION command.
`'NARG-[Action],[Arg Name],[Min],[Max],[Step,][Default]'`	Describes a number argument to Action.
`'LARG-[Action],[Arg Name],[Min],[Max],[Step],[Default]'`	Describes a level argument to Action.
`'SARG-[Action],[Arg Name],[Min],[Max],[Default]'`	Describes a string argument to Action.
`'EARG-[Action],[Arg Name],[Default],[Enum1],[Enum2]...'`	Describes an enum argument to Action.
`'UUID-[UUID]'`	Provides the UUID for this i!-ConnectLinx. This ID can be used to identify this instance of i!-ConnectLinx from all other instances of i!-ConnectLinx.
`'ROOM NAME-[Room Name]'`	Provide the room name as displayed in the action list file.
`'ROOM LOCATION-[Room Location]'`	Provides the room location as displayed in the action list file.
`'ROOM OWNER-[Owner Name]'`	Provides the room owner as displayed in the action list file.
`'FILE WRITE'`	Notification that the i!-ConnectLinx file is being written.
`'FILE SAVED'`	Notification that the i!-ConnectLinx file is has been saved.

Module

The **i!-ConnectLinxEngineMod** Module definition code is displayed below.

```
DEFINE_MODULE 'i!-ConnectLinxEngineMod' mdlCL(vdvCLActions)
```

Where **`mdlCL`** is a unique module name.

i!-ConnectLinxEngineMod Module Parameter	
`vdvCLActions`	A virtual device number for programming NetLinx actions.

i!-ConnectLinx Standard Function List

i!-ConnectLinx Standard Function List				
Category	**ID**	**Action**	**ID**	**Action**
System Controls	1001	Power On	1021	Select Rack Computer
	1002	Power Off	1022	Select Aux PC Input
	1011	Select VHS	1023	Select Aux Vid Input
	1012	Select Hi-8	1024	Select Slide (slide to video)
	1013	Select Umatic	1025	Select Digital Media Player
	1014	Select DVD	1041	Select Music (AM/FM Tuner or DSS)
	1015	Select Laser Disc	1042	Select CD Player
	1016	Select TV-DSS-Cable	1043	Select Cassette
	1017	Select Video Conference	1044	Select DAT
	1018	Select Document Camera	1045	Select Minidisc
	1019	Select Room Camera	1046	Select Aux Audio Input
	1020	Select Whiteboard	1047	Select Digital Audio Player
Lighting	1061	Lights All Off	1064	Lights Presentation Mode
	1062	Lights All On	1065	Lights Conference Mode
	1063	Lights Meeting Mode		
Room Volume	1071	Program Mute	1075	Program 50%
	1072	Program Unmute	1076	Program 75%
	1073	Program 0%	1077	Program 100%
	1074	Program 25%	1078	Program Set Volume
Speech Volume	1081	Speech Mute	1085	Speech 50%
	1082	Speech Unmute	1086	Speech 75%
	1083	Speech 0%	1087	Speech 100%
	1084	Speech 25%	1088	Speech Set Volume
Screen	1091	Screen Up		
	1092	Screen Down		
Drapes	1093	Drapes Open		
	1094	Drapes Close		
Blinds	1095	Blinds Open		
	1096	Blinds Close		
Lift	1097	Lift Up		
	1098	Lift Down		
Video Projector/Display	1101	Display Power On	1104	Display Picture Unmute
	1102	Display Power Off	1105	Display Picture Mute Toggle
	1103	Display Picture Mute		
VCR - VHS	1131	VCR Play	1135	VCR Rewind
	1132	VCR Stop	1136	VCR Search Fwd
	1133	VCR Pause	1137	VCR Search Rev
	1134	VCR Fast Forward	1138	VCR Record
VCR - Hi-8	1161	Hi-8 Play	1165	Hi-8 Rewind
	1162	Hi-8 Stop	1166	Hi-8 Search Fwd
	1163	Hi-8 Pause	1167	Hi-8 Search Rev
	1164	Hi-8 Fast Forward	1168	Hi-8 Record

i!-ConnectLinx Standard Function List				
Category	**ID**	**Action**	**ID**	**Action**
VCR - Umatic	1191	Umatic Play	1195	Umatic Rewind
	1192	Umatic Stop	1196	Umatic Search Fwd
	1193	Umatic Pause	1197	Umatic Search Rev
	1194	Umatic Fast Forward	1198	Umatic Record
DVD	1221	DVD Play	1226	DVD Search Fwd
	1222	DVD Stop	1227	DVD Search Rev
	1223	DVD Pause	1228	DVD Go To Track
	1224	DVD Skip Forward	1229	DVD Go To Chapter
	1225	DVD Skip Back	1230	DVD Go To Time
Laser Disc Player	1251	LDP Play	1256	LDP Search Fwd
	1252	LDP Stop	1257	LDP Search Rev
	1253	LDP Pause	1258	LDP Go To Track
	1254	LDP Skip Forward	1259	LDP Go To Chapter
	1255	LDP Skip Back	1260	LDP Go To Time
TV - DSS - Cable	1281	TV Channel Up	1305	TV-FOX
	1282	TV Channel Down	1306	TV-UPN
	1283	TV Go To Channel	1307	TV-WB
	1291	TV Keypad 0	1308	TV-CNBC
	1292	TV Keypad 1	1309	TV-CNN
	1293	TV Keypad 2	1310	TV-CNNH
	1294	TV Keypad 3	1311	TV-MSNBC
	1295	TV Keypad 4	1312	TV-NEWSINT
	1296	TV Keypad 5	1313	TV-BLOOM
	1297	TV Keypad 6	1314	TV-CNNFN
	1298	TV Keypad 7	1315	TV-FNC
	1299	TV Keypad 8	1316	TV-CSPAN
	1300	TV Keypad 9	1317	TV-ESPN
	1301	TV-ABC	1318	TV-TBS
	1302	TV-CBS	1319	TV-TECHTV
	1303	TV-NBC	1320	TV-TWC
	1304	TV-PBS		
Video Conference	1341	Vconf Hang-up Video	1354	Vconf Keypad 3
	1342	Vconf Dial Video	1355	Vconf Keypad 4
	1343	Vconf Hang-up Audio	1356	Vconf Keypad 5
	1344	Vconf Dial Audio	1357	Vconf Keypad 6
	1345	Vconf Speed Dial	1358	Vconf Keypad 7
	1346	Vconf Privacy On	1359	Vconf Keypad 8
	1347	Vconf Privacy Off	1360	Vconf Keypad 9
	1351	Vconf Keypad 0	1361	Vconf Keypad *
	1352	Vconf Keypad 1	1362	Vconf Keypad #
	1353	Vconf Keypad 2		
Document Camera	1371	Doccam Upper Lights		
	1372	Doccam Lower Lights		
	1373	Doccam Preset		

i!-ConnectLinx Standard Function List				
Category	**ID**	**Action**	**ID**	**Action**
Room Camera	1401	Camera Preset		
Rack Computer	1461	Computer Play	1466	Computer Last Slide
	1462	Computer Stop	1467	Computer First Slide
	1463	Computer Pause	1468	Computer Goto Slide
	1464	Computer Next	1469	Computer Goto Presentation
	1465	Computer Prev		
Aux PC Input	1491	Laptop Play	1496	Laptop Last Slide
	1492	Laptop Stop	1497	Laptop First Slide
	1493	Laptop Pause	1498	Laptop Goto Slide
	1494	Laptop Next	1499	Laptop Goto Presentation
	1495	Laptop Prev		
Slide	1521	Slide Power On	1524	Slide Previous
	1522	Slide Power Off	1525	Slide Go To Slide
	1523	Slide Next		
Digital Media Player	1551	Digital Media Play	1555	Digital Media Skip Back
	1552	Digital Media Stop	1556	Digital Media Search Fwd
	1553	Digital Media Pause	1557	Digital Media Search Rev
	1554	Digital Media Skip Forward		
Select Music	1701	Music - 60's	1704	Music - Country
	1702	Music - 70's	1705	Music - Hits
	1703	Music - 80's		
Select Music	1706	Music - Rock	1712	Music - Classical
	1707	Music - Urban	1713	Music - News
	1708	Music - Jazz+Blues	1714	Music - Sports
	1709	Music - Dance	1715	Music - Comedy
	1710	Music - Latin	1716	Music - Talk
	1711	Music - World		
CD Player	1731	CD Play	1736	CD Search Fwd
	1732	CD Stop	1737	CD Search Rev
	1733	CD Pause	1738	CD Go To Track
	1734	CD Skip Forward	1739	CD Go To Disc
	1735	CD Skip Back		
Cassette	1761	Cass A Play	1771	Cass B Play
	1762	Cass A Stop	1772	Cass B Stop
	1763	Cass A Pause	1773	Cass B Pause
	1764	Cass A FastForward	1774	Cass B FastForward
	1765	Cass A Rewind	1775	Cass B Rewind
	1766	Cass A Record	1776	Cass B Record
Digital Audio Tape (DAT)	1791	DAT Play	1795	DAT Rewind
	1792	DAT Stop	1796	DAT Search Fwd
	1793	DAT Pause	1797	DAT Search Rev
	1794	DAT Fast Forward	1798	DAT Record

i!-ConnectLinx Standard Function List				
Category	**ID**	**Action**	**ID**	**Action**
Mini Disc	1821	MD Play	1826	MD Search Fwd
	1822	MD Stop	1827	MD Search Rev
	1823	MD Pause	1828	MD Record
	1824	MD Fast Forward	1829	MD Go To Track
	1825	MD Rewind		
Audio Conference	1851	Aconf Hangup	1865	Aconf Keypad 4
	1852	Aconf Dial	1866	Aconf Keypad 5
	1853	Aconf Speed Dial	1867	Aconf Keypad 6
	1854	Aconf Privacy on	1868	Aconf Keypad 7
	1855	Aconf Privacy off	1869	Aconf Keypad 8
	1861	Aconf Keypad 0	1870	Aconf Keypad 9
	1862	Aconf Keypad 1	1871	Aconf Keypad *
	1863	Aconf Keypad 2	1872	Aconf Keypad #
	1864	Aconf Keypad 3		
Digital Audio Player	1881	Digital Audio Play	1885	Digital Audio Skip Back
	1882	Digital Audio Stop	1886	Digital Audio Search Fwd
	1883	Digital Audio Pause	1887	Digital Audio Search Rev
	1884	Digital Audio Skip Forward		

Multiple RMS Instances

Overview

RMS supports multiple instances loaded on 1 NetLinx master. There can either be 4 instances controlling and monitoring as many devices that may be physically connected to one master, or 12 instances scheduling and displaying with no devices other than touch panels. There is an example of multiple instance programming included in the RMS SDK, **RMS (Multi-Instance).apw**.

Each master instance must declare it's own communications socket, RMS Engine device, touch panels and specific devices for the instance.

Only one set of i!-ConnectLinx functions is necessary in multiple RMS instances. It is possible to control unique functions within the multi-instanced rooms but source usage and system power only works for one room.

Below is an example of two instances from RMSMain-4Systems, which is included in the SDK:

```
 (* RMS INSTANCE #1 *)
dvTPMain_1         = 10001:1:0       (* RMS Main Touch Panel *)
dvTPWelcome_1      = 10002:1:0       (* RMS Welcome Touch Panel *)

dvRMSSocket_1      = 0:3:0           (* RMS IP Socket *)
vdvRMSEngine_1     = 33001:1:0       (* RMS Virtual Device *)

vdvProjModule_1    = 33002:1:0       (* Projector Virtual Device *)
dvProj_1           = 5001:1:0        (* Projector Real Device *)

vdvSWTModule_1     = 33003:1:0       (* Switcher Virtual Device *)
dvSWT_1            = 5001:2:0        (* Switcher Real Device *)

vdvVCRModule_1     = 33004:1:0       (* VCR Virtual Device *)
dvVCR_1            = 5001:9:0        (* VCR Real Device *)

dvRELAY_1          = 5001:8:0        (* NI-3000 Relay *)

 (* RMS INSTANCE #2 *)
dvTPMain_2         = 10003:1:0       (* RMS Main Touch Panel *)
dvTPWelcome_2      = 10004:1:0       (* RMS Welcome Touch Panel *)

dvRMSSocket_2      = 0:4:0           (* RMS IP Socket *)
vdvRMSEngine_2     = 33011:1:0       (* RMS Virtual Device *)

vdvProjModule_2    = 33012:1:0       (* Projector Virtual Device *)
dvProj_2           = 5002:1:0        (* Projector Real Device *)

vdvSWTModule_2     = 33013:1:0       (* Switcher Virtual Device *)
dvSWT_2            = 5002:2:0        (* Switcher Real Device *)

vdvVCRModule_2     = 33014:1:0       (* VCR Virtual Device *)
dvVCR_2            = 5002:9:0        (* VCR Real Device *)

dvRELAY_2          = 5002:8:0        (* NI-3000 Relay *)
```

The rest of the multi instance information is in the include file **RMSMain-Multi.axi** in the SDK.

Declare a Dev Array of RMS Engine Instances

In the include, declare a dev array of RMS Engine instances using the previously declared engines, e.g., vdvRMSEngine_1, a dev array for each master instances of touch panels (one each for main and welcome panels), and dev arrays for keyboards for each panel (the main ones only). See below:

```
// create an array of all RMS engines
VOLATILE DEV vdvRMSEngines[] = {vdvRMSEngine_1, vdvRMSEngine_2}

// create a touch panel device array for all main touch panels for each RMS
instance
VOLATILE DEV dvRMSTP_1[] = {dvTPMain_1}
VOLATILE DEV dvRMSTP_2[] = {dvTPMain_2}


// create a touch panel device array for all welcome touch panels for each RMS
instance
VOLATILE DEV dvRMSTPWelcome_1[] = {dvTPWelcome_1}
VOLATILE DEV dvRMSTPWelcome_2[] = {dvTPWelcome_2}
```

Module Defining

Each instance must define it's own modules, which directly affects how many instances can be run (memory constraint). The example uses two instances.

```
// 1st RMS instance...
// RMSSrcUsageMod - Tracks equipment usage
DEFINE_MODULE 'RMSSrcUsageMod' mdlSrcUsage_1(vdvRMSEngine_1, vdvCLActions)

// Switcher
DEFINE_MODULE 'RMSBasicDeviceMod' mdlBasicDev_1(vdvSWTModule_1, dvSWT_1,
vdvRMSEngine_1)

// VCR
DEFINE_MODULE 'RMSTransportMod' mdlXport_1(vdvVCRModule_1, dvVCR_1,
vdvRMSEngine_1)

// Display
DEFINE_MODULE 'RMSProjectorMod' mdlProj_1(vdvProjModule_1, dvProj_1,
vdvRMSEngine_1)

// RMSEngineMod - The RMS engine. Requires i!-ConnectLinxEngineMod.
DEFINE_MODULE 'RMSEngineMod' mdlRMSEng_1(vdvRMSEngine_1, dvRMSSocket_1,
vdvCLActions)

// RMSUIMod - The RMS User Interface. Requires KeyboardMod.
// Channel And Variable Text Code Defined Inside The Module
DEFINE_MODULE 'RMSUIMod' mdlRMSUI_1(vdvRMSEngine_1, dvRMSTP_1,
dvRMSTPWelcome_1, RMS_MEETING_DEFAULT_SUBJECT, RMS_MEETING_DEFAULT_MESSAGE)
```

```
// 2nd RMS instance...
// RMSSrcUsageMod - Tracks equipment usage
DEFINE_MODULE 'RMSSrcUsageMod' mdlSrcUsage_2(vdvRMSEngine_2, vdvCLActions)

// Switcher
DEFINE_MODULE 'RMSBasicDeviceMod' mdlBasicDev_2(vdvSWTModule_2, dvSWT_2,
vdvRMSEngine_2)

// VCR
DEFINE_MODULE 'RMSTransportMod' mdlXport_2(vdvVCRModule_2, dvVCR_2,
vdvRMSEngine_2)


// Display
DEFINE_MODULE 'RMSProjectorMod' mdlProj_2(vdvProjModule_2, dvProj_2,
vdvRMSEngine_2)

// RMSEngineMod - The RMS engine. Requires i!-ConnectLinxEngineMod.
DEFINE_MODULE 'RMSEngineMod' mdlRMSEng_2(vdvRMSEngine_2, dvRMSSocket_2,
vdvCLActions)

// RMSUIMod - The RMS User Interface. Requires KeyboardMod.
// Channel And Variable Text Code Defined Inside The Module
DEFINE_MODULE 'RMSUIMod' mdlRMSUI_2(vdvRMSEngine_2, dvRMSTP_2, dvRMSTPWelcome_2,
RMS_MEETING_DEFAULT_SUBJECT, RMS_MEETING_DEFAULT_MESSAGE)
```

NOTE

Define modules for whatever is running on each master's instance.

The function RMSDevMonRegisterCallback() has a call to DevMonRegisterHelper(…) for each master instance and their respective devices. These two functions have to be modified to register instances and devices as desired by the user. The example registers the same devices for each instance.

Stacking and Handling Events

Because the rest of the code (common include, and others) is single instance driven, there is a switch going on for every event. Determine who is raising the event and then switch the global RMSEngine to the engine the event is coming from. The example uses a shared function SetEngine(engine instance) to do the dirty work. Each event checks to see who raised it and sets the correct instance for the rest of the work.

Example:

```
 (*********************************************)
 (* DATA: Main Touch Panel                   *)
 (* DATA: TP Battery Level                   *)
 (*********************************************)
DATA_EVENT[dvTPMain_1]
DATA_EVENT[dvTPMain_2]
DATA_EVENT[dvTPMain_3]
DATA_EVENT[dvTPMain_4]
{
    ONLINE:
    {
     STACK_VAR INTEGER nInstance

    // set the current RMS engine instance based on the TP device
    IF(DATA.DEVICE = dvTPMain_1)
      SetEngine(vdvRMSEngine_1)
    ELSE IF(DATA.DEVICE = dvTPMain_2)
      SetEngine(vdvRMSEngine_2)
    ELSE IF(DATA.DEVICE = dvTPMain_3)
      SetEngine(vdvRMSEngine_3)
    ELSE IF(DATA.DEVICE = dvTPMain_4)
      SetEngine(vdvRMSEngine_4)
    // get the current RMS engine instance index
    nInstance = GetRMSEngineInstance()
    }
}
```

Multi-Instancing RFID Device Tracking in RMS

This section describes the steps necessary to support RFID tracking in a NetLinx program with multiple instance of RMS.

1. DEFINE_DEVICE

A single virtual device is needed for the Anterus Duet module and a device definition is needed for each RFID reader device.

NOTE *Only a single instance of the Anterus Duet module is needed.*

```
vdvAnterusGateway = 41001:1:0 (* Duet RFID Virtual Device (global instance) *)

  // RMS INSTANCE #1
  dvAnterusReader1 = 84:1:0 (* AxLink Anterus RFID Reader #1 *)
  dvAnterusReader2 = 85:1:0 (* AxLink Anterus RFID Reader #2 *)
  dvAnterusReader3 = 86:1:0 (* AxLink Anterus RFID Reader #3 *)
  dvAnterusReader4 = 87:1:0 (* AxLink Anterus RFID Reader #4 *)

  // RMS INSTANCE #2
  dvAnterusReader5 = 88:1:0 (* AxLink Anterus RFID Reader #5 *)
  dvAnterusReader6 = 89:1:0 (* AxLink Anterus RFID Reader #6 *)
  dvAnterusReader7 = 90:1:0 (* AxLink Anterus RFID Reader #7 *)
  dvAnterusReader8 = 91:1:0 (* AxLink Anterus RFID Reader #8 *)

  // RMS INSTANCE #3
  dvAnterusReader9 = 92:1:0 (* AxLink Anterus RFID Reader #9  *)
  dvAnterusReader10 = 93:1:0 (* AxLink Anterus RFID Reader #10 *)
  dvAnterusReader11 = 94:1:0 (* AxLink Anterus RFID Reader #11 *)
  dvAnterusReader12 = 95:1:0 (* AxLink Anterus RFID Reader #12 *)

  // RMS INSTANCE #4
  dvAnterusReader13 = 96:1:0 (* AxLink Anterus RFID Reader #13 *)
  dvAnterusReader14 = 97:1:0 (* AxLink Anterus RFID Reader #14 *)
  dvAnterusReader15 = 98:1:0 (* AxLink Anterus RFID Reader #15 *)
  dvAnterusReader16 = 99:1:0 (* AxLink Anterus RFID Reader #16 *)
```

2. DEFINE_CONSTANT

Define a string array (two-dimensional array) constant for the list of RFID reader address labels for each instance of RMS.

The third parameter of the 'RMSRFIDTrackingMod-Multi' module is used to identify which Anterus RFID readers an instance of the 'RMSRFIDTrackingMod-Multi' module should monitor and relay RFID tag status to the RMS server. Each RFID reader configured in Anterus supports a user defined reader address label. (This address label is separate from the reader's physical AxLink DPS address).

By default, the Anterus module will automatically label each RFID reader's address label field using its AxLink DPS address. Once a RFID reader address label is recorded in Anterus, that label is persisted unless the RFID reader is deleted or manually renamed via the 'RFID Configuration Manager' web page hosted on the NetLinx master.

Although Anterus does provide a default RFID reader address label, it is a user customizable field and can be overridden and manually defined via the 'RFID Configuration Manager' web page hosted on the NetLinx master.

It is important to note that if using the multi-instanced 'RMSRFIDTrackingMod-Multi' module, these RFID reader address labels are required to be known by the module and the module will not work properly if the RFID Reader address labels are changed by the user and are no longer in sync with the labels defined in code.

```
//  IMPORTANT !!
//
//  By default when Anterus creates a RFID reader address
//  label, it uses the fully-qualifed
//  D:P:S with the actual system number.
//  The real system number, not '0', should be used
//  in the RFID reader address labels listed below.
//

// RMS INSTANCE #1
CHAR RFID_READER_ADDRESS_LABELS_1[][15] =  { '84:1:1',
                                             '85:1:1',
                                             '86:1:1',
                                             '87:1:1' }
// RMS INSTANCE #2
CHAR RFID_READER_ADDRESS_LABELS_2[][15] =  { '88:1:1',
                                             '89:1:1',
                                             '90:1:1',
                                             '91:1:1' }
// RMS INSTANCE #3
CHAR RFID_READER_ADDRESS_LABELS_3[][15] =  { '92:1:1',
                                             '93:1:1',
                                             '94:1:1',
                                             '95:1:1' }
// RMS INSTANCE #4
CHAR RFID_READER_ADDRESS_LABELS_4[][15] =  { '96:1:1',
                                             '97:1:1',
                                             '98:1:1',
                                             '99:1:1' }
```

3. DEFINE_MODULE

Define modules. A single module definition is needed for the Anterus Duet module.

```
   // Anterus RFID Duet Module
   DEFINE_MODULE 'AMX_Anterus_Comm_dr1_0_0'
 mdlAnterusDuetMod(vdvAnterusGateway,dvAnterusReaders[1])
```

Multiple module definitions are needed for the RFIDTrackingMod-Multi module. A module definition is needed for each RMS instance supporting RFID device tracking.

- The module parameters include: the global Anterus Duet module virtual device,
- the unique RMS Engine virtual device instance and
- the specific RFID reader address label array containing the RFID readers associated with this instance of RMS.

```
 // RMS INSTANCE #1
    DEFINE_MODULE 'RMSRFIDTrackingMod-Multi'  mdlRMSRFIDTracking_1
    (vdvAnterusGateway,vdvRMSEngine_1,
    RFID_READER_ADDRESS_LABELS_1)


 // RMS INSTANCE #2
    DEFINE_MODULE 'RMSRFIDTrackingMod-Multi' mdlRMSRFIDTracking_2(vdvAnterusGateway,
    vdvRMSEngine_2,
    RFID_READER_ADDRESS_LABELS_2)


 // RMS INSTANCE #3
    DEFINE_MODULE 'RMSRFIDTrackingMod-Multi' mdlRMSRFIDTracking_3
    (vdvAnterusGateway,vdvRMSEngine_3,
    RFID_READER_ADDRESS_LABELS_3)


 // RMS INSTANCE #4
    DEFINE_MODULE 'RMSRFIDTrackingMod-Multi' mdlRMSRFIDTracking_4
    (vdvAnterusGateway,vdvRMSEngine_4,
    RFID_READER_ADDRESS_LABELS_4)
```

4. DATA_EVENT

A data event is needed when the Anterus Duet module virtual device comes online to identify all of the RFID reader devices to the Duet module.

See the Anterus documentation for more information on the '*PROPERTYIdentifiers*' command.

```
   DATA_EVENT[vdvAnterusGateway]
   {
     ONLINE:
     {
       // send the IDENTIFIERS property to the Anterus
       // Duet module to identify additional reader devices
       SEND_COMMAND vdvAnterusGateway,"'PROPERTY-
 Identifiers,84;85;86;87;88;89;90;91;92;93;94;95;96;97;98;99'"

       // send the REINT command to the Anterus Duet
       // module to re-initialize with the updated reader addresses
       SEND_COMMAND vdvAnterusGateway,"'REINIT'"
     }
   }
```

The multi-instance sample code included in the RMS SDK includes all the necessary code to implement the Anterus Duet module and RFID tracking with multiple instances of RMS.

In the RMSMain-4Systems.AXS file, search and find the #DEFINE RMS_RFID_ENABLED statement.

If this statement is un-commented, then the multi-instance sample code will compile including all the necessary RFID implementation code.

The implementation code can be found in the RMSMain-4Systems.AXS and RMSMain-Multi.AXI files.

It's Your World - Take Control™

93-3002-04 REV R