

# Rimage Client API™ Programming Guide

For Rimage Software Development Kit 8.1



#### Corporate Headquarters:

Rimage Corporation  
7725 Washington Avenue South  
Minneapolis, MN 55439  
USA  
800-553-8312 (toll free US)  
Service: +1 952-946-0004 (Asia/Pacific,  
Mexico/Latin America)  
Fax: +1 952-944-6956

#### European Headquarters:

Rimage Europe GmbH  
Albert-Einstein-Str. 26  
63128 Dietzenbach Germany  
Tel: +49-(0) 6074-8521-0  
Fax: +49-(0) 6074-8521-100

Rimage Corporation reserves the right to make improvements to the equipment and software described in this document at any time without any prior notice. Rimage Corporation reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Rimage Corporation to notify any person or organization of such revisions or changes.

This document may contain links to web sites that were current at the time of publication, but may have moved or become inactive since. This document may contain links to sites on the Internet that are owned and operated by third parties. Rimage Corporation is not responsible for the content of any such third-party site.

©2011, Rimage Corporation

Rimage® is a registered trademark of the Rimage Corporation. SDK™ is a trademark of the Rimage Corporation. Dell® is registered trademark of Dell Computer Corporation. FireWire® is a registered trademark of Apple Computer, Inc.

All other trademarks and registered trademarks are the property of their respective owners.



# Contents

<b>Important Information .....</b>	<b>1</b>
Support Information .....	1
Learn More Online .....	1
<b>Introduction .....</b>	<b>3</b>
Overview .....	3
<b>Client API Design.....</b>	<b>5</b>
Client API Use of XML.....	7
XML Encoding Format for Production Server and Imaging Server .....	7
Rimage DTDs .....	7
DTD Location .....	8
DTD Versions .....	8
Client ID and Order ID Uniqueness Rules .....	8
Client API Programming Class Definitions .....	9
System-Related Operations Group .....	9
Server-Related Operations Group .....	9
Order-Related Operations Group .....	9
<b>System Management .....</b>	<b>11</b>
Connect to the System .....	11
SystemManager.Connect() .....	11
SystemManager.Disconnect() .....	11
Sample Code .....	11
Start/End Session using .NET with C# .....	11
Start/End Session using Java .....	12
Start/End Session using C++ .....	12
Start/End Session using C .....	13
Start/End Session using VB 6 .....	13
Listen for System Events.....	14
SystemManager.listenForSystemStatus(SystemListener) .....	14
SystemListener.onSystemStatus() .....	14
SystemListener.onSystemException() .....	14
SystemListener.onClusterCreated() .....	14
SystemManager.onClusterDeleted() .....	14
SystemManager.removeSystemListener() .....	14
Sample Code .....	14
Listen for System Events using .NET with C# .....	14
Listen for System Events using Java .....	15
Listen for System Events using C++ .....	16
Listen for System Events using C .....	16
Listen for System Events using VB 6 .....	17
<b>Server Management .....</b>	<b>19</b>
Listening for Server Events .....	19
ServerManager.listenForServerEvents(ServerEventListener) .....	19
ServerManager.removeServerEventListener() .....	19
Sample Code .....	19
Listen for Server Events using .NET with C# .....	19
Listen for Server Events using Java .....	21
Listen for Server Events using C++ .....	22
Listen for Server Events using C .....	24
Listen for Server Events using VB 6 .....	24
Synchronous Server Methods .....	25
Sample Code .....	25
Server Methods using .NET with C# .....	25

Server Methods using Java .....	26
Server Methods using C++ .....	27
Server Methods using C.....	27
Server Methods using VB 6 .....	28
<b>Order Management.....</b>	<b>29</b>
Submit Orders.....	29
Order Management Methods.....	30
OrderDescription Parameter .....	30
XMLOrder Parameter .....	30
OrderStatusListener .....	30
OrderDescription .....	31
OrderDescription Object as a Return Value .....	31
Cancel an Order in Process .....	31
Recover Orders .....	32
OrderDescription Base Class .....	32
ImageOrderDescription Sub Class .....	33
ProductionOrderDescription Sub Class .....	33
Streaming .....	34
Spanning .....	35
Order Management Sample Code.....	36
Order Management using .NET with C# .....	36
Order Management using Java .....	38
Order Management using C++ .....	39
Order Management using C.....	41
Order Management using VB 6 .....	42
<b>Server Status and Control Protocol .....</b>	<b>45</b>
Server Command Synchronization .....	45
Password Protection on Commands .....	46
Production Server Commands .....	46
Command Summary .....	47
Command Reply .....	48
Command Details .....	49
Imaging Server Commands.....	52
Command Summary .....	52
Command Reply .....	53
Command Details .....	54
<b>Deployment.....</b>	<b>55</b>
Java Deployment.....	55
Build Information .....	55
Required Files .....	55
.NET Deployment .....	55
Build Information .....	55
Required .NET Assembly Files .....	55
C / C++ / VB 6 Deployment.....	55
Build Information .....	55
Required Linker Options.....	56
Required Files and Folders .....	56
Required DLL Files (Non-Unicode) .....	56
Required DLL Files (Unicode).....	56
Microsoft visual C++ 2008 SP1 Redistributable Pack is Required. ....	56
Required LIB Files (Non-Unicode) .....	56
Required LIB Files (Unicode) .....	56
Required Include Directories.....	56
Required #include Statements .....	56



Optional files .....	57
64 bit deployment .....	57
<b>Appendix A – Sample Source Code Projects .....</b>	<b>59</b>
<b>Appendix B – Sample XML Documents .....</b>	<b>61</b>
Image Order Samples.....	61
ISO L2 with Editlist Image Order.....	61
ISO L2 from Parent Folder Image Order .....	61
RockRidge Image Order .....	62
Production Order Samples.....	63
Audio Production Order .....	64
Blue Book Production Order .....	65
Mode 1 Production Order .....	66
Print Only Production Order .....	67
Data Disc Production Order .....	67
Order Status Samples.....	68
Image Order Status .....	68
Production Order Status .....	68
Spanning XML Samples .....	68
Image Order .....	68
Image Order Status .....	69
Order Set.....	69
Order Set Status.....	69
Production Orders.....	70
Production Order Statuses .....	71
Server Configuration Samples.....	71
Production Server Configuration.....	71
Imaging Server Configuration .....	72
Server Dialog Samples .....	72
Alert Dialog .....	72
Error Dialog .....	72
Server Request / Reply Samples.....	73
GetServerStatus Request .....	73
GetServerStatus Reply .....	73
SetParameter Request .....	73
SetParameter Reply.....	74
<b>Appendix C – Server Status and Control Password Encryption.....</b>	<b>75</b>
Encryption Method.....	75
Rimage Core Encryption Algorithm.....	75
Password Encoding Samples Using C++ .....	76
Encoding and Decoding a MBCS String .....	76
Encoding and Decoding a Unicode String.....	77
<b>Appendix D – Error Codes.....</b>	<b>81</b>

## Important Information

This section provides support contact information, cautions and warnings, and product specifications for the SDK.

### Support Information

US, Asia/Pacific, Mexico/Latin America	Europe
<b>Rimage Corporation</b> 7725 Washington Avenue South Minneapolis, MN 55439 USA <b>Attn: Rimage Services</b>	<b>Rimage Europe GmbH</b> Albert-Einstein-Str. 26 63128 Dietzenbach Germany
<b>Contact Rimage Services:</b> <b>Website:</b> <a href="http://www.rimage.com/support">www.rimage.com/support</a> <b>KnowledgeBase:</b> <a href="http://www.rimage.custhelp.com">http://www.rimage.custhelp.com</a> Log in and select the Ask a Question tab. <b>Telephone:</b> North America: 800-553-8312 Asia/Pacific, Mexico/ Latin America: 952-946-0004 <b>Fax:</b> 952-946-6956	<b>Contact Rimage Services Europe:</b> <b>Website:</b> <a href="http://www.rimage.de">www.rimage.de</a> <b>Email:</b> <a href="mailto:support@rimage.de">support@rimage.de</a> <b>Telephone:</b> +49-(0) 1805-7462-43 <b>Fax:</b> +49-(0) 6074-8521-101
<b>When you contact Rimage Services, please provide:</b> <ul style="list-style-type: none"> <li>• System serial number and software version.</li> <li>• Functional and technical description of the problem.</li> <li>• Exact error message received.</li> </ul>	<b>My Rimage Product Information:</b> Copy this information from your Rimage Product for future reference. <b>Note:</b> Make sure you update the Serial Number here anytime you receive a replacement autoloader.
	<b>Serial Number:</b>
	<b>Product Name:</b>
	<b>Date of Purchase:</b>

### Learn More Online

At [www.rimage.com/support](http://www.rimage.com/support), you can experience Rimage's world-class Support and Services.

From the <b>Support</b> home page: 1. Select your <b>product series</b> . 2. Select your <b>product</b> . 3. Learn more on the <b>product page</b> .	From the product page you can access: Y Information about the latest software and firmware updates Y Product specifications Y The latest documents Y Current firmware and driver downloads
---	--

## Introduction

The Client API allows third-party developers and users to access Rimage Production Orders and Image Orders. The Client API includes Rimage XML DTDs as well as a set of Java and C/C++ APIs. The Client API provides an open and platform-independent way to access the publishing power of Rimage systems.

This document provides programming information necessary to create a custom application using the Rimage Software Development Kit (SDK). This includes Client API programming, Rimage XML APIs, and XML-based Status and Control for Production and Imaging Server.

## Overview

The Client API consists of two parts:

The programming interface

The XML interface

The programming interface acts as a conduit for passing information between the client and the Rimage server. The data that is passed from the programming interface to the Rimage server is described in XML. The XML document can include order information, order status, and server information. The use of XML allows the programming interface to stay fairly simple and look almost identical across languages (Java, C++, etc.). When future additions are needed, only the XML definitions (DTDs) should require change and not the programming interface.

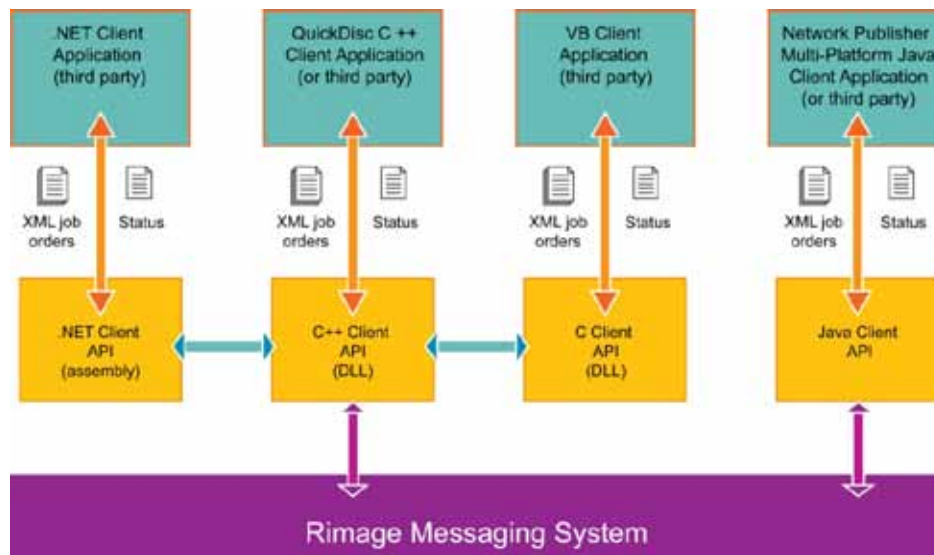
## Client API Design

Client APIs exist in Java, C / C++, and .NET.

- The C++ and C APIs are written for Windows only and are packaged as DLLs. The C++ and C APIs are Microsoft Visual C++ compatible.
- The Java Client APIs are compatible with JDK version 1.4 and above.

The Client API layer presents a client-based interface that allows Rimage applications and third-party applications to perform actions that include submitting orders, receiving unsolicited order statuses, canceling orders, receiving unsolicited system statuses, and setting server parameters.

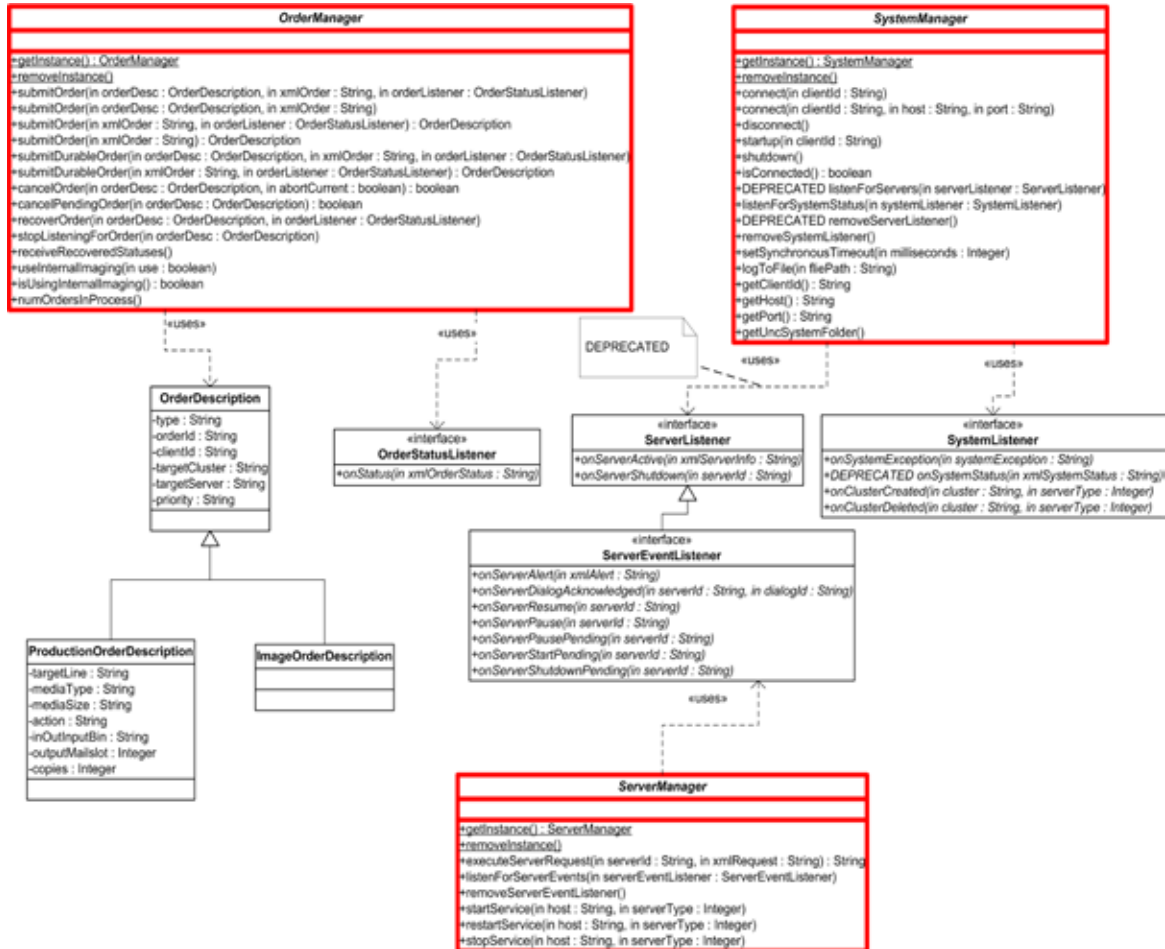
The illustration below shows the Rimage Client API layers and overall organization in a distributed deployment.



**Rimage Client API Layers**



The UML (Unified Modeling Language) diagram of the Client API is shown in the figure below.



Client API Interface UML Diagram

## Client API Use of XML

The Client API uses XML to pass data from the programming interface to the Rimage server. The XML document must conform to a Rimage DTD. To validate incoming and outgoing XML documents, each software component must access the Rimage DTD.

- **Note:** It is the end user's responsibility to validate the XML strings before sending the XML document.

### XML Encoding Format for Production Server and Imaging Server

The XML encoding format varies between Production Server (PS) and Imaging Server (IS).

- **Note:** Rimage recommends leaving the encoding format options unspecified to allow the servers to determine the correct encoding.
- In Imaging Server, the XML encoding format options are blank, UTF-8, and UTF-16.
- In Production Server, the XML encoding format options are blank and UTF-16.

### Rimage DTDs

The following DTDs are provided by the Producer Software Suite 8.x installation and are used by the Rimage components and third-party applications.

These Rimage DTDs can be grouped into several categories:

- **Note:** The actual file that is installed has a version number appended (e.g., *ProductionServerConfiguration\_1.10.dtd*).

DTD Type	Related DTDs
Order	ImageOrder.dtd ProductionOrder.dtd CdText.dtd
Order Status	ProductionOrderStatus.dtd ImageOrderStatus.dtd
Spanning	OrderSet.dtd OrderSetStatus.dtd
Server Configuration	ProductionServerConfiguration.dtd ImageServerConfiguration.dtd BridgeServerConfiguration.dtd
Server Dialogs	AlertDialog.dtd ErrorDialog.dtd
Server Status & Control	ProductionServerRequest.dtd ProductionServerReply.dtd ImageServerRequest.dtd ImageServerReply.dtd
Miscellaneous	DiscMap.dtd FullDiscMap.dtd

## DTD Location

All Rimage components (e.g., Production Server, QuickDisc, or a third-party client application) must have access to the DTDs used in the Rimage messaging system. During the Rimage SDK installation, the DTDs are placed by default in the specified folder location *C:\Program Files\RimageSDK\ApiSdk\XML*. The SDK installation procedure allows this folder location to be changed.

- **Note:** You can also change the folder location by running the install in *Repair mode* after an initial install has been performed.

XML parsers must know the location of the DTD against which the XML document is verified. The full path to the DTD must be specified in the XML document itself.

**J Important!** If the document specifies just the name of the DTD, the parser assumes the DTD exists in the current folder and errors out if this is not the case.

For example, if a client submits a Production Order for processing, the SYSTEM line must read something like this:

```
<!DOCTYPE ProductionOrder SYSTEM "C:\rimage\xml\ProductionOrder_1.0.DTD">
```

## DTD Versions

The version of a Rimage DTD is reflected in the DTD file name. For example, *ProductionOrder\_1.0.dtd* is a 1.0 version of the ProductionOrder. Any change to the ProductionOrder increments the version and alter the DTD file name accordingly. This allows multiple DTD versions to coexist in a single folder and ensures that the version of each DTD is easily identified.

## Client ID and Order ID Uniqueness Rules

Each *ClientID* and *OrderID* must be unique. To ensure order uniqueness, Rimage makes the following recommendations:

- The *ClientID* must be used to connect to the Messaging Server (eMS). The Messaging Server requires *ClientID* uniqueness and returns an error at connect time if a non-unique *ClientID* is detected. This ensures *ClientID* uniqueness.
  - **Note:** Because more than one instance of an application can be run on one machine, the Rimage applications' *ClientID* is in the form of *<HostName> + "\_" + <ApplicationInstanceld>*.
- Developers must ensure that the *OrderIDs* that they generate are unique for a particular client application. This still leaves the chance for two clients to generate identical *OrderIDs*, which is resolved in the following requirement.
- Production and Imaging servers must take both *OrderIDs* and *ClientIDs* into account to ensure order uniqueness internally to the server.

**The *ClientID* and *OrderID* uniqueness rules:**

- The *OrderID* is unique in the client application's namespace.
- The *ClientID* is unique in the Messaging Server namespace.
- The *ClientID* includes the *<HostName> + "\_" + <ApplicationInstanceld>*.
- Alphanumeric character entries are typical for the *ClientID* and *OrderID*. The entries are not case-sensitive; however, use of the period "." and backslash "\" must be excluded from the *ClientID* and *OrderID* entries.

## Client API Programming Class Definitions

Rimage Client API usage breaks down into three groups of operations.

### System-Related Operations Group

**SystemManager** is the main interface class for this group. This class allows you, the API user, to connect and disconnect from the Messaging Server (eMS). It also allows you to set and get system-wide parameters, such as synchronous calling timeout.

### Server-Related Operations Group

**ServerManager** is the main interface class for this group. This class allows you to listen for server events, control server states, and set and get server parameters.

### Order-Related Operations Group

**OrderManager** is the main interface class for this group. This class allows you to submit orders, listen for order status, cancel orders, etc.

## System Management

Before you can use the Rimage system, the client application must connect to the Messaging Server (eMS) using the SystemManager object. A client application connects to only one Messaging Server (eMS) at a time. The client application is free to disconnect and connect to a Messaging Server running on another computer at any time. After the client application connects to a Messaging Server, it can set up a listener to listen for system events, such as exceptions.

### Connect to the System

#### SystemManager.Connect()

This overloaded method allows the client to connect to the Messaging Server either on the localhost or anywhere on the network by specifying a host IP address and port. The ClientID passed into this method must be unique in the system. This uniqueness is the responsibility of the client and is enforced by the messaging system. If the host IP address and port are not passed into the *connect()* method, connection is attempted to a broker running on the local host with a default port. After the client process is connected to the Messaging Server all the other system or order related operations are possible.

#### SystemManager.Disconnect()

When the client session is over, a SystemManager.disconnect() method is called. Once this method has been called, all further operations throws an exception.

### Sample Code

#### Start/End Session using .NET with C#

```
//Start session.
// This is the first method to call before any other in the API,
// typically when the client application initializes.
CSystemManager.Initialize();
if (bLocal)
{
    // Connect locally with ClientID of "Client1",
    // host "localhost" (default), port "4664" (default).
    CSystemManager.GetInstance().Connect("Client1");
}
else
{
    // Connect with clientId of "Client1", host "Computer1",
    // port "4664".
    CSystemManager.GetInstance().Connect("Client1", "Computer1", "4664");
}
//End session.
// No operations can be done with the API after this method is
// called.
SystemManager.GetInstance().Disconnect();

// This method needs to be the very last method called in the API
// typically when the client application is shutting down.
CSystemManager.Terminate();
```

## Start/End Session using Java

```
//Start session.
if (bLocal)
{
    // Connect locally with ClientID of "Client1",
    // host "localhost" (default), port "4664" (default).
    SystemManager.getInstance().connect("Client1");
}
else
{
    // Connect with ClientID of "Client1", host "Computer1",
    // port "4664".
    SystemManager.getInstance().connect("Client1", "Computer1", "4664");
}
//End session.
// No operations can be done with the API after this method is called.
SystemManager.getInstance().disconnect();
```

## Start/End Session using C++

```
//Start session.
if(bLocal)
{
    // Connect locally with ClientID of "Client1",
    // host "localhost" (default), port "4664" (default).
    SystemManager::getInstance()->connect("Client1");
}
else
{
    // Connect with ClientID of "Client1", host "Computer1", port "4664".
    SystemManager::getInstance()->connect("Client1", "Computer1", "4664");
}
//End session.
// No operations can be done with the API after this method is called.
SystemManager::getInstance()->disconnect();

// Clean up memory.
OrderManager::removeInstance();
SystemManager::removeInstance();
```

## Start/End Session using C

```
/*Start session.*/
if(bLocalMessaging)
{
/* Connect locally with ClientID of "Client1", host "localhost" (default), port
"4664" (default).*/
    RCA_connect("Client1");
}
else
{
/* Connect with ClientID of "Client1", host "Computer1", port "4664".*/
    RCA_connectEx("Client1", "Computer1", "4664");
}
/*End session.*/
/* No operations can be done with the API after this method is called.*/
RCA_disconnect();
```

## Start/End Session using VB 6

- **Note:** Rimage recommends the use of the .NET API for VB.NET.

```
` Start session.
` Declare the functions.
Public Declare Function RCA_connect Lib "RmClient_7_3_n_3.dll" _
    (ByVal clientId As String) As Integer
Public Declare Function RCA_connectEx Lib "RmClient_7_3_n_3.dll" _
    (ByVal clientId As String, _
    ByVal host As String, _
    ByVal port As String) As Integer

Dim ret As Integer

` Connect locally with client "client1_VB".
ret = RCA_connect("client1_VB")
` End session.
` Declare the functions.
Public Declare Function RCA_disconnect Lib "RmClient_7_3_n_3.dll" () As Integer

Dim ret As Integer

` No operations can be done with the API after this method is called.
ret = RCA_disconnect();
```

## Listen for System Events

### **SystemManager.listenForSystemStatus(SystemListener)**

This method allows the client process to receive information about system-wide events. The listener class argument provides four callback methods while listening for system events:

#### **SystemListener.onSystemStatus()**

This callback method is called whenever a Production Server publishes a message consisting of an XML instance of either the *AlertDialog.DTD* or the *ErrorDialog.DTD*. This method also receives any future additions or changes to system type messages.

- **Note:** This callback is deprecated, instead use `ServerEventListener.onServerAlert()`.

#### **SystemListener.onSystemException()**

This callback method is called if there is a problem with the messaging system, a *connection broken* event for example. There is no XML passed into this method because the only errors are errors of connection to the Broker.

- **Note:** The client application should go into a reconnect loop once the `SystemListener.onSystemException()` callback method is received. It could take up to a minute for the client application to reconnect once the connection has been reestablished.

#### **SystemListener.onClusterCreated()**

This callback method is called if a new cluster is created through an administration tool.

#### **SystemManager.onClusterDeleted()**

This callback method is called if a cluster is deleted through an administration tool.

#### **SystemManager.removeSystemListener()**

This callback method allows the client to stop receiving the system status messages.

## Sample Code

### Listen for System Events using .NET with C#

```
// Start listening.
// Class that will receive system status.
public class MySystemListener : ISystemListener
{
    public void OnSystemException(SystemException e)
    {
        Console.WriteLine("--- System Exception");
        Console.WriteLine(e);
    }
    public void OnClusterCreated(String cluster, int serverType)
    {
        Console.WriteLine("--- Cluster Created");
        Console.WriteLine(cluster);
    }
    public void OnClusterDeleted(String cluster, int serverType)
    {
        Console.WriteLine("--- Cluster Deleted");
    }
}
```



```
        Console.WriteLine(cluster);
    }
}
// Set up the system listener.
MySystemListener systemListener = new MySystemListener();
CSystemManager.GetInstance().ListenForSystemStatus(systemListener);
// Stop listening.
// We are done listening for system status.
CSystemManager.GetInstance().RemoveSystemListener();
```

## Listen for System Events using Java

```
// Start listening.
// Class that will receive system status.
public class MySystemListener implements SystemListener
{
    public void onSystemException(SystemException e)
    {
        System.out.println("\n--- System Exception");
        System.out.println(e.getMessage());
        e.printStackTrace();
    }
    public void onClusterCreated(String cluster, int serverType)
    {
        System.out.println("\n--- Cluster Created");
        System.out.println(cluster);
    }
    public void onClusterDeleted(String cluster, int serverType)
    {
        System.out.println("\n--- Cluster Deleted");
        System.out.println(cluster);
    }
    public void onSystemStatus(String xmlSystemStatus)
    {
        // The onSystemStatus method is deprecated - use
        // ServerEventListener.onServerAlert.
        System.out.println("\n--- System Status");
        System.out.println(xmlSystemStatus);
    }
}

// Set up the system listener.
MySystemListener systemListener = new MySystemListener ();
SystemManager.getInstance().listenForSystemStatus(systemListener);

// Stop listening.
// We are done listening for system status.
SystemManager.getInstance().removeSystemListener();
```

## Listen for System Events using C++

```
// Start listening.
// Class that will receive system status.
class MySystemListener : public SystemListener
{
    void onSystemStatus(LPCTSTR xmlSystemStatus);
    void onClusterCreated(LPCTSTR cluster, int serverType);
    void onClusterDeleted(LPCTSTR cluster, int serverType);
    void onSystemException(SystemException* e);
};

void MySystemListener::onSystemStatus(LPCTSTR xmlSystemStatus)
{
    // The onSystemStatus method is deprecated - use
    // ServerEventListener::onServerAlert.
    printf("\n--- System Status");
    printf (xmlSystemStatus);
}

void MySystemListener::onSystemException(SystemException* e)
{
    printf("\n--- System Exception");
    e->printMessage();
    e->printStackTrace();
}

void MySystemListener::onClusterCreated(LPCTSTR cluster, int serverType)
{
    printf "\n--- Cluster Created");
    printf(cluster);
}

void MySystemListener::onClusterDeleted(LPCTSTR cluster, int serverType)
{
    printf("\n--- Cluster Deleted");
    printf(cluster);
}

// Set up the system listener.
MySystemListener systemListener;
SystemManager::getInstance()->listenForSystemStatus(&systemListener);
// Stop listening.
// We are done listening for system status.
SystemManager::getInstance()->removeSystemListener();
```

## Listen for System Events using C

```
/* Start listening. */
/* Callback function to receive system status notifications. */
void CALLBACK systemStatusCallback(LPCTSTR xmlSystemStatus)
{
    printf("\n--- System Status");
    printf (xmlSystemStatus);
}

/* Callback function to receive new cluster notifications. */
```

```

void CALLBACK clusterCreatedCallback(LPTSTR cluster)
{
    printf "\n--- Cluster Created";
    printf(cluster);
}
/* Callback function to receive deleted cluster notifications. */
void CALLBACK clusterDeletedCallback(LPTSTR cluster)
{
    printf("\n--- Cluster Deleted");
    printf(cluster);
}
/* Callback function to receive system error notifications. */
void CALLBACK systemExceptionCallback(LPTSTR error)
{
    printf("\n--- System Exception");
    printf(error);
}
/* Set up to listen for system status. */
RCA_listenForSystemStatus(systemStatusCallback, clusterCreatedCallback,
clusterDeletedCallback, systemExceptionCallback);
/* Stop listening. */
/* We are done listening for system status. */
RCA_removeSystemListeners();

```

## Listen for System Events using VB 6

- **Note:** Rimage recommends the use of the .NET API for VB.NET.

```

' Start listening.
' Declare the functions.
Public Declare Function RCA_listenForSystemStatusBstr Lib
"RmClient_7_3_n_3.dll" _
    (ByVal systemStatusCallback As Long, _
    ByVal clusterCreatedCallback As Long, _
    ByVal clusterDeletedCallback As Long, _
    ByVal systemExceptionCallback As Long) As Integer
' Declare callback functions.
Public Function systemExceptionCallback(ByVal error As String) As Long
MsgBox "Error : " & error
End Function

Public Function systemStatusCallback(ByVal systemStatus As String) As Long
Debug.Print "System status is " & systemStatus
End Function

Public Function clusterCreatedCallback(ByVal cluster As String) As Long
Debug.Print "Created cluster is " & cluster
End Function

Public Function clusterDeletedCallback(ByVal cluster As String) As Long
Debug.Print "Deleted cluster is " & cluster
End Function

```

```
` Set up to listen for system status.
ret = RCA_listenForSystemStatusBstr(AddressOf systemStatusCallback,_
AddressOf clusterCreatedCallback,_
AddressOf clusterDeletedCallback,_
AddressOf systemExceptionCallback)
` Stop listening.
` Declare the functions.
Public Declare Function RCA_removeSystemListeners Lib "RmClient_7_3_n_3.dll" ()
As Integer
Dim ret As Integer
` We are done listening for system status.
ret = RCA_removeSystemListeners()
```

# Server Management

Server management through the Client API consists of asynchronous and synchronous portions:

- **Asynchronous** – Information that the servers send to the Client API. For example, notification of server configuration and server states.
- **Synchronous** – Requests that the caller sends to the server and the replies that the server sends back.

The following two sections describe listening for server events and synchronous server methods in detail and provide code samples for reference.

## Listening for Server Events

### ServerManager.listenForServerEvents(ServerEventListener)

This method allows the client process to receive information about the servers that are active, becoming active, or shutting down on the network. The *ServerEventListener* object passed into this method is called when any of the above occurs. The information passed into the *ServerEventListener.OnServerActive()* method is in the form of an XML document (string), conforming to one of the following DTDs:

- ProductionServerConfiguration DTD
- ImageServerConfiguration DTD
- BridgeServerConfiguration DTD
- AlertDialog DTD
- ErrorDialog DTD

The rest of the methods take a single string whose value is the *ServerID* of the server that is changing states.

### ServerManager.removeServerEventListener()

This method allows the client to stop receiving the above messages.

## Sample Code

### Listen for Server Events using .NET with C#

```
// Start listening.
// Class that will receive server notifications.
public class MyServerListener : IServerEventListener
{
    public void OnServerActive(String xmlServerInfo)
    {
        Console.WriteLine("--- Active Server Information");
        Console.WriteLine(xmlServerInfo);
    }
    public void OnServerStartPending(String serverId)
    {
        Console.WriteLine("--- Start Pending Server Id");
        Console.WriteLine(Server Id);
    }
    public void OnServerPause(String serverId)
    {
        Console.WriteLine("--- Pause Server Id");
    }
}
```

```
        Console.WriteLine(serverId);
    }
    public void OnServerResume(String serverId)
    {
        Console.WriteLine("--- Resume Server Id");
        Console.WriteLine(serverId);
    }
    public void OnServerPausePending(String serverId)
    {
        Console.WriteLine("--- Pause Pending Server Id ");
        Console.WriteLine(serverId);
    }
    public void OnServerShutdownPending(String serverId)
    {
        Console.WriteLine("--- Shutdown Pending Server Id");
        Console.WriteLine(serverId);
    }
    public void OnServerShutdown(String serverId)
    {
        Console.WriteLine("--- Shutting down Server Id");
        Console.WriteLine(serverId);
    }
    public void OnServerAlert(String xmlDialog)
    {
        Console.WriteLine("--- Server Dialog");
        Console.WriteLine(xmlDialog);
    }
    public void OnServerDialogAcknowledged(String serverId, String dialogId)
    {
        Console.WriteLine("--- Server Dialog Acknowledged");
        Console.WriteLine(serverId + ", " + dialogId);
    }
    void onServerStartupMessage(String serverId, String message)
    {
        Console.WriteLine("--- Server Startup message");
        Console.WriteLine(serverId + ": " + message);
    }
}

// Set up the server listener.
MyServerListener serverListener = new MyServerListener();
CServerManager.GetInstance().ListenForServerEvents(serverListener);
// Stop listening.
// We are done listening for servers.
CServerManager.GetInstance().RemoveServerEventListener();
```

## Listen for Server Events using Java

```
// Start listening.
// Class that will receive server notifications.
public class MyServerListener implements ServerEventListener
{
    public void onServerActive(String xmlServerInfo)
    {
        System.out.println("\n--- Active Server Information");
        System.out.println(xmlServerInfo);
    }
    public void onServerStartPending(String serverId)
    {
        System.out.println("\n--- Start Pending Server ID");
        System.out.println(serverId);
    }
    public void onServerPause(String serverId)
    {
        System.out.println("\n--- Pause Server ID");
        System.out.println(serverId);
    }
    public void onServerResume(String serverId)
    {
        System.out.println("\n--- Resume Server ID");
        System.out.println(serverId);
    }
    public void onServerPausePending(String serverId)
    {
        System.out.println("\n--- Pause Pending Server Id");
        System.out.println(serverId);
    }
    public void onServerShutdownPending(String serverId)
    {
        System.out.println("\n--- Shutdown Pending Server Id");
        System.out.println(serverId);
    }
    public void onServerShutdown(String serverId)
    {
        System.out.println("\n--- Shutting down Server Id");
        System.out.println(serverId);
    }
    public void onServerAlert(String xmlDialog)
    {
        System.out.println("\n--- Server Dialog");
        System.out.println(xmlDialog);
    }
    public void onServerDialogAcknowledged(String serverId, String dialogId)
    {
        System.out.println("\n--- Server Dialog Acknowledged");
    }
}
```

```
        System.out.println(serverId + ", " + dialogId);
    }
    void onServerStartupMessage(String serverId, String message)
    {
        System.out.println("--- Server Startup message");
        System.out.println(serverId + ": " + message);
    }
}
// Set up the server listener.
MyServerListener serverListener = new MyServerListener();
ServerManager.getInstance().listenForServerEvents(serverListener);
// Stop listening.
// We are done listening for servers.
ServerManager.getInstance().removeServerEventListener();
```

### Listen for Server Events using C++

```
// Start listening.
// Class that will receive server notifications.
class MyServerListener : public ServerEventListener
{
    void onServerActive(LPCTSTR xmlServerInfo);
    void onServerStartPending(LPCTSTR serverId);
    void onServerPause(LPCTSTR serverId);
    void onServerPausePending(LPCTSTR serverId);
    void onServerResume(LPCTSTR serverId);
    void onServerShutdownPending(LPCTSTR serverId);
    void onServerShutdown(LPCTSTR serverId);
    void onServerAlert(LPCTSTR xmlDialog);
    void onServerDialogAcknowledged(LPCTSTR serverId);
    void onServerStartupMessage(LPCTSTR serverId, LPCTSTR message);
};

void MyServerListener::onServerActive(LPCTSTR xmlServerInfo)
{
    printf("\n--- Active Server Information");
    printf(xmlServerInfo);
}

void MyServerListener::onServerStartPending(LPCTSTR serverId)
{
    printf("\n--- Start Pending Server Id");
    printf(serverId);
}

void MyServerListener::onServerPause(LPCTSTR serverId)
{
    printf("\n--- Pause Server Id");
    printf(serverId);
}

void MyServerListener::onServerResume(LPCTSTR serverId)
{
    printf("\n--- Resume Server Id");
```



```
    printf(serverId);
}
void MyServerListener::onServerPausePending(LPCTSTR serverId)
{
    printf("\n--- Pause Pending Server Id");
    printf(serverId);
}
void MyServerListener::onServerShutdownPending(LPCTSTR serverId)
{
    printf("\n--- Shutting down Pending Server Id");
    printf(serverId);
}
void MyServerListener::onServerShutdown(LPCTSTR serverId)
{
    printf("\n--- Shutting down Server Id");
    printf(serverId);
}
void MyServerListener::onServerAlert(LPCTSTR xmlDialog)
{
    printf("\n--- Server Dialog");
    printf(xmlDialog);
}
void MyServerListener:: onServerDialogAcknowledged(LPCTSTR serverId, LPCTSTR
dialogId)
{
    printf("\n--- Dialog Acknowledged ");
    printf(dialogId);
}
void MyServerListener:: onServerStartupMessage(LPCTSTR serverId, LPCTSTR
message)
{
    printf("\n--- Server Startup message ");
    printf(message);
}
// Set up the server listener.
MyServerListener serverListener;
ServerManager::getInstance()->listenForServerEvents(&serverListener);
// Stop listening.
// We are done listening for servers.
ServerManager::getInstance()->removeServerEventListener();
```

## Listen for Server Events using C

```
/* Start listening. */
/* Callback function to receive active server notifications. */
void CALLBACK serverActiveCallback(LPTSTR xmlServerInfo)
{
    printf("\n--- Active Server");
    printf(xmlServerInfo);
}
/* Callback function to receive server shutdown notifications. */
void CALLBACK serverShutdownCallback(LPTSTR serverId)
{
    printf("\n--- Shutdown Server");
    printf(serverId);
}
/* Set up to listen for servers. */
RCA_listenForServers(serverActiveCallback, serverShutdownCallback);
/* Stop listening. */
/* We are done listening for servers. */
RCA_removeServerListeners();
```

## Listen for Server Events using VB 6

- **Note:** Rimage recommends the use of the .NET API for VB.NET.

```
` Start listening.
` Declare the functions.
Public Declare Function RCA_listenForServersBstr Lib "RmClient_7_3_n_3.dll" _
    (ByVal serverActiveCallback As Long, _
     ByVal serverShutdownCallback As Long) As Integer
` Declare callback functions.
Public Function serverActiveCallback(ByVal xmlServerActive As String) As Long
    Debug.Print "Active server is " & xmlServerActive
End Function

Public Function serverShutdownCallback(ByVal server As String) As Long
    Debug.Print "Shutdown server is " & server
End Function

` Set up to listen for servers.
ret = RCA_listenForServersBstr(AddressOf serverActiveCallback, AddressOf
serverShutdownCallback)
` Stop listening.
` Declare the functions.
Public Declare Function RCA_removeServerListeners Lib "RmClient_7_3_n_3.dll" ()
    As Integer
Dim ret As Integer
` We are done listening for servers.
ret = RCA_removeServerListeners()
```

## Synchronous Server Methods

The caller of the Client API can start, restart, and stop any of the Rimage servers when running as a Windows Service and using the following *ServerManager* methods directly:

- `ServerManager.startService()`
- `ServerManager.restartService()`
- `ServerManager.stopService()`

The caller can also choose to take advantage of the Production and Imaging server **Status and Control protocols**. The **Status and Control protocols** allow the caller to request and receive server status, server parameters, current orders in process, etc. The protocols also allow the caller to set server parameters, acknowledge server dialogs, control server state, etc. For more information about the Production and Imaging server protocols, refer to '*Server status and control*' on page **Error! Bookmark not defined.**

The **Status and Control protocols** are accessed through the `ServerManager.executeServerRequest()` method. This method accepts an XML string conforming to one of the following DTDs:

- `ProductionServerRequest` DTD
- `ImageServerRequest` DTD

The `ServerManager.executeServerRequest()` method returns an XML string conforming to one of the following DTDs:

- `ProductionServerReply` DTD
- `ImageServerReply` DTD

The methods described above are synchronous.

& **Tip:** Use the `SystemManager.setSynchronousTimeout()` method to change the timeout value of all the synchronous methods in the Client API.

### Sample Code

#### Server Methods using .NET with C#

```
// Start/Stop/Restart services.
// Start Production Service on the local computer.
CServerManager.GetInstance().StartService("computer1",
CSystemManager.PRODUCTION_SERVER_TYPE);

// Stop Imaging Service on the local computer.
CServerManager.GetInstance().StopService("computer1",
CSystemManager.IMAGE_SERVER_TYPE);

// Restart Messaging Service on the local computer.
// NOTE: This will break the current Messaging Server(eMS) connection.
CServerManager.GetInstance().RestartService("computer1",
CSystemManager.MESSAGING_SERVER_TYPE);
// Execute Server Request.

// Create a request XML string conforming to either the
// ProductionServerRequest dtd or ImageServerRequest dtd.
String request = CreateServerRequest();
```

```
// Call the server to execute the request. Get back a reply XML string
// conforming to either the ProductionServerReply dtd or
ImageServerReply
// dtd.

String reply =
CServerManager.GetInstance().ExecuteServerRequest("computer1_PS01",
request);

// Parse the reply using any XML parser.
ParseServerReply(reply);
```

### Server Methods using Java

```
// Start/Stop/Restart services.
// Start production service on the local computer.
ServerManager.GetInstance().startService("computer1",
SystemManager.PRODUCTION_SERVER_TYPE);

// Stop Imaging Service on the local computer.
ServerManager.GetInstance().stopService("computer1",
SystemManager.IMAGE_SERVER_TYPE);

// Restart Messaging Service on the local computer.
// NOTE: This will break the current Messaging Server(eMS) connection.
ServerManager.GetInstance().restartService("computer1",
SystemManager.MESSAGING_SERVER_TYPE);

// Execute Server Request.
// Create a request XML string conforming to either
// the ProductionServerRequest dtd or ImageServerRequest dtd.
String request = createServerRequest();

// Call the server to execute the request. Get back a reply XML
// string conforming to either the ProductionServerReply dtd or
// ImageServerReply dtd.
String reply =
ServerManager.GetInstance().executeServerRequest("computer1_PS01", request);
// Parse the reply using any XML parser.
parseServerReply(reply);
```

## Server Methods using C++

```
// Start/Stop/Restart services.
// Start Production Service on the local computer.
ServerManager::getInstance()->startService("computer1",
SystemManager::PRODUCTION_SERVER_TYPE);

// Stop Imaging Service on the local computer.
ServerManager::getInstance()->stopService("computer1",
SystemManager::IMAGE_SERVER_TYPE);

// Restart Messaging Service on the local computer.
// NOTE: This will break the current Messaging Server(eMS) connection.
ServerManager::getInstance()->restartService("computer1",
SystemManager::MESSAGING_SERVER_TYPE);
// Execute Server Request.
// Create a request XML string conforming to either the
// ProductionServerRequest dtd or ImageServerRequest dtd.
std::string request = createServerRequest();

// Call the server to execute the request. Get back a reply XML string
// conforming to either the ProductionServerReply dtd or
// ImageServerReply dtd.
std::string reply =
ServerManager::getInstance()->executeServerRequest("computer1_PS01",
request.c_str());

// Parse the reply using any XML parser.
parseServerReply(reply);
```

## Server Methods using C

```
/* Execute Server Request. */
/* Create a request XML string conforming to either the
ProductionServerRequest dtd or ImageServerRequest dtd.*/
LPTSTR request = createServerRequest();

/* Call the server to execute the request. Get back a reply XML string
conforming to either the ProductionServerReply dtd or ImageServerReply
dtd.*/
LPTSTR reply = RCA_executeServerRequest("computer1_PS01", request);

/* Parse the reply using any XML parser. */
parseServerReply(reply);

/* Remove returned reply. */
RCA_removeServerResponse(reply);
```

## Server Methods using VB 6

- **Note:** Rimage recommends the use of the .NET API for VB.NET.

```
` Execute Server Request.
` Declare the functions.
Public Declare Function RCA_executeServerRequestBstr Lib
"RmClient_7_3_n_3.dll" _
(ByVal server As String, _
ByVal request As String, _
ByVal responseCallback As Long) As Integer

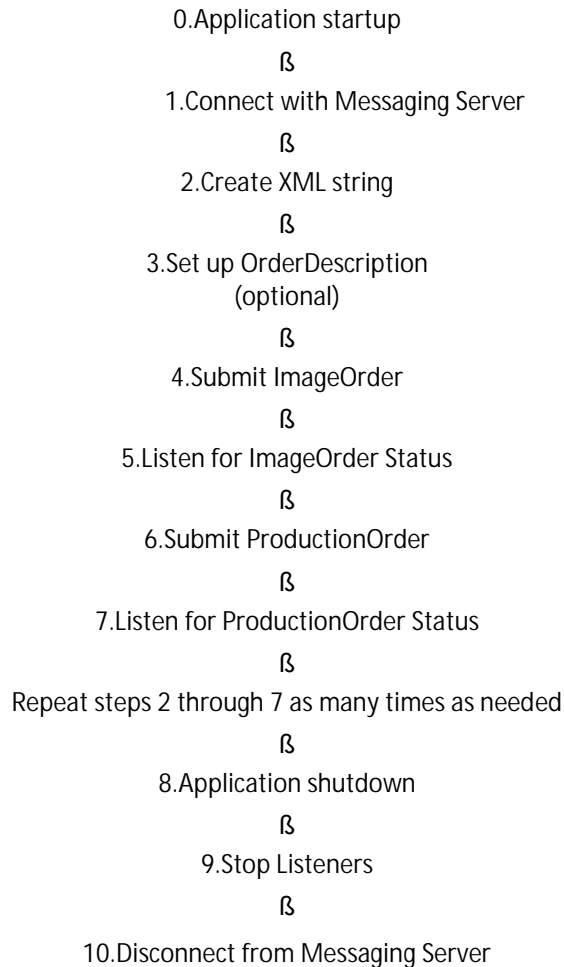
` Declare callback functions.
Public Function responseCallback(ByVal response As String) As Long
Debug.Print "Server response is " & response
End Function

` Call a synchronous method.
ret = RCA_executeServerRequestBstr("computer1_PS01", request, AddressOf
serverActiveCallback)
```

## Order Management

A client application typically completes the following steps to connect to the Messaging Server and then submit and manage orders. Refer to this process when using the Rimage client API to develop a client application to the Rimage Publishing system:

### Application Programming Flow Chart



- **Note:** An additional step, Recovering Orders, may be required after reconnecting to Messaging Server (eMS) if your client application shut down before order(s) were completed, and you used *OrderManager.submitDurableOrder()* to submit the order(s). If this is the case, you can recover order statuses that have not yet been received.

## Submit Orders

After you are connected to the Rimage system you are free to begin submitting orders. You can submit one or more orders to either Production Server or Imaging Server.

- Use the **OrderManager** class to submit orders. Only one OrderManager object per application is required. Either Production or Imaging orders can be submitted by reusing a single OrderManager object.
- Use the **OrderManager.getInstance()** static method to create a single, internal object to use for submitting, canceling, removing, and recovering orders.

When you submit an order you can also provide a listener object to receive periodic status messages for that order. You can create a separate listener object for each order or create a single listener object to manage all the orders your application submits.

## Order Management Methods

One of the following methods is called:

- **OrderManager.submitOrder(XMLOrder, OrderStatusListener)** returns an **OrderDescription** object. This method returns immediately, it does not wait while the order is processed and completed.
- **OrderManager.submitDurableOrder(XMLOrder, OrderStatusListener)** returns an **OrderDescription** object.
- **OrderManager.submitOrder(OrderDescription, XMLOrder, OrderStatusListener)**
- **OrderManager.submitDurableOrder(OrderDescription, XMLOrder, OrderStatusListener)**
- **Note:** Rimage recommends using one of the first two signatures to submit orders.

The first two methods [**OrderManager.submitOrder(XMLOrder, OrderStatusListener)** and **OrderManager.submitDurableOrder(XMLOrder, OrderStatusListener)**] are overloaded to contain two parameters.

The last two methods [**OrderManager.submitOrder(OrderDescription, XMLOrder, OrderStatusListener)** and **OrderManager.submitDurableOrder(OrderDescription, XMLOrder, OrderStatusListener)**] are overloaded to contain three parameters.

It is completely up to the user of this API which signatures to use. The last two methods require the caller to build the **OrderDescription** object, which in turn requires its values to match values in the XML order. The first two methods do not place that burden on the API user, but use a little more resources to parse the order.

### OrderDescription Parameter

- **Note:** The **OrderStatusListener** parameter has been deprecated for the last two methods.

The first required parameter is an **OrderDescription**, which is an object containing summary information for the order.

### XMLOrder Parameter

The **XMLOrder** parameter is the order itself represented in an XML string. This XML document conforms to either *ProductionOrder.DTD* or *ImageOrder.DTD*.

### OrderStatusListener

The third parameter, **OrderStatusListener** is optional. This allows the caller to listen for the status information on the order just submitted. If this parameter is omitted (or is null), the order is still submitted, but status is not propagated back to the client.

The durable versions of these methods allow for recovery of an order's statuses in the event of a client application shutdown before an order is completed. If this is the case, then calling **OrderManager.recoverOrder()** in combination with **OrderManager.receiveRecoveredStatuses()** during application restart recovers all the missed order statuses.



## OrderDescription

The OrderManager class has two signatures for each of the submitOrder() and submitDurableOrder() methods:

1. A signature that takes an OrderDescription object as well as the XML order string as parameters.
  2. A signature that takes only the XML order string as a parameter and returns a reference to an OrderDescription object.
- **Note:** Rimac recommends option 2. This is a simpler call to make for the caller, because the caller doesn't have to create an OrderDescription object before the method call. However in some circumstances, clients require creation of an OrderDescription object prior to the call.

When an order is submitted to the system with an OrderDescription object, the values in the OrderDescription must match the values in the XML order itself. This section describes how data in the XML order document is related to the data in the OrderDescription object. This section does not apply if you are using the methods that return these objects.

- **Note:** All values are case sensitive.

### OrderDescription Object as a Return Value

If the OrderDescription object is not one of the parameters, it is derived from the XML order string and returned to the caller. This object should be used in all subsequent OrderManager method calls related to this order.

- **Note:** In a language like C++, the caller is responsible for deleting the OrderDescription object by calling `OrderManager::removeOrderDescription()` after it is no longer needed. In a language like Java, the garbage collector deletes the object.

There are two types of OrderDescription objects that you actually use:

1. ImageOrderDescription for imaging orders.
2. ProductionOrderDescription for production orders.

#### Example 1:

```
OrderManager.getInstance().submitDurableOrder(orderDesc, XMLOrder String,  
orderStatusListener)
```

#### Example 2:

```
orderDesc = OrderManager.getInstance().submitDurableOrder(XMLOrder String,  
orderStatusListener);
```

### Cancel an Order in Process

While the order is in process, the user can cancel the order. There are two options for canceling an order in process:

1. `OrderManager.cancelPendingOrder(OrderDescription)`
2. `OrderManager.cancelOrder(OrderDescription)`

The `OrderManager.cancelPendingOrder()` method tries to remove a pending order from a cluster. If the server has not picked up the order for processing, this method will succeed.

- **Note:** Remember to use the same OrderDescription object to cancel an order in process that was used to submit the order.

The *OrderManager.cancelOrder()* method first tries to remove a pending order – an order that is still waiting to be processed. If the pending order cannot be removed, a cancel request is sent to the server that is processing the order, and return *false* to the caller. The server attempts to cancel the processing of the order and send an appropriate status, which the client receives via the *OrderStatusListener.onStatus()* method.

**Example:** `OrderManager.getInstance().cancelOrder(orderDescription,true);`

- **Note:** If the *OrderManager.cancelOrder()* method returns true, the order has been removed from the cluster because no server has begun work on it. If *OrderManager.cancelOrder()* method returns false, then you have to wait for a status message from either the Imaging Server or the Production Server that is currently processing the order. The *OrderManager.cancelOrder()* method sends the cancel request to the appropriate Imaging Server or Production Server.

After the order has been processed, failed, or canceled, the user is required to call **OrderManager.stopListeningForOrder()** method. This stops listening for the order status related to this order only – all other submitted orders are not affected.

**Example:** `OrderManager.getInstance().stopListeningForOrder(orderDescription);`

- **Note:** You must call *stopListeningForOrder* for a particular Order ID before you can reuse that Order ID since the system uses Order ID as a means of directing statuses for the order. Calling the *OrderManager.stopListeningForOrder()* method effectively tells the system that the client is finished with this order. The final order statuses returned are COMPLETED, FAILED, or CANCELED.

## Recover Orders

- **Note:** Order recovery should be planned during the design of the application. If order recovery is required, then only the *OrderManager.submitDurableOrder()* method should be used to submit orders.

There are cases when a client program crashes or is shut down before all orders are finished processing. The next time the client program starts up it should call **OrderManager.recoverOrder(OrderDescription, OrderStatusListener)** for each order previously submitted with *OrderManager.submitDurableOrder()* method. After *OrderManager.recoverOrder()* has been called for all orders to be recovered, *OrderManager.receiveRecoveredStatuses()* must be called. This allows the caller to receive order statuses while the client application is down.

**Example:**

```
OrderManager.getInstance().recoverOrder(orderDescription,
orderStatusListener); OrderManager.getInstance().receiveRecoveredStatuses();
```

## OrderDescription Base Class

Elements and attributes that are common to *ProductionOrder* and *ImageOrder* are represented in the *OrderDescription* base class. The following applies to both *ProductionOrder* and *ImageOrder*.

OrderDescription AttributeElement	XML Attribute
<i>OrderDescription.setOrderId()</i>	Set to the "OrderId" attribute value of <i>ProductionOrder</i> or <i>ImageOrder</i> element.
<i>OrderDescription.setClientId()</i>	Set to the "ClientId" attribute value of <i>ProductionOrder</i> or <i>ImageOrder</i> element.
<i>OrderDescription.setOriginator()</i>	Set to the "Originator" attribute value of <i>ProductionOrder</i> or <i>ImageOrder</i> element.
<i>OrderDescription.setPriority()</i>	Set to the "Priority" attribute value of <i>ProductionOrder</i> or <i>ImageOrder</i> element.

OrderDescription AttributeElement	XML Attribute
OrderDescription.setTargetCluster()	Set to the "Cluster" attribute of Target element.
OrderDescription.setTargetServer()	Set to the "Server" attribute of Target element.

### ImageOrderDescription Sub Class

The Image order description requires 3 *set* parameter methods to be called. Most of the parameters have defaults, but you still must make sure that these parameters have identical values in the XML order document. The required Image order *set* parameters are:

- OrderID, ClientID, Originator – no default values assigned
- Priority, TargetCluster, TargetServer – default values assigned
- **Note:** Currently, there is no additional data in an ImageOrderDescription other than what is in the base class.

### ProductionOrderDescription Sub Class

The Production order description requires 13 *set* parameter methods to be called. Most of the parameters have defaults, but you still must make sure that these parameters have identical values in the XML order document. The required Production order *set* parameters are:

- OrderID, ClientID, Originator - no default values assigned
- TargetCluster, Action, MediaType, MediaSize, TargetLine, InOutInputBin, OutputMailSlot, Copies, TargetServer, and Priority – default values assigned

Submitting a ProductionOrder requires information in addition to what is already contained in the OrderDescription base class.

OrderDescription AttributeElement	XML Attribute
ProductionOrderDescription.setCopies()	Set to the "Copies" attribute value of ProductionOrder element.
ProductionOrderDescription.setTargetLine()	Set to the "Line" attribute of Target element.
ProductionOrderDescription.setMediaType()	Set to the "Type" attribute of Media element.
ProductionOrderDescription.setMediaSize()	Set to the "Size" attribute of Media element.
ProductionOrderDescription.setInOutInputBin()	Set to the "InputBin" attribute of InOut element.
ProductionOrderDescription.setOutputMailslot ()	Set to the "OutputMailslot" attribute of InOut element.

OrderDescription AttributeElement	XML Attribute
ProductionOrderDescription.setAction()	<p>Set to the type of the first action found in a ProductionOrder XML document.</p> <ul style="list-style-type: none"> <li>• If first action element is Record – setAction() to "Record".</li> <li>• If first action element is Read – setAction() to "Read".</li> <li>• If first action element is Label – setAction() to "Label".</li> <li>• If first action element is Collate – setAction() to "Collate".</li> <li>• If first action element is Copy – setAction() to "Copy".</li> <li>• If first action element is Destroy – setAction() to "Destroy".</li> </ul>
ProductionOrderDescription.setLabelPresent()	<p>Set to boolean "true" or "false" to signify if the disc is to be printed on.</p>

## Streaming

As of PSS 7.2 Production Server (ePS) is able to stream image data (image file or a Rimage PowerImage file) directly from the computer hosting the Imaging Server (eIS).

The client application programmer needs to add information to the ImageOrder and ProductionOrder XML orders to facilitate this process.

### Changes to ImageOrder

Set the "Output" element's "Type" attribute to "PowerImage". This ensures the fastest Image file creation possible. PowerImage file is resolved during the streaming process to the Production Server.

- **Note:** This value can be set to "Normal" however optimal performance will not be achieved.

### Changes to ProductionOrder

- Set the "ProductionOrder" element's "ImagerHost" attribute to the name of the computer hosting the Imaging Server which produced the image.
- Set the "ProductionOrder" element's "ExternallImager" attribute to "true" since all imaging is done by a component other than the client application.
- Set the "ProductionOrder" element's "LogonId" attribute to the currently logged in Windows User's ID.
  - **Note:** If the above information is not entered in the ProductionOrder, the order is processed without streaming and performance may not be optimal.

## Spanning

As of PSS 7.3 Rimage software is able to record a single set of data on multiple discs, producing a spanned disc set.

The client application programmer needs to add information to the ImageOrder and ProductionOrder XML orders to facilitate this process. There are also additional XML orders involved in spanning: OrderSet and OrderSetStatus.

### Process Flow of a Spanned Disc Set

1. An ImageOrder is submitted to the Imaging Server specifying whether spanning is allowed.
2. Imaging Server produces multiple Image files for the specified data. As the Image file is being written to disc, the Imaging Server sends out ImageOrderStatus XML to the client. The order status specifies the total number of volumes (or discs) involved in this order, the current volume being worked on (1, 2, n), and the name of the current volume Image file.
  - **Note:** The size of each Image file in the set is determined by the Image size specified in the ImageOrder, which relates to the Media (CDR, DVDR, etc.) for this spanned disc set.
3. As soon as the first ImageOrderStatus comes back with the information on the total number of volumes, an OrderSet XML can be created and submitted through OrderManager just like any other order. The OrderSet includes a list of ProductionOrders that are involved in the creation of this spanned disc set.
4. Once an Image file for one of the volumes in the set is completed, a ProductionOrder for this Image file can be submitted. This pattern is repeated for each volume in the set.
  - **Note:**
    - Each ProductionOrder in the spanned disc set specifies one of the Image files as Data Tracks. Therefore there are as many ProductionOrders as there are image files in the set.
    - Production Server makes sure the discs in the spanned disc set are produced in the correct order - 1, then 2, etc.
5. Production Server sends out statuses for the spanned disc set as a whole (OrderSetStatus XML) as well as statuses for the individual ProductionOrders (ProductionOrderStatus XML).
6. Once the entire spanned disc set is produced, cancelled, or failed, the client application needs to call OrderManager.stopListeningForOrder() for every order involved in the set: ImageOrder, OrderSet, and multiple ProductionOrders.

### Changes to ImageOrder

Set the "Output" element's "PowerSpan" attribute to "true". This tells the Imaging Server that spanning is allowed.

### Interpreting ImageOrderStatus

There are three new IMPLIED fields in the Status element related to a spanned disc set. These fields tell the client application how many volumes are in the set, which volume is being worked on now, etc. The new fields are:

- "SpanTotalVolumes" – indicates the number of volumes in the spanned disc set.
- "SpanVolume" – indicates the volume eIS is currently working on. Numbering starts with "1".
- "SpanVolumeName" – indicates the file name of the volume eIS is currently working on. This is the filename to be specified in the "WriteTrack" element's "Filename" attribute of the ProductionOrder.

### OrderSet (New)

- Fill the attributes of the “OrderSet” element with the same information normally specified for any type of order.
- Add an “OrderReference” element for each ProductionOrder in this spanned disc set.
  - **Note:** Typical naming convention for orders in a spanned disc set is: <orderSetId> for OrderSet and <orderSetId>\_000x for each ProductionOrder in the spanned disc set.

Fill the “ProductionOrderSet” element with the same information normally specified for a ProductionOrder. This element exists to facilitate Production Server allocation of resources.

### Changes to ProductionOrder

- Set the “ProductionOrder” element’s “ReferencedSet” attribute to the order ID of the OrderSet submitted previously.
- Set the “WriteTrack” element’s “Filename” attribute to one of the Image files returned in the ImageOrderStatus XML.
- The rest of the ProductionOrder remains the same.

## Order Management Sample Code

### Order Management using .NET with C#

```
// Submit Production order.
    // Image order management is almost identical to Production order
    // management and is not covered here.
// Create the XML Production order document using any of the available
// XML parsers.
String xmlOrder = createProductionOrder();

// Submit order and get an OrderDescription object back.
// If recovery is not required, use a submitOrder() signature.
CProductionOrderDescription orderDescription =
COrderManager.GetInstance().SubmitDurableOrder(xmlOrder, orderListener);

// OR
// Create a ProductionOrderDescription object. Most of the parameters are
// defaulted, but you have to make sure that these parameters have
// identical values in the XML order document.
CProductionOrderDescription orderDescription = new
CProductionOrderDescription();
orderDescription.OrderId = "ProductionOrder01";
orderDescription.ClientId = "PO_Client_ID";
orderDescription.Originator = "PO_ORIGINATOR";

// Class that will receive order status notifications.
public class MyOrderListener : IOrderStatusListener
{
    public void OnStatus(String xmlOrderStatus)
    {
        // Check for the state of your order here, using an XML parser.
```

```
// This code will run in a client API managed thread different from
// the thread that submitted the order.
...
Console.WriteLine("--- Order status");
Console.WriteLine(xmlOrderStatus);
}
}

// Submit the Production order.
// If recovery is not required, use submitOrder() signature.
MyOrderListener orderListener = new MyOrderListener();
COrderManager.GetInstance().SubmitDurableOrder(orderDescription, xmlOrder,
orderListener);
// Order is being processed by the server and MyOrderListener is
// receiving statuses.
...
// If we want to cancel the order for some reason.
// If the order is cancelled by the server, the next status received
// by MyOrderListener will be CANCELLED.
COrderManager.GetInstance().CancelOrder(orderDescription);

// Stop listening for order status.
// Final order status received by MyOrderListener is either
// COMPLETED, FAILED, or CANCELLED.
// We need to stop listening for this order's status.
COrderManager.GetInstance().StopListeningForOrder(orderDescription, true);

// Recover orders.
// If your client application shuts down before order(s) were completed,
// and you used OrderManager.submitDurableOrder() to submit the order(s),
// then you can recover order statuses that were not received previously.
// The sequence for recovering orders is as follows:
// 1. Call OrderManager.RecoverOrder() for each order to be recovered.
// 2. Call OrderManager.ReceiveRecoveredStatuses() to initiate order
// recovery and to start receiving order statuses.
// Using the same mechanisms to create ProductionOrderDescription and
// MyOrderListener as in the above section, call the following for each
// order you want to recover.
CProductionOrderDescription orderDescription = new
CProductionOrderDescription();
orderDescription.OrderId = "TestOrder";
orderDescription.ClientId = "TESTCLIENT";
orderDescription.Originator = "RimageSample";
MyOrderListener orderListener = new MyOrderListener();
COrderManager.GetInstance().RecoverOrder(orderDescription, orderListener);

// Once all the orders have been recovered, call the following to start
// receiving order statuses.
COrderManager.GetInstance().ReceiveRecoveredStatuses();
```

## Order Management using Java

```
// Submit Production order.
// Image order management is almost identical to Production order
// management, and is not covered here.
// Create the XML Production order document using any of the available
// XML parsers.
String xmlOrder = createProductionOrder(...);

// Submit order and get an OrderDescription object back.
// If recovery is not required, use a submitOrder() signature.
ProductionOrderDescription orderDescription =
OrderManager.getInstance().submitDurableOrder(xmlOrder, orderListener);

// OR
// Create a ProductionOrderDescription object. Most of the parameters are
// defaulted, but you have to make sure that these parameters have
// identical values in the XML order document.
ProductionOrderDescription orderDescription = new ProductionOrderDescription();
orderDescription.setOrderId("ProductionOrder01");
orderDescription.setClientId("PO_Client_ID");
orderDescription.setOriginator("PO_ORIGINATOR")

// Class that will receive order status notifications
public class MyOrderListener implements OrderStatusListener
{
public void onStatus(String xmlOrderStatus)
{
    // Check for the state of your order here, using an XML parser.
    // This code will run in a client API managed thread different from
    // the thread that submitted the order.
    ...
    System.out.println("\n--- Order status");
    System.out.println(xmlOrderStatus);
}
}

// Submit the Production order.
// If recovery is not required, use a submitOrder() signature.
MyOrderListener orderListener = new MyOrderListener();
OrderManager.getInstance().submitDurableOrder(orderDescription, xmlOrder,
orderListener);
// Order is being processed by the server and MyOrderListener is
// receiving statuses.
// Cancel order.
// If we want to cancel the order for some reason.
// If the order is cancelled by the server, the next status received
// by MyOrderListener will be CANCELLED.
OrderManager.getInstance().cancelOrder(orderDescription);
// Stop listening for order status.
// Final order status received by MyOrderListener is either
```



```
// COMPLETED, FAILED, or CANCELLED.
// We need to stop listening for this order's status.
OrderManager.getInstance().stopListeningForOrder(orderDescription);

// Recover orders
// If your client application shuts down before order(s) were completed,
// and you used OrderManager.submitDurableOrder() to submit the order(s),
// then you can recover order statuses that were not received previously.
// The sequence for recovering orders is as follows:
// 1. Call OrderManager.recoverOrder() for each order to be recovered.
// 2. Call OrderManager.receiveRecoveredStatuses() to initiate order
// recovery and to start receiving order statuses.
// Using the same mechanisms to create ProductionOrderDescription and
// MyOrderListener as in the above section, call the following for each
// order you want to recover.
ProductionOrderDescription orderDescription;
orderDescription.setOrderId("ProductionOrder01");
orderDescription.setTargetCluster("DefaultProductionCluster");
MyOrderListener orderListener = new MyOrderListener();
OrderManager.getInstance().recoverOrder(orderDescription, orderListener);

// Once all the orders have been recovered, call the following to start
// receiving order statuses.
OrderManager.getInstance().receiveRecoveredStatuses();
```

## Order Management using C++

```
// Submit Production order.
// Image order management is almost identical to Production order
// management, and is not covered here.
// Create the XML Production order document using any of the available
// XML parsers.
LPCTSTR xmlOrder = createProductionOrder(...);
// Submit order and get an OrderDescription object back.
// If recovery is not required, use submitOrder() signature.
ProductionOrderDescription* orderDescription =
OrderManager::getInstance()->submitDurableOrder(xmlOrder, &orderListener);
// OR
// Create a ProductionOrderDescription object. Most of the parameters are
// defaulted, but you have to make sure that these parameters have
// identical values in the XML order document.
ProductionOrderDescription orderDescription;
orderDescription.setOrderId("ProductionOrder01");
orderDescription.setClientId("PO_Client_ID");
orderDescription.setOriginator("PO_ORIGINATOR")
orderDescription.setTargetCluster("DefaultProductionCluster");

// Class that will receive order status notifications.
class MyOrderListener : public OrderStatusListener
{
```

```
void onStatus(LPCTSTR xmlOrderStatus);
};
void MyOrderListener::onStatus(LPCTSTR xmlOrderStatus)
{
    // Check for the state of your order here, using an XML parser.
    // This code will run in a client API managed thread different from
    // the thread that submitted the order.
    .
    printf("\n--- Order status");
    printf(xmlOrderStatus);
}

// Submit the Production order.
// If recovery is not required, use submitOrder() signature.
MyOrderListener orderListener;
OrderManager::getInstance()->submitDurableOrder(&orderDescription, xmlOrder,
&orderListener);
// Order is being processed by the server and MyOrderListener is
// receiving statuses.
.
.
// Cancel order.
// If we want to cancel the order for some reason.
// If the order is cancelled by the server, the next status received
// by MyOrderListener will be CANCELLED.
OrderManager::getInstance()->cancelOrder(&orderDescription);

// Stop listening for order status.
// Order is either COMPLETED, FAILED, or CANCELLED
// We need to stop listening for this order's status
OrderManager::getInstance()->stopListeningForOrder(&orderDescription) ;

// If OrderDescription is returned from submitOrder()
OrderManager::getInstance()->removeOrderDescription(orderDescription);

// Recover orders.
// If your client application shuts down before order(s) were completed,
// and you used OrderManager.submitDurableOrder() to submit the order(s),
// then you can recover order statuses that were not received previously.
// The sequence for recovering orders is as follows:
// 1. Call OrderManager::recoverOrder() for all the orders to be recovered.
// 2. Call OrderManager::receiveRecoveredStatuses() to initiate order
// recovery and to start receiving order statuses.
// Using the same mechanisms to create ProductionOrderDescription and
// MyOrderListener as in the above section, call the following for each
// order you want to recover.
ProductionOrderDescription orderDescription;
orderDescription.setOrderId("ProductionOrder01");
orderDescription.setTargetCluster("DefaultProductionCluster");
```

```
MyOrderListener orderListener;
OrderManager::getInstance()->recoverOrder(&orderDescription, &orderListener);

// After recoverOrder() has been called for each order to be recovered,
// call the following to initiate recovery for all orders and to start
// receiving order statuses.
OrderManager::getInstance()->receiveRecoveredStatuses();
```

## Order Management using C

```
/* Submit Production order.*/
/* Image order management is almost identical to Production order management,
and is not covered here.*/
/* Create the XML Production order document using any of the available XML
parsers.*/
LPCTSTR xmlOrder = createProductionOrder();
/* Allocate memory for the structure.*/
RCA_production_order_description rcaPOD;
/* Callback function that will receive order status notifications.*/
void CALLBACK orderStatusCallback(LPTSTR xmlOrderStatus)
{
    /* Check for the state of your order here, using an XML parser.*/
    /* This code will run in a Client API managed thread different from
       the thread that submitted the order.*/
    .
    .
    printf("\n--- Order status");
    printf(xmlOrderStatus);
}

/* Submit the Production order.*/
/* If recovery is not required, use submitOrder() signature.*/
/* rcaPOD is filled in with values from the order. Use it in all subsequent
calls related to this order.*/
RCA_submitDurableProductionOrderBstr(&rcaPOD, xmlOrder, orderStatusCallback);

/* Order is being processed by the server and orderStatusCallback is receiving
statuses */
.
.
/* Cancel order. */
/* If we want to cancel the order for some reason. */
/* If the order is cancelled by the server, the next status received
by MyOrderListener will be CANCELLED. */
RCA_cancelProductionOrder(&rcaPOD, true);

/* Stop listening for order status */
/* Order is either COMPLETED, FAILED, or CANCELLED.
/* We need to stop listening for this order's status */
RCA_stopListeningForProductionOrder(&rcaPOD);
```

```

/* Recover orders */
/* If your client application shut down before order(s) were completed,
and you used OrderManager.submitDurableOrder() to submit the order(s),
then you can recover order statuses that were not received previously.*/

/* The sequence for recovering orders is as follows: */
/* 1. Call RCA_recoverProductionOrder() or RCA_recoverImageOrder() for each
order to be recovered. */
/* 2. Call RCA_receiveRecoveredStatuses() to initiate order recovery and to
start receiving order statuses.*/
/* Using the same mechanisms to create RCA_production_order_description and
callback function as in the above section, call the following for each order
you want to recover.*/
RCA_production_order_description rcaPOD;
rcaPOD.orderId = malloc(128);
rcaPOD.targetCluster = malloc(128);
strcpy(rcaPOD.orderId, "ProductionOrder1");
strcpy(rcaPOD.targetCluster, "DefaultProductionCluster");
RCA_recoverProductionOrder(&rcaPOD, orderStatusCallback);

/* Once all the orders have been recovered, call the following to start
receiving order statuses.*/
RCA_receiveRecoveredStatuses();

/* Free order description memory.*/
free(rcaPOD.orderId);
free(rcaPOD.targetCluster);

```

## Order Management using VB 6

- Note:** Rimage recommends the use of the .NET API for VB.NET.

```

` Submit Production order
` Image order management is almost identical to Production order
` management, and is not covered here.
` Declare the structure
Public Type RCA_production_order_description

    orderId As String * 128          ' This field cannot be NULL
    clientId As String * 128         ' This field is optional and can be NULL
    originator As String * 128       ' This field is optional and can be NULL
    targetCluster As String * 128    ' This field cannot be NULL
    targetServer As String * 128     ' This field is optional, if null then any
                                    ' server in the cluster will process this
order
    priority As String * 32          ' Default for this field is
    PRIORITY_MEDIUM
    action As String * 32            ' Default for this field is ACTION_RECORD
    mediaType As String * 32         ' Default for this field is MT_CDR
    mediaSize As String * 32         ' Default for this field is MS_120
    targetLine As String * 32        ' Default for this field is TL_ANY
    copies As String * 32            ' Default for this field is 1

```

```

        inOutInputBin As String * 32      ' Default for this field is IOB_ANY
        outputMailslot As String * 32     ' Default for this field is 0

End Type

' Declare the functions
Public Declare Function RCA_submitProductionOrderBstr Lib
"RmClient_7_3_n_3.dll" _
    (ByRef orderDesc As RCA_production_order_description, _
    ByVal xmlOrder As String, _
    ByVal orderStatusCallback As Long) As Integer

Public Declare Function RCA_cancelProductionOrder Lib "RmClient_7_3_n_3.dll" _
    (ByRef orderDesc As RCA_production_order_description, _
    ByVal abortCurrent As Boolean) As Integer

Public Declare Function RCA_stopListeningForProductionOrder Lib
"RmClient_7_3_n_3.dll" _
    (ByRef orderDesc As RCA_production_order_description) As Integer

' Declare callback function for order statuses
Public Function orderStatusCallback(ByVal orderStatus As String) As Long
    'On Error Resume Next
    Debug.Print orderStatus
    orderStatusCallback = 0
End Function

' Create a RCA_production_order_description structure.
' The values of this structure will be filled in during the
' submitProductionOrderBstr() call
Dim pod As RCA_production_order_description
Dim ret As Integer
Dim xmlProductionOrder As String

' Create the XML Production order document using any of the available XML
parsers.
xmlProductionOrder = createProductionOrder(...)

' Submit the Production order.
' If recovery is not required, use submitOrder() signature
ret = RCA_submitProductionOrderBstr(pod, xmlProductionOrder, AddressOf
orderStatusCallback)
' Order is being processed by the server and orderStatusCallback is receiving
statuses
' Cancel order.
' If we want to cancel the order for some reason.
' If the order is cancelled by the server, the next status received
' by MyOrderListener will be CANCELLED.
ret = RCA_cancelProductionOrder(pod, True)

```

```
` Stop listening for order status
` Order is either COMPLETED, FAILED, or CANCELLED
` We need to stop listening for this order's status
ret = RCA_stopListeningForProductionOrder(pod)

` Recover orders.
` If your client application shut down before order(s) were completed,
` and you used OrderManager.submitDurableOrder() to submit the order(s),
` then you can recover order statuses that were not received previously.
` The sequence for recovering orders is as follows:
` 1. Call RCA_recoverProductionOrderBstr() or RCA_recoverImageOrderBstr() for
each order to be recovered.
` 2. Call RCA_receiveRecoveredStatuses() to initiate order recovery and to
start receiving order statuses.

` Declare the function
Public Declare Function RCA_recoverProductionOrderBstr Lib
"RmClient_7_3_n_3.dll" _
    (ByRef orderDesc As RCA_production_order_description, _
    ByVal callback As Long) As Integer
Public Declare Function RCA_receiveRecoveredStatuses Lib "RmClient_7_3_n_3.dll"
() As Integer
` Using the same mechanisms to create RCA_production_order_description and
callback function as in the above section, call the following for each order
you want to recover.
Dim pod As RCA_production_order_description
Dim ret As Integer

pod.orderId = "ProductionOrder001_VB"
pod.targetCluster = "DefaultProductionCluster"
ret = RCA_recoverProductionOrderBstr(&rcaPOD, orderStatusCallback)
` Once all the orders have been recovered, call the following to start
receiving order statuses.
ret = RCA_receiveRecoveredStatuses()
```

## Server Status and Control Protocol

This section describes the protocol used to communicate with Production Server to obtain status information, modify parameter settings, control orders, and carry out Production Server operations. These operations include sending a response to any dialog that the Production Server may generate in the processing of orders.

This section also describes the protocol used to communicate with the Imaging Server to obtain status information, modify parameter settings, control orders, and carry out Server operations.

The XML-based Production and Imaging server Status and Control protocols are a means for applications to do any of the following:

- Get server status
- Get server parameters
- Set server parameters
  - **Note:** Not all parameters are changeable via the Status and Control protocol. For example, the `ServerID` is a read only parameter.
- Get a list of orders currently in process by a server
- Cancel or suspend a specific order
- Control a server state (Pause, Resume, Shut down, etc.)

The Server Status and Control protocol is realized through using the `ServerManager.executeServerRequest` method in combination with the following DTDs:

- `ProductionServerRequest` DTD
- `ProductionServerReply` DTD
- `ImageServerRequest` DTD
- `ImageServerReply` DTD

## Server Command Synchronization

There is a set of commands for the Production Server and a set for the Imaging server. There is no restriction on which user may send commands. If a command requires a password, the encrypted password is included in the request XML and is verified by the server before the command is run. These commands are described as elements with supporting attributes in `ProductionServerRequest` and `ImageServerRequest` DTDs. Many commands are identical for the two servers, i.e. Get server parameters. In these cases, the elements in the two DTDs are the same. Most of the commands are synchronous. When a client makes a request, the response from the Production or Imaging Server is typically immediate or takes place within a few seconds.

- `ServerManager.executeServerRequest` method returns an XML string that conforms to either the `ProductionServerReply` or the `ImageServerReply` DTD. There is also a set of replies that are the same between the two servers. In these cases, the elements in the two DTDs are the same.
- The **FlashUpload** command, which may take longer to complete, requires the client to wait until the operation completes.
- The `SystemManager.setSynchronousTimeout` method is used to change the timeout value, the value is given in milliseconds.

- The **PauseServer**, **ResumeServer**, and **StopServer** operations are implemented as asynchronous commands. The normal server response is immediate and the response message indicates that these operations are in progress. Because these operations can take a while to complete, clients using this server receive notification through **ServerEventListener.onServerxxx()** methods when the pause or resume operation is complete. In the meantime, any user can make requests for Server status that return the state of the Server. Because of the nature of the pause and resume operations, the Server locks out a small number of commands until the pause or resume operation has completed. These commands are also disabled when the Server is processing a **StopServer** request or during initialization in which the Server is in the 'Start Pending' state.

The commands that are locked out during pause or resume operations are:

- CancelOrder
- ChangeOrder
- ResetInputBins
- EnableDevice
- SetParameter
- FlashUpload
- StopServer

## Password Protection on Commands

Some server commands are password protected. Each Server has its own password to use any of the password protected commands. When there is more than one Server in a configuration, the Servers can have the same password or they can all be different. A password may consist of up to 20 characters. The default password is a null password, meaning that it is not initially set and has a length of zero (or 0 characters). The protocol uses the **SetServerPassword** command to permit a client application to set the password. When the Server password is set, this password is required for performing protected operations on the Server.

When used in an XML message, passwords must be encoded. Clients that send passwords must encrypt the password prior to sending it.

## Production Server Commands

When Production Server processes either the **EnableDevice** or the **ResetInputBins** commands, Production Server may send a completed response even though the operation on the device involved has not completed. The device cannot accept another command unless it is ready, so there may be a delay before the next operation begins.

For the Production Server to respond to any request, it must first be online (successfully connected to the Messaging Server).

- **Note:** Typically, the Server is online before it has completely initialized, this may cause some commands to return an error until the Server has completed initialization.



## Command Summary

Command	Description	Needs Password	Command Reply
GetServerStatus	Gets information about Server's operational state and system settings	No	Server Password Set = (True or False) Remaining days for trial use of a feature returned. Overall counts of produced discs and rejected discs. Command line switches that are in use.
GetParameterSettings	Gets list of parameters and their values	No	Sends the contents of the element ProductionServerParameters in the ProductionServerReply.dtd
GetOrderList	Obtains a list of currently running (or suspended) orders on the Server for all users	No	OrderState = IN_PROCESS or CANCELLING OrderStage = BUSY, WAITING, RECORDING, PRINTING
CancelOrder	Cancels an order	Yes, if client updates someone else's order	
ChangeOrder	Suspends, resumes, or changes quantity of an order	Yes, if client updates someone else's order	
SetDialogAction	This command is used to respond to either an alert dialog or an error dialog that is posted by the Production Server.	No	COMPLETED
GetSessionLog	Obtains a list of the most recent log messages posted by the Server	No	Returns most recent log messages posted since startup of the client, up to 200 entries.
ResetInputBins	Causes Server to reset input bins for a specified autoloader that has any bin set up for both input and output.	No	COMPLETED
EnableDevice	Re-enables a autoloader, recorder, or printer	Yes	COMPLETED
SetParameter	Modifies one or more parameter settings	Yes	Returns an error for invalid requests.
PauseServer	Stops order scanning; Pause mode for	Yes	IN_PROCESS

Command	Description	Needs Password	Command Reply
	Service		
ResumeServer	Starts order scanning; Run mode for Service	Yes	IN_PROCESS
StopServer	Stops order scanning, stops order processing, and shuts down Server. In process orders are canceled.	Yes	
FlashUpload	Uploads firmware to autoloader, recorders, or printers	Yes	
SetServerPassword	Initially sets, changes, or resets the Server password	Only to change or reset password	
VerifyServerPassword	Confirms if the current password provided is still valid or not.	No	

## Command Reply

All command replies contain the following information:

- **ServerId** – Unique ID assigned to the Production Server. This has the format <computer\_name>\_<base id>. The <base id> is the name assigned to the Server containing up to 4 characters.
- **ClientID** – Name of the client user that sent the original command. Each client ID is unique on the Messaging Server.
- **CommandState** – Refers to the state of the command at the time of the response. The only state that indicates command success is the COMPLETED state. The FAILED state indicates that the request failed, the command may or may not have run. The details of any failure are in the error code and message. If the command or request is started but not immediately completed, the *CommandState* is IN\_PROCESS.
- **CommandErrorCode** – A number ranging from 0 – 999 indicating the error status of the command. A value of 0 is returned if the command has the COMPLETED state. The code has a non-zero value when the state is FAILED.
- **CommandErrorMessage** – Text indicating what the error message is. It is present if the *CommandErrorCode* is non-zero.
- **ReplyTimestamp** – Text in the form of “CCYY-MM\_DD HH:MM:SS” indicating the time the Server generated a reply.
- **Automation** – The state of Server operation.

Automation State	Description
START_PENDING	Server is in initialization.
RUNNING	Server is scanning for orders and/or processing orders.
PAUSED	Server is not scanning for new orders; it has suspended any orders that were processing.
PAUSE_PENDING	Server is not scanning for more orders; it is in the process of suspending any active orders.
STOP_PENDING	Server is not scanning for more orders; Recordings in process are being canceled; The system is in the process of shutdown.

## Command Details

This section discusses Production Server request commands and is intended to clarify some of the less obvious details about data processed or returned by the Imaging Server.

- **GetServerStatus** – Obtains information about Server software settings and configuration.

*GetServerStatus* is used to obtain information about the software, how the software is configured to run, and some configuration information. Most of the settings are not directly configurable by the client, but some dynamic status information is available on the Server and attached devices. This command returns overall counts of produced (good) discs and rejects. As an option, device status is available which contain data on loader bin levels and printer disc cumulative throughput.

One item returned by this request is whether or not the Server password is set. When the server password attribute is true, the Server requires a server password to be submitted with some commands. When a password is not set, then a password is not used with any request.

One option available with this request is information pertaining to license activation. If a feature has a license file but is not activated, the number of remaining days for trial use of the feature is returned.

A less-used option is to send a flag with this request to get the command line switches that are in use. These switches are sometimes used by QA and development personnel doing diagnostic work on the Server.

- **GetParameterSettings** – Obtains a complete list of parameters and their settings.

The Server replies by sending the contents of the element 'ProductionServerParameters' in the ProductionServerReply.dtd.

- **GetOrderList** – Obtains a list of all currently running (or suspended) orders on the Server.

This command provides current data on all of the orders being processed by the Production Server. The orders in the list are not in any particular sequence. The *OrderID* and *ClientID* for each order are returned. These values are needed whenever making a request to cancel or to update an order.

*Order State* – If the Server has resources available to process an order, the normal state of the order is 'IN\_PROCESS'. If an action is taken to cancel an order, the state remains as 'IN\_PROCESS' but the extended order stage (OrderStageEx) goes to *CANCELLING*. After the order has completed the cancellation process, it is removed from the order list.

*Order Stage* – This attribute of an order indicates the predominant activity that best describes what is happening with the order at the Server.

Two stages to note are:

1. **WAITING:** means that the disc(s) are waiting for a physical resource on the autoloader before they can go to the next stage. This may occur for an extended period if an order loses its resources due to a hardware malfunction or during a crash recovery when there may be several orders running, but not enough resources to run them all at once.
2. **BUSY:** usually means that the disc is in a transition from one stage to another (such as *RECORDING* to *PRINTING*).

- **CancelOrder** – Cancels an active order.

This command allows a client to cancel an order that the client has submitted without a password. The protocol allows the choice to either stop the recording immediately (`AbortRecordingsInProgress = true`), or to stop after the current discs being recorded are finished (`AbortRecordingsInProgress = false`).

The Server checks if the client that is making the request matches the *ClientID* that is associated with the order. If the clients do not match, then the Server verifies that the message contains the Server password (provided this password has been set).

- **Note:** Both the *OrderID* and *ClientID* attributes are used to properly identify orders. This information is obtainable with the **GetOrderList** command.

- **ChangeOrder** – Suspends, resumes, or changes quantity of an order.

When changing the number of discs to produce in an order, the requested number may be more or less than the number in the order. However, the quantity specified must *exceed* the sum of the discs already produced in the order plus the number of discs currently in recording (`Discs (new quantity) > {Discs completed + Discs recording}`.)

The Server checks to see if the client that is making the request matches the *ClientID* that is associated with the order. If the clients do not match, then the Server verifies that the message contains the Server password (provided this password has been set).

- **Note:** Both the *OrderID* and *ClientID* attributes are used to properly identify orders, and this information is obtainable with the **GetOrderList** command.

- **SetDialogAction** – Sends a selection a user has made to a dialog.

This command is used to respond to either an alert dialog or an error dialog that is posted by the Production Server. There is no restriction on which client can respond. Normally, with an error dialog, the condition must be corrected at the affected autoloader before answering the dialog. After the Server receives this command, it returns the COMPLETED status and performs the specified action.

- **Note:** The completed status is simply the Server receiving a course of action, rather than a repeat of the action that caused the error. If the course of action produces another error, then another dialog is posted.

- **GetSessionLog** – Obtains a list of the most recent log messages posted by Server.

This request causes the Server to return its most recent messages (that it normally logs into a file) that have been posted since startup to the client. Up to 200 entries may be returned. The client can specify a value for the number of entries, but if the actual number is less than the value specified, then only the actual log entries are returned (up to a maximum of 200).

Each log entry has a *Timestamp* and a *MessageId* associated with it. The timestamp has the format "CCYY-MM-DD HH:MM:SS". The message ID is the numeric value that is assigned to the message. A value of zero for the message ID indicates that the log entry is an informational message, although informational messages can also have non-zero IDs.

- **ResetInputBins** – Causes the Server to perform the reset input bins operation on a specified autoloader. This command applies to only autoloaders that have any bin configured for both input and output.

The bins on a specified autoloader are essentially reinitialized. It is recommended that the client application confirm with the local operator that the bins have been emptied and refilled as required. When the Server receives this request, it immediately proceeds with the reset operation regardless of the state that the autoloader is in.

- **Note:** This command typically returns COMPLETED immediately, although the physical operation can take a few seconds.

- **EnableDevice** – Enables a disabled autoloader, recorder, or printer.

This command allows a user to put a disabled autoloader, recorder, or printer back online. It is always necessary to specify the number of the autoloader. The Server assigns numbers to autoloaders during initialization. The first autoloader is numbered as 1. Recorder numbers are assigned by the Server and refer to the physical locations of the recorders on the autoloader. Both recorders and printer are numbered starting at 1.

- **Note:** This command typically returns COMPLETED immediately, although the physical operation can take a measurable amount of time.

- **SetParameter** – Modifies one or more parameter settings.

This command allows a user to change Production Server settings. As viewed in the 'Setting' sub element of the element *SetParameter*, a single command can be used to change multiple settings on the Server or on one or more autoloaders that are connected. When a request is made to modify more than one parameter setting, all of the requested settings must be valid for the request to be completed. If any of the requested settings is invalid, the request returns an error, and **no settings** are changed.

- **PauseServer** – Stops order scanning and brings the Production Server to a Paused automation state.

The Server returns an immediate response. The *CommandState* normally is IN\_PROCESS. Once the command is started, the Server goes into the Pause Pending state until it reaches the Pause state. The command fails if the Server is in either the Start Pending or Stop Pending state.

- **ResumeServer** – Resumes order scanning and brings the Production Server to a Running automation state.

If any orders are suspended, the orders are resumed. The Server returns an immediate response. The *CommandState* normally returns as IN\_PROCESS. However, the Server normally should reach the running state within a few seconds. The Server must already be in the Paused state for this command to succeed.

- **StopServer** – Shuts down the Server.

The Server shuts down when it receives this command. Any orders in process are canceled and no new orders are picked up. The client has the option of aborting any recording in progress or allowing the current discs in recording to finish.

- **FlashUpload** – Uploads the firmware to the autoloader, recorders, or printers.

No matter what type of device is specified for uploading flash firmware, the Server always attempts to update **all** similar devices on a system. If the autoloader firmware is being uploaded, then all the autoloaders that match the firmware can be updated. As with the *EnableDevice* command, the first autoloader is numbered as 1.

Before running this command, the client application should ensure that the firmware file has been copied to the Rimage system folder. The Production Server accesses the file using the full file path that is specified in the request.

- **Note:** The Server must be in a Paused state before this command can be run. This command can take more than three minutes to complete. It does not return until the operation has succeeded or failed. The client application should expect to wait until completion.

- **SetServerPassword** – Initially sets, changes, or resets the password of the Server.  
Initially, the Server password is not set. This command allows any client to set the password so that the Server must receive this password with any password-protected commands that follow. The fact that the password is set on the Server is known by the response to the **GetServerStatus** command.  
The Server password must be 0 to 20 characters in length. The [encryption algorithm](#) is provided on page 75.
- **VerifyPassword** – Checks to see if the entered password is correct.  
This command is used by client applications to determine if a string of characters matches the Server's stored password.

## Imaging Server Commands

Before the Imaging Server can respond to any request, it must be online. This means that it has completed its initialization process and has successfully connected to the Messaging Server.

### Command Summary

Command	Description	Needs Password
GetServerStatus	Obtains information about Server's operational state and system settings	No
GetParameterSettings	Obtains list of parameters and their values	No
GetOrderList	Obtains the job order of the currently processing order, if there is one	No
CancelOrder	To cancel the current order	Yes, if client updates someone else's order
GetSessionLog	Obtains a list of the most recent log messages posted by Server	No
SetParameter	Modifies one or more parameter settings	Yes
PauseServer	Stops order scanning; Pause mode for Service	Yes
ResumeServer	Starts order scanning; Run mode for Service	Yes
StopServer	Stops order scanning, stops order processing, and shuts down Server	Yes
SetServerPassword	Initially sets, changes, or resets the Server password	Only to change or reset password
VerifyServerPassword	Used to confirm if the current password that was provided is still valid or not	No

## Command Reply

All responses contain the following information. See the DTD for the exact format.

- **ServerId** – Unique ID assigned to the Production Server. This has the format <computer\_name>\_<base id>; the <base id> is the name up to 4 characters that is assigned to the Server. The Imaging Server uses IS01 by default, but may be changed during installation.
- **ClientID** – The name of the client user that sent the original command. It is unique on the Messaging Server.
- **CommandState** – The state of the command at the time of the response. The only states that indicates command successes are the COMPLETED or IN\_PROCESS states. The FAILED state indicates that the request failed. The command may or may not have run. The details are in the error code.
- **CommandErrorCode** – A number indicating the error status of the command. A value of 0 is returned if the command has the COMPLETED state. The code has a non-zero value when the state is FAILED.
- **CommandErrorMessage** – Text indicating what the error message is. It is present if the *CommandErrorCode* is non-zero.
- **ReplyTimestamp** – Has the format CCYY-MM-DD HH:MM:SS. It indicates the time that the reply was sent.
- **Automation** – The current run state of the Server. Possible automation states are *StartPending*, *Running*, *Paused*, *PausePending*, or *StopPending*.

The actual state of the Imaging Server is in the Automation attribute. Automation states are:

Automation	Description
StartPending	The Server has loaded and is preparing to run. This state usually only lasts a few seconds or less.
Running	Server is scanning for orders and/or processing orders
Paused	Server is not scanning for new orders; it has suspended any jobs that were processing
PausePending	Server in not scanning for new orders; it is finishing up processing orders
StopPending	Server is not scanning for new orders; Orders in process are ending; The system starts to shutdown

## Command Details

This section discusses Imaging Server request commands.

- **GetServerStatus** – Obtains information about Server's operational state and system settings.
- **GetOrderList** – Obtains a list of all currently running orders on the Server.

The Imaging Server currently processes only one order at a time. This command returns the current order, if there is one. A future version may work on several jobs concurrently.

This command provides current data on all of the orders that are being processed by the Imaging Server. The orders in the list are not in any particular sequence. The *OrderID* and *ClientID* are both needed to identify the order. When an operation is done with the **ChangeOrder** command, these values are necessary.

*Order State* – If the Server is working on a job, it has the state of ACTIVE. If an action is taken to cancel an order, the order has the state of CANCELLING for a few seconds until the job is cleaned up and the output image file deleted. Following this, the order is removed from the order list.

- **CancelOrder** – Cancels the current order.

The command allows a client to cancel the order that is currently being processed.

The Server checks if the client making the request matches the *ClientID* associated with the order. If the clients do not match, then the Server verifies that the message contains the Server password (provided this password has been set).

- **Note:** Both the *OrderID* and *ClientID* attributes are used to properly identify orders. This information is obtainable with the **GetOrderList** command.

- **GetSessionLog** – Obtains a list of the most recent log messages posted by Server.

This request causes the Server to return its messages (that it normally logs into a file to the clients) that have been posted since startup. Up to 200 entries are returned. The client can specify a value for the number of entries, but if the actual number is less, then only the actual number of log entries is returned, up to a maximum value of 200.

- **PauseServer** – Stops order scanning and brings the Imaging Server to a Pause mode when it runs as a Service.

The Server returns an immediate response. The *CommandStatus* normally is IN\_PROCESS. When the command is started, the Server goes into the Pause Pending state until it reaches the Pause state.

It is a command error to attempt a Pause command while the state is Pause Pending.

- **SetServerPassword** – Initially sets, changes, or resets the password of the Server.

Initially, the Server password is not set. This command allows any client to set the password so that the Server then must have the password before it performs certain commands. The fact that the password is set on the Server is known by the response to the **GetServerStatus** command.

Any string of characters may be used, including Asian Unicode. The [encryption algorithm](#) is provided on page 75.



# Deployment

## Java Deployment

### Build Information

The Client API .jar files have been compiled for target Java VM version 1.4 or higher.

### Required Files

The .jar files, Import statements and Properties files required by Java in the Class path to compile and run are listed below.

Required JAR Files	Required Import Statements	Optional Property Files
AdminApi.jar	import com.rimage.client.api.*;	log.properties
RmClient_8_0_n_1.jar	import com.rimage.client.api.exception.*;	rmapi_log.properties
RmRmsApi_1_3_n_1.jar	import com.rimage.msg.exception.*;	
RmRmsClient_1_3_n_1.jar	import com.rimage.exception.*;	
test.jar		
CommonApp.jar		

## .NET Deployment

### Build Information

The Rapid Integration API has been compiled using Visual Studio 2008 and .NET Version 3.5.

Rimage.Client.Api assembly implements the .NET API. This assembly can be used in any application written in a .NET supported language.

This assembly is strongly named, which among other things means that Common Run Time (CLR) takes the Assembly version of this assembly into account at load time.

### Required .NET Assembly Files

The following files are required in C#, VB.NET, or any other .NET project.

Installed by default in C:\Program Files\RimageSdk\ApiSdk\bin.

Rimage.Client.Api.dll

The rest of the file list is identical to the [Unicode](#) list in [C++ Required Files and Folders](#) section.

## C / C++ / VB 6 Deployment

### Build Information

The Client API has been compiled using Microsoft Visual Studio 2008 (VC9) compiler. Rimage DLLs include their version in the name of the file. The name/version has the following format:

<name>\_<major>\_<minor>\_n\_<interface>.dll

- Major version is seldom incremented, and only if a Rimage system undergoes a significant architectural change. For example, version 5.x to version 6.x – the Rimage system changed from file based to messaging/XML based.

- Minor version is incremented if a DLL is changed for a new release. Applications using this DLL need to be rebuilt.
  - “n” represents an internal build/bug fix version for a specific minor version. The actual File version of the dll has a number in place of “n”. For example if the dll is named RmClient\_8\_0\_n\_5.dll, the File version of this dll could be 8.0.n.5.
  - Interface version represents iterations of the API itself. If the exported interface of the DLL itself is changed, this version is incremented and the applications using this DLL needs to be rebuilt.
- **Note:** The `_u` option indicates Unicode versions; no `_u` indicates non-Unicode versions.

## Required Linker Options

If your application is intended to run on Windows XP Service Pack 2, then do the following:

In Project Properties > Linker > Command Line > Additional options, enter `/SAFESEH:NO`.

- **Note:** This is a workaround for the Service Pack 2 Exception handling problem.

## Required Files and Folders

The following files and directories are required in VB 6, C and C++ projects. Specify the paths and specify the .lib files (either Unicode or non-Unicode) as indicated.

### Required DLL Files (Non-Unicode)

Installed by default at:

C:\Program Files\RimageSdk\ApiSdk\bin  
and \bin (x64).

RmClient\_8\_0\_n\_5.dll

RmRms\_1\_3\_n\_1.dll

### Required DLL Files (Unicode)

Installed by default at:

C:\Program Files\RimageSdk\ApiSdk\bin  
and \bin (x64).

RmClient\_8\_0\_n\_5\_u.dll

RmRms\_1\_3\_n\_1.dll

## Microsoft visual C++ 2008 SP1 Redistributable Pack is Required.

### Required LIB Files (Non-Unicode)

- **Note:** This section does not apply to VB deployment.

RmClient\_8\_0\_n\_5.lib

### Required LIB Files (Unicode)

- **Note:** This section does not apply to VB deployment.

RmClient\_8\_0\_n\_5\_u.lib

## Required Include Directories

- **Note:** This section does not apply to VB or .NET deployment.

Installed by default at:

- C:\Program Files\RimageSdk\ApiSdk\include\client
- C:\Program Files\RimageSdk\ApiSdk\include\exception

## Required #include Statements

- **Note:** This section does not apply to VB or .NET deployment.

`#include <ClientApiInclude.h>` (must be specified in project settings for C++)

`#include <ClientApi_C_Include.h>` (must be specified in project settings for C)

## Optional files

`rmapi_log.properties`

Place this file in your application's working folder to produce a Client API log file. This file can be found at:

`C:\Program Files\RimageSdk\ApiSdk\bin` folder.

## 64 bit deployment

SDK 8.1 includes x86 and x64 DLLs and libs.

File system location for x86 files:

`C:\Program Files\RimageSdk\ApiSdk\bin` and `\lib`

File system location for x64 files:

`C:\Program Files\RimageSdk\ApiSdk\bin (x64)` and `\lib (x64)`

## Appendix A – Sample Source Code Projects

Rimage SDK install includes sample projects for working with the Rapid API. By default these projects are placed in the `C:\Program Files\RimageSdk\ApiSdk\Samples\ClientApi` folder. The samples are broken into Java, C++, and .NET (written in C#) samples.

## Appendix B – Sample XML Documents

### Image Order Samples

The examples in this section include:

- [XML ISO L2 with EditList Image Order](#)
- [XML ISO L2 from Parent Folder Image Order](#)
- [XML RockRidge Image Order](#)

#### ISO L2 with Editlist Image Order

The example below shows an XML order including ISO L2 with an EditList Image.

```
<?xml version="1.0" ?>
<!--Sample XML file generated by XML Spy v4.2 U (http://www.xmlspy.com)-->
<!DOCTYPE ImageOrder SYSTEM "C:\Rimage\XML\ImageOrder_1.11.DTD">
<ImageOrder
    Priority="Normal"
    OrderId="Project1_IO"
    ClientId="SOFTWARE4_QuickDiscJ"
    Originator="SOFTWARE4_QuickDiscJ">
<Target Cluster="DefaultImageCluster" Server="ANY"/>
<Format>
    <PCMACFormat ISO="2" Apple="none" Joliet="false" Rockridge="false"/>
    <FormatOptions ForceUpperCase="false" AllowMultipleFilePaths="true"
ForceDot="true" ForceShort="false"
    Versions="true" IgnoreBadFiles="false" CaseSensitive="false" Zip="false"
AllowBootableCD="true"/>
</Format>
<Source>
    <EditList EditListPath="\\Mainserver\D_drive\tmp\test.edl"/>
</Source>
<Output Type="Normal" CDXA="false" Postgap="true" Size="74"
ImageFile="\\Mainserver\D_drive\tmp\test.img"/>
<Rules CheckNames="false" AllowDirExt="false" CheckLevels="false"/>
<VolumeName/>
</ImageOrder>
```

#### ISO L2 from Parent Folder Image Order

The example below shows an XML order including ISO L2 image from the Parent folder.

```
<?xml version="1.0"?>
<!--Sample XML file generated by XML Spy v4.2 U (http://www.xmlspy.com)-->
<!DOCTYPE ImageOrder SYSTEM "C:\Rimage\XML\ImageOrder_1.11.DTD">
<ImageOrder OrderId="IO1234" ClientId="ClientID" Originator="Tester"
Priority="Normal">
    <Target Cluster="DefaultImageCluster" Server="ANY"/>
    <Format>
        <PCMACFormat ISO="2" Apple="none" Joliet="false" Rockridge="false"/>
        <FormatOptions ForceUpperCase="false" AllowMultipleFilePaths="true"
ForceDot="true" ForceShort="false" Versions="true" IgnoreBadFiles="false"
CaseSensitive="false" Zip="false" AllowBootableCD="true"/>
    </Format>
</ImageOrder>
```

```

</Format>
<Source>
  <ParentFolder ParentFolderPath="c:\tmp\thumbnails" Destination="both"/>
</Source>
<Output ImageFile="c:\tmp\test.img" Type="Normal" CDXA="false"
Postgap="true" Size="74"/>
<Rules CheckNames="true" AllowDirExt="false" CheckLevels="true"/>
<VolumeName VolName="LabelTest"/>
<PVDInfo PVDSystem="System" PVDVolumeSet="VolumeSet" PVDCopyright="Copyright
Martin Nohr" PVPublisher="Publisher" PVPreparer="Preparer"
PVDAApplication="Application" PVDAbstract="Abstract"
PVD Bibliography="Bibliography" PVDEExpirationDate="" PVDEffectiveDate=""
GMTOffset="-32"/>
</ImageOrder>

```

## RockRidge Image Order

The example below shows an XML order including a RockRidge image.

```

<?xml version="1.0" ?>
<!--Sample XML file generated by XML Spy v4.2 U (http://www.xmlspy.com)-->
<!DOCTYPE ImageOrder SYSTEM "C:\Rimage\XML\ImageOrder_1.11.DTD">
<ImageOrder OrderId="IO1234" ClientId="ClientID" Originator="Tester"
Priority="Normal">
  <Target Cluster="DefaultImageCluster" Server="ANY"/>
  <Format>
    <PCMACFormat ISO="2" Apple="none" Joliet="false" Rockridge="true"/>
    <FormatOptions ForceUpperCase="false" AllowMultipleFilePaths="true"
ForceDot="true" ForceShort="false" Versions="true" IgnoreBadFiles="false"
CaseSensitive="false" Zip="false" AllowBootableCD="true"/>
  </Format>
  <Source>
    <EditList EditListPath="c:\tmp\test.edl" Destination="both"/>
  </Source>
  <Output ImageFile="c:\tmp\test.img" Type="Normal" CDXA="false"
Postgap="true" Size="74"/>
  <Rules CheckNames="true" AllowDirExt="false" CheckLevels="true"/>
  <VolumeName VolName="LabelTest"/>
  <PVDInfo PVDSystem="System" PVDVolumeSet="VolumeSet" PVDCopyright="Copyright
Martin Nohr" PVPublisher="Publisher" PVPreparer="Preparer"
PVDAApplication="Application" PVDAbstract="Abstract"
PVD Bibliography="Bibliography" PVDEExpirationDate="" PVDEffectiveDate=""
GMTOffset="-32"/>
</ImageOrder>

```

## Production Order Samples

This section includes examples of the following XML Production Order types:

- [XML Audio Production Order](#)
- [XML Blue Book Production Order](#)
- [XML Mode 1 Production Order](#)
- [XML Print Only Production Order](#)
- [XML Data Disc Production Order](#)

These are provided in detail in the following section.

A sample Production Order DTD would take up to 125 lines or more to illustrate here. To simplify things, Rimage XML DTDs make extensive use of defaults when defining a DTD. Rimage also ships “cookie cutter” XML documents with the Messaging Server software to give customers a higher level starting point.

- **Note:** It is the end user's responsibility to validate the XML strings before sending the XML document.

The 125-line Production Order DTD example mentioned above is provided as a sample Production Order XML document. It is actually an instance of the Production Order DTD and would occupy only 16 lines. It would look like this:

```
<?xml version="1.0"?>
<!DOCTYPE ProductionOrder SYSTEM "C:\Rimage\XML\ProductionOrder_1.14.dtd">
<ProductionOrder OrderId="POOrder_1" ClientId="kbttest" Originator=" "
Copies="1">
  <Media Type="CDR" />
  <Target/>
  <Action>
    <Record>
      <WriteTrack Filename="c:\rimage\cd-r_images\Order_1.img"
DeleteAfterRecording="false">
        <Data Type="Model">
          <VolumeId volume_id="Mydisc"/>
        </Data>
      </WriteTrack>
    </Record>
  </Action>
  <Action>
    <Record>
      <Fixate Type="SAO" Final="true"/>
    </Record>
  </Action>
  <Action>
    <Label Filename="myfirst_label">
      <BTW Merge_Filename="c:\rimage\labels\merge.txt"
DeleteMergeFileOnCompletion="true"/>
    </Label>
  </Action>
</ProductionOrder>
```

## Audio Production Order

```
<?xml version="1.0"?>
<!DOCTYPE ProductionOrder SYSTEM "c:\rimage\xml\ProductionOrder_1.14.dtd">
<ProductionOrder Copies="1" OrderId="Index Test" ClientId="POF-XML"
Priority="Normal" Originator="POF-XML">
  <Media Size="120mm" Type="CDR"/>
  <Target Cluster="DefaultProductionCluster"/>
  <Action>
    <Record>
      <WriteTrack Filename="C:\Rimage\CD-R_Images\WS010002.A01">
        <Audio/> </WriteTrack> </Record>
      </Action>
    <Action>
      <Record>
        <WriteTrack Filename="C:\Rimage\CD-R_Images\WS010002.A02">
          <Audio/> </WriteTrack> </Record>
        </Action>
      <Action>
        <Record>
          <WriteTrack Filename="C:\Rimage\CD-R_Images\WS010002.A03">
            <Audio/> </WriteTrack> </Record>
          </Action>
        <Action>
          <Record>
            <WriteTrack Filename="C:\Rimage\CD-R_Images\WS010002.A04">
              <Audio/> </WriteTrack> </Record>
            </Action>
          <Action>
            <Record>
              <WriteTrack Filename="C:\Rimage\CD-R_Images\WS010002.A05">
                <Audio/> </WriteTrack> </Record>
              </Action>
            <Action>
              <Record>
                <WriteTrack Filename="C:\Rimage\CD-R_Images\WS010002.A06">
                  <Audio/> </WriteTrack> </Record>
                </Action>
              <Action>
                <Record>
                  <WriteTrack Filename="C:\Rimage\CD-R_Images\WS010002.A07">
                    <Audio/> </WriteTrack> </Record>
                  </Action>
                <Action>
                  <Record>
                    <WriteTrack Filename="C:\Rimage\CD-R_Images\WS010002.A08">
                      <Audio/> </WriteTrack> </Record>
                    </Action>
                  <Action>
```



```
<Record>
  <WriteTrack Filename="C:\Rimage\CD-R_Images\WS010002.A09">
    <Audio/>  </WriteTrack>  </Record>
</Action>
<Action>
  <Record>
    <WriteTrack Filename="C:\Rimage\CD-R_Images\WS010002.A10">
      <Audio/>  </WriteTrack>  </Record>
    </Action>
  <Action>
    <Record>
      <Fixate Type="SAO" Final="true"/>  </Record>
    </Action>
  </ProductionOrder>
```

### Blue Book Production Order

```
<?xml version="1.0"?>
<!DOCTYPE ProductionOrder SYSTEM "c:\rimage\xml\ProductionOrder_1.14.dtd">
<ProductionOrder Copies="1" OrderId="Blue Book" ClientId="POF-XML"
Priority="Normal" Originator="POF-XML">
  <Media Size="120mm" Type="CDR"/>
  <Target Cluster="DefaultProductionCluster"/>
  <Action>
    <Record>
      <WriteTrack Filename="F:\Images\WS010012.A01">
        <Audio/>
      </WriteTrack>
    </Record>
  </Action>
  <Action>
    <Record>
      <WriteTrack Filename="F:\Images\WS010012.A02">
        <Audio/>
      </WriteTrack>
    </Record>
  </Action>
  <Action>
    <Record>
      <WriteTrack Filename="F:\Images\WS010012.A03">
        <Audio/>
      </WriteTrack>
    </Record>
  </Action>
  <Action>
    <Record>
      <WriteTrack Filename="F:\Images\WS010012.A04">
        <Audio/>
      </WriteTrack>
    </Record>
```

```

</Action>
<Action>
  <Record>
    <WriteTrack Filename="F:\Images\WS010012.A05">
      <Audio/>
    </WriteTrack>
  </Record>
</Action>
<Action>
  <Record>
    <Fixate Type="SAO" Final="false"/>
  </Record>
</Action>
<Action>
  <Record>
    <WriteTrack Filename="F:\Images\WS010012.D10">
      <Data Type="Model"/>
    </WriteTrack>
  </Record>
</Action>
<Action>
  <Record>
    <Fixate Type="SAO" Final="true"/>
  </Record>
</Action>
<Action>
  <Label Filename="C:\Rimage\Labels\SAMPLE.BTW">
    <BTW SaveAfterRendering="default"/>
  </Label>
</Action>
</ProductionOrder>

```

## Mode 1 Production Order

```

<?xml version="1.0"?>
<!DOCTYPE ProductionOrder SYSTEM "c:\rimage\xml\ProductionOrder_1.14.dtd">
<ProductionOrder Copies="1" OrderId="QD010005" ClientId="POF-XML"
Priority="Normal" Originator="QD01">
  <Media Size="120mm" Type="CDR"/>
  <Target Cluster="DefaultProductionCluster"/>
  <Action>
    <Record>
      <WriteTrack Filename="C:\Rimage\CD-R_Images\QD010005.IMG">
        <Data Type="Model"/>
      </WriteTrack>
    </Record>
  </Action>
  <Action>
    <Record>
      <Fixate Type="SAO" Final="true"/>
    </Record>
  </Action>
</ProductionOrder>

```

```

    </Record>
  </Action>
  <Action>
    <Label Filename="C:\Rimage\Temp\QD010005.BTW">
      <BTW SaveAfterRendering="false"/>
    </Label>
  </Action>
</ProductionOrder>

```

## Print Only Production Order

```

<?xml version="1.0"?>
<!DOCTYPE ProductionOrder SYSTEM "C:\Rimage\XML\ProductionOrder_1.14.dtd">
<ProductionOrder Copies="10" OrderId="Print-Only" ClientId="POF-XML"
Priority="Normal" Originator="POF-XML">
  <Media Size="120mm" Type="CDR"/>
  <Target Cluster="DefaultProductionCluster"/>
  <Action>
    <Label Filename="C:\Rimage\Labels\Barcode.btw">
      <BTW Merge_Filename="C:\Rimage\Labels\Barcode.txt"
SaveAfterRendering="default"/>
    </Label>
  </Action>
</ProductionOrder>

```

## Data Disc Production Order

Below is a Production Order of 1 copy of OrderID "Record Data Disc" from ClientID "POF-XML" with Normal Priority, Originated from "db" on 120mm CDR Media Size targeted for "Dave's" Cluster (*RecordDataDisc.xml*).

```

<?xml version="1.0" ?>
<!-- edited with XMLSPY v5 U (http://www.xmlspy.com) by Jeff Schierman (Rimage Corp.) -->
<!DOCTYPE ProductionOrder SYSTEM "c:\rimage\xml\ProductionOrder_1.14.dtd">
<ProductionOrder Copies="1" OrderId="Record Data Disc" ClientId="POF-XML"
Priority="Normal" Originator="db">
  <Media Size="120mm" Type="CDR"/>
  <Target Cluster="Dave's"/>
  <Action>
    <Record>
      <WriteTrack Filename="E:\Images\ClipArt.IMG">
        <Data Type="Model" MergeSessions="true"/>
      </WriteTrack>
    </Record>
  </Action>
  <Action>
    <Record>
      <Fixate Type="SAO" Final="true"/>
    </Record>
  </Action>
  <Action>
    <Label Filename="C:\Rimage\Labels\CD Creator.btw">
      <BTW SaveAfterRendering="default"/>
    </Label>
  </Action>
</ProductionOrder>

```

```
</Label>
</Action>
</ProductionOrder>
```

## Order Status Samples

XML Image Order status (*dh\_i01.xml*) and Production Order status (*dh\_p01.xml*) samples are provided below.

### Image Order Status

```
<?xml version="1.0"?>
<!DOCTYPE ImageOrderStatus SYSTEM
"\\SWRASKINREST\Rimage\XML\ImageOrderStatus_1.6.dtd">
<ImageOrderStatus OrderId="QD_DHASSELER_ENG-DHASSLER_0142"
ClientId="QD_DHASSELER_ENG-DHASSLER" ServerId="SWRASKINREST_IS01"
Originator="QD_DHASSELER_ENG-DHASSLER" MessagingHost="SWRASKINREST"
MessagingPort="4664" OriginalOrder="">
  <Status State="COMPLETED" CurrentStatus="Writing File 27 of 26"
PercentCompleted="100"/>
  <Timestamps OrderRead="2006-10-19 16:45:00" OrderCompleted="2006-10-19
16:45:09"/>
</ImageOrderStatus>
```

### Production Order Status

```
<?xml version="1.0"?>
<!DOCTYPE ProductionOrderStatus SYSTEM
"\\SWRASKINREST\Rimage\XML\ProductionOrderStatus_1.10.dtd">
<ProductionOrderStatus OrderId="QD_DHASSELER_ENG-DHASSLER_0142"
ClientId="QD_DHASSELER_ENG-DHASSLER" ServerId="SWRASKINREST_PS01"
Originator="QD_DHASSELER_ENG-DHASSLER" MessagingHost="SWRASKINREST"
MessagingPort="4664" OriginalOrder="" SimulatePrint="true"
SimulateWrite="true">
  <Status Stage="RECORDING" State="IN_PROCESS" CopiesCompleted="0"
CopiesRequested="1" PercentCompleted="0"/>
  <Timestamps OrderRead="2006-10-19 16:45:14"/>
  <Device Identifier="Recorder 1, Cache 1" IsStreaming="false"
CurrentState="LOADING"/>
</ProductionOrderStatus>
```

## Spanning XML Samples

### Image Order

```
<?xml version="1.0"?>
<!DOCTYPE ImageOrder SYSTEM "\\SWRASKINREST\Rimage\XML\ImageOrder_1.11.DTD">
<ImageOrder OrderId="QD_DHASSELER_ENG-DHASSLER_0143_I001"
ClientId="QD_DHASSELER_ENG-DHASSLER" Priority="Normal"
Originator="QD_DHASSELER_ENG-DHASSLER">
  <Target Server="ANY" Cluster="DefaultImageCluster"/>
  <Format>
    <PCMACFormat ISO="2"/>
    <FormatOptions Zip="false" ForceDot="false" Versions="false"
ForceShort="false" CaseSensitive="false" ForceUpperCase="false"
IgnoreBadFiles="false" AllowBootableCD="false" AllowMultipleFilePaths="true"/>
  </Format>
```

```

<Source>
  <EditList EditListPath="//SWRASKINREST\Rimage\CD-
R_Images\QD_DHASSLER_ENG-DHASSLER_0143.EDL"/>
</Source>
  <Output CDXA="false" Size="80" Type="RimageHeader" Postgap="false"
ImageFile="//SWRASKINREST\Rimage\CD-R_Images\QD_DHASSLER_ENG-
DHASSLER_0143.img" PowerSpan="true"/>
  <Rules CheckNames="true" AllowDirExt="false" CheckLevels="false"/>
  <VolumeName VolName="My Disc"/>
  <PVDInfo GMTOffset="-24" PVDSystem="" PVDAbstract="" PVDPreparer="UNTITLED"
PVDCopyright="" PVPublisher="" PVDVolumeSet="" PVDApplication=""
PVDBibliography="" PVDEffectiveDate="00/00/00" PVDEExpirationDate="00/00/00"/>
  <Controls Overwrite="true" WaitForSpace="-1"/>
</ImageOrder>

```

## Image Order Status

```

<?xml version="1.0"?>
<!DOCTYPE ImageOrderStatus SYSTEM
"\SWRASKINREST\Rimage\XML\ImageOrderStatus_1.6.dtd">
<ImageOrderStatus OrderId="QD_DHASSLER_ENG-DHASSLER_0143_I001"
ClientId="QD_DHASSLER_ENG-DHASSLER" ServerId="SWRASKINREST_IS01"
Originator="QD_DHASSLER_ENG-DHASSLER" MessagingHost="SWRASKINREST"
MessagingPort="4664" OriginalOrder="">
  <Status State="IN_PROCESS" SpanVolume="1" CurrentStatus="Writing File 1 of
8" SpanVolumeName="//SWRASKINREST\Rimage\CD-R_Images\QD_DHASSLER_ENG-
DHASSLER_0143001.img" CurrentOperation="Volume 1 of 2" PercentCompleted="5"
SpanTotalVolumes="2" SpanVolumePercent="5">
    <VolumeNameListEntry VolumeName="//SWRASKINREST\Rimage\CD-
R_Images\QD_DHASSLER_ENG-DHASSLER_0143001.img"/>
    <VolumeNameListEntry VolumeName="//SWRASKINREST\Rimage\CD-
R_Images\QD_DHASSLER_ENG-DHASSLER_0143002.img"/>
  </Status>
  <Timestamps OrderRead="2006-10-19 16:49:56"/>
</ImageOrderStatus>

```

## Order Set

```

<?xml version="1.0"?>
<!DOCTYPE OrderSet SYSTEM "\SWRASKINREST\Rimage\XML\OrderSet_1.1.DTD">
<OrderSet ClientId="QD_DHASSLER_ENG-DHASSLER" Priority="Normal"
OrderSetId="QD_DHASSLER_ENG-DHASSLER_0143" Originator="QD_DHASSLER_ENG-
DHASSLER" TargetServer="SWRASKINREST_PS01"
TargetCluster="DefaultProductionCluster">
  <OrderReference OrderId="QD_DHASSLER_ENG-DHASSLER_0143_P002"/>
  <OrderReference OrderId="QD_DHASSLER_ENG-DHASSLER_0143_P003"/>
  <ProductionOrderSet Copies="1" MediaSize="120mm" MediaType="CDR"
TargetLine="1" OrdersHaveLabels="false" TargetOutputMailslot="0"/>
</OrderSet>

```

## Order Set Status

```

<!DOCTYPE OrderSetStatus SYSTEM
"\SWRASKINREST\Rimage\XML\OrderSetStatus_1.4.dtd">
<OrderSetStatus ClientId="QD_DHASSLER_ENG-DHASSLER"
ServerId="SWRASKINREST_PS01" OrderSetId="QD_DHASSLER_ENG-DHASSLER_0143"
Originator="QD_DHASSLER_ENG-DHASSLER" MessagingHost="SWRASKINREST"
MessagingPort="4664" OriginalOrderSet="">

```

```
<Status Stage="BUSY" State="IN_PROCESS" PercentCompleted="0"/>
<Timestamps OrderRead="2006-10-19 16:50:40"/>
<OrderReference OrderId="QD_DHASSELER_ENG-DHASSLER_0143_P002"/>
<OrderReference OrderId="QD_DHASSELER_ENG-DHASSLER_0143_P003"/>
<ProductionOrderSetStatus CopiesCompleted="0" CopiesRequested="2"/>
</OrderSetStatus>
```

## Production Orders

```
<?xml version="1.0"?>
<!DOCTYPE ProductionOrder SYSTEM
"\\SWRASKINREST\Rimage\XML\ProductionOrder_1.14.DTD">
<ProductionOrder Copies="1" LogonId="dhasseler" OrderId="QD_DHASSELER_ENG-
DHASSLER_0143_P002" ClientId="QD_DHASSELER_ENG-DHASSLER" Priority="Normal"
ImagerHost="SWRASKINREST" Originator="QD_DHASSELER_ENG-DHASSLER"
ReferencedSet="QD_DHASSELER_ENG-DHASSLER_0143" ExternalImager="true"
SimulatePrinting="false" SimulateRecording="true">
  <Media Size="120mm" Type="CDR"/>
  <Target Line="1" Server="SWRASKINREST_PS01"
Cluster="DefaultProductionCluster"/>
  <InOut OutputMailslot="0"/>
  <Action>
    <Record>
      <WriteTrack Filename="\\SWRASKINREST\Rimage\CD-
R_Images\QD_DHASSELER_ENG-DHASSLER_0143001.img">
        <Data Type="Model" MergeSessions="false"/>
      </WriteTrack>
    </Record>
  </Action>
  <Action>
    <Record>
      <Fixate Type="SAO" Final="true"/>
    </Record>
  </Action>
</ProductionOrder>
```

```
<?xml version="1.0"?>
<!DOCTYPE ProductionOrder SYSTEM
"\\SWRASKINREST\Rimage\XML\ProductionOrder_1.14.DTD">
<ProductionOrder Copies="1" LogonId="dhasseler" OrderId="QD_DHASSELER_ENG-
DHASSLER_0143_P003" ClientId="QD_DHASSELER_ENG-DHASSLER" Priority="Normal"
ImagerHost="SWRASKINREST" Originator="QD_DHASSELER_ENG-DHASSLER"
ReferencedSet="QD_DHASSELER_ENG-DHASSLER_0143" ExternalImager="true"
SimulatePrinting="false" SimulateRecording="true">
  <Media Size="120mm" Type="CDR"/>
  <Target Line="1" Server="SWRASKINREST_PS01"
Cluster="DefaultProductionCluster"/>
  <InOut OutputMailslot="0"/>
  <Action>
    <Record>
      <WriteTrack Filename="\\SWRASKINREST\Rimage\CD-
R_Images\QD_DHASSELER_ENG-DHASSLER_0143002.img">
        <Data Type="Model" MergeSessions="false"/>
      </WriteTrack>
    </Record>
  </Action>
```

```

        </WriteTrack>
    </Record>
</Action>
<Action>
    <Record>
        <Fixate Type="SAO" Final="true"/>
    </Record>
</Action>
</ProductionOrder>

```

## Production Order Statuses

```

<?xml version="1.0"?>
<!DOCTYPE ProductionOrderStatus SYSTEM
"\SWRASKINREST\Rimage\XML\ProductionOrderStatus_1.10.dtd">
<ProductionOrderStatus OrderId="QD_DHASSELER_ENG-DHASSLER_0143_P002"
ClientId="QD_DHASSELER_ENG-DHASSLER" ServerId="SWRASKINREST_PS01"
Originator="QD_DHASSELER_ENG-DHASSLER" MessagingHost="SWRASKINREST"
MessagingPort="4664" OriginalOrder="" SimulatePrint="true"
SimulateWrite="true">
    <Status Stage="RECORDING" State="IN_PROCESS" CopiesCompleted="0"
CopiesRequested="1" PercentCompleted="0"/>
    <Timestamps OrderRead="2006-10-19 16:50:44"/>
    <Device Identifier="Recorder 1, Cache 1" IsStreaming="false" PercentDone="0"
CurrentState="RECORDING"/>
</ProductionOrderStatus>

```

## Server Configuration Samples

### Production Server Configuration

```

<!DOCTYPE ProductionServerConfiguration SYSTEM
"C:\Rimage\XML\ProductionServerConfiguration_1.9.dtd">
<ProductionServerConfiguration>
    <ServerInfo ID="RIMAGE-4WEHMIR2_PS01" Cluster="DefaultProductionCluster"
Hostname="RIMAGE-4WEHMIR2" IsService="true" OSVersion="Windows XP Embedded"
SystemFolder="C:\Rimage" IsPasswordSet="false" SoftwareVersion="7.3.23.0"/>
    <Transporter Type="Rimage DLN5200" Offline="false" InquiryString="Autoloader
1, COM1: DESKTOP 2 6.022E SN-U049536 ">
        <TransporterCapabilities MediaSize="120mm" MediaType="CDR"
PerfectPrint="false"/>
        <Bin Level="86" Usage="Input"/>
        <Bin Level="0" Usage="Output_Reject"/>
        <Mailslot Level="20" Usage="Output" NumberOfSlots="5"/>
        <Recorder Offline="false" DiscCount="735" InquiryString="Recorder 1,
Drive E: PLEXTOR CD-R PREMIUM 1.02 SN-0186554">
            <RecorderCapabilities CanRecordCD-R="true" CanDestroyCD-R="true"
CanRecordDVD-R="false" CanDestroyDVD-R="false" CanRecordPocketCD-R="true"
MaxCDRecordingSpeed="52" CanDestroyPocketCD-R="true"/>
            <Cache InquiryString="d:\Cache0"/>
            <Cache InquiryString="c:\Cache0"/>
        </Recorder>
        <Recorder Offline="false" DiscCount="726" InquiryString="Recorder 2,
Drive F: PLEXTOR CD-R PREMIUM 1.02 SN-0186553">

```

```
<RecorderCapabilities CanRecordCD-R="true" CanDestroyCD-R="true"
CanRecordDVD-R="false" CanDestroyDVD-R="false" CanRecordPocketCD-R="true"
MaxCDRecordingSpeed="52" CanDestroyPocketCD-R="true"/>
<Cache InquiryString="d:\Cache1"/>
<Cache InquiryString="c:\Cache1"/>
</Recorder>
<Printer Type="Everest-II" Ribbon="Color" Offline="false" DiscCount="0"
InquiryString="Printer - USBPRINT-0, COM1: EVEREST V1.06 SN- E012002"/>
</Transporter>
<Transporter Type="Manual" Offline="false" InquiryString="Loader 2, MANUAL
LOADER">
  <TransporterCapabilities MediaSize="ANY" MediaType="BOTH"
PerfectPrint="false"/>
</Transporter>
</ProductionServerConfiguration>
```

## Imaging Server Configuration

```
<!DOCTYPE ImageServerConfiguration SYSTEM
"C:\Rimage\XML\ImageServerConfiguration_1.4.dtd">
<ImageServerConfiguration>
  <ServerInfo ID="RIMAGE-4WEHMIR2_IS01" Cluster="DefaultImageCluster"
Hostname="RIMAGE-4WEHMIR2" IsService="true" OSVersion="Windows XP Professional"
Description="Rimage Imaging Server" SupportsSCP="true"
SystemFolder="C:\Rimage\" IsPasswordSet="false" SoftwareVersion="7.3.25.0"/>
  <Options Overwrite="true"/>
</ImageServerConfiguration>
```

## Server Dialog Samples

### Alert Dialog

```
<!DOCTYPE AlertDialog SYSTEM "C:\Rimage\XML\AlertDialog_1.6.dtd">
<AlertDialog ID="190448" Title="Alert" Message="Transporter Clear?"
ServerId="RIMAGE-4WEHMIR2_PS01">
  <Type>
    <OneButton Text="OK"/>
  </Type>
</AlertDialog>
```

### Error Dialog

```
<!DOCTYPE ErrorDialog SYSTEM "\\software10\Rimage\XML>ErrorDialog_1.4.dtd">
<ErrorDialog ID="530004" Title="TRANSPORTER ERROR!" Device="Autoloader 1, COM2"
Message="Error gripping disc.
Center disc in open drawer manually to retry Transporter Sense Code = 8 (medium
not present)" ServerId="software10_PS01" ErrorCode="140">
  <Buttons>
    <Top Text="Retry"/>
    <Bottom Text="Disable Transporter"/>
  </Buttons>
</ErrorDialog>
```



## Server Request / Reply Samples

### GetServerStatus Request

```
<?xml version="1.0"?>
<!DOCTYPE ProductionServerRequest SYSTEM
c:\Rimage\XML\ProductionServerRequest_1.11.dtd">
<ProductionServerRequest ServerId="software10_PS01" ClientId="software10_RSM" >
  <GetServerStatus GetAutoloaderStatus="true" />
</ProductionServerRequest>
```

### GetServerStatus Reply

```
<?xml version="1.0"?>
<!DOCTYPE ProductionServerReply SYSTEM
c:\Rimage\XML\ProductionServerReply_1.11.dtd">
<ProductionServerReply ClientId="software10_RSM" ServerId="software10_PS01"
Automation="Running" CommandState="COMPLETED" ReplyTimestamp="2004-05-06
08:25:01" CommandErrorCode="0">
  <ServerStatus>
    <ServerInfo Cluster="DefaultProductionCluster" Hostname="software10"
IsService="false" Description="Dave's Production &Server1"
PasswordSet="false" SystemFolder="\\software10\Rimage" MessagingPort="4664"
SoftwareVersion="6.4.26.0"/>
    <ProductionCount CopiesProduced="0" CopiesRejected="2"/>
    <AutoloaderStatus Offline="true" LoaderNumber="1"/>
    <AutoloaderStatus Offline="false" LoaderNumber="2">
      <Recorder Number="1" Offline="false"/>
    </AutoloaderStatus>
  </ServerStatus>
</ProductionServerReply>
```

### SetParameter Request

```
<?xml version="1.0"?>
<!DOCTYPE ProductionServerRequest SYSTEM
c:\Rimage\XML\ProductionServerRequest_1.11.dtd">
<ProductionServerRequest ServerId="software10_PS01" ClientId="software10_RSM" >
  <SetParameter >
    <Setting>
      <Recording>
        <MaxRecordingSpeed Value="Max" />
      </Recording>
    </Setting>
    <Setting>
      <Printing>
        <SimulatePrinting Value="true" />
      </Printing>
    </Setting>
  </SetParameter>
</ProductionServerRequest>
```

## SetParameter Reply

```
<?xml version="1.0"?>
<!DOCTYPE ProductionServerReply SYSTEM
c:\Rimage\XML\ProductionServerReply_1.11.dtd">
<ProductionServerReply ClientId="software10_RSM" ServerId="software10_PS01"
Automation="Running" CommandState="COMPLETED" ReplyTimestamp="2004-05-06
08:36:02" CommandErrorCode="0">
  <AckOnly/>
</ProductionServerReply>
```

## Appendix C – Server Status and Control Password Encryption

### Encryption Method

The encryption method is designed to work only with Unicode. Existing servers and clients that use MBCS need to convert to and from Unicode for this algorithm to work. Windows library calls are available for converting between MBCS and Unicode. Clients already using Unicode will find the algorithm simple to implement. The following steps include the MBCS to/from Unicode operations.

The following steps are performed to encrypt a password for transmission to a server.

1. Get the Unicode password. If this is in MBCS in a Windows client or server it must first be translated to Unicode.
2. Encrypt this using the Rimage encryption algorithm that treats the array as a stream of bytes. Sample code follows later in this document.
3. Encode the resulting string of bytes using the well-known Base64 standard. This translates the stream of bytes into a stream of ASCII characters. Every 6 bits are changed to one of the characters in the set: [A-Za-z0-9+/>. This results in 4 characters from every 3 bytes and is also known as 3-4 encoding. These characters have the important property that they are represented in all versions of ISO 646, including US-ASCII. The advantage of this is that this stream of characters is guaranteed to pass through any kind of transmission system with no damage. It can be freely translated between Unicode, UTF, etc and still come out correct.

When a server receives the password, the following steps are taken.

1. Decode the Base64 to get the byte stream back.
2. Decrypt this byte stream using the Rimage encryption algorithm. Sample code for the algorithm follows later in this document. The byte stream is now the original Unicode password. On any server or client requiring MBCS this string can be translated using Windows library calls.

• **Note:**

- Byte swapping may be necessary in some cases depending on the language and hardware used.
- This method allows password support in Unicode so that all languages will work.

### Rimage Core Encryption Algorithm

The new core encryption algorithm is similar to the old one, with the exception that no character wrapping is performed. In other words, each byte is simple incremented or decremented as necessary and overflow is not considered.

This is the pseudo-code for encryption. Start with a byte array (really the Unicode bytes) and a code number from 1 to 20. Note that the code number is restricted in the range 1 to 20.

```
For first byte to last byte in array
  currentbyte = currentbyte + codenum
  codenum = codenum + 1
  if (codenum > 20)
    codenum = 1
```

Here is the pseudo-code for decryption. As before, start with a byte array and a code number from 1 to 20.

```
For first byte to last byte in array
  currentbyte = currentbyte - codenum
  codenum = codenum + 1
```

```
if (codenum > 20)
    codenum = 1
```

## Password Encoding Samples Using C++

This code implements the encode, decode, and core encryption algorithms in C++ for Windows. This code also includes the MBCS translations, which are not necessary for a Unicode service or client. The Base64 routines are part of the Microsoft VC++ library, but could easily be coded in almost any programming language.

The encryption/decryption algorithm is implemented using recursion instead of a loop. If passwords were thousands of characters long this might be better coded using a loop.

### Encoding and Decoding a MBCS String

```
// Encode an MBCS string
// first change it to Unicode
CString Encode(LPCSTR txt, int code)
{
    CString cs;
    wchar_t *wbuf;
    int size = strlen(txt);
    // get the size we need
    size = MultiByteToWideChar(CP_ACP, MB_PRECOMPOSED, txt, size, NULL, 0);
    // allocate a buffer
    wbuf = new wchar_t[size];
    // change to Unicode
    size = ::MultiByteToWideChar(CP_ACP, MB_PRECOMPOSED, txt, size, wbuf, size);
    // set to byte count
    size *= sizeof wchar_t;
    // encrypt here
    Crypt((BYTE*)wbuf, code, true, size);
    // figure out how big we need a buffer for base64
    int b64size = Base64EncodeGetRequiredLength(size);
    // translate
    Base64Encode((BYTE*)wbuf, size, cs.GetBuffer(b64size), &b64size);
    cs.ReleaseBuffer(b64size);
    // free the buffer
    delete []wbuf;
    return cs;
}

// decode a base 64 string into MBCS
// change input to bytes
// decrypt it
// change it to MBCS
CString Decode(LPCSTR b64txt, int code)
{
    CString cs;
    BYTE *buf;
    // figure out the buffer size we need
    int size = Base64DecodeGetRequiredLength(strlen(b64txt));
    // get a buffer
```

```

    buf = new BYTE[size];
    // translate
    Base64Decode(b64txt, strlen(b64txt), buf, &size);
    // decrypt here
    Crypt(buf, code, false, size);
    // get the MBCS length
    int msize = ::WideCharToMultiByte(CP_ACP, WC_COMPOSITECHECK, (LPCWSTR)buf,
size/(sizeof wchar_t), NULL, 0, NULL, NULL);
    // translate the string
    ::WideCharToMultiByte(CP_ACP, WC_COMPOSITECHECK, (LPCWSTR)buf, size/(sizeof
wchar_t), cs.GetBuffer(msize), msize, NULL, NULL);
    cs.ReleaseBuffer(msize);
    // free the buffer
    delete []buf;
    return cs;
}
/*
    encrypt or decrypt a byte array
    similar to old Rimage encryption, except the values are not limited
*/
void Crypt(BYTE* pwd, int dnum, bool encode, int size)
{
    // see if done or no encoding needed
    if (dnum==0 || size==0)
        return;
    // modify the value
    *pwd += encode ? dnum:-dnum;
    // handle the next character
    Crypt(pwd+1, (dnum%20)+1, encode, size-1);
}

```

### Encoding and Decoding a Unicode String

```

void Crypt(BYTE* pwd, int dnum, bool encode, int size)
{
    // see if done or no encoding needed
    if (dnum==0 || size==0)
        return;
    // modify the value
    *pwd += encode ? dnum:-dnum;
    // handle the next character
    Crypt(pwd+1, (dnum%20)+1, encode, size-1);
}

///// _UNICODE should be defined
// Encode an Unicode string
// _UNICODE should be defined
// Encode an Unicode string

CString Encode (const wchar_t *csTxt , int code)

```

```
{
    CString cs;
    wchar_t *wbuf;
    int size = wcslen(csTxt);

    // allocate a buffer
    wbuf = new wchar_t[size+1];
    // Copy the Unicode characters
    wcscpy (wbuf, csTxt);
    // set to byte count
    size *= sizeof wchar_t;
    // encrypt here
    Crypt((BYTE*)wbuf, code, true, size);
    // figure out how big we need a buffer for base64
    int b64size = Base64EncodeGetRequiredLength(size);
    // translate
    char *destBuf;
    destBuf = new char[b64size];
    Base64Encode((BYTE*)wbuf, size, destBuf, &b64size);
    // Convert to CString
    _TCHAR *tsEncoded = cs.GetBuffer(b64size + 1);
    for (int i=0; i < b64size; i++)
        tsEncoded[i] = destBuf[i];
    tsEncoded[i] = 0;

    cs.ReleaseBuffer(b64size+1);
    // free the buffers
    delete []wbuf;
    delete []destBuf;
    return cs;
}

// _UNICODE should be defined
// decode a UNICODE base64 string
// change input to 8-bit characters
// decrypt input to bytes
// change bytes to UNICODE string
CString Decode(const wchar_t *csB64txt, int code)
{
    CString cs;
    BYTE *buf;
    int lenIn = wcslen(csB64txt);
    // figure out the buffer size we need
    int size = Base64DecodeGetRequiredLength(lenIn);
    // get a buffer
    buf = new BYTE[size];

    // Convert string to 8 bit characters
    char *sourceBuf;
    sourceBuf = new char[lenIn + 1];
```

```

    for (int i=0; i < size; i++)
        sourceBuf[i] = static_cast<char>(csB64txt[i]);
    sourceBuf[i] = 0;
    // translate
    Base64Decode(sourceBuf, lenIn, buf, &size);
    // decrypt here
    Crypt(buf, code, false, size);
    wchar_t *tsDecoded = (wchar_t *)buf;
    int usize = size/sizeof(wchar_t);
    _TCHAR *tsBuf = cs.GetBuffer (usize + 1);

    //Copy the decoded bytes into the CString
    for (i=0; i < usize; i++)
        tsBuf[i] = tsDecoded[i];
    tsBuf[i] = 0;

    cs.ReleaseBuffer(usize + 1);

    // free the buffers
    delete []buf;
    delete []sourceBuf;
    return cs;
}

```

## Appendix D – Error Codes

For a complete list of current error codes, visit the FAQs at [www.rimage.com/developers.html](http://www.rimage.com/developers.html).