CISCO CONFIDENTIAL

CHAPTER 18

Using the Job and Resource Manager

The Job and Resource Manager (JRM) provides a general-purpose interface that allows applications to:

- Schedule jobs—Jobs are general-purpose and application-defined.
- Lock resources by name—Resource locking is done by name and is advisory; that is, JRM is intended to be a *repository* of the currently locked devices; it does not lock a device.

Note JRM locking is meant to aid *cooperating applications* so they can prevent simultaneously updating the same device.

The following topics describe JRM and how to use it in your applications:

- Understanding JRM Services
- Understanding the JRM Architecture
- Enabling JRM
- Using JRM from a Java Application
- Using JRM from a Web Browser
- Customizing the Job Browser Button Behaviors
- Using JRM from the Command Line
- JRM Command Reference

For more information about the Job and Resource Manager, see the Job and Resource Management Functional Specification (BG 1.0/Rigel), ENG 21104.

This document reflects the information found in Revision L of the JRM functional specification, Job and Resource Management Functional Specification (BG 1.0/Rigel), ENG 21104. For the most recent updates to the JRM, refer to the current release of this specification.

CISCO CONFIDENTIAL

Understanding JRM Services

Applications that use Job and Resource Management services can schedule an activity, or job, to occur based on several conditions, including:

- Launch readiness— Only one instance of a periodic job can be running at any given time. You need to check whether the job been approved and enabled. Also whether there are any dependent jobs still running. All dependent jobs must finish successfully or new jobs cannot start.

  Using JRM, you can schedule a job to run pending approval. For example, device image update operations are often scheduled by network administrators, but must be approved by a manager.

  In this case, the network administrator can schedule an update, but it will be run by the time it is scheduled to run only if it has been approved by a manager. Anyone on the list of authorized approvers can review the jobs that require approval and either approve or reject them.

- Scheduling options—You can schedule jobs to run once or periodically.

  Applications often provide users with the means to schedule a task for a given time. SWIM, for example, lets the user specify when to update a device image. In this case, the application runs on a server, downloads the image to the specified device, and reboots the device. Normally software update tasks are run when the traffic on the device is minimal—for example, 2:00 a.m. Sunday morning. Using JRM services, SWIM can let users schedule a job to run at a specific time.

  Another application might need to periodically obtain and analyze device configuration information. Using JRM services, the application can schedule a job to run once a day, once an hour, only on Friday at 3:00 p.m., and so on. JRM also provides the functionality to browse the list of scheduled jobs.

- Tracking job instances—JRM helps you track each instances of a recurring job separately. Each instance of the execution of the job will have a unique entry in the job browser. Results of all instances are retained and tracked through entries in the job browser. The instances and the results can be individually purged. The purge policy applies to the instances rather than the entire job.

- Resource locks—A resource lock secures a device or device subnode, making it inaccessible for a period of time while a job is performed. Resource locks provide a way to serialize access to a device.

Note JRM is intended to be a *repository* of the currently locked devices; it does not lock a device. JRM locking is meant to aid *cooperating applications* so they can prevent simultaneously updating the same device.

- Event notification—Job Management uses the Event Distribution System to post job state and resource state events to other applications. Events can include when a job is started or when it ends; when a job has been canceled, approved or rejected; when a resource has been locked or unlocked; and so on.

JRM also provides job and resource lock attributes that allow applications to create their own customized functionality.

The following topics describe how JRM schedules a job and locks resources:

- Managing JRM Services
- Scheduling Jobs
- Locking Resources
- Locking Resources from Another Application
- Locking Parts of a Device

CISCO CONFIDENTIAL

Managing JRM Services

JRM services use the following logic to schedule and run a job:

1. A job is scheduled (for example, upgrade device image or change device configuration).
2. The job is created and scheduled to run, optionally after approval.
3. At the scheduled time, JRM:
   a. Determines if the conditions for this job have been met (see the "Understanding JRM Services" section on page 18-2)
   b. Creates a task to run the job
   c. Locks resources as it needs to work with them. Locking a resource prevents other jobs that also use JRM's locking functionality from simultaneously updating the same device. When a job is done with a device, it unlocks it explicitly.

      Automatic lock release and time-based locking prevent a rogue job from locking the device indefinitely.
   d. Optionally, reports its progress and sets its completion status.

Scheduling Jobs

A job can be scheduled to run if it is enabled and approved (or does not require approval), and its start time is in the future. Jobs can be scheduled to run once or periodically.

When the scheduled time arrives, the Job Manager checks for the following conditions before running the job:

- The job has been approved and enabled.
- The job is not running. Only one instance of a periodic job can be running at any given time.
- Even if a periodic job does not begin because the start conditions were not satisfied, the job for the next start time will be scheduled anyway.

Table 18-1 summarizes the job scheduling options.

Table 18-1 Job Scheduling Options

Schedule Type	Frequency	Description
Run-once	Start time	Time job is to start.
Periodic	Calendar-based	Specifies the day the job is to be run next. The units can be: • Days: Run job every *n* days. • Weeks: Run job on the given day of the week every *n* weeks. • Month: Run job on the given day of the month every *n* months. • Month-end: Run job on the last day of the month every *n* months. • Month-weekday: Run job on the given day of the first/second/third/fourth/last week every *n* months.
	Time-based	• Start time: Start the next job at a fixed time after the start of the previous invocation. • End time: Start the next job at a fixed time after the finish of the previous invocation.

CISCO CONFIDENTIAL

Locking Resources

Resource locks provide a way to ensure exclusive access to a device. A resource lock secures a device or device subnode, making it inaccessible *to other cooperating applications* using JRM for a period of time while a job is performed.

Note JRM serves as a *repository* of the currently locked devices—an application can ask whether a device it is about to update is being used by another application. *JRM does not lock a device*, which means an application can use a device by just ignoring the fact that it failed to lock the device first. Resource locking is meant to be a means to aid *cooperating applications* so that they can prevent a situation where two applications are simultaneously updating the same device.

When JRM receives a request to lock a resource, it checks the name of the resource against existing locks and performs one of these actions:

- If a resource can be locked, it is added to the locks list.
- If a resource is leased (that is, locked for a certain duration), when the lease expires the resource is unlocked.
- If a resource cannot be locked, the return code indicates this state.

Resources are locked for a certain period of time. When a job cannot estimate how long it will need a resource, it can either:

- Periodically renew the lock.

  or
- Lock the resource, specifying infinite time.

  Locking a resource for infinite time is *not recommended*. A lock can be "stuck," but that only happens when
  - A job that locked it is does not end (when a job ends, JRM automatically releases all its locks).
  - A resource was locked by an application that is not a job. When a resource is stuck, the only remedy is for you to force the lock using the JRM browser.

The resource is unlocked when:

- The job explicitly asks JRM to unlock the resource.
- The lock expires.
- The job that locked the resource has ended.

Locking Resources from Another Application

Although resources are typically locked by jobs run by the JRM server, they can be locked by any application. An application that wants to lock a resource must establish a connection with the JRM CORBA object and request a resource lock by providing the resource path and its ID string.

There are two differences between Job Manager and application resource locks:

- IDs used by the jobs are string representations of the job ID numbers. An ID supplied by an application should start with alphabetic character to avoid ID conflicts.

CISCO CONFIDENTIAL

- JRM automatically unlocks all locks owned by a job on job termination. For applications not locked by JRM, the Job Manager cannot sense that the application ended. Therefore, all the resources locked by an application without specifying a lock time (locked forever) must be explicitly unlocked by that application.

Locking Parts of a Device

Applications that use JRM might need to serialize access to certain parts of the device without necessarily locking the whole device. The device associated with the particular resource must also be easily identified.

JRM's resource naming scheme allows resources to form a hierarchy. The top-level nodes of the hierarchy are fully qualified device names (for example, nm7501.cisco.com) and the subnodes correspond to the parts of the device (for example, card0). Each lock is identified by its resource path, starting from the top level (nm7501.cisco.com/card0).

Locking a particular node prevents other applications from locking any nodes below it and all the nodes on the path to it. For example, if there is a lock for nm7501.cisco.com/card0:

- nm7501.cisco.com/card1 can be locked.
- nm7501.cisco.com cannot be locked.
- nm7501.cisco.com/card0/port0 cannot be locked.

Understanding the JRM Architecture

The following topics describe the JRM architecture:

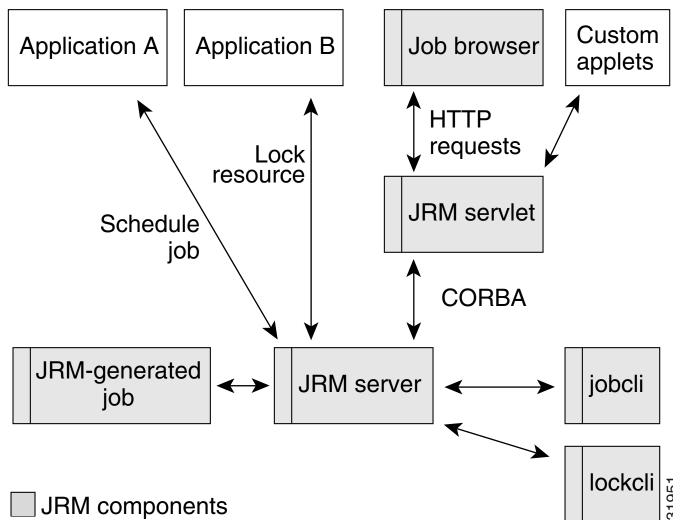
- An Overview of the JRM Architecture
- Understanding the JRM Server
- Understanding the Job Browser
- How JRM Relates to Other CWCS Components

An Overview of the JRM Architecture

The following figure shows the relationship between JRM, its components, and its clients.

CISCO CONFIDENTIAL

Figure 18-1 JRM Architecture



JRM consists of the following components:

- The JRM server provides job and locking services to the following clients:
  - Applications that schedule jobs
  - Jobs that JRM creates in response to a schedule request
  - Applications that lock and unlock resources
  - The JRM servlet
  - The command reference interfaces, jobcli and lockcli, that expose JRM to non-Java applications such as Perl and C++

  To learn more about the JRM server, see the "Understanding the JRM Server" section on page 18-7.
- The JRM servlet—Provides a URL interface to the JRM server process. Since there is no easy way to use CORBA calls directly from a web browser, this intermediary piece of code runs on the server and communicates with the JRM server via CORBA to execute the commands it receives from the web browser. All responses from the JRM servlet are XML-encoded.

  The JRM servlet:
  - Accepts HTTP requests from the Job Browser and other customized applets and translates them into CORBA calls to the JRM server.
  - Accepts JRM server responses, translates them into HTTP responses, and sends them back to the Job Browser or applet.

  For information about using the JRM servlet, see the "Using JRM from a Web Browser" section on page 18-21.
- The Job Browser—A configurable applet that displays the current jobs and locked resources and allows the users to stop, terminate, and remove jobs. The Job Browser applet runs on the web browser and communicates (exchanges XML documents) with the JRM server via the JrmServlet. The Job Browser can be embedded into HTML pages to provide a GUI for browsing and managing JRM jobs and resources.

  To learn more about the Job Browser, see the"Understanding the Job Browser" section on page 18-10.

CISCO CONFIDENTIAL

- Applets and Java components—Used to design custom HTML pages. Since there is a wide variation in GUI requirements for creating and editing different job types, individual applications must provide screens for creating, editing, and displaying details of their job types. To assist in these efforts, JRM provides applets to prompt for the date or time, display the JRM job panel, and display various types of JRM schedules.

  For information about designing custom HTML pages, see the "Customizing the Job Browser Button Behaviors" section on page 18-22.

- jobcli and lockcli applications—Provide a command-line interface for scheduling jobs and locking resources. These applications, which are primarily used for debugging purposes, also provide a JRM interface for non-Java applications such as Perl or C++.

  For information about using command line applications, see the "Using JRM from the Command Line" section on page 18-24.

Understanding the JRM Server

The JRM server provides job and locking services to various clients, including applications that schedule jobs or lock and unlock resources, jobs that JRM creates in response to a schedule request, the JRM servlet, and the command reference interfaces.

The following topics describe the JRM server components:

- About Jobs and Resources
- About JRM Server Classes
- About the IDL Interface
- About the Helper API
- About JRM Events

About Jobs and Resources

The two fundamental JRM entities are jobs and resource locks:

- Using JRM, you can schedule a job to run, pending approval. Job information consists of:
  - The command to be run
  - The schedule (run immediately, once, or periodically)
  - An approval flag (approval is required or not required before the job can run)
  - A status string that can be set by the job

  Job information is stored in a database table, where each row represents a single job. On startup, JRM reads the table and creates job objects. While running, JRM automatically updates the corresponding row to reflect every change in a job object. When a new job is created, a new row is added to the table. When a job is deleted, the corresponding row in the table is deleted.

- JRM locks resources as it needs to work with them. Resource locks provide a way to ensure exclusive access to a device. Resource lock information consists of:
  - The resource path
  - The owner (a job or any process)
  - The expiration time

CISCO CONFIDENTIAL

Note Resource information is not kept in a database. If JRM is stopped, all resource information is lost.

About JRM Server Classes

The JRM server application class performs these functions:

- Reads configuration files, such as the XML column and action configuration files for the Job Browser.
- Starts various threads. For example, when a job is about to run, it is added to the launch queue. The LaunchQueue thread dequeues the job and invokes the Daemon Manager to run the job.
- Keeps pointers to these JRM implementation classes:
  - JobManagerImpl—A facade to the Jobs class, a singleton containing everything related to jobs. It contains the following inner classes:

    ```
    Jobs.JobTable
    Jobs.LaunchQueue
    Jobs.Terminator
    ```

  - LockManagerImpl—A facade to the Locks class, a singleton containing everything related to locks. It contains the inner class, Locks.Lock.
  - AlarmQueue—Maintains the timer queue, runs the timer thread, and invokes a handler when the node timer expires.
  - DBConnection—Provides an interface to database-related functions.
  - DMConnection—Runs a thread that listens to Daemon Manager events and provides a listener interface.
  - EDSConnection—Runs a thread that sends events to EDS.
  - Client—A simplified JRM communication interface for jobs running under JRM (see the "About the Helper API" section on page 18-9).
- Provides utility functions.

About the IDL Interface

The JRM server implements two objects: JobManager and LockManager. Enumeration interfaces are implemented by the iterator objects, JobIter and LockIter. The JRM IDL (Interface Definition Language) file includes the interfaces of the objects supplied by the JRM server.

Related Topics

- For an example of the JRM IDL file, refer to the jrm.idl file in the CodeSamples directory on the CWCS SDK CD.
- For more information about CORBA, refer to:
  - Object Management Group website, http://www.omg.org
  - OMG's CORBA website, http://www.corba.org

CISCO CONFIDENTIAL

About the Helper API

The Client class provides a collection of static methods that might be helpful for clients that manipulate jobs and resources. The methods can be categorized by the main class and several inner classes:

- The top-level class, Client, implements Constants. Use this class to return printable representations of the schedule string or the job's run and schedule states. These methods can be used by any client.

  The Client class contains these methods, which can be used by any client:

  - Return a printable representation of the schedule string
  - Return a printable representation of the job's run and schedule states
  - Initialize the ORB and locate servers

- The inner class, MyJob, is a collection of static methods that can be used only by a job running under JRM control. This class provides methods to:

  - Set a job's completion state (success, success with info, failed)
  - Set a job's progress string
  - Lock resources
  - Unlock resources
  - Unlock all a job's resources
  - Get job information

Related Topics

See the "About the Helper API Methods" section on page 18-49.

About JRM Events

JRM can use Event Services Software (ESS) and the Event Distribution System (EDS) to publish events of interest to applications. JRM sends events when:

- A job starts.
- A job ends.
- A job start fails.
- A job is canceled.
- A job is rejected or approved.
- A resource is locked or unlocked.
- A process has ended.

These events belong to the event category status (EventCategory_Status). The event and resource atoms are listed in com.cisco.nm.cmf.jrm.JrmEdsAtomDev.java.

JRM publishes the following topics using ESS:

- cisco.mgmt.cw.cmf.jrm.EventJobReject
- cisco.mgmt.cw.cmf.jrm.EventJobEnd
- cisco.mgmt.cw.cmf.jrm.EventJobStart
- cisco.mgmt.cw.cmf.jrm.EventJobCancel
- cisco.mgmt.cw.cmf.jrm.EventLock

CISCO CONFIDENTIAL

- cisco.mgmt.cw.cmf.jrm.EventJobApprove
- cisco.mgmt.cw.cmf.jrm.EventDaemonEnd
- cisco.mgmt.cw.cmf.jrm.EventJobLaunchFail
- cisco.mgmt.cw.cmf.jrm.EventUnlock

JRM publishes the following topics using EDS:

- EventJobStart
- EventJobEnd
- EventJobCancel
- EventJobApprove
- EventJobReject
- EventDaemonEnd
- EventLock
- EventUnlock

JRelated Topics

See:

- Chapter 19, "Using Event Services Software."
- Chapter 20, "Using the Event Distribution System." Note that EDS is deprecated.

Understanding the Job Browser

The Job Browser is a configurable Java applet that you can embed in HTML pages to provide a GUI for browsing and managing JRM jobs and resources. The Job Browser uses XML files to specify:

- Job and resource table column names, sizes, and visibility.
- URLs to call that carry out the actions entered by the user.

Figure 18-2 shows the Job Browser interface, which provides the user actions shown in Table 18-2.

Figure 18-2 Sample Job Browser Dialog Box

Show only: All Jobs

Showing 4 records

	☐	Job ID	Type	Run Status	Sched Type	Description	Run Sched	Status
1.	☐	1004.1	TJob	Succeeded: 04/23/2004 11:33:16-04/23/2004 11:33:46	Periodic	Test job	Every 1 minutes(s), starting 04/23/2004 11:33:16	FARO: u r too good
2.	☐	1004.2	TJob	Running: 04/23/2004 11:34:16-	Periodic	Test job	Every 1 minutes(s), starting 04/23/2004 11:34:16	
3.	☐	1005	Faro		Scheduled	Once	At 04/23/2004 11:34:45	
4.	☐	1006	Faro		Periodic	Daily	At 11:36:03 daily, starting 04/23/2004	

↑--Select an item(s) then take an action --> Stop Delete Refresh

113542

CISCO CONFIDENTIAL

Table 18-2 Job Browser User Actions

Action	Description
Stop	Calls the corresponding JRMserver method. Select a Job ID, then click Stop.
Delete	Calls the corresponding JRMserver method. Select a Job ID, then click Delete.
Click a Job ID	Uses the URL registered in the JrmButtonActions.xml file for the selected job type to show job details for the application that owns the job.

Figure 18-3 Sample Job Resource Dialog Box

Showing 3 records

		Resource	Job Id/Owner	Time Locked	Expire Time
1.	☐	res1	faro	Fri Apr 23 11:36:13 IST 2004	Never
2.	☐	res2	faro	Fri Apr 23 11:36:13 IST 2004	Fri Apr 23 11:38:13 IST 2004
3.	☐	res3	faro	Fri Apr 23 11:36:13 IST 2004	Fri Apr 23 11:41:13 IST 2004

--Select an item(s) then take an action --> Free Resources Refresh

113543

Table 18-3 Job Browser User Actions

Button	Default Action
Free Resource...	• Explicitly frees resources without waiting for the associated job to end. • Frees orphaned resources that no longer have an associated job that is running. Shown only when user has administrative privileges. **Note** Using this button is *not* recommended unless resource is orphaned.

Related Topics

- Customizing the Job Browser Button Behaviors, page 18-22
- About the Helper API Methods, page 18-49

How JRM Relates to Other CWCS Components

JRM relies on:

- The built-in CWCS database to maintain job states. JRM lists a database as a dependency. Therefore, the Daemon Manager starts JRM only after the database is running. For more on the CWCS database, see Chapter 11, "Using the Database APIs."
- The built-in CWCS Daemon Manager to run and control jobs. JRM jobs run as processes under the CWCS Daemon Manager. For more on the CWCS Daemon Manager, see Chapter 17, "Using the Daemon Manager."

CISCO CONFIDENTIAL

Enabling JRM

JRM is part of CWCS System Services. Since CWCS release 3.0, JRM services are enabled by default. If your application requires services from JRM, remember to register for this service at installation. For instructions, refer to the "Registering for CWCS Services" section on page 5-4. If you prefer to request services after installation, refer to the "Enabling New Service Bundles from the Command Line" section on page 5-5.

Using JRM from a Java Application

To use JRM from a Java application, you must, for example, know how to establish a connection with the Job Manager, create a job, and set the status of the job. The following topics describe some typical job and lock management tasks:

- Establishing a Connection
- Creating a Job
- Setting the Job Status
- Getting Job Descriptions
- Handling an Unapproved Job
- Enabling a Disabled Job
- Handling a Crashed Job
- Locking and Unlocking a Device
- Handling an Unavailable Resource
- Accessing a Locked Device

For a description of the JRM APIs, see the "JRM Command Reference" section on page 18-26.

Establishing a Connection

Example 18-1 shows how to establish a connection with the Job Manager. The host where the JRM server is running is passed as a parameter.

Note This example disables automatic rebinding. If automatic rebinding is enabled and the JRM server aborts for any reason, the ORB will try to find another JRM server and reconnect to it. This is not a desirable action.

Example 18-1 Connecting with Job Manager

```
import java.lang.*;
import java.net.*;
import com.cisco.nm.cmf.jrm.*;
import com.cisco.nm.cmf.util.CmfException;
import com.cisco.nm.cmf.util.Util;
```

CISCO CONFIDENTIAL

```
import org.omg.CORBA.*;
import com.inprise.vbroker.CORBA.BindOptions;
import java.util.*;
public class testJpp {
public static void main (String[] args) {
        JrmServiceManager jrm=null;
        JobManager jm=null;
        String nmsroot;
        String host;

        try {
                Util.loadBGProperties("md.properties");
                nmsroot=System.getProperty("NMSROOT");
                System.out.println("NMSROOT is "+nmsroot);
        } catch(CmfException cmf) {
                System.out.println("unable to load md.properties");
        }

        try {

                host=(InetAddress.getLocalHost()).getHostName();
                System.out.println("host = " + host);

                Properties ORBProperties = Client.getOrbConnectionProperties();
                ORBProperties.put("org.omg.CORBA.ORBClass","com.inprise.vbroker.orb.ORB");
                org.omg.CORBA.ORB orb =
(com.inprise.vbroker.CORBA.ORB)com.cisco.nm.util.OrbUtils.initORB(null,ORBProperties);
                jrm = JrmServiceManagerHelper.bind(orb,Client.getJrmName(),host,null);

                System.out.println("Connected to JRM service Manager.");
                LoginInfo loginInfo = new LoginInfo("admin","admin","");
                jm = jrm.getJobHandle(loginfInfo);
        } catch (org.omg.CORBA.SystemException e){
                e.printStackTrace();
                System.err.println(e.toString());
                return;
        } catch (Exception e) {
                e.printStackTrace();
                System.err.println(e.toString());
                return ;
        }

        if (jm ==  null) {
                System.out.println("Job Manager not bound");
                return;
        }

        //Foll code to create a job

        long start=System.currentTimeMillis()+20000;
        int type=Constants.SCHTYPE_S_Minutes;
        int increment=3;
        Schedule sch=new Schedule(start,type,increment);

        int precedents[]={};

        JobInfo ji =new JobInfo(0,// id
                                "TestJob",// type
                                new String("Test job"),// description
                                "D:\\progra~1\\mkstoo~1\\mksnt\\sleep.exe 30",
                                sch,// schedule
                                precedents,// dependencies
                                Constants.RUNST_NeverRan,// state
                                Constants.SCHST_Enabled,// enabled
```

CISCO CONFIDENTIAL

```
                                System.currentTimeMillis(),// Time created
                                System.currentTimeMillis(),// Time modified
                                0,// Start
                                0,// Stop
                                "Scheduling Job",// Progress
                                host,// Host default=localhost)
                                new String("system"),// Account (default=system)
                                new String("Reference"),// Reference
                                "admin",// Owner
                                ""// Approver
                        );

        IntHolder jid=new IntHolder(0);

        if(ji == null) {
                System.out.println("Job info is null");
                System.exit(1);
        }

        int status=jm.job_create_hist(ji,jid);


        if(status != Constants.STATUS_Ok) {
                System.out.println("Failed to create Job");
                System.exit(-1);  // job creation failed
        } else {
                System.out.println("Job "+jid.value+" created sfly.");
        }
}
```

Creating a Job

Example 18-2 shows how to create a job with these attributes:

- It will run the Java application whose main class is myJavaClass and use a standard Java classpath (that is, the same one that was used to run the JRM server) to locate the classes.
- The job's ID will be passed as a command-line option. A job uses this ID to communicate its progress and completion status to JRM.
- The job's type is ACLM.
- The job requires approval before it can be run.
- The job will run in one minute.

Example 18-2 Creating a Job

```
// Create a job
JRM.Schedule sch =
    new Schedule(System.currentTimeMillis()+60*1000,// Start in a minute
        SCHTYPE_Once,
        SCHINC_Months,// Doesn't matter
        0);
        int precedents[] = {};
        JRM.JobInfo ti =
        new JobInfo(           0,// id
        "ACLM",// type
        "Description",// description
```

CISCO CONFIDENTIAL

```
        "$JP -cp $JC $JJ myJavaClass",// command:
        // run myJavaClass
        sch,// schedule
        precedents,// dependencies
        RUNST_NeverRan,// state
        SCHST_RequiresApproval // Approval state:
        | SCHST_AM_WAITING // requires approval,
        | SCHST_ENABLED,// enabled
                    0,// Time created
                    0,// Time modified
                    0,// Start
                    0,// Stop
                    "",// Progress
                    "",// Host default=localhost)
                    "",// Account (default=system)
                    "",// Reference
                    "",// Owner
                    ""// Approver
                    );

// Create holder for the returned value
IntHolder h_id = new IntHolder(0);

// Create a job, test its status
try {
    int stat = job_manager.job_create(ti,h_id);
    if (STATUS_Ok == stat) {
        System.out.println("Created job with id = ", h_id.value);
    }
    else {
        …
    }
    catch (org.omg.CORBA.SystemException e) {
    // Attempt to reconnect explicitly
    }
}
```

Setting the Job Status

The following code fragment tells JRM that the job has ended successfully and sets its progress string (which will become the completion string) to "Download successful":

```
import com.cisco.nm.cmf.jrm.Client;
…
    Client.MyJob.set_completion_state(Client.RUNST_Succeeded);
    Client.MyJob.set_progress("Download successful");
```

You need to keep the following in mind:

- Execute this code from a job executing under JRM.
- Add $JJ to the command line that starts this job (see the "About the Job and Resource Lock Attributes" section on page 18-26).

The displayed job status is a dynamic attribute of the job. JRM calculates the status based on the job's run state, scheduled state attributes, and the current time.

- For run-once jobs, the displayed job status reflects either:
  - The job's scheduling state (if the job's scheduled time is in the future)
  - The job's run result (if the job's scheduled time is in the past).

CISCO CONFIDENTIAL

- For periodic jobs, the displayed job status displays the result of the last run and the scheduling state of the next run.

The job status values for both run-once and periodic jobs are summarized in the tables in the "About Displayed Job Status Values" section on page 18-28.

Getting Job Descriptions

Example 18-3 shows how to get the job descriptions for all scheduled jobs.

Example 18-3 Getting Job Descriptions

```
jrm.JobIterHolder iter = new JobIterHolder();
jrm.JobInfoHolder job_info = new JobInfoHolder();

try
{
    //Get the job enumerator
    int status = job_manager.job_enum(iter);
    if (STATUS_Ok == status)
    {

        while (STATUS_EOF != iter.value.next(job_info))
        {
            System.out.println(job_info.value.szDescription);
        }
        iter.value.release();
    }
}
catch (org.omg.CORBA.SystemException e)
{
//      .....
}
```

Handling an Unapproved Job

Use the code fragment in Example 18-4 when a job that requires approval is scheduled and has not been approved by the scheduled time. The job execution is abandoned, and the job deleted if it is not periodic.

Example 18-4 Handling Unapproved Jobs

```
jrm.JobInfoHolder job_info = new JobInfoHolder();
// Find out the job details corresponding to the job.
int status = job_manager.job_get_info(idJob,job_info);

// If the job is still waiting for approval, then execution of the
// job is abandoned.
if ((STATUS_Ok == status)&&(SCHST_AM_Waiting == (SCHST_AM_Masks &
job_info.value.sch_state)))
{
    System.out.println ("Still waiting for approval, so can't start now");
    jrm.ScheduleHolder schedule = new ScheduleHolder();
```

CISCO CONFIDENTIAL

```
    // Get the job schedule
    int stat = job_manager.job_get_schedule(idJob,schedule);

    // Check the schedule type.If the job is not periodic, delete the job
    if (STATUS_Ok == stat)
    {
    if ((SCHTYPE_Immediate == schedule.value.type)||(SCHTYPE_Once == schedule.value.type))
        {
        System.out.println ("Now deleting the job") ;
        job_manager.job_delete(idJob);
        }
    }
}
```

Enabling a Disabled Job

Example 18-5 shows how to create a job in the disabled state, do some operations, and then enable the job.

Example 18-5 Enabling a Disabled Job

```
// Create a job
jrm.Schedule sch = new Schedule(0,
    SCHTYPE_Immediate,
    SCHTYPE_Monthly //Ignored for SCHTYPE_Immediate
    );

int precedents[] = {};

// Create the JobInfo structure with appropriate values
jrm.JobInfo job_info = new JobInfo(0,// id
    "ACLM",// type
    "Description", // description
    "$JP -cp $JC $JJ myJavaClass",// command:
    // run myJavaClass
    sch,// schedule
    precedents, // dependencies
    RUNST_NeverRan,// state
    SCHST_AM_Approved,// Approval state:
    0, // Time created
    0, // Time modified
    0, // Start
    0, // Stop
    "", // Progress
    "", // Host default=localhost)
    "", // Account (default=system)
    "", // Reference
    "", // Owner
    "" // Approver
    );

// IntHolder for holding the JobId
org.omg.CORBA.IntHolder h_id = new org.omg.CORBA.IntHolder(0);
// Create a job, test its status
try
{
    int stat = job_manager.job_create(job_info,h_id);
```

CISCO CONFIDENTIAL

```
    if (STATUS_Ok == stat)
    {
        System.out.println("Job created with id = "+ h_id.value);
    }
    else
    {
        System.out.println("Job creation failed ");
        System.exit(0);
    }
    //Perform some operations involving the newly created job

    //  ...........

    // Now enable the job
    int status = job_manager.job_set_resume(h_id.value,true);
    if (STATUS_Ok != status)
    {
        System.out.println("Job resumption failed");
    }
}
catch (org.omg.CORBA.SystemException e)
{
    System.out.println("Exception while job creation ");
    System.exit(0);
}
```

Handling a Crashed Job

Example 18-6 shows how to get a job's current running state and delete a crashed job.

Example 18-6 Handling a Crashed Job

```
org.omg.CORBA.IntHolder result = new org.omg.CORBA.IntHolder();

int status = job_manager.job_get_result(idJob, result);
if (STATUS_Ok == status)
{
    if (result.value == RUNST_Crashed)
    {
        // Delete the job
        status = job_manager.job_delete(idJob);
        if (STATUS_Ok != status)
        {
        System.out.println("No such job exists");
        }
    }
}
else
{
    System.out.println(" Getting the run state failed!");
}
```

CISCO CONFIDENTIAL

Locking and Unlocking a Device

In Example 18-7, a job locks a device, does some processing, and releases the lock.

Example 18-7 Locking and Unlocking a Device

```
LockManagerImpl lock_manager = new LockManagerImpl("TEST");

int status = lock_manager.lock("device1", "my_app", 1000);

/* If no job has locked device1 yet, then status = STATUS_Ok */
if (STATUS_Ok == status)
{
    System.out.println("No lock exists now for the device ");
//... do some processing....
    lock_manager.unlock("device1","my_app");
}
```

Handling an Unavailable Resource

In Example 18-8, a job is enabled and approved and then, at the scheduled time, it tries to lock a resource and fails.

Example 18-8 Handling an Unavailable Resource

```
int status = 0;
jrm.JobInfoHolder job_info = new JobInfoHolder();

try
{
     // Find out the job details corresponding to the job.
     status = job_manager.job_get_info(idJob,job_info);
}
catch (org.omg.CORBA.SystemException e)
{
    System.exit(0);
}

//If the job is approved and is enabled then try to run the job
if ((STATUS_Ok == status)&& (SCHST_AM_Approved == (SCHST_AM_Masks &
job_info.value.sch_state))&&
     (SCHST_Enabled == (job_info.value.sch_state & SCHST_Enabled)))
{
    try
    {
        //Lock the required devices
           status = lock_manager.lock("device name","owner",1000);

        // If locking failed
        if (STATUS_Ok != status)
        {
            jrm.ScheduleHolder schedule = new ScheduleHolder();
            // Get the job schedule
            int stat = job_manager.job_get_schedule(idJob,schedule);
```

CISCO CONFIDENTIAL

```
                if (STATUS_Ok == stat)
                {
                // If the job is not periodic, then delete the job
                    if ((SCHTYPE_Once == schedule.value.type) ||
                    (SCHTYPE_Immediate == schedule.value.type))
                    {
                    System.out.println ("Now deleting the job");
                    job_manager.job_delete(idJob);
                    }
                }
            }
            // Run the job
            else
            {
                status = job_manager.job_run(idJob);
                if (STATUS_Ok != status)
                {
                System.out.println ("job run failed");
                }
            }
        }
        catch(org.omg.CORBA.SystemException e)
        {
        System.exit(0);
        }
}
```

Accessing a Locked Device

In Example 18-9, a job is trying to lock a device that is already locked by another job. The code finds the information about the other job and, if that job is not running, releases all resources locked by it. Then it tries to lock the device. After the device is locked, the job does some processing and then releases the lock.

Example 18-9 Accessing a Locked Device

```
/* Current job is trying to lock a device device1 */
int status = lock_manager.lock("device1", "my_app", 2000);

/* If some job has already locked device1, then status = STATUS_Exists */
if (STATUS_Exists == status)
{
    /* Find out the complete Lock_info for device1 */
    jrm.LockInfoHolder lock_info = new LockInfoHolder();
    status = lock_manager.get_lock("device1",lock_info);
    /* If Lock_info found for device1, status = STATUS_Ok  */
    if (status==STATUS_Ok)
    {
        Integer int_id = new Integer(lock_info.value.szJob);
        int job_id = int_id.intValue();
        jrm.JobInfoHolder job_info = new JobInfoHolder();
        // Find out the job details corresponding to the job id obtained.
        status = job_manager.job_get_info(job_id,job_info);
        if (status==STATUS_Ok)
        {
            if (job_info.value.run_state != RUNST_Running)
            {
```

CISCO CONFIDENTIAL

```
            // Release all resources locked by the job
            lock_manager.unlock_job(lock_info.value.szJob);
            status = lock_manager.lock("device1","my_app", 1000);
            //    ... do some processing....
            lock_manager.unlock("device1","my_app");
            }
        }
        else
        System.out.println("No job exists");
    }
```

Using JRM from a Web Browser

The JRM servlet provides the URL interface to the JRM server process. The servlet communicates with the JRM server via CORBA to execute the commands it receives. All responses from the JRM servlet are XML-encoded.

Table 18-4 summarizes the URL commands which the JRM servlet supports via HTTP POST and GET requests.

Table 18-4 JRM Servlet URL Commands

URL Command	Description
getJobAndResourceList	Returns an XML-encoded list of currently scheduled jobs and locked resources. Example: `http://server:1741/CSCOnm/servlet/com.cisco.nm.cmf.jrm.JrmServlet?cmd=getJobAndResourceList`

stop	Requests that the specified job be stopped. Returns "true" if successful, "false: *error message*" otherwise, where *error message* provides the message to display to the user.
	For job history jobs, "instance id" and a boolean variable "stop instance" is used. The "stop instance" should be true if the user selects "Stop this instance only" and false if the user selects "Stop all instances".
	Example:
	For jobs that maintain job history:
	`http://server:1741/CSCOnm/servlet/com.cisco.nm.cmf.jrm.JrmServlet?button=stop&jobid=1001&instanceid=2&stopinstance=true`
	`For jobs that do not maintain job history:`
	`http://server:1741/CSCOnm/servlet/com.cisco.nm.cmf.jrm.JrmServlet?button=stop&jobid=1001&instanceid=2&stopinstance=true`

CISCO CONFIDENTIAL

Table 18-4 JRM Servlet URL Commands (continued)

URL Command	Description

kill	Requests that the specified job be killed. Returns "true" if successful, "false: *error message*" otherwise where *error message* provides the message to display to the user.
	For job history jobs, "instance id" and a boolean variable "stop instance" is used. The "stop instance" should be true if the user selects "Stop this instance only" and false if the user selects "Stop all instances".
	Example:
	jobs that maintain job history:
	`http://server:1741/CSCOnm/servlet/com.cisco.nm.cmf.jrm.JrmServlet?button=kill&jobid=1001&instanceid=2&stopinstance=true`
	`jobs that do not maintain job history:`
	`http://server:1741/CSCOnm/servlet/com.cisco.nm.cmf.jrm.JrmServlet?button=kill&jobid=1001&instanceid=2&stopinstance=true`

remove	Removes specified job from JRM scheduler. Returns "true" if successful, "false: *error message*" otherwise where *error message* provides the message to display to the user.
	Instance id is also passed with the jobid.
	Example:
	For jobs that maintain job history:
	`http://server:1741/CSCOnm/servlet/com.cisco.nm.cmf.jrm.JrmServlet?button=remove&jobid=1001&instanceid=2`
	`For jobs that do not maintain job history:`
	`http://server:1741/CSCOnm/servlet/com.cisco.nm.cmf.jrm.JrmServlet?button=remove&jobid=1001&instanceid=2`

Customizing the Job Browser Button Behaviors

To customize the behavior of the buttons in the Job Browser dialog box, modify the action configuration file summarized in Table 18-5. The action configuration file contains the tags listed in Table 18-6.

Table 18-5 Job Browser Action Configuration File

Name	JrmButtonActions.xml
Description	When the user selects a job and clicks one of the buttons in the Job Browser dialog box, JrmJobApplet uses this file to determine the URL to be called. All action URLs are invoked via an HTTP GET request.
Runtime Location	*$NMSROOT*/htdocs/jrm/JrmButtonActions.xml where *$NMSROOT* is the directory in which the product was installed.
Guidelines/ Restrictions	Applications that do not want the default JRM actions must add the action URLs for their job type to this file.

CISCO CONFIDENTIAL

Table 18-6 Job Browser Action Configuration File Contents

Tag	Attributes	Description
ACTIONS	Container for all button actions.	
JOBTYPE	Container for a job type.	
	ID	A string identifying the job type and subtypes (for example, SWIM:update.)
ACTION	Defines the URL that is called when the user requests an action.	
	BUTTON	Allowed values: • details—Invoked when the user clicks Job Details. The details button URL is displayed in a separate browser instance. • remove—Invoked when the user clicks Remove Job. • stop or kill—When the user clicks Stop Job, the Job Browser presents two options: – Stop the job (finish gracefully). – Kill the job unconditionally.
	URL	The URL to be called to perform an action. • details—If there is no URL in the actions file for the selected job type, an error dialog box is displayed. • remove—Default action is to call the JRM servlet to remove the job. • stop or kill—Default action for both stop and kill is to ask the Daemon Manager to kill the job. **Note** A stop action can be specified for a particular job type without specifying a kill action and vice versa.

Return values for the BUTTON and URL tags shown in Table 18-6 is as follows:

- details: Returns an application-specific HTML page that displays the job details. The application must display any error messages.
- remove, stop, kill: Returns “true” if the operation *initiated* successfully (does not mean it completed); “false:Error Message” if an error occurred. A dialog box displays the error message.

Example 18-10 shows the default Job Browser action configuration file.

Example 18-10 Default Job Browser Action Configuration File

```
<?xml version=”1.0"?>

<ACTIONS>
  <JOBTYPE ID="Test">
    <ACTION BUTTON="details" URL="/jrm/TestDetails.html" />
    <ACTION BUTTON="stop" URL="/CSCOnm/servlet/com.cisco.nm.cmf.jrm.JrmServlet" />
    <ACTION BUTTON="kill" URL="/CSCOnm/servlet/com.cisco.nm.cmf.jrm.JrmServlet" />
    <ACTION BUTTON="remove" URL="/CSCOnm/servlet/com.cisco.nm.cmf.jrm.JrmServlet" />
  </JOBTYPE>
```

CISCO CONFIDENTIAL

```
  <JOBTYPE ID="NetConfigJob">
    <ACTION BUTTON="stop"
URL="/CSCOnm/servlet/com.cisco.nm.config.netconfig.server.NetConfigUIServlet" />
    <ACTION BUTTON="kill"
URL="/CSCOnm/servlet/com.cisco.nm.config.netconfig.server.NetConfigUIServlet" />
    <ACTION BUTTON="remove"
URL="/CSCOnm/servlet/com.cisco.nm.config.netconfig.server.NetConfigUIServlet" />
    <ACTION BUTTON="details" URL="/netconfig/netconfig.jsp" />
  </JOBTYPE>

  <JOBTYPE ID="NetConfigPurge">
    <ACTION BUTTON="remove"
URL="/CSCOnm/servlet/com.cisco.nm.config.netconfig.server.NetConfigUIServlet" />
    <ACTION BUTTON="kill"
URL="/CSCOnm/servlet/com.cisco.nm.config.netconfig.server.NetConfigUIServlet" />
    <ACTION BUTTON="stop"
URL="/CSCOnm/servlet/com.cisco.nm.config.netconfig.server.NetConfigUIServlet" />
  </JOBTYPE>
</ACTIONS>
```

To customize the Job Details and Stop Job buttons but rely on the default JRM action for the Remove Job button for the ACL Manager, add the following element to the action configuration file:

```
<JOBTYPE ID="acl">
    <ACTION VERB="details" URL="/acl/editjob"/>
    <ACTION Verbosity" URL="/acl/stopjob"/>
</JOBTYPE>
```

Using this action configuration, if the user selects an ACL job with Id=42 and clicks “Job Details”, the JRM browser will issue the following GET request and display the result in a new browser window:

`http://`*server*`:1741/acl/editjob?jobid=42&button=details`

If the user clicks “Stop Job” and selects “stop” (not “kill”) from the dialog box, the JRM browser will issue the following GET request:

`http://`*server*`:1741/acl/stopjob?jobid=42&button=stop`

If the response is “true,” a dialog box is displayed indicating that the operation was initiated successfully. If the response is “false: device not responding” (for example), a dialog box will be displayed with the text, “device not responding.”

Using JRM from the Command Line

JRM includes two command line applications, jobcli and lockcli, that provide a command language interface for scheduling jobs and locking resources. These applications are used for debugging purposes and to provide a JRM interface for non-Java applications such as Perl or C++.

The following topics describe jobcli and lockcli:

- Job Command Line Interface
- Lock Command Line Interface

Related Topics

See the “Using the Job Command-Line Commands” section on page 18-59

CISCO CONFIDENTIAL

Job Command Line Interface

The job command line application, jobcli (shown in Table 18-7), is a Java application that provides a simple job manipulation command language.

Table 18-7 Jobcli Interface

Name	com.cisco.nm.cmf.jrm.jobcli	
Description	Provides a simple command language that allows you to: • Create or create and run a job • Approve or reject a job • Cancel, delay, delete, suspend, or resume a job • Change job schedule You can provide inputs to jobcli using either: • Standard input • A file of commands The jobcli commands are described in the "Using the Job Command-Line Commands" section on page 18-59.	
Syntax	`jre -cp classpath com.cisco.nm.cmf.jrm.jobcli [ -f clifile ]`	
Arguments	**Name**	**Description**
	classpath	Environment variable that tells the interpreter where to look for user-defined classes.
	-f *clifile*	Reads commands from *clifile*. If –f option is missing, commands are read from standard input.
Outputs	Sent to stdout/stderr.	

Lock Command Line Interface

The lock command line application, lockcli (shown in Table 18-8), is a Java application that provides a simple lock manipulation command language.

.

Table 18-8 Lockcli Interface

Name	com.cisco.nm.cmf.jrm.lockcli
Description	Provides a command language that allows you to lock and unlock a single or several resources for the owner running outside JRM. **Note** There is no automatic unlocking. If you lock a resource without specifying the lock duration, be sure you unlock it.
Syntax	`jre -cp classpath com.cisco.nm.cmf.jrm.lockcli {-l | -u} owner resource[@duration]…`

CISCO CONFIDENTIAL

Table 18-8 Lockcli Interface

Arguments	Name	Description
	-l or -u	Lock or unlock resource(s).
	owner	Resource owner. Should contain at least one alphabetic character to distinguish it from jobs run under Job Browser.
	resource	Specifies resource path. If followed by @*duration*: • If *duration* is greater than zero, *resource* will be locked for *duration* seconds. • If *duration* is less than or equal to zero, an error message will be displayed. • If no *duration* is specified, resource will be locked until it is explicitly unlocked by its owner. *Duration* is ignored when unlocking.
Output	Success = 0 Error = >0 and stderr contains a diagnostic message.	

Examples	The following example locks switch1.cisco.com for 30 seconds and switch2.cisco.com until explicitly unlocked on behalf of swim1 job:
	`jre -cp … com.cisco.nm.cmf.jrm.lockcli -l swim1 switch1.cisco.com@30 switch2.cisco.com`
	The following example unlocks switch2.cisco.com:
	`jre -cp … com.cisco.nm.cmf.jrm.lockcli -u swim1 switch2.cisco.com`	

JRM Command Reference

JRM provides interfaces from Java, IDL, and servlets, and via command line utilities. These topics describe the reference information for these interfaces:

- About the Job and Resource Lock Attributes
- About Displayed Job Status Values
- About the Job Manager Methods
- About the Lock Manager Methods
- About the Helper API Methods
- About the JRM Java Constants
- Using the Job Command-Line Commands

About the Job and Resource Lock Attributes

Table 18-9 describes the available job attributes.

CISCO CONFIDENTIAL

Table 18-9 Job Attributes

Attribute	Description
ID	A unique number assigned to this job at creation time. This number is never reused.
InstanceID	A unique number that is incremented for every instance of job history jobs. For jobs without multiple instances, value is 0.
Type	String identifying the job type and job subtypes (for example, SWIM:update.)
Description	String that describes the job.
Command line	The command line to start the job. JRM performs the following parameter substitutions on the command line: **String/Result** $JC — Java classpath. Prefix with –cp. $JI — Job ID. $JJ — Sets nm.jrm.jobid Java property to the job ID; equivalent to -Dnm.jrm.jobid=$JI. $II — Instance ID. $IJ — Sets nm.jrm.instanceid Java property to the instance ID; equivalent to -Dnm.jrm.instanceid=$II. $JP — Path to Java interpreter. $JR — RME installation root directory. $: — Path separator, the value of the path.separator system property (':' on UNIX, ';' on Windows). $/ — File separator, the value of the file.separator system property ('/' on UNIX, '\' on Windows).
Host	Machine name or IP address where the job will run. (For future extensions. Currently, the job is always started on the local machine.)
Account	Account under which the job is run. (For future extensions.)
Schedule	How often this job will run. Options include: run immediately, run once, run on a calendar basis (periodic), run on a time-start basis, or run on a time-stop basis.
Dependencies	A list of the Job IDs that must complete successfully. (Not currently implemented.)
Completion state	Describes the current state or last run result of the job. Job states include: running, never, suspended, wait for approval, scheduled (pending), rescheduled, completed succeeded, failed, crashed, canceled, rejected, or ERROR.
Schedule state	Determines if the job can be scheduled to run based on whether it is enabled, requires approval, or has already been approved.
Start and stop times from last run	Time stamps from the last time the job was run or attempted to run.
Progress status	Updates or diagnostic information.
Reference	An application-specific string. May contain the URL of job results.
Owner	Account of the person that created the job.
Creation time	Time the job was created.
Last modification time	Time the job was last modified.
Approver	Account of the approver. Valid only if approval is required.

CISCO CONFIDENTIAL

Table 18-10 summarizes the available resource lock attributes.

Table 18-10 Resource Lock Attributes

Field	Description
Resource path	String defining the device name and any subnode.
Owner	Job ID represented as a string.
Time stamp	Time the lock was established.
Expiration time	Time the lock expires.

About Displayed Job Status Values

Displayed job status value vary according to how often the job is run and whether approval is required.

Table 18-11 summarizes the displayed job status for run-once, approval-required jobs.

Table 18-11 Run-Once Approval-Required Job Status Values

Schedule vs. Current Time	Run State	Enabled	Approval State	Displayed Schedule Status	Displayed Completion Status
Future	Never	N		Suspended	
Future	Never	Y	Wait for approval	Wait for approval	
Future	Never	Y	Approved	Scheduled (pending)	
Future	Never	Y	Rejected	Rejected	
Future	Canceled			Canceled	
Past	Never	N			Suspended
Past	Never	Y	Wait for approval — rejected		Rejected
Past	Never	Y	Approved		ERROR
Past	Canceled				Canceled
Past	All others				Same as run state

Table 18-12 summarizes the displayed job status for run-once, no-approval-required jobs.

Table 18-12 Run-Once No-Approval Job Status Values

Schedule vs. Current Time	Run State	Enabled	Displayed Schedule Status	Displayed Completion Status
Future	Never	N	Suspended	
Future	Never	Y	Scheduled (pending)	
Future	Canceled		Canceled	

CISCO CONFIDENTIAL

Table 18-12 Run-Once No-Approval Job Status Values (continued)

Past	Never	N		Suspended
Past	Never	Y		ERROR
Past	Canceled			Canceled
Past	All others			Same as run state

Table 18-13 summarizes the displayed job status for periodic, approval-required jobs.

Table 18-13 Periodic Approval-Required Job Status Values

Run State	Enabled	Approval State	Displayed Schedule Status	Displayed Completion Status
*	N		Suspended	Same as Run state
Canceled	Y		Canceled	Canceled
All others	Y	Wait for approval	Wait for approval	Same as Run state
All others	Y	Approved	Scheduled (pending)	Same as Run state
All others	Y	Rejected	Rejected	Same as Run state

Table 18-14 summarizes the displayed job status for periodic, no-approval-required jobs.

Table 18-14 Periodic No-Approval Job Status Values

Run State	Enabled	Displayed Schedule Status	Displayed Completion Status
	N	Suspended	Same as Run State
Canceled	Y	Canceled	Canceled
All others	Y	Scheduled	Same as Run State

About the Job Manager Methods

Use the Job Manager methods summarized in Table 18-15 to add JRM scheduling functionality to your application. These methods return Java constants described in the "About the JRM Java Constants" section on page 18-56.

Table 18-15 Job Manager Method Summary

Returns	Syntax and Description
`int`	`job_cancel(int idJob);` Cancels a running job
`int`	`job_cancel_instance(int idJob, int instanceId, boolean cancelAllInstances);` Cancels a running job with instance ID.
`int`	`job_cancel_event(int idJob);` Cancels a running event

CISCO CONFIDENTIAL

Table 18-15 Job Manager Method Summary (continued)

Returns	Syntax and Description
int	**job_cancel_instance_event**(int idJob, int instanceId, boolean cancelAllInstances); Cancels a running job with instance ID, and specified whether to cancel the instance alone or the entire job
int	**job_create**(JobInfo job_info, org.omg.CORBA.IntHolder idJob); Creates a job
int	**job_create_hist**(JobInfo jiJob, org.omg.CORBA.IntHolder idJob); Creates a job with job history
int	**job_delete**(int idJob); Deletes a job
int	**job_delete_instance**(int idJob, int instanceId, boolean delFlag); Deletes a job with the given id, instance id.
int	**job_enum**(JobIterHolder job_iter); Creates a job enumerator
int	**job_enum_hist**(JobIterHist job_iter); Creates a job enumerator
int	**job_get_info**(int idJob, JobInfoHolder job_info); Gets information about a job
int	**job_get_info_hist**(int idJob, int instanceId, JobInfoHistHolder jiJobHist); Adds job information history about a job
int	**job_get_result**(int idJob, IntHolder status); Gets job run state
int	**job_get_schedule**(int idJob, ScheduleHolder schedule); Fills schedule with job schedule
int	**job_get_schedule_string**(int idJob, StringHolder schedule); Gets job schedule information
int	**job_run**(int idJob); Runs a job immediately
int	**job_set_approved**(int idJob, boolean bApproved, String szApprover) Approves or rejects a job
int	**job_set_info**(JobInfo job_info); Updates job information
int	**job_set_info_hist**(JobInfoHist jiJobHist); Updates job information
int	**job_set_progress_string**(int idJob, String szStatus); Sets progress string
int	**job_set_reference**(int idJob, String szReference); Sets job reference attribute
int	**job_set_result**(int idJob, int state); Sets job run state
int	**job_set_resume**(int idJob, boolean bResume); Enables or disables a job
int	**job_set_schedule**(int idJob, Schedule schedule); Sets job schedule
int	**next**(JobInfoHolder job_info); Fills job_info with job information

CISCO CONFIDENTIAL

Table 18-15 Job Manager Method Summary (continued)

Returns	Syntax and Description
int	**next_n**(int max_jobs, JobInfoSequenceHolder job_seq); Fills job_seq with job descriptions
int	**release**(); Releases an iterator

job_cancel

```
int job_cancel (int idJob);
```

Cancels a job if it is running. The job sends a request to stop.

Note JRM only issues the request. It does not wait until the process actually stops.

Input Arguments

idJob	[int] Unique number assigned to a job at creation time.

Return Values

STATUS_Ok	Job was canceled or is not running.
STATUS_NotFound	No such job.

Usage Guidelines

To cancel a running job, send a request to the Daemon Manager to stop the process.

job_cancel_instance

```
int job_cancel_instance (int idJob, int instanceId, boolean cancelAllInstances);
```

Cancels an instance job if it is running.

Note JRM only issues the request. It does not wait until the process actually stops.

Input Arguments

idJob	[int] Unique number assigned to a job at creation time.
idInstance	[int] Unique number assigned to an instance at creation time.
cancelAllInstances	[boolean] Indicates whether you want to cancel all future instances or just this instance

CISCO CONFIDENTIAL

Return Values

`STATUS_Ok`	Job instance was canceled or is not running.
`STATUS_NotFound`	No such instance.

Usage Guidelines

To cancel a running job, send a request to the Daemon Manager to stop the process.

job_cancel_event

```
int job_cancel_event (int idJob)
```

Cancels a job if it is running. Sends a cancel event to the running job. The job should process the event and stop the event by itself.

Note JRM only issues the request. It does not wait until the process actually stops.

Input Arguments

`idEvent`	[int] Unique number assigned to an event at creation time.

Return Values

`STATUS_Ok`	Event was canceled or is not running.
`STATUS_NotFound`	No such Event.

Usage Guidelines

To cancel a running job, send a request to the Daemon Manager to stop the process.

job_cancel_instance_event

```
int job_cancel (int idJob, int instanceId, boolean cancelAllInstances);
```

Cancels an instance of an event if it is running.

Note JRM only issues the request. It does not wait until the process actually stops.

CISCO CONFIDENTIAL

Input Arguments

`idJob`	[int] Unique number assigned to a job at creation time.
`instanceId`	[int] Unique number assigned to a job instance at creation time.
`cancelAllInstances`	[boolean] Indicates whether you want to cancel all future instances or just this instance

Return Values

`STATUS_Ok`	Instance of the job was canceled or is not running.
`STATUS_NotFound`	No such job instance.

Usage Guidelines

To cancel a running instance of the job, send a request to the Daemon Manager to stop the process.

job_create

```
int job_create (JobInfo job_info, org.omg.CORBA.IntHolder idJob);
```

Creates a job. The ID field and all fields related to the last job execution are ignored.

Input Arguments

`job_info`	[JobInfo] Job information. The JobInfo structure is defined in the IDL file (see the "About the IDL Interface" section on page 18-8).

Output Arguments

`id_Job`	[org.omg.CORBA.IntHolder] Unique number assigned to a job at creation time.

Return Values

`STATUS_Ok`	Success. On return, idJob contains the unique job ID.
`STATUS_NotFound`	Job not found.

job_create_hist

```
int job_create_hist (JobInfo jiJob, org.omg.CORBA.IntHolder idJob);
```

Creates a job. The ID field and all fields related to the last job execution are ignored.

CISCO CONFIDENTIAL

Input Arguments

`job_info`	[JobInfo] Job information. The JobInfo structure is defined in the IDL file (see the "About the IDL Interface" section on page 18-8).

Output Arguments

`id_Job`	[org.omg.CORBA.IntHolder] Unique number assigned to a job at creation time.

Return Values

`STATUS_Ok`	Success. On return, idJob contains the unique job ID.
`STATUS_NotFound`	Job not found.

job_delete

```
int job_delete (int idJob);
```

Deletes the job with the given ID.

Input Arguments

`idJob`	[int] Unique number assigned to a job at creation time.

Return Values

`STATUS_Ok`	Success.
`STATUS_NotFound`	No such job.

job_delete_instance

```
int job_delete_instance (int idJob, int instanceId, boolean delFlag);
```

Deletes the job with the given ID.

Input Arguments

`idJob`	[int] Unique number assigned to a job at creation time.
`instanceId`	[int] Unique number assigned to an instance of a job at creation time.
`delFlag`	[boolean] Indicates whether you want to delete all instances or just this instance.

CISCO CONFIDENTIAL

Return Values

`STATUS_Ok`	Success.
`STATUS_NotFound`	No such job.

job_enum

`int` **`job_enum`** `(JobIterHolder job_iter);`

Creates the job enumerator.

Output Arguments

`job_iter`	[JobIterHolder] Object used to retrieve the next job.

Return Values

`STATUS_Ok`	Success.

Example

Use this method with the next and *release* methods to retrieve the next job.

```
JobIterHolder jih = new JobIterHolder ();
JobInfoHolder jobinfo = new JobInfoHolder ();
/* Get the JobIter and browse through it */
if (STATUS_Ok == job_manager.job_enum(jih))
{
    while (STATUS_Ok == jih.value.next(jobinfo))
    {
    // do the required operations on JobInfo
    }
}
//Calling the release of JobIter
jih.value.release();
```

The functions next (), next_n() and release() are to be called on the JobIter reference, which can be obtained by calling the job_enum API.

job_enum_hist

`int` **`job_enum_hist`** `(JobIterHist job_iter);`

Creates the job enumerator.

Output Arguments

`job_iter`	[JobIterHolder] Object used to retrieve the next job.

CISCO CONFIDENTIAL

Return Values

`STATUS_Ok`	Success.

Example

Use this method with the next and *release* methods to retrieve the next job.

```
JobIterHolder jih = new JobIterHolder ();
JobInfoHolder jobinfo = new JobInfoHolder ();
/* Get the JobIter and browse through it */
if (STATUS_Ok == job_manager.job_enum(jih))
{
    while (STATUS_Ok == jih.value.next(jobinfo))
    {
    // do the required operations on JobInfo
    }
}
//Calling the release of JobIter
jih.value.release();
```

The functions next (), next_n() and release() are to be called on the JobIter reference, which can be obtained by calling the job_enum API.

job_get_info

```
int job_get_info (int idJob, JobInfoHolder job_info);
```

Fills the job information data structure with information about a given job.

Input Arguments

`idJob`	[int] Unique number assigned to a job at creation time.

Output Arguments

`job_info`	[JobInfoHolder] Job information. The JobInfo structure is defined in the IDL file (see the "About the IDL Interface" section on page 18-8).

Return Values

`STATUS_Ok`	Success.
`STATUS_NotFound`	No such job.

job_get_info_hist

```
int job_get_info (int idJob, int instanceId, JobInfoHistHolder jiJobHist);
```

Fills the job information data structure with information about a given job.

CISCO CONFIDENTIAL

Input Arguments

idJob	[int] Unique number assigned to a job at creation time.
instanceId	[int] Unique number assigned to an instance of a job at creation time.

Output Arguments

job_info	[JobInfoHolder] Job information. The JobInfo structure is defined in the IDL file (see the "About the IDL Interface" section on page 18-8).

Return Values

STATUS_Ok	Success.
STATUS_NotFound	No such job.

job_get_result

```
int job_get_result (int idJob, IntHolder status);
```

Retrieves the current run state of a job.

Input Arguments

idJob	[int] Unique number assigned to a job at creation time.

Output Arguments

status	[IntHolder] Current run state.

Return Values

STATUS_Ok	Success.
STATUS_NotFound	No such job.

job_get_schedule

```
int job_get_schedule (int idJob, ScheduleHolder schedule);
```

Fills the schedule data structure with the job's scheduling information.

CISCO CONFIDENTIAL

Input Arguments

idJob	[int] Unique number assigned to a job at creation time.

Output Arguments

schedule	[ScheduleHolder] Job scheduling information. The Schedule structure, which is defined in the IDL file (see the "About the IDL Interface" section on page 18-8), includes the next time to start, the type of schedule, and the time increment.

Return Values

STATUS_Ok	Success.
STATUS_NotFound	No such job.

job_get_schedule_string

```
int job_get_schedule_string (int idJob, StringHolder schedule);
```

Puts a displayable representation of the job schedule into the string contained in schedule.

Input Arguments

idJob	[int] Unique number assigned to a job at creation time.

Output Arguments

schedule	[StringHolder] Displayable representation of the job's schedule.

Return Values

STATUS_Ok	Success.
STATUS_NotFound	No such job.

job_run

```
int job_run (int idJob);
```

Runs the job immediately.

Input Arguments

idJob	[int] Unique number assigned to a job at creation time.

CISCO CONFIDENTIAL

Return Values

`STATUS_Ok`	Success.
`STATUS_NotFound`	No such job.

job_set_approved

```
int job_set_approved (int idJob,
        boolean bApproved,
        String szApprover)
```

Approves or rejects a job. This method approves or rejects a job and records the approver name.

Input Arguments

`idJob`	[int] Unique number assigned to a job at creation time.
`bApproved`	[boolean] True = approve job. False = reject job.
`szApprover`	[String] Account of the approver.

Return Values

`STATUS_Ok`	Success.
`STATUS_NotFound`	No such job.

Usage Guidelines

If a nonperiodic job that requires approval has not been approved by the time it is scheduled to run, it is automatically rejected.

job_set_info

```
int job_set_info (JobInfo job_info);
```

Replaces all job information.

Input Arguments

`job_info`	[JobInfo] Job information. The JobInfo structure is defined in the IDL file (see the "About the IDL Interface" section on page 18-8).

CISCO CONFIDENTIAL

Return Values

STATUS_Ok	Success.
STATUS_NotFound	No such job.

job_set_info_hist

```
int job_set_info_hist (JobInfoHist jiJobHist);
```

Replaces all job information.

Input Arguments

job_info	[JobInfo] Job information. The JobInfo structure is defined in the IDL file (see the "About the IDL Interface" section on page 18-8).

Return Values

STATUS_Ok	Success.
STATUS_NotFound	No such job.

job_set_progress_string

```
int job_set_progress_string (int idJob, String szStatus);
```

Sets the progress string with update or diagnostic information.

Input Arguments

idJob	[int] Unique number assigned to a job at creation time.
szStatus	[String] Updates or diagnostic information.

Return Values

STATUS_Ok	Success.
STATUS_NotFound	No such job.

job_set_reference

```
int job_set_reference (int idJob, String szReference);
```

Sets the job's reference attribute.

CISCO CONFIDENTIAL

Input Arguments

`idJob`	[int] Unique number assigned to a job at creation time.
`szReference`	[String] An application-specific string. May contain the URL of the job results.

Return Values

`STATUS_Ok`	Success.
`STATUS_NotFound`	No such job.

job_set_result

```
int job_set_result (int idJob, int state);
```

Sets the job's current run state. The only states the application is allowed to set are Succeeded, SucceededWithInfo, or Failed.

Input Arguments

`idJob`	[int] Unique number assigned to a job at creation time.
`state`	[int] Current run state.

Return Values

`STATUS_Ok`	Success.
`STATUS_NotFound`	No such job.
`STATUS_BadArgument`	State was not Succeeded, SucceededWithInfo, Never Ran, Canceled, CanceledInstance, or Failed

job_set_resume

```
int job_set_resume (int idJob, boolean bResume);
```

Resumes or suspends a job. When a previously suspended job is resumed, it is scheduled to run according to its schedule type (run once or periodic) provided that it is approved or does not require approval.

Input Arguments

`idJob`	[int] Unique number assigned to a job at creation time.
`bResume`	[boolean] True = resume job. False = suspend job.

CISCO CONFIDENTIAL

Return Values

`STATUS_Ok`	Success.
`STATUS_NotFound`	No such job.

Usage Guidelines

You can use the following technique when a job needs to be run immediately but only after certain actions are performed by the job creator:

- Create a job with schedule specifying to run it immediately but in the suspended state. You now have a job ID.
- Perform whatever actions are needed that reference job ID.
- Enable (resume) the job. If approved, the job will run immediately.

job_set_schedule

```
int job_set_schedule (int idJob, Schedule schedule);
```

Sets the job's schedule to **schedule**.

Input Arguments

`idJob`	[int] Unique number assigned to a job at creation time.
`schedule`	[Schedule] Job scheduling information. The Schedule structure, which is defined in the IDL file (see the "About the IDL Interface" section on page 18-8), includes the next time to start, the type of schedule, and the time increment.

Return Values

`STATUS_Ok`	Success.
`STATUS_NotFound`	No such job.

next

```
int next (JobInfoHolder job_info);
```

Returns the JobInfo instance for the next task entry.

Output Arguments

`job_info`	[JobInfoHolder] Job information. The JobInfo structure is defined in the IDL file (see the "About the IDL Interface" section on page 18-8).

CISCO CONFIDENTIAL

Return Values

STATUS_Ok	Filled *job_info*.
STATUS_EOF	No more entries.

Example

See the“job_enum” section on page 18-35.

next_n

```
int next_n (int max_jobs, JobInfoSequenceHolder job_seq);
```

Fills the job holder array with the next group of jobs.

Input Arguments

max_jobs	[int] Maximum number of jobs that can be returned in the job holder array.

Output Arguments

job_seq	[JobInfoSequenceHolder] An array of objects that allows you to retrieve the next *max_jobs* jobs.

Return Values

STATUS_Ok	Put at least one element into *job_seq*.
STATUS_EOF	No more jobs.

Example

See the “job_enum” section on page 18-35.

release

```
int release();
```

Releases the iterator and makes it unavailable to the clients.

Arguments

None

Example

See the “job_enum” section on page 18-35.

CISCO CONFIDENTIAL

About the Lock Manager Methods

Use the Lock Manager methods summarized in Table to add JRM resource locking functionality to your application. These methods return Java constants described in the "About the JRM Java Constants" section on page 18-56.

Table 18-16 JRM Lock Manager Method Summary

Returns	Syntax and Description
`int`	**`enum_job_locks`**`(String szJob, LockIterHolder lock_iter);` Creates an iterator
`int`	**`find_lock`**`(String szResource, LockInfoHolder lock_info);` Finds a lock entry
`int`	**`get_lock`**`(String szResource, LockInfoHolder lock_info);` Gets lock information
`int`	**`lock`**`(String szResource, String szOwner, int duration);` Locks a resource
`int`	**`lock_n`**`(LockRequest[] Locks, String szOwner);` Locks multiple resources
`int`	**`next`**`(LockInfoHolder lock_info);` Fills *lock_info*
`int`	**`next_n`**`(int max_locks, LockInfoSequenceHolder lock_seq);` Fills *lock_seq*
`int`	**`release`**`();` Releases an iterator
`int`	**`unlock`**`(String szResource, String szOwner);` Unlocks a resource
`int`	**`unlock_job`**`(String szJob);` Unlocks all locks for a job
`int`	**`unlock_n`**`(String[] szResource, String szOwner);` Unlocks multiple resources

enum_job_locks

```
Status enum_job_locks (String szJob, LockIterHolder lock_iter);
```

Creates an iterator for all the locks for this job or process.

Input Arguments

`szJob`	[String] For a job, the string representation of the job ID. For a process, the name known to the Daemon Manager.

Output Arguments

`lock_iter`	[LockIterHolder] Object used to retrieve the next lock. The LockIter structure is defined in the IDL file (see the "About the IDL Interface" section on page 18-8).

CISCO CONFIDENTIAL

Return Values

STATUS_Ok	Success. *lock_iter* is returned.
STATUS_NotFound	No such owner.

Example

Use this method with the *next* and *release* methods to retrieve the next lock.

```
LockIterHolder lih = new LockIterHolder ();
LockInfoHolder lockInfo = new LockInfoHolder ();
/* Get the LockIter and browse through it */
if (STATUS_Ok == lock_manager.enum_job_locks(lih))
{
    while (STATUS_Ok == lih.value.next(lockInfo))
    {
    // do the required operations on LockInfo
    }
}
//Calling the release of LockIter
lih.value.release();
```

find_lock

```
Status find_lock (String szResource, LockInfoHolder lock_info);
```

Finds the lock entry that *prevents* a device from being locked. Unlike get_lock, which returns the lock information for a specific device, find_lock returns the lock information for the device that is preventing another resource from being locked.

For more information about the locking hierarchy, see the "Locking Parts of a Device" section on page 18-5.

Input Arguments

szResource	[String] Device name and any subnode.

Output Arguments

lock_info	[LockInfoHolder] Lock information. The LockInfo structure is defined in the IDL file (see the "About the IDL Interface" section on page 18-8).

Return Values

STATUS_Ok	Success.
STATUS_NotFound	No such resource.

get_lock

```
Status get_lock(String szResource, LockInfoHolder lock_info);
```

CISCO CONFIDENTIAL

Returns lock information for a device. This method differs from find_lock, which finds the lock entry that is *preventing* a device from being locked.

Input Arguments

`szResource`	[String] Device name and any subnode.

Output Arguments

`lock_info`	[LockInfoHolder] Lock information. The LockInfo structure is defined in the IDL file (see the "About the IDL Interface" section on page 18-8).

Return Values

`STATUS_Ok`	Success.
`STATUS_NotFound`	No such resource.

lock

```
Status lock (String szResource, String szOwner, int duration);
```

Locks the resource for *duration* seconds. If the job already owns this resource, this method will change the lock expiration time.

Input Arguments

`szResource`	[String] Device name and any subnode.
`szOwner`	[String] For a job, the string representation of the resource owner. For a process, the name known to the Daemon Manager.
`duration`	[int] Time (seconds) for which the resource is to be locked.

Return Values

`STATUS_Ok`	Success (job was killed or is not running).
`STATUS_Exists`	The lock for that resource already exists.

lock_n

```
Status lock_n (LockRequest[] Locks, String szOwner);
```

Locks multiple resources.

CISCO CONFIDENTIAL

Input Arguments

`Locks`	[LockRequest] An array of objects that allows you specify the resources to be locked.
`szOwner`	[String] For a job, the string representation of the resource owner. For a process, the name known to the Daemon Manager.

Return Values

`STATUS_Ok`	Success. (*All* resources have been successfully locked.)
`STATUS_Exists`	At least one resource could not be locked.

next

```
Status next (LockInfoHolder lock_info);
```

Fills the lock holder with the lock entry information and advances to the next lock entry in the locks list.

Output Arguments

`lock_info`	[LockInfoHolder] Lock information. The LockInfo structure is defined in the IDL file (see the "About the IDL Interface" section on page 18-8).

Return Values

`STATUS_Ok`	Success (job was killed or is not running).
`STATUS_EOF`	End of iteration.

Example

See the "enum_job_locks" section on page 18-44.

next_n

```
Status next_n (int max_locks, LockInfoSequenceHolder lock_seq);
```

Fills the lock holder array with the next group of locks.

Input Arguments

`max_locks`	[int] Maximum number of locks that can be returned in LockInfoSequence.

CISCO CONFIDENTIAL

Output Arguments

`lock_seq`	[LockInfoSequenceHolder] An array of objects that allows you to retrieve the next *max_locks* locks. The LockInfo structure is defined in the IDL file (see the "About the IDL Interface" section on page 18-8).

Return Values

`STATUS_Ok`	Success (at least one element put into *lock_seq*).
`STATUS_EOF`	No more elements.

Example

See the "enum_job_locks" section on page 18-44.

release

```
Status release();
```

The **release** method releases an iterator and makes it unavailable to the clients.

Arguments

None

Example

See the "enum_job_locks" section on page 18-44.

unlock

```
Status unlock (String szResource, String szOwner);
```

Unlocks the specified resource.

Input Arguments

`szResource`	[String] Name of the resource to be unlocked.
`szOwner`	[String] For a job, the string representation of the resource owner. For a process, the name known to the Daemon Manager.

Return Values

`STATUS_Ok`	Success.
`STATUS_NotFound`	Resource was not locked.

CISCO CONFIDENTIAL

unlock_job

```
Status unlock_job (String szJob);
```

Release all locks for the specified job.

Input Arguments

`szJob`	[String] For a job, the string representation of the job ID. For a process, the name known to the Daemon Manager.

Return Values

`STATUS_Ok`	Success. Resources released successfully or there were no resources locked by this job.

unlock_n

```
Status unlock_n (String[] Resource, String szOwner);
```

Unlocks all the resources in the specified list.

Input Arguments

`Resource`	[String] Array of device names and any subnodes.
`szOwner`	[String] For a job, the string representation of the resource owner. For a process, the name known to the Daemon Manager.

Return Values

`STATUS_Ok`	Success. Released resources.
`STATUS_NotFound`	At least one resource was not locked.

About the Helper API Methods

The Helper API consists of the class Client and the inner class of Client, MyJob. Only jobs running under JRM can use the helper methods in the MyJob class. These methods return Java constants described in the "About the JRM Java Constants" section on page 18-56.

Related Topics

- About the Helper API
- About the JRM Java Constants
- Parsing ESS Messages

CISCO CONFIDENTIAL

Location	com.cisco.nm.cmf.jrm.Client

Client Class Constructor Summary

`public class **Client** implements Constants`

This class is a collection of helper functions that can be called by JRM clients. It contains two groups of functions:

- Those usable by any client.
- Those usable only by jobs running under JRM.

Client Class Method Summary

Returns	Syntax and Description
`static String`	**getScheduleString** (Schedule sch) Returns a printable representation of the schedule string. Defines SCHTYPE constants.
`static void`	**getStateStrings** (JobInfo ji, StringHolder h_szRunState, StringHolder h_szSchState) Returns a printable representation of a job's run and schedule states.
`Properties`	**getOrbConnectionProperties**() Initializes the ORB and locates servers.

MyJob Class Constructor Summary

`public static class **MyJob**`

This inner class is a collection of the static methods that can be used only from jobs running under JRM.

The methods in MyJob automatically establish connection with ORB. They obtain the value of Job Id (which they need to communicate to the JRM) from the nm.jrm.jobid property.

The easiest way to set this property is to add the $JJ parameter to the job's command line (see the "About the Job and Resource Lock Attributes" section on page 18-26).

Table 18-17 MyJob Class Method Summary

Returns	Syntax and Description
`int`	**get_job_id**(); Gets the job ID
`int`	**get_job_instance_id**(); Gets the job instance ID
`int`	**get_job_info**(JobInfoHolder h_ji); Fills h_ji with job information
`int`	**get_job_info_hist**(JobInfoHistHolder h_ji); Fills h_ji with job information with additional parameters for req_hist and instance_id

CISCO CONFIDENTIAL

Table 18-17 MyJob Class Method Summary (continued)

Returns	Syntax and Description
int	**get_lock_info**(string szLockPath, LockInfoHolder h_li); Fills h_li with lock information
boolean	**is_server_running**(); Checks server status
int	**lock**(string szLockPath, int duration); Locks the resource for the current job
int	**lock_n**(LockRequestSequence Locks); Locks multiple resources
int	**set_completion_state**(int run_state); Sets the running job's status
int	**set_progress**(string szProgress); Sets the running job's progress string
void	**unlock**(string szLockPath); Unlocks the resource
int	**unlock_all**(); Unlocks all resources for the current job

get_job_id

```
static int get_job_id();
```

MyJob method returns the job ID by retrieving the value of nm.jrm.jobid property. This property is set by adding $JJ on the job's command line.

Arguments

None

Return Values

0	Called outside the running job.
an integer	Job ID.

get_job_instance_id

```
static int get_job_instance_id();
```

MyJob method returns the job instance ID by retrieving the value of nm.jrm.jobinstanceid property. This property is set by adding $JJ on the job's command line.

Arguments

None

CISCO CONFIDENTIAL

Return Values

`0`	Called outside the running job.
`an integer`	Job Instance ID.

get_job_info

```
static int get_job_info (JobInfoHolder h_ji);
```

MyJob method sets h_ji.value to JobInfo of self.

Output Arguments

`h_ji`	[JobInfoHolder] Contains job information.

Return Values

`STATUS_Ok`	Success.
`STATUS_NotFound`	Not called from job.

get_job_info_hist

```
static int get_job_info_hist (JobInfoHistHolder h_ji);
```

MyJob method sets h_ji.value to JobInfoHist of self.

Output Arguments

`h_ji`	[JobInfoHistHolder] Contains job history information.

Return Values

`STATUS_Ok`	Success.
`STATUS_NotFound`	Not called from job.

get_lock_info

```
static int get_lock_info (string szLockPath, LockInfoHolder h_li);
```

MyJob method that returns lock information for this lock if the resource **szLockPath** is locked.

Input Arguments

`szLockPath`	[string] Device name and any subnode.

CISCO CONFIDENTIAL

Output Arguments

`h_li`	[LockInfoHolder] Contains lock information.

Return Values

`STATUS_Ok`	Success.
`STATUS_NotFound`	Lock not found.

getOrbConnectionProperties

```
static public Properties getOrbConnectionProperties()
```

Initializes the ORB and locates servers.

Arguments

None

Returns

`Properties`	[Properties] Server properties (host, port number)

getScheduleString

```
static String getScheduleString (Schedule sch)
```

Returns a printable representation of the schedule string. Defines SCHTYPE constants.

Input Arguments

`sch` — [Schedule] Schedule object, which includes:

- sch.start—Next time to start
- sch.increment—Increment amount

Returns

`szFormat`	[String] Printable schedule string

getStateStrings

```
static void getStateStrings (JobInfo ji, StringHolder h_szRunState, StringHolder
h_szSchState)
```

Returns a printable representation of a job's run and schedule states.\

CISCO CONFIDENTIAL

Input Arguments

`ji`	[JobInfo] JobInfo structure

Output Arguments

`h_szRunState`	[StringHolder] Run state
`h_szSchState`	[StringHolder] Schedule state

is_server_running

```
static boolean is_server_running();
```

MyJob method that checks to see if the server is running.

Arguments

None

Return Values

`True`	Server is running.
`False`	Server is not running.

lock

```
static int lock (string szLockPath, int duration);
```

MyJob method that locks the resource for the current job for **duration** seconds.

Input Arguments

`szLockPath`	[string] Device name and any subnode.
`duration`	[int] Number of seconds to lock the resource.

Return Values

`STATUS_Ok`	Success.
`STATUS_NotFound`	Not called from job.
`STATUS_Exists`	szLockPath cannot be locked.

CISCO CONFIDENTIAL

lock_n

```
static int lock_n (LockRequestSequence Locks);
```

MyJob methods that locks multiple resources.

Input Arguments

`Locks`	[LockRequestSequence] An array of objects that allows you to specify the resources to be locked.

Return Values

`STATUS_Ok`	Success.
`STATUS_NotFound`	Not called from job.
`STATUS_Exists`	At least one resource cannot be locked.

set_completion_state

```
static int set_completion_state (int run_state);
```

MyJob method that sets the running job's status (completed successfully, failed, canceled).

Input Arguments

`run_state`	[JobRunState] Current run state.

Return Values

`STATUS_Ok`	Success. If some of the resources were not locked, they are ignored.
`STATUS_NotFound`	Not called from job.

set_progress

```
static int set_progress (string szProgress);
```

MyJob method that sets the running job's progress string.

Input Arguments

`szProgress`	[string] Updates or diagnostic information.

CISCO CONFIDENTIAL

Return Values

STATUS_Ok	Success.
STATUS_NotFound	Not called from job.

unlock

```
static void unlock (string szLockPath);
```

MyJob method that unlocks a resource.

Input Arguments

szLockPath	[string] Device name and any subnode.

Return Values

STATUS_Ok	Success.
STATUS_NotFound	No such job or no such lock.

unlock_all

```
static int unlock_all();
```

MyJob method that releases all the resources for the current job.

Arguments

None

Return Values

STATUS_Ok	Success.
STATUS_NotFound	Not called from job.

About the JRM Java Constants

This section describes the symbolic constants for Java applications. These constants are initialized in the IDL file (see the "About Displayed Job Status Values" section on page 18-28).

Table 18-18 JRM Java Method Return Codes

Constant	Description
STATUS_Ok	Success

CISCO CONFIDENTIAL

Table 18-18 JRM Java Method Return Codes

STATUS_Exists	Entry already exists
STATUS_NotFound	Entry not found
STATUS_EOF	End of iteration

Table 18-19 JRM Job Completion States

Constant	Description
RUNST_NeverRan	The task was never run
RUNST_Running	The task is currently running
RUNST_Succeeded	Task completed successfully
RUNST_SucceededWithInfo	Task completed successfully, returning information
RUNST_Failed	Task ran and failed
RUNST_Crashed	Crashed (“core dump”)
RUNST_LaunchFailed	Job Manager could not start the task for this job
RUNST_Canceled	Canceled by client
RUNST_CanceledInstance	Canceled Instance by client

Table 18-20 JRM Schedule State Bits

Constant	Description
SCHST_RequiresApproval	Set if job requires approval.
SCHST_Enabled	Job is enabled.
SCHST_AM_Mask	Mask for the job approval state. Use (schedule_state & SCHST_AM_Mask) to compare with values below.
SCHST_AM_Waiting	Waiting for approval.
SCHST_AM_Approved	Approved.
SCHST_AM_Rejected	Rejected.

Table 18-21 JRM Schedule Types

Constant	Description
SCHTYPE_Immediate	Run job immediately
SCHTYPE_Once	Run job once
SCHTYPE_Daily	Run every *n* days
SCHTYPE_Weekly	Run every *n* weeks[1]
SCHTYPE_Monthly	Run every *n* months[2]
SCHTYPE_MonthLastDay	Run on the last day of the month every *n* months

CISCO CONFIDENTIAL

Table 18-21 JRM Schedule Types

SCHTYPE_MonthSameXday	Run on the given day (Sunday/Monday/…) of the first/second/… week of the month every *n* months[3]
SCHTYPE_MonthLastXday	Run on the given day of the last week of the month every *n* months[4]
SCHTYPE_S_Seconds	Run every *n* seconds
SCHTYPE_S_Minutes	Run every *n* minutes
SCHTYPE_S_Hours	Run every *n* hours
SCHTYPE_E_Seconds	Run *n* seconds after the previous run ended
SCHTYPE_E_Minutes	Run *n* minutes after the previous run ended
SCHTYPE_E_Hours	Run *n* hours after the previous run ended

1. Start date day of the week.
2. Start date day of the month.
3. Start date week number and the week day.
4. Start date week day.

Note Some calendar options can produce impossible values (for example, run on the 31st of every month or on the 5th Friday of every month). Those impossible dates will be skipped. For example, the job scheduled to run on the 31st of the month will run only for the months that have 31 days.

Parsing ESS Messages

Use the helper class EssMessageCreator to parse and read the variables in the ESS message. After reading the message, your application can create an EssMessageCreator object using the constructor `EssMessageCreator(String message)`. This will parse the details in the message. Your application can then get the values for variables using the member variable of the object. The member variables are shown in Table 18-22.

Table 18-22 ESS Member Variables

Member Variable	Description
Public String Action;	Provides commands such as start, end, etc.
public String szProgress;	Job progress status for all events,. For approve and reject events, this contains approver comments.
public int idJob;	Contains the Job ID.
public int rc;	Return code for some jobs.
public int signalNo;	Contains the signal number for daemon jobs,
public int runState;	Contains the run state value.
public String resource;	Identifies the locked resource.
public String owner;	Identifies the owner who locked or unlocked the resource.
public int instanceid;	Stores the instance ID for job-history jobs.

CISCO CONFIDENTIAL

Using the Job Command-Line Commands

Use jobcli, the job command-line application, to run JRM functions. Table 18-23 summarizes the jobcli commands.

Related Topics

- Understanding the JRM Architecture, page 18-5
- Using JRM from the Command Line, page 18-24

Table 18-23 jobcli Command Summary

Syntax and Description
approve jobId approver Approves a job
cancel jobId Cancels a job
create cmd=command [,descr=description] [,owner=user] [,type=string] [, schedule] Creates a job
delay cmd=command [,descr=description] [,owner=user] [,type=string] Creates and suspends a job
delete jobId Deletes a job
reject jobId rejecter Rejects a job
resume jobId Resumes a job
run cmd=command [,descr=description] [,owner=user] [,type=string] Creates and runs a job
schedule jobId schedule Reschedules a job
suspend jobId Suspends a job
getnextschedule jobId Prints the next shcheduled run time for the job details on clicking on the command link

approve

approve jobId approver

Approves the job **jobId**.

Input Arguments

jobId	[integer] Unique number assigned to a job at creation time.
approver	[string] Account of the person who approved the job. Valid only if approval is required.

CISCO CONFIDENTIAL

cancel

cancel jobId

Cancels the job **jobId**.

Input Arguments

jobId	[integer] Unique number assigned to a job at creation time.

create

create cmd=command [,descr=description] [,owner=user] [,type=string] [, schedule]

Creates a job.

Input Arguments

cmd	[string] Command required to identify the job.
descr	[string] Describes the job.
owner	[string] Account of the person who created the job.
type	[string] Identifies the job type and subtypes (for example, SWIM:update.)
schedule	[string] List of comma-separated fragments that specify when the job will be run initially, how often it is repeated, and the initial schedule state:

- at {date | +minutes}

  date specifies the start datetime as a string. Alternatively, +minutes can be used to start the job in *minutes* minutes from the current time.

- repeat {weekly | monthly | daily | month Last Day | monthSameXday | monthLastXday} [(n)]

  Schedule the job to run periodically on a calendar basis.

- repeat every n {h | m | s}
- repeat after n {h | m | s}

  Schedule a job to run periodically on a time basis. Using the "every" option, the job will run every *N* hours/minutes/seconds. Using the "after" option, the job will run *N* hours/minutes/seconds after the end of the previous execution.

- schst= {W | A | R}

  Sets the state to Waiting for approval / Approved / Rejected.

delay

delay cmd=command [,descr=description] [,owner=user] [,type=string]

CISCO CONFIDENTIAL

Creates a job for immediate execution but in the suspended state. The effect is that the job will be run once it is enabled with the **resume** command.

Input Arguments

`cmd`	[string] Command required to identify the job
`descr`	[string] Describes the job
`owner`	[string] Account of the person who created the job
`type`	[string] Identifies the job type and subtypes (for example, SWIM:update)

delete

`delete` `jobId`

Deletes the job **jobId**.

Input Arguments

`jobId`	[integer] Unique number assigned to a job at creation time.

getnextschedule

`getnextschedule` `jobId`

Prints the next shcheduled run time for the job when the Instance of Job is scheduled for the future.

Input Arguments

`jobId`	[integer] Unique number assigned to a job at creation time.

reject

`reject` `jobId rejecter`

Rejects the job **jobId**.

Input Arguments

`jobId`	[integer] Unique number assigned to a job at creation time.
`rejecter`	[string] Account of the person who rejected the job. Valid only if approval is required.

resume

`resume` `jobId`

CISCO CONFIDENTIAL

Resumes the job **jobId** so it can be scheduled.

Input Arguments

`jobId`	[integer] Unique number assigned to a job at creation time.

run

`run` `cmd=command [,descr=description] [,owner=user] [,type=string]`

Creates a job and runs it immediately.

Input Arguments

`cmd`	[string] Command required to identify the job
`descr`	[string] Describes the job
`owner`	[string] Account of the person who created the job
`type`	[string] Identifies the job type and subtypes (for example, SWIM:update)

schedule

`schedule` `jobId schedule`

Reschedules the job **jobId**.

Input Arguments

`jobId`	[integer] Unique number assigned to a job at creation time.
`schedule`	[string] See "cancel."

suspend

`suspend` `jobId`

Suspends the job **jobId** so it will not be scheduled until it is resumed.

Input Arguments

`jobId`	[integer] Unique number assigned to a job at creation time.