CHAPTER 17

Using the Daemon Manager

The CWCS Daemon Manager provides the following services:

- Maintains the startup dependencies among processes
- Starts and stops processes based on their dependency relationships
- Restarts processes if an abnormal termination is detected
- Monitors the status of processes

The Daemon Manager is useful to applications that have long-running processes that must be monitored and restarted, if necessary. It is also used to start processes in a dependency sequence and to start transient jobs.

The following topics describe the Daemon Manager and how to use it in your applications:

- Understanding the Daemon Manager—Explains the basic concepts of the Daemon Manager and what it can do.
- Using the Daemon Manager—Provides guidelines on using the Daemon Manager in CLI, C, C++, and Java environments.
- Daemon Manager Command Reference—Describes the Daemon Manager CLI, C, C++, and Java methods and commands.

For more information about the Daemon Manager, refer to the engineering specification (EDCS document number ENG 21240).

This specification is also available at the following URL:

`http://wwwin-eng.cisco.com/Eng/ENM/BG_10/Specs/BG1PRocessMgrSpec.doc`

Understanding the Daemon Manager

Applications use a programmatic interface to the Daemon Manager to register and control processes. The Daemon Manager provides interfaces from C, C++, and Java, and through a set of command line utilities. CWCS also provides a GUI for process monitoring, shutdown, and startup.

A process opens a socket connection with the Daemon Manager to:

- Provide status information to the server
- Receive messages concerning changes to the CWCS Server environment

The Daemon Manager can create and start processes:

- Automatically, at startup time: These are called regular processes (or regular daemons), which means they run all the time.

- As needed (for example, by using inetd): These are called transient processes (or transient daemons).

Applications obtain and save run-time configuration information using a standard set of environment variables. The full set of predefined variables is defined in ENG-21240.

Using the Daemon Manager

You can access the Daemon Manager either from the command line or by using an API written in C, C++, or Java. The following topics describe how to create an application that uses the Daemon Manager:

Starting and Stopping the Daemon Manager

On a UNIX Platform

Step 1 Log in as root.

Step 2 To start the Daemon Manager, enter

```
/etc/init.d/dmgtd start
```

Step 3 To stop the Daemon Manager, enter

```
/etc/init.d/dmgtd stop
```

Note You cannot start the Daemon Manager if there are non-SSL compliant applications installed with the web server running in SSL-enabled mode.

On a Windows Platform

Step 1 Open a DOS window.

Step 2 To start the Daemon Manager, enter

```
net start CRMdmgtd
```

Step 3 To stop the Daemon Manager, enter

```
net stop CRMdmgtd
```

Using the Daemon Manager Command Line Interface

The Daemon Manager provides several command line interface (CLI) commands. These commands, which are typically used at install time or for debugging purposes, allow you to query the status of a process. They also help you to stop, start, register, or unregister a process.

The CLI commands can be run from a command line interface or using a web browser. They are summarized in the"Daemon Manager Command Line Utilities" section on page 17-6.

Note The commands to register and unregister a process are only available from the command line interface. They cannot be accessed using a web browser.

Using the Daemon Manager Application Programming Interface

To use the Daemon Manager API:

Step 1 Include the header **dmgt.h** in your C, C++, or Java application.

Step 2 After your application has finished its initialization and before it goes into its main loop, instantiate the connection to the Daemon Manager.

- The C and C++ commands are summarized in the "Daemon Manager ANSI C and C++ Commands" section on page 17-11.
- The Java methods are summarized in the "Daemon Manager Java Methods" section on page 17-17.

Step 3 Send a status message (such as the initialization completed successfully or initialization failed and the reason for the failure).

Step 4 Update your main loop to process messages from the Daemon Manager. If you are not interested in any messages, redirect all messages to the default message handler routine.

Step 5 If the status of your application changes, send a new message to the Daemon Manager so that users can be warned about any problems. This may duplicate some status messages that are already being sent to the system logging service. However, this helps serviceability because the "Last" status message is easily retrieved via the Daemon Manager interface.

Use these guidelines when you register your process under the Daemon Manager:

- Programs should not fork or execute themselves.
- Programs should exit when they receive SIGTERM from the Daemon Manager.
- Programs should create core files only in well-known locations. The Daemon Manager starts processes from the directory where the executables reside.

Using the Daemon Manager C++ Interface

Follow these guidelines when you create a C++ application that interfaces with the Daemon Manager:

- Instantiate the C++ singleton object after completing program initialization. The instantiation should exist as long as the program is active.
- Any message loop that is created must listen to the Daemon Manager and deal with any messages. Send any messages not processed by the application to the default handler dMgr::ProcessMsg().
- The program must not fork or execute itself.
- The program must not perform daemon actions, such as trying to restart itself after it fails. Only the Daemon Manager should do this.
- After initializing, run the command, **SendOkMsg()** (see the "SendOkMsg" section on page 17-22).

Related Topics

- The ANSI C and C++ commands are summarized in the "Daemon Manager ANSI C and C++ Commands" section on page 17-11.
- For examples that use the C++ interface, refer to these files in the CodeSamples directory on the SDK CD:
  - daemon1.cpp
  - sample1.dsp
  - sample1.cpp

Using the Daemon Manager Java Interface

The **cwjava** command provides a controlled run-time environment for CWCS-based Java server applications. For information about launching a Java application, see:

- Understanding the Java Application Launch Process, page 4-1
- Launching a Java Application, page 4-2

Related Topics

- The methods you can use to manage processes in Java applications are summarized in the "Daemon Manager Java Methods" section on page 17-17.
- For examples that use the Java interface, see these files in the CodeSamples directory on the SDK CD:
  - daemon1.java
  - sample1.java

Using a Ready File to Ensure Process Dependencies are Met

If you have a process that takes a long time to start up, and has many dependent processes that start up quickly, you may experience problems with Daemon Manager starting the dependent processes before the underlying process has completed initialization.

In cases like these, call dMgtCreateReadyFile (for C and C++ processes) or CreateReadyFile (for Java) immediately after the underlying process' initialization code segment. Daemon Manager will wait until the API is called and the Ready file created before starting up any dependent processes. For more information on these Daemon Manager APIs, see the "dMgtCreateReadyFile" section on page 17-12 and the "CreateReadyFile" section on page 17-18.

If you want to use a Ready file to ensure startup dependencies are met, you must also specify a timeout value for the Ready file creation process when it first registers with Daemon Manager. You can do this using DmgrRegisterWR (for C and C++ processes) or DmgrRegisterJavaWR (for Java processes) . For more information on these installation APIs, see the "DmgrRegister" section on page 21-51.

Ensuring that underlying processes are fully initialized is especially important if your application uses multiple database engines on which other processes depend. To ensure that this is possible, the CWCS database APIs provide an alternative database monitor class, DBPing. For more information about how to use DBPing in combination with the Ready file creation and special "WR" registration APIs, see the "DBPing" section on page 11-56.

Writing Messages to Log Files

There are two types of log files:

- The Daemon Manager log contains information regarding process start, termination, and Daemon Manager warning and error messages.
- The Application log stores information logged by an application. To write a message to the application log, direct the output to stderr/stdout.

The location of the log files is determined by the operating system:

- On Windows platforms:
  - The Daemon Manager log is located under *NMSROOT*/log/syslog.log.
  - Each application has its own application log under *NMSROOT*/log.
  - The application log has same name as the application with the extension ".log".
- On UNIX platforms:
  - The Daemon Manager log is located at /var/adm/CSCOpx/log/dmgtd.log.
  - All applications share the same application log: /var/adm/CSCOpx/log/daemons.log.

Daemon Manager Command Reference

The Daemon Manager provides these interfaces:

- Daemon Manager Command Line Utilities
- Daemon Manager ANSI C and C++ Commands
- Daemon Manager Java Methods

Daemon Manager Command Line Utilities

Use the CLI commands shown in Table 17-1 to query the status of a process and perform other tasks.

Table 17-1 Daemon Manager CLI Commands

Syntax	Description
pdexec *AppName*	Starts a process
pdrapp *AppName*	Establishes root access for an application
pdreg -r *AppName* **-e** *PathName* **-f** *flags* **-d** *OtherAppName* **-n -t** *code* **pdreg -l** *AppName* **pdreg -u** *AppName*	Register, list, or unregister a process
pdshow *AppName*	Monitors all registered processes
pdterm *AppName*	Shuts down a process

pdexec

pdexec *AppName*

Starts a process.

Before starting a process, the Daemon Manager examines the dependencies of the starting process and if they have not already been started, starts those processes first.

If the process is being restarted *after a shutdown*, any dependent processes registered with the Daemon Manager is not automatically restarted. Dependent processes are automatically restarted only when the Daemon Manager itself is restarted.

Arguments

AppName	[string] Application name.

Return Values

OK	Starts the process.
Error	Prints an error message on stderr and syslog.

Usage Guidelines

Normal process startup may occur in two ways:

- If the process enables autostart (the default), the server invokes the process when the server is started.
- If the operator invokes a command or runs a shell script to start the application.

pdrapp

pdrapp *AppName*

Establishes root access for an application. This command adds the application name to the file **rootapps.conf** so the application can be launched as root.

Arguments

AppName	[string] Application name.

Return Values

OK	Appends the application name in the file.
Application name is already present in the file	Prints “appname already exists” on stderr and syslog.

pdreg

```
pdreg -r AppName -e PathName -f flags -d OtherAppName -n -t code -w timeout
pdreg -l AppName
pdreg -u AppName
```

The pdreg command provides three functions:

- Register a process (-r)—A process must be registered before it can be monitored or controlled by the Daemon Manager.
- Unregister a process (-u)—You must unregister a process if you no longer want the process be managed by the Daemon Manager. The process must be registered again to bring it back under the control of the Daemon Manager.
- Display the registration information of a registered process(-l).

Note After a process is registered, all administrative tasks performed on the process must use the Daemon Manager; otherwise the Daemon Manager information will become unreliable, resulting in an *unstable system*. When a process is registered, stop it using the *pdterm* command; do not terminate it directly.

Arguments

-r *AppName*	[string] A descriptive name for the application.
-e *PathName*	[string] Fully qualified path name to where the program exists.
-f *flags*	[integer] Any startup flags required by the application.

-d *OtherAppName* [string] List of application names that must be started before *AppName.*

- On UNIX platforms, use a comma-separated list. For example:

```
/opt/CSCOpx/bin/pdreg -r Daemon1 -e /opt/CSCOpx/bin/Daemon1.os -d
Daemon2,Daemon3 -f -debug^1^-trace^0 -n
```

- On Windows platforms, group within quotes. For example:

```
Pdreg -r Daemon1 -e D:\Progra~1\CSCOpx\bin\Daemon1.os -d "Daemon2
Daemon3" -f "-debug 1 -trace 0" -n
```

-n [string] Do *not* automatically start this application when the Daemon Manager is started. Default: Daemon Manager starts the application.

-t *returnCode* [string] Specifies the normal return code of a transient process. Upon the termination of a process, its return code is checked against the normal *returnCode*. The process is restarted immediately by the Daemon Manager if they do not match. The *returnCode* can be one of the following values:

0—Normal return code is zero (default)

n—Normal return code is a negative value.

p—Normal return code is a positive value.

o—Return code is not checked. No restart is required.

-l *AppName* [string] Displays the registry contents for an application, the registered processes, and their attributes.

-u *AppName* [string] Unregisters a process.

If a server that is being stopped and unregistered is needed by other applications (as specified in the dependency list) the effect is to cascade the shutdown of those servers as well. Dependencies on unregistered (unknown) applications are ignored.

-w *timeout* [integer] Specifies the maximum amount of time (in milliseconds) that Daemon Manager should wait for the process to finish initialization.

This timeout value can be passed to `pdreg` during installation using a DmgrRegisterWR or DmgrRegisterJavaWR API call (see the "Registering and Unregistering CWCS Daemons" section on page 21-51).

Return Values

OK -r—Process is registered.

-l—Registry contents (PathName, flags, OtherAppName, etc.) are displayed.

-u—Process is unregistered.

Error Prints an error message on stderr and syslog.

Usage Guidelines

- If the program being registered is autostart, start the process now.

- A *dependency list* defines other programs that must be running for this program to be successfully invoked. If the dependency list is valid, other programs are started if they are not already running.
- Specify dependencies on other servers as a comma-separated list with no blanks. For example:

```
pdreg -r MyServer -e /bin/ls -d AppName1,AppName2
```

- Registering an existing name returns an error. To change an entry, first unregister the process.
- Invalid or nonexisting application names can be registered; they are ignored on startup.
- To register more than one command line flag, or one that requires spaces, use the caret (^) instead of spaces. For example:

  On UNIX platforms, enter (all on one line):

```
/opt/CSCOpx/bin/pdreg -r Daemon1 -e /opt/CSCOpx/bin/Daemon1.os -d Daemon2,Daemon3 -f
-debug^1^-trace^0 -n
```

  On Windows platforms, enter (all on one line):

```
Pdreg -r Daemon1 -e D:\Progra~1\CSCOpx\bin\Daemon1.os -d "Daemon2 Daemon3" -f "-debug
1 -trace 0" -n
```

  After registration, the Daemon Manager tries to start the autostart process if the dependency requirement has been met.
- Java programs should use the cwjava executable, which is the CWCS Java2 wrapper. For example, to register a JRM process, use the following statement:
  - On UNIX platforms, enter (all on one line):

```
pdreg -r jrm -e /opt/CSCOpx/bin/cwjava -d CmfDbMonitor,RmeOrb,EDS -f
com.cisco.nm.cmf.jrm.Server
```

  - On Windows platforms, enter (all on one line):

```
pdreg -r jrm -e C:\PROGRA~1\CSCOpx\bin\cwjava.exe -d "CmfDbMonitor RmeOrb EDS" -f
com.cisco.nm.cmf.jrm.Server
```

    Where com.cisco.nm.cmf.jrm is the jrm package path and Server.class is the main program.

  To check the possible cwjava options, enter the following statement:
  - On UNIX platforms, enter:

```
/opt/CSCOpx/bin/cwjava -cw /opt/CSCOpx -help
```

  - On Windows platforms, enter:

```
C:\PROGRA~1\CSCOpx\bin\cwjava.exe -cw C:\PROGRA~1\CSCOpx -help
```

    Where -cw option is used to specify your CiscoWorks2000 installation directory.

pdshow

pdshow *AppName*

Generates a list of all registered processes, their current state (“up”, “down,” and so on), and the attributes used while registering a process.

Arguments

AppName	[string] Application name.

Return Values

OK	Lists all registered processes and their current state.
Error	Prints an error message on stderr and syslog.

Usage Guidelines

If no name is supplied, a list of current applications and their status is sent to stdout. The output detects if the environment is Web-based and displays the status in HTML markup or straight text to a TTY display.

pdterm

`pdterm` *AppName*

Shuts down a process.

Note Any processes registered with the Daemon Manager as dependents of this process are also shut down. However, if the process is then restarted after a shutdown, these dependent processes are *not* automatically restarted. Dependent processes are automatically restarted only when the Daemon Manager itself is restarted.

Arguments

AppName	[string] Application name.

Return Values

OK	Shuts down the process.
Error	Prints an error message on stderr and syslog.

Usage Guidelines

- Normal shutdown of a process may occur in two ways:
  - The operator issues a shutdown to the Daemon Manager for a particular server.
  - The server (if transient) sends a request to the Daemon Manager to shut down.

  The Daemon Manager then kills the application.
- Processes that are registered with the Daemon Manager and started manually and are then connected to the Daemon Manager to report that status cannot be stopped with **pdterm**. The pid value is shown as zero.

Daemon Manager ANSI C and C++ Commands

Use the ANSI C and C++ programming interfaces shown in Table 17-2 to query the status of a process and perform other tasks. The C++ features are defined in the C++ class wrapper.

Table 17-2 Daemon Manager ANSI C and C++ Commands

Returns	Syntax and Description
int	**dMgtClose** (const dMGTHDL dHandle) Closes a connection
int	**dMgtCreateReadyFile** (char **AppName*, char **errmsg*) Creates a Ready file after an underlying process (on which other processes depend) has initialized.
const char *const	**dMgtErr** (const dMGTHDL hdl) Retrieves the error code after Daemon Manager has returned an error (-1).
int	**dMgtGetMsg** (const dMGTHDL hdl) Reads the Daemon Manager message and places it in the application's buffer.
int	**dMgtInit** (dMGTHDL *pdHandle, const char *pszAppName) Establishes connection to the Daemon Manager. The Daemon Manager returns a handle (dMGTHDL) to be used by the client application in subsequent requests.
int	**dMgtIsShutdown** (const dMGTHDL hdl) Checks the incoming request to determine if it is a STOP command.
void	**dMgtProcessMsg** (const dMGTHDL hdl) Processes Daemon Manager requests.
int	**dMgtSendStatus** (const dMGTHDL hdl, dMGT_STATE state, const char *pszStatusMsg) Sends a message and changes the status of the application.
char*	**GetConFile** (char *pszConfile) Gets the Daemon Manager configuration filename.
int	**GetDescriptor** (const dMGTHDL hdl) Gets the socket descriptor for the I/O port.
int	**GetDmgtHostAndPort** (char **ppHost, short *psPort, char **ppErrMsg) Gets the name of the Daemon Manager host and port number. This information is defined in the environment variable PX_DMGTHOST. If PX_DMGTHOST is not set, then "localhost" (127.0.0.1) is used.
int	**ValidatePgmPath** (char *pszPgm, char **ppErrMsg) Checks to determine if the *pszPgm* is a fully qualified path to a program (that is, /path/filename).

dMgtClose

```
int   dMgtClose (const dMGTHDL dHandle)
```

Closes the connection to the Daemon Manager.

Input Arguments

`dHandle` [dMGTHDL] Handle to the client application. dMGTHDL is defined when the application uses dMgtInit to connect to the Daemon Manager (see the "dMgtInit" section on page 17-13).

Return Values

`0` OK.

`-1` Invalid handle, message error, etc.).

dMgtCreateReadyFile

`int` **`dMgtCreateReadyFile`** `(char *`*`AppName`*`, char *`*`errmsg`*`)`

Creates a Ready file after an underlying process (on which other processes depend) has initialized. The Daemon Manager checks the Ready file to ensure that the underlying process has met the dependency requirements before starting the dependent processes.

Input Arguments

AppName [char] Client application name.

Output Arguments

errmsg [char] ASCII string.

Return Values

`0` OK.

`-1` Error (failed to open file, etc.).

dMgtErr

`const char *const` **`dMgtErr`** `(const dMGTHDL hdl)`

Retrieves the error code after Daemon Manager has returned an error (-1).

Input Arguments

`hdl` [dMGTHDL] Handle to the client application that encountered the error condition. dMGTHDL is defined when the application uses dMgtInit to connect to the Daemon Manager (see the "dMgtInit" section on page 17-13).

Return Values

`Error code`	Error code.
`"no details on error"`	No error code is available.

dMgtGetMsg

```
int   dMgtGetMsg (const dMGTHDL hdl)
```

Reads the Daemon Manager message and places it in the application's buffer.

Output Arguments

`hdl`	[dMGTHDL] Handle to the client application. dMGTHDL is defined when the application uses dMgtInit to connect to the Daemon Manager (see the "dMgtInit" section on page 17-13).

Return Values

`0`	OK.
`-1`	Invalid handle, read error, etc.

dMgtInit

```
int   dMgtInit (dMGTHDL *pdHandle, const char *pszAppName)
```

Establishes connection to the Daemon Manager. The Daemon Manager returns a handle (dMGTHDL) to be used by the client application in subsequent requests.

Input Arguments

pszAppName	[char] Client application name in ASCII format.

Output Arguments

`pdHandle`	[dMGTHDL] Handle to the client application.

Return Values

`0`	OK.
`-1`	Error.

dMgtIsShutdown

```
int   dMgtIsShutdown (const dMGTHDL hdl)
```

Checks the incoming request to determine if it is a STOP command.

Output Arguments

`hdl` [dMGTHDL] Handle to the client application. dMGTHDL is defined when the application uses dMgtInit to connect to the Daemon Manager (see the "dMgtInit" section on page 17-13).

Return Values

`0`	Incoming request is NOT a STOP command.
`1`	Incoming request is a STOP command.
`-1`	Invalid handle, message error, etc.

dMgtProcessMsg

```
void dMgtProcessMsg (const dMGTHDL hdl)
```

Processes Daemon Manager requests.

Output Arguments

`hdl` [dMGTHDL] Handle to the client application. dMGTHDL is defined when the application uses dMgtInit to connect to the Daemon Manager (see the "dMgtInit" section on page 17-13).

Return Values

None

dMgtSendStatus

```
int dMgtSendStatus (const dMGTHDL hdl,
        dMGT_STATE state,
        const char *pszStatusMsg)
```

Sends a message and changes the status of the application.

Input Arguments

`state` [dMGT_STATE] Application status. Allows these values:

- DMGT_READY: Application is OK.
- DMGT_FAILED: Application failed to initialize.
- DMGT_BUSY: Application does not respond to server message.

`pszStatusMsg` [char] Application message in ASCII format. Maximum size is 120 characters.

Output Arguments

`hdl` [dMGTHDL] Handle to the client application. dMGTHDL is defined when the application uses dMgtInit to connect to the Daemon Manager (see the "dMgtInit" section on page 17-13).

Return Values

`0` Operation successfully executed.

`-1` Invalid handle, invalid state, failed to write to message buffer.

GetConFile

`char* **GetConFile** (char *`*pszConfile*`)`

Gets the Daemon Manager configuration filename.

Output Arguments

pszConfile [char] Pointer to the Daemon Manager configuration file.

Return Values

pszConfile Pointer to the Daemon Manager configuration file.

GetDescriptor

`int **GetDescriptor** (const dMGTHDL hdl)`

Gets the socket descriptor for the I/O port.

Output Arguments

`hdl` [dMGTHDL] Handle to the client application. dMGTHDL is defined when the application uses dMgtInit to connect to the Daemon Manager (see the "dMgtInit" section on page 17-13).

Return Values

`socket descriptor`	Socket descriptor for I/O port.
`-1`	Invalid handle, read error, etc.

GetDmgtHostAndPort

```
int GetDmgtHostAndPort (char **ppHost,
        short *psPort,
        char **ppErrMsg)
```

Gets the name of the Daemon Manager host and port number. This information is defined in the environment variable PX_DMGTHOST. If PX_DMGTHOST is not set, then "localhost" (127.0.0.1) is used.

Note The caller has to free the memory pointed to by ppHost, using *free()*.

Output Arguments

`ppHost`	[char] Pointer to Daemon Manager host name.
`psPort`	[short] Pointer to Daemon Manager port number.
`ppErrMsg`	[char] ASCII string to indicate any error.

Return Values

`0`	OK.
`-1`	Invalid handle, read error, etc.

ValidatePgmPath

```
int ValidatePgmPath (char *pszPgm,
        char **ppErrMsg)
```

Checks to determine if the string to be validated is a fully qualified path to a program (that is, /path/filename).

Input Arguments

`pszPgm`	[int] String to be validated.

Output Arguments

`ppErrMs`	[char] ASCII string to indicate any error.

Return Values

`0`	OK.
`-1`	Invalid handle, read error, etc.

Daemon Manager Java Methods

Use the public Java interfaces shown in Table 17-3 to query the status of a process and perform other tasks.

Table 17-3 Daemon Manager Java Methods

Returns	Syntax and Description
`int`	**`CreateReadyFile`** `(String` *AppName*`)` Creates a Ready file after an underlying process (on which other processes depend) has initialized.
`int`	**`GetCmdType`** `()` Gets the command type exacted from the message.
`Socket`	**`GetDescriptor`** `()` Gets the Socket object to listen for messages from Daemon Manager.
`String`	**`GetErr`** `()` Gets the current "last error text" from the Daemon Manager. This routine is usually called when an error condition is returned from the Daemon Manager.
`boolean`	**`GetMsg`** `()` Retrieves a Daemon Manager message from the IP stack.
`boolean`	**`GetServerInfo`** `(String` *hostname*`, Integer` *portinfo*`)` Gets the Daemon Manager server name and port number.
`String`	**`GetStatusMsg`** `()` Gets the status of the process from Daemon Manager.
`boolean`	**`IsShutdownRequest`** `()` Checks to determine if the incoming request is a STOP command.
`void`	**`ProcessMsg`** `()` Processes Daemon Manager requests.
`int`	**`ReqStatus`** `(String` *appName*`)` Asks Daemon Manager to report the status of the process specified by appName.
`int`	**`SendBusyMsg`** `(String` *StatusMsg*`)` Notifies Daemon Manager that it is ready and running. Also tells the Daemon Manager not to send any broadcast messages to this process. The input string *StatusMsg* appears in the pdshow output.
`int`	**`SendErrMsg`** `(String` *StatusMsg*`)` Notifies Daemon Manager that the process failed to run. The input string *StatusMsg* is the reason for the failure and appears in the pdshow output.
`int`	**`SendOkMsg`** `(String` *StatusMsg*`)` Notifies Daemon Manager that it is ready and running. The input string *StatusMsg* appears in the pdshow output.

Table 17-3 Daemon Manager Java Methods (continued)

Returns	Syntax and Description
int	**StartProcess** (String *appName*)) Asks Daemon Manager to start another process specified by appName.
int	**StopProcess** (String *appName*) Asks Daemon Manager to stop another process specified by appName.
int	**Status** () Gets current dMgt object status after the dMgt constructor is called.

CreateReadyFile

int **CreateReadyFile** (String *AppName*)

Creates a Ready file after an underlying process (on which other processes depend) has initialized. The Daemon Manager checks the Ready file to ensure that the underlying process has met the dependency requirements before starting the dependent processes.

Input Arguments

AppName	[string] Application name.

Return Values

0	OK
-1	Error (failed to open file, etc.).

GetCmdType

int **GetCmdType** ()

Gets the command type exacted from the message.

Arguments

None

Return Values

Returns the command type.

GetDescriptor

Socket **GetDescriptor** ()

Gets the Socket object to listen for messages from Daemon Manager.

Arguments

None

Return Values

Success	Returns the Socket object.
Failure	Null.

GetErr

```
String GetErr ()
```

Gets the current "last error text" from the Daemon Manager. This routine is usually called when an error condition is returned from the Daemon Manager.

Arguments

None

Return Values

Returns the current "last error text" associated with this process.

GetMsg

```
boolean GetMsg ()
```

Retrieves a Daemon Manager message from the IP stack. This routine waits (blocks) if no data is pending.

Arguments

None

Return Values

True	Message retrieved.
False	Message not retrieved. Call GetErr() to examine the reason for the failure.

GetServerInfo

```
boolean GetServerInfo (String hostname, Integer portinfo)
```

Gets the Daemon Manager server name and port number.

Input Arguments

portinfo	[integer] The port number to be used for the DMGTD connection.

Output Arguments

`hostname`	[string] The host name where the DMGTD is located.

Return Values

Always returns true.

GetStatusMsg

```
String GetStatusMsg ()
```

Gets the status of the process from Daemon Manager.

Arguments

None

Return Values

Returns the current status text associated with this process.

IsShutdownRequest

```
boolean IsShutdownRequest ()
```

Checks to determine if the incoming request is a STOP command.

Arguments

None

Return Values

`0`	Incoming request *is not* a STOP command.
`1`	Incoming request *is* a STOP command.
`-1`	Invalid handle, message error, etc.

ProcessMsg

```
void ProcessMsg ()
```

Processes Daemon Manager requests.

Arguments

None

Return Values

None

ReqStatus

`int` **`ReqStatus`** `(String` *appName*`)`

Asks Daemon Manager to report the status of the process specified by appName.

Input Arguments

appName	[String]—Application name.

Return Values

`0`	Success. Status report sent.
`-1`	Failure. Status report not sent.

SendBusyMsg

`int` **`SendBusyMsg`** `(String` *StatusMsg*`)`

Notifies Daemon Manager that it is ready and running. Also tells the Daemon Manager not to send any broadcast messages to this process. The input string *StatusMsg* appears in the pdshow output (see the "pdshow" section on page 17-9).

Input Arguments

StatusMsg	[string] Status message, "Program not listening to dmgtd."

Return Values

`0`	Success
`-1`	Failure

SendErrMsg

`int` **`SendErrMsg`** `(String` *StatusMsg*`)`

Notifies Daemon Manager that the process failed to run. The input string *StatusMsg* is the reason for the failure and appears in the pdshow output (seethe "pdshow" section on page 17-9).

Input Arguments

StatusMsg	[string] Status message, "Application failed to initialize."

Return Values

`0`	Success
`-1`	Failure

SendOkMsg

`int` **`SendOkMsg`** `(String` *StatusMsg*`)`

Notifies Daemon Manager that it is ready and running. The input string *StatusMsg* appears in the pdshow output (seethe "pdshow" section on page 17-9).

Input Arguments

StatusMsg	[string] Status message, "Program initialized ok."

Return Values

`0`	Success
`-1`	Failure

StartProcess

`int` **`StartProcess`** `(String` *appName*`))`

Asks Daemon Manager to start another process specified by appName.

Input Arguments

appName	[string] Application name.

Return Values

`0`	Success. Start request sent.
`-1`	Failure. Start request was not sent.

StopProcess

`int` **`StopProcess`** `(String` *appName*`)`

Asks Daemon Manager to stop another process specified by appName.

Input Arguments

appName	[string] Application name.

Return Values

`0`	Success. Stop request sent.
`-1`	Failure. Stop request was not sent.

Status

```
int Status ()
```

Gets current dMgt object status after the dMgt constructor is called.

Arguments

None

Return Values

`0`	Success. Constructor completed.
`-1`	Failure. Constructor not completed.