Java Generic e*Way Extension Kit Developer's Guide

Release 5.0.5 for Schema Run-time Environment (SRE)

Sun

ORACLE®

Contents

Chapter 5

Contents

Contents

Chapter 7

Chapter 8

Chapter 9

154

Introduction

The *Java Generic e*Way Extension Kit Developer's Guide* enables you to develop e*Ways or other e*Gate applications using Java. This document describes how to install, extend, and configure the Java Generic e*Way and the Java Monk Extension e*Way.

The Extension Kit is comprised of two components:

- The Java Generic e*Way, an executable component that manipulates Events or other data using instructions written in Java.

- An extension to the Monk programming language, which a developer can use to access Java methods or objects from within Monk code.

1.1 Intended Reader

The reader of this guide is presumed to be a developer or system administrator with responsibility for maintaining the e*Gate system; to have expert-level knowledge of Windows and UNIX operations and administration, and to be thoroughly familiar with Windows-style GUI operations. We also recommend that the reader have a thorough understanding of the following:

- C and C++ programming languages, Java, Java Native Interface (JNI).

- Basic knowledge of the Monk programming language.

- The external application for which the extension is being written.

1.2 Supported Operating Systems

For more information about supported operating systems, see the **readme.txt** file provided on the installation CD.

1.3 System Requirements

To use the Java Generic e*Way, you need the following:

- An e*Gate Participating Host.

- A TCP/IP network connection.

- Java JDK (see the **readme.txt** file on the installation media for version information)

- Additional disk space for e*Way executable, configuration, library, and script files. The disk space is required on both the Participating and the Registry Host. Additional disk space is required to process and queue the data that this e*Way processes. The amount necessary varies based on the type and size of the data being processed and any external applications performing the processing.

1.4 Introducing the Java Virtual Machine

The Java Virtual Machine (JVM) is an abstract computing machine. Like a physical machine, it has an instruction set and manipulates various memory areas at run-time. The JVM is the software implementation of a CPU (Central Processing Unit) designed to run compiled Java code. This includes stand-alone Java applications, as well as "applets" that are downloaded and run in Web browsers.

Code for the JVM is contained within **.class** files. Each **.class** file contains the code for a public class. The JVM format uses byte code. Byte code resembles machine code, but is not limited to any one processor. It is executable on any operating system that supports the Java run-time system. As a run-time system, JVM links, initializes and executes Java classes.

The JVM automatically performs garbage collection. If the JVM can verify that a given Java object will not be accessed again during the execution of the Java program, the garbage collector can free the memory consumed by that object for reuse. This contrasts with other languages that require the programmer to track run-time memory usage and de-allocate memory that is no longer in use.

Installation

This chapter explains procedures for installing the Java Generic e*Way.

- **Installing on Windows Systems** on page 12
- **Installing on UNIX** on page 13
- **Files/Directories Created by the Installation** on page 17

2.1 Installing on Windows Systems

2.1.1. Pre-installation

- Exit all Windows programs before running the setup program, including any anti-virus applications.
- You must have Administrator privileges to install this e*Way.

2.1.2. Installation Procedure

To install the Java Generic e*Way on a Windows system

1 Log in as an Administrator to the workstation on which you are installing the e*Way.

2 Insert the e*Way installation CD-ROM into the CD-ROM drive.

3 If the CD-ROM drive's Autorun feature is enabled, the setup application launches automatically; skip ahead to step 4. Otherwise, use the Windows Explorer or the Control Panel's **Add/Remove Applications** feature to launch the file **setup.exe** on the CD-ROM drive.

4 The InstallShield setup application launches. Follow the installation instructions until you come to the **Please choose the product to install** dialog box.

5 Select **e*Gate Integrator**, then click **Next**.

6 Follow the on-screen instructions until you come to the second **Please choose the product to install** dialog box.

7 Clear the check boxes for all selections except **Add-ons**, and then click **Next**.

8 Follow the on-screen instructions until you come to the **Select Components** dialog box.

9 Highlight (but do not check) **e*Ways**, and then click the **Change** button. The **Select Sub-components** dialog box appears.

10 Select the **Java Generic e*Way**. Click the continue button to return to the **Select Components** dialog box, then click **Next**.

11 Follow the rest of the on-screen instructions to install the Java Generic e*Way. Be sure to install the e*Way files in the suggested **client** installation directory. The installation utility detects and suggests the appropriate installation directory. *Unless you are directed to do so by Oracle support personnel, do not change the suggested installation directory setting.*

Note: *Once you have installed and configured this e*Way, you must incorporate it into a schema by defining and associating the appropriate Collaborations, Collaboration Rules, IQs, and Event Types before this e*Way can perform its intended functions. For more information about any of these procedures, please see the online Help.*

*For more information about configuring e*Ways or how to use the e*Way Editor, see the **e*Gate Integrator User's Guide**.*

2.2 Installing on UNIX

2.2.1. Pre-installation

You do not require root privileges to install this e*Way. Log in under the user name that you wish to own the e*Way files. Be sure that this user has sufficient privileges to create files in the e*Gate directory tree.

Important Requirements for the Java 2 SDK on UNIX Systems

▪ Do not move **Java 2 SDK** to any other location than where it was installed by the installation process. Upon installation, the location of the **Java 2 SDK** was entered into the operating system's Online Database Management (ODM). Changing the location prevents the proper execution of the Java JNI DLL needed by the JCS.

▪ The user environment on the Participating Host must have the DLL search path environment variable (actual names vary according to the OS) set appropriately to include all directories of the Java 2 SDK installation that contain shared libraries (extensions vary according to OS). See the table below for more information.

OS	DLL Search Path Environment Variable	Extension
AIX	LIBPATH	.a
Solaris, Linux	LD_LIBRARY_PATH	.so
HP-UX11	SHLIB_PATH	.sl

For AIX Participating hosts only:

To prevent any problems with the Java 2 SDK please apply the following Program Temporary Fixes (PTFs):

PTF#	APAR#
PTF 1	IYO8084
PTF 2	IY09226
PTF 3	IY10020
PTF 4	IY10427
PTF 5	IY11206

In the event the above PTFs are not installed, the LIBPATH environment variable must be set to the following:

- The **jre/bin** directory first, followed by the **jre/bin/classic** directory, followed by the directories of other software as needed.

    For example, if Java 2 SDK was installed under **/usr/java_dev**, then:

    for Bourne Shell or Korn Shell users:

    ```
    LIBPATH=/usr/java_dev/jre/bin:/usr/java_dev/jre/bin/classic:$LIBPATH
    ```

    should be added into the egateclient.sh file, **immediately** prior to the "export LIBPATH" statement.

    For C-shell users:

    ```
    setenv LIBPATH /usr/java_dev/jre/bin:/usr/java_dev/jre/bin/
    classic:`printenv LIBPATH`
    ```

    should be added after the current statements that set LIBPATH.

This intervention is necessary because the Java JNI DLL may cause a core unless the LIBPATH is set as described above.

Important Requirements for the Java JDK on UNIX Systems

- The user environment on the Participating Host must have the DLL search path environment variable (actual names vary according to the OS) set appropriately to include all directories of the Java JDK installation that contain shared libraries (extensions vary according to OS). See the table below for more information.

OS	DLL Search Path Environment Variable	Extension
AIX	LIBPATH	.a
Solaris, Linux	LD_LIBRARY_PATH	.so
HP-UX11	SHLIB_PATH	.sl

- For Solaris, the "**native_threads**" version of the JNI DLL, **libjava.so**, must be used.

- For HP-UX, the "**green_threads**" version of the JNI DLL, **libjava.sl**, must be used.

2.2.2. Installation Procedure

To install the Java Generic e*Way on a UNIX system

1 Log in on the workstation containing the CD-ROM drive, and insert the CD-ROM into the drive.

2 If necessary, mount the CD-ROM drive.

3 At the shell prompt, type

   cd /cdrom

4 Start the installation script by typing

   setup.sh

5 A menu of options will appear. Select the **Install e*Way** option. Then, follow the additional on-screen directions.

*Note: Be sure to install the e*Way files in the suggested **client** installation directory. The installation utility detects and suggests the appropriate installation directory.*
Unless you are directed to do so by Oracle support personnel, do not change the suggested "installation directory" setting.

6 After installation is complete, exit the installation utility and launch the Schema Designer.

*Note: Once you have installed and configured this e*Way, you must incorporate it into a schema by defining and associating the appropriate Collaborations, Collaboration Rules, IQs, and Event Types before this e*Way can perform its intended functions. For more information about any of these procedures, please see the online Help system.*

*For more information about configuring e*Ways or how to use the e*Way Editor, see the **e*Gate Integrator User's Guide**.*

2.3 Defining e*Way Components

2.3.1. Creating a Java Generic e*Way

To create and configure a new Java Generic e*Way:

1 Launch the Schema Designer.

2 In the Component editor, create a new e*Way.

3 Display the new e*Way's properties.

4   On the General tab, under **Executable File**, click **Find**.

5   Select the file **stcewgenericjava.exe**.

6   Under **Configuration file**, click **New**.

7   The e*Way Editor will launch. Make any necessary changes, then save the configuration file.

8   You will return to the e*Way's property sheet. Click **OK** to close the properties sheet, or continue to configure the e*Way.

Note: *Once you have installed and configured this e*Way, you must incorporate it into a schema by defining and associating the appropriate Collaborations, Collaboration Rules, IQs, and Event Types before this e*Way can perform its intended functions. For more information about any of these procedures, please see the online Help system.*

*For more information about configuring e*Ways or how to use the e*Way Editor, see the* **e*Gate Integrator User's Guide**.

2.3.2. Creating a Java Monk Extension e*Way

To create and configure a new Java Monk Extension e*Way:

1   Launch the Schema Designer.

2   In the Component editor, create a new e*Way.

3   Display the new e*Way's properties.

4   On the General tab, under **Executable File**, click **Find**.

5   Select the file **stcewgenericmonk.exe**.

6   Under **Configuration file**, click **New**.

7   From the **Select an e*Way template** list, select **stcewjava** and click **OK**.

8   The e*Way Editor will launch. Make any necessary changes, then save the configuration file.

9   You will return to the e*Way's property sheet. Click **OK** to close the properties sheet, or continue to configure the e*Way.

Note: *Once you have installed and configured this e*Way, you must incorporate it into a schema by defining and associating the appropriate Collaborations, Collaboration Rules, IQs, and Event Types before this e*Way can perform its intended functions. For more information about any of these procedures, please see the online Help system.*

*For more information about configuring e*Ways or how to use the e*Way Editor, see the* **e*Gate Integrator User's Guide**.

2.4   Files/Directories Created by the Installation

The Java Generic Extension Kit includes the following files, which are stored within the e*Gate directory tree. Files will be stored within the "egate\client" tree on the Participating Host and committed to the "default" schema on the Registry Host.

Table 1   Files created by the installation

e*Gate Directory	File(s)
bin\	stcewgenericjava.exe stcewgenericmonk.exe stc_monkjava.dll stc_monkjava2.dll
configs\stcgenericmonk\	stcewgenericjava.def stcewjava.def
monk_library\	ewjava.gui
monk_library\ewjava\	java-verify.monk java-startup.monk java-shutdown.monk java-outgoing.monk java-notify.monk java-nack.monk java-init.monk java-extconnect.monk java-exchange.monk java-ack.monk

Introducing the Java Generic e*Way

This *Java Generic e*Way Extension Kit* allows the developer to use either the Java Generic e*Way or the Java Monk Extension e*Way to interchange data with external applications. The Java Generic e*Way uses Java exclusively to exchange data, while the Java Monk Extension e*Way uses Monk to access Java objects and to call Java methods. The first seven chapters of this manual include instructions on how to:

- Create a **.def** file for use with your extended Java Generic e*Way configuration window.

- Implement Java methods to control the data transmission to/from the Java Generic e*Way.

Programming through the **Exchanger** Interface framework enables you to create a bridge between the e*Gate system and external applications, using Java methods to perform a wide range of operations such as wrapping legacy applications.

3.1 Java Generic e*Way Components

The Java Generic e*Way connects the e*Gate system to an external system or database, using the appropriate communication protocol and applicable libraries.

The Java Generic e*Way contains the following components:

- **stcewgenericjava.exe**, an executable file

- **stcewgenericjava.def**, an executable configuration definition file

- **Exchanger.java** Interface, an interface containing Java methods that are required to be implemented by a class file in order to access e*Gate Collaborations

stcewgenericjava.exe

This executable component, **stcewgenericjava.exe**, is the core of the e*Way that communicates and manipulates Events traveling between an external system and e*Gate, and loads and interprets the configuration file used by the e*Way to determine how to deal with data to and from the external system. Communication between the external system is implemented by methods as specified by the **Exchanger Java** Interface.

stcewgenericjava.def

The configuration definition file, **stcewgenericjava.def**, contains all the configuration parameters used by the e*Way executable. Some of these parameters form the basic characteristics for the e*Way itself, while others specify the Java code that allows the e*Way to communicate with a specific external system. The remaining parameters control specific characteristics of the Java Virtual Machine (JVM). These configuration parameters are set using the e*Way Editor.

Exchanger.java and Java Template Methods

e*Ways call Java methods to perform such basic operations as startup, data exchange, positive and negative acknowledgment, and establish and shut down the connection to the external system. The Java Generic e*Way kit includes a sample that illustrates the required input and return values for each basic function. For example, the **exhangeData()** method that reads data from a file will be different from a function written to obtain that data from a database.

3.2 e*Way Extensions and External Applications

The following diagram illustrates how the Java Generic e*Way accesses an external application.

Figure 1 Extending the Java Generic e*Way

Extending the Java Generic e*Way to an External Application
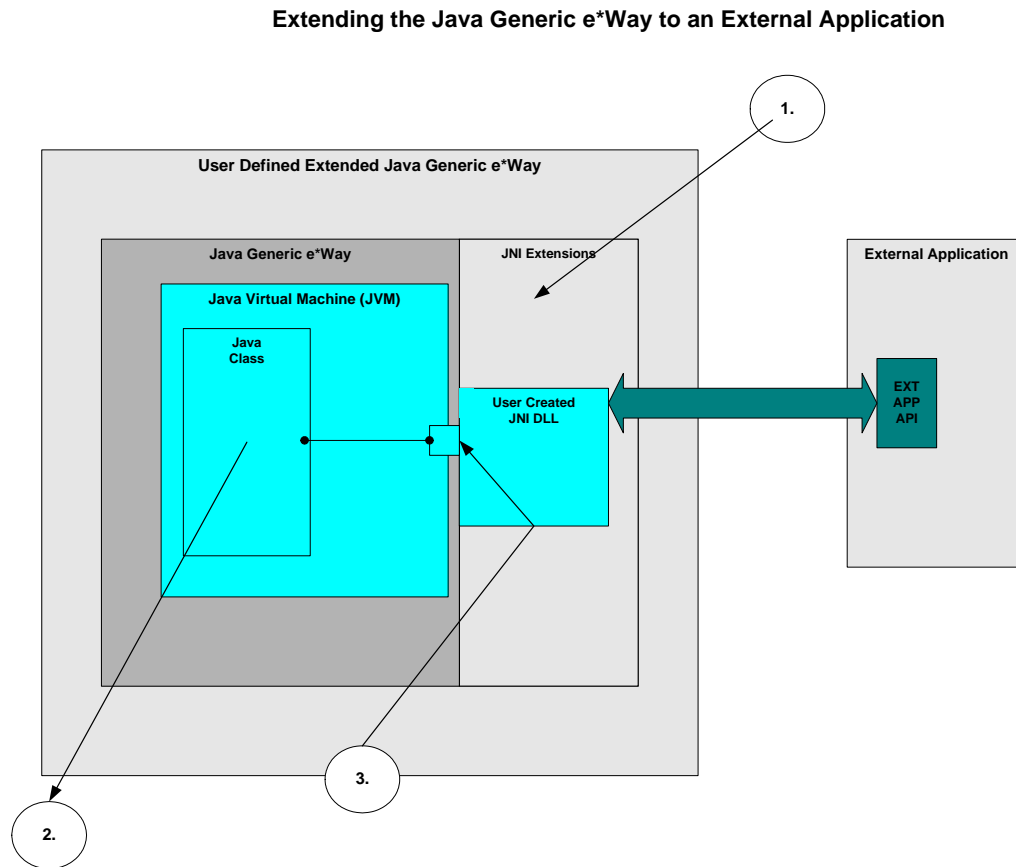
1 The Java Native Interface (JNI) *dynamic link libraries* (on Windows) or *shared libraries* (on UNIX) are created from user-created C or C++ source code to extend the standard Java capability.

2 The Java Generic e*Way is configured to use the Java class implemented by the Exchanger Interface.

3 A user-written Java class uses the JNI and the user-created library to access the external application.

3.2.1. Basics Steps to Extend a Java Generic e*Way

To extend the Java Generic e*Way for access to an external application, follow these basic steps:

1 If necessary, create a JNI dynamic link library or shared library for the Java Generic e*Way to use at run-time to access the external application. To do this, create source code in C or C++ using JNI to "wrap" the external application's API calls; then, compile and link the source code to create the dynamic or shared library.

2 Modify the **stcewgenericjava.def** file template as needed to allow proper configuring of the Java Generic e*Way with the Configuration Window. If you do modify the file template, you must import the changed template to the appropriate schema.

3 Write Java Exchanger Interface class methods that invoke the JNI "wrapped" external application API calls.

4 Run the extended Java Generic e*Way in your e*Gate environment.

Extending the .def File

This chapter describes how to extend the **.def** file and discusses the **.def** file keywords and their arguments. In addition, it discusses how to test and debug the **.def** file and lists some of the common error messages. It also provides information on configuration parameters and the **.cfg** file.

4.1 Introduction

The Java Generic e*Way is configured using the e*Way Editor. It enables you to change configuration parameters quickly and easily. A definition file (**.def**) configures the e*Way Editor to gather those parameters by specifying the name and type of each parameter, as well as other information (such as the range of permissible options for a given parameter).

The e*Way Editor stores the values that you assign to those parameters within two configuration files, the **.cfg** file and the **.sc** file. The configuration files contain similar information but are formatted differently. The **.cfg** file contains the parameter values in delimited records and is parsed by the e*Way at run time. The **.sc** file contains the parameter values and additional information. The e*Way Editor loads the **.sc** file—not the **.cfg** file— when you edit the configuration settings for an e*Way. Both configuration files are generated automatically by the e*Way Editor whenever the configuration settings are saved.

The **.def** file for the Java Generic e*Way contains a set of parameters that are required and may not be modified. You can extend the **.def** file if your modifications to the Java Generic e*Way require the definition of user-defined parameters. This chapter discusses the structure of the **.def** and the configuration files and the syntax of the keywords used to configure the e*Way Editor to gather the desired configuration parameters. The e*Way Editor itself is discussed elsewhere; for more information, see the *e*Gate Integrator User's Guide* or the e*Way Editor's online Help.

Important

We strongly recommend that you make no changes whatsoever to the default **stcewgenericjava.def** file. However, you should use that file as a base from which you create your extensions. Save a copy of the file under a unique name and make your changes to the copy.

4.1.1. Layout

The **.def** file has three major divisions:

- The **header** describes basic information about the file itself, such as version number, modification history, and comments.

- The **sub-header** contains several read-only variables that are for internal use only and must not be modified from their default values.

- The **body** contains configuration parameters grouped into sections. Three sections (General Settings, Communications Parameters, and Java VM Configuration) must be included in all Java Generic e*Way **.def** files; additional sections can be added as needed to support user-created methods.

4.2 .def file Keywords: General Information

All keywords and their arguments are enclosed in balanced parenthesis. Keyword arguments can be a quoted string, a quoted character, an integer, a parenthesis-bounded list, a keyword modifier, or additional keywords.

Examples:

```
(name "TCP Port Number" )

(eway-type
    (direction "<ANY">)
)

(set
    (value (1 2 3))
    (config-default (1 2 3))
)

(range
    (value (const 1 const 1024)
)
```

4.2.1. White Space

White space that is not contained within double-quotation marks, including tabs and newlines, is ignored except as a separator between keywords.

For example, the following are equivalent:

- (user-comment (value "") (config-default ""))

- (user-comment
    (value "")
    (config-default "")
)

Whitespace within quotation marks is interpreted literally. For example, **(name "Extra   Spaces")** will display as:

```
    Extra        Spaces
```

in the e*Way Editor's list of names.

4.2.2. Integer Parameters

The maximum value for integer parameters ranges from approximately -2 billion to 2 billion (specifically, -2,147,483,648 to 2,147,483,647). Most ranges will be smaller, such as "1 to 10" or "1 to 1,000."

4.2.3. Floating-point Parameters

Floating-point parameters and floating-point arithmetic are not supported.

4.2.4. String and Character Parameters

String and character parameters may contain all 255 ASCII characters. The "extended" characters are entered using an escaped format:

- Characters such as tab, newline, and carriage return can be entered as **\t**, **\n**, and **\c**, respectively.
- Characters may also be entered in octal or hexadecimal format using **\o** or **\x**, respectively (for example, \x020 for ASCII character 32).

Strings are delimited by double quotes, characters by single quotes. Examples:

- Strings: `"abc" "Administrator"`
- Characters: `'0' '\n'`

Single quotation marks, double-quotation marks, and backslashes that are not used as delimiters (for example, when used within the text of a description) must be escaped with a backslash, as shown respectively below:

- \'
- \"
- \\

4.2.5. Path Parameters

Path parameters can contain the same characters as other string parameters; however, the characters entered should be valid for pathnames within the operating system on which the e*Way runs.

Backslashes in DOS pathnames must be escaped (as in **c:\\home\\egate**).

4.2.6. Comments

Comments within the **.def** file begin with a semi-colon (;). Any semi-colon that appears in column 1, or that is preceded by at least one space character and that does not appear within quotation marks, is interpreted as a comment character.

Examples

```
; this is a valid comment, because it begins in column 1
(name "Section name") ; this is also a valid comment, because it is
separated by a space
```

4.2.7. "Header" Information

"Header" information that developers may use to maintain a revision history for the **.def** file is stored within the **(general-info)** section. All the information in this section is maintained by the user; no e*Gate product modifies this information.

Table 2 describes the user-editable parameters in the **(general-info)** section. The use of these fields is not required and they may be left blank, but all the fields must be present. The format and contents of these fields is completely at the developer's discretion, as long as rules for escaped characters are observed (see **"String and Character Parameters" on page 24** for more information). Any **(general-info)** parameters that are not shown in the table below are reserved and should not be modified except by direction of Oracle support personnel.

Table 2   User-editable **(general-info)** parameters

Parameter name	Describes
version	The version number
revision	The revision number
user	The user who last edited the file
modified	The modification date
creation	The creation date
description	A description for this .def file, displayed within the e*Way Editor from the File menu's **Tips** option. Quotation marks within the description must be escaped (\").
user-comment	Comments left by the user (rather than the developer), accessed within the e*Way Editor from the File menu's **User Notes** option. Unless you wish to provide a default set of "user notes," we recommend you leave this field blank.

4.3   Defining a New Section

The **(section)** keyword defines a section within the **.def** file. The syntax of the new section is described immediately below. Each section requires at least one parameter; see **"Parameter Syntax" on page 27** for more information on defining parameters.

> *Note:* *Section names and parameter names within a section must be unique.*

4.3.1. Section Syntax

Sections within the **.def** file have the following syntax:

```
(section
    (name "section name")

... at least one parameter definition ...

(description "description text")
(user-comment
    (value "")
    (config-default "")
) ; end of user comment
) ; end of section
```

The section name, description text, and user-comment "value" will appear in the e*Way Editor, as shown in Figure 2.

Figure 2   e*Way Editor main controls
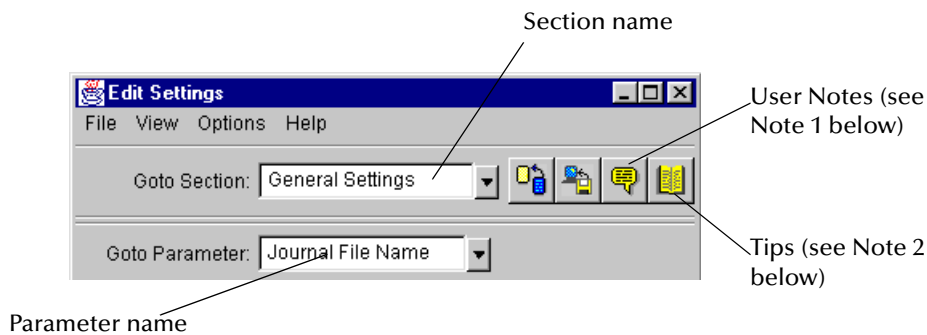
Notes

1   The user-comment feature enables users to make notes about a section or parameter that will be stored along with the configuration settings and save those notes along with the configuration settings. Under most circumstances, we recommend that developers leave the **User Notes** field blank, but you can enter information in the **User Notes** field if you want to ensure that all user notes for a given section begin with preset information.

2   The description is displayed when the user clicks the **Tips** button. Use this field to create online Help for a section or parameter. We recommend that you provide a description for every section and every parameter that you create.

4.3.2. Parameter Syntax

Parameters within the **.def** file use the following basic structure:

```
(param-keyword
    (name "Parameter name goes here")
    (value val)
    (config-default val)

    ...additional keywords(range, units, set)as required...

    (description "description text")
    (user-comment
      (value "")
      (config-default "")
    )
)  ; end of parameter definition
```

The keywords that are invariably required to define a parameter are

- A parameter keyword, discussed below

- The parameter's name: **(name)**

- The initial default value: **(value)**

- The "configuration default": **(config-default)**, which the user can restore by clicking

  . This value can be overridden by the **config-default** keyword specified within a **(set)** command; see **"Parameters Accepting a Single Value From a Set" on page 29** and **"Parameters Accepting Multiple Values From a Set" on page 30** for more information.

Note: *The **(value)** keyword is **always** followed immediately by the **(config-default)** keyword.*

- The "description" (see the Notes for **"Section Syntax" on page 26** for additional information)

- The "user comment" (see the Notes for **"Section Syntax" on page 26** for additional information), which has its own value and configuration default.

Additional keywords may be required, based upon the parameter keyword and user requirements; these will be discussed in later sections.

Order of Keywords

Keywords must appear in this order:

1 parameter definition*

2 name*

3 value*

4 config-default*

5 set

6 range

7 units

8 show-as

9 factor

10 description*

11 user-comment*

Note: *Keywords marked with * are mandatory for all parameters. The **set** keyword is mandatory for **-set** and **-set-multi** parameters. The remaining keywords (items 6 through 9) are optional and depending on developer requirements may appear in any combination, but they must appear in the above order.*

Parameter Types

There are eight types of parameters. The table below lists the types of parameters that can be defined, the keyword required to define them, and the values that the keyword can accept for the **(value)** and **(config-default)** keywords.

Table 3   Basic parameter keywords

Type	Parameter keyword	Values Accepted	Example
Integer	int	integer	7500
Character	char	single-quoted character	'a' '!' '\o123' (octal)
String	string	double-quoted string	"Hello, world"
Date	date	comma-delimited date in *MMM,dd,yyyy* format	AUG,13,2000
Time	time	colon-delimited time in 24-hour *hh:mm:ss* format	15:30:00
Path	path	path; DOS pathnames should use escaped backslashes	/home/egate/client (UNIX) c:\\home\\egate\\client (DOS)
Schedule	schedule	schedule	See **"Schedule Syntax" on page 41**

Parameters Requiring Single Values

Parameters requiring single values are defined within the basic structure shown in **"Parameter Syntax" on page 27**.

Figure 3   A parameter requiring a single value
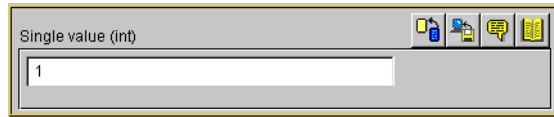
The parameter is defined using a parameter keyword, as listed in **Table 3 on page 28**.

Example

To create a parameter that accepts a single integer as input, and to specify "3" as the default and configuration-default value, enter the following:

```
(int
    (name "Parameter requiring a single integer")
    (value 3)
    (config-default 3)
    (description "This parameter requires a single integer as
      input.")
    (user-comment
      (value "")
      (config-default "")
    )
)  ;  end of parameter definition
```

If you want to limit the values that the user may enter, you may include the optional **(range)** keyword; see **"Specifying Ranges" on page 32** for more information.

Parameters Accepting a Single Value From a Set

Adding the suffix **-set** to the basic parameter keyword (**int-set**, **string-set**, **path-set**, and so on) defines a parameter that accepts one of a given list of values.

Figure 4   A parameter requiring one of a set of values
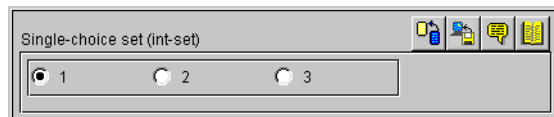
Sets require modifications to the basic parameter syntax (shown in **"Parameter Syntax" on page 27**):

- An additional required keyword, **(set)**, defines the elements of the set.

- Within the **(set)** keyword, **(value)** and **(config-default)** require arguments within parenthesis-bound lists, as in the following:

  ```
  (value (1 2 3))
  (config-default (1 2 3))
  ```

- To prevent a user from to adding or removing choices from the list you provide, add the **const** keyword to the "value" declaration:

  ```
  (value const (1 2 3))
  (config-default (1 2 3))
  ```

- To specify an empty set, enter the keyword **none**, as below:

```
(value none)
(config-default none)
```

"-set-multi" keywords use a different syntax to define an empty set; see **"Parameters Accepting Multiple Values From a Set" on page 30** for more information.

Other important considerations:

- The value specified as the initial **(value)** for the parameter must match at least one of the values specified for **(config-default)** within the **(set)** keyword.

- The initial value within the **(set)** keyword's **(config-default)** list must be within the **(set)** keyword's **(value)** list. However, we strongly recommend that you simply make the two lists identical.

Example

To create a parameter that accepts one of a fixed set of integers (like the one shown in Figure 4 above), enter the following:

```
(int-set
    (name "Single-choice set (int-set)")
    (value 1)
    (config-default 1)
    (set
      (value const (1 2 3))
      (config-default (1 2 3))
    )
    (description "Provides a single choice from a list of integers.")
    (user-comment
      (value "")
      (config-default "")
    )
)  ; end of int-set
```

Note: *The values specified by the **(set)** keyword must be within any values specified by the **(range)** keyword. See* **"Specifying Ranges" on page 32** *for more information.*

Parameters Accepting Multiple Values From a Set

Adding the suffix **-set-multi** to the basic parameter keyword (**int-set-multi**, **string-set-multi**, **path-set-multi**, and so on) defines a parameter that accepts one or more options from a given list of values.

Figure 5 A parameter requiring one of a set of values
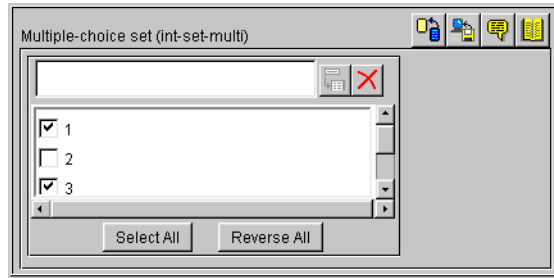
Sets require modifications to the basic parameter syntax (shown in **"Parameter Syntax" on page 27**):

- An additional required keyword, **(set)**, defines the elements of the set.

- Within the **(set)** keyword, **(value)** and **(config-default)** require arguments within parenthesis-bound lists, as in the following:

```
(value (1 2 3))
(config-default (1 2 3))
```

- To prevent a user from to adding or removing choices from the list you provide, add the **const** keyword to the "value" declaration:

```
(value const (1 2 3))
(config-default (1 2 3))
```

- To specify an empty set, enter an empty pair of parentheses "()", as below:

```
(value () )
(config-default () )
```

"-set" keywords use a different syntax to define an empty set; see **"Parameters Accepting a Single Value From a Set" on page 29** for more information.

Other important considerations:

- The value specified as the initial **(value)** for the parameter must match at least one of the values specified for **(config-default)** within the **(set)** keyword.

- The initial value within the **(set)** keyword's **(config-default)** list must be within the **(set)** keyword's **(value)** list. However, we strongly recommend that you simply make the two lists identical.

Examples

To create a parameter that accepts one of a fixed set of integers (like the one shown in Figure 5 above), enter the following:

```
(int-set-multi
    (name "Multiple-choice set (int-set-multi)")
    (value (1 3))
    (config-default (1 3))
    (set
      (value (1 2 3 4 5))
      (config-default (1 2 3 4 5))
    )
    (description "Integer with a modifiable multiple-option set")
    (user-comment
      (value "")
      (config-default "")
    )
)  ; end of int-set-multi
```

Note: *The order in which keywords appear is very important. See* **"Order of Keywords" on page 27** *for more information.*

4.3.3. Specifying Ranges

The **(range)** keyword enables you to limit the range of options that the user may input as a parameter value for **int** and **char** parameters. You may specify a fixed range, or allow the user to modify the upper limit, the lower limit, or both limits. Range limits are inclusive. The values you specify as limits indicate the lowest or highest acceptable value.

The syntax of (range) is as follows:

```
(range
    (value ([const] lower-limit [const] upper-limit))
    (config-default (lower-limit upper-limit))
)
```

The optional **const** keyword specifies that the limit is fixed; if the keyword is omitted, the limit can be modified by the user. The **const** keyword must precede each limit if both limits are to be fixed.

Example

This example illustrates how to define a parameter that accepts an integer as input and limits the range of legal values from zero to ten.

```
(int
    (name "Single integer with fixed range")
    (value 5)
    (config-default 5)
    (range
      (value (const 0 const 10))
      (config-default (0 10))
    )
    (description "Accepts a single integer, limited to a fixed
     range.")
    (user-comment
      (value "")
      (config-default "")
    )
) ; end of int parameter
```

You may also use **(range)** to specify a character range; for example, a range of "A to Z" would limit input to uppercase letters, and a range of "! to ~" limits input to the standard printable ASCII character set (excluding space).

Note: *You may also specify ranges for **-set** and **-set-multi** parameters (**int-set**, **char-set**, and so on).*

4.3.4. Specifying Units

The **(units)** keyword enables **int** parameters to accept input and display the list of available options in different units, provided that each unit is an integer multiple of a base unit.

Figure 6   A parameter that performs unit conversion
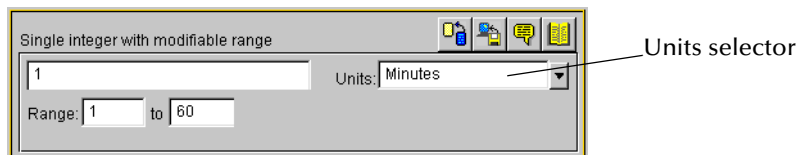
Acceptable groups of units include:

- Seconds, minutes, hours, days

- Bytes, kilobytes, megabytes

Unit conversions that require floating-point arithmetic are not supported.

The syntax of the **(units)** keyword is

```
(units
    ("base-unit":1 "first-unit":a "second-unit":b ... "nth-unit":n)
        (value "default-unit")
        (config-default "default-unit")
)
```

where *a*, *b*, and *n* are the numbers by which the base unit size should be multiplied to perform the conversion to the respective units. The base unit should normally have a value of 1, as shown above; while the e*Way Editor will permit other values, it is highly unlikely that an application would require any other number. The units themselves have no meaning to the e*Way Editor other than the relationships you define (in other words, the Editor does not identify or process "seconds" or other common units as such).

Example

To specify a set of time units (seconds, minutes, hours, and days), enter the following:

```
(units
    ("Seconds":1 "Minutes":60 "Hours":3600 "Days":86400)
        (value "Seconds")
        (config-default "Seconds")
)
```

Units, Default Values, and Ranges

Any time you use the **(units)** keyword within a parameter, you must make sure that the default values can be expressed as integer values of *each* unit. Observing this principle prevents end users from receiving error messages when changing e*Way Editor values in a specific order. For example, if you specified the time units in the example above, but assigned the parameter a default value of "65 seconds," any user who selects the minutes unit *without changing the default value* will receive an error message, because the e*Way Editor cannot convert 65 seconds to an integral number of minutes. Ranges, however, will be rounded to the nearest integer.

Note: *No matter what default value you specify, a user will always see an error message if an inconvertible value is entered and the unit selector is changed. We recommend that you design your parameters so that error messages are not displayed when* ***default*** *values are entered.*

Example

To define a time parameter that displays values in seconds or minutes, with a default of 120 seconds and a fixed range of 60 to 3600 seconds (1 minute to 60 minutes), enter the following:

```
(int
    (name "Single integer with fixed range")
    (value 120)
    (config-default 120)
    (range
      (value (const 60 const 3600))
      (config-default (60 3600))
    )
    (units
      ("Seconds":1 "Minutes":60)
      (value "Seconds")
      (config-default "Seconds")
    )
    (description "Accepts a value between 1 and 60 minutes, with a
        default units value in seconds.")
    (user-comment
      (value "")
      (config-default "")
    )
```

```
) ; end parameter
```

Note: *The order in which keywords appear is very important. See* **"Order of Keywords"**
on page 27 *for more information.*

4.3.5. Displaying Options in ASCII, Octal, Hex, or Decimal

The **(show-as)** keyword enables you to create **int** or **char** parameters that a user can
display in ASCII, octal, hexadecimal, or decimal formats.

The syntax of the **(show-as)** keyword is

```
(show-as
    (format-keyword1 [format-keyword2 ... format-keywordn])
    (value format-keyword)
    (config-default format-keyword)
)
```

where *format-keyword* is one of the following:

- ascii

- octal

- hex

- decimal

Format keywords are case-insensitive, and may be used in any combination and in any
order.

Be sure that any default values you specify for a parameter that uses **(show-as)** can be
represented in each of the **(show-as)** formats. For example, if you are using **(show-as)** to
show an integer parameter in both decimal and hex formats, the default value must be
non-negative.

Example

To create a parameter that accepts a single character in the character-code range
between 32 and 127 and that can display the character value in ASCII, hex, or octal,
enter the following:

```
(char
    (name "A single ASCII character")
    (value '\o100')
    (config-default '\o100')
    (range
      (value (const '\o040' const '\o177'))
      (config-default ('\o040' '\o177'))
    )
    (show-as
      (Ascii Octal Hex)
      (value Octal)
      (config-default Octal)
    )
    (description "Accepts a single character between ASCII 32 and
     ASCII 127.")
    (user-comment
      (value "")
      (config-default "")
```

```
    )
) ; end char parameter
```

Note: *The order in which keywords appear is very important. See* **"Order of Keywords"**
on page 27 *for more information.*

Factor

The **(factor)** keyword enables users to enter an arithmetic operator (+, –, *, or /) as part
of an **int** parameter; for example, to indicate that a value should increase by five units,
the user would enter the integer "5" and the factor "+".

Figure 7 A parameter using **(factor)**
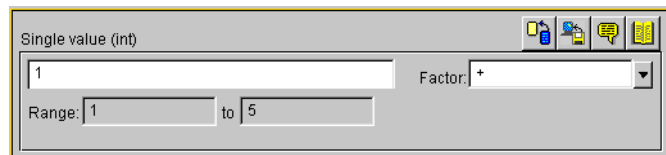
The syntax of the **(factor)** keyword is

```
(factor
    ('operator1' ['operator2'... 'operatorN'])
    (value 'operator'
    (config-default 'operator')
)
```

where *operator* is one of the four arithmetic operators +, –, *, or / (forward slash).

Example

To define a parameter that accepts an integer between 1 and 5 with a factor of + or – (as
in Figure 7 above), enter the following:

```
(int
    (name "Integer with factor")
    (value 1)
    (config-default 1)
    (range
        (value (const 1 const 5))
        (config-default (1 5))
    )
    (factor
        ('+' '-')
        (value '+')
        (config-default '+')
    )
    (description "Enter an integer between 1 and 5 and a factor of +
     or -.")
    (user-comment
        (value "")
        (config-default "")
    )
); end int parameter
```

Note: *The* **(factor)** *keyword must be the final keyword before the* **(description)** *keyword.*
See **"Order of Keywords" on page 27** *for more information.*

Encrypting Strings

Encrypted strings (such as for passwords) are stored in string parameters; to specify encryption, use the **encrypt** keyword, as in the following:

```
(string encrypt
    ...additional keywords follow...
```

The e*Way Editor uses the parameter that immediately precedes the encrypted parameter as its encryption key; therefore, be sure that the parameter that prompts for the encrypted data is not the first parameter in a section. The easiest way to accomplish this is to define a "username" parameter that immediately precedes the encrypted "password" parameter. If you need to specify an encryption key other than the user name, you must define a separate parameter for this purpose.

Text entered into an encrypted-string parameter is displayed as asterisks ("***").

Example

To create a password parameter, enter the following *immediately following* the parameter definition for the corresponding user name (not shown):

```
(string encrypt
    (name "Password")
    (value "")
    (config-default "")
    (description "The e*Way Editor will encrypt this value.")
    (user-comment
      (value "")
      (config-default "")
    )
)
```

Note: *The* **encrypt** *keyword can only follow the* **string** *keyword. The only parameter type that can be encrypted is* **string**; *integer, character, path, time, date, or schedule parameters cannot be encrypted.*

4.4   Configuration Keyword Reference

Table 4 lists the keywords that may appear in the **.def** file.

Table 4   .def-file keywords

Keyword	Purpose	For more information, see this section
app-protocol	Reserved; do not change from the default "<ANY>".	
cfg-icon	Reserved; do not change from the default "" (null string).	
char	Declares a character parameter	**"Parameter Types" on page 28**
char-set	Declares a set of characters, one of which must be selected (via radio button)	**"Parameters Accepting a Single Value From a Set" on page 29**

Table 4   .def-file keywords

Keyword	Purpose	For more information, see this section
char-set-multi	Declares a set of characters, any of which may be selected (via check boxes)	**"Parameters Accepting Multiple Values From a Set" on page 30**
config-default	Specifies the values that will be restored when the user clicks the e*Way Editor's ⬛ button	**"Parameter Syntax" on page 27**
const	Specifies that a value cannot be changed by the user	**"Specifying Ranges" on page 32**
creation	Records creation date or other information.	**""Header" Information" on page 25**
date	Declares a date parameter	**"Parameter Types" on page 28**
date-set	Declares a set of dates, one of which must be selected (via radio button)	**"Parameters Accepting a Single Value From a Set" on page 29**
date-set-multi	Declares a set of dates, one of which must be selected (via radio button)	**"Parameters Accepting Multiple Values From a Set" on page 30**
delim1	Defines the line-separator delimiter used within **.cfg** files. We recommend that you do not modify this value.	
delim2	Defines the parameter-name delimiter used within **.cfg** files. We recommend that you do not modify this value.	
delim3	Defines the value-separating delimiter used within **.cfg** files. We recommend that you do not modify this value.	
delim4	Defines the list-item-separating delimiter used within **.cfg** files. We recommend that you do not modify this value.	
description	A description for the entry (displayed using the e*Way Editor's 🗨 button	**"Notes" on page 26**
direction	Reserved; do not change from the default "<ANY>".	
encrypt	Encrypts a string, such as for passwords. Valid only after the **string** keyword.	**"Encrypting Strings" on page 37**
factor	Defines an arithmetic operator to be associated with an integer parameter	**"Factor" on page 36**

Table 4   .def-file keywords

Keyword	Purpose	For more information, see this section
general-info	Defines the "general information" division of the **.def** file	**""Header" Information" on page 25**
generated-cfg-path	Specifies the path in which the **.cfg** file will be stored. We recommend that you do not modify this field.	
int	Declares an integer parameter	**"Parameter Types" on page 28**
int-set	Declares a set of integers, one of which must be selected (via radio button)	**"Parameters Accepting a Single Value From a Set" on page 29**
int-set-multi	Declares a set of integers, any of which may be selected (via check boxes)	**"Parameters Accepting Multiple Values From a Set" on page 30**
modified	Records modification date or other information	**""Header" Information" on page 25**
name	Specifies the name of a parameter or a section	**"Parameter Syntax" on page 27**
network-protocol	Reserved; do not change from the default "<ANY>".	
os-platform	Reserved; do not change from the default "<ANY>".	
path	Declares a path parameter	**"Parameter Types" on page 28**
path-set	Declares a set of paths, one of which must be selected (via radio button)	**"Parameters Accepting a Single Value From a Set" on page 29**
path-set-multi	Declares a set of paths, any of which may be selected (via check boxes)	**"Parameters Accepting Multiple Values From a Set" on page 30**
protocol-api-version	Reserved; do not change from the default "<ANY>".	
range	Specifies a range of values that represent the upper and lower limits of acceptable user input	**"Specifying Ranges" on page 32**
revision	Records revision numbering or other information (entered manually by the developer)	**""Header" Information" on page 25**
schedule	Declares a schedule parameter	**"Parameter Types" on page 28** and **"Schedule Syntax" on page 41**

Table 4   .def-file keywords

Keyword	Purpose	For more information, see this section
schedule-set	Declares a set of schedules, one of which must be selected (via radio button)	**"Parameters Accepting a Single Value From a Set" on page 29** and **"Schedule Syntax" on page 41**
schedule-set-multi	Declares a set of schedules, any of which may be selected (via check boxes)	**"Parameters Accepting Multiple Values From a Set" on page 30** and **"Schedule Syntax" on page 41**
section	Defines a "section" of the **.def** file	See **"Section Syntax" on page 26**
set	Defines the elements in a set	**"Parameters Accepting a Single Value From a Set" on page 29** and **"Parameters Accepting Multiple Values From a Set" on page 30**
show-as	Selects the format in which character or integer parameters will be displayed	**"Displaying Options in ASCII, Octal, Hex, or Decimal" on page 35**
string	Declares a string parameter	**"Parameter Types" on page 28**
string-set	Declares a set of strings, one of which must be selected (via radio button)	**"Parameters Accepting a Single Value From a Set" on page 29**
string-set-multi	Declares a set of strings, any of which may be selected (via check boxes)	**"Parameters Accepting Multiple Values From a Set" on page 30**
super-client-type	Reserved; do not change from the default "<ANY>".	
time	Declares a time parameter	**"Parameter Types" on page 28**
time-set	Declares a set of times, one of which must be selected (via radio button)	**"Parameters Accepting a Single Value From a Set" on page 29**
time-set-multi	Declares a set of times, any of which may be selected (via check boxes)	**"Parameters Accepting Multiple Values From a Set" on page 30**
units	Determines in which units a parameter will be displayed	**"Specifying Units" on page 33**
user	Records the name of the user who last edited the file (entered manually by the developer)	**""Header" Information" on page 25**

Table 4   .def-file keywords

Keyword	Purpose	For more information, see this section
user-comment	Records a general comment to be applied to the file (accessible via the e*Way editor)	**"Notes" on page 26**
value	Defines the initial value for a parameter	**"Parameter Syntax" on page 27**
version	Records the name of the user who last edited the file (entered manually by the developer)	**""Header" Information" on page 25**

4.4.1. Schedule Syntax

Schedules can be time-based (as in "every ten minutes" or "every hour"), or calendar-based (for a daily, weekly, monthly, or yearly schedule). The syntax for specifying schedules as values and configuration defaults appears in the table below (all times are specified in 24-hour format):

Table 5   Schedule syntax
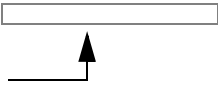
For this schedule…	…use this syntax	Example
Every *s* seconds	*s* (s=seconds)	1800 (every 1800 seconds, or every 30 minutes)
Number of seconds after the minute	:::::*s* (s=seconds)	:::::10 (every ten seconds after the minute)
Number of minutes and seconds past the hour	::::*m*:*s* (m=minutes; s=seconds)	::::15:00 (every fifteen minutes and zero seconds after the hour)
Daily at *time*	:::*hh*:*mm*:*ss*	:::12:00:00 (daily at noon)
Weekly at *day-of-week* at *time*	::*DD*:*hh*:*mm*:*ss* (DD=day of week)	::Su:12:00:00 (weekly, Sundays at noon)
Monthly, every *nth day-of-week* at *time*	::*DDn*:*hh*:*mm*:*ss* (DD=day of week; n=1, 2, 3, 4, or 5)	::Su1:12:00:00 (monthly, the first Sunday, at noon)
Monthly, every *nth* day at *time*	::*n*:*hh*:*mm*:*ss* (n=day of month)	::3:12:00:00 (monthly, the third day of the month, at noon)
Yearly, at a given *date* at *time*	:*MM*:*dd*:*hh*:*mm*:*ss* (MM=month; dd=day)	:08:13:04:00:00 (every August 13th at 4:00 AM)
Yearly, every *nth* day of *month* at *time*	:*MM*:*DDn*:*hh*:*mm*:*ss* (MM=month; DD=day of week; n=1, 2, 3, 4, or 5)	:05:We3:12:00:00 (yearly, every third Wednesday of May, at noon)

Defining Default Schedules

It is *significantly* simpler to define schedules using the e*Way Editor than it is to create schedule entries manually within the **.def** file, especially for complex schedules. The only reason to define a schedule within a **.def** file is to establish a default schedule. If you want to create a default schedule entry, and do not want to construct the entry manually, use this procedure:

1   Define a schedule parameter with a blank ("") default.

2   Commit the **.def** file to a schema, and use the e*Gate Editor to define an entry for the **Start Exchange Data Schedule** parameter. In this entry, create the schedule that you eventually wish to use as a default. (Don't be concerned if this is not the parameter for which you want to define a default schedule; this is just a temporary file.)

3   Save the configuration as **temp** (do not specify an extension) and exit the e*Way Editor.

4   Pull down the Schema Designer's File menu and select **Edit File**.

5   Use the file-selection controls to open the file **/configs/stcewgenericjava/temp.cfg**.

6   The Notepad editor will launch. Scroll down until you find the "Communications Setup" section; a sample appears below.

```
# ---------------------------------------------------------------
-----------
#   Section:Communication Setup
# ---------------------------------------------------------------
-----------
#                                       [            ]
```
                                                    ↑
                          Schedule definition ──────┘

7   Use "copy and paste" to copy the schedule-definition string (in the figure above, "`:::12:00:00`").

8   Exit the editor; there is no need to save the file.

9   Pull down the Schema Designer's File menu and select **Edit File**.

10 Use the file-selection controls to open the file **/configs/stcewgenericjava/** *your_def_file* (substituting the name of the **.def** file you want to modify).

11 Modify the **(value)** and **(config-default)** keywords within the desired schedule parameter by pasting in the string that you copied in step 7 above.

12 Save the file and commit the modified file to the Registry.

4.5   Configuration Parameters and the Configuration Files

Parameters defined within the **.def** file are stored within two "configuration" files (**.cfg** and **.sc**), which are generated by the e*Way Editor's "Save" command. The following rules apply to both **.cfg** and **.sc** files:

- Keywords are not case sensitive, as they are converted to uppercase internally before matching.

- Comments begin with the "#" character, which must appear in column one (see the example in the section immediately below).

- Unlike the **.def** file, the **.cfg** and **.sc** files are sensitive to white space. White space consists of single space characters, tabs, and newlines. Be careful not to insert extra white space around delimiters or equal signs (for example "|value=3|" is legal, but "|value = 3|" and "| value=3 |" are illegal).

The following rule applies only to the **.cfg** file:

- Each line and each element in the **.cfg** file is separated using delimiters (see **delim1, delim2, delim3, and delim4** in **Table 4 on page 37**). We strongly recommend that you do not modify any of the default delimiters.

Note: *The e*Way Editor will create a **.cfg** and **.sc** file automatically when you save your configuration changes in the e*Way Editor. You should not need to modify either file manually unless directed to do so by Oracle support personnel.*

*Although e*Ways are shipped with default **.def** files, no configuration files are provided, because there is no "standard" configuration for any given e*Way. Users must manually create a configuration profile using the e*Way Editor for every e*Way component.*

Examples

.cfg File

This example is excerpted from the "General Settings" section of a **.cfg** file that is generated by the default **stcewgenericjava.def** file.

```
# -------------------------------------------------------------------
#        Section: General Settings
# -------------------------------------------------------------------
#
General Settings|Journal File Name|value=|set=
General Settings|Max Resends Per Message|value=5|set=5|range=1,1024
General Settings|Max Failed Messages|value=3|set=3|range=1,1024
General Settings|Forward External Errors|value=NO|set=NO,YES
```

.sc File

This example is excerpted from the "General Settings" section of a **.sc** file that is generated by the default **stcewgenericjava.def** file. Notice the amount of additional information as compared to the **.cfg** file example of the same section above.

```
; -----------------------------------------------------------------
;    Section: "General Settings"
; -----------------------------------------------------------------
(section
  (name "General Settings")
  (string-set
    (name "Journal File Name")
    (value "")
    (config-default "")
    (set
      (value (""))
```

```
        (config-default (""))
    )
    (description "
Journal File is used for the following conditions:
- Journal a message when it exceeds the number of retries.
- Journal an external error when it's not configured to
  forward to Egate.

If an absolute path is not specified,  the system data
directory is prepended to the path.
")
    (user-comment
      (value "")
      (config-default "")
    )
  )
  (int-set
    (name "Max Resends Per Message")
    (value 5)
    (config-default 5)
    (set
      (value (5))
      (config-default (5))
    )
    (range
      (value (const 1 const 1024))
      (config-default (1 1024))
    )
    (description "Max Resends Per Message:

This parameter is the maximum number of times the e*Way
will attempt to resend a message to the extenal after
receiving an error.  When this maximum is reached, the
message is considered a failed message and is written to
a journal file.
")
    (user-comment
      (value "")
      (config-default "")
    )
  )
  (int-set
    (name "Max Failed Messages")
    (value 3)
    (config-default 3)
    (set
      (value (3))
      (config-default (3))
    )
    (range
      (value (const 1 const 1024))
      (config-default (1 1024))
    )
    (description "Max Failed Messages:

This parameter is the maximum number of failed messages
the e*Way will allow.  If this many messages fail
and are journaled, the e*Way will shutdown and exit.
")
    (user-comment
      (value "")
      (config-default "")
    )
  )
```

```
    (string-set
      (name "Forward External Errors")
      (value "NO")
      (config-default "NO")
      (set
        (value const ("NO" "YES"))
        (config-default ("NO" "YES"))
      )
      (description "Forward External Errors:

 If this parameter is set to YES then error messages that
 starts with DATAERR received from the external will be
 queued to the configured queue.  If this parameter is set
 to NO then error messages will not be forward.
")
      (user-comment
        (value "")
        (config-default "")
      )
  )
  (description "General Settings:

 This section contains a set of top level parameters:

        o Journal File Name
        o Max Resends Per Message
        o Max Failed Messages
        o Forward External Errors
")
  (user-comment
    (value "")
    (config-default "")
  )
)
```

4.6 Testing and Debugging the .def File

Testing the **.def** file is very straightforward; simply open the file with the e*Way Editor. If the syntax of all parameters is correct, the e*Way Editor will launch, and you can confirm that your sections, parameters, ranges, and options are as you intended.

You may encounter the following error types:

- **Logical errors**: The e*Way Editor will load the **.def** file and will display no error message, but the parameters are not defined as desired (for example, default options are omitted, or a range was not properly defined). These errors are corrected simply by replacing the undesired values with the desired ones.

- **Syntax errors**: These "mechanical" errors involve missing parentheses, invalid keywords and similar problems. These errors will cause the e*Way Editor to display an error message and exit. This section deals primarily with errors of this type.

Note: *You may also encounter syntax errors if you try to edit an existing configuration profile that contains a corrupted **.sc** file. You should not attempt to modify **.sc** or **.cfg** files outside of the e*Way Editor unless specifically instructed to do so by Oracle personnel.*

The e*Way Editor component that interprets the **.def** file provides only elementary error messages when it encounters an error in the **.def** file. This section discusses the most common errors you may encounter, and the steps you should take to debug a **.def** file under development.

By far, the most common errors are:

- **Missing parentheses**: Proper indentation will help you catch most of these, and some editors have features that find matching parentheses (such as the **vi** editor's SHIFT+% function).

- **Missing quotation marks**: Be sure that characters are delimited by single quotes and strings/paths by double quotes.

- **Quotation marks** (that should be escaped but are not): This usually occurs in the argument to the **(description)** keyword; double-check that all quotations within descriptions use \"escaped\" quotation marks.

- **Missing parameters**: Refer to the examples in this chapter, or to the sample **.def** file for the required parameters for each keyword.

- **Keywords (**out of order): See **"Order of Keywords" on page 27**.

Note: *Using the templates provided in the sample* **.def** *file will help prevent many errors before they occur; see* **"Sample .def File" on page 48** *for more information.*

4.6.1. Common Error Messages

The following section contains common error messages and their most common causes. Each error message will contain the string **L<*nnn*>**, which indicates a line number (for example, **L<124>** signifies "line 124").

SCparse : parse error, expecting 'LP_*keyword-name*'
The *keyword-name* was expected but not found. The keyword could be missing or out of order, the keyword's initial parenthesis could be missing, or the previous keyword could have been terminated prematurely (for example, by an out-of-place parenthesis or quote-parenthesis combination) or misspelled.

SCparse : parse error, expecting 'RIGHT_PAREN'
The right parenthesis is missing, a close-quote is missing, as in **(user-comment "   )**, or there is an extra (or unescaped) close-quote within a **(description)** keyword argument.

SCparse : parse error, expecting `LEFT_PAREN'
This error appears under a very wide range of conditions. A keyword could be misspelled, there could be extraneous or unbalanced quotes or parentheses, a keyword could be missing a left parenthesis, or extraneous material may have been found between parameter declarations. Sometimes this error appears in conjunction with **expecting `LP_*keyword-name*'**.

*Param-Type<keyword>***: Value is not within the allowed range.**
An argument to a keyword has exceeded the limits defined by its accompanying **(range)** keyword. Change either the **(value)** argument or the **(range)** limit.

param-typeTypeSet<*keyword*> : "*n*" is not in this Set.
A default value for a parameter has been specified that does not appear within the default value of the **(set)** keyword.

SCparse : parse error, expecting `*arg-type*'
One type of argument was expected, but another has been found (for example, an integer where as string was expected). Errors expecting LITERAL_STRING are commonly caused by missing quotation marks. Errors expecting TIME_VAL, DATE_VAL, or SCHEDULE_VAL can also be due to invalid data (such as a time of 12:00:99), or missing/extra delimiters.

CharVal : "*sequence*" is not legal character.
There is an error in an escape sequence.

SCparse : parse error
This "general" error can be caused by a number of problems, such as misspelled arguments within keywords.

4.7 Accessing Configuration Parameters Within the JVM Environment

The Java Generic e*Way automatically loads configuration parameters stored in the **.cfg** file into a Java **Properties** object within the JVM.

4.7.1. Property-name Format

Property keys are named using the format

> *SECTION-NAME.PARAM-NAME*

where *SECTION-NAME* is the name of the section and *PARAM-NAME* is the name of the parameter. The value of the parameter is stored as the value of the variable.

Variable names are in all upper case, and are case-sensitive. The section and parameter names are separated by a period (.), and any spaces contained within section or parameter names are also converted into underscores.

Examples

The value of the parameter named "Password" within the section "Authentication" would be stored under the property key "AUTHENTICATION.PASSWORD" (all upper case).

The value of the parameter named "Gateway ID" within the section "Connection Parameters" would be stored under:
"CONNECTION_PARAMETERS.GATEWAY_ID".

4.7.2. **Getting Property Values**

Property values are read using the Java method **(EGate.getEwayConfigProp())**. The **(EGate.getEwayConfigProp()** method requires the name of the Property key whose value you wish to retrieve as an argument, and returns a string containing that value or **null** if the specified variable does not exist.

Examples

```
EGate.getEwayConfigProp("AUTHENTICATION.PASSWORD");

EGate.getEwayConfigProp("CONNECTION_PARAMETERS.GATEWAY_ID");
```

4.8 **Sample .def File**

A **.def** file containing commented samples of a wide range of parameter definitions is available on the e*Gate installation CD-ROM:

/samples/genjavaeway/FileExchange.def

Note: *The **stcewgenericjava.def** file does not contain configuration options for any specific e*Way, and cannot be used for that purpose. It merely provides working templates from which you can build your own **.def** file.*

You can use the **FileExchange.def** file as a sample from which you can build your own extensions to your own **.def** file. Simply open the file with a text editor, select the desired parameter-definition template, and "copy and paste" the template into your own **.def** file, where you can modify it as needed.

To open the FileExchange .def file in the e*Way Editor:

1 Using the Schema Designer, commit the **FileExchange.def** file to the directory **/configs/stcewgenericjava/** within any desired schema. We recommend that you do not commit the file to the **default** schema; rather, use a schema reserved for testing and development.

2 Create or select an e*Way, and display its properties. Remember that this e*Way cannot be used to manipulate data; it serves merely as a "placeholder" so you can open the **FileExchange.def** file with the e*Way Editor.

3 On the e*Way property sheet's General tab, under **Executable file**, click **Find**.

4 Select **stcewgenericjava.exe** and click **OK**.

5 Under **Configuration file**, click **New**.

6 From the list of e*Way templates, select **FileExchange**.

When the e*Way Editor launches, you will see several sections of sample parameters (for example, "Single integer with modifiable lower limit," "Single integer with modifiable upper limit," and so on), as shown in the Figure 8.

Figure 8   The **FileExchange.def** file in the e*Way Editor
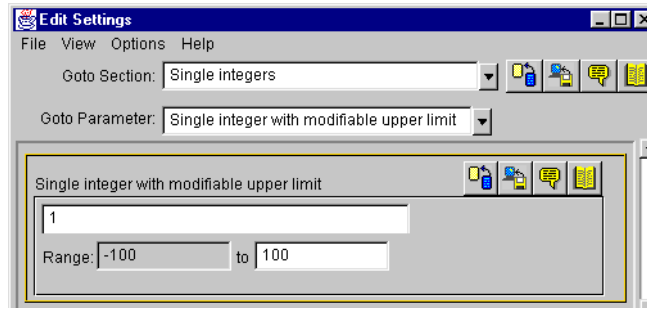
After identifying the parameter you wish to copy, open **sample.def** in a text editor and search for the parameter name. Then, simply copy the parameter and change the sample values to the values you wish to use (as shown in Figure 9 on the next page).

Figure 9   The **sample.def** file in Wordpad
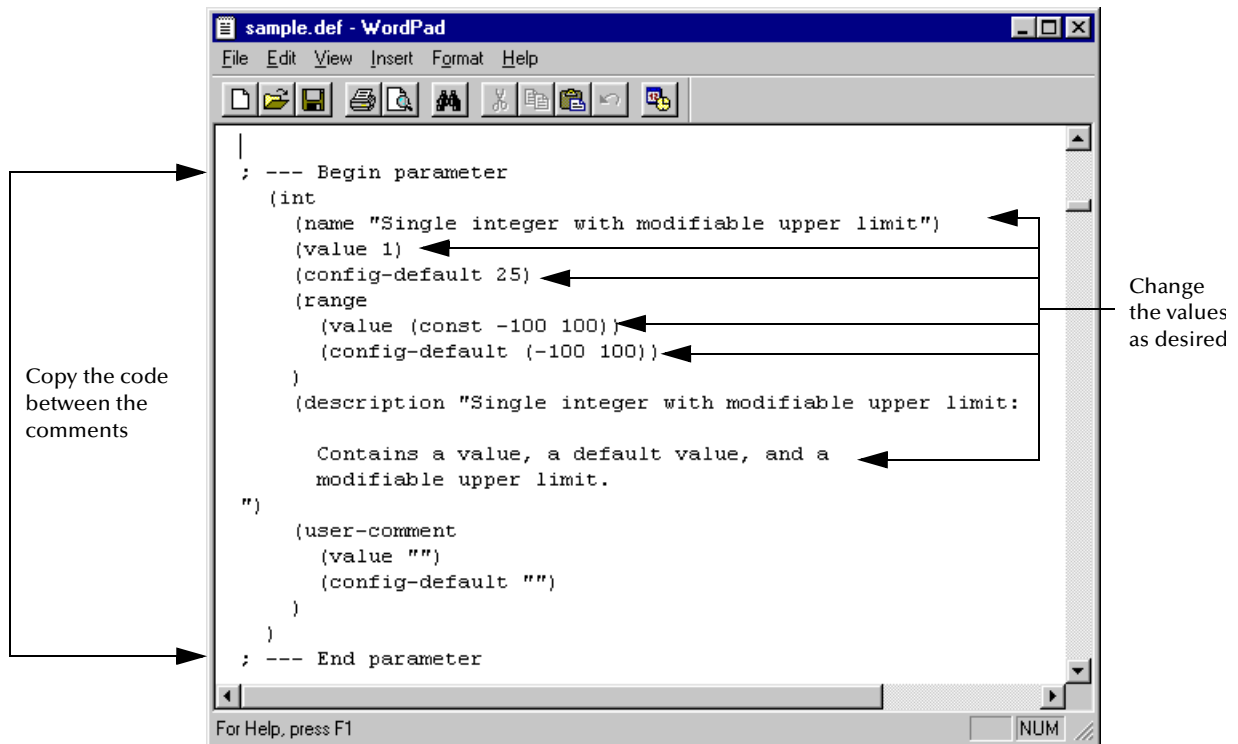
4.8.1. Sample Code for FileExchange.java

The **FileExchange.java** class file illustrates an implementation of the Exchanger interface. The code is commented and straightforward.

```
/**
 *  A sample class to illustrate implementation of the Exchanger interface.
 *  A flat file is considered as the "external" system for both inbound and
 *  outbound processing.
 *
 */
```

```
//  Java specific package imports

import java.io.*;

//  e*Gate specific package imports

import com.stc.common.collabService.*;
import com.stc.common.registry.*;
import com.stc.common.utils.*;

public class FileExchange implements Exchanger
{
  private String              eGateLogsDir    = null;
  private FileOutputStream     outFos          = null;
  private File                 outFile         = null;
  private String               customInputDir  = null;

  //  ----------------------------------------------------------------------

  /**
   *  Zero-argument constructor is needed (Java will provide one if not
   *  defined, but it's better to be explicit).
   *
   */
  public FileExchange()
  {
    super();
  }

  //  ----------------------------------------------------------------------

  /**
   *  This gets called when the Java e*Way initially starts up.  Going to
   *  use it to discover where the e*Gate client logs/ directory is and
   *  to get the user customized configuration parameter "Inbound Directory"
   *  under section "Sample Test".
   *
   *  @exception com.stc.collabService.CollabConnException      thrown if
   *             problem encountered
   *
   */
  public void startUp() throws CollabConnException
  {
    //  Determine from the e*Gate repository where the client logs/
    //  directory is.

    RepositoryDirectories repDir = new RepositoryDirectories();
    if (repDir.readRepositoryDirectories())
    {
      eGateLogsDir = repDir.getLogs();
      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
                    "startUp(): e*Gate client logs/ is at: " + eGateLogsDir);
    }
    else
    {
      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_ERROR,
                    "startUp(): Can't read repository directories");
      throw new CollabConnException("Can't read repository directories");
    }

    //  Get the user customized configuration parameter

    customInputDir = EGate.getEwayConfigProp("SAMPLE_TEST.INBOUND_DIRECTORY");
    if (null == customInputDir)
    {
      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_WARNING,
                    "startUp(): No Inbound Directory defined in Sample Test");
    }
    else
    {
      File inputDir = new File(customInputDir);

      if (!inputDir.isAbsolute())
      {
        inputDir = new File(repDir.getSystemData(), customInputDir);
        customInputDir = inputDir.getAbsolutePath();
      }

      if (!inputDir.isDirectory())
      {
        EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_ERROR,
                      "startUp(): Inbound Directory " +
                      customInputDir + " doesn't exist!");
        throw new CollabConnException(customInputDir + " doesn't exist!");
      }
    }
  }

  //  ----------------------------------------------------------------------

  /**
   *  This gets called when there's an outbound event from e*Gate to the
```

```
         *   external.  We're simply going to write it out to the output file.
         *
         *   @param     outEvent         output event data given as a byte array
         *   @exception com.stc.common.collabService.CollabConnException  thrown if
         *                               problem encountered with a connection
         *   @exception com.stc.common.collabService.CollabDataException  thrown if
         *                               problem encountered with data translation
         *   @exception com.stc.common.collabService.CollabResendException  thrown if
         *                               the outgoing event is to be resent
         *
         */
        public void processOutgoing(byte[] outEvent)
                    throws CollabConnException, CollabDataException,
                           CollabResendException
        {
          if (outFos != null && outEvent.length > 0)
          {
            try
            {
              outFos.write(outEvent);
              outFos.write(System.getProperty("line.separator").getBytes());
              outFos.flush();
              EGate.traceln(EGate.TRACE_EWAY_VERBOSE, EGate.TRACE_EVENT_TRACE,
                            outEvent, "processOutgoing(): wrote to file");
            }
            catch (IOException e)
            {
              EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_ERROR,
                            "processOutgoing(): Can't write to output file: " +
                            e.getMessage());
              throw new CollabConnException(e.getMessage());
            }
          }
        }

        // ---------------------------------------------------------------------

        /**
         *   This gets called when the Java e*Way's exchange data with external
         *   schedule is due.  We're just going to look for an input file in the
         *   user customized configured inbound directory.
         *
         *
         *   @return    a byte array for the data received from the external
         *   @exception com.stc.common.collabService.CollabConnException  thrown if
         *                               problem encountered with a connection
         *   @exception com.stc.common.collabService.CollabDataException  thrown if
         *                               problem encountered with data translation
         *
         */
        public byte[] exchangeData()
                    throws CollabConnException, CollabDataException
        {
          if (customInputDir != null)
          {
            File inputFile = new File(customInputDir, "TestIn.txt");
            long len;

            if (inputFile.exists() && (len = inputFile.length()) > 0)
            {
              FileInputStream fis = null;

              try
              {
                fis = new FileInputStream(inputFile);
                len = fis.available();
                byte[] retBytes = new byte[(int) len];
                fis.read(retBytes);

                EGate.traceln(EGate.TRACE_EWAY_VERBOSE, EGate.TRACE_EVENT_TRACE,
                              retBytes, "exchangeData(): received from file");
                return retBytes;
              }
              catch (Exception e)
              {
                EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_ERROR,
                              "exchangeData(): Input file problems: " +
                              e.getMessage());
                throw new CollabConnException("Input file problems: " +
                                              e.getMessage());
              }
              finally
              {
                if (fis != null)
                {
                  try
                  {
                    fis.close();
                  }
                  catch (Exception e)
                  {
                  }
```

```
          }

          inputFile.renameTo(new File(customInputDir, "TestIn.~xt"));
        }
      }
    }

    return null;
}

// -----------------------------------------------------------------------

/**
 *  This gets called to establish a connection with an external system.
 *  We're simply going to open the output file here.
 *
 *  @return    a boolean <code>true</code> when successfully connected;
 *             otherwise a <code>false</code>
 *
 */
public boolean connectionEstablish()
{
    // Open an output file in the logs/ directory

    try
    {
      outFile = new File(eGateLogsDir, "TestOut.txt");
      outFos = new FileOutputStream(outFile);
      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
        "connectionEstablish(): Successfully opened output file TestOut.txt");
    }
    catch (Exception e)
    {
      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_WARNING,
        "connectionEstablish(): Failed to open output file TestOut.txt: " +
        e.getMessage());

      return false;
    }
    return true;
}

// -----------------------------------------------------------------------

/**
 *  This gets called to verify a connection with the external.  We're
 *  just going to test if the output file exists.
 *
 *  @return    a boolean <code>true</code> when connection is intact;
 *             otherwise a <code>false</code>
 *
 */
public boolean connectionVerify()
{
    EGate.traceln(EGate.TRACE_EWAY_VERBOSE, EGate.TRACE_EVENT_TRACE,
                  "connectionVerify(): test if output file exists?");
    return (outFile != null && outFile.exists());
}

// -----------------------------------------------------------------------

/**
 *  This gets called to shut down a connection with the external.  We're
 *  going to close the output file here.
 *
 *  @param     notif   a notification string "SUSPEND_NOTIFICATION" will be
 *                     passed in to indicate the connection should be
 *                     shut down
 *  @return    a boolean <code>true</code> when connection has been severed;
 *             otherwise a <code>false</code>
 *
 */
public boolean connectionShutdown(String notif)
{
    EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
                  "connectionShutdown(): got request: " + notif);
    if (outFos != null)
    {
      try
      {
        outFos.close();
        outFos = null;
      }
      catch (Exception e)
      {
      }
    }

    return true;
}

// -----------------------------------------------------------------------
```

```
     /**
      *  Gets called to positively acknowledge an external when all the events
      *  received have been processed successfully by all e*Way collaborations.
      *
      *  @param    ackevt  an acknowledgment event to be sent to the external
      *  @exception com.stc.common.collabService.CollabConnException  thrown if
      *                          problem encountered with a connection
      *
      */
     public void ACK(byte[] ackevt)
                  throws CollabConnException
     {
       EGate.traceln(EGate.TRACE_EWAY_VERBOSE, EGate.TRACE_EVENT_TRACE,
                  ackevt, "ACK(): gotten for");
     }

     // -----------------------------------------------------------------------

     /**
      *  Gets called to negatively acknowledge an external when not all the events
      *  received have been processed successfully by all e*Way collaborations.
      *
      *  @param    nakevt  an acknowledgment event to be sent to the external
      *  @exception com.stc.common.collabService.CollabConnException  thrown if
      *                          problem encountered with a connection
      *
      */
     public void NAK(byte[] nakevt)
                  throws CollabConnException
     {
       EGate.traceln(EGate.TRACE_EWAY_VERBOSE, EGate.TRACE_EVENT_WARNING,
                  nakevt, "NAK(): gotten for");
     }

     // -----------------------------------------------------------------------

     /**
      *  Gets called to notify the Java exchange class that the e*Way is shutting
      *  down.
      *
      *  @param    notif   a notification string, "SHUTDOWN_NOTIFICATION", to
      *                    advise that the e*Way is about to shut down.
      *  @return   a boolean <code>true</code> if the shutdown process can
      *            proceed; otherwise a <code>false</code>
      *
      */
     public boolean shutdown(String notif)
     {
       EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
                  "shutdown(): got request: " + notif);
       return true;
     }
}
```

4.8.2. e*Gate Registry Configuration Properties

There are additional configuration properties and utility functions that are pre-defined. The table below lists the available properties.

Note: *You cannot set these parameters. They are read only.*

Property	Description	Example
EGATE_REGISTRY.HOST	Access the e*Gate Registry Host	String regHost = EGate.getEwayConfigProp("EGATE_REGISTRY.HOST");
EGATE_REGISTRY.SCHEMA	Access the e*Gate Registry Schema	String regSchema = Egate.getEwayConfigProp("EGATE_REGISTRY.SCHEMA");

Property	Description	Example
EGATE_REGISTRY.PORT	Access the e*Gate Registry Port	long regPort = Long.parseLong(EGate.getEwayConfigProp("EGATE_REGISTRY_PORT"));
EGATE_REGISTRY.LOGICALNAME	Access the e*Gate Registry logical name	String regLName = EGate.getEwayConfigProp("EGATE_REGISTRY.LOGICALNAME");
EGATE_REGISTRY.USERNAME	Access the e*Gate Registry user name	String regLName = EGate.getEwayConfigProp("EGATE_REGISTRY.USERNAME");
EGATE_REGISTRY.PASSWORD	Access the e*Gate Registry user password	String regUPass = EGate.getEwayConfigProp("EGATE_REGISTRY.PASSWORD");

Note: *EGATE_REGISTRY.PASSWORD is internally encrypted and a special method,*
RegistryControlFile.intraDecrypt(), *needs to be called to decrypt it:*

For example:

```
String clearPass =
RegistryControlFile.intraDecrypt(regUPass);
```

4.8.3. Accessing e*Gate Participating Host Installation Information

In order to access e*Gate's Participating Host installation information:

1 The following package must be imported by the Java code:

```
import com.stc.common.registry.*;
```

2 A RepositoryDirectories object must be instantiated:

```
RepositoryDirectories rpd = new RepositoryDirectories();
```

3 The .egate.store file must be read in by calling:

readRepositoryDirectories():boolean readRepositoryDirectories()

For example:

```
if (rpd.readRepositoryDirectories() ==true) {your code}
```

4 The object methods can be used to access the following:

Method	Description	Example
String getSystemData()	Access the System Data Directory	String systemData = rpd.getSystemData();
String getIqueueIndex	Access the IQ Index Directory	String iqueueIndex = rpd.getIqueueIndex()

Method	Description	Example
String getIqueueData()	Access the IQ Data Directory	String iqueueData = rpd.getIqueueData();
String getLogs()	Access the Logs Directory	String logs = rpd.getLogs()
String getShareExe()	Access the Shared Exe Directory	String sharedExe = rpd.getSharedExe()

4.8.4. Accessing e*Gate Registry Files

In order to access the e*Gate Registry Files:

1 The following package must be imported by the Java code:

```
import com.stc.common.registry
```

2 A connection to the e*Gate registry must be made by:

▪ Instantiating and initializing a CallerID object, for example by:

```
CallerID callerID = new CallerID();
callerID.setLogicalName("MyApp");
callerID.setRegistryHost(EGate.getEwayConfigProp("EGATE_REGISTRY.HOST"));
callerID.setRegistryPort(Long.parseLong(EGate.getEwayConfigProp("EGATE_REGISTRY.PORT")));
callerID.setUserName(EGate.getEwayConfigProp("EGATE_REGISTRY.USERNAME"));
callerID.setPassword(RegistryControlFile.intraDecrypt(EGate.getEwayConfigProp("EGATE_REGISTRY.PASS
WORD")));
```

▪ Acquiring a provider context with the e*Gate Registry, for example by:

```
Registry reg = new Registry(false);
if (reg.acquireProvider(reg, callerID) == true)
{
        your code to access the e*Gate registry
}
```

3 To retrieve a file, such as classes/MyTest.class, from the e*Gate registry, you'll need to do the following:

▪ Instantiate and initialize a FileRef object:

```
FileRef fr = new FileRef();
fr.setDirectory("class");
fr.setFile("MyTest.class");
fr.setFileType(RegistryControlFile.FILETYPE_BINTEXT);
```

▪ Call the Registry method **retrieveFile()** after you've acquired the provider context:

```
String regFile = reg.retrievFile (fr,
Registry.RETRIEVE_BYPASSANDBOX);
```

where **regFile** will be the absolute pathname from which the file was retrieved.

4.8.5. Decoding configuration File Encrypted Passwords

In order to decode configuration file encrypted passwords:

1 The following package must be imported by the Java code:

```
import com.stc.common.utils.*;
```

2   To decrypt the encrypted password, (such as parameter "Password") for user name (such as parameter "User Name") in the "My Test" section in the e*Way configuration file, call the **ScEncrypt.decrypt()** method:

```
String clearPass;
try
{
    clearPass =
ScEncrypt.decrypt(EGate.getEwayConfigProp("MY_TEST.USER_NAME"),EGate.
getEwayConfigProp("MY_TEST.PASSWORD"));
}
catch (Exception e)
{
    can decrypt this password???
}
```

Configuring the Java Generic e*Way

This chapter describes how to set the required e*Way configuration parameters for **stcewgenericjava.def**.

5.1 Required e*Way Configuration Parameters

The e*Way configuration parameters discussed in this section are required by the Java Generic e*Way. The configuration parameters themselves are set using the e*Way Editor.

To change e*Way configuration parameters:

1 In the Schema Designer's Component editor, select the e*Way you want to configure and display its properties.

2 Under **Configuration File**, click **New** to create a new file, **Find** to select an existing configuration file, or **Edit** to edit the currently selected file.

3 In the **Additional Command Line Arguments** box, type any additional command line arguments that the e*Way may require, taking care to insert them *at the end* of the existing command-line string. Be careful not to change any of the default arguments unless you have a specific need to do so.

For more information about how to use the e*Way Editor, see the e*Way Editor's online Help or the *e*Gate Integrator User's Guide*.

The e*Way's configuration parameters are organized into the following sections:

- General Settings
- Communication Setup
- Java VM Configuration

5.1.1. General Settings

The General Settings control basic operational parameters.

Journal File Name

Description

Specifies the name of the journal file.

Required Values

A valid filename, optionally including an absolute path (for example, **c:\temp\filename.txt**). If an absolute path is not specified, the file will be stored in the e*Gate "SystemData" directory. See the *e*Gate Integrator System Administration and Operations Guide* for more information about file locations.

Additional Information

An Event will be journaled for the following conditions:

- When the number of resends is exceeded (see Max Resends Per Message below).

- When its receipt is due to an external error, but Forward External Errors is set to **No**. (See **"Forward External Errors" on page 58** for more information.)

Max Resends Per Message

Description

Specifies the number of times the e*Way will attempt to resend a message (Event) to the external system after receiving an error. When this maximum is reached, the message is considered "Failed" and is written to the journal file.

Required Values

An integer between 1 and 1,024. The default is 5.

Max Failed Messages

Description

Specifies the maximum number of failed messages (Events) that the e*Way will allow. When the specified number of failed messages is reached, the e*Way will shut down and exit.

Required Values

An integer between 1 and 1,024. The default is 3.

Forward External Errors

Description

Selects whether data translation errors indicated by a **CollabDataException** thrown by the **_exchangeData( )_** method will be queued to the e*Way's configured queue. See **exchangeData()** on page 72 for more information.

Required Values

Yes or **No**. The default value, **No**, specifies that error messages will not be forwarded.

5.1.2. Communication Setup

The Communication Setup parameters control the schedule by which the e*Way obtains data from the external system.

Note: *The schedule you set using the e*Way's properties in the Schema Designer controls when the e*Way executable will run. The schedule you set within the parameters discussed in this section (using the e*Way Editor) determines when data will be exchanged. Be sure you set the "exchange data" schedule to fall within the "run the executable" schedule.*

Exchange Data Interval

Description

Specifies the number of seconds the e*Way waits between calls to the **`exchangeData()`** method during scheduled data exchanges.

Required Values

An integer between 0 and 86,400. The default is 120.

Additional Information

If **Zero Wait Between Successful Exchanges** is set to **Yes** and the **`exchangeData()`** method returns data, The **Exchange Data Interval** setting will be ignored and the e*Way will invoke the **`exchangeData()`** method immediately.

If this parameter is set to zero, there will be no exchange data schedule set and the **`exchangeData()`** will never be called.

See **"Down Timeout" on page 60** and **"Stop Exchange Data Schedule" on page 60** for more information about the data-exchange schedule.

Zero Wait Between Successful Exchanges

Description

Selects whether to initiate data exchange after the **Exchange Data Interval** or immediately after a successful previous exchange.

Required Values

Yes or **No**. If this parameter is set to **Yes**, the e*Way will immediately invoke the **exchangeData()** method if the previous exchange method returned data. If this parameter is set to **No**, the e*Way will always wait the number of seconds specified by **Exchange Data Interval** between invocations of the **exchangeData** method. The default is **No**.

See **exchangeData()** for more information.

Start Exchange Data Schedule

Description

Establishes the schedule to invoke the e*Way's **`exchangeData()`** method.

Required Values

One of the following:

- One or more specific dates/times

- A single repeating interval (such as yearly, weekly, monthly, daily, or every *n* seconds).

Also required: If you set a schedule using this parameter, you must also implement all three of the following `com.stc.common.collabService.Exchanger` interface methods:

- **exchangeData() on page 72**

- **ACK() on page 70**

- **NAK() on page 72**

If you do not do so, the e*Way will not start.

Additional Information

When the schedule starts, the e*Way determines whether it is waiting to send an ACK or NAK to the external system (using the `ACK()` and `NACK()` methods) and whether the connection to the external system is active. If no ACK/NAK is pending and the connection is active, the e*Way immediately executes the `exchangeData()` method. Thereafter, the `exchangeData()` method will be called according to the **Exchange Data Interval** parameter until the **Stop Exchange Data Schedule** time is reached.

See **exchangeData()** on page 72, **"Exchange Data Interval" on page 59**, and **"Stop Exchange Data Schedule" on page 60** for more information.

Stop Exchange Data Schedule

Description

Establishes the schedule to stop data exchange.

Required Values

One of the following:

- One or more specific dates/times

- A single repeating interval (such as yearly, weekly, monthly, daily, or every *n* seconds).

Down Timeout

Description

Specifies the number of seconds that the e*Way will wait between calls to the **connectionEstablish()** method. See **connectionEstablish()** method for more information.

Required Values

An integer between 1 and 86,400. The default is 15.

Up Timeout

Description

Specifies the number of seconds the e*Way will wait between calls to the **External Connection Verification** method. See **connectionVerify()** on page 71 for more information.

Required Values

An integer between 1 and 86,400. The default is 15.

Resend Timeout

Description

Specifies the number of seconds the e*Way will wait between attempts to resend a message (Event) to the external system, after receiving an error message from the external system.

Required Values

An integer between 1 and 86,400. The default is 10.

5.1.3. Java VM Configuration

The parameters in this section help you set up the information required by the e*Way to utilize the Java VM.

Conceptually, an e*Way is divided into two halves. One half of the e*Way (shown on the left in **Figure 10**) handles communication with the external system; the other half manages the Collaborations that process data and subscribe or publish to other e*Gate components.

Figure 10 e*Way internal architecture
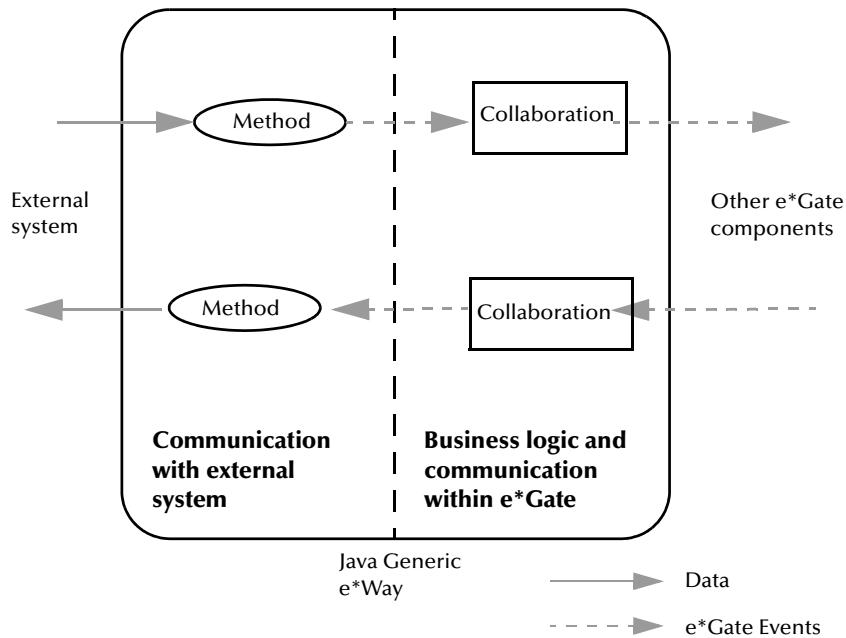
The "communications half" of the e*Way uses Java methods to start and stop scheduled operations, exchange data with the external system, package data as e*Gate "Events" and send those Events to Collaborations, and manage the connection between the e*Way and the external system. The **Java VM Configuration** options discussed in this section control the Java VM environment and define the Java methods used to perform these basic e*Way operations.

Operational Details

The Java methods in the "communications half" of the e*Way fall into the following groups:

Type of Operation	Name
Initialization	**startUp()** on page 74 (also see **Exchanger Java Class** on page 65)
Connection	**connectionEstablish()** on page 70 **connectionVerify()** on page 71 **connectionShutdown()** on page 71
Schedule-driven data exchange	**exchangeData()** on page 72 **ACK()** on page 70 **NAK()** on page 72
Shutdown	**shutdown()** on page 73
Event-driven data exchange	**processOutgoing()** on page 73

A series of figures on the next several pages illustrate the interaction and operation of these methods.

Initialization methods

Figure 11 illustrates how the e*Way executes its initialization methods.

Figure 11   Initialization Methods
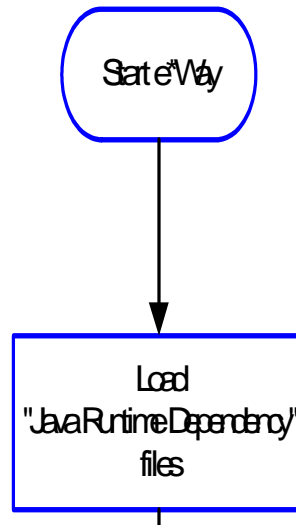
Connection Methods

Figure 12 illustrates how the e*Way executes the connection establishment and verification methods.

Figure 12   Connection establishment and verification methods

Note: *The e*Way selects the connection method based on an internal "up/down" flag rather than a poll to the external system.*
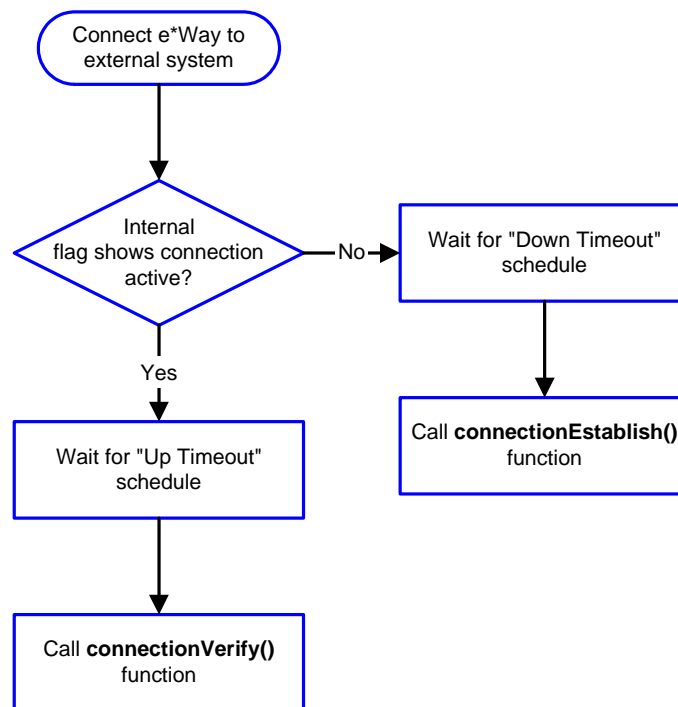*User methods can manually set this flag using Java methods. See* **sendExternalUp** *on page 93 and* **sendExternalDown** *on page 92 for more information.*

Figure 13 illustrates how the e*Way executes its **connectionShutdown()** method.

Figure 13 Connection shutdown method
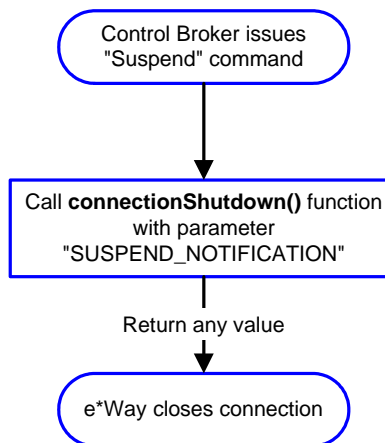
Java Release

Description

Specifies the Java Release this e*Way will use. This parameter is **mandatory**.

Note: *If Java 1 is selected, some of the following parameters may not pertain and their corresponding functionality may not be available as well.*

Required Values

A string. The configured default values are **1** (**1.1.7B and higher**), or **2**, (**1.3 and higher**).

Additional Information

Although Java byte codes for classes and methods, common to both Java 1 and Java 2, are typically forward and backward compatible, it is recommended that this e*Way be set to the Java release appropriate to the version of the **javac** compiler used to compile the Java source code that is used.

See also:

- **JNI DLL** on page 65

- **Disable JIT** on page 68

- **DLL Load Path Prepend** on page 69

JNI DLL

Description

Specifies the absolute pathname to where the JNI DLL installed by the *Java 2 SDK* or *JRE* is located on the Participating Host. This parameter is **mandatory**.

Required Values

A valid pathname.

Additional Information

The JNI dll name varies on different O/S platforms:

OS	Java 2 JNI DLL Name	Java 1 JNI DLL Name
Windows	jvm.dll	javai.dll
Solaris	libjvm.so	libjava.so
Linux	libjvm.so	libjava.so
HP-UX	libjvm.sl	libjava.sl
AIX	libjvm.a	libjava.a

The value assigned can contain a reference to an environment variable, by enclosing the variable name within a pair of % symbols. For example:

```
%MY_JNIDLL%
```

Such variables can be used when multiple Participating Hosts are used on different platforms.

See also **Java Release** on page 64.

Note: *To ensure that the JNI DLL loads successfully, the DLL search path environment variable must be set appropriately to include all the directories under the Java 2 SDK (or JDK) installation directory that contain shared libraries (UNIX) or DLLs (Windows). See page 14 for more information.*

Exchanger Java Class

Description

Specifies the location, relative to the e*Gate Registry, of the Java class which implements the Exchanger interface: **com.stc.common.collabService.Exchanger**.

Required Values

A pathname. This parameter is **mandatory**. For example:

```
classes\com\mystc\collabpkg\MyExchanger.class
```

Additional information

Alternatively, if the implementing Java class resides inside a .jar or .zip file specified below as a **Runtime Dependency**, enter the class name with periods.

For example:

```
com.mystc.collabpkg.MyExchange
```

Runtime Dependency

Description

Specifies a .jar, .zip, or .class file that needs to be downloaded from the e*Gate Registry before the above specified **Exchanger Java Class** can run. If multiple dependencies exist, place all necessary references within an e*Gate Registry Control file (ends with a .ctl suffix), which itself is checked into the e*Gate Registry, and specify the Control file instead.

Required Values

A valid .jar, .zip, .class or .ctl file. This parameter is optional.

Enable Custom Data Error Handling

Description

Specifies whether data errors that occur during the execution of the **processOutgoing()** method are to be handled in a customized way.

Required Values

Yes or **No**.

Additional Information

When data errors occur during the transmission of an outgoing Event, an exception is thrown by the processOutgoing() method. The **dataErrorHandled()** method of the Exchanger Java Class will be called, and provided with the Event for which the exception was thrown and the exception.

The Event and the exception received by **dataErrorHandled()** can optionally be forwarded to a "dead-letter" IQ and a boolean true returned to indicate that the data error has been handled successfully. This allows the e*Way to continue with the next outbound Event.

If the data error cannot be handled, a boolean false must be returned. The e*Way will follow the standard recourse of resending the outbound Event as specified in **Max Failed Messages**, before shutting down.

Note: *To use the Enable Custom Data Error Handling parameter, the Exchanger Java Class must implement the **com.stc.common.collabService.DataErrorHandler** interface.*

See also :

- **"Exchanger Java Class" on page 65**
- **"Max Failed Messages" on page 58**

Initial Heap Size

Description

Specifies the value for the initial heap size in bytes. If set to 0 (zero), the preferred value for the initial heap size of the Java VM will be used.

Required Values

An integer between 0 and 2147483647. This parameter is optional.

Maximum Heap Size

Description

Specifies the value of the maximum heap size in bytes. If set to 0 (zero), the preferred value for the maximum heap size of the Java VM will be used.

Required Values

An integer between 0 and 2147483647. This parameter is optional.

CLASSPATH Override

Description

Specifies the complete CLASSPATH variable to be used by the Java VM. This parameter is optional. If left empty, an appropriate CLASSPATH environment variable (consisting of required e*Gate components concatenated with the system version of CLASSPATH) will be set.

*Note: All necessary .jar and .zip files needed by both e*Gate and the Java VM must be included. It is advised that the **CLASSPATH Prepend** parameter should be used.*

Required Values

An absolute path or an environmental variable. This parameter is optional.

Additional Information

Existing environment variables may be referenced in this parameter by enclosing the variable name in a pair of % signs. For example:

```
%MY_CLASSPATH%
```

See also **CLASSPATH Prepend** on page 67.

CLASSPATH Prepend

Description

Specifies the paths to be prepended to the CLASSPATH environment variable for the Java VM.

Required Values

An absolute path or an environmental variable. This parameter is optional.

Additional Information

If left unset, no paths will be prepended to the CLASSPATH environment variable.

Existing environment variables may be referenced in this parameter by enclosing the variable name in a pair of % signs. For example:

```
%MY_PRECLASSPATH%
```

See also **CLASSPATH Override** on page 67.

Disable Class Garbage Collection

Description

Specifies whether the Class Garbage Collection will be done automatically by the Java VM. The selection affects performance issues.

Required Values

YES or **NO**.

Additional Information

If set to **NO**, the size of the Java VM and this e*Way will grow uncontrollably larger unless the executed Java code calls the garbage collector, **System.gc()**, itself.

Enable Garbage Collection Activity Reporting

Description

Specifies whether garbage collection activity will be reported for debugging purposes.

Required Values

YES, or **NO**.

Report Java VM Class Loads

Description

Specifies whether the Java VM information and all class loads will be reported for debugging purposes.

Required Values

YES, or **NO**.

Disable JIT

Description

Specifies whether the Just-In-Time (JIT) compiler will be disabled.

Required Values

YES or **NO**.

Note: This parameter is not supported for Java Release 1.

See also **Java Release** on page 64.

DLL Load Path Prepend

Description

Specifies any paths to be prepended to the dll load path used by the Java VM.

Required Values

An absolute path or an environmental variable. This parameter is optional.

Additional Information

Existing environment variables may be referenced in this parameter by enclosing the variable name in a pair of % signs. For example:

```
%MY_PRELOADPATH%
```

Note: *This parameter is not supported for Java Release 1.*

See also **Java Release** on page 64.

5.2 Methods Required by the Exchanger Interface

The following methods are required by the Exchanger interface that the class file must implement in order for the e*Way to operate.

ACK() on page 70

connectionEstablish() on page 70

connectionShutdown() on page 71

connectionVerify() on page 71

exchangeData() on page 72

NAK() on page 72

processOutgoing() on page 73

shutdown() on page 73

startUp() on page 74

See **Exchanger Java Class** on page 65 for more information.

ACK()

Description

This method is called when the e*Way succeeds to process and queue data from external.

Parameters

Name	Type	Description
ackevt	byte array	An acknowledgment event to be sent to the external.

Return Values

void
Returns a void, if successful; otherwise

Exception
com.stc.common.collabService.CollabConnException, indicating that a problem with the connection occurred.

connectionEstablish()

Description

This method is called repeatedly at the set interval whenever the connection to the external is down or in a down state, to attempt to establish the connection.

Parameters

None

Return Values

Boolean
Returns **true** if successful; otherwise, returns **false**.

connectionShutdown()

Description

This method shuts down the connection to external.

Parameters

Name	Type	Description
notif	string	A notification string "SUSPEND_NOTIFICATION" will be passed in to indicate the connection should be shut down.

Return Values

Boolean
Returns **true** if successful; otherwise, returns **false**.

Additional Information

This method will only be invoked when the e*Way receives a "suspend" command from a Control Broker. When the "suspend" command is received, the e*Way will invoke this method, passing the string "SUSPEND_NOTIFICATION" as an argument.

connectionVerify()

Description

This method is called repeatedly at the set interval whenever the connection to the external is thought to be up, and either confirms that it is still up or discovers that it has gone down.

Parameters

None

Return Values

Boolean
Returns **true** if successful; otherwise, returns **false**.

exchangeData()

Description

This method will be invoked at the **Exchange Data Interval** schedule as long as the exchange schedules are defined. If this method returns data, it will be queued up for e*Gate. This method will not be invoked if the **Exchange Data Interval** is set to 0 (zero).

See also:

- **Start Exchange Data Schedule** on page 59
- **Stop Exchange Data Schedule** on page 60

Parameters

None

Return Values

byte array
if successful, otherwise,

exception

- **com.stc.common.collabService.CollabConnException**, if a problem with the connection occurred, or

- **com.stc.common.collabService.CollabDataException**, if a problem with the data occurred.

Additional Information

The method may return an empty byte array or NULL which is not considered an error.

NAK()

Description

This method is called when the e*Way fails to process and queue data from external.

Parameters

Name	Type	Description
nakevt	byte array	A negative acknowledgment event to be sent to the external.

Return Values

void
Returns a void if successful; otherwise,

exception
com.stc.common.collabService.CollabConnException, indicating that a problem with the connection occurred.

processOutgoing()

Description

This method sends outgoing Events from e*Gate to the external. When the e*Way has an Event to send to the external, it will invoke this method.

Parameters

Name	Type	Description
inputEvent	byte array	The Event to be sent.

Return Value

void
Returns a void if successful; otherwise,

exception

- **com.stc.common.collabService.CollabConnException** indicates that there is a problem communicating with the external system.
- **com.stc.common.collabService.CollabDataException** indicates that there is a problem with the Event data itself.
- **com.stc.common.collbService.CollabResendException** indicates that the Event can be resent.

Note: *To return the data back to the e*Gate system, the method should call* **EGate.eventSendToEgate()** *method. See* **eventSendToEgate** *on page 91 for more information.*

shutdown()

Description

This method is called to shut down the e*Way, and thus notifies the Java exchange class that it is about to shut down. This method can be used to shutdown the connection with external.

Parameters

Name	Type	Description
notif	string	A notification string, "SHUTDOWN_NOTIFICATION" will be passed in to indicate the e*Way is shutting down.

Return Values

Boolean
Returns **true** if successful; otherwise, returns **false**.

Additional Information

If this method returns Boolean false, the e*Way will assume that the method will call the **EGate.shutdownRequest** method when it is ready to shut down.

Note: *If the method is going to control the shutdown, it must do so in a timely manner. The rest of the system is expecting the e*Way to exit.*

startUp()

Description

This method should be used to initialize the external system before data exchange starts. The **startUp** method is invoked by the e*Way at the startup time and when the configuration changes before it enters into its initial Communication State. This method is called after the e*Way will exit if it fails to invoke this method or this method throws a **com.stc.common.collabService.CollabConnException**.

Parameters

None

Return Values

void
Returns a void if successful; otherwise,

exception
com.stc.common.collabService.CollabConnException, indicating that a problem with the connection occurred.

Additional Information

This method is called after the e*Way loads any **Runtime Dependency**.

5.2.1. CollabConnException Class

The CollabConnException class implements an Exception to be thrown when a Connection Error occurs in an e*Gate Collaboration. The CollabConnException class extends java.lang.Exception.

The interface is located in:

```
com.stc.common.collabService
```

CollabConnException

Description

CollabConnException constructs an Exception due to Collaboration Connection errors.

Syntax

```
public CollabConnException()
```

Parameters

None.

Return Value

None.

CollabConnException

Description

CollabConnException constructs an Exception due to Collaboration Connection errors.

Syntax

```
public CollabConnException(java.lang.String s)
```

Parameters

Name	Type	Description
s	String	The associated exception message string.

Return Value

None.

5.2.2. CollabDataException Class

The CollabDataException class implements an Exception to be thrown when a Data translation Error occurs in an e*Gate Collaboration. The CollabDataException class extends java.lang.Exception.

The interface is located in:

```
com.stc.common.collabService
```

CollabDataException

Description

CollabDataException constructs an Exception due to Collaboration Data errors.

Syntax

```
public CollabDataException()
```

Parameters

None.

Return Value

None.

CollabDataException

Description

CollabDataException constructs an Exception due to Collaboration Data errors.

Syntax

```
public CollabDataException(java.lang.String s)
```
Parameters

Name	Type	Description
s	String	The associated exception message string.

Return Value

None.

5.2.3. CollabResendException Class

The CollabResendException class implements an Exception to be thrown when a Connection Error occurs in an e*Gate Collaboration. The CollabResendException class extends java.lang.Exception.

The interface is located in:

```
com.stc.common.collabService
```

CollabResendException

Description

CollabResendException constructs an Exception for resending due to Collaboration Connection errors.

Syntax

```
public CollabResendException()
```
Parameters

None.

Return Value

None.

CollabResendException

Description

CollabResendException constructs an Exception for resending due to Collaboration Connection errors.

Syntax

```
public CollabResendException(java.lang.String s)
```

Parameters

Name	Type	Description
s	String	The associated exception message string.

Return Value

None.

5.3 Exchanger Interface

The Exchanger.class Interface required for the e*Way to operate is located in the following package of the **stcjcs.jar** file:

```
com.stc.common.collabService
```

The code is fully commented. The primary methods are discussed in **"Methods Required by the Exchanger Interface" on page 70**. The core methods will be discussed in greater detail in the following chapter.

```
package com.stc.common.collabService;

/**
 *  An interface a class must implement in order to serve as an e*Gate
 *  Collaboration.
 *  <p>
 *  The implementing class must ALSO have a no-argument constructor so that
 *  the class can be dynamically instantiated.
 *
 */
public interface Exchanger
{
  // ============================================================================
  //                              Abstract Methods
  // ============================================================================

   /**
    *  Called at start up of a Java exchange class
    *
    *  @exception com.stc.common.collabService.CollabConnException  thrown if
    *                          problem encountered with a connection
    *
    */
   public void startUp()
                throws CollabConnException;

   /**
    *  Called to process an outgoing event from e*Gate
    *
    *  @param    outEvent        output event data given as a byte array
    *  @exception com.stc.common.collabService.CollabConnException  thrown if
    *                          problem encountered with a connection
    *  @exception com.stc.common.collabService.CollabDataException  thrown if
    *                          problem encountered with data translation
    *  @exception com.stc.common.collabService.CollabResendException  thrown if
    *                          the outgoing event is to be resent
    *
    */
   public void processOutgoing(byte[] inputEvent)
                throws CollabConnException, CollabDataException,
                       CollabResendException;

   /**
    *  Called to exchange data from an external with e*Gate on a predefined
    *  schedule.
    *
    *  @return   a byte array for the data received from the external
    *  @exception com.stc.common.collabService.CollabConnException  thrown if
    *                          problem encountered with a connection
    *  @exception com.stc.common.collabService.CollabDataException  thrown if
```

```
 *                            problem encountered with data translation
 *
 */
public byte[] exchangeData()
              throws CollabConnException, CollabDataException;

/**
 *  Called to establish a connection with the external.
 *
 *  @return    a boolean <code>true</code> when successfully connected;
 *             otherwise a <code>false</code>
 *
 */
public boolean connectionEstablish();

/**
 *  Called to verify a connection with the external.
 *
 *  @return    a boolean <code>true</code> when connection is intact;
 *             otherwise a <code>false</code>
 *
 */
public boolean connectionVerify();

/**
 *  Called to shut down a connection with the external.
 *
 *  @param    notif   a notification string "SUSPEND_NOTIFICATION" will be
 *                    passed in to indicate the connection should be
 *                    shut down
 *  @return    a boolean <code>true</code> when connection has been severed;
 *             otherwise a <code>false</code>
 *
 */
public boolean connectionShutdown(String notif);

/**
 *  Called to positively acknowledge an external when all the events
 *  received have been processed successfully by all e*Way collaborations.
 *
 *  @param    ackevt  an acknowledgment event to be sent to the external
 *  @exception com.stc.common.collabService.CollabConnException  thrown if
 *                            problem encountered with a connection
 *
 */
public void ACK(byte[] ackevt)
              throws CollabConnException;

/**
 *  Called to negatively acknowledge an external when not all the events
 *  received have been processed successfully by all e*Way collaborations.
 *
 *  @param    nakevt  an acknowledgment event to be sent to the external
 *  @exception com.stc.common.collabService.CollabConnException  thrown if
 *                            problem encountered with a connection
 *
 */
public void NAK(byte[] nakevt)
              throws CollabConnException;

/**
 *  Called to notify the Java exchange class that the e*Way is shutting
 *  down.
 *
 *  @param    notif   a notification string, "SHUTDOWN_NOTIFICATION", to
 *                    advise that the e*Way is about to shut down.
 *  @return    a boolean <code>true</code> if the shutdown process can
 *             proceed; otherwise a <code>false</code>
 *
 */
public boolean shutdown(String notif);
}
```

5.4 Methods Required by the DataErrorHandler Interface

When data errors occur during the transmission of an outgoing Event(s) to external, an exception is thrown by the **processOutgoing()** method. The **dataErrorHandled()** method of the Exchanger Java Class will be called and provided with the offending Event along with the associated exception.

The following methods are required by the **DataErrorHandler Interface** that the class file must implement in order for the e*Way to operate.

dataErrorHandled() on page 79

dataErrorHandled()

Description

This method is called when a CollabDataException is thrown by the collaboration.

Parameters

Name	Type	Description
Event	byte array	The offending Event as a byte array.
e		The exception thrown

Return Value

Boolean

Returns a true if the Data Exception was handled and the e*Way can continue processing the next Event; otherwise, returns false.

5.5   DataErrorHandler Interface

The DataErrorHandler.class Interface must be implemented for the e*Way if custom data error handling is enabled (See **Enable Custom Data Error Handling** on page 66 for more information). The interface is located in the following package of the **stcjcs.jar** file:

```
com.stc.common.collabService
```

The code is fully commented. The primary methods are discussed in **"Methods Required by the DataErrorHandler Interface" on page 78**.

```java
package com.stc.common.collabService;

import com.stc.common.utils.StcCorp;

/**
 * An interface to handle Data Error from an e*Way processing of an outgoing
 * event.
 *
 */
public interface DataErrorHandler
{
  /**
   *  Called when a CollabDataException is thrown by the collaboration.
   *
   *  @param    event   the offending event as a byte array
   *  @param    e       the Exception thrown
   *
   *  @return   <code>true</code> if the Data Exception was handled and the
   *            e*Way can continue processing the next event;
   *            <code>false</code> otherwise
   *
   */
  public boolean dataErrorHandled(byte[] event, CollabDataException e);
}
```

5.6 Configuring the Java Generic e*Way with the Schema Designer

The instructions in this section discuss how to implement the Java Generic e*Way using the Schema Designer.

After you have created the extension DLL, any required Java methods, and the .def file (if necessary) for the new e*Way, you must do the following:

1 Commit any files you have created to the appropriate directories within a schema.

2 Create an e*Way component within the schema.

3 Configure the e*Way as required.

5.6.1. Step 1: Commit files to the schema

Note: *Do not commit files to the **default** schema unless you want those files to be inherited by all new schemas. Even if this is the desired outcome, we recommend that you always commit files to a non-default schema during testing and development of new e*Way components.*

1 Make sure the files you wish to commit to the e*Gate schema are accessible from the same system as the Schema Designer, either from a local file system or from a mapped network drive (you cannot commit files to the schema using a UNC path).

2 Using the Schema Designer, log in to the schema that will support the new e*Way.

3 Pull down the File menu and select **Commit to Sandbox**.

4 The **Select Local File to Commit** dialog appears. Use the file-selection controls to locate the file you want to commit and click **Open**.

5 The **Select Directory for Committed File** dialog appears. Use the directory-selection controls to locate the directory to which you want to commit the file and click **Select**. Select the directory according to the table below:

Table 6   Schema directories

| For a file of this type…	…commit to this directory
.def	/configs/stcewgenericjava
.java .jar Or any other Java files	classes/eway_name (We recommend that you create a separate directory for your custom e*Way scripts.)
.dll or other library files	/bin

Note: *Remember that committing files to the Sandbox makes them available for testing. Files must be promoted to the run-time schema before they can be used in the working "production" environment. For more information, see the Team Registry User's Guide or the Schema Designer's online Help.*

5.6.2. ## Step 2: Create an e*Way Component

After all the required files have been committed to the schema, you can create the e*Way component.

1 In the Component editor, create a new e*Way.

2 Display the new e*Way's properties.

3 On the General tab, under **Executable File**, click **Find**.

4 Select the file **stcewgenericjava.exe**.

5 Under **Configuration file**, click **New**.

6 The **e*Way Template Selection** dialog box appears. From the list, select the **.def** file that you created for this e*Way and click **OK**. The name will be listed without the ".def" extension. For example, if you created the file **my_eway.def**, the file will be listed as **my_eway**.
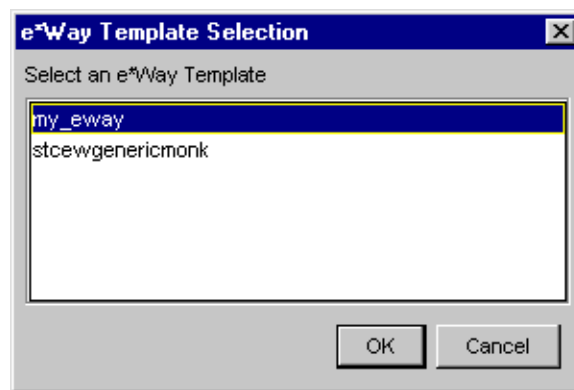
7 The e*Way Editor will launch. You are ready to configure the e*Way; continue with the next section.

5.6.3. ## Step 3: Configure the e*Way

Once you have selected your e*Way template, you are ready to use the e*Way Editor to configure this e*Way component.

1 If you followed the instructions in the previous two sections, the e*Way Editor has now launched, as shown in Figure 14.

Figure 14   e*Way Editor
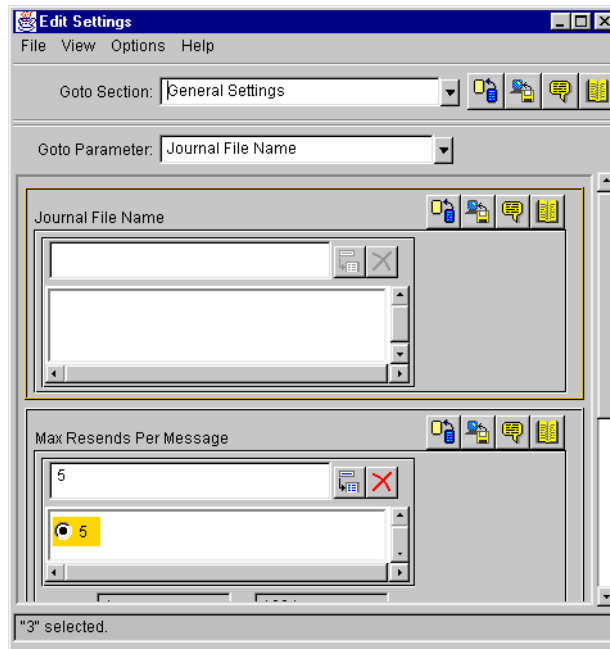
Use the e*Way Editor to make any configuration changes you require. For more information about configuring e*Ways or how to use the e*Way Editor, see the *e*Gate Integrator User's Guide.*

2   When you have finished making configuration changes, pull down the **File** menu and select **Save**.

3   Enter a name for the configuration file and click **OK**.

4   Exit the e*Way Editor. You will return to the e*Way's property sheet. Click **OK** to close the properties sheet, or continue to make other changes to the e*Way component's properties.

*Note: Once you have installed and configured this e*Way, you must incorporate it into a schema by defining and associating the appropriate Collaborations, Collaboration Rules, IQs, and Event Types before this e*Way can perform its intended functions. For more information about any of these procedures, please see the Schema Designer's online Help.*

5.6.4. Editing a .def File Within a Schema

To edit a .def file that has already been committed to a schema:

1   Launch the Schema Designer and log in to the schema containing the **.def** file that you want to edit.

2   Pull down the **File** menu and select **Edit File.**

3   Use the file-selection controls to open the **.def** file. The Notepad editor will launch and open the file you have selected.

4 Save any changes and exit the editor.

5 Commit the edited file back to the schema (the Schema Designer will automatically prompt you to perform this procedure).

See the Schema Designer's online Help for more information.

5.7 Developing the Java Business Logic Class

In the sample code in this section, `FileExchange.java` is the Java class you have created. The ***.class** file must be imported into the schema in which the Java Exchanger Interface runs. See the following **Samples** folder on the e*Gate installation CD-ROM to obtain a copy of the sample:

`samples\genjavaeway\FileExchange.java`

5.7.1 Sample Java Business Logic

Java Business Logic Classes use the following basic format as illustrated by the following sample.

```
/**
 *  A sample class to illustrate implementation of the Exchanger interface.
 *  A flat file is considered as the "external" system for both inbound and
 *  outbound processing.
 *
 */


//  Java specific package imports


import java.io.*;


//  e*Gate specific package imports


import com.stc.common.collabService.*;
import com.stc.common.registry.*;
import com.stc.common.utils.*;


public class FileExchange implements Exchanger
{
  private String              eGateLogsDir   = null;
  private FileOutputStream     outFos        = null;
  private File                 outFile       = null;
  private String               customInputDir = null;


  // ------------------------------------------------------------------------


  /**
   *  Zero-argument constructor is needed (Java will provide one if not
   *  defined, but it's better to be explicit).
   *
   */
  public FileExchange()
  {
    super();
  }


  // ------------------------------------------------------------------------


  /**
   *  This gets called when the Java e*Way initially starts up.  Going to
   *  use it to discover where the e*Gate client logs/ directory is and
   *  to get the user customized configuration parameter "Inbound Directory"
```

```java
 *  under section "Sample Test".
 *
 *  @exception com.stc.collabService.CollabConnException     thrown if
 *              problem encountered
 *
 */
public void startUp() throws CollabConnException
{
  //  Determine from the e*Gate repository where the client logs/
  //  directory is.

  RepositoryDirectories repDir = new RepositoryDirectories();
  if (repDir.readRepositoryDirectories())
  {
    eGateLogsDir = repDir.getLogs();
    EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
                  "startUp(): e*Gate client logs/ is at: " + eGateLogsDir);
  }
  else
  {
    EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_ERROR,
                  "startUp(): Can't read repository directories");
    throw new CollabConnException("Can't read repository directories");
  }

  //  Get the user customized configuration parameter

  customInputDir = EGate.getEwayConfigProp("SAMPLE_TEST.INBOUND_DIRECTORY");
  if (null == customInputDir)
  {
    EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_WARNING,
                  "startUp(): No Inbound Directory defined in Sample Test");
  }
  else
  {
    File inputDir = new File(customInputDir);

    if (!inputDir.isAbsolute())
    {
      inputDir = new File(repDir.getSystemData(), customInputDir);
      customInputDir = inputDir.getAbsolutePath();
    }

    if (!inputDir.isDirectory())
    {
      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_ERROR,
                    "startUp(): Inbound Directory " +
                    customInputDir + " doesn't exist!");
      throw new CollabConnException(customInputDir + " doesn't exist!");
    }
  }
}

//  ----------------------------------------------------------------------

/**
 *  This gets called when there's an outbound event from e*Gate to the
 *  external.  We're simply going to write it out to the output file.
 *
```

```
 *   @param     outEvent       output event data given as a byte array
 *   @exception com.stc.common.collabService.CollabConnException  thrown if
 *                              problem encountered with a connection
 *   @exception com.stc.common.collabService.CollabDataException  thrown if
 *                              problem encountered with data translation
 *   @exception com.stc.common.collabService.CollabResendException  thrown if
 *                              the outgoing event is to be resent
 *
 */
public void processOutgoing(byte[] outEvent)
            throws CollabConnException, CollabDataException,
                    CollabResendException
{
  if (outFos != null && outEvent.length > 0)
  {
    try
    {
      outFos.write(outEvent);
      outFos.write(System.getProperty("line.separator").getBytes());
      outFos.flush();
      EGate.traceln(EGate.TRACE_EWAY_VERBOSE, EGate.TRACE_EVENT_TRACE,
                    outEvent, "processOutgoing(): wrote to file");
    }
    catch (IOException e)
    {
      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_ERROR,
                    "processOutgoing(): Can't write to output file: " +
                    e.getMessage());
      throw new CollabConnException(e.getMessage());
    }
  }
}

// ----------------------------------------------------------------------

/**
 *  This gets called when the Java e*Way's exchange data with external
 *  schedule is due.  We're just going to look for an input file in the
 *  user customized configured inbound directory.
 *
 *
 *  @return   a byte array for the data received from the external
 *  @exception com.stc.common.collabService.CollabConnException  thrown if
 *                              problem encountered with a connection
 *  @exception com.stc.common.collabService.CollabDataException  thrown if
 *                              problem encountered with data translation
 *
 */
public byte[] exchangeData()
            throws CollabConnException, CollabDataException
{
  if (customInputDir != null)
  {
    File inputFile = new File(customInputDir, "TestIn.txt");
    long len;

    if (inputFile.exists() && (len = inputFile.length()) > 0)
    {
      FileInputStream fis = null;
```

```
        try
        {
          fis = new FileInputStream(inputFile);
          len = fis.available();
          byte[] retBytes = new byte[(int) len];
          fis.read(retBytes);

          EGate.traceln(EGate.TRACE_EWAY_VERBOSE, EGate.TRACE_EVENT_TRACE,
                        retBytes, "exchangeData(): received from file");
          return retBytes;
        }
        catch (Exception e)
        {
          EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_ERROR,
                        "exchangeData(): Input file problems: " +
                        e.getMessage());
          throw new CollabConnException("Input file problems: " +
                                        e.getMessage());
        }
        finally
        {
          if (fis != null)
          {
            try
            {
              fis.close();
            }
            catch (Exception e)
            {
            }
          }

          inputFile.renameTo(new File(customInputDir, "TestIn.~xt"));
        }
      }
    }

    return null;
}


// ------------------------------------------------------------------------

/**
 *  This gets called to establish a connection with an external system.
 *  We're simply going to open the output file here.
 *
 *  @return   a boolean <code>true</code> when successfully connected;
 *            otherwise a <code>false</code>
 *
 */
public boolean connectionEstablish()
{
  //  Open an output file in the logs/ directory

  try
  {
    outFile = new File(eGateLogsDir, "TestOut.txt");
    outFos = new FileOutputStream(outFile);
```

```
      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
        "connectionEstablish(): Successfully opened output file TestOut.txt");
  }
  catch (Exception e)
  {
      EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_WARNING,
        "connectionEstablish(): Failed to open output file TestOut.txt: " +
        e.getMessage());

      return false;
  }
  return true;
}

// ------------------------------------------------------------------------

/**
 *  This gets called to verify a connection with the external.  We're
 *  just going to test if the output file exists.
 *
 *  @return   a boolean <code>true</code> when connection is intact;
 *            otherwise a <code>false</code>
 *
 */
public boolean connectionVerify()
{
  EGate.traceln(EGate.TRACE_EWAY_VERBOSE, EGate.TRACE_EVENT_TRACE,
                "connectionVerify(): test if output file exists?");
  return (outFile != null && outFile.exists());
}

// ------------------------------------------------------------------------

/**
 *  This gets called to shut down a connection with the external.  We're
 *  going to close the output file here.
 *
 *  @param    notif   a notification string "SUSPEND_NOTIFICATION" will be
 *                    passed in to indicate the connection should be
 *                    shut down
 *  @return   a boolean <code>true</code> when connection has been severed;
 *            otherwise a <code>false</code>
 *
 */
public boolean connectionShutdown(String notif)
{
  EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
                "connectionShutdown(): got request: " + notif);
  if (outFos != null)
  {
    try
    {
      outFos.close();
      outFos = null;
    }
    catch (Exception e)
    {
    }
  }
```

```
  return true;
}

// ------------------------------------------------------------------------

/**
 *  Gets called to positively acknowledge an external when all the events
 *  received have been processed successfully by all e*Way collaborations.
 *
 *  @param    ackevt  an acknowledgment event to be sent to the external
 *  @exception com.stc.common.collabService.CollabConnException  thrown if
 *                          problem encountered with a connection
 *
 */
public void ACK(byte[] ackevt)
              throws CollabConnException
{
  EGate.traceln(EGate.TRACE_EWAY_VERBOSE, EGate.TRACE_EVENT_TRACE,
                ackevt, "ACK(): gotten for");
}

// ------------------------------------------------------------------------

/**
 *  Gets called to negatively acknowledge an external when not all the events
 *  received have been processed successfully by all e*Way collaborations.
 *
 *  @param    nakevt  an acknowledgment event to be sent to the external
 *  @exception com.stc.common.collabService.CollabConnException  thrown if
 *                          problem encountered with a connection
 *
 */
public void NAK(byte[] nakevt)
              throws CollabConnException
{
  EGate.traceln(EGate.TRACE_EWAY_VERBOSE, EGate.TRACE_EVENT_WARNING,
                nakevt, "NAK(): gotten for");
}

// ------------------------------------------------------------------------

/**
 *  Gets called to notify the Java exchange class that the e*Way is shutting
 *  down.
 *
 *  @param    notif   a notification string, "SHUTDOWN_NOTIFICATION", to
 *                    advise that the e*Way is about to shut down.
 *  @return   a boolean <code>true</code> if the shutdown process can
 *            proceed; otherwise a <code>false</code>
 *
 */
public boolean shutdown(String notif)
{
  EGate.traceln(EGate.TRACE_EWAY, EGate.TRACE_EVENT_INFORMATION,
                "shutdown(): got request: " + notif);
  return true;
}
}
```

Core Java Generic e*Way Methods

This chapter describes the core methods used within the Java Generic e*Way.

6.1 Core Functions

The following static methods of the **EGate class** are available to all Java Generic e*Ways. The **EGate class** is found in the **com.stc.common.collabService** package.

- **eventSendToEgate** on page 91
- **getEwayConfigProp** on page 91
- **getLogicalName** on page 92
- **sendExternalDown** *on page 92*
- **sendExternalUp** *on page 93*
- **shutdownRequest** on page 93
- **startSchedule** on page 94
- **stopSchedule** on page 94
- **traceln** on page 95
- **traceln** on page 95

eventSendToEgate

Syntax

```
boolean eventSendToEgate(byte[] event);
```

Description

eventSendToEgate sends data that the e*Way has already received from the external system into the e*Gate system as an Event.

Parameters

Name	Type	Description
event	byte array	The data to be sent to the e*Gate system expressed as byte array.

Return Values

Boolean
Returns **true** if the data is sent successfully; otherwise, returns **false**.

Throws

None.

Additional information

This method can be called by any e*Way method when it is necessary to send data to the e*Gate system in a blocking fashion, that is, when the Event has been appropriately processed by a configured inbound collaboration and posted if necessary to an IQ.

Examples

```
EGate.eventSendToEgate("Test event");
```

getEwayConfigProp

Syntax

string getEwayConfigProp(String key);

Description

getEwayConfigProp() returns the string corresponding to key used to identify the property requested.

Parameters

Name	Type	Description
key	String	The property key used to identify the configuration value requested.

Return Values

String

Returns a string containing the configuration property requested; otherwise, returns a null string.

Examples

```
EGate.getEwayConfigProp("AUTHENTICATION.PASSWORD");

EGate.getEwayConfigProp("CONNECTION_PARAMETERS.GATEWAY_ID");
```

getLogicalName

Syntax

```
string getLogicalName();
```

Description

getLogicalName() returns the logical name of the e*Way.

Parameters

None.

Return Values

String

Returns the name of the e*Way (as defined by the Schema Designer).

Throws

None.

Examples

```
EGate.getLogicalName();
```

sendExternalDown

Syntax

```
void sendExternalDown();
```

Description

sendExternalDown() instructs the e*Way that the connection to the external system is down.

Parameters

None.

Return Values

None.

Throws

None.

Examples

```
EGate.sendExternalDown();
```

sendExternalUp

Syntax

```
void sendExternalUp();
```

Description

sendExternalUp() instructs the e*Way that the connection to the external system is up.

Parameters

None.

Return Values

None.

Throws

None.

Examples

```
EGate.sendExternalUp();
```

shutdownRequest

Syntax

```
void shutdownRequest();
```

Description

shutdownRequest completes the e*Gate shutdown procedure that was initiated by the Control Broker but was interrupted by returning a **false** value within the Shutdown Command Notification Method (see **connectionShutdown()** on page 71). Once this method is called, shutdown proceeds immediately.

Once interrupted, the e*Way's shutdown cannot proceed until this Java method is called. If you do interrupt an e*Way shutdown, we recommend that you complete the process in a timely fashion.

Parameters

None.

Return Values

None.

Throws

None.

Examples

```
EGate.shutdownRequest();
```

startSchedule

Syntax

```
void startSchedule();
```

Description

startSchedule requests that the e*Way execute the **exchangeData()** method specified within the e*Way's configuration file. Does not effect any defined schedules.

Parameters

None.

Return Values

None.

Throws

None.

Examples

```
EGate.startSchedule();
```

stopSchedule

Syntax

```
void stopSchedule();
```

Description

stopSchedule requests that the e*Way halt execution of the **exchangeData()** method specified within the e*Way's configuration file. Execution will be stopped when the e*Way concludes any open transaction. Does not affect any defined schedules, and does not halt the e*Way process itself.

Parameters

None.

Return Values

None.

Throws

None.

Examples

```
EGate.stopSchedule();
```

traceln

Syntax

> void traceln (long tid, long event, String message);

Description

> **traceln** adds a trace entry into the e*Way's log file. The end of line is automatically appended.

Parameters

Name	Type	Description
tid	long	The trace id code. Two possibilities are: **EGate.TRACE_EWAY** EGate.TRACE_EWAY_VERBOSE
event	long	The Event upon which the trace is set. The possibilities are: **EGate.TRACE_EVENT_TRACE** **EGate.TRACE_EVENT_ERROR** **EGate.TRACE_EVENT_WARNING** **EGate.TRACE_EVENT_DEBUG** **EGate.TRACE_EVENT_INFORMATION**
message	String	The message to appended to the log file data.

Return Values

> None.

Examples

```
EGate.traceln(EGate.TRACE_EWAY,EGate.TRACE_EVENT_TRACE,
          "This goes to the e*Way log file");
```

traceln

Syntax

> void traceln(long tid, long event, byte[] blob, String tracestr);

Description

> **traceln** adds a trace entry for a blob in hex-dump format into the e*Way's log file. The end of line is automatically appended.

Parameters

Name	Type	Description
tid	long	The trace id code. Two possibilities are: **EGate.TRACE_EWAY** EGate.TRACE_EWAY_VERBOSE

Name	Type	Description
event	long	The Event upon which the trace is set. For example: **EGate.TRACE_EVENT_INFORMATION**
blob	byte array	The blob. For example: **byte[ ]**
tracestr	String	The message to appended to the log file data. For example: **String**

Return Values

None.

Examples

```
byte[ ] hugeByteArray;
EGate.traceln(EGate.TRACE_EWAY_VERBOSE,EGate.TRACE_EVENT_DEBUG,
          hugeByteArray, "This is a huge byte array");
```

Note: *For more information on trace flags, see **e*Gate Integrator System Administration and Operations Guide**.*

Introducing the Java Monk Extension e*Way

The Java Monk Extension e*Way enables the e*Gate system to interchange data with an external application by using Monk to access Java objects and call their methods. This portion of the document describes how to configure the e*Way Intelligent Adapter for Java.

7.0.1. Components

The following components comprise the Java Monk Extension e*Way:

- **stcewgenericmonk.exe**, the executable component
- Configuration files, which the e*Way Editor uses to define configuration parameters
- Monk function scripts
- Library files

A complete list of installed files appears in **Table 1 on page 17**.

Java Monk Extension e*Way Functions

The Java Monk Extension e*Way functions fall into the following categories:

- **Basic Functions** on page 98
- **Standard e*Way Functions** on page 102
- **Java Monk Extension e*Way Native Functions** on page 108

8.1 Basic Functions

The functions in this category control the e*Way's most basic operations.

The basic functions are

start-schedule on page 98

stop-schedule on page 99

send-external-up on page 99

send-external-down on page 100

get-logical-name on page 100

event-send-to-egate on page 100

shutdown-request on page 101

start-schedule

Syntax

```
(start-schedule)
```

Description

start-schedule requests that the e*Way execute the "Exchange Data with External" function specified within the e*Way's configuration file. Does not effect any defined schedules.

Parameters

None.

Return Values

None.

Throws

None.

stop-schedule

Syntax

```
(stop-schedule)
```

Description

stop-schedule requests that the e*Way halt execution of the "Exchange Data with External" function specified within the e*Way's configuration file. Execution will be stopped when the e*Way concludes any open transaction. Does not affect any defined schedules, and does not halt the e*Way process itself.

Parameters

None.

Return Values

None.

Throws

None.

send-external-up

Syntax

```
(send-external-up)
```

Description

send-external-up instructs the e*Way that the connection to the external system is up.

Parameters

None.

Return Values

None.

Throws

None.

send-external-down

Syntax

```
(send-external-down)
```

Description

send-external down instructs the e*Way that the connection to the external system is down.

Parameters

None.

Return Values

None.

Throws

None.

get-logical-name

Syntax

```
(get-logical-name)
```

Description

get-logical-name returns the logical name of the e*Way.

Parameters

None.

Return Values

string
Returns the name of the e*Way (as defined by the Schema Designer).

Throws

None.

event-send-to-egate

Syntax

```
(event-send-to-egate string)
```

Description

event-send-to-egate sends data that the e*Way has already received from the external system into the e*Gate system as an Event.

Parameters

Name	Type	Description
string	string	The data to be sent to the e*Gate system

Return Values

Boolean
Returns **#t** (true) if the data is sent successfully; otherwise, returns **#f** (false).

Throws

None.

Additional information

This function can be called by any e*Way function when it is necessary to send data to the e*Gate system in a blocking fashion.

shutdown-request

Syntax

```
(shutdown-request)
```

Description

shutdown-request completes the e*Gate shutdown procedure that was initiated by the Control Broker but was interrupted by returning a non-null value within the Shutdown Command Notification Function (see **"Shutdown Command Notification Function" on page 151**). Once this function is called, shutdown proceeds immediately.

Once interrupted, the e*Way's shutdown cannot proceed until this Monk function is called. If you do interrupt an e*Way shutdown, we recommend that you complete the process in a timely fashion.

Parameters

None.

Return Values

None.

Throws

None.

8.2 Standard e*Way Functions

Note: *The functions described in this section can only be used by the functions defined within the e*Way's configuration file. None of the functions are available to Collaboration Rules scripts executed by the e*Way.*

The current suite of Java Monk Extension e*Way standard functions are:

java-ack

Syntax

```
(java-ack message-string)
```

Description

java-ack is used to send a positive acknowledgment to the external system, and for post processing after successfully sending data to e*Gate.

Parameters

Name	Type	Description
message-string	string	The Event for which an acknowledgment is sent.

Return Values

string
Returns one of the following strings:

- An empty string indicates a successful operation. The e*Way will then be able to proceed with the next request.

- "CONNERR" indicates a problem with the connection to the external system. When the connection is re-established, the function will be called again.

Additional Information

See **"Positive Acknowledgment Function" on page 149** for more information.

java-exchange

Syntax

```
(java-exchange)
```

Description

java-exchange is used for sending a received Event from the external system to e*Gate. The function expects no input.

Parameters

None.

Return Values

string
Returns one of the following strings:

- An empty string indicates a successful operation. Nothing is sent to e*Gate.

- A string containing Event data indicates successful operation, and the returned Event is sent to e*Gate.

- "CONNERR" indicates a problem with the connection to the external system. When the connection is re-established this function will be re-executed with the same input Event.

Throws

None.

Additional Information

See **"Exchange Data with External Function" on page 147** for more information.

java-extconnect

Syntax

```
(java-extconnect)
```

Description

java-extconnect is used to establish external system connection.

Parameters

None.

Return Values

string
"UP" indicates the connection is established. Anything else indicates no connection.

Throws

None.

Additional Information

See **"External Connection Establishment Function" on page 148** for more information.

java-init

Syntax

```
(java-init)
```

Description

java-init begins the initialization process for the e*Way. This function loads the **stc_monkjava.dll** or **stc_monkjava2.dll** file, based on the user's choice of **JVMVersion** setting, thereby making the function scripts available for future use.

Parameters

None.

Return Values

string
If a "FAILURE" string is returned, the e*Way will shutdown. Any other return value indicates success.

Throws

None.

Additional Information

Within this function, any necessary global variables to be used by the function scripts could be defined. The internal function that loads this file is called once when the e*Way first starts up.

java-nack

Syntax

```
(java-nack message-string)
```

Description

java-nack is used to send a negative acknowledgment to the external system, and for post processing after failing to send data to e*Gate.

Parameters

Name	Type	Description
message-string	string	The Event for which a negative acknowledgment is sent.

Return Values

string
    Returns one of the following strings:

- An empty string indicates a successful operation.
- "CONNERR" indicates a problem with the connection to the external system. When the connection is re-established, the function will be called again.

Throws

    None.

Additional Information

    See **"Negative Acknowledgment Function" on page 150** for more information.

java-notify

Syntax

```
(java-notify)
```

Description

    java-notify notifies the external system that the e*Way is shutting down.

Parameters

Name	Type	Description
command	string	When the e*Way calls this function, it will pass the string "SHUTDOWN_NOTIFICATION" as the parameter.

Return Values

string
    Returns a null string.

Throws

    None.

Additional Information

    See **"Shutdown Command Notification Function" on page 151** for more information.

java-outgoing

Syntax

```
(java-outgoing event-string)
```

Description

java-outgoing is used for sending a received message from e*Gate to the external system.

Parameters

Name	Type	Description
event-string	string	The Event to be processed.

Return Values

string

Returns one of the following strings:

- An empty string indicates a successful operation.

- "RESEND" causes the Event to be immediately resent.

- "CONNERR" indicates a problem with the connection to the external system. When the connection is re-established this function will be re-executed with the same input Event.

- "DATAERR" indicates the function had a problem processing data. If the e*Gate journal is enabled, the Event is journaled and the failed Event count is increased. (The input Event is essentially skipped in this process.) Use the **event-send-to-egate** function to place bad events in a bad event queue. See **event-send-to-egate** on page 100 for more information.

Throws

None.

Additional Information

See **"Process Outgoing Message Function" on page 147** for more information.

java-shutdown

Syntax

```
(java-shutdown shutdown)
```

Description

java-shutdown requests that the external connection shut down. A return value of "SUCCESS" indicates that the shutdown can occur immediately. Any other return value indicates that the shutdown Event must be delayed. The user is then required to execute a (**"shutdown-request" on page 101**) call from within a Monk function to allow the requested shutdown to process to continue.

Parameters

Name	Type	Description
shutdown	string	When the e*Way calls this function, it will pass the string "SUSPEND_NOTIFICATION" as the parameter.

Return Values

string
"SUCCESS" allows an immediate shutdown to occur. Any other return value causes the e*Way to delay shutdown until the **shutdown-request** function is executed successfully.

Throws

None.

Additional Information

See **"External Connection Shutdown Function" on page 149** for more information.

java-startup

Syntax

```
(java-startup)
```

Description

java-startup is used for function loads that are specific to this e*Way and invokes startup.

Parameters

None.

Return Values

string
"FAILURE" causes shutdown of the e*Way. Any other return value indicates success.

Throws

None.

Additional Information

This function should be used to initialize the external system before data exchange starts. Any additional variables may be defined here.

See **"Startup Function" on page 146** for more information.

java-verify

Syntax

```
(java-verify)
```

Description

java-verify is used to verify whether the connection to the external system is established.

Parameters

None.

Return Values

string

"UP" if the connection is established. Any other return value indicates that the connection is not established.

Throws

None.

Additional Information

See **"External Connection Verification Function" on page 149** for more information.

8.3   Java Monk Extension e*Way Native Functions

The Java Monk Extension Native functions adhere to the following rules as they pertain to Java.

8.3.1. Accessing Java Methods

Before accessing the Java Monk Extension e*Way native functions it is important to discuss issues relevant to accessing the Java Virtual Machine and its methods.

Within Java, it is possible for two methods to be referred to by the same name, while each takes different parameters. Several of the following Monk APIs take references to specified Java method names as arguments; they also take a vector of arguments that refer to the type and number of arguments, and the expected return value of the specified method. This reference (also known as a *signature*) provides a way of identifying Java methods and data fields using a character string. The signatures used within the following APIs are the same signatures used within the JNI (Java Native Interface). The *Java Native Interface* further discusses the definition of a signature.

8.3.2. Java Data Types

Within the JVM, the various data types are defined to accommodate different platforms. Java has two data types: *primitive* and *reference*. Primitive types are either

numeric or **boolean**. The numeric types are **byte**, **short**, **int**, **long**, **char**, **float** and **double**. The following table describes the format for these data types.

Figure 15   Size of Java Primitive Data Types

Type Name	Description
btye	8-bit two's-complement
short	16-bit two's-complement
int	32-bit two's-complement
long	64-bit two's-complement
float	32-bit IEEE 754 floating point
double	64-bit IEEE 754 floating point
char	16-bit Unicode

The *Java Language Specification* defines the primitive type **boolean** as either **true** or **false**.

Reference types specify either a `class`, an `interface`, or an array. The *Java Language Specification* refers to these as pointers. The term "pointer" in Java refers to an **object reference**, whereas in C/C++, pointers are addresses.

8.3.3. Type Signatures

Contained within the JNI type signature is a character string description of the parameter. A signature identifies information about the number and type of arguments to the method and the type of return value. Within a signature, the encoding of the arguments appears within the parentheses. For example:

```
java string methodName (long l, string s, boolean b)
        "(JLjava/lang/string;z)Ljava/lang/string;"
```

8.3.4. Method Signatures

Contained within the JNI method signature is a character string description of the formal parameters to a method and its return value. A method signature identifies information about the number and type of arguments to the method and the type of its return value. Within a signature, the encoding of the arguments appears within the parentheses. For example:

```
(<sigtype-list>)<return-sigtype> methodsignature
```

Figure 16 defines the output (return) types for use in signature encoding.

Figure 16   Signature Encoding

Signature	Description
B	byte
C	char
D	double

Signature	Description
F	float
I	int
J	long
S	short
V	void
Z	boolean
L<fully-qualified-class>;	fully qualified class

Note: *It is important to note that the type signature encoding for Java long is 'J' not 'L'.*
Inserting the incorrect signature is a common error.
V (void) can only be used for return values and not input parameters.
*When passing an object as a parameter, it is important **not** to neglect to include the*
'L' at the beginning of the fully-qualified class and the semi-colon (;) at the end of the
fully-qualified class.

Signature and Constructors

Type signatures for constructors follow the same rules as those for instance or class
methods. Constructors type signatures must include a v, representing void, as the return
value. For example:

```
java myConstructor (String s)
                "(Ljava/lang/string;)V"
```

The following functions are native to the Java Monk Extension e*Way.

java-call-method

Syntax

```
(java-call-method hJavaObj sMethodName)
```

Description

java-call-method invokes the specified method for the Java object.

Parameters

Name	Type	Description
hObj	opaque handle	The handle for an object
sMethodName	string	The name of the method to invoke.

Return Values

Boolean
Returns **#t** (true) if successful; otherwise, returns **#f** (false).

Throws

None.

Additional Information

The Java method defined as a parameters in this API, takes no input and has a return type of void.

java-call-method-with-params

Syntax

```
(java-call-method-with-params hObj sMethodName pszMethodSignature
vector_of_parameters)
```

Description

java-call-method-with-params invokes the specified method for the Java object.

Parameters

Name	Type	Description
hObj	opaque handle	The handle for an object
sMethodName	string	The name of the method to invoke.
pszMethodSignature	string	The string containing the formal parameters to the specified method and its return value.
vector_of_parameters	vector	An unspecified number of parameters, dependant on the parameters expected by the desired method.

Return Values

vector
Returns a vector that contains the return value and the signature to that return.

Throws

None.

Additional Information

To avoid memory leakage, **java-call-method-with-params** (if return string object) should be used as a pair with **java-get-string-value**:

```
(set! buf (java-call-method-with-params hCLASS "callYantra3" "(Ljava/
lang/String;Ljava/util/Vector;)Ljava/lang/String;" args2))
(set! strvalres2 (java-get-string-value (vector-ref buf 1)))
```

Note: *java-get-string-value can only be called once after the method call java-call-method-with-params.*

---

java-call-static-class-method-with-params

Syntax

```
(java-call-static-class-method-with-params hJVM hJavaObj sMethodName
pszMethodSignature vector_of_parameters)
```

Description

java-call-static-class-method-with-params invokes the specified static method for the Java object.

Parameters

Name	Type	Description
hJVM	opaque handle	The handle for a JVM
hJavaObj	opaque handle	The handle to the Java Object
sMethodName	string	The name of the method to invoke

Name	Type	Description
pszMethodSignature	string	The string containing the formal parameters to the specified method and its return value.
vector_of_parameters	vector	An unspecified number of parameters, dependant on the parameters expected by the desired method.

Return Values

vector

Returns a vector that contains return value and its signature.

Throws

None.

Note: *In calling this method, it is not necessary to precede the fully-qualified class with 'L'*
nor to end with the ';' (semi-colon). For example:

```
(java-call-static-class-method-with-params hJVM hJavaObj "currentTimeMillis" "()J" '#())
```

java-call-static-method-with-params

Syntax

```
(java-call-static-method-with-params hJVM sClassName sMethodName
pszMethodSignature vector_of_parameters)
```

Description

java-call-static-method-with-params invokes the specified static method for the
specified Java class.

Parameters

Name	Type	Description
hJVM	opaque handle	The handle for a JVM
sClassName	string	The fully-qualified class name
sMethodName	string	The name of the method to invoke
pszMethodSignature	string	The string containing the formal parameters to the specified method and its return value.
vector_of_parameters	vector	An unspecified number of parameters, dependant on the parameters expected by the desired method.

Return Values

vector

Returns a vector that contains return value and its signature.

Throws

None.

Note: *In calling this method, it is not necessary to precede the fully-qualified class with 'L'*
nor to end with the ';' (semi-colon). For example:

```
(java-call-static-method-with-params hJVM "java/lang/System" "currentTimeMillis" "()J" '#())
```

java-call-method-with-1-int-param

Syntax

```
(java-call-method-with-1-int-param hObj sMethodName iValue)
```

Description

java-call-method-with-1-int-param invokes a Java method, passing to it the specified
integer value as a parameter.

Parameters

Name	Type	Description
hObj	opaque handle	The handle for an object
sMethodName	string	The name of the method to invoke.
iValue	integer	The integer to be passed into the method invocation.

Return Values

Boolean

Returns **#t** (true) if successful, otherwise, returns **#f** (false).

Throws

None.

Additional Information

The Java method being called or invoked as a parameter in this Monk function, must
indicate a void (V) as it's return type. For example:

```
java
void someMethod (int i)
```

java-call-method-with-1-double-param

Syntax

```
(java-call-method-with-1-double-param hObj sMethodName dValue)
```

Description

java-call-method-with-1-double-param invokes a Java method, passing to it the specified double value as a parameter.

Parameters

Name	Type	Description
hObj	opaque handle	The handle to an object
sMethodName	string	The name of the method to invoke.
dValue	double	The double passed into the method invocation.

Return Values

Boolean
Returns **#t** (true) if successful; otherwise, returns **#f** (false).

Throws

None.

Additional Information

The Java method being called or invoked as a parameter in this Monk function must indicate a void (V) as its return type.

java-call-method-with-1-string-param

Syntax

```
(java-call-method-with-1-string-param hObj sMethodName sValue)
```

Description

java-call-method-with-1-string-param invokes a Java method, passing to it the specified string as a parameter.

Parameters

Name	Type	Description
hObj	opaque handle	The object handle
sMethodName	string	The name of the method to invoke.
sValue	string	The string passed into the method invocation.

Return Values

Boolean

Returns **#t** (true) if successful; otherwise, returns **#f** (false).

Throws

None.

Additional Information

The Java method being called or invoked as a parameter in this Monk function must indicate a void (V) as its return type.

---

java-call-method-with-1-object-param

Syntax

```
(java-call-method-with-1-object-param hObj sMethodName hObj
sClassName)
```

Description

java-call-method-with-1-object param passes the specified object to the Java method at invocation.

Parameters

Name	Type	Description
hObj	opaque handle	The handle of an object
sMethodName	string	The name of the method to invoke.
hObj	opaque handle	The handle to the object being passed into the method invocation.
sClassName	string	The fully qualified class name of the object being passed in.

Return Values

Boolean

Returns **#t** (true) if successful; otherwise, returns **#f** (false).

Throws

None.

Additional Information

The Java method being called or invoked as a parameter in this Monk function must indicate a void (V) as its return type.

java-call-method-with-int-return

Syntax

```
(java-call-method-with-int-return hObj sMethodName)
```

Description

java-call-method-with-int-return invokes the specified Java method and returns an integer.

Parameters

Name	Type	Description
hObj	opaque handle	The handle to an object
sMethodName	string	The name of the method to invoke.

Return Values

integer
Returns an integer from the method invoked.

Throws

None.

Additional Information

The Java method being called as a parameter must take no input.

java-call-method-with-double-return

Syntax

```
(java-call-method-with-double-return hObj sMethodName )
```

Description

java-call-method-with-double-return invokes the specified Java method and returns a double.

Parameters

Name	Type	Description
hObj	opaque handle	The handle to an object
sMethodName	string	The name of the method to invoke.

Return Values

Double
Returns the double from the method invoked.

Throws

None.

Additional Information

The Java method being called as a parameter must take no input.

java-call-method-with-string-return

Syntax

```
(java-call-method-with-string-return hObj sMethodName)
```

Description

java-call-method-with-string-return invokes the specified Java method and returns a string.

Parameters

Name	Type	Description
hObj	opaque handle	The handle to an object
sMethodName	string	The name of the metod to invoke.

Return Values

string
Returns a string from the method invoked.

Throws

None.

Additional Information

The Java method being called as a parameter must take no input.

java-call-method-with-object-return

Syntax

```
(java-call-method-with-object-return hObj sMethodName sClassName)
```

Description

java-call-method-with-object-return invokes the specified Java method and returns an object.

Parameters

Name	Type	Description
hObj	opaque handle	The handle to an object
sMethodName	string	The name of the method to invoke.

Name	Type	Description
sClassName	string	The fully qualified class name of the object being returned.

Return Values

object
Returns an object from the method called.

Throws

None.

Additional Information

The Java method being called as a parameter must take no input.

java-create-vm

Syntax

```
(java-create-vm sClasspath)
```

Description

java-create-vm instantiates a new instance of the Java virtual machine and returns the handle to that machine each time it is called.

Parameters

Name	Type	Description
sClasspath	string	This parameter is prepended to the environment variable "CLASSPATH" used by the JVM.

Return Values

handle
Returns the opaque handle to the Java virtual machine.

Throws

None.

Additional Information

If a Java vector is passed or retrieved using these methods, you must pass in the string "java/util/Vector" as the fully qualified class name.

Note: *The fully qualified name is case sensitive and it must use the '/' character as the separator instead of the usual '.' character. Any ClassPath defined must end with platform-specific CLASSPATH separators. For example, under DOS, use semi-colons (;), and under UNIX, use colons (:). If the classpath is not defined correctly the e*Way will fail.*

The Java virtual machine must be created by calling **java-create-vm** before any other methods may be called; otherwise, an error will occur.

java-create-vm-with-parameters

Syntax

```
(java-create-vm-with-parameters iVersion sClasspath iStackSize
iJavaStackSize iMinHeap iMaxHeap fVerboseGC fClassGC fDisableAsyncGC
fVerbose)
```

Description

java-create-vm-with-parameters instantiates a new instance of the JVM, returns a handle to the JVM, and allows the user to override the virtual machine's default parameters. If this method is called all of the parameters **must** be defined.

Parameters

Name	Type	Description All values are entered as "bytes".
iVersion	integer	Specifies the version of the JVM that you want to run. The default is 1.1.*(see **Note** at end of section)
sClasspath	string	This parameter is prepended to the environment variable "CLASSPATH" used by the JVM.
iStackSize	integer	Specifies the maximum stack size in bytes for native threads. The default is 128KB.
iJavaStackSize	integer	Specifies the maximum stack size in bytes for any JVM thread. The default is 400KB.
iMinHeap	integer	Specifies the initial heap size in bytes for the virtual machine. The default is 1024 KB.
iMaxHeap	integer	Specifies the maximum heap size in bytes for the virtual machine. The default is 16384KB.
fVerboseGC	TRUE or FALSE	Specifies whether to turn on/off reporting of garbage collection activity. The default is FALSE.
fClassGC	TRUE or FALSE	Specifies whether to turn on/off class garbage collection. The default is TRUE.
fDisableAsynGC	TRUE or FALSE	Specifies whether to disable asynchronous garbage collection. Passing TRUE will disable, the default is FALSE.

Name	Type	**Description** **All values are entered as "bytes".**
fVerbose	TRUE or FALSE	Specifies whether to turn on/off reporting of classes loaded by the virtual machine. The default is FALSE.

Return Values

handle
Returns the opaque handle to the Java virtual machine.

Throws

None.

Additional Information

The number before the decimal point in a program version (i.e., 1.2) indicates the major change in a program and is referred to as the major number. The release number to the right of the decimal point indicates a minor change and is referred to as the minor number.

This value (**iVersion**) encodes the major version of the virtual machine in the first 16 bytes of the integer, and stores the minor version in the lower 16 bytes. For example, to run a virtual machine with

version 1.0, the value of iVersion would be 0x00010000
version 1.1 would be 0x00010001 and
version 2.0 would be 0x00020000.

Note: *Any ClassPath defined must end with platform specific CLASSPATH separators. For example, under DOS, use semi-colons (;), and under UNIX, use colons (:). If the classpath is incorrectly defined, the e*Way will fail.*

To avoid memory leakage, java-create-vm-with-parameters should be used as a pair with java-destroy-vm:

```
(set!hJVM (java-creat-vm-with-parameters 131072 ".;" 131072 524288
8000000 16000000 #f #t #f #f))
;;use the JVM
(java-destroy-vm hJVM)
```

Note: *Since current JDK 1.1 and JDK 1.2 don't completely support the DestroyJavaVM() in JNI, it is recommended that these two APIs not be put into a loop. For example, java-create-vm-with-parameters should be called when the e*Way is up, and the java-destroy-vm should be called when the e*Way is shut down.*

java-create-class-instance

Syntax

```
(java-create-class-instance hJVM sClassName)
```

Description

java-create-class-instance instantiates the Java object and returns the handle to the object created.

Parameters

Name	Type	Description
hJVM	opaque handle	The handle a Java virtual machine returned by java-create-vm.
sClassName	string	The fully qualified class name to create.

Return Values

vector
Returns a vector that contains the return value and its signature.

Throws

None.

Additional Information

The fully qualified class name refers to the package name as well as the class name of a Java class. As an example, if a Java Vector is passed or retrieved using these methods, you must pass in the string "java/util/Vector" as the fully qualified class name.

Note: The fully qualified name is case sensitive and it must use the '/' character as the separator instead of the usual '.' character.

The Java virtual machine must be created by calling **java-create-vm** before any other methods may be called, otherwise, an error will occur.

Note: In creating the class instance, it is not necessary to begin the string with 'L' nor to end it with the ';' (semi-colon). For example:

```
(java-create-class-instance hJVM "java/lang/String")
```

To avoid memory leakage, **java-create-class-instance** should be used as a pair with **java-destroy-class-instance**:

```
(define myVec2 (java-create-class-instance hJVM "java/util/Vectory"))
;;usage of the vector class
(java-destroy-class-instance myVec2)
```

java-create-class-instance-with-params

Syntax

```
(java-create-class-instance-with-params hJVM sClassName
pszConstructorSignature vector_of_Parameters)
```

Description

> **java-create-class-instance-with-params** instantiates the Java object and returns the handle to object created.

Parameters

Name	Type	Description
hJVM	opaque handle	The handle a Java virtual machine returned by java-create-vm.
sClassName	string	The fully qualified class name to create. (see note below)
pszConstructorSignature	string	The string containing the specified signature of the constructor.
vector_of_parameters	vector	An unspecified number of parameters, dependant on the parameters expected by the desired method.

Return Values

vector
> Returns a vector that contains the return value and its signature.

Throws

> None.

Additional Information

> The fully qualified class name refers to the package name as well as the class name of a Java class. As an example, if a Java Vector is passed or retrieved using these methods, you must pass in the string "java/util/Vector" as the fully qualified class name.

Note: *The fully qualified name is case sensitive and it must use the'/' character as the separator instead of the usual '.' character.*

> The Java virtual machine must be created by calling **java-create-vm** before any other methods may be called, otherwise, an error will occur.

Note: *In creating the class instance, it is not necessary to begin the fully-qualified class with 'L' nor to end with the ';' (semi-colon). For example:*

```
(java-create-class-instance hJVM "java/util/Vector" '#(10))
```

java-create-string

Syntax

```
(java-create-string hJVM pszString)
```

Description

java-create-string creates a Java string and returns a handle to that string.

Parameters

Name	Type	Description
hJVM	opaque handle	The handle to the JVM.
pszString	string	The text string to be created in Java.

Return Values

handle
Returns an handle to the Java string.

Throws

None.

Additional Information

To avoid memory leakage, java-create-string should be used as a pair with java-release-string:

```
(define s2 (java-create-string hJVM "SHIP_ADVICE"))
(java-call-method-with-1-object-param myVec2 "addElement" s2 "java/
lang/Object")
(java-call-method-with-params-myVec2 "removeAllElements" "()V"
(quote#()))
(java-release-string hJVM s2)
```

*Note: **java-release-string** must be called after the method call "removeAllElements" of
java.util.Vector class, since it is still used as one of the elements inside the Vector at
that time.*

java-destroy-class-instance

Syntax

```
(java-destroy-class-instance hJavaObj)
```

Description

java-destroy-class-instance de-references the object reference.

Parameters

Name	Type	Description
hJavaObj	opaque handle	The handle for an object

Return Values

Boolean
Returns **#t** (true) if successful; otherwise, returns **#f** (false).

Throws

None.

java-destroy-vm

Syntax

```
(java-destroy-vm hJVM)
```

Description

java-destroy-vm destroys the JVM and releases the handle associated with that JVM. Before calling **java-destroy-vm**, all user threads pertinent to this JVM must be destroyed.

Parameters

Name	Type	Description
hJVM	opaque handle	The handle to the Java virtual machine to be destroyed.

Return Values

Boolean
Returns **#t** (true) if successful; otherwise, returns **#f** (false).

Throws

None.

Additional Information

To avoid memory leakage, **java-create-vm-with-parameters** should be used as a pair with **java-destroy-vm**:

```
(set!hJVM (java-creat-vm-with-parameters 131072 ".;" 131072 524288
8000000 16000000 #f #t #f #f))
;;use the JVM
(java-destroy-vm hJVM)
```

Note: *Since current JDK 1.1 and JDK 1.2 don't completely support the DestroyJavaVM() in JNI, it is recommended that these two APIs not be put into a loop. For example, java-create-vm-with-parameters should be called when the e*Way is up, and the java-destroy-vm should be called when the e*Way is shut down.*

java-get-property

Syntax

```
(java-get-property hJavaObj sPropertyName sPropertySignature )
```

Description

> **java-get-property** retrieves the specified property name for the corresponding Java object.

Parameters

Name	Type	Description
hJavaObj	opaque handle	The handle to an object
sPropertyName	string	The name of the property to get.
sPropertySignature	string	The string containing the signatures to the property being retrieved.

Return Values

vector
> Returns the vector that contains the signature of the return value and the return value.

Throws

> None.

java-get-property-int

Syntax

```
(java-get-property-int hObj sPropertyName)
```

Description

> **java-get-property-int** retrieves the integer associated with the specified property name.

Parameters

Name	Type	Description
hObj	opaque handle	The handle to an object
sPropertyName	string	The name of the property to get.

Return Values

integer
> Returns the integer associated with the property name.

Throws

> None.

java-get-property-string

Syntax

```
(java-get-property-string hObj sPropertyName)
```

Description

java-get-property-string retrieves the string associated with the specified property name.

Parameters

Name	Type	Description
hObj	opaque handle	The handle of an object
sPropertyName	string	The name of the property to get.

Return Values

string
Returns the string associated with the property name.

Throws

None.

java-get-property-object

Syntax

```
(java-get-property-object hObj sPropertyName sClassName)
```

Description

java-get-property-object retrieves the property associated with the specified object.

Parameters

Name	Type	Description
hObj	opaque handle	The handle of an object
sPropertyName	string	The name of the property to get.
sClassName	string	The fully qualified class name of the specified object.

Return Values

handle
Returns the opaque handle to the retrieved object.

Throws

None.

java-get-static-property

Syntax

```
(java-get-static-property hJavaObj sPropertyName sPropertySignature)
```

Description

> **java-get-static-property** retrieves the specified static property name for the corresponding Java object.

Parameters

Name	Type	Description
hJavaObj	opaque handle	The handle to an object
sPropertyName	string	The name of the property to get.
sPropertySignature	string	The string containing the signatures to the static property being retrieved.

Return Values

vector
> Returns the vector that contains the signature of the return value and the return value.

Throws

> None.

java-get-string-value

Syntax

```
(java-get-string-value hJavaStringObj)
```

Description

> **java-get-string-value** returns a string containing the value from a specified Java string.

Parameters

Name	Type	Description
hJavaStringObj	opaque handle	The handle to a Java object.

Return Values

string
> Returns string containing the value of the specified Java string.

Throws

> None.

Additional Information

> To avoid memory leakage, **java-call-method-with-params** (if return string object) should be used as a pair with **java-get-string-value**:

```
(set! buf (java-call-method-with-params hCLASS "callYantra3" "(Ljava/
lang/String;Ljava/util/Vector;)Ljava/lang/String;" args2))
```

```
(set! strvalres2 (java-get-string-value (vector-ref buf 1)))
```

Note: *java-get-string-value can only be called once after the method call java-call-method-*
with-params.

java-release-string

Syntax

```
(java-release-string hJVM hObj)
```

Description

java-release-string releases the resource associated with the string object during
creation time.

Parameters

Name	Type	Description
hJVM	opaque handle	The handle to the JVM
hObj	opaque handle	The handle to the string object

Return Values

Boolean
Returns **#t** (true) if successful; otherwise, returns **#f** (false).

Throws

None.

Additional Information

java-release-string must be called after the created string is no longer needed or used.

To avoid memory leakage, java-create-string should be used as a pair with java-release-
string:

```
(define s2 (java-create-string hJVM "SHIP_ADVICE"))
(java-call-method-with-1-object-param myVec2 "addElement" s2 "java/
lang/Object")
(java-call-method-with-params-myVec2 "removeAllElements" "()V"
(quote#()))
(java-release-string hJVM s2)
```

Note: *java-release-string must be called after the method call "removeAllElements" of*
java.util.Vector class, since it is still used as one of the elements inside the Vector at
that time.

java-set-property

Syntax

```
(java-set-property hJavaObj pszPropertyName pszPropertySignature
vector_of_parameters)
```

Description

java-set-property sets the property of a specified object to a specific value.

Parameters

Name	Type	Description
hJavaObj	opaque handle	The handle of an object
pszPropertyName	string	The name of property to set
pszPropertySignature	string	The string containing the signatures to the property being set.
vector_of_parameters	vector	An unspecified number of parameters, dependant on the parameters expected by the desired method.

Return Values

Boolean
Returns **#t** (true) if successful; otherwise, returns **#f** (false).

Throws

None.

java-set-static-property

Syntax

```
(java-set-static-property hJavaObj pszPropertyName
pszPropertySignature vector_of_parameters)
```

Description

java-set-static-property sets the static property of a specified object to a specific value.

Parameters

Name	Type	Description
hJavaObj	opaque handle	The handle of an object
pszPropertyName	string	The name of property to set
pszPropertySignature	string	The string containing the signatures to the static property being set.
vector_of_parameters	string	An unspecified number of parameters, dependant on the parameters expected by the desired method.

Return Values

Boolean
Returns **#t** (true) if successful; otherwise, returns **#f** (false).

Throws

None.

java-set-property-int

Syntax

```
(java-set-property-int hObj sProperty iValue)
```

Description

java-set-property-int sets the property of a specified object to a specific integer value.

Parameters

Name	Type	Description
hObj	opaque handle	The handle of an object
sProperty	string	The name of property to set
iValue	integer	The integer to which the property is set.

Return Values

Boolean
Returns **#t** (true) if successful; otherwise, returns **#f** (false).

Throws

None.

java-set-property-string

Syntax

```
(java-set-property-string hObj sProperty sValue)
```

Description

java-set-property-string sets the property of a specified object to a specified string.

Parameters

Name	Type	Description
hObj	opaque handle	The handle of an object
sProperty	string	The name of the property to set.

Name	Type	Description
sValue	string	The string to which the property is set.

Return Values

Boolean
Returns **#t** (true) if successful; otherwise, returns **#f** (false).

Throws

None.

java-set-property-object

Syntax

```
(java-set-property-object hObj sProperty hPassedObjj sClassName)
```

Description

java-set-property-object assigns a handle associated with an object to the property of a specified object.

Parameters

Name	Type	Description
hObj	opaque handle	The handle of an object
sProperty	string	The name the property to set.
hPassedObj	opaque handle	The handle to the object being used as a reference.
sClassName	string	The fully qualified class name of the object being set.

Return Values

Boolean
Returns **#t** (true) if successful; otherwise, returns **#f** (false).

Throws

None.

Configuring the Java Monk Extension e*Way

This chapter describes how to configure the Java Monk Extension e*Way.

9.1 e*Way Configuration Parameters

e*Way configuration parameters are set using the e*Way Editor.

To change e*Way configuration parameters:

1 In the Schema Designer's Component editor, select the e*Way you want to configure and display its properties.

2 Under **Configuration File**, click **New** to create a new file, **Find** to select an existing configuration file, or **Edit** to edit the currently selected file.

3 In the **Additional Command Line Arguments** box, type any additional command line arguments that the e*Way may require, taking care to insert them *at the end* of the existing command-line string. Be careful not to change any of the default arguments unless you have a specific need to do so.

For more information about how to use the e*Way Editor, see the e*Way Editor's online Help or the *e*Gate Integrator User's Guide*.

The e*Way's configuration parameters are organized into the following sections:

- General Settings
- Communication Setup
- Monk Configuration
- Java VM Configuration

9.1.1. General Settings

The General Settings control basic operational parameters.

Journal File Name

Description

Specifies the name of the journal file.

Required Values

A valid filename, optionally including an absolute path (for example, **c:\temp\filename.txt**). If an absolute path is not specified, the file will be stored in the e*Gate "SystemData" directory. See the *e*Gate Integrator User's Guide* for more information about file locations.

Additional Information

An Event will be journaled for the following conditions:

- When the number of resends is exceeded (see Max Resends Per Message below)

- When its receipt is due to an external error, but Forward External Errors is set to **No**. (See **"Forward External Errors" on page 134** for more information.)

Max Resends Per Message

Description

Specifies the number of times the e*Way will attempt to resend a message (Event) to the external system after receiving an error. When this maximum is reached, the message is considered "Failed" and is written to the journal file.

Required Values

An integer between 1 and 1,024. The default is 5.

Max Failed Messages

Description

Specifies the maximum number of failed messages (Events) that the e*Way will allow. When the specified number of failed messages is reached, the e*Way will shut down and exit.

Required Values

An integer between 1 and 1,024. The default is 3.

Forward External Errors

Description

Selects whether error messages that begin with the string "DATAERR" that are received from the external system will be queued to the e*Way's configured queue. See **"Exchange Data with External Function" on page 147** for more information.

Required Values

Yes or **No**. The default value, **No**, specifies that error messages will not be forwarded.

See **"Schedule-driven Data Exchange Functions" on page 141** for information about how the e*Way uses this function.

9.1.2. Communication Setup

The Communication Setup parameters control the schedule by which the e*Way obtains data from the external system.

Note: *The schedule you set using the e*Way's properties in the Schema Designer controls when the e*Way executable will run. The schedule you set within the parameters discussed in this section (using the e*Way Editor) determines when data will be exchanged. Be sure you set the "exchange data" schedule to fall within the "run the executable" schedule.*

Start Exchange Data Schedule

Description

Establishes the schedule to invoke the e*Way's **Exchange Data with External** function.

Required Values

One of the following:

- One or more specific dates/times

- A single repeating interval (such as yearly, weekly, monthly, daily, or every *n* seconds).

Also required: If you set a schedule using this parameter, you must also define all three of the following:

- Exchange Data With External Function

- Positive Acknowledgment Function

- Negative Acknowledgment Function

If you do not do so, the e*Way will terminate execution when the schedule attempts to start.

Additional Information

When the schedule starts, the e*Way determines whether it is waiting to send an ACK or NAK to the external system (using the Positive and Negative Acknowledgment functions) and whether the connection to the external system is active. If no ACK/NAK is pending and the connection is active, the e*Way immediately executes the **Exchange Data with External** function. Thereafter, the **Exchange Data with External** function will be called according to the **Exchange Data Interval** parameter until the **Stop Exchange Data Schedule** time is reached.

See **"Exchange Data with External Function" on page 147**, **"Exchange Data Interval" on page 136**, and **"Stop Exchange Data Schedule" on page 135** for more information.

Stop Exchange Data Schedule

Description

Establishes the schedule to stop data exchange.

Required Values

One of the following:

- One or more specific dates/times
- A single repeating interval (such as yearly, weekly, monthly, daily, or every *n* seconds).

Exchange Data Interval

Description

Specifies the number of seconds the e*Way waits between calls to the **Exchange Data with External** function during scheduled data exchanges.

Required Values

An integer between 0 and 86,400. The default is 120.

Additional Information

If **Zero Wait Between Successful Exchanges** is set to **Yes** and the **Exchange Data with External Function** returns data, The **Exchange Data Interval** setting will be ignored and the e*Way will invoke the **Exchange Data with External Function** immediately.

If this parameter is set to zero, there will be no exchange data schedule set and the **Exchange Data with External Function** will never be called.

See **"Down Timeout" on page 136** and **"Stop Exchange Data Schedule" on page 135** for more information about the data-exchange schedule.

Down Timeout

Description

Specifies the number of seconds that the e*Way will wait between calls to the **External Connection Establishment** function. See **"External Connection Establishment Function" on page 148** for more information.

Required Values

An integer between 1 and 86,400. The default is 15.

Up Timeout

Description

Specifies the number of seconds the e*Way will wait between calls to the **External Connection Verification** function. See **"External Connection Verification Function" on page 149** for more information.

Required Values

An integer between 1 and 86,400. The default is 15.

Resend Timeout

Description

Specifies the number of seconds the e*Way will wait between attempts to resend a message (Event) to the external system, after receiving an error message from the external system.

Required Values

An integer between 1 and 86,400. The default is 10.

Zero Wait Between Successful Exchanges

Description

Selects whether to initiate data exchange after the **Exchange Data Interval** or immediately after a successful previous exchange.

Required Values

Yes or **No**. If this parameter is set to **Yes**, the e*Way will immediately invoke the **Exchange Data with External** function if the previous exchange function returned data. If this parameter is set to **No**, the e*Way will always wait the number of seconds specified by **Exchange Data Interval** between invocations of the **Exchange Data with External** function. The default is **No**.

See **"Exchange Data with External Function" on page 147** for more information.

9.1.3. Monk Configuration

The parameters in this section help you set up the information required by the e*Way to utilize Monk for communication with the external system.

Conceptually, an e*Way is divided into two halves. One half of the e*Way (shown on the left in Figure 17 below) handles communication with the external system; the other half manages the Collaborations that process data and subscribe or publish to other e*Gate components.

Figure 17 e*Way internal architecture
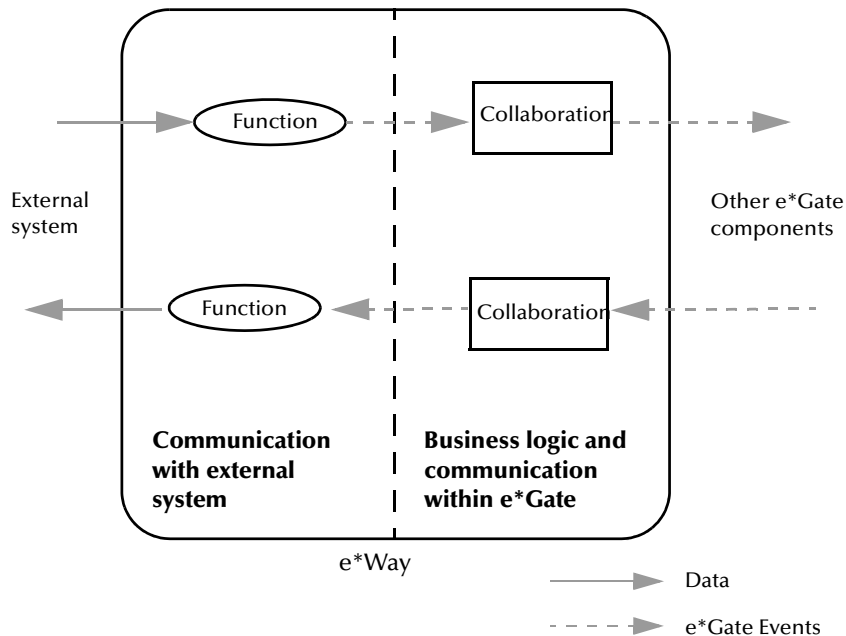
The "communications half" of the e*Way uses Monk functions to start and stop scheduled operations, exchange data with the external system, package data as e*Gate "Events" and send those Events to Collaborations, and manage the connection between the e*Way and the external system. The **Monk Configuration** options discussed in this section control the Monk environment and define the Monk functions used to perform these basic e*Way operations. You can create and modify these functions using the Collaboration Rules Editor or a text editor (such as **Notepad**, or UNIX **vi**).

The "communications half" of the e*Way is single-threaded. Functions run serially, and only one function can be executed at a time. The "business logic" side of the e*Way is multi-threaded, with one executable thread for each Collaboration. Each thread maintains its own Monk environment; therefore, information such as variables, functions, path information, and so on cannot be shared between threads.

Operational Details

The Monk functions in the "communications half" of the e*Way fall into the following groups:

Type of Operation	Name
Initialization	**Startup Function** on page 146 (also see **Monk Environment Initialization File** on page 145)
Connection	**External Connection Establishment Function** on page 148 **External Connection Verification Function** on page 149 **External Connection Shutdown Function** on page 149
Schedule-driven data exchange	**Exchange Data with External Function** on page 147 **Positive Acknowledgment Function** on page 149 **Negative Acknowledgment Function** on page 150
Shutdown	**Shutdown Command Notification Function** on page 151
Event-driven data exchange	**Process Outgoing Message Function** on page 147

A series of figures on the next several pages illustrate the interaction and operation of these functions.

Initialization Functions

Figure 18 illustrates how the e*Way executes its initialization functions.

Figure 18   Initialization Functions
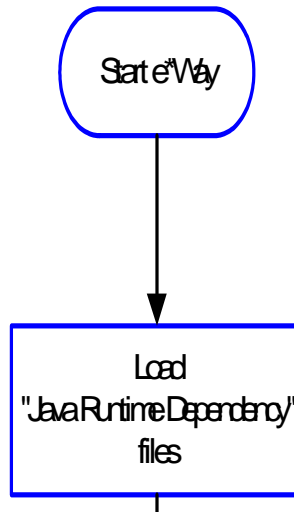
Connection Functions

Figure 19 illustrates how the e*Way executes the connection establishment and verification functions.

Figure 19   Connection establishment and verification functions
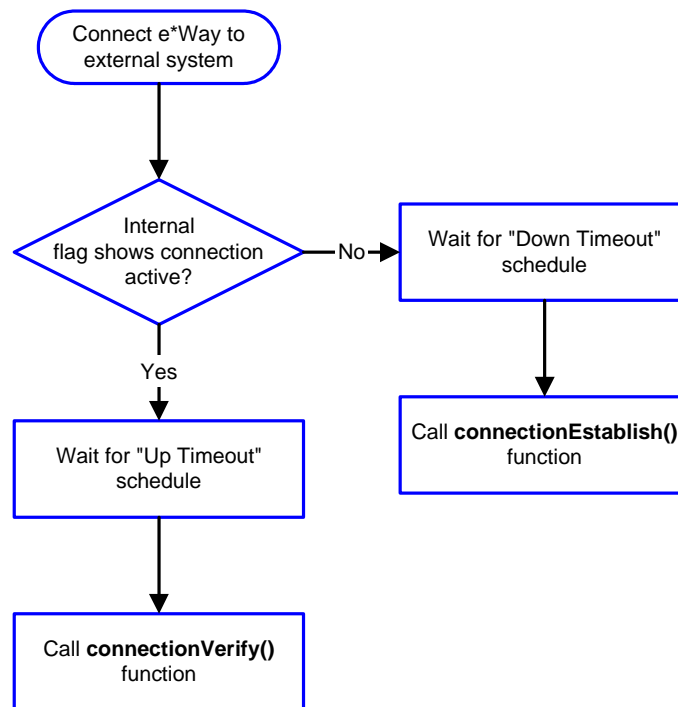
Note: *The e*Way selects the connection function based on an internal "up/down" flag rather than a poll to the external system. See* **Figure 21 on page 142** *and* **Figure 23 on page 144** *for examples of how different functions use this flag.*

User functions can manually set this flag using Monk functions. **send-external-up** *on page 99 See and* **send-external-down** *on page 100 for more information.*

Figure 20 illustrates how the e*Way executes its "connection shutdown" function.

Figure 20 Connection shutdown function

```
        ┌─────────────────────────┐
        │   Control Broker issues  │
        │    "Suspend" command     │
        └─────────────────────────┘
                     │
                     ▼
        ┌─────────────────────────┐
        │ Call connectionShutdown()│
        │    function              │
        │      with parameter      │
        │  "SUSPEND_NOTIFICATION"  │
        └─────────────────────────┘
                     │
              Return any value
                     │
                     ▼
        ┌─────────────────────────┐
        │  e*Way closes connection │
        └─────────────────────────┘
```
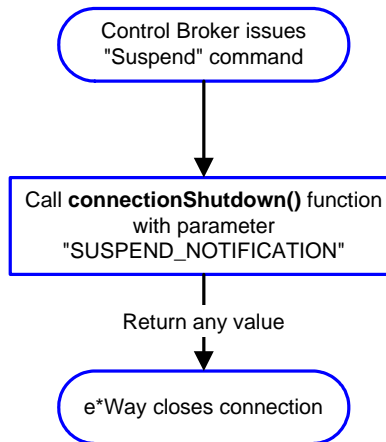
Schedule-driven Data Exchange Functions

Figure 21 (on the next page) illustrates how the e*Way performs schedule-driven data exchange using the **Exchange Data with External Function**. The **Positive Acknowledgment Function** and **Negative Acknowledgment Function** are also called during this process.

"Start" can occur in any of the following ways:

- The "Start Data Exchange" time occurs

- Periodically during data-exchange schedule (after "Start Data Exchange" time, but before "Stop Data Exchange" time), as set by the Exchange Data Interval

- The **start-schedule** Monk function is called

After the function exits, the e*Way waits for the next "start schedule" time or command.

Figure 21   Schedule-driven data exchange functions
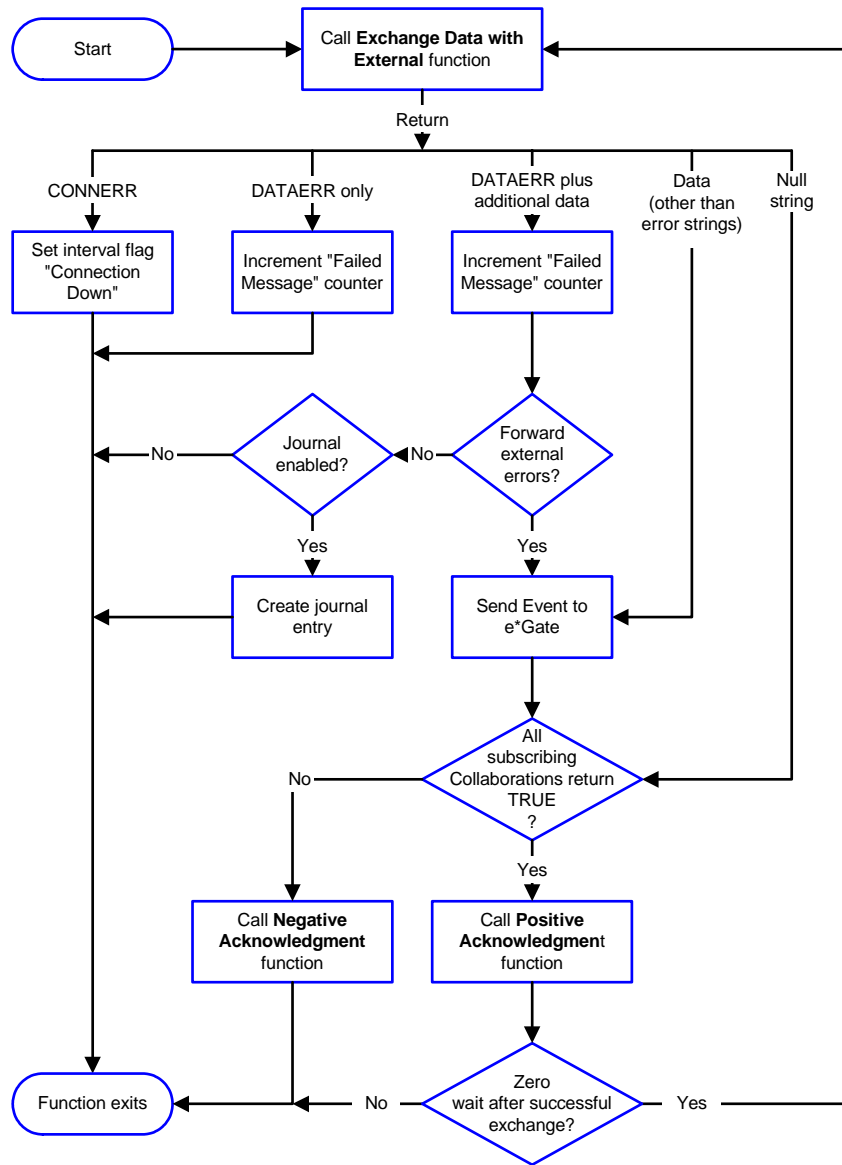
Shutdown Functions

Figure 22 illustrates how the e*Way implements the shutdown request function.

Figure 22   Shutdown functions
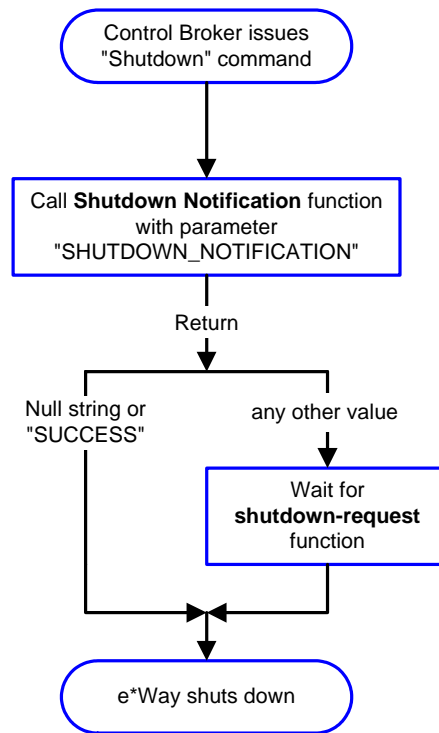
Event-driven Data Exchange Functions

Figure 23 on the next page illustrates event-driven data-exchange using the **Process Outgoing Message Function**.

Every two minutes, the e*Way checks the "Failed Message" counter against the value specified by the **Max Failed Messages** parameter. When the "Failed Message" counter exceeds the specified maximum value, the e*Way logs an error and shuts down.

After the function exits, the e*Way waits for the next outgoing Event.

Figure 23   Event-driven data-exchange functions
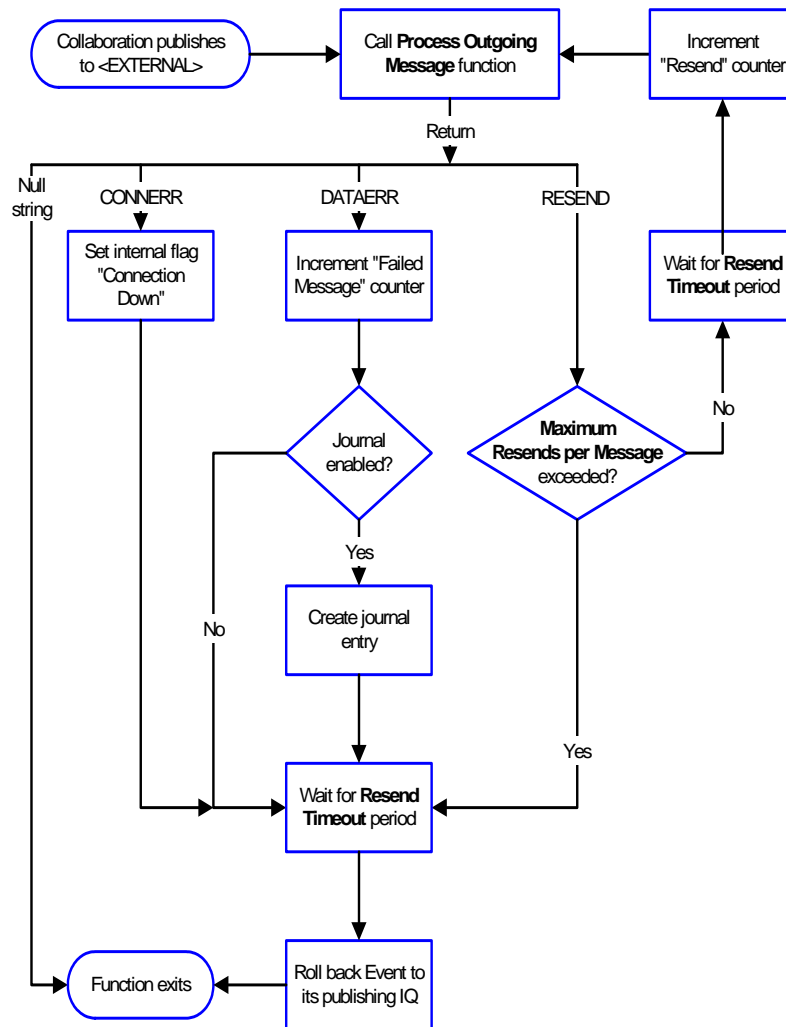
How to Specify Function Names or File Names

Parameters that require the name of a Monk function will accept either a function name or a file name. If you specify a file name, be sure that the file has one of the following extensions:

- .monk
- .tsc
- .dsc

Additional Path

Description

Specifies a path to be appended to the "load path," the path Monk uses to locate files and data (set internally within Monk). The directory specified in Additional Path will be searched after the default load paths.

Required Values

A pathname, or a series of paths separated by semicolons. This parameter is optional and may be left blank.

Additional information

The default load paths are determined by the "bin" and "Shared Data" settings in the .egate.store file. See the *e*Gate Integrator User's Guide* for more information about this file.

To specify multiple directories, manually enter the directory names rather than selecting them with the **Find File** selection button. Directory names must be separated with semicolons, and you can mix absolute paths with relative e*Gate paths. For example:

```
monk_scripts\my_dir;c:\my_directory
```

The internal e*Way function that loads this path information is called only once, when the e*Way first starts up.

Auxiliary Library Directories

Description

Specifies a path to auxiliary library directories. Any **.monk** files found within those directories will automatically be loaded into the e*Way's Monk environment. This parameter is optional and may be left blank.

Required Values

A pathname, or a series of paths separated by semicolons.

Additional information

To specify multiple directories, manually enter the directory names rather than selecting them with the **Find File** selection button. Directory names must be separated with semicolons, and you can mix absolute paths with relative e*Gate paths. For example:

```
monk_scripts\my_dir;c:\my_directory
```

The internal e*Way function that loads this path information is called only once, when the e*Way first starts up.

This parameter is optional and may be left blank.

Monk Environment Initialization File

Specifies a file that contains environment initialization functions, which will be loaded after the auxiliary library directories are loaded. Use this feature to initialize the

e*Way's Monk environment (for example, to define Monk variables that are used by the e*Way's function scripts).

Required Values

A filename within the "load path", or filename plus path information (relative or absolute). If path information is specified, that path will be appended to the "load path." See **"Additional Path" on page 145** for more information about the "load path." The default is **java-init.monk**

Additional information

Any environment-initialization functions called by this file accept no input, and must return a string. The e*Way will load this file and try to invoke a function of the same base name as the file name (for example, for a file named **my-init.monk**, the e*Way would attempt to execute the function **my-init**).

Typically, it is a good practice to initialize any global Monk variables that may be used by any other Monk Extension scripts.

The internal function that loads this file is called once when the e*Way first starts up (see **Figure 18 on page 140**).

Startup Function

Description

Specifies a Monk function that the e*Way will load and invoke upon startup or whenever the e*Way's configuration is reloaded. This function should be used to initialize the external system before data exchange starts.

Required Values

The name of a Monk function, or the name of a file (optionally including path information) containing a Monk function. This parameter is optional and may be left blank. The default is **java-startup** on page 107.

Additional information

The function accepts no input, and must return a string.

The string "FAILURE" indicates that the function failed; any other string (including a null string) indicates success.

This function will be called after the e*Way loads the specified "Monk Environment Initialization file" and any files within the specified **Auxiliary Directories**.

The e*Way will load this file and try to invoke a function of the same base name as the file name (see **Figure 18 on page 140**). For example, for a file named **my-startup.monk**, the e*Way would attempt to execute the function **my-startup**.

Process Outgoing Message Function

Description

Specifies the Monk function responsible for sending outgoing messages (Events) from the e*Way to the external system. This function is event-driven (unlike the **Exchange Data with External Function**, which is schedule-driven).

Required Values

The name of a Monk function, or the name of a file (optionally including path information) containing a Monk function. *You may not leave this field blank.* The default is **java-outgoing** on page 105.

Additional Information

The function requires a non-null string as input (the outgoing Event to be sent) and must return a string.

The e*Way invokes this function when one of its Collaborations publishes an Event to an <EXTERNAL> destination (as specified within the Schema Designer). The function returns one of the following (see **Figure 23 on page 144** for more details):

- Null string: Indicates that the Event was published successfully to the external system.
- "RESEND": Indicates that the Event should be resent.
- "CONNERR": Indicates that there is a problem communicating with the external system.
- "DATAERR": Indicates that there is a problem with the message (Event) data itself.
- If a string other than the following is returned, the e*Way will create an entry in the log file indicating that an attempt has been made to access an unsupported function.

*Note: If you wish to use **event-send-to-egate** to enqueue failed Events in a separate IQ, the e*Way must have an inbound Collaboration (with appropriate IQs) configured to process those Events. See **event-send-to-egate** on page 100 for more information.*

Exchange Data with External Function

Description

Specifies a Monk function that initiates the transmission of data from the external system to the e*Gate system and forwards that data as an inbound Event to one or more e*Gate Collaborations. This function is called according to a schedule (unlike the **Process Outgoing Message Function**, which is event-driven).

Required Values

The name of a Monk function, or the name of a file (optionally including path information) containing a Monk function. This parameter is optional and may be left blank. The default is **java-exchange** on page 103.

Additional Information

The function accepts no input and must return a string (see **Figure 21 on page 142** for more details):

- Null string: Indicates that the data exchange was completed successfully. No information will be sent into the e*Gate system.

- "CONNERR": Indicates that a problem with the connection to the external system has occurred.

- "DATAERR": Indicates that a problem with the data itself has occurred. The e*Way handles the string "DATAERR" and "DATAERR" plus additional data differently; see **Figure 21 on page 142** for more details.

- Any other string: The contents of the string are packaged as an inbound Event. The e*Way must have at least one Collaboration configured suitably to process the inbound Event, as well as any required IQs.

This function is initially triggered by the **Start Data Exchange** schedule or manually by the Monk function **start-schedule**. After the function has returned true and the data received by this function has been ACKed or NAKed (by the **Positive Acknowledgment Function** or **Negative Acknowledgment Function**, respectively), the e*Way checks the **Zero Wait Between Successful Exchanges** parameter. If this parameter is set to **Yes**, the e*Way will immediately call the **Exchange Data with External** function again; otherwise, the e*Way will not call the function until the next scheduled "start exchange" time or the schedule is manually invoked using the Monk function **start-schedule** (see **start-schedule** on page 98 for more information).

External Connection Establishment Function

Description

Specifies a Monk function that the e*Way will call when it has determined that the connection to the external system is down.

Required Values

The name of a Monk function, or the name of a file (optionally including path information) containing a Monk function. *This field cannot be left blank.* The default is **java-extconnect** on page 103.

Additional Information

The function accepts no input and must return a string:

- "SUCCESS" or "UP": Indicates that the connection was established successfully.

- Any other string (including the null string): Indicates that the attempt to establish the connection failed.

This function is executed according to the interval specified within the **Down Timeout** parameter, and is *only* called according to this schedule.

The **External Connection Verification** function (see below) is called when the e*Way has determined that its connection to the external system is up.

External Connection Verification Function

Description

Specifies a Monk function that the e*Way will call when its internal variables show that the connection to the external system is up.

Required Values

The name of a Monk function. This function is optional; if no **External Connection Verification** function is specified, the e*Way will execute the **External Connection Establishment** function in its place. The default is **java-verify** on page 108.

Additional Information

The function accepts no input and must return a string:

- "SUCCESS" or "UP": Indicates that the connection was established successfully.

- Any other string (including the null string): Indicates that the attempt to establish the connection failed.

This function is executed according to the interval specified within the **Up Timeout** parameter, and is *only* called according to this schedule.

The **External Connection Establishment** function (see above) is called when the e*Way has determined that its connection to the external system is down.

External Connection Shutdown Function

Description

Specifies a Monk function that the e*Way will call to shut down the connection to the external system.

Required Values

The name of a Monk function. This parameter is optional. The default is **java-shutdown** on page 106.

Additional Information

This function requires a string as input, and may return a string.

This function will only be invoked when the e*Way receives a "suspend" command from a Control Broker. When the "suspend" command is received, the e*Way will invoke this function, passing the string "SUSPEND_NOTIFICATION" as an argument.

Any return value indicates that the "suspend" command can proceed and that the connection to the external system can be broken immediately.

Positive Acknowledgment Function

Description

Specifies a Monk function that the e*Way will call when *all* the Collaborations to which the e*Way sent data have processed and enqueued that data successfully.

Required Values

The name of a Monk function, or the name of a file (optionally including path information) containing a Monk function. This parameter is required if the **Exchange Data with External** function is defined. The default is **java-ack** on page 102.

Additional Information

The function requires a non-null string as input (the Event to be sent to the external system) and must return a string:

- "CONNERR": Indicates a problem with the connection to the external system. When the connection is re-established, the Positive Acknowledgment function will be called again, with the same input data.

- Null string: The function completed execution successfully.

After the **Exchange Data with External** function returns a string that is transformed into an inbound Event, the Event is handed off to one or more Collaborations for further processing. If the Event's processing is completed successfully by *all* the Collaborations to which it was sent, the e*Way executes the Positive Acknowledgment function (otherwise, the e*Way executes the Negative Acknowledgment function).

Negative Acknowledgment Function

Description

Specifies a Monk function that the e*Way will call when the e*Way fails to process and queue Events from the external system.

Required Values

The name of a Monk function, or the name of a file (optionally including path information) containing a Monk function. This parameter is required if the **Exchange Data with External** function is defined. The default is **java-nack** on page 104.

Additional Information

The function requires a non-null string as input (the Event to be sent to the external system) and must return a string:

- "CONNERR": Indicates a problem with the connection to the external system. When the connection is re-established, the function will be called again.

- Null string: The function completed execution successfully.

This function is only called during the processing of inbound Events. After the **Exchange Data with External** function returns a string that is transformed into an inbound Event, the Event is handed off to one or more Collaborations for further processing. If the Event's processing is not completed successfully by *all* the Collaborations to which it was sent, the e*Way executes the Negative Acknowledgment function (otherwise, the e*Way executes the Positive Acknowledgment function).

Shutdown Command Notification Function

Description

Specifies a Monk function that will be called when the e*Way receives a "shutdown" command from the Control Broker. This parameter is optional.

Required Values

The name of a Monk function.

Additional Information

When the Control Broker issues a shutdown command to the e*Way, the e*Way will call this function with the string "SHUTDOWN_NOTIFICATION" passed as a parameter.

The function accepts a string as input and must return a string:

- A null string or "SUCCESS": Indicates that the shutdown can occur immediately.

- Any other string: Indicates that shutdown must be postponed. Once postponed, shutdown will not proceed until the Monk function **shutdown-request** is executed (see **shutdown-request** on page 101).

Note: *If you postpone a shutdown using this function, be sure to use the (**shutdown-request**) function to complete the process in a timely manner.*

9.1.4. Java VM Configuration

The parameters in this section specify the required information for the e*Way to configure the Java Virtual Machine.

JVMVersion

Description

Specifies the version of the JDK that is used.

Required Values

1 or **2**. If using JDK 1.1.7B, choose 1, if using JDK 1.2 or later, choose 2. This parameter is REQUIRED. Choose the Java Release you want this e*Way to use. If "Java 1" is chosen, some of the following parameters may not be pertinent, and their corresponding functionality will not be available. Although Java bytecodes for classes and methods, common to both Java 1 and Java 2, are typically forward and backward compatible, it is recommended that this e*Way be set to the Java release appropriate to the version of the "javac" compiler used to compile the Java source code that is used.

Note: *For Linux platform support, you must select Java 2. If the JVM version specified cannot be found in the library path, the e*Way fails.*

JVMClasspath

Description

Specifies the classpath to use for the Java Virtual Machine. *This field cannot be left blank.*

Required Values

A pathname, or a series of paths separated by semicolons.

Note: *For UNIX the paths are separated by a colon.*

Additional Information

If the classpath is incorrectly defined the e*Way will fail.

Native Stack Size

Description

Specifies the maximum stack size in bytes for any JVM thread used to store items which are retrieved in last-in first-out order (LIFO).

Required Values

An integer between a constant value of 2000 and a constant value of 1000000000. The default is 131072.

Additional Information

A stack can be used to keep track of the sequence of subroutines called in a program. Data is entered or retrieved by "pushing" a new item onto the stack or "popping" the top item off the stack.

Java Stack Size

Description

An integer between a constant value of 2000 and a constant value of 1,000,000,000. The default is 524,288.

Required Values

An integer between a constant 2000 and a constant value of

Initial Heap Size

Description

Specifies the initial size in bytes of the heap.

Required Values

An integer between a constant value of 2000 and a constant value of 1,000,000,000. The default is 4,194,304.

Additional Information

The heap acts as a common pool of free memory usable by a program. A part of the computer's memory used for dynamic memory allocation, in which blocks of memory are used in an arbitrary order.

Max Heap Size

Description

Specifies the maximum size in bytes of the heap.

Required Values

An integer between a constant value of 2000 and a constant value of 1,000,000,000. The default is 16,777,216.

Additional Information

The heap acts as a common pool of free memory usable by a program. A part of the computer's memory used for dynamic memory allocation, in which blocks of memory are used in an arbitrary order. If an OutOfMemory or StackOverflowError error occurs, the JVM will be destroyed. This can be prevented by increasing the maximum heap size.

Enable Class GC

Description

Specifies whether to "garbage collect" loaded classes that are no longer in use.

Required Values

Yes or **No**.

Additional Information

The garbage collector verifies that a given Java object will not be accessed again during the execution of the Java program. The garbage collector can free the memory consumed by that object for reuse. The programmer need not de-allocate objects explicitly.

Enable Verbose GC

Description

Specifies whether or not to enable the garbage collector to print a message whenever memory is freed.

Required Values

Yes or **No**.

Disable Async GC

Description

Specifies whether to "garbage collect" asynchronously.

Required Values

Yes or **No**.

Index