SeeBeyond ICAN Suite

Generic e*Way Extension Kit Developer's Guide

Release 5.0.5 for Schema Run-time Environment (SRE)

Monk Version

SeeBeyond®

Contents

Chapter 5

Interface API Functionality 80

Introduction

The *Generic e*Way Extension Kit Developer's Guide* describes how to extend the Generic e*Way for use with external applications using the Generic e*Way Extension Kit. The kit includes instructions on how to

- Use the External Interface (EI) to create a dynamic or shared library that allows the Generic e*Way access to external Application Programming Interfaces (APIs) written in C or C++.

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

- Use dynamic or shared libraries with the Generic e*Way executable (shipped with the base e*Gate product).

Note: *This dynamic or shared library created with the EI can also be used by a Business Object Broker (BOB). For more information on BOBs, see the e*Gate Schema Designer's online help.*

1.1 Overview

The External Interface (EI) is the programming interface which allows a dynamic or shared library that follows the EI protocol to have access to other libraries that follow the same protocol. Because the Monk engine follows this protocol, Monk scripts can run within a Generic e*Way to operate with applications and libraries written in other languages such as C and C++. Programmers use the EI to write native methods to handle those situations when an application cannot be written entirely in the Monk programming language.

For example, you may want to use native methods and the EI in the following situations:

- The standard Monk function library may not support the platform-dependent features needed by your application.

- You have a library or application written in another programming language and you want to make it accessible to Generic e*Way applications.

- You want to implement a small portion of time-critical code in a lower-level programming language, such as C, and have the Generic e*Way application call these functions.

Programming through the EI framework enables you to create a bridge between the e*Gate system and native applications, using native methods to perform a wide range of operations such as wrapping legacy applications or solving problems better handled outside the Monk programming environment.

1.1.1 Intended Audience

The intended audience for this document is experienced programmers who want to write e*Way interfaces using the Generic e*Way. We also recommend that the reader have a thorough understanding of the following:

- C and C++ programming languages.
- Windows NT operating system (as well as the UNIX operating system for UNIX applications).
- Basic knowledge of SeeBeyond's Monk programming language.
- External application for which the extension is to be written.

1.2 Generic e*Way Components

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

The Generic e*Way contains the following components:

- **stcewgenericmonk.exe**, an executable file
- **stcewgenericmonk.def**, an executable configuration definition file
- Monk template scripts
- e*Way Monk APIs

stcewgenericmonk.exe

This executable component, **stcewgenericmonk.exe**, is the core of the e*Way that communicates and manipulates Events traveling between an external system and e*Gate, using the Monk external function scripts. It implements the communication between the 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.

stcewgenericmonk.def

The configuration definition file, **stcewgenericmonk.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 are Monk functions that allow the e*Way to communicate with a specific external system. The remaining parameters consist of a set of Monk variables used by the Monk environment. These configuration parameters are set using the e*Way Editor.

Monk Template Scripts

e*Ways use Monk functions to perform such basic operations as startup, data exchange, positive and negative acknowledgement, and establish and shut down the connection to the external system. The Generic e*Way kit includes templates that illustrate the required input and return values for each basic function. These functions must be customized to meet the requirements of the e*Way you wish to design. For example, the **exchange data with external** function that reads data from a file will be different from a function written to obtain that data from a database.

e*Way APIs

The e*Way Monk APIs are described in the section **"Monk Configuration" on page 62**. The Generic e*Way can also be configured to use application-specific API libraries.

1.3 Supported Operating Systems

The Generic e*Way Extension Kit is available on the following operating systems:

- Windows 2000, Windows XP, and Windows Server 2003
- HP-UX 11.0 and 11i (PA-RISC)
- IBM AIX 5.1L and 5.2
- IBM z/OS V1.3 and V1.4
- Red Hat Enterprise Linux AS 2.1 (Intel x86)
- Sun Solaris 8 and 9
- Japanese Windows 2000, Windows XP, and Windows Server 2003
- Japanese HP-UX 11.0 and HP-UX 11i (PA-RISC)
- Japanese Sun Solaris 8 and 9
- Korean HP-UX 11.0 and 11i (PA-RISC)

1.4 System Requirements

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

- An e*Gate Participating Host, version 4.5.1 or later, except for the following operating system:
  - The z/OS 1.3 and 1.4 operating systems are supported by e*Gate versions 4.5.2 and 4.5.3.
- A TCP/IP network connection.
- Java JDK version 1.3.1._02 or later.

- 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.

- Open and review the Readme.txt for the Generic e*Way regarding any additional requirements prior to installation. The Readme.txt is located on the Installation CD_ROM at **setup\addons\ewmux**.

1.5 e*Way Extensions and External Applications

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

Figure 1 Extending the Generic e*Way

Extending the Generic e*Way to an External Application
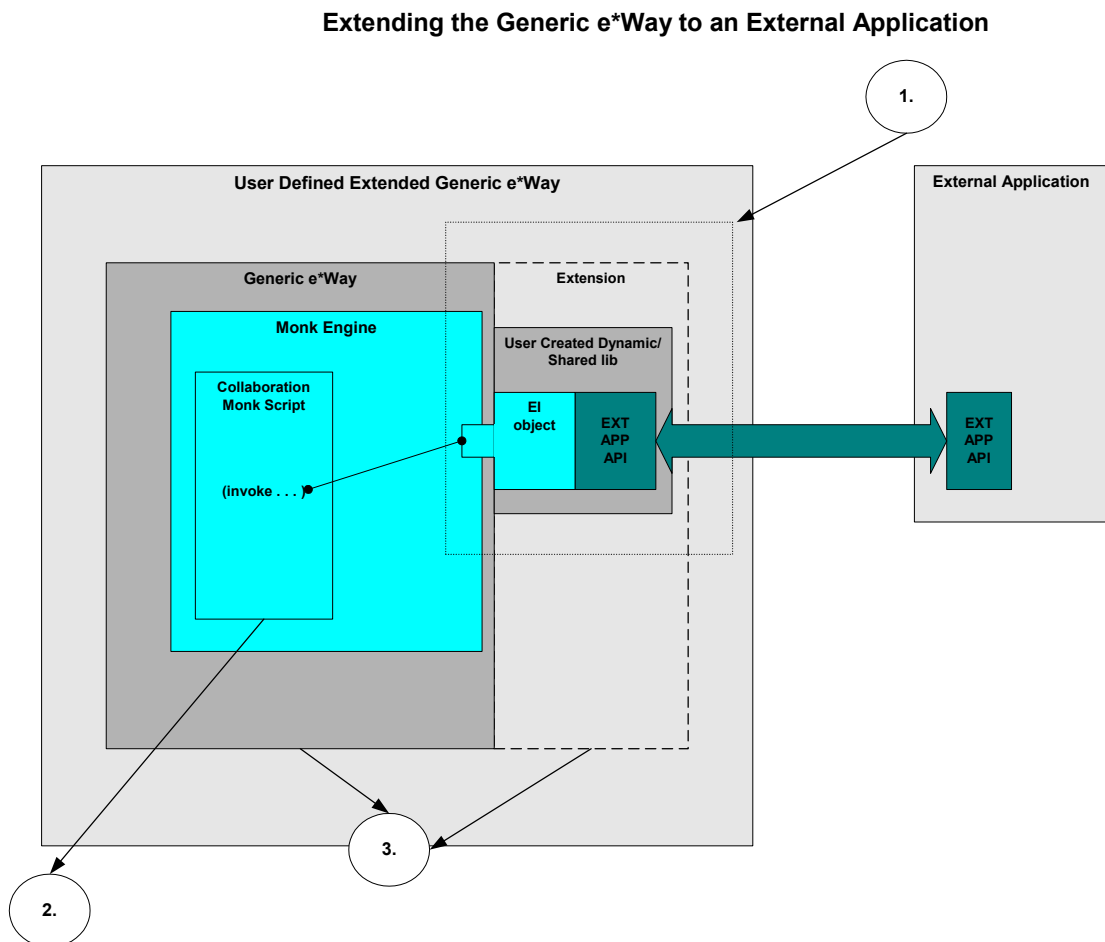
1 A *dynamic link library* or *dll* (on NT) or *shared library* (on UNIX) is created from user-created source code to extend the Generic e*Way.

2 The Generic e*Way is configured to use the user-created dll or shared library.

3   A user-written Monk script uses the EI protocol and the user-created library to access the external application.

1.5.1 Basics Steps to Extend a Generic e*Way

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

1   Create a dynamic link library or shared library for the Generic e*Way to use at run-time to access the external application. To do this, create source code in C or C++ using the EI protocol 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 **stcewgenericmonk.def** file template as needed to allow proper configuring of the Generic e*Way with the Configuration GUI. If you do modify the file template, you must import the changed template to the appropriate schema.

3   Write Monk code that invokes the EI object to enable use of the "wrapped" external application API calls.

4   Configure the Generic e*Way to use the dynamic link library or shared library you created to extend the Generic e*Way.

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

External Interface

This chapter describes the components of the external interface and how to implement them.

2.1 Header File - stcextif.h

The External Interface (EI) allows you to create a dll library (for NT) or shared library (for UNIX) that contains an External Interface object. The EI object holds the types and methods that can be passed between an external application and the Generic e*Way. This chapter describes how the EI header file is used with the EI object. The overview lists the entire header file; subsequent sections discuss each type definition in detail.

2.1.1 Overview

The external interface object defines a structure that contains a location where the user can place data for the object as well as four functions that implement the interface. The four functions are defined in the section **"Methods" on page 18**.

```
#define EXTIF_IN
#define EXTIF_OUT
#define EXTIF_OPTIONAL

#ifdef __cplusplus
extern "C"
{
#endif

typedef void              *HEXTIFUSEROBJECT;

typedef enum
{
    EXTIF_NOOP_PARAM          = 0,
    EXTIF_CHARBLOB_PARAM      = 1,
    EXTIF_WCHARBLOB_PARAM     = 2,
    EXTIF_BOOLEAN_PARAM       = 3,
    EXTIF_CHAR_PARAM          = 4,
    EXTIF_WCHAR_PARAM         = 5,
    EXTIF_DOUBLE_PARAM        = 6,
    EXTIF_LONG_PARAM          = 7,
    EXTIF_OBJECT_PARAM        = 8,
    EXTIF_VECTOR_PARAM        = 9

} ExtIFParamTypes_;
```

```
typedef enum
{
    EXTIF_OK_STATUS            = 0,
    EXTIF_INVOKE_CALL_FAILED   = 1,
    EXTIF_MEM_ALLOC_FAILED     = 2,
    EXTIF_NO_FUNCTION          = 3,
    EXTIF_BAD_FUNC_PARAM       = 4,
    EXTIF_BAD_ARG_PARAM        = 5

} ExtIFResult_;

typedef struct EXTIF_CHARBLOB_
{
    unsigned int          cbBlob;
    unsigned char         *pbBlob;

} EXTIF_CHARBLOB, *PEXTIF_CHARBLOB;

typedef struct EXTIF_WCHARBLOB_
{
    unsigned int          cwcBlob;
    long                  *pwcBlob;

} EXTIF_WCHARBLOB, *PEXTIF_WCHARBLOB;

typedef struct EXTIF_VECTOR_
{
    unsigned int          cElements;
    struct EXTIF_PARAM_   *rgElements;

} EXTIF_VECTOR, *PEXTIF_VECTOR;

typedef struct EXTIF_PARAM_
{
    ExtIFParamTypes_              eType;

    union
    {
        EXTIF_CHARBLOB          CharBlobParam;
        EXTIF_WCHARBLOB         WCharBlobParam;
        unsigned char           fBooleanParam;
        unsigned char           bCharParam;
        unsigned long           lWCharParam;
        double                  dblDoubleParam;
        long int                liLongParam;
        EXTIF_VECTOR            VectorParam;
        struct EXTIF_OBJECT_    *pObjectParam;

    } u;

} EXTIF_PARAM, *PEXTIF_PARAM;
```

2.1.2 Type Definitions List

The following structures define the External interface:

- ExtIFParamTypes_

- ExtIFResult_

- EXTIF_PARAM_

- EXTIF_CHARBLOB

- EXTIF_WCHARBLOB
- EXTIF_VECTOR
- EXTIF_OBJECT

ExtIFParamTypes_

Structure

```
typedef enum
{
    EXTIF_NOOP_PARAM            = 0,
    EXTIF_CHARBLOB_PARAM        = 1,
    EXTIF_WCHARBLOB_PARAM       = 2,
    EXTIF_BOOLEAN_PARAM         = 3,
    EXTIF_CHAR_PARAM            = 4,
    EXTIF_WCHAR_PARAM           = 5,
    EXTIF_DOUBLE_PARAM          = 6,
    EXTIF_LONG_PARAM            = 7,
    EXTIF_OBJECT_PARAM          = 8,
    EXTIF_VECTOR_PARAM          = 9

} ExtIFParamTypes_;
```

Description

These are the types of parameters that can be passed to and from an External Interface.

EXTIF_NOOP_PARAM	A no-op parameter
EXTIF_CHARBLOB_PARAM	A non-specific char data
EXTIF_WCHARBLOB_PARAM	A non-specific wide char data
EXTIF_BOOLEAN_PARAM	A boolean
EXTIF_CHAR_PARAM	A character
EXTIF_WCHAR_PARAM	A wide character
EXTIF_DOUBLE_PARAM	A double floating point number
EXTIF_LONG_PARAM	A long integer value
EXTIF_OBJECT_PARAM	An object interface object that can be passed in or out of the invoke call.
EXTIF_VECTOR_PARAM	A vector (array) containing any type of parameter

ExtIFResult_

Structure

```
typedef enum
{
    EXTIF_OK_STATUS             = 0,
    EXTIF_INVOKE_CALL_FAILED    = 1,
    EXTIF_MEM_ALLOC_FAILED      = 2,
    EXTIF_NO_FUNCTION           = 3,
    EXTIF_BAD_FUNC_PARAM        = 4,
    EXTIF_BAD_ARG_PARAM         = 5

} ExtIFResult_;
```

Description

These are the available return values from the **invoke** call.

EXTIF_OK_STATUS	Status ok.
EXTIF_INVOKE_CALL_FAILED	Generic failure of the invoke call. Most interfaces should perform a get last error method.
EXTIF_MEM_ALLOC_FAILED	Memory allocation error. Somewhere during the call a malloc function call failed to allocate memory.
EXTIF_NO_FUNCTION	The function name that was passed to the invoke call is not supported in this interface.
EXTIF_BAD_FUNC_PARAM	The number of arguments passed to the function was invalid.
EXTIF_BAD_ARG_PARAM	One of the arguments to the function of the invoke call was invalid.

EXTIF_PARAM_

Structure

```
typedef struct EXTIF_PARAM_
{
    ExtIFParamTypes_            eType;

    union
    {
        EXTIF_CHARBLOB             CharBlobParam;
        EXTIF_WCHARBLOB            WCharBlobParam;
        unsigned char              fBooleanParam;
        unsigned char              bCharParam;
        unsigned long              lWCharParam;
        double                     dblDoubleParam;
        long int                   liLongParam;
        EXTIF_VECTOR               VectorParam;
      struct EXTIF_OBJECT_   *pObjectParam;

    } u;
} EXTIF_PARAM, *PEXTIF_PARAM;
```

Description

etype specifies the type of the parameter. The **etype** will indicate which of the union members contains the appropriate data.

EXTIF_CHARBLOB

Structure

```
typedef struct EXTIF_CHARBLOB_
{
    unsigned int            cbBlob;
    unsigned char           *pbBlob;

} EXTIF_CHARBLOB, *PEXTIF_CHARBLOB;
```

Description

This structure is used to handle blob data. CharBlobParam has two parameters.

Parameters

Parameter Name	Description
cbBlob	Count of char data passed
pbBlob	An array of char data.

EXTIF_WCHARBLOB

Structure

```
typedef struct EXTIF_WCHARBLOB_
{
    unsigned int            cwcBlob;
    long                    *pwcBlob;

} EXTIF_WCHARBLOB, *PEXTIF_WCHARBLOB;
```

Description

This structure is used to handle wide character blob data. WCharBlobParam has two parameters:

Parameter Name	Description
cwcBlob	The length of the data
pwcBlob	A long character array

EXTIF_VECTOR

Structure

```
typedef struct EXTIF_VECTOR_
{
    unsigned int            cElements;
    struct EXTIF_PARAM_     *rgElements;

} EXTIF_VECT
```

Description

This structure is used to handle a vector of parameter elements. VectorParam contains two parameters:

Parameter Name	Description
cElements	The count of vector elements
rgElements	An array of parameter elements (EXTIF_PARAM_)

EXTIF_OBJECT

Structure

```
typedef struct EXTIF_OBJECT_
{
    unsigned int        cbStruct; /* set to sizeof(EXTIF_OBJECT) */

    HEXTIFUSEROBJECT    hUserObject;

    PFNEXTIFINVOKE      pfnInvoke;
    PFNEXTIFFREEARGS    pfnFreeArgs;
    PFNEXTIFADDREF      pfnAddRef;
    PFNEXTIFREMOVEREF   pfnRemoveRef;

    unsigned char       bLast4v1;

} EXTIF_OBJECT, *PEXTIF_OBJECT;
```

Description

This is the External Interface object that the **init** function (see below) returns.

Parameter Name	Description
cbStruct	Size of this struct; must be set to **size of (EXTIF_OBJECT)**
hUserObject	A pointer to user-defined object data for the specified interface

2.2 Init Function

The **init** function is defined in the loadable extension library. It is called directly after loading the library. It initializes an External Interface object and returns it to the calling function.

Syntax

```
typedef ExtIFResult_ (*PFNEXTIFINIT)
        (IN struct EXTIF_OBJECT_ **pExtIf,
         IN unsigned long ulFlags,
         IN OUT OPTIONAL void *pvReserved);
```

Description

func ptr: PFNEXTIFINIT (perform next interface object initialization)

Parameters

Parameter Name	Description
pExtIf	(IN) A pointer to the External Interface Object
ulFlags	(IN) Bit flags reserved for future use
pvReserved	This parameter is reserved for future use and must be set to NULL.

Parameter Name	Description
return	EXTIF_OK_STATUS, others if not

2.3 Methods

The External Interface object consists of these interface methods:

- invoke
- freeargs
- addref
- removeref

2.3.1 List of Methods

invoke

Syntax

```
typedef ExtIFResult_ (*PFNEXTIFINVOKE)
            (EXTIF_IN struct EXTIF_OBJECT_ *pExtIf,
                EXTIF_IN const char *pcszFunctionName,
                EXTIF_IN const PEXTIF_VECTOR ppivInArgs,
                EXTIF_OUT PEXTIF_VECTOR *ppivOutArgs,
                EXTIF_IN unsigned long ulFlags,
                EXTIF_IN EXTIF_OUT EXTIF_OPTIONAL void *pvReserved);
```

Description

func ptr: PFNEXTIFINVOKE

Purpose

The **invoke** function is attached to the External Interface object that will invoke methods defined in that interface object.

Parameters

Parameter	Description
pExtIf	(IN) A pointer to the External Interface Object
pcszFunctionName	(IN) The name of the method that will be invoked by the interface object. This is a null terminated char string.
pcmivInArgs	(IN) A vector of input arguments
ppivOutArgs	(OUT) A vector of output arguments
ulFlags	(IN) This parameter is reserved for future use and **must** be set to NULL.

Parameter	Description
pvReserved	(IN) (OUT) This parameter is reserved for future use and **must** be set to NULL.
return	EXTIF_OK_STATUS, others if not

freeargs

Syntax

```
typedef ExtIFResult_ (*PFNEXTIFFREEARGS)
            (EXTIF_IN struct EXTIF_OBJECT_ *pExtIf,
                EXTIF_IN PEXTIF_VECTOR *ppivArgs,
                EXTIF_IN unsigned long ulFlags,
                EXTIF_IN EXTIF_OUT EXTIF_OPTIONAL void *pvReserved);
```

Description

func ptr: PFNEXTIFFREEARGS

Purpose

The **freeargs** function is attached to the External Interface object that will be called after an **invoke** function is called. It will free the return arguments that the **invoke** function returned.

Parameters

Parameter	Description
pExtIf	(IN) A pointer to the External Interface Object
pcivInArgs	(IN) A vector of input arguments
ulFlags	(IN) Bit flags reserved for future use. **Must** be set to NULL.
pvReserved	(IN OUT) This parameter is reserved for future use and **must** be set to NULL.
return	EXTIF_OK_STATUS, others if not

addref

Syntax

```
typedef ExtIFResult_ (*PFNEXTIFADDREF)
            (EXTIF_IN struct EXTIF_OBJECT_ *pExtIf,
                EXTIF_IN unsigned long ulFlags,
                EXTIF_IN EXTIF_OUT EXTIF_OPTIONAL void *pvReserved);
```

Description

func ptr: PFNEXTIFADDREF

Purpose

The **addref** function should be called whenever a new reference is to be kept for this object.

Parameters

Parameter	Description
pExtIf	(IN) A pointer to the External Interface Object
ulFlags	(IN) Bit flags reserved for future use. **Must** be set to NULL.
pvReserved	(IN OUT) This parameter is reserved for future use and **must** be set to NULL.
return	EXTIF_OK_STATUS, others if not.

removeref

Syntax

```
typedef ExtIFResult_ (*PFNEXTIFREMOVEREF)
            (EXTIF_IN struct EXTIF_OBJECT_ **pExtIf,
             EXTIF_IN unsigned long ulFlags,
             EXTIF_IN EXTIF_OUT EXTIF_OPTIONAL void *pvReserved);
```

Description

func ptr: PFNEXTIFREMOVEREF

Purpose

The **removeref** function should be called whenever a reference is to be removed for this object.

Parameters

Parameter	Description
pExtIf	(IN) A pointer to the External Interface Object
ulFlags	(IN) Bit flags reserved for future use. **Must** be set to NULL.
pvReserved	(IN OUT) This parameter is reserved for future use and **must** be set to NULL.
return	EXTIF_OK_STATUS, others if not.

2.4 Template Source Code

The following sample source code is a template that provides an example of how to use the External Interface's headers, types, methods, and functions to provide access to an external application. This external application represents a database system.

A detailed description of the source code (sampleext.c) follows the sample which can be located on the Installation CD in the root directory under samples\sdk\ewayext.

```
#include <malloc.h>
#include <stdlib.h>
```

```c
#include <stdio.h>
#include <string.h>

#include "stcextif.h"

struct object {
    int refcount;
};

/* Example functions */
static void open_db(char *username, char *password)
{
    /* Code to implement database open here */
}

static void close_db(void)
{
    /* Code to implement database close here */
}

static char *get_db(char *key)
{
    /* Code to implement database get here */
    return "look up value";
}

static void put_db(char *key, char *value)
{
    /* Code to implement database put here */
}

static ExtIFResult_ free_args_helper(PEXTIF_VECTOR pivArgs)
{
    unsigned int i;

    for (i = 0; i < pivArgs->cElements;i++)
    {
        switch(pivArgs->rgElements[i].eType)
        {
        case EXTIF_NOOP_PARAM:
            break;
        case EXTIF_CHARBLOB_PARAM:
            if (pivArgs->rgElements[i].u.CharBlobParam.pbBlob)
                free(pivArgs->rgElements[i].u.CharBlobParam.pbBlob);
            break;
        case EXTIF_WCHARBLOB_PARAM:
            if (pivArgs->rgElements[i].u.WCharBlobParam.pwcBlob)
                free(pivArgs-
>rgElements[i].u.WCharBlobParam.pwcBlob);
            break;
        case EXTIF_BOOLEAN_PARAM:
            break;
        case EXTIF_CHAR_PARAM:
            break;
        case EXTIF_WCHAR_PARAM:
            break;
        case EXTIF_DOUBLE_PARAM:
            break;
        case EXTIF_LONG_PARAM:
            break;
        case EXTIF_OBJECT_PARAM:
            if (pivArgs->rgElements[i].u.pObjectParam)
            {
                pivArgs->rgElements[i].u.pObjectParam->pfnRemoveRef
```

```
          (pivArgs->rgElements[i].u.pObjectParam, 0, 0);
                }
                break;
           case EXTIF_VECTOR_PARAM:
                free_args_helper(&(pivArgs-
>rgElements[i].u.VectorParam));
                break;
           }
      }
      if (pivArgs->rgElements)
           free(pivArgs->rgElements);
      return EXTIF_OK_STATUS;
}

static ExtIFResult_ free_args(struct EXTIF_OBJECT_ *pExtIf,
                                PEXTIF_VECTOR *ppivArgs,
                                unsigned long ulFlags,
                                void *pvReserved)
{
      if (ppivArgs && *ppivArgs)
      {
           free_args_helper(*ppivArgs);
           free(*ppivArgs);
           *ppivArgs = 0;
      }
      return EXTIF_OK_STATUS;
}

static ExtIFResult_ remove_ref(struct EXTIF_OBJECT_ **pExtIf,
                                unsigned long ulFlags,
                                void *pvReserved)
{
      if (--(((struct object *) ((*pExtIf)->hUserObject))->refcount) <=
0)
      {
           free((*pExitIf)->hUserObject);
           free((*pExitIf));
           *pExitIf = 0;
      }
      return EXTIF_OK_STATUS;
}

static ExtIFResult_ addref(struct EXTIF_OBJECT_ *pExtIf,
                                unsigned long ulFlags,
                                void *pvReserved)
{
      ((struct object *) (pExtIf->hUserObject))->refcount++;
      return EXTIF_OK_STATUS;
}

static ExtIFResult_ invoke(struct EXTIF_OBJECT_ *pExtIf,
                                const char *pcszFunctionName,
                                const PEXTIF_VECTOR pcmivInArgs,
                                PEXTIF_VECTOR *ppivOutArgs,
                                unsigned long ulFlags,
                                void *pvReserved)
{
      if (pExtIf)
      {
           if (strcmp("open", pcszFunctionName) == 0)
           {
                if (pcmivInArgs->cElements == 2 && pcmivInArgs-
>rgElements[0].eType == EXTIF_CHARBLOB_PARAM
```

```
                       && pcmivInArgs->rgElements[1].eType ==
EXTIF_CHARBLOB_PARAM)
            {
                    open_db(pcmivInArgs-
>rgElements[0].u.CharBlobParam.pbBlob,
                          pcmivInArgs-
>rgElements[1].u.CharBlobParam.pbBlob);
                    return EXTIF_OK_STATUS;
            }
            else
                    return EXTIF_BAD_ARG_PARAM;
        }
        else if (strcmp("close", pcszFunctionName) == 0)
        {
            if (pcmivInArgs->cElements == 0)
            {
                    close_db();
                    return EXTIF_OK_STATUS;
            }
            else
                    return EXTIF_BAD_ARG_PARAM;
        }
        else if (strcmp("get", pcszFunctionName) == 0)
        {
            if (pcmivInArgs->cElements == 1 && pcmivInArgs-
>rgElements[0].eType == EXTIF_CHARBLOB_PARAM)
            {
                    char *var;

                    var = get_db(pcmivInArgs-
>rgElements[0].u.CharBlobParam.pbBlob);

                    (*ppivOutArgs) = (PEXTIF_VECTOR)
malloc(sizeof(EXTIF_VECTOR));
                    (*ppivOutArgs)->cElements = 1;
                    (*ppivOutArgs)->rgElements = (PEXTIF_PARAM)
malloc(sizeof(EXTIF_PARAM));

                    (*ppivOutArgs)->rgElements[0].eType =
EXTIF_CHARBLOB_PARAM;
                    (*ppivOutArgs)->rgElements[0].u.CharBlobParam.cbBlob
= strlen(var);
                    (*ppivOutArgs)->rgElements[0].u.CharBlobParam.pbBlob
=
                           (unsigned char *) malloc((*ppivOutArgs)-
>rgElements[0].u.CharBlobParam.cbBlob);
                    strncpy((*ppivOutArgs)-
>rgElements[0].u.CharBlobParam.pbBlob, var,
                           (*ppivOutArgs)-
>rgElements[0].u.CharBlobParam.cbBlob);
                    return EXTIF_OK_STATUS;
            }
            else
                    return EXTIF_BAD_ARG_PARAM;
        }
        else if (strcmp("put", pcszFunctionName) == 0)
        {
            if (pcmivInArgs->cElements == 2 && pcmivInArgs-
>rgElements[0].eType == EXTIF_CHARBLOB_PARAM
                    && pcmivInArgs->rgElements[1].eType ==
EXTIF_CHARBLOB_PARAM)
            {
                    put_db(pcmivInArgs-
>rgElements[0].u.CharBlobParam.pbBlob,
```

```
                            pcmivInArgs-
>rgElements[1].u.CharBlobParam.pbBlob);
                    return EXTIF_OK_STATUS;
                }
                else
                    return EXTIF_BAD_ARG_PARAM;
            }
            else
                return EXTIF_NO_FUNCTION;
        }
        else
            return EXTIF_BAD_FUNC_PARAM;
    }

    #if defined(WIN32)
    __declspec(dllexport)
    #endif
    ExtIFResult_ init_sampleext(struct EXTIF_OBJECT_ **pExitIf,
                                unsigned long ulFlags,
                                void *pvReserved)
    {
        (*pExitIf) = (struct EXTIF_OBJECT_ *) malloc(sizeof(struct
    EXTIF_OBJECT_));
        if (*pExitIf)
        {
            struct object *obj = (struct object *) malloc(sizeof(struct
    object));
            if (obj)
            {
                obj->refcount = 1;
                (*pExitIf)->cbStruct = sizeof(struct EXTIF_OBJECT_);
                (*pExitIf)->hUserObject = (void *) obj;
                (*pExitIf)->pfnAddRef = addref;
                (*pExitIf)->pfnFreeArgs = free_args;
                (*pExitIf)->pfnInvoke = invoke;
                (*pExitIf)->pfnRemoveRef = remove_ref;
                return EXTIF_OK_STATUS;
            }
            else
            {
                free(*pExitIf);
                *pExitIf = 0;
                return EXTIF_MEM_ALLOC_FAILED;
            }
        }
        else
            return EXTIF_MEM_ALLOC_FAILED;
    }
```

2.5 Sample Source Code Description

This section is a description of the Simple DB Source Code Sample that uses the
External Interface (EI) to create an application-specific dynamic link library (.dll) or
shared library (.so or .sl) for extending the Generic e*Way functionality.

These are the standard header files needed by this EI dynamic or shared library:

```
#include <malloc.h>
#include <stdlib.h>
#include <stdio.h>
```

```
#include <string.h>
```

This is the header file that defines the structures, parameters, methods and functions used by the EI:

```
#include "stcextif.h"
```

This is used to store the data members:

```
struct object {
    int refcount;
};
```

Here are examples of function stubs that could be written to "wrap" an external application's API functions.

```
/* Example functions */
static void open_db(char *username, char *password)
{
    /* Code to implement database open here */
}

static void close_db(void)
{
    /* Code to implement database close here */
}

static char *get_db(char *key)
{
    /* Code to implement database get here */
    return "look up value";
}

static void put_db(char *key, char *value)
{
    /* Code to implement database put here */
}
```

This function "free_args_helper" will free up an array of characters.

```
static ExtIFResult_ free_args_helper(PEXTIF_VECTOR pivArgs)
{
    unsigned int i;

    for (i = 0; i < pivArgs->cElements;i++)
    {
        switch(pivArgs->rgElements[i].eType)
        {
        case EXTIF_NOOP_PARAM:
            break;
        case EXTIF_CHARBLOB_PARAM:
            if (pivArgs->rgElements[i].u.CharBlobParam.pbBlob)
                free(pivArgs->rgElements[i].u.CharBlobParam.pbBlob);
            break;
        case EXTIF_WCHARBLOB_PARAM:
            if (pivArgs->rgElements[i].u.WCharBlobParam.pwcBlob)
                free(pivArgs-
>rgElements[i].u.WCharBlobParam.pwcBlob);
            break;
        case EXTIF_BOOLEAN_PARAM:
            break;
        case EXTIF_CHAR_PARAM:
            break;
        case EXTIF_WCHAR_PARAM:
            break;
        case EXTIF_DOUBLE_PARAM:
            break;
        case EXTIF_LONG_PARAM:
```

```
                 break;
            case EXTIF_OBJECT_PARAM:
                if (pivArgs->rgElements[i].u.pObjectParam)
                {
 pivArgs->rgElements[i].u.pObjectParam->pfnRemoveRef(pivArgs-
 >rgElements[i].u
 .pObjectParam, 0, 0);
                    free(pivArgs->rgElements[i].u.pObjectParam);
                }
```

The "free_args_helper" function is used here recursively to delete individual parameters

```
            case EXTIF_VECTOR_PARAM:
                free_args_helper(&(pivArgs-
 >rgElements[i].u.VectorParam));
                break;
            }
        }
        if (pivArgs->rgElements)
            free(pivArgs->rgElements);
        return EXTIF_OK_STATUS;
    }
```

The function "free_args" will delete the arguments returned from the previous "invoke" call.

```
    static ExtIFResult_ free_args(struct EXTIF_OBJECT_ *pExtIf,
                        PEXTIF_VECTOR *ppivArgs,
                        unsigned long ulFlags,
                        void *pvReserved)
    {
        if (ppivArgs && *ppivArgs)
        {
            free_args_helper(*ppivArgs);
            free(*ppivArgs);
            *ppivArgs = 0;
        }
        return EXTIF_OK_STATUS;
    }
```

Call the "remove_ref" function when an interface object is no longer needed. If "refcount" goes to zero, this function frees the user object.

```
    static ExtIFResult_ remove_ref(struct EXTIF_OBJECT_ *pExtIf,
                        unsigned long ulFlags,
                        void *pvReserved)
    {
        if (--(((struct object *) (pExtIf->hUserObject))->refcount) <= 0)
            free(pExtIf->hUserObject);
        return EXTIF_OK_STATUS;
    }
```

Call the "addref" function when an interface object is used. Increment the "refcount".

```
    static ExtIFResult_ addref(struct EXTIF_OBJECT_ *pExtIf,
                        unsigned long ulFlags,
                        void *pvReserved)
    {
        ((struct object *) (pExtIf->hUserObject))->refcount++;
        return EXTIF_OK_STATUS;
    }
```

The "invoke" function will call an external function via the "wrapped" function (see above for the "wrapped" function stubs open_db, close_db, get_db and put_db).

```
    static ExtIFResult_ invoke(struct EXTIF_OBJECT_ *pExtIf,
                        const char *pcszFunctionName,
                        const PEXTIF_VECTOR pcmivInArgs,
                        PEXTIF_VECTOR *ppivOutArgs,
```

```
                            unsigned long ulFlags,
                            void *pvReserved)
{
    if (pExtIf)
    {
        if (strcmp("open", pcszFunctionName) == 0)
        {
            if (pcmivInArgs->cElements == 2 && pcmivInArgs-
>rgElements[0].eType ==
EXTIF_CHARBLOB_PARAM
                    && pcmivInArgs->rgElements[1].eType ==
EXTIF_CHARBLOB_PARAM)
            {
                open_db(pcmivInArgs-
>rgElements[0].u.CharBlobParam.pbBlob,
                        pcmivInArgs-
>rgElements[1].u.CharBlobParam.pbBlob);
                return EXTIF_OK_STATUS;
            }
            else
                return EXTIF_BAD_ARG_PARAM;
        }
        else if (strcmp("close", pcszFunctionName) == 0)
        {
            if (pcmivInArgs->cElements == 0)
            {
                close_db();
                return EXTIF_OK_STATUS;
            }
            else
                return EXTIF_BAD_ARG_PARAM;
        }
        else if (strcmp("get", pcszFunctionName) == 0)
        {
            if (pcmivInArgs->cElements == 1 && pcmivInArgs-
>rgElements[0].eType ==
EXTIF_CHARBLOB_PARAM)
            {
                char *var;

                var = get_db(pcmivInArgs-
>rgElements[0].u.CharBlobParam.pbBlob);

                (*ppivOutArgs) = (PEXTIF_VECTOR)
malloc(sizeof(EXTIF_VECTOR));
                (*ppivOutArgs)->cElements = 1;
                (*ppivOutArgs)->rgElements = (PEXTIF_PARAM)
malloc(sizeof(EXTIF_PARAM));

                (*ppivOutArgs)->rgElements[0].eType =
EXTIF_CHARBLOB_PARAM;
                (*ppivOutArgs)->rgElements[0].u.CharBlobParam.cbBlob
= strlen(var);
                (*ppivOutArgs)->rgElements[0].u.CharBlobParam.pbBlob
=
                    (unsigned char *)
malloc((*ppivOutArgs)->rgElements[0].u.CharBlobParam.cbBlob);
                strncpy((*ppivOutArgs)-
>rgElements[0].u.CharBlobParam.pbBlob, var,
                    (*ppivOutArgs)-
>rgElements[0].u.CharBlobParam.cbBlob);
                return EXTIF_OK_STATUS;
            }
            else
```

```
                              return EXTIF_BAD_ARG_PARAM;
          }
          else if (strcmp("put", pcszFunctionName) == 0)
          {
                if (pcmivInArgs->cElements == 2 && pcmivInArgs-
    >rgElements[0].eType ==
    EXTIF_CHARBLOB_PARAM
                        && pcmivInArgs->rgElements[1].eType ==
    EXTIF_CHARBLOB_PARAM)
                {
                        put_db(pcmivInArgs-
    >rgElements[0].u.CharBlobParam.pbBlob,
                              pcmivInArgs-
    >rgElements[1].u.CharBlobParam.pbBlob);
                        return EXTIF_OK_STATUS;
                }
                else
                        return EXTIF_BAD_ARG_PARAM;
          }
          else
                return EXTIF_NO_FUNCTION;
      }
      else
          return EXTIF_BAD_FUNC_PARAM;
}

#if defined(WIN32)
__declspec(dllexport)
#endif
```

The "init_sampleext" function creates the individual interface object and puts the functions above defined in that object and returns it to the caller.

```
ExtIFResult_ init_sampleext(struct EXTIF_OBJECT_ *pExtIf,
                         unsigned long ulFlags,
                         void *pvReserved)
{
    struct object *obj = (struct object *) malloc(sizeof(struct
object));
    if (obj)
    {
        obj->refcount = 1;
        pExtIf->cbStruct = sizeof(struct EXTIF_OBJECT_);
        pExtIf->hUserObject = (void *) obj;
        pExtIf->pfnAddRef = addref;
        pExtIf->pfnFreeArgs = free_args;
        pExtIf->pfnInvoke = invoke;
        pExtIf->pfnRemoveRef = remove_ref;
        return EXTIF_OK_STATUS;
    }
    else
        return EXTIF_MEM_ALLOC_FAILED;
)
```

2.6 Loading the DLL

Use this sample code to load the newly created DLL.

```
(define obj (load-interface "s:/tests/sampleext/debug/
sampleext.dll" "init_sampleext"))
(invoke obj "open_db")
```

```
(define ret-vec (invoke obj "get_db" "key"))

(define ret-string (vector-ref ret-vec 0))
(invoke obj "put_db" "key" ret-string)
(invoke obj "close_db")

; The following is short hand notation (the invoke is optional)
(obj "open_db")
(define ret-vec (obj "get_db" "key"))
(define ret-string (vector-ref ret-vec 0))
(obj "put_db" "key" ret-string)
(obj "close_db")
```

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 also 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.

3.1 Introduction

The Generic e*Way is configured using the e*Way Editor, a graphical user interface (GUI) that 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 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 needed by the GUI. 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 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 Generic e*Way require the definition of user-settable 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 Help system.

Important

We strongly recommend that you make no changes whatsoever to the default **stcewgenericmonk.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.

3.1.1 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 Monk Configuration) must be included in all Generic e*Way **.def** files; additional sections can be added as needed to support user-created functions.

3.2   .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" )

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

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

3.2.1 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.

3.2.2 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."

3.2.3 Floating-point Parameters

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

3.2.4 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:

- \'
- \"
- \\

3.2.5 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**).

3.2.6 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
```

3.2.7 "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 1 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 32** 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 SeeBeyond support personnel.

Table 1 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.

3.3   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 34** for more information on defining parameters.

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

3.3.1 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
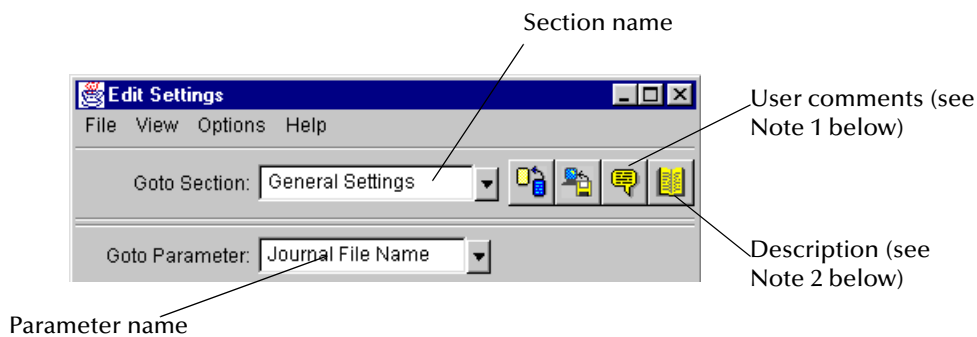
Notes

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-comment)** fields blank, but you can enter information in
    the **(user-comment)** 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.

3.3.2 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 37** and **"Parameters Accepting Multiple Values From a Set" on
  page 38** 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 33** for additional
  information)

- The "user comment" (see the Notes for **"Section Syntax" on page 33** 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. Table 2 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 2   Basic parameter keywords

Type	Parameter keyword	Accepts values	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 48**

Parameters Requiring Single Values

Parameters requiring single values are defined within the basic structure shown in **"Parameter Syntax" on page 34**.

Figure 3   A parameter requiring a single value
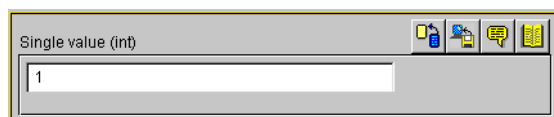
The parameter is defined using a parameter keyword, as listed in **Table 2 on page 36**.

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 39** 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 34**):

- 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 38 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 39** *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
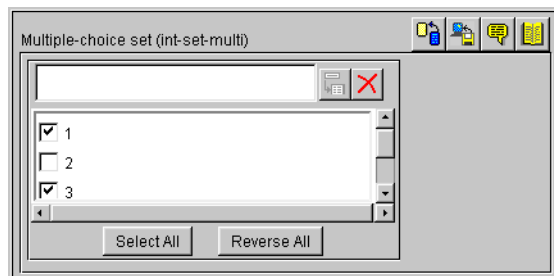
Sets require modifications to the basic parameter syntax (shown in **"Parameter Syntax" on page 34**):

- 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 37** 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 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 35** *for more information.*

3.3.3 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).*

3.3.4 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
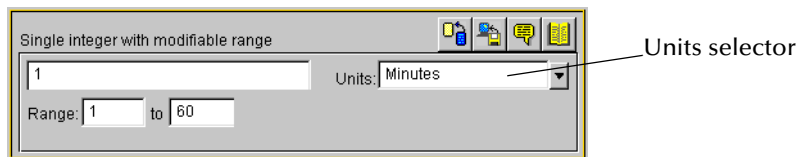
Acceptable groups of units include

- 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"**
for more information.

3.3.5 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 35 *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)**
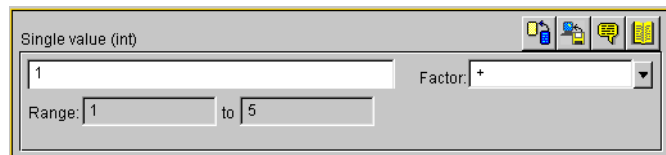
The syntax of the **(factor)** keyword is

```
(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 35** *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.*

3.4 Configuration Keyword Reference

Table 3 lists the keywords that may appear in the **.def** file.

Table 3   .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 36**

Table 3   .def-file keywords

Keyword	Purpose	For more information, see this section
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 37**
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 38**
config-default	Specifies the values that will be restored when the user clicks the e*Way Editor's ⬜ button	**"Parameter Syntax" on page 34**
const	Specifies that a value cannot be changed by the user	**"Specifying Ranges" on page 39**
creation	Records creation date or other information.	**""Header" Information" on page 33**
date	Declares a date parameter	**"Parameter Types" on page 36**
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 37**
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 38**
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 34**
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 44**

Table 3   .def-file keywords

Keyword	Purpose	For more information, see this section
factor	Defines an arithmetic operator to be associated with an integer parameter	**"Factor" on page 43**
general-info	Defines the "general information" division of the **.def** file	**""Header" Information" on page 33**
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 36**
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 37**
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 38**
modified	Records modification date or other information	**""Header" Information" on page 33**
name	Specifies the name of a parameter or a section	**"Parameter Syntax" on page 34**
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 36**
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 37**
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 38**
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 39**
revision	Records revision numbering or other information (entered manually by the developer)	**""Header" Information" on page 33**
schedule	Declares a schedule parameter	**"Parameter Types" on page 36** and **"Schedule Syntax" on page 48**

Table 3   .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 37** and **"Schedule Syntax" on page 48**
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 38** and **"Schedule Syntax" on page 48**
section	Defines a "section" of the **.def** file	See **"Section Syntax" on page 33**
set	Defines the elements in a set	**"Parameters Accepting a Single Value From a Set" on page 37** and **"Parameters Accepting Multiple Values From a Set" on page 38**
show-as	Selects the format in which character or integer parameters will be displayed	**"Displaying Options in ASCII, Octal, Hex, or Decimal" on page 42**
string	Declares a string parameter	**"Parameter Types" on page 36**
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 37**
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 38**
super-client-type	Reserved; do not change from the default "<ANY>".	
time	Declares a time parameter	**"Parameter Types" on page 36**
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 37**
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 38**
units	Determines in which units a parameter will be displayed	**"Specifying Units" on page 40**
user	Records the name of the user who last edited the file (entered manually by the developer)	**""Header" Information" on page 33**

Table 3   .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 34**
value	Defines the initial value for a parameter	**"Parameter Syntax" on page 34**
version	Records the name of the user who last edited the file (entered manually by the developer)	**""Header" Information" on page 33**

3.4.1 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):
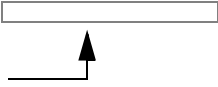
Table 4   Schedule syntax

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/stcewgenericmonk/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/stcewgenericmonk/*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 (see **"Editing a .def File Within a Schema" on page 92** for more information).

3.5 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 3 on page 44**). 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 SeeBeyond 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 **stcewgenericmonk.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 **stcewgenericmonk.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 "")
    )
  )
```

3.6   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.

There are two types of errors that you may encounter:

▪ 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 SeeBeyond 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+5 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 35**.

Note: *Using the templates provided in the sample .def file will help prevent many errors before they occur; see* **"Sample .def File" on page 54** *for more information.*

3.6.1 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 *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-type***TypeSet<*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.

3.7 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/geneway/sample.def

Note: *The **sample.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 **sample.def** file as a template 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 sample.def file in the e*Way Editor:

1 Using the Schema Designer, commit the **sample.def** file to the directory **/configs/stcewgenericmonk/** 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 **sample.def** file with the e*Way Editor.

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

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

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

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

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 Figure 8.

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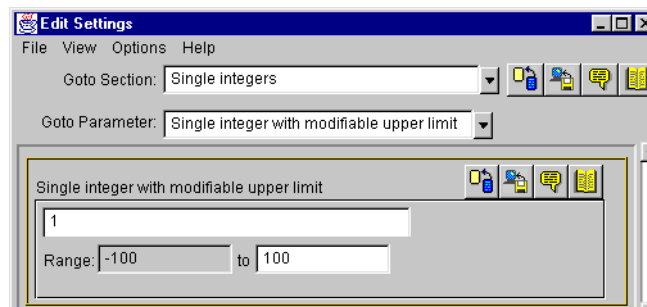
After identifying the parameter you wish to copy, open sample.def in a text editor and search for the parameter name. Then, simply copy the parameter and change the sample values to the values you wish to use (as shown in Figure 9 on the next page).

Figure 9 The **sample.def** file in Wordpad

Copy the code between the comments

Change the values as desired
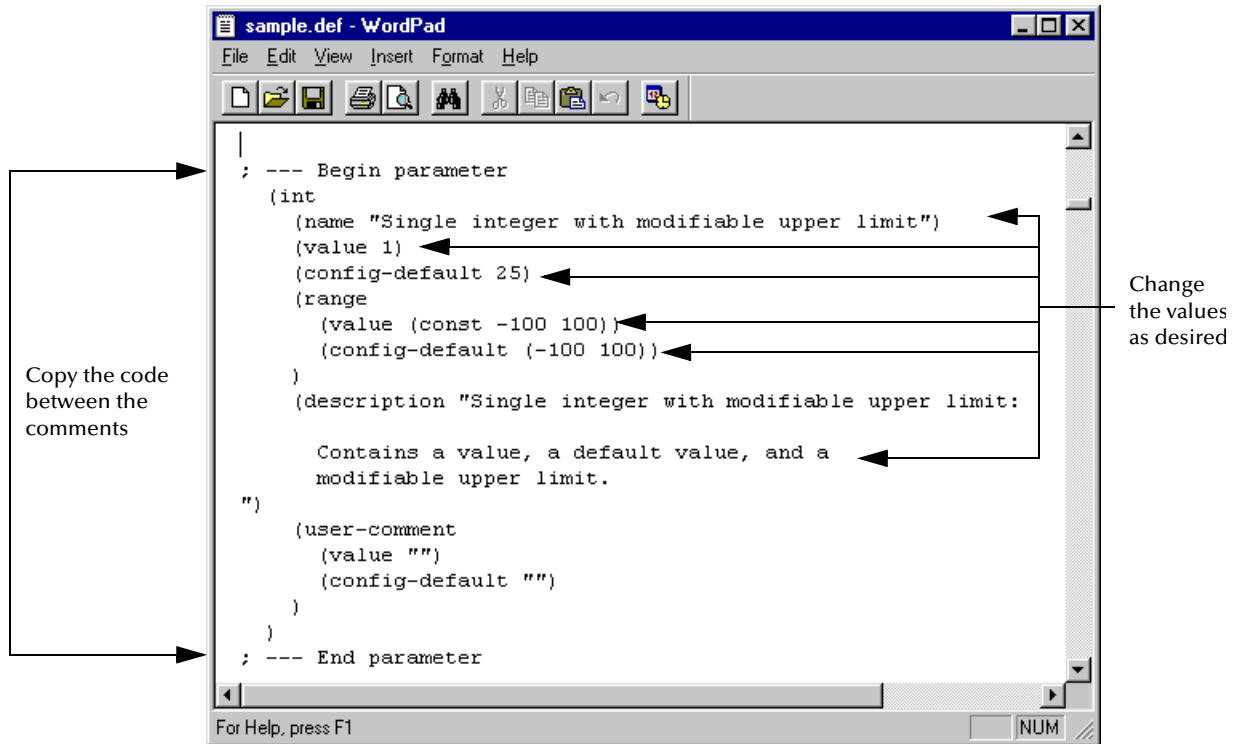
3.8 Accessing Configuration Parameters Within the Monk Environment

The Generic e*Way automatically loads configuration parameters stored in the **.cfg** file into variables within the Monk environment.

3.8.1 Variable-name Format

Variables 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 an underscore, 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 in the variable "AUTHENTICATION_PASSWORD" (all upper case).

The value of the parameter named "Gateway ID" within the section "Connection Parameters" would be stored in the variable "CONNECTION_PARAMETERS_GATEWAY_ID".

3.8.2 Getting Variable Values

To access variable values, use the above conventions, and returns a string containing that value or a Monk exception, "Unresolved Variable" if the specified variable does not exist.

Examples

```
(string=? CONNECTION_PARAMETERS_GATEWAY_ID "ABC")
```

See the *Monk Developer's Reference* for more information.

Configuration

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

4.1 Required e*Way Configuration Parameters

The e*Way configuration parameters discussed in this section are required by the 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
- Monk Configuration

4.1.1 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 59** 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 73** 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 67** for information about how the e*Way uses this function.

4.1.2 # 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 73**, **"Exchange Data Interval" on page 61**, 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).

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 61** and **"Stop Exchange Data Schedule" on page 60** 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 74** 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 74** 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**.
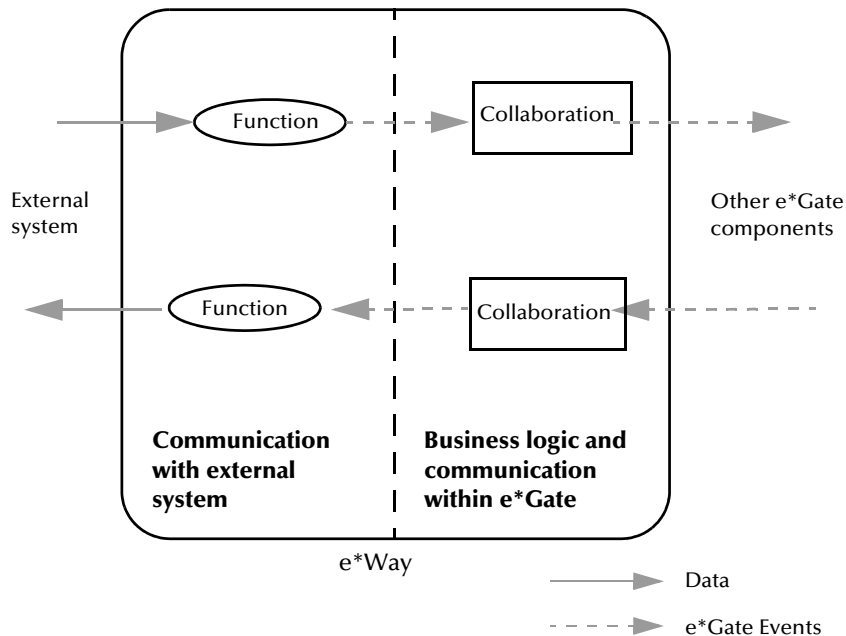
See **"Exchange Data with External Function" on page 73** for more information.

4.1.3 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 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 10   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 SeeBeyond 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:

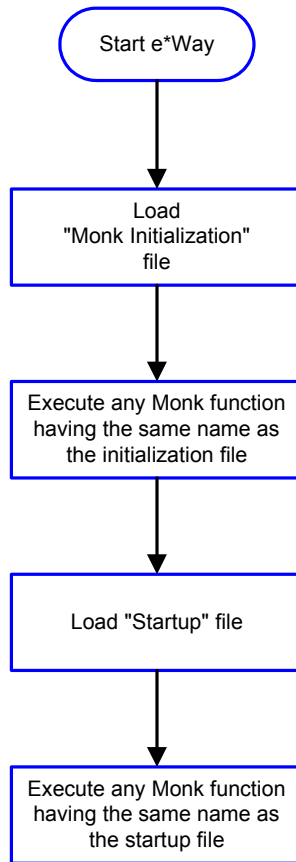| Type of Operation	Name
Initialization	**Startup Function** on page 72 (also see **Monk Environment Initialization File** on page 71)

Type of Operation	Name
Connection	**External Connection Establishment Function** on page 74 **External Connection Verification Function** on page 74 **External Connection Shutdown Function** on page 75
Schedule-driven data exchange	**Exchange Data with External Function** on page 73 **Positive Acknowledgment Function** on page 75 **Negative Acknowledgment Function** on page 76
Shutdown	**Shutdown Command Notification Function** on page 76
Event-driven data exchange	**Process Outgoing Message Function** on page 72

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

Initialization Functions

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

Figure 11 Initialization Functions

```
                    ╭─────────────────╮
                    │   Start e*Way   │
                    ╰─────────────────╯
                             │
                             ▼
              ┌──────────────────────────┐
              │          Load            │
              │   "Monk Initialization"  │
              │          file            │
              └──────────────────────────┘
                             │
                             ▼
              ┌──────────────────────────┐
              │  Execute any Monk function│
              │  having the same name as │
              │   the initialization file│
              └──────────────────────────┘
                             │
                             ▼
              ┌──────────────────────────┐
              │                          │
              │   Load "Startup" file    │
              │                          │
              └──────────────────────────┘
                             │
                             ▼
              ┌──────────────────────────┐
              │  Execute any Monk function│
              │  having the same name as │
              │     the startup file     │
              └──────────────────────────┘
```

Connection Functions

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

**Figure 12** Connection establishment and verification functions
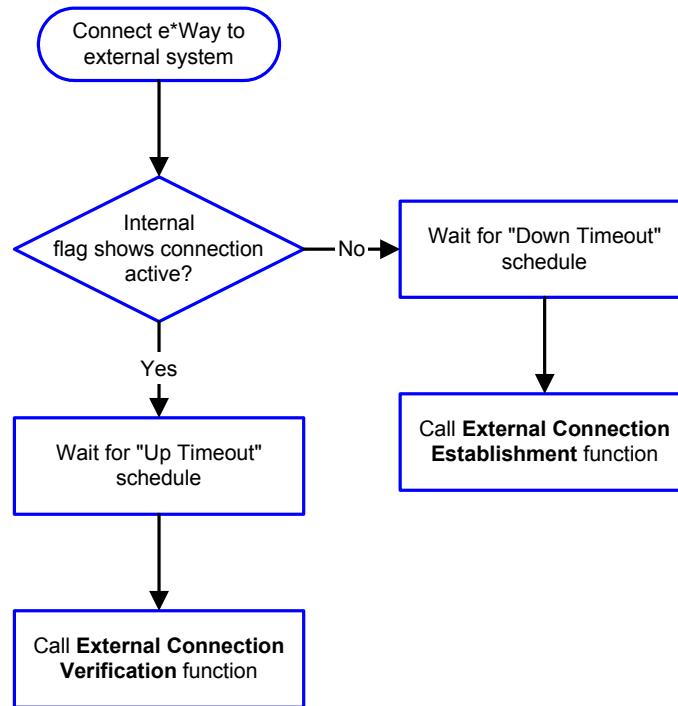
Note: *The e*Way selects the connection function based on an internal "up/down" flag rather than a poll to the external system. See* **Figure 14 on page 68** *and* **Figure 16 on page 70** *for examples of how different functions use this flag.*

User functions can manually set this flag using Monk functions. See **send-external-up** *on page 84 and* **send-external-down** *on page 84 for more information.*

Figure 13 illustrates how the e*Way executes its "connection shutdown" function.

Figure 13 Connection shutdown function
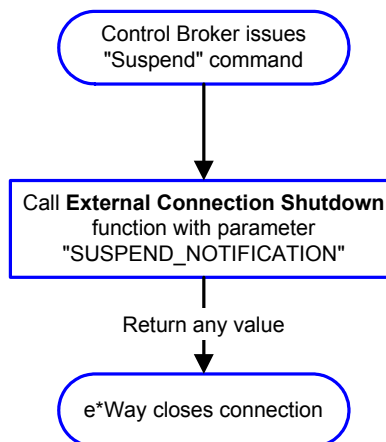
Schedule-driven Data Exchange Functions

Figure 14 (on the next page) illustrates how the e*Way performs schedule-driven data exchange using the **Exchange Data with External Function**. The **Positive Acknowledgement Function** and **Negative Acknowledgement 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 14   Schedule-driven data exchange functions
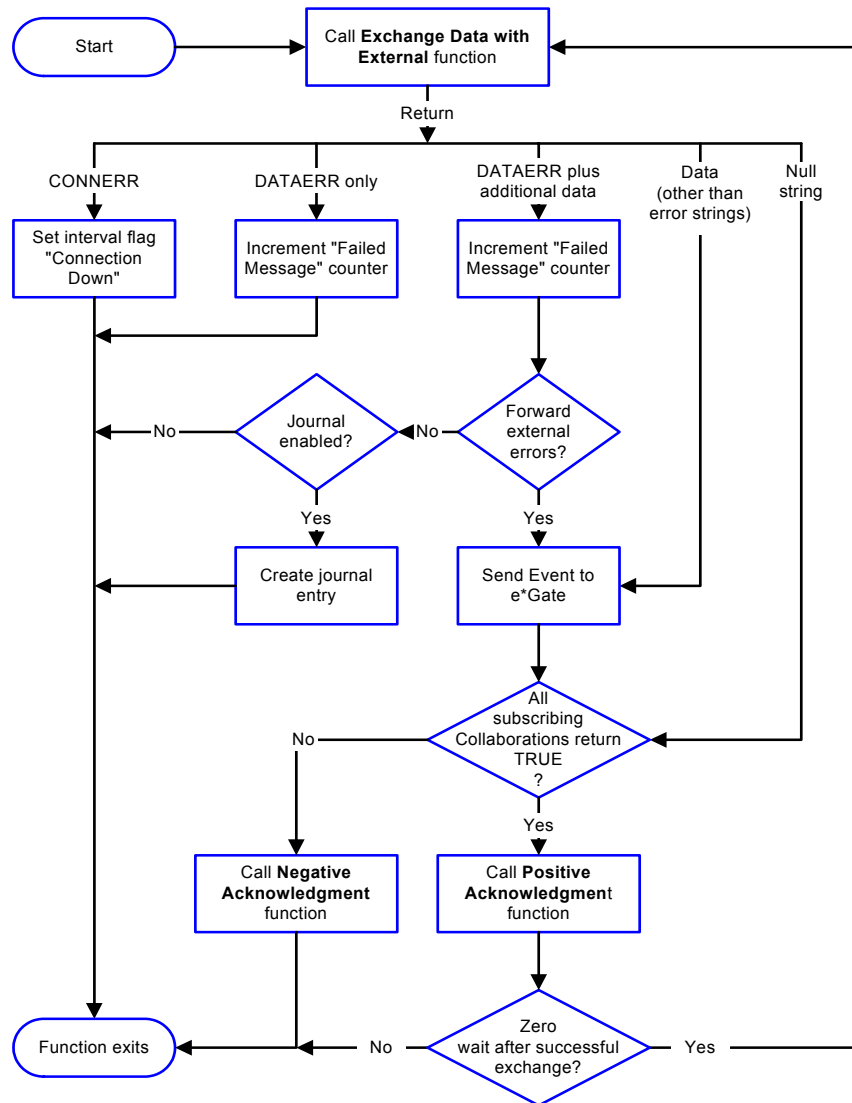
Shutdown Functions

Figure 15 illustrates how the e*Way implements the shutdown request function.

Figure 15   Shutdown functions

```
        ┌─────────────────────────┐
        │   Control Broker issues  │
        │    "Shutdown" command    │
        └─────────────────────────┘
                     │
                     ▼
        ┌─────────────────────────────────┐
        │ Call Shutdown Notification function│
        │          with parameter           │
        │     "SHUTDOWN_NOTIFICATION"       │
        └─────────────────────────────────┘
                     │
                   Return
                     │
          ┌──────────┴──────────┐
          │                     │
   Null string or        any other value
    "SUCCESS"                  │
          │                    ▼
          │          ┌──────────────────┐
          │          │     Wait for     │
          │          │ shutdown-request │
          │          │     function     │
          │          └──────────────────┘
          │                    │
          └─────────►◄─────────┘
                     │
                     ▼
        ┌─────────────────────────┐
        │     e*Way shuts down    │
        └─────────────────────────┘
```
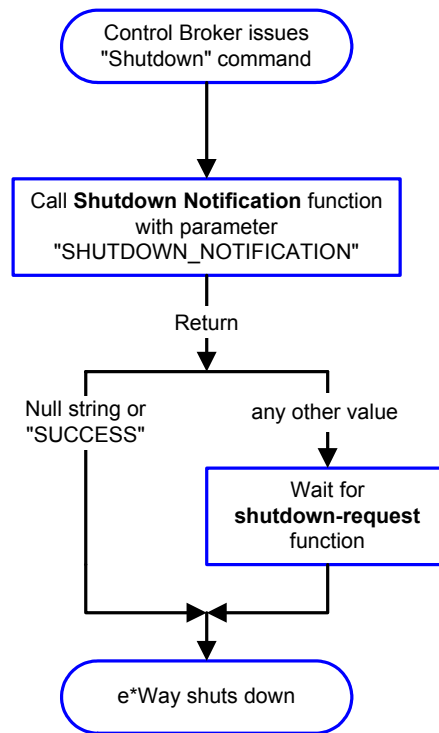
Event-driven Data Exchange Functions

Figure 16 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 16 Event-driven data-exchange functions
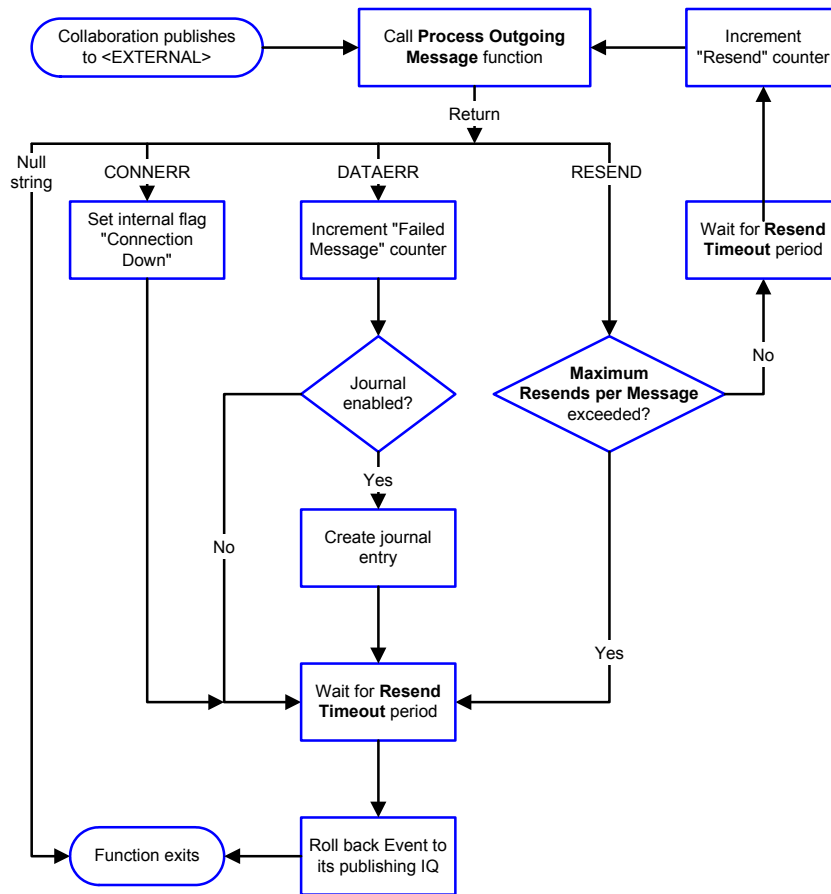
How to Specify Function Names or File Names

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 System Administration and Operations Guide* for more information about this file.

To specify multiple directories, manually enter the directory names rather than selecting them with the "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 "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 70** for more information about the "load path."

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 11 on page 65**).

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.

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 11 on page 65**). 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.*

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 16 on page 70** 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 82 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.

Additional Information

The function accepts no input and must return a string (see **Figure 14 on page 68** 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 14 on page 68** 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 85 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.***

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.

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.

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.

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.

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 "shut down" 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 85).

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.*

4.2 Template Scripts

These template scripts illustrate the following:

1 They demonstrate the required arguments and return values for each "generic" function.

2 They contain "display" messages that you can use to confirm that a given function is launching the appropriate script.

4.2.1 Startup Function

Example:

```
(display "\n")
(display "start up file\n")
; in startup function
(define startup
 (lambda ()
    (display "Executing external startup function.")
    ; Initializing and load monk scripts if necessary
    "SUCCESS"
 )
)
```

4.2.2 Process Outgoing Event Function

Example:

```
(define ProcessOutEventFunc
 (lambda (message-string)
   (display "INFO: Inside Processing Outgoing Message Function.\n")
   (display message-string)
   (newline)
   (display "INFO: Inside Processing Outoing Message Function.\n")
   ; to return data error
   ;   "DATAERR-should not send this data"

   ; to forward external event
   ; "data to be forward in Egate queue"

   ; no data to return
    ""
 )
)
```

4.2.3 Exchange Data with External Function

Example:

```
(define exchangeDataFunc
 (lambda ()
  (newline)
  (display "INFO: Inside Data Exchange Function.\n")
  (newline)

  ; to forward external event
  ; "data to be forward in Egate queue"
```

```
    ; no data to return
    ""
  )
)
```

4.2.4 External Connection Establishment Function

Example:

```
(define establishConnectionFunc
 (lambda ()
  (display "INFO: Inside connection establishment Function.\n")
  ; if established connection
  (send-external-up)
  "UP"
 )
)
```

4.2.5 External Connection Verification Function

Example:

```
(define connectionVerificationFunc
 (lambda ()
  (display "INFO: Inside connection verification Function.\n")
  ; verify connection, if it's still connected
  "UP"
  ; else
  "DOWN"
 )
)
```

4.2.6 External Connection Shutdown Function

Example:

```
(define shutdownFunc
 (lambda (message-string)
  (display "INFO: Inside connection shutdown Function.\n")
  (display message-string)
  (newline)
  "SUCCESS"
 )
)
```

4.2.7 Positive Acknowledgment Function

Example:

```
; positive ack function
(define ackFunc
 (lambda (message-string)
  (display "INFO: Inside Positive Ack Function.\n")
  (display message-string)
  (newline)
  (display "INFO: Inside Positive Ack Function.\n")
  ""
 )
```

```
)
```

4.2.8 Negative Acknowledgment Function

Example:

```
(define nakFunc
 (lambda (message-string)
  (display "INFO: Inside Negative Ack Function.\n")
  (display message-string)
  (newline)
  (display "INFO: Inside Negative Ack Function.\n")
  ""
 )
)
```

4.2.9 Shutdown Command Notification Function

Example:

```
(define shutdownNotificationFunc
 (lambda (message-string)
 (display "INFO: Inside Shutdown Command Notification Function.\n")
 "SUCCESS"
 )
)
```

Interface API Functionality

This chapter describes the core and extension functions.

5.1 Core Functions

The following functions are available to all e*Ways based on the Extension Kit:

event-commit-to-egate

Syntax

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

Description

event-commit-to-egate marks all messages as revealed for the internal work slice to pickup.

Parameters

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

Return Values

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

Throws

None.

Additional information

This function is used after the messages sent using **event-sent-to-egate-no-commit** are ready to be revealed.

event-rollback-to-egate

Syntax

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

Description

event-rollback-to-egate rolls back data that has been inserted but not revealed.

Parameters

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

Return Values

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

Throws

None.

Additional information

This function is used in conjunction with **event-send-to-egate-no-commit**. Once the message has been revealed, it can not be rolled back.

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

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.

event-send-to-egate-ignore-shutdown

Syntax

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

Description

event-send-to-egate-ignore-shutdown behaves similar to event-send-to-egate, except that if there is a shutdown issue, **event-sent-to-egate-ignore-shutdown** ignores it..

Parameters

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

Return Values

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

Throws

None.

event-send-to-egate-no-commit

Syntax

```
(event-send-to-egate-no-commit message)
```

Description

event-send-to-egate-no-commit temporarily stores messages and does not allow the internal work slice to access them. The internal work slice cannot access the messages until they are marked as revealed. Access is only allowed once the external translation calls event-commit-to-egate, which then reveals the messages to the internal work slice.

Parameters

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

Return Values

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

Throws

None.

Additional information

This function can be called by any of the Generic e*Way functions when it is necessary to send data to the e*Gate system without committing it.

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.

insert-exchange-data-event

Syntax

```
(insert-exchange-data-event)
```

Description

insert-exchange-data-event inserts an exchange data Event independent of the set schedules, and triggers the exchange data function.

Parameters

None.

Return Values

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

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.

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.

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 76**). 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.

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.

---

waiting-to-shutdown

Syntax

```
(waiting-to-shutdown)
```

Description

> **waiting-to-shutdown** informs the external workslice code whether or not someone has issued a shut down command.

Parameters

> None.

Return Values

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

Throws

> None.

5.2 Extension Functions

Two functions, **invoke** and **load-interface**, are particularly useful to developers extending e*Gate. The are included in this manual for easy reference, and are also discussed in the *Monk Developer's Reference Guide*.

- **invoke** on page 87

- **load-interface** on page 88

invoke

Syntax

```
(invoke object string [params...])
```

Description

invoke calls the function contained in the interface handle, passing the function name and parameter values as input.

Parameters

Name	Type	Description
object	handle	An interface handle returned by the **load-interface** function.
string	string	The name of the function invoked.
params...	varies, depending on the function invoked	Optional. The parameter(s) specified is dependent upon the argument list in the function invoked.

Return Values

Vector

A vector of the values returned by the call to the specified object.

Additional Information

The **invoke** function is a generic interface to a set of functions within a dll. The interface dll must use the architecture and protocols defined in the **stcextif.h** file (see **"Header File - stcextif.h" on page 12** for more information), and must first be loaded via the **load-interface** function (see **load-interface** on page 88).

An object that can be called by the **invoke** function can optionally be called using the object's name alone. For example, the following are equivalent:

```
(invoke my_object my_function)
```

```
(my_object my_function)
```

load-interface

Syntax

```
(load-interface dll_file [init_fn])
```

Description

load-interface loads a dll. The dll must adhere to the architecture and protocols defined in the **stcextif.h** file (see **"Header File - stcextif.h" on page 12** for more information).

Parameters

Name	Type	Description
dll_file	string	The path to the dll to be loaded.
init_fn	string	The name of the init function called. Optional.

Return Values

Returns an interface handle.

Examples

```
(define obj (load-interface "sample_ext.dll"))
```

<div align="right">

Chapter 6

</div>

Configuring the e*Way with the Schema Designer

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

6.1 Implementing the Generic e*Way

After you have created the extension DLL, any required Monk functions, 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.

6.1.1 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 that the files you wish to commit to the e*Gate schema are accessible from the same system as the Schema Designer GUI, 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, login into 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 you want to commit the file and click **Select**. Select the directory according to the table below:

Table 5 Schema directories

For a file of this type…	…commit to this directory
.def	/configs/stcewgenericmonk
.monk (e*Way functions)	monk_scripts/**eway_name** (We recommend that you create a separate directory for your custom e*Way scripts.)
.dll	/bin

Any ETD (**.ssc**) and Collaboration Rules (**.tsc**) files that you create for this e*Way should be stored in the schema within the **/monk_scripts/common** directory, but you do not need to commit any such files manually if you create them using the e*Gate ETD or Collaboration Rules Editors. If you use another editor to create these files (such as **Notepad**, **Wordpad**, or **vi**), you must commit the files manually.

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 Help system.*

6.1.2 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 **stcewgenericmonk.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**.

Figure 17 e*Way Template Selection
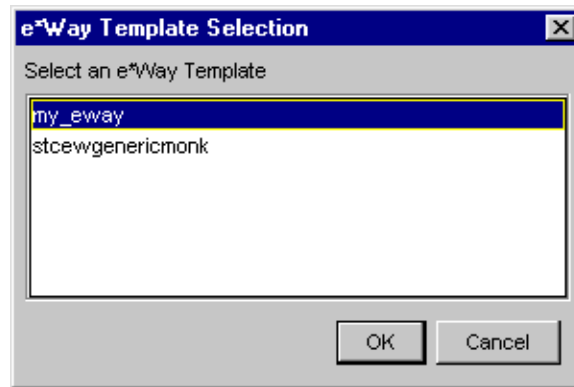
7   The e*Way Editor will launch. You are ready to configure the e*Way; continue with the next section.

6.1.3 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 (shown in the figure below).

Figure 18 Edit Settings
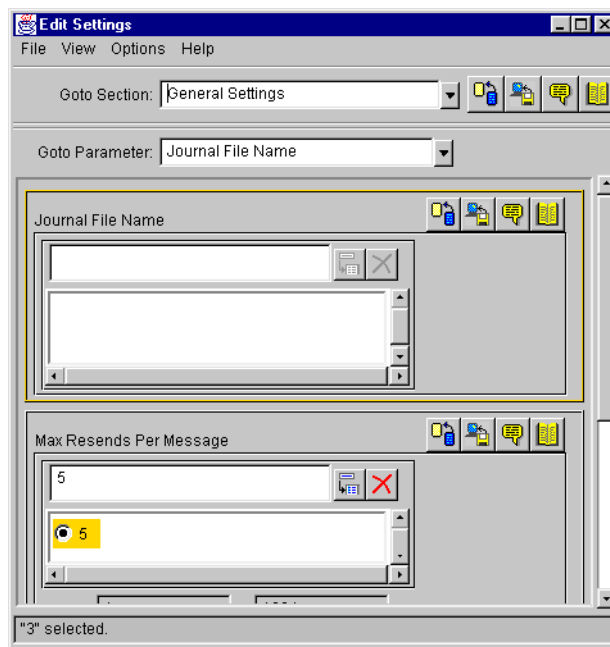
Use the e*Way Editor to make any configuration changes you require. For more information about configuring e*Ways or how to use the e*Way Editor, see the *e*Gate Integrator User's Guide.*

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 system.*

6.1.4   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 login 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.

Index

A

accessing parameter values within Monk **56**
Additional Path parameter **70**
addref method **19**
ASCII codes, displaying in different formats **42**
Auxiliary Library Directories parameter **71**

B

basic steps to extend a generic e*Way **11**

C

.cfg file **50**
(char) keyword **36**
character parameter syntax **32**
comments
    within the .def file **32**
    within the configuration file **50**
components **8**
configuration **50**, **58**
configuration definition file **8**
configuration files **50**
configuration parameters
    accessing within Monk environment **56**
    Additional Path **70**
    Auxiliary Library Directories **71**
    Down Timeout **61**
    Exchange Data Interval **61**
    Exchange Data With External Function **73**
    External Connection Establishment Function **74**
    External Connection Shutdown Function **75**
    External Connection Verification Function **74**
    Forward External Errors **59**
    Journal File Name **58**
    Max Failed Messages **59**
    Max Resends Per Message **59**
    Monk Environment Initialization File **71**
    Negative Acknowledgment Function **76**
    Positive Acknowledgement Function **75**
    Process Outgoing Message Function **72**
    Resend Timeout **62**
    Shutdown Command Notification Function **76**
    Start Exchange Data Schedule **61**
    Startup Function **72**
    Stop Exchange Data Schedule **60**
    Up Timeout **61**
    Zero Wait Between Successful Exchanges **62**
const keyword **39**

D

(date) keyword **36**
debugging the .def file **52**
delim keywords **45**, **50**
description keyword **34**
displaying ASCII codes **42**
Down Timeout parameter **61**

E

encrypting string parameters **44**
error messages in .def file parsing **53**
escape character, using **32**
event-commit-to-egate **80**
event-rollback-to-egate **81**
event-send-to-egate function **82**
event-send-to-egate-ignore-shutdown **82**
event-send-to-egate-no-commit **83**
Exchange Data Interval parameter **61**
exchange data with external function **77**
Exchange Data with External Function parameter **73**
External Connection Establishment Function parameter **74**
External Connection Establishment Function template **78**
External Connection Shutdown Function parameter **75**
External Connection Shutdown Function template **78**
External Connection Verification Function parameter **74**
External Connection Verification Function template **78**
EXTIF_CHARBLOB **15**
EXTIF_PARAM_ **15**
EXTIF_VECTOR **16**
extIFParamTypes_ **14**
ExtIFResult_ **14**

F

(factor) keyword **43**
floating-point numbers **32**
formats, displaying parameters in varying **42**
Forward External Errors parameter **59**
freeargs method **19**