e*Index Global Identifier Product Suite

e*Index™ Global Identifier Technical Reference

Release 5.0.5 for Schema Run-time Environment (SRE)

SEEBEYOND

Table of Contents

Introduction

About this Chapter

Overview

This Introduction welcomes new and experienced e*Index™ Global Identifier (e*Index) users and explains how to use this guide. An overview of e*Index APIs is also provided.

The following diagram illustrates the contents of each major topic in this chapter. For the page numbers on which specific topics appear, see the next page of this chapter.

Welcome	Learn where to start in this guide if you are a new or experienced user
About this Guide	Learn how to use this guide
About e*Index	Learn about e*Index and the API library for e*Index
Additional Resources	Learn about other e*Index publications you may wish to review

What's Inside

This chapter provides information related to the topics listed below.

Welcome

What is e*Index?

e*Index is SeeBeyond's enterprise-wide master person index, designed to help you maintain information about your members, and to ensure that the information is the most current and accurate data available. e*Index works together with SeeBeyond's e*Gate™ Integrator and a Database e*Way™ to transfer information among various computer systems within your business. Using the API functions provided with e*Index, you can create your own Monk scripts to transfer information into and out of the e*Index database, and to ensure that the data you transfer is the most up-to-date and accurate information available.

To New Users

If you are new to e*Index, you should browse through this guide before you begin to use the Monk APIs in your Collaboration scripts. Please pay particular attention to the "Learning About" sections provided at the beginning of each chapter. These sections are designed to provide background and explanatory information you may need to understand. After reading this overview information, you will be ready to work with the standard set of e*Index APIs to create your own customized Collaboration scripts.

To Established Users

If you are a more advanced e*Index user, you may prefer to use this guide as a quick reference to find information about forgotten or unfamiliar Monk APIs or functions. If you know what you need to do, but can't remember exactly how to do it, you can easily find what you need in the Table of Contents. Or, you can browse through the guide and find the appropriate background information or API description by scanning headings and titles.

About this Guide

What is the Purpose of this Guide?

This guide provides the information you need to include e*Index APIs in your e*Gate Collaboration scripts so you can transfer information into and out of the e*Index database. It also provides an overview of the data processing flow for e*Index, the e*Index database, and the default data structure. This guide describes each e*Index Monk API and provides examples of usage.

What is the Scope of this Guide?

This guide includes:

- A complete reference to each e*Index Monk API, including descriptions, syntax, parameters, return values, and examples

- An overview of how e*Index processes data

- A description of the e*Index database tables

- An overview of the e*Index sample schema, along with configuration tips

This guide does not explain how to perform any of the tasks listed below. For a list of publications that contain this information, see "Additional Resources" at the end of this chapter.

- How to use the GUI front-end for e*Index applications

- How to install and configure e*Index

- How to implement e*Gate Schemas, Event Definitions, or Collaborations

- How to create and implement Monk scripts

Who Should Use this Guide?

This guide should be read by any one who works with the e*Index schema, or who writes or modified Monk scripts using e*Index Monk APIs. To understand the information in this guide, a reasonably good understanding of the following areas is recommended:

- e*Gate for SRE

- The Database e*Way specific to the database platform in use (Oracle, Sybase, or ODBC for Microsoft SQL Server)

- Data transfers using e*Gate

■   The data formats used by the systems you work with

■   The Monk scripting language

■   The database platform used by e*Index

How Should this Guide be Used?

Before you begin to use this guide:

1   You may want to review information presented in other e*Index guides. See "Additional Resources" at the end of this chapter for a list of available publications.

2   Familiarize yourself with the information presented in "About e*Index," provided later in this chapter.

3   Skim through this guide to familiarize yourself with the locations of essential functions you need to use or API descriptions you need to understand. Each chapter begins with a simple graphic that identifies the information contained in the chapter. The second page of each chapter contains a list of topics and corresponding page numbers.

How is this Guide Organized?

This guide is divided into three chapters that cover the topics shown below.

Chapter	Topics
Chapter 1, Introduction	■ Welcome ■ About this Guide ■ Learning About e*Index ■ Additional Resources
Chapter 2, Understanding Operational Processes	■ Learning About e*Index ■ Learning About Event Processing ■ Learning About the Default ETD ■ Learning About e*Index Database Tables
Chapter 3, Customizing e*Index	■ Learning About the e*Index Sample Schema ■ Learning About e*Way Configuration Parameters
Chapter 4, e*Index Monk APIs	■ Learning About e*Index Monk APIs ■ e*Index Monk API Descriptions ■ Standard e*Index Monk API Descriptions
Chapter 5, e*Index Monk Functions	■ Learning About e*Index Monk Functions ■ e*Index Monk Function Descriptions

What Conventions are Used in this Guide?

Before you read this guide, it's important to understand the typographic, icon, special notation, and other conventions used in this guide.

Typographic Conventions

The following typographic conventions are used in this and other e*Index publications.

Item	Convention	Example
Book titles	Title caps, italic	See the *e*Index Global Identifier User's Guide*
Chapter titles (and section titles within chapters)	Title caps, in quotation marks	See Chapter 4, " e*Index Monk Functions" See "e*Index Monk API Descriptions" later in this chapter
New terms	Italic	A set of *Monk lists* is predefined to help you perform a variety of functions.
Typed command syntax	Bold for constants Bold-italic and lower case for user-specified values Brackets denote optional values	Type **mkdir** ***release***

Icon and Special Notation Conventions

The following conventions are used in this and other e*Index publications to identify special types of information.

Icon or Notation	Type of information
Note	Supplemental information that is helpful to know, but not essential to completing a particular task.
Tip	Information that helps you to apply techniques and procedures described in the text to your specific needs. May also suggest alternative methods.
Important!	Information that is essential to the completion of a task.
Caution!	Advises you to take specific action to avoid loss of data.
▶	Indicates the beginning of a step-by-step instruction.
✓	Specifies a task to perform before you begin a step-by-step instruction.
ⓘ	Indicates a cross-reference to other sections of the guide or to other publications.

Learning About e*Index

Overview

This section of the chapter provides an overview of the information you need to know in order to customize the way e*Index processes data.

What is e*Index?

e*Index is an enterprise-wide index that maintains data and enables accurate identification of the members who participate throughout a business enterprise. e*Index centralizes the identification and demographic information for all members in one shared index, so that all individual look-ups and data retrievals obtain the most recent information on each person. e*Index uses a single data source regardless of the location or computer system from which member information is received. e*Index is able to cross-reference a member's records throughout several systems by assigning each member a unique global identifier.

e*Index was designed specifically to support geographically dispersed sites and disparate information systems across an enterprise, as well as various applications from multiple vendors. Maintaining a centralized database for multiple systems enables e*Index to integrate data in the enterprise while allowing local systems to continue operating independently.

How is Data Processed?

e*Index works with two other SeeBeyond components, e*Gate and a Database e*Way, to transfer data into and out of the e*Index database. Through additional e*Ways, the information in the e*Index database can be shared with external systems throughout your business enterprise. You can customize the way data is processed by including specific commands in your e*Way Collaboration scripts. These commands are written in the Monk scripting language, specifically for use with e*Gate e*Ways. You can also customize the format of the data by adding certain Monk functions to the file **ui-custom.monk**.

*Note: ui-custom.monk is located in the Monk library in your e*Gate environment. This file contains the commands that pull the demographic, transaction, alias, address, and telephone information that creates the Monk lists that are used as parameters for e*Index APIs.*

For more information about e*Gate, e*Ways, and Events, see your suite of e*Gate user's guides. For more information about the Database e*Way, see the user's guide for the e*Way Intelligent Adapter for Oracle, Sybase, or ODBC, depending on the database platform of the e*Index database.

What is Monk?

Monk is a special programming language developed by SeeBeyond that you can use to define, identify, and process Events (data). Monk performs these activities in relation to SeeBeyond's e*Gate Integrator. Monk also provides the ability to retain information on the actual structure of the Events that pass through e*Gate so you can easily and quickly manipulate your Event Type Definitions (ETDs). Adding Monk to your e*Ways lets you expand the basic set of schema tools available to you through e*Gate and e*Index. The architecture of Monk is based on the Scheme Programming language developed by MIT.

For more information about the Monk scripting language, see your *Monk Developer's Reference.*

What are e*Index Monk APIs?

SeeBeyond provides a set of *e*Index Monk APIs*, or Application Program Interfaces, with the e*Index application. The e*Index APIs are provided in the Monk scripting language to allow you greater flexibility when designing your e*Ways for e*Index. These APIs are sets of routines that you can call in a Monk script to perform functions specific to the e*Index database. The APIs include routines to perform tasks such as finding a person based on their UID or local ID, inserting demographic or alias information into a person's records, merging records, committing or rolling back database transactions, and so on.

For more *information* about the features and functions of e*Index, see your *e*Index Global Identifier User Guide.*

What are e*Index Monk Functions?

*e*Index Monk functions* are expressions that you can use to manipulate the data in an Event so it is formatted in a way that the target application can read. These functions allow you to perform specific operations on a parameter or series of parameters in a Monk API. The available e*Index Monk functions allow you to strip non-numeric characters from a telephone number or social security number, filter out unwanted dates, remove unwanted spaces from a field, and check a string for specific characters.

Monk functions are typically given names that reveal the function's purpose. For example, the strip-phone function strips a telephone number of any non-numeric characters, such as dashes or parenthesis.

Additional Resources

SeeBeyond has developed a suite of e*Index user's guides and related publications that are distributed in an electronic library.

■ *e*Index Global Identifier User's Guide*
Helps e*Index quality workstation users to perform database maintenance tasks, such as merging and unmerging records, finding and resolving potential duplicates, adding and updating records, and viewing the audit trail.

■ *e*Index Administrator User's Guide*
Helps system administrators configure system parameters, customize e*Index, work with Vality rule set files, and processing codes. This guide also describes how to maintain the information in the database that is used to populate the drop-down lists in the e*Index.

■ *e*Index Security User's Guide*
Helps system administrators add users and user groups to e*Index, to grant security permissions to users and user groups, to maintain user and user group information, and to configure certain system parameters.

■ *e*Index Initial Load User's Guide*
Provides the background information and instructions that system and database administrators need in order to load legacy data into the e*Index database, including a description of the expected data format and the schema files included with the load program.

■ *Working with Reports for e*Index Global Identifier*
Provides background information about the GUI and standard reports provided with e*Index, and explains how to modify and run the standard reports (for Oracle installations only).

■ *e*Index Global Identifier Installation Guide*
Helps system and database administrators install a new e*Index environment for the current release, including e*Index schema files, the e*Index GUI, and database installation.

■ *e*Index Global Identifier Upgrade Guide*
Helps system and database administrators upgrade an existing e*Index environment to the most current release, including e*Index Schema files, the e*Index GUI, and database upgrades.

■ *Java Programmer's Guide for e*Index Active Integration*
Provides background and implementation information about the Java APIs for e*Index Active Integration. This guide also provides a complete listing of e*Index Java functions, along with a description, parameters, syntax, return values, and examples for each.

Understanding Operational Processes

About this Chapter

Overview

This chapter describes and illustrates the processing flow of Events (data) to and from e*Index, providing background information to help you work with the e*Index Monk API library.

The following diagram illustrates the contents of each major topic in this chapter. For the page numbers on which specific topics appear, see the next page of this chapter.

About e*Index	Learn about the functions and components of e*Index
About Event Processing	Learn how events are transmitted into and out of e*Index, and how inbound data is transformed
About the Database	Learn about the tables used in event processing and view a physical data model
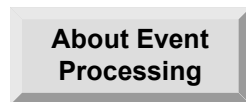
What's Inside

This chapter provides information related to the topics listed below.

Learning About e*Index

Overview

This section of the chapter provides an overview of the processes and functions of e*Index, and the components that comprise the e*Index system.

About e*Index Components

e*Index includes of several different components, each acting independently of the others but working together to provide accurate data maintenance and identification. These components interact with other SeeBeyond integration products, such as the e*Gate Integrator and the Database e*Ways, and through them with the external data processing products and systems used throughout a business enterprise. When one local system transfers data to another using e*Gate, e*Index consults the e*Index database and retrieves the appropriate local identifier to identify the member. This is a transparent process, allowing each computer system within this network to continue to use its own local identifiers.
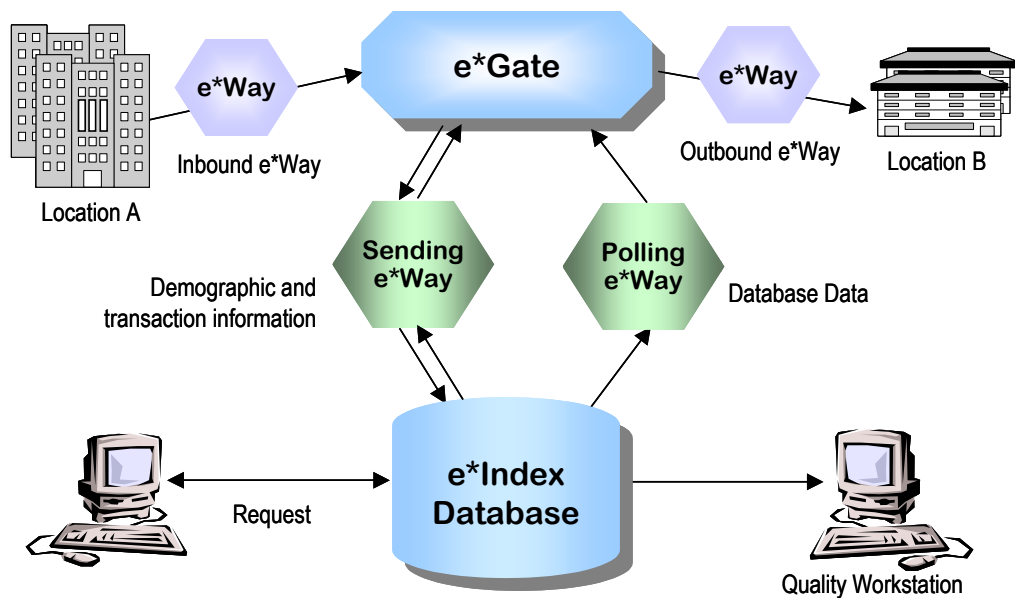
The components of the e*Index system include:

■ **e*Index Database**
The e*Index database is a relational database used to store member data, security information, code table values, and configuration information. The database also stores all incoming and outgoing Events. You

configure the database using the e*Index Administrator GUI on the quality workstation.

- **e*Index API**
  The Monk-wrapped C functions are designed to help you access and modify the e*Index database. The Monk APIs use the capabilities of the Database e*Way to connect with the database, and to manipulate and transform the data that moves through the e*Index system. The Vality® INTEGRITY™ matching algorithm logic is called in the C code functions.

- **e*Index Sending e*Way**
  This e*Way is based on the Database e*Ways, and works with the Database e*Ways' database connection capabilities. The e*Index sending e*Way transmits the data received from external sources into the e*Index database, and then transmits the information back out to e*Gate with a unique global identifier (UID) attached. You may have several e*Ways sending data to the database. The components of this e*Way include:

  - the executable file, **stcewgenericmonk.exe**

  - configuration files (the sample configuration file included with your installation is **uidb.cfg**)

  - e*Way Monk functions and APIs

  - Monk external function scripts (provided in the file **ui-stdver-eway-funcs.monk**)

- **e*Index Out-Queue Polling e*Way**
  This e*Way queries the *ui_msg_detail* table (the out queue) in the e*Index database for outgoing Events. The polling e*Way then sends the events to e*Gate to be routed to the appropriate external systems. Most of the Events in the out queue originate in the e*Index GUI. The e*Way accesses the e*Index database using e*Index APIs. The components of this e*Way include:

  - the executable file, **stcewgenericmonk.exe**

  - configuration files (the sample configuration file included with your installation is **uipoll.cfg**)

  - e*Way Monk functions and APIs

  - Monk external function scripts (provided in the file **ui-stdver-eway-funcs.monk**)

Notes:

- *The e*Way files included with the sample e*Index schema are listed in Chapter 3, "Customizing e*Index". This chapter also includes information on configuring your e*Ways for e*Index.*

- *The e*Way files are provided in a sample schema. You can customize the sample schema to create your production schema, or you can create a new schema for your production environment.*

■ **Quality Workstation**
The Quality Workstation is where the e*Index GUI resides. On the Quality Workstation, you can monitor and maintain member data and transactions, print reports, and perform manual changes to member information. You can also add processing codes, create the data elements that populate the drop-down lists for e*Index, and configure certain e*Index processing attributes, such as data formatting rules, GUI window appearance, search limits, and so on. Security for e*Index also resides on the Quality Workstation.

About e*Index Functionality

This section describes the basic functions of e*Index. e*Index was designed to uniquely identify, match, and maintain member information throughout your business enterprise. The e*Index Monk APIs provide the following functionality.

■ **Unique Identifier**
e*Index assigns a unique, enterprise-wide identifier to each member added to the database. This identifier is known as the global identifier, or UID. e*Index uses the UID to cross-reference a member's local IDs throughout the system. See "About Inbound Event Processing" later in this chapter for a description of the identification process.

■ **Audit Trail**
The system provides full audit capabilities. All changes to a member's demographic data are recorded in the history table. This allows e*Index to generate an audit trail that compares the demographic information before and after each modification.

■ **Data Maintenance**
e*Index provides the ability to add, update, deactivate, and delete data within the database tables. Data updates from external systems can occur in real-time or as batch processes.

■ **Search**
You can look up demographic information from the *ui_person* table using various search criteria. You can perform a search for a specific member or a set of members. Each record that is returned as a possible match is assigned a matching probability weight, which indicates how closely each record matches the search criteria you specified.

■ **Potential Duplicates**
Using algorithm matching logic, e*Index can to identify potential duplicate records, and provides the functionality to correct duplication. A new record is considered a potential duplicate of an existing record when the matching probability of the two records falls within a range that you specify (for more information, see "Defining Control Key Values" in chapter 5 of the e**Index Administrator User's Guide*). You can resolve

potential duplicate records by either merging the records in question or removing their potential duplicate flags.

■ **Merge Demographic Records**
Member records can be merged if they are found to be actual duplicates of one another. To merge two records, you need to specify the UIDs of the records to be merged. The record that is not kept after a merge transaction receives a status of **Merged**. The information from the old record is retained in the database, providing the ability to unmerge the two records if necessary.

Note: *Each time a member record is updated, added, merged, or unmerged from the GUI front-end, an Event is placed in the ui_msg_detail table so the e*Index polling e*Way can retrieve the message, making the modified information available to external systems. When these transactions occur through the backend, an Event is returned through the e*Index sending e*Way with the member's UID attached. In the case of merges, or other transactions that you specify, an Event is also placed in the ui_msg_detail table. For more information, see "Learning About Event Processing" next in this chapter.*

Learning About Event Processing

Overview

This section of the chapter provides a summary of how inbound and outbound Events are processed in the e*Index environment. The diagram below illustrates the flow of information through the e*Index system.
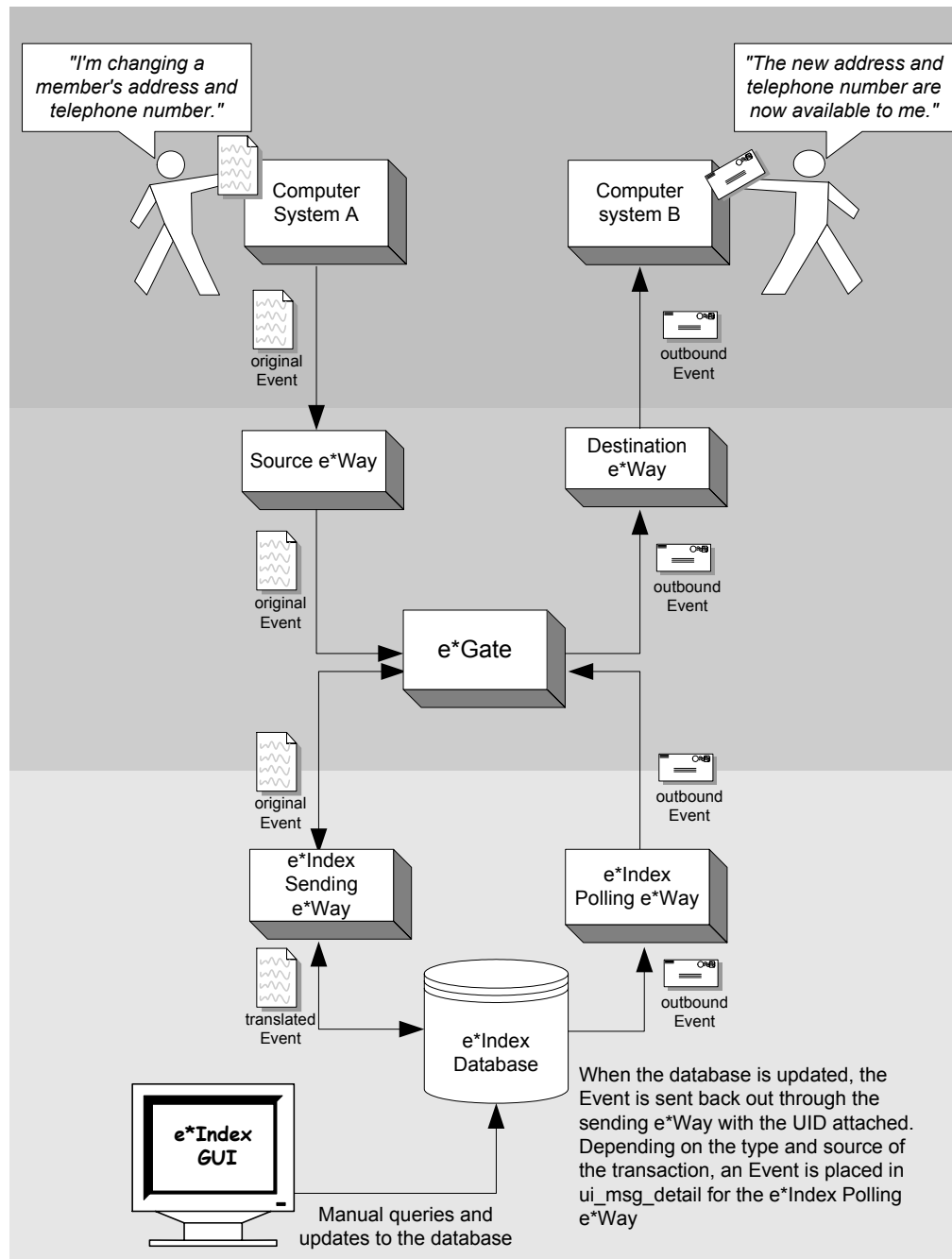
About Inbound Events

An inbound Event refers to the transmission of data from external systems to e*Gate and then to the e*Index database. These Events may be sent into the database via a number of e*Index sending e*Ways. The steps below describe how inbound Events are processed.

■ Events are created in an external system, and the enveloped Event is transmitted to e*Gate via that system's e*Way.

■ e*Gate identifies the Event and the appropriate e*Index e*Way to which the Event should be sent. The Event is then routed to the appropriate e*Index e*Way for processing.

■ The Event is modified into the appropriate format for the e*Index database, and certain validations are performed against the data elements of the Event to ensure accurate delivery. The Event is validated using the Monk scripts in the e*Way's Collaboration file and other information stored in the e*Index database.

■ If the Event was successfully transmitted to the e*Index database, the appropriate changes to the database are processed and a positive acknowledgement (ack) is returned to the sending system. If the Event was not successfully transmitted, a negative acknowledgement (nack) is returned, and the Event is resent.

■ Inbound Events are stored and tracked in the e*Gate log files. Inbound merge Events generate merge messages, which are then placed in the *ui_msg_detail* table for the e*Index polling e*Way.

■ If the inbound Event causes a member record to be added, updated, deactivated, merged, or unmerged, the member's UID is attached to the Event, and the Event is sent back out through the same e*Way, making the new information available to external systems.
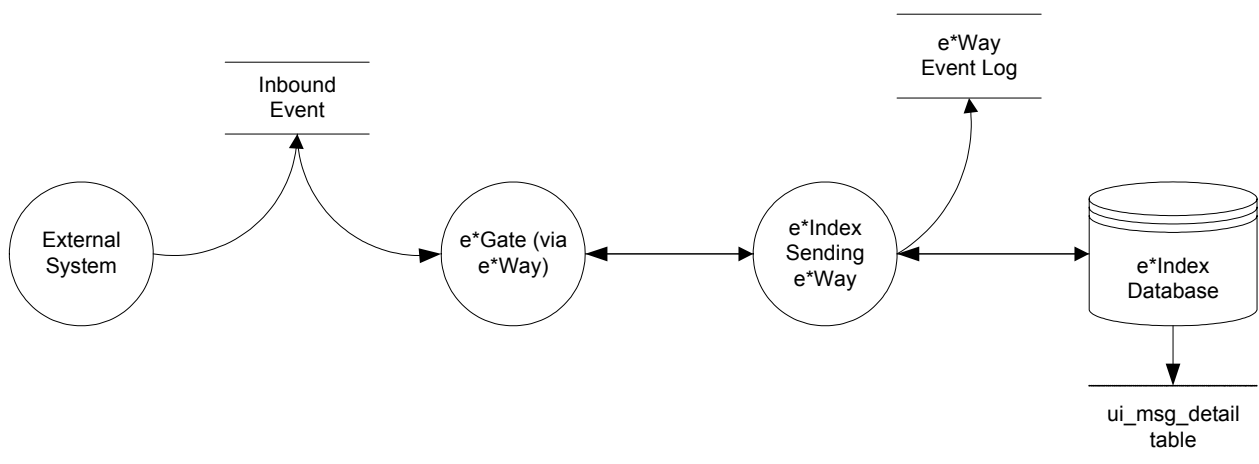
About Outbound Events

An outbound Event refers to the transmission of data from the e*Index database to any external system. Outbound Events are generated when updates are made to the database. These Events fall into two different categories. Events received through the sending e*Ways are sent back out through those e*Ways (as described earlier in "About Inbound Events"). Event updates made through the GUI are placed in the *ui_msg_detail* table and made available for retrieval by the polling e*Way. This section describes how Events placed in the *ui_msg_detail* table are processed.

Note: *You should always configure the polling e*Way to retrieve messages from* ui_msg_detail, *even if the messages are simply sent to an eater file. Otherwise the* ui_msg_detail *table will continue to grow, slowing down transaction processing.*

- Outgoing Events are generated in the e*Index database, and are stored in the *ui_msg_detail* table. The polling e*Way continually checks the table for outgoing Events.

- When the polling e*Way finds an Event to retrieve, it flags the Event in *ui_msg_detail* using the value in the msg_id field to identify the Event. The polling e*Way sends the Event to e*Gate for routing.

- e*Gate identifies the Event and the external systems to which it should be sent, and then routes the Event to the appropriate e*Ways for processing.

- The receiving systems' e*Ways modify the Event into the appropriate format, and perform certain validations against the data elements of the Event to ensure accurate delivery. The e*Ways perform these validations using the Monk scripts.

- If the Event was successfully sent to the receiving systems, a positive acknowledgement (ack) is returned, and the Event is removed from the *ui_msg_detail* table. If the Event was not successfully sent, a negative acknowledgement (nack) is returned, and the Event is resent until the maximum number of resends is reached. The Event is flagged with an error if it cannot be successfully sent. Both the ack and nack functions use the msg_id field in the Event to identify the record. The msg_id corresponds with the ui_msg_header_id column in *ui_msg_detail*.

- Outbound Events are stored and tracked in the e*Gate log files.
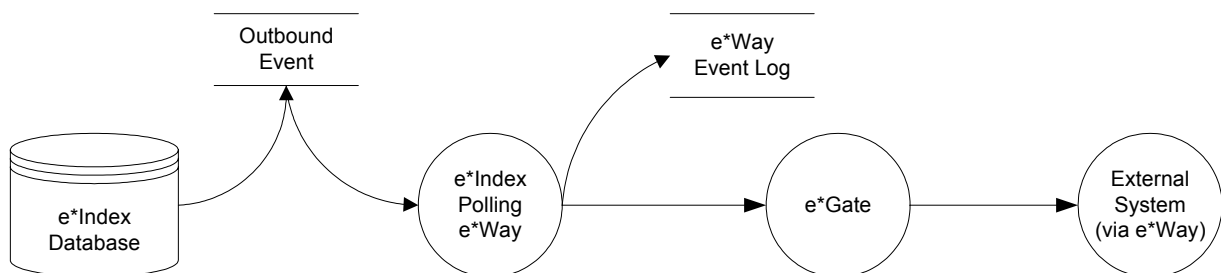
About Inbound Event Processing Logic

When demographic records are transmitted to e*Index, a series of processes are performed to ensure that accurate and current data is maintained in the database. In the default configuration, these processes are called by the e*Index Monk function **ui-process-person**, which is defined in the file *ui-process-person.monk*. The steps performed by e*Index using the *default configuration* are outlined below, and the diagrams on the following pages illustrate the Event processing flow. The processing steps performed in your e*Index environment may vary from this depending on how you customize **ui-process-person**.

1 When an Event containing member demographic data is received by e*Index, a search is performed for any existing records in the *ui_local_id* table with the same local ID and system as those contained in the Event (by calling **ui-get-uid**). This search only includes records with a status of **A**, meaning only active records are included. If a matching record is found, an existing UID is returned.

2 If an existing record is found with the same system and local ID as the incoming Event, it is assumed that the two records represent the same person. Using the UID of the existing record, e*Index performs an update of the member's demographic information in the *ui_person* table (by calling **ui-update-person**).

---

*Note: In the default setup provided for e*Index, the alias, address, telephone, and non-unique ID information from the incoming Event is processed outside of the call to* **ui-process-person** *(as demonstrated in the default configuration file* **uidb.dsc***).*

- If the update does not make any changes to the member's information, no further processing is required and the existing UID is returned.
- If there are changes member information, the updated record is inserted into the *ui_person_history* table to provide an audit trail of the changes.
- If there are changes to the member's last name, first name, middle name, date of birth, gender, or SSN, then potential duplicates are re-evaluated for the updated record.

3 If no records are found that match the member's system and local identifier, a second search is performed using the matching algorithm. A search is performed on each of the following combinations of criteria to determine a matching probability.

- Last name phonetic code and first name phonetic code
- First name phonetic code and date of birth and gender
- Last name phonetic code and mother's maiden name

- Social security number

Each record that is returned from the search is weighted against the demographic information in the inbound Event.

Important! *The criteria combinations described above are configurable, so you can modify the combinations of data used for the search. For more information, see "Configuring Queries" in your* e*Index Administrator User's Guide.

---

4 After the search is performed, the number of resulting records is calculated.

- If a record or records are returned from the search with a matching probability weight above the matching threshold, e*Index performs exact match processing (see Step 5).

- If no matching records are found, the inbound Event is treated as a new record and a new UID is generated. A new member record is inserted into the *ui_person* table and the *ui_person_history* table. In addition, the new UID is inserted into the *ui_local_id* table with the new system and local ID.

5 If records were found within the required match probability range, exact match processing is performed as follows:

- If only one record is returned from this search with a matching probability that is equal to or greater than the match threshold, then additional checking is performed to verify whether the records originated from the same system (see Step 6).

- If more than one record is returned with a matching probability that is equal to or greater than the match threshold and exact matching is *disabled*, then the record with the highest matching probability is checked against the incoming Event to see if they originated from the same system (see Step 6).

- If more than one record is returned with a matching probability that is equal to or greater than the match threshold and exact matching is *enabled*, then the member record is inserted as a new record and potential duplicate processing is performed (see Step 7).

- If no record is returned from the database search, or if none of the matching records have a weight in the exact match range, then the member record is inserted as a new record and potential duplicate processing is performed (see Step 7).

Note: *Exact matching is determined by the control key* 1XACTMTCH. *For more information about exact match processing, see "Defining Control Key Values" in your* e*Index Administrator User's Guide.

---

6 When records are checked for same system entries (using a call to **ui-get-local-id**), e*Index tries to retrieve an existing local ID from the *ui_local_id*

table using the system of the new record and the UID of the record that has the highest match weight.

- If a local ID is found, the new information originated from the same system but under a different local ID, indicating that the two records are not the same person. A new record is inserted, and the two records are considered to be potential duplicates and are inserted into the *ui_duplic* table. These records are marked as same system potential duplicates.

- If no local ID is found, then it is assumed that the two records represent the same person. The existing UID is inserted into the *ui_local_id* table with the new system and local ID. The member's demographic information is then updated using the same process as in step 2 above. If the assumed match option is on, the assumed match information is inserted into the *ui_assumed_match* table.

7 If a new record is inserted, all records that were returned from the search are weighed against the new record using the matching algorithm. If the matching probability weight of a record is greater than or equal to the potential duplicate threshold and is less than the maximum threshold, the record is flagged as a potential duplicate (for more information about thresholds, see "Defining Control Key Values" in your *e*Index Administrator User's Guide*). Records that are considered potential duplicates are inserted into the *ui_duplic* table and these records are tagged with a message stating the matching probability weight between each potential duplicate pair.

Note: *If a record has gone through same system checking as described in step 6, potential duplicates are only processed if exact matching is disabled.*

The flow charts on the following pages provide a visual representation of the processes performed by the e*Index Monk function **ui-process-person**. Figures 1 and 2 represent the primary flow of information. Figures 3 and 4 expand upon the update procedures that are performed in Figures 1 and 2 respectively.

Figure 1 – Inbound Message Processing

Note: *The numbers next to certain objects in the diagrams indicate the number of the step outlined in the previous section that corresponds with that portion of the diagram. These diagrams represent the default configuration, in which address, telephone, and alias information is processed outside of* **ui-process-person**. *You may customize this process so it differs from the following representations.*

```
              ┌─────────────────┐
             /  Message           \
            /  containing a         \
           |  system and local       |
            \  ID, plus demographic  /
             \  data, is received   /
              └────────┬──────────┘
                       │
         ①  ┌──────────▼──────────┐
            │ Lookup system and    │
            │ local ID in ui_local_id │
            │ table                │
            └──────────┬──────────┘
                       │
                    ◇──▼──◇
                  /   Are the    \
                 /  system and local ID \──── Yes ──┐
                 \   pair found?  /                 │
                  \◇───┬───◇/                        │
                       │                            ▼
                      No                  ②  ┌─────────────────┐
                       │                      │ Perform a         │
         ③  ┌──────────▼──────────┐           │ demographic       │
            │ Perform              │          │ update on the     │
            │ matching             │          │ existing record (see │
            │ algorithm search     │          │ Figure 3          │
            └──────────┬──────────┘           │ expansion)        │
                       │                       └─────────────────┘
                    ◇──▼──◇
                 /  Are records   \
                /  found with       \
               /  matching weights    \
               \  equal to or greater  /── No ──┐
                \  than the duplicate  /         │
                 \  threshold?  /                ▼
                  \◇───┬───◇/         ④  ┌─────────────────┐
                       │                  │ Perform a new     │
                      Yes                 │ person insert     │
                       │                  └────────┬────────┘
                       ▼                            │
                    ┌─────┐                          ▼
                    │  A  │                  (  Processing  )
                    └─────┘                  (  complete    )
```

Figure 2 – Inbound Message Processing (cont'd)

A

⑤ Were records found with matching weights above or equal to the match threshold?

1 record found

0 records found

More than 1 record found

Yes

⑥ Did matching records originate from the same system?

Yes

No

④ Perform new person insert

⑥ Perform 'same system' potential duplicate processing on matched record

Processing complete

⑥ Perform demographic update on matching record (see Figure 4 expansion)

Processing complete

Is exact match configuration enabled?

No

④ Perform new person insert

④ Perform new person insert

⑦ Process potential duplicates for records above and equal to the duplicate threshold

Processing complete

⑥ Did highest matching record originate from the same system?

Yes

No

④ Perform new person insert

⑥ Perform 'same system' potential duplicate processing on matched record

⑦ Perform potential duplicate processing on remaining matching records

Processing complete

⑥ Perform demographic update on matching record (see Figure 4 expansion)

⑦ Perform potential duplicate processing on remaining matching records

Processing complete

Figure 3 – Demographic Update Expansion

This diagram illustrates the update process performed in Figure 1.

```
[Find the existing    ] → <Are there changes      > —Yes→ [Update         ] → [Insert updated
 demographic            in demographic                     demographic         demographic
 record using its       information?>                      information]        information into
 UID]                        |                                                 history table]
                             No                                                     |
                             |                                              <Is DUPCHECK enabled?>
                             |                                                     |
                             |                                                    Yes
                             |                                                     |
                             |                                    No      <Were changes made
                             |                                            to significant* fields?>
                             |                                                     |
                             |                                                    Yes
                             |                                    No      [Perform potential
                             |                                             duplicate processing]
```
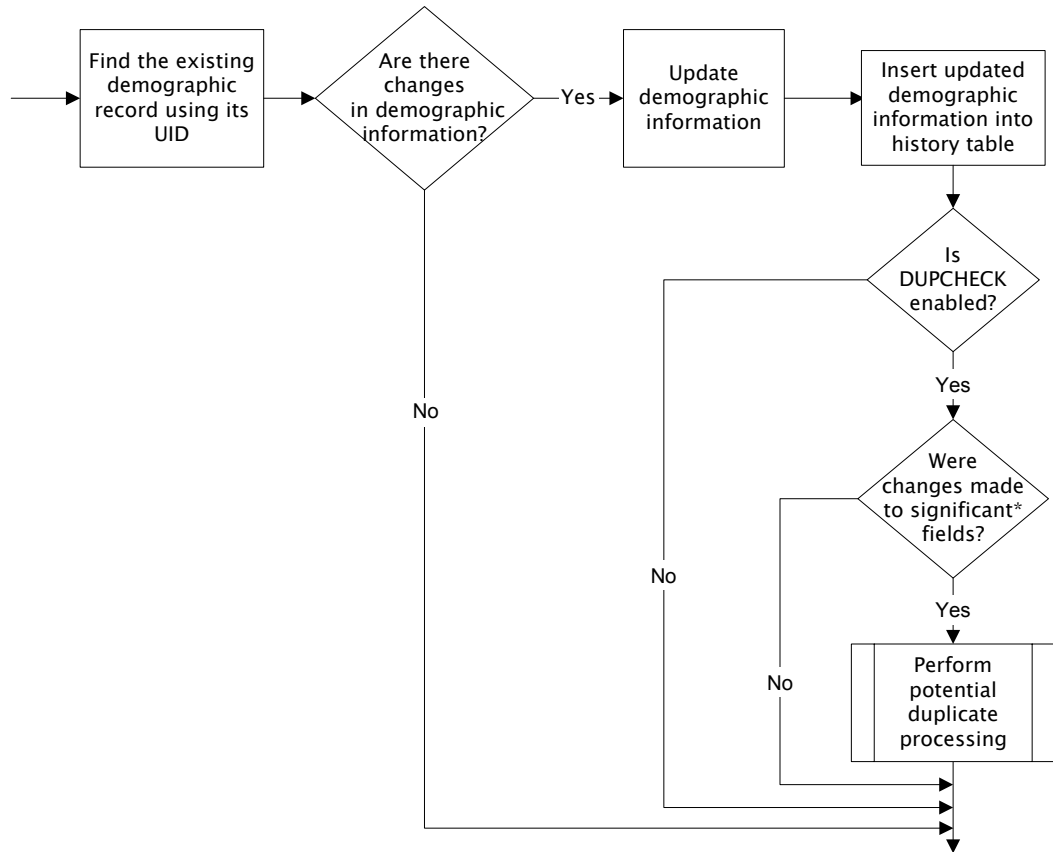
* Significant fields for potential duplicate processing include those defined by the
  configurable query. By default, these are last name, first name,
  middle name, date of birth, gender, and SSN

Figure 4 – Different System Demographic Update Expansion

This diagram illustrates the update process performed in Figure 2.

```
[Insert Local ID  ] → [Demographic      ] → <Is assumed   > —Yes→ [Insert records   ]
 into ui_local_id]     Update              match option          into
                       Processing          on?>                  assumed_match
                       (see Figure 3                             table]
                       expansion)]                                      No
```
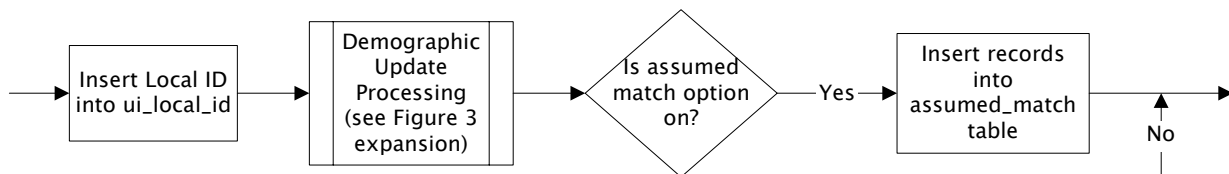
Learning About the Default ETD

Overview

This section describes the desired format of the data to be loaded into the database. You can translate the data from external systems into this format using the e*Ways of the external systems. You can also modify the default ETD (in the file **eiEvent.ssc**).

Formatting Guidelines

On order to comply with the sample ETD, the format of the data being transmitted into the e*Index database needs to be reformatted as follows:

- Each record consists of two types of information: Transaction details and identification details. These are delimited by a pair of angled brackets (<>).

- The records must be delimited. Each segment is separated by an ampersand (&), each field is separated by a pipe (|), and each sub-field is separated by a caret (^). When a field can repeat, each repetition is separated by a tilde (~). There are four segments, which appear as follows:

  EVNT segment <> ID segment & DEMO Segment & AUX segment <>

For information about each field, see the Input ETD Structure table below. Note that most fields in e*Index are configurable, so you are not restricted to the fields listed in the table. Also, some of the required fields are required because they are included in the configurable queries (sex, SSN, and dob). If you modify the queries, the required fields may change.

If you perform any of the customizations available in e*Index Administrator, make sure you perform an analysis of how those customizations affect the data you process from the e*Index schema.   The fields you display on the e*Index GUI should be the same as the fields you process through the e*Ways. Modify the ETD accordingly.

For more information about configuring the fields and queries for e*Index, see Chapter 5, "Customizing e*Index", in your *e*Index Administrator User's Guide*. The table below describes the default configuration for e*Index.

Note: This should be reviewed for each site to simplify where applicable. For example, fields for which the sending systems do not collect data can be removed.

Table 3-1 – Input ETD Structure

Transaction Details

Field	Description	Repeating?	Required?
segment_id	"EVNT"	No	Yes
msg_id	Always leave this field blank. It is populated in Events in the out-queue, which are created by GUI transactions.	No	No
event_type_code	Always leave this field blank. e*Index automatically determines the transaction type.	No	No
user_id	The user ID of the user who performed the transaction.	No	Yes
assigning_system	The system code for the system on which the transaction was performed.	No	Yes
source	The source code of the application on which the transaction was performed.	No	No
department	The department code for the transaction.	No	No
terminal_id	The ID of the terminal on which the transaction was performed.	No	No
date_of_event	The date the transaction occurred in the format **YYYY-MM-DD**.	No	No
time_of_event	The time the transaction occurred in the format **HH:MM:SS** using a 24-hour clock (for example, 23:59:59).	No	No

Identification Details

Field	Description	Repeating?	Required?
segment_id	"ID"	No	Yes
eid_1	Always leave this field blank. It is populated with the UID of the member after e*Index processes the Event.	No	No
local_id	The person's local identifier at a specified system. This field has two sub-fields: ID and system. For example, if the local ID **12345** was assigned within the system **SeeBeyond**, which has a code of **SBYN**, this field should appear like this: |12345^SBYN|	Yes	Yes

Field	Description	Repeating?	Required?
non_unique_id	The person's non-unique identifiers. This field has two sub-fields: ID and type. For example, if a person's account number is **003487**, and the type code for account is **ACCT**, this field should appear like this: |003487^ACCT| ***Note:*** *If non-unique ID information is included, then both an ID and an ID type must be included.*	Yes	No
segment_id	"DEMO"	No	Yes
person_category	The code for the person category to which the person is assigned.	No	No
person_name	The name of the person. This field consists of five sub-fields. **last_name:** The person's last name. **first_name**: The person's first name. **middle_name:** The person's middle name. **title:** The title code of the person's title. **suffix:** The suffix code of the person's suffix to their name.	No	 Yes Yes No No No
person_alias	The alias names for the person. This consists of three sub-fields: **last_name:** The alias last name. **first_name**: The alias first name. **middle_name:** The middle name of the alias.	Yes	No
alt_name	Alternative names associated with this person. This field consists of five sub-fields: **maiden_name:** The person's maiden name. **spouse_name**: The name of the person's spouse. **mother_name:** The name of the person's mother. **fathers_name:** The name of the person's father. **mothers_maiden_name:** The maiden name of the person's mother.	No	No

Field	Description	Repeating?	Required?
date_of_birth	The person's date of birth, in **YYYY-MM-DD** format.	No	Yes
time_of_birth	The time the person was born, in **HH:MM:SS** format on a 24-hour clock.	No	No
sex	The table code of the person's gender.	No	Yes
marital_status	The table code of the person's marital status.	No	No
SSN_number	The person's social security number in ###-##-#### format. *Note: If necessary, you can modify the format and length of this field. For more information, see "Configuring Country-Specific Options" in your* e*Index Administrator User's Guide.	No	Yes
driver_license	The driver license details for the person. This has two sub-fields: **state_country:** The state or country that issued the drivers license. **number:** The driver license number.	No	No
race	The table code of the person's race.	No	No
ethnic_group	The table code of the person's ethnic group.	No	No
nationality	The table code of the person's nationality.	No	No
religion	The table code of the person's religion.	No	No
language	The table code of the language spoken by the person.	No	No
death	Death information about the person. This field consists of three sub-fields: **flag:** An indicator of whether the person is deceased. Should be **Y** if deceased. **date_of_death:** If deceased, the date of death. **death_certificate_number:** The ID number on the death certificate.	No	No

Field	Description	Repeating?	Required?
birth_place	The location in which the person was born. This field consists of three sub-fields: **city:** The city where the person was born. **state:** The state where the person was born. **country:** The country code where the person was born.	No	No
vip	The table code of the person's VIP status.	No	No
veteran_status	The table code of the person's veteran status.	No	No
military	The military details for the person. This field consists of three sub-fields: **status:** The code of the person's military status. **rank_grade:** The person's military rank or grade. **branch:** The military branch in which the person has served.	No	No
citizenship	Citizenship for the person	No	No
pension	The pension details for the person. This field consists of two sub-fields: **number:** The person's pension card number. **expiration_date:** The expiration date of the pension card.	No	No
repatriation_number	The person's repatriation number.	No	No
district_of_residence	The code of the district of residence in which the person resides.	No	No
LGA_code	The LGA code for the person.	No	No
address	Address information for the person. This field consists of eleven of sub-fields: **type:** The table code for the type of address. **street_1:** The first line of the street address. **street_2:** The second line of the street address. **street_3:** The third line of the street address.	Yes	No

Field	Description	Repeating?	Required?
	street_4: The fourth line of the street address. **city:** The city or suburb of the address. **state_or_province:** State or province **zip:** The zip code of the address. **zip_ext:** The zip code extension of the address. **county:** The table code of the county in which the address is located. **country:** The table code of the address's country. *Note: If address information is included in an Event, the following fields must be present for each address: **type**, **street_1**, and **city**.*		
phone	Telephone information for the person. This field consists of three sub-fields: **type:** The table code of the telephone type. **phone_number:** The telephone number, with no punctuation characters. **phone_ext:** The extension to the telephone number. *Note: If telephone information is included in an Event, the **type** and **phone_number** fields must be present for each telephone number.*	Yes	No
segment_id	"AUX"	No	Yes
class	Five 20-character strings for site-specific purposes.	Yes maximum of five	No
string	Additional strings for site-specific purposes. The first six are a maximum of 40 characters. Strings seven to nine are a maximum of 100 characters. The tenth string is a maximum of 255 characters	Yes maximum of ten	No
date	Five miscellaneous date fields in **YYYY-MM-DD** format.	Yes maximum of five	No

Sample Inbound Event

Below is a sample data record that follows the default format described in the previous tables.

```
EVNT|||JJONES|SBYN|SBYN|||2001-06-15|10:20:24<>
ID||239487209^SC|23438742^ACC&DEMO|C|WARREN^ELIZABETH^JUNE^PHD
^|MILLER^ELIZABETH^J|MILLER^ANDREW^JULIE^MARK^MARTIN|1960-05-
14|15:01:08|F|M|555-44-4555|^|W|28||AG|ENGL|||^^|^^|N|N|^^|
USA|^||||H^2347 SHORELINE DRIVE^UNIT 3^^^SHEFFIELD^CT^09876^^
CAPE BURR^UNST~O^1490 WAYFIELD ROAD^FLOOR 5^SUITE 519^^CAPE
BURR^CT^09877^^^UNST|CH^9895557811^~CB^9895553214^1212&AUX|~~~
~|STANDARD MEMBERSHIP~~~~~~~~~|1999-09-12~2000-12-15~~~|<>
```

About Outbound Events

The Events that are placed in the outbound queue (the *ui_msg_detail*) table are similar in structure to the inbound events described earlier. There are two differences between the two structures. In the inbound Events, the local ID field consists of two sub-fields, ID and system. In outbound Events, the local ID field consists of three sub-fields, ID, system, and status. The status sub-field is an indicator of whether the local ID is active (**A**) or inactive (**D**).

The second difference is the addition of the ZEN segment at the end of outbound events. This segment includes the e-mail addresses of users who should receive notification of the Event. The outbound event segments appear as follows, with a carriage return at the end of the ZEN segment.

EVNT segment <> ID segment & DEMO Segment & AUX segment <>ZEN segment

The ZEN segment contains one repeating field, which is the e-mail address of the notification recipients. Below is a sample illustrating the structure of an outbound message. Notice that the local ID field has three sub-fields, and the ZEN segment is appended to the end.

```
EVNT|153|ADD|JJONES|SBYN|SBYN|||2001-06-15|10:20:24<>
ID|1001300021|239487209^SC^A|23438742^ACC&DEMO|C|WARREN^
ELIZABETH^JUNE^PHD^|MILLER^ELIZABETH^J|MILLER^ANDREW^JULIE^MAR
K^MARTIN|1960-05-14|15:01:08|F|M|555-44-4555|^|W|28||AG|ENGL||
|^^|^^|N|N|^^| USA|^||||H^2347 SHORELINE DRIVE^UNIT 3^^^
SHEFFIELD^CT^ 09876^^CAPE BURR^UNST~O^1490 WAYFIELD ROAD^FLOOR
5^SUITE 519^^CAPE BURR^CT^09877^^^UNST|CH^9895557811^~
CB^9895553214^1212&AUX|~~~~|STANDARD MEMBERSHIP~~~~~~~~~|1999-
09-12~2000-12-15~~~|<>ZEN|crazouli@here.org~gsmythe@here.org
```

Learning About the e*Index Database

Overview

This section of the chapter describes the e*Index database tables, and categorizes the tables by function. To view an illustration of the e*Index database, see the *e*Index Global Identifier Installation Guide*. The Oracle database model appears in chapter 3; the Sybase model appears in chapter 4; and the Microsoft SQL Server model appears in chapter 5.

Database Tables

The e*Index database contains several types of tables. Some of the tables store information for the e*Index Administrator, others store security information, and others are used for processing, storing, and tracking member information. Additional tables are used to configure the Vality INTEGRITY matching algorithm.

Member Information Tables

■ **ui_person**
The primary table in the e*Index database is the *ui_person* table. This table stores each member's demographic data, as well as their UID and phonetic codes. The primary key in this table is the u_id column. The *ui_person* table is linked to *ui_assumed_match*, *ui_aux_id*, *ui_aux_id_history*, *ui_address*, *ui_address_history*, *ui_mrg_trans*, *ui_phone*, *ui_phone_history*, *ui_duplic*, *ui_local_id*, *ui_local_id_history*, *ui_alias*, and *ui_alias_history* through the u_id column. It is also linked to *ui_transaction* through the transaction_no column.

■ **ui_person_history**
The *ui_person_history* table stores a history of transactions and demographic information for each member in your database. Each time a member record is updated, this table is populated with the new information. The information in this table provides a complete audit trail of each member, maintaining a complete before and after image of each transaction that occurs. The ui_person_history_id column is the primary key, and the table is linked to the *ui_transaction* table through the column transaction_no.

■ **ui_transaction**
This table maintains a sequential transaction number for each event processed through e*Index, and links all of the database tables used to store member information. The transaction_no column links the *ui_transaction* table to these database tables: *ui_person*, *ui_person_history*, *ui_assumed_match*, *ui_aux_id*, *ui_aux_id_history*, *ui_address*,

ui_address_history, *ui_mrg_trans*, *ui_phone*, ui_p*hone-history*, *ui_duplic*, *ui_local_id*, *ui_local_id_history*, *ui_alias*, and *ui_alias_history*. The transaction number unites the information from all of the member information tables for each transaction against each record.

■ **ui_alias**
The *ui_alias* table stores each member's alias information. The primary key in this table is the ui_alias_id column. If extensive searching is enabled through the configurable query, then alphanumeric searches check for data in both the *ui_person* and *ui_alias* tables. This table is linked to the *ui_person* table through the u_id column, and to the *ui_transaction* table through the transaction_no column.

■ **ui_alias_history**
The *ui_alias_history* table stores a history of each member's alias information. This table stores the information for the alias portion of each member's audit trail. This table is linked to the *ui_person* table through the u_id column, and to the *ui_transaction* table through the transaction_no column.

■ **ui_address**
The *ui_address* table stores information about each member's addresses. The address_id column is the primary key in this table. This table is linked to the *ui_person* table through the u_id column, and to the *ui_transaction* table through the transaction_no column.

■ **ui_address_history**
The *ui_address_history* table stores a history of each member's addresses. This table stores the information for the address portion of each member's audit trail. This table is linked to the *ui_person* table through the u_id column, and to the *ui_transaction* table through the transaction_no column.

■ **ui_assumed_match**
The *ui_assumed_match* table keeps a record of all member records that were assumed by e*Index to be matches of one another. Information is only written to this table if the control key ASSMTCH is set to **Yes**. The primary key for this table is assumed_match_id. This table is linked to the *ui_person* table through the u_id column, and to the *ui_transaction* table through the transaction_no column.

■ **ui_aux_id**
This tables stores the non-unique IDs assigned to each member, along with the ID type name. The aux_id_id is the primary key in this table. *ui_aux_id* is linked to the *ui_person* table through the u_id column, and to the *ui_transaction* table through the transaction_no and prev_transaction_no columns.

■ **ui_aux_id_history**
This table stores a history of each member's non-unique ID and ID types. This table stores the information for the non-unique ID portion of each

member's audit trail. *ui_aux_id_history* is linked to the *ui_person* table through the u_id column, and to the *ui_transaction* table through the transaction_no column.

■ **ui_comment**
The *ui_comment* table stores user-entered and system-generated comments to a member profile. The primary key in this table is the ui_comment_id.

■ **ui_duplic**
The *ui_duplic* table stores a record of all potential duplicate pairs. This table is linked to the u_id column in *ui_person* through the duplic_id and existing_id columns. This table is also linked to the *ui_transaction* table through the transaction_no column.

■ **ui_local_id**
The *ui_local_id* table stores each member's local ID and system pairs. The ul_id column is the primary key. The *ui_local_id* table is used frequently in data lookups. This table is linked to the *ui_person* table through the u_id column, and to the *ui_transaction* table through the transaction_no column.

■ **ui_local_id_history**
The *ui_local_id_history* table stores a history of each member's local ID and system pairs. The *ui_local_id_history* table stores the information for the local ID portion of each member's audit trail. This table is linked to the *ui_person* table through the u_id column, and to the *ui_transaction* table through the transaction_no column.

■ **ui_mrg_trans**
The *ui_mrg_trans* table stores information about each merge and unmerge transaction that occurs in the database, including UIDs and transaction numbers. The primary key is the ui_mrg_trans_id column. This table is linked to the *ui_person* table through the u_id column, and to the *ui_transaction* table through the transaction_no column.

■ **ui_phone**
The *ui_phone* table stores information about each member's telephone numbers. The phone_id column is the primary key in this table. This table is linked to the *ui_person* table through the u_id column, and to the *ui_transaction* table through the transaction_no column.

■ **ui_phone_history**
The *ui_phone_history* table stores a history of each member's telephone numbers. This table is used to form the telephone number portion of a member's audit trail. This table is linked to the *ui_person* table through the u_id column, and to the *ui_transaction* table through the transaction_no column.

■ **ui_person_x_name**
The *ui_person_x_name* table stores the name information from each record in *ui_person* in upper case only, and enables case-insensitive name

searching from the e*Index GUI. It is queried when an alphanumeric demographic search or a general search is performed in the e*Index GUI. If matching records are found, the corresponding member records from the *ui_person* table are retrieved. The primary key in *ui_person_x_name* is u_id.

- **ui_alias_x_name**
  This table stores each alias name in *ui_alias* in upper case only, and enables case-insensitive *extensive* searching (that is, searching on alias names as well as common names). It is queried when an alphanumeric demographic search or a general search is performed in the e*Index GUI. If matching records are found, the corresponding member records from the *ui_person* table are retrieved. The primary key in *ui_alias_x_name* is ui_alias_id.

Configuration Tables

- **ui_control**
  The *ui_control* table contains the value of each control key in the e*Index Administrator.   This information determines how data is processed, and is read each time the e*Index e*Ways start up. The primary key in this table is ctrl_key. For more information about the instructions contained in this table, see "Defining Control Key Values" in your *e*Index Administrator User's Guide*.

Country Configuration Tables

- **ui_misc_opt_control**
  The *ui_misc_opt_control* table stores information about the areas of e*Index that are affected by the data stored in *ui_misc_option*. It provides descriptions for the codes entered in the control_type column of *ui_misc_option*. The control_type column is also the primary key for this table, and links *ui_misc_opt_control* with *ui_misc_option*.

- **ui_misc_opt_country**
  The *ui_misc_opt_country* table stores information about the country codes used in *ui_misc_option*, describing which country corresponds with which code. The country_code column is also the primary key for this table, and links *ui_misc_opt_country* with *ui_misc_option*.

- **ui_misc_option**
  This table provides the ability to configure country-specific attributes of e*Index by reformatting certain fields depending on which country you select. This table also contains label information for the tabs that appear on the e*Index GUI windows. Currently, available country formats include the United Kingdom, Australia, Ireland, and the United States. This table is controlled by the COUNTRY control key in e*Index Administrator. For more information, see "Configuring Country-Specific Options" in Chapter 5 of your *e*Index Administrator User's Guide*. The primary key for this table is misc_option_id. The table is linked to

ui_misc_opt_control by the control_type column, and to
ui_misc_opt_country by the country_code column.

Display Configuration Tables

■ **ui_table**
This table stores a list of all tables that contain columns whose field labels on the e*Index GUI can be modified. These tables appear in the table list on the Display Configuration window of e*Index Administrator. The primary key is ui_table_id, and the table is linked to *ui_table_column* through the primary key.

■ **ui_table_column**
The *ui_table_column* table stores information about the different database columns whose field labels on the e*Index GUI can be modified. These fields are listed on the Display Configuration window of e*Index Administrator. The primary key is table_column_id, and the table is linked to *ui_table* through the ui_table_id column.

Configurable Candidate Selection (CCS) Tables

The CSS tables stores information about the configurable queries. Each portion of the SQL statement for the queries is stored in a separate database table. For more information about configurable queries, see "Configuring Queries" in Chapter 5 of the *e*Index Administrator User's Guide*.

■ **ui_cand_from_table**
This table stores the names of the tables used in the SQL statements generated by the configurable queries. For example, if you created this SQL statement

select first_name, last_name from ui_person where u_id = '100'

this table stores the 'ui_person' portion of the statement. The primary key for ui_*cand_from_table* is the from_table_id column. This table is linked to *ui_cand_where_clause* by the where_clause_id column, and to *ui_cand_sql_table* by the sql_table_id column.

■ **ui_cand_select_column**
This table stores the columns in the 'select' portion of the SQL statements generated by the configurable queries. For example, in the above SQL statement, this table stores the 'first_name, last_name' portion of the statement. The select_column_id column is the primary key. This table is linked to *ui_cand_sql* by the cand_sql_id column, and to *ui_cand_sql_column* by the sql_column_id column.

■ **ui_cand_where_column**
This table stores the columns called in the 'where' portion of the SQL statements generated by the configurable queries. For example, in the above SQL statement, this table would store the 'u_id' portion of the

statement. The primary key for this table is the where_column_id column. It is linked to *ui_cand_where_clause* by the where_column_id column, and to *ui_cand_sql_column* through the sql_column_id column.

■ **ui_cand_sql**
This table stores the descriptions and identification codes for the two types of configurable queries. The primary key is the cand_sql_id column. This table is linked to *ui_object_type* by the object_type column, to *ui_cand_select_column* by the cand_sql_id column, and to *ui_cand_where_clause* by the cand_sql_id column.

■ **ui_cand_sql_column**
This table stores information about the available and selected columns for the configurable queries. The primary key is the sql_column_id column. This table is linked to *ui_cand_select_column* and *ui_cand_where_column* by the sql_column_id column, and to *ui_cand_sql_table* by the sql_table_id column.

■ **ui_cand_sql_table**
This table stores information about the database tables that are available for the configurable queries, and the tables that are selected. The primary key for this table is the sql_table_id column. It is linked to *ui_cand_from table* and *ui_cand_sql_column* by the sql_table_id column.

■ **ui_cand_where_clause**
This table stores information about the 'where' clauses for the configurable queries, along with the criteria list description of each clause. The primary key for this table is the where_clause_id column. It is linked to *ui_cand_from_table* and *ui_cand_where_column* by the where_clause_id column, and to *ui_cand_sql* by the cand_sql_id column.

■ **ui_object_type**
This table is reserved for future functionality and is not used in this release.

Code Data Tables

■ **stc_common_detail**
This table stores detailed information about the data elements you add using the Common Table maintenance functions of e*Index Administrator. The common header ID associated with each data element in this table specifies the table maintenance function with which the data element is associated. The primary key in this table is common_detail_id, and the table is linked to *stc_common_header* by the common_header_id column.

■ **stc_common_header**
This table is the header file for the items in the *stc_common_detail* table. It stores a list of the types of data elements you can add using the Common Table maintenance functions of e*Index Administrator, such as races, languages, driver's license issuers, and so on. The primary key of this

table is common_header_id, which is also the column that links *stc_common_header* with *stc_common_detail*.

- **ui_aux_id_def**
  This table stores the various non-unique ID types that you define in e*Index Administrator. It's primary key is aux_id_def, and it is linked to the *ui_aux_id* and *ui_aux_id_history* tables through the primary key.

- **ui_canned_msg**
  This table stores the predefined messages that you defined in the Table Maintenance function of e*Index Administrator. Elements defined in this table can be accessed from a drop-down list on the Predefined Comments window. The primary key for this table is *code*.

- **ui_dept**
  This table is not in use at this time.

- **ui_facility**
  This table stores information about the systems (previously known as facilities) that you defined in the Table Maintenance function of e*Index Administrator. Each system assigns a local ID to member records, and the local ID and system pairs are stored in *ui_local_id*. The primary key for this table is facility_code, and *ui_facility* is linked to *ui_local_id* and *ui_local_id_history* by the facility_code column.

- **ui_message**
  This table stores the application messages that appear on the e*Index GUIs. For example, it contains error messages, confirmation messages, and warnings. These data elements can be modified using the Message Maintenance function of e*Index Administrator. The primary key for this table is *code*.

- **ui_zip**
  This table stores city, state, and zip code information. The values stored in this table validate the addresses entered in e*Index, and allow you to automatically populate the city and state fields when you enter the zip code of an address. The primary key in this table is zip_code.

Vality Integrity Matching Algorithm Tables

- **ui_ctrl_column**
  This table is reserved for possible future functionality, and is not currently being used.

- **ui_ctrl_field**
  This table is reserved for possible future functionality, and is not currently being used.

- **ui_ctrl_file**
  This table stores the information that is currently located in the Vality rule set file. This information is used for member matching using the Vality Integrity matching algorithm, and defines the fields and weights to

be used for matching. The primary key for this table is ui_ctrl_file_id. *ui_ctrl_file* is linked to the *ui_ctrl_rule* table by the ui_ctrl_rule_id column, and to the *ui_ctrl_file_hist* table by the ui_ctrl_file_id column.

- **ui_ctrl_file_hist**
  The *ui_ctrl_file_hist* table stores a history of Vality information and tracks changes made to the rule set files for the matching algorithm. This table is linked to the *ui_ctrl_file* table by the ui_ctrl_file_id column.

- **ui_ctrl_rule**
  The *ui_ctrl_rule* table stores information about the rule sets that have been defined for e*Index. This table describes the name of the rule set files and indicates which rule sets are in use. This file is a header file to *ui_ctrl_file*, and these tables are linked by the ui_ctrl_rule_id column (the primary key).

- **ui_ctrl_table**
  This table is reserved for possible future functionality, and is not currently being used.

- **ui_nickname**
  The *ui_nickname* table stores information about common nicknames of your member's first names. This table is used when records are matched using the Vality matching algorithm. The primary key for this table is ui_nickname_id.

Security Tables

- **stc_acc_def**
  This table stores information about the access permissions that appear in the access list on the Access List window of e*Index Security. The primary key for this table is acc_def_id. *stc_acc_def* is linked to *stc_module* by the module_id column, and to *stc_group_acc* and *stc_user_acc* by the acc_def_id column. The module_id column indicates the primary function in *stc_module* with which each access permission is associated.

- **stc_group**
  This table stores information about the user groups defined in e*Index Security. The primary key for this table is group_id. It is linked to the *stc_user_group* and *stc_group_acc* tables by the group_id column. This table does not store information about the user profiles or access permissions assigned to each user group.

- **stc_group_acc**
  The *stc_group_acc* table stores information about the access permissions granted to each user group. The primary key for this table is group_acc_id. It is linked to *stc_acc_def* by the acc_def_id column and to *stc_group* by the group_id column. The group_id column indicates the user groups to which access permissions are granted, and the acc_def_id column indicates the access permissions granted to each user group.

- **stc_module**
  The *stc_module* table stores information about the primary functions that appear in the access list on the Access List window of e*Index Security. The primary key for this table is module_id. *stc_module* is linked to *stc_appl* by the appl_id column, and to *stc_acc_def* by the module_id column. The appl_id column indicates the application in *stc_appl* with which each function in *stc_module* is associated (currently only e*Index).

- **stc_user**
  This table stores information about the user profiles defined in e*Index Security. The primary key for this table is usersl_id. It is linked to the *ui_login*, *ui_login_current*, *stc_user_region*, *ui_notify_user*, *ui_user_passwd_hist*, *stc_user_acc*, and *stc_user_group* tables by the usersl_id column. This table does not store information about the access permissions assigned to each user profile, or the user groups and regions to which each profile is assigned.

- **stc_user_acc**
  The *stc_user_acc* table stores information about the access permissions granted to each user profile. The primary key for this table is user_acc_id. It is linked to *stc_acc_def* by the acc_def_id column and to *stc_user* by the usersl_id column. The usersl_id column indicates the user profiles to which access permissions are granted, and the acc_def_id column indicates the access permissions granted to each user profile.

- **stc_user_region**
  This table stores information about the regions to which each user profile is assigned. This table is only used if region-specific security is installed (see your *e*Index Security User's Guide* for more information). The primary key for this table is user_region_id, and the table is linked to *stc_user* by the usersl_id column.

- **stc_user_group**
  This table stores information about the user groups to which each user profile is assigned. The primary key for this table is usergp_id. *stc_user_group* is linked to *stc_user* by the usersl_id column, and to *stc_group* by the group_id column.

- **ui_control_sec**
  The *ui_control_sec* table stores the values assigned to each e*Index Security control key. These values are specified in the Control Key Maintenance function of e*Index Security. The primary key for this table is ctrl_key.

- **ui_login**
  The *ui_login* table stores a log of all of the users who have logged on to the e*Index system. This log includes the users' IDs, the time they logged on, and the time they logged off. The primary key for this table is ui_login_id. *ui_login* is linked to *stc_user* by the column usersl_id.

■ **ui_login_current**
The *ui_login-current* table stores the user IDs, log on time, and log off time of the users who are currently logged on to the e*Index system. It is linked to *stc_user* by the column usersl_id. The primary key for this table is ui_login_current_id.

■ **ui_no_passwd**
The *ui_no_passwd* table stores the passwords that cannot be used by the users of any of the e*Index applications. You can add and delete passwords in this table as required. This table is linked to *stc_user* by the usersl_id column. The primary key for this table is no_passwd.

■ **ui_notify_user**
This table stores information about the event notifications to which each user profile is assigned. You can assign event notifications to user profiles on the Event Notify window in e*Index Security. When you remove notification for an event from a user profile, the corresponding row in this table is removed. The primary key for this table is ui_notify_user_id, and *ui_notify_user* is linked to *stc_user* by the usersl_id column.

■ **ui_user_passwd_hist**
This table stores a list of the most recent login passwords for each e*Index user. The number of passwords retained in this table is determined by the value you specify for the PASSHIST control key. A user cannot reuse one of their passwords still in this table.

Outbound Event Tables

■ **dequeue_lock**
The *dequeue_lock* table is reserved for possible future functionality, and is not currently being used.

■ **ui_msg_detail**
The *ui_msg_detail* table stores the Events that are in the outbound queue, and are waiting to be picked up by the e*Index polling e*Way. This table is associated with the *ui_msg_header* table through the ui_msg_header_id column. The primary key for this table is the ui_msg_detail_id column.

■ **ui_msg_header**
This table is the header file for the items in the *ui_msg_detail* table. It stores the queue ID number for each Event and any errors in processing those Events. The polling e*Way continually checks this table to see if any outgoing Events are ready to be processed. This table is linked to *ui_msg_detail* by its primary key, ui_msg_header_id.

Miscellaneous Tables

- **stc_appl**
  This table is linked to the *stc_common_header* and *stc_module* tables and indicates the application with which each item in *stc_common_header* and *stc_module* is associated. The primary key is appl_id, and is linked to *stc_common_header* and *stc_modules* by this column. Currently the only item in this table is e*Index.

- **ui_audit**
  The *ui_audit* table provides a log of the instances that users look up member data in the *ui_person* table. *ui_audit* includes the user ID of the person who updated the database, the date and time the database was modified, and the type of transaction that occurred. The primary key is the ui_audit_id column. This table can grow rapidly, depending on how often the *ui_person* table is accessed from the GUI. Be sure to archive this table regularly.

- **ui_config**
  The *ui_config* table stores information about the current version and schema of e*Index. When you login into one of the GUIs, this table tells the system the database schema that is required in order to access the specified database. This table has two primary keys: *interface* and *code*.

- **ui_monthly_stat**
  The *ui_monthly_stat* table stores the information that appears in the monthly reports.

- **ui_seq_no**
  This table defines the sequential codes that are used in other tables in the e*Index database. The primary key for this table is the table_name column.

e*Index Oracle Database Model

The diagrams on the following pages illustrate the table structure for e*Index for Oracle. The *ui_person* and *ui_transaction* tables are displayed on two different pages to better illustrate the connections to these two tables. For illustrations of Sybase and Microsoft SQL Server databases, see chapters 4 and 5 of the *e*Index Global Identifier Installation Guide*.

where_clause_id = where_clause_id

ui_cand_from_table

from_table_id	<pk>	NUMBER(10)	not null
where_clause_id	<fk>	NUMBER(10)	not null
sql_table_id	<fk>	NUMBER(10)	not null
table_alias		VARCHAR2(10)	null
sequence		NUMBER(10)	not null

ui_cand_from_table_1

ui_cand_where_clause

where_clause_id	<pk>	NUMBER(10)	not null
cand_sql_id	<fk>	NUMBER(10)	not null
where_clause_name		VARCHAR2(40)	not null
where_clause		LONG	null

where_clause_id = where_clause_id

sql_table_id = sql_table_id

ui_object_type

object_type	<pk>	VARCHAR2(5)	not null
description		VARCHAR2(40)	null

ui_cand_sql_table

sql_table_id	<pk>	NUMBER(10)	not null
table_name		VARCHAR2(30)	not null
selected		CHAR(1)	not null
readonly		CHAR(1)	not null

cand_sql_id = cand_sql_id

object_type = object_type

ui_cand_where_column

where_column_id	<pk>	NUMBER(10)	not null
where_clause_id	<fk>	NUMBER(10)	not null
sql_column_id	<fk>	NUMBER(10)	not null
sequence		NUMBER(10)	not null

ui_cand_sql

cand_sql_id	<pk>	NUMBER(10)	not null
object_type	<fk>	VARCHAR2(5)	not null
use_type		CHAR(1)	not null
sql_name		VARCHAR2(40)	not null

sql_column_id = sql_column_id

cand_sql_id = cand_sql_id

ui_cand_sql_column

sql_column_id	<pk>	NUMBER(10)	not null
sql_table_id	<fk>	NUMBER(10)	not null
column_name		VARCHAR2(30)	not null
column_type		VARCHAR2(5)	not null
column_length		NUMBER(10)	not null
attr_name		VARCHAR2(40)	null
phonetic_source		VARCHAR2(30)	null
phonetic_type		CHAR(1)	null
match_column		CHAR(1)	null
selected		CHAR(1)	not null
readonly		CHAR(1)	not null

ui_cand_select_column

select_column_id	<pk>	NUMBER(10)	not null
cand_sql_id	<fk>	NUMBER(10)	not null
sql_column_id	<fk>	NUMBER(10)	not null
sequence		NUMBER(10)	not null
match_column		CHAR(1)	not null

_column_id = sql_column_id

dequeue_lock

sess_id	NUMBER	null
sess_activity	DATE	null
last_seq	NUMBER(20)	null
del_seq	NUMBER(20)	null

ui_monthly_stat

be_new_u_id	NUMBER	null
fe_new_u_id	NUMBER	null
be_merge	NUMBER	null
fe_merge	NUMBER	null
be_unmerge	NUMBER	null
fe_unmerge	NUMBER	null
be_deact	NUMBER	null
fe_deact	NUMBER	null
pot_dup	NUMBER	null
pot_dup_res	NUMBER	null
month	VARCHAR2(20)	null
year	NUMBER	null

ui_audit

ui_audit_id	<pk>	NUMBER(10)	not null
user_id		VARCHAR2(20)	not null
timestamp		DATE	not null
uid1		VARCHAR2(15)	null
uid2		VARCHAR2(15)	null
funct		VARCHAR2(40)	null
detail		VARCHAR2(150)	null
pat_acct_num		VARCHAR2(20)	null
terminal_id		VARCHAR2(25)	null
facility		VARCHAR2(20)	null
location		VARCHAR2(15)	null
order_num		VARCHAR2(20)	null
app		VARCHAR2(10)	null

ui_seq_no

table_name	<pk>	VARCHAR2(30)	not null
seq_no		NUMBER(15)	not null

ui_config

interface	<pk>	VARCHAR2(255)	not null
code	<pk>	VARCHAR2(255)	not null
value		NUMBER	not null
🗋 pk_idx_ui_config			

ui_control

ctrl_key	<pk>	VARCHAR2(10)	not null
description		VARCHAR2(50)	null
ctrl_value		VARCHAR2(30)	null
create_date		DATE	null
🗋 pk_idx_control			

ui_dept

dept_code	<pk>	VARCHAR2(5)	not null
description		VARCHAR2(20)	null
date_time		DATE	null
🗋 pk_idx_dept			

ui_canned_msg

code	<pk>	VARCHAR2(5)	not null
description		VARCHAR2(80)	not null
create_date		DATE	null
🗋 pk_idx_canned			

ui_message

code	<pk>	VARCHAR2(5)	not null
description		LONG	not null
message_box_header		VARCHAR2(50)	not null
icon		VARCHAR2(15)	null
button		VARCHAR2(20)	null
default_button		NUMBER(1)	null
message_type		VARCHAR2(8)	null
application		VARCHAR2(10)	null
date_time		DATE	null
🗋 pk_idx_ui_message			

ui_zip

zip_code	<pk>	VARCHAR2(8)	not null
zip4		VARCHAR2(4)	null
city	<pk>	VARCHAR2(30)	not null
state	<pk>	VARCHAR2(10)	not null
county		VARCHAR2(3)	null
residence_code		VARCHAR2(4)	null
create_date		DATE	null
🗋 pk_idx_zip			

ui_comment

ui_comment_id	<pk>	NUMBER(10)	not null
u_id		VARCHAR2(15)	not null
type		VARCHAR2(8)	not null
timestamp		DATE	not null
comment_text		LONG	null
ui_org		VARCHAR2(15)	null
🗋 ui_id_comment			

ui_msg_header

ui_msg_header_id	<pk>	NUMBER(20)	not null
queue_id		CHAR(1)	not null
errors		NUMBER(10)	not null
create_date		DATE	not null
create_userid		VARCHAR2(20)	not null
🗋 ui_msg_header_1			

ui_msg_header_id = ui_msg_header_id

ui_msg_detail

ui_msg_detail_id	<pk>	NUMBER(20)	not null
ui_msg_header_id	<fk>	NUMBER(20)	null
msg		VARCHAR2(512)	not null
🗋 fk_ui_msg_detail			

ui_person_history

ui_person_history_id	<pk>	NUMBER(10)	not null
u_id		VARCHAR2(15)	not null
person_cat_code		VARCHAR2(8)	null
last_name		VARCHAR2(40)	null
first_name		VARCHAR2(40)	null
middle_name		VARCHAR2(30)	null
suffix		VARCHAR2(10)	null
title		VARCHAR2(8)	null
dob		DATE	null
death		VARCHAR2(1)	null
sex		VARCHAR2(8)	null
mstatus		VARCHAR2(8)	null
ssn		VARCHAR2(16)	null
race		VARCHAR2(8)	null
ethnic		VARCHAR2(8)	null
religion		VARCHAR2(8)	null
language		VARCHAR2(8)	null
spouse_name		VARCHAR2(100)	null
mother_name		VARCHAR2(100)	null
mother_mn		VARCHAR2(40)	null
father_name		VARCHAR2(100)	null
maiden		VARCHAR2(40)	null
pob_city		VARCHAR2(30)	null
pob_state		VARCHAR2(10)	null
pob_country		VARCHAR2(20)	null
vip_flag		VARCHAR2(8)	null
vet_status		VARCHAR2(8)	null
status		VARCHAR2(8)	null
fname_phonetic_code		VARCHAR2(8)	null
lname_phonetic_code		VARCHAR2(8)	null
mname_phonetic_code		VARCHAR2(8)	null
mother_mn_phonetic_code		VARCHAR2(8)	null
maiden_phonetic_code		VARCHAR2(8)	null
spouse_phonetic_code		VARCHAR2(8)	null
mother_phonetic_code		VARCHAR2(8)	null
father_phonetic_code		VARCHAR2(8)	null
drivers_license		VARCHAR2(20)	null
drivers_license_st		VARCHAR2(10)	null
dod		DATE	null
death_certificate		VARCHAR2(10)	null
nationality		VARCHAR2(8)	null
citizenship		VARCHAR2(8)	null
pension_no		VARCHAR2(15)	null
pension_exp_date		DATE	null
repatriation_no		VARCHAR2(16)	null
district_of_residence		VARCHAR2(8)	null
lga_code		VARCHAR2(4)	null
military_branch		VARCHAR2(4)	null
military_rank		VARCHAR2(4)	null
military_status		VARCHAR2(4)	null
dummy_date		DATE	null
class1		VARCHAR2(20)	null
class2		VARCHAR2(20)	null
class3		VARCHAR2(20)	null
class4		VARCHAR2(20)	null
class5		VARCHAR2(20)	null
string1		VARCHAR2(40)	null
string2		VARCHAR2(40)	null
string3		VARCHAR2(40)	null
string4		VARCHAR2(40)	null
string5		VARCHAR2(40)	null
string6		VARCHAR2(40)	null
string7		VARCHAR2(100)	null
string8		VARCHAR2(100)	null
string9		VARCHAR2(100)	null
string10		VARCHAR2(255)	null
date1		DATE	null
date2		DATE	null
date3		DATE	null
date4		DATE	null
date5		DATE	null
transaction_no	<fk>	NUMBER(10)	not null

transaction_no = transaction_no

TO UI_TRANSACTION

FROM UI_PERSON_HISTORY

transaction_no = transaction_no

transaction_no = trans_no
transaction_no = unmrg_trans_no_from
transaction_no = trans_no_from
transaction_no = unmrg_trans_no
transaction_no = transaction_no
transaction_no = transaction_no
transaction_no = transaction_no
transaction_no = transaction_no

ui_transaction

transaction_no	<pk>	NUMBER(10)	not null
u_id		VARCHAR2(15)	not null
system		VARCHAR2(8)	null
dept		VARCHAR2(8)	null
terminal_id		VARCHAR2(25)	null
session_id		VARCHAR2(15)	null
update_facility_id		VARCHAR2(20)	not null
update_function		VARCHAR2(8)	not null
update_date		DATE	not null
update_userid		VARCHAR2(20)	not null

transaction_no = prev_transaction_no
transaction_no = transaction_no

u_id = u_id
transaction_no = transaction_no

ui_assumed_match

assumed_match_id	<pk>	NUMBER(10)	not null
u_id	<fk>	VARCHAR2(15)	not null
weight		FLOAT	not null
prev_transaction_no	<fk>	NUMBER(10)	not null
transaction_no	<fk>	NUMBER(10)	not null

ui_person

u_id	<pk>	VARCHAR2(15)	not null
person_cat_code		VARCHAR2(8)	null
last_name		VARCHAR2(40)	not null
first_name		VARCHAR2(40)	not null
middle_name		VARCHAR2(30)	null
suffix		VARCHAR2(10)	null
title		VARCHAR2(8)	null
dob		DATE	null
death		VARCHAR2(1)	null
sex		VARCHAR2(8)	null
mstatus		VARCHAR2(8)	null
ssn		VARCHAR2(16)	null
race		VARCHAR2(8)	null
ethnic		VARCHAR2(8)	null
religion		VARCHAR2(8)	null
language		VARCHAR2(8)	null
spouse_name		VARCHAR2(100)	null
mother_name		VARCHAR2(100)	null
mother_mn		VARCHAR2(40)	null
father_name		VARCHAR2(100)	null
maiden		VARCHAR2(40)	null
pob_city		VARCHAR2(30)	null
pob_state		VARCHAR2(10)	null
pob_country		VARCHAR2(20)	null
vip_flag		VARCHAR2(8)	null
vet_status		VARCHAR2(8)	null
status		VARCHAR2(8)	null
fname_phonetic_code		VARCHAR2(8)	null
lname_phonetic_code		VARCHAR2(8)	null
mname_phonetic_code		VARCHAR2(8)	null
mother_mn_phonetic_code		VARCHAR2(8)	null
maiden_phonetic_code		VARCHAR2(8)	null
spouse_phonetic_code		VARCHAR2(8)	null
mother_phonetic_code		VARCHAR2(8)	null
father_phonetic_code		VARCHAR2(8)	null
drivers_license		VARCHAR2(20)	null
drivers_license_st		VARCHAR2(10)	null
dod		DATE	null
death_certificate		VARCHAR2(10)	null
nationality		VARCHAR2(8)	null
citizenship		VARCHAR2(8)	null
pension_no		VARCHAR2(15)	null
pension_exp_date		DATE	null
repatriation_no		VARCHAR2(16)	null
district_of_residence		VARCHAR2(8)	null
lga_code		VARCHAR2(4)	null
military_branch		VARCHAR2(4)	null
military_rank		VARCHAR2(4)	null
military_status		VARCHAR2(4)	null
dummy_date		DATE	null
class1		VARCHAR2(20)	null
class2		VARCHAR2(20)	null
class3		VARCHAR2(20)	null
class4		VARCHAR2(20)	null
class5		VARCHAR2(20)	null
string1		VARCHAR2(40)	null
string2		VARCHAR2(40)	null
string3		VARCHAR2(40)	null
string4		VARCHAR2(40)	null
string5		VARCHAR2(40)	null
string6		VARCHAR2(40)	null
string7		VARCHAR2(100)	null
string8		VARCHAR2(100)	null
string9		VARCHAR2(100)	null
string10		VARCHAR2(255)	null
date1		DATE	null
date2		DATE	null
date3		DATE	null
date4		DATE	null
date5		DATE	null
transaction_no	<fk>	NUMBER(10)	null

ui_aux_id

aux_id_id	<pk>	NUMBER(10)	not null
u_id	<fk>	VARCHAR2(15)	not null
aux_id_def	<fk>	NUMBER(10)	not null
id		VARCHAR2(40)	not null
transaction_no	<fk>	NUMBER(10)	null

u_id = u_id
aux_id_def = aux_id_def

ui_aux_id_def

aux_id_def	<pk>	NUMBER(10)	not null
id_type		VARCHAR2(8)	not null
id_name		VARCHAR2(30)	not null
mask		CHAR(1)	not null
id_length		NUMBER(10)	not null
format		VARCHAR2(40)	not null
variable_length		char(1)	not null

aux_id_def = aux_id_def

ui_aux_id_history

aux_id_id		NUMBER(10)	not null
u_id	<fk>	VARCHAR2(15)	not null
aux_id_def	<fk>	NUMBER(10)	null
id		VARCHAR2(40)	null
transaction_no	<fk>	NUMBER(10)	not null

u_id = u_id

ui_address

address_id	<pk>	NUMBER(10)	not null
u_id	<fk>	VARCHAR2(15)	not null
address_type		VARCHAR2(8)	not null
address1		VARCHAR2(40)	null
address2		VARCHAR2(40)	null
address3		VARCHAR2(40)	null
address4		VARCHAR2(40)	null
city		VARCHAR2(30)	null
state_code		VARCHAR2(10)	null
postal_code		VARCHAR2(8)	null
postal_code_ext		VARCHAR2(4)	null
county		VARCHAR2(20)	null
country_code		VARCHAR2(20)	null
house_number		VARCHAR2(10)	null
street_dir		VARCHAR2(5)	null
street_name		VARCHAR2(40)	null
street_name_phonetic		VARCHAR2(8)	null
street_type		VARCHAR2(5)	null
transaction_no	<fk>	NUMBER(10)	null

u_id = u_id

ui_address_history

address_id		NUMBER(10)	not null
u_id	<fk>	VARCHAR2(15)	not null
address_type		VARCHAR2(8)	null
address1		VARCHAR2(40)	null
address2		VARCHAR2(40)	null
address3		VARCHAR2(40)	null
address4		VARCHAR2(40)	null
city		VARCHAR2(30)	null
state_code		VARCHAR2(10)	null
postal_code		VARCHAR2(8)	null
postal_code_ext		VARCHAR2(4)	null
county		VARCHAR2(20)	null
country_code		VARCHAR2(20)	null
house_number		VARCHAR2(10)	null
street_dir		VARCHAR2(5)	null
street_name		VARCHAR2(40)	null
street_name_phonetic		VARCHAR2(8)	null
street_type		VARCHAR2(5)	null
transaction_no	<fk>	NUMBER(10)	not null

u_id = u_id

ui_mrg_trans

ui_mrg_trans_id	<pk>	NUMBER(10)	not null
u_id	<fk>	VARCHAR2(15)	not null
u_id_from	<fk>	VARCHAR2(15)	not null
trans_no	<fk>	NUMBER(10)	not null
trans_no_from	<fk>	NUMBER(10)	null
unmrg_trans_no	<fk>	NUMBER(10)	null
unmrg_trans_no_from	<fk>	NUMBER(10)	null

u_id = u_id
u_id = u_id_from

FROM UI_PERSON_HISTORY

transaction_no = transaction_no

ui_transaction

transaction_no	<pk>	NUMBER(10)	not null
u_id		VARCHAR2(15)	not null
system		VARCHAR2(8)	null
dept		VARCHAR2(8)	null
terminal_id		VARCHAR2(25)	null
session_id		VARCHAR2(15)	null
update_facility_id		VARCHAR2(20)	not null
update_function		VARCHAR2(8)	not null
update_date		DATE	not null
update_userid		VARCHAR2(20)	not null

transaction_no = transaction_no
transaction_no = transaction_no
transaction_no = transaction_no
transaction_no = transaction_no
transaction_no = transaction_no
transaction_no = transaction_no
transaction_no = transaction_no
transaction_no = transaction_no

u_id = u_id

ui_phone

phone_id	<pk>	NUMBER(10)	not null
u_id	<fk>	VARCHAR2(15)	not null
phone_type		VARCHAR2(8)	not null
phone		VARCHAR2(20)	not null
phone_ext		VARCHAR2(6)	null
transaction_no	<fk>	NUMBER(10)	null

transaction_no = transaction_no

ui_person

u_id	<pk>	VARCHAR2(15)	not null
person_cat_code		VARCHAR2(8)	null
last_name		VARCHAR2(40)	not null
first_name		VARCHAR2(40)	not null
middle_name		VARCHAR2(30)	null
suffix		VARCHAR2(10)	null
title		VARCHAR2(8)	null
dob		DATE	null
death		VARCHAR2(1)	null
sex		VARCHAR2(8)	null
mstatus		VARCHAR2(8)	null
ssn		VARCHAR2(16)	null
race		VARCHAR2(8)	null
ethnic		VARCHAR2(8)	null
religion		VARCHAR2(8)	null
language		VARCHAR2(8)	null
spouse_name		VARCHAR2(100)	null
mother_name		VARCHAR2(100)	null
mother_mn		VARCHAR2(40)	null
father_name		VARCHAR2(100)	null
maiden		VARCHAR2(40)	null
pob_city		VARCHAR2(30)	null
pob_state		VARCHAR2(10)	null
pob_country		VARCHAR2(20)	null
vip_flag		VARCHAR2(8)	null
vet_status		VARCHAR2(8)	null
status		VARCHAR2(8)	null
fname_phonetic_code		VARCHAR2(8)	null
lname_phonetic_code		VARCHAR2(8)	null
mname_phonetic_code		VARCHAR2(8)	null
mother_mn_phonetic_code		VARCHAR2(8)	null
maiden_phonetic_code		VARCHAR2(8)	null
spouse_phonetic_code		VARCHAR2(8)	null
mother_phonetic_code		VARCHAR2(8)	null
father_phonetic_code		VARCHAR2(8)	null
drivers_license		VARCHAR2(20)	null
drivers_license_st		VARCHAR2(10)	null
dod		DATE	null
death_certificate		VARCHAR2(10)	null
nationality		VARCHAR2(8)	null
citizenship		VARCHAR2(8)	null
pension_no		VARCHAR2(15)	null
pension_exp_date		DATE	null
repatriation_no		VARCHAR2(16)	null
district_of_residence		VARCHAR2(8)	null
lga_code		VARCHAR2(4)	null
military_branch		VARCHAR2(4)	null
military_rank		VARCHAR2(4)	null
military_status		VARCHAR2(4)	null
dummy_date		DATE	null
class1		VARCHAR2(20)	null
class2		VARCHAR2(20)	null
class3		VARCHAR2(20)	null
class4		VARCHAR2(20)	null
class5		VARCHAR2(20)	null
string1		VARCHAR2(40)	null
string2		VARCHAR2(40)	null
string3		VARCHAR2(40)	null
string4		VARCHAR2(40)	null
string5		VARCHAR2(40)	null
string6		VARCHAR2(40)	null
string7		VARCHAR2(100)	null
string8		VARCHAR2(100)	null
string9		VARCHAR2(100)	null
string10		VARCHAR2(255)	null
date1		DATE	null
date2		DATE	null
date3		DATE	null
date4		DATE	null
date5		DATE	null
transaction_no	<fk>	NUMBER(10)	null

u_id = u_id

ui_phone_history

phone_id		NUMBER(10)	not null
u_id	<fk>	VARCHAR2(15)	not null
phone_type		VARCHAR2(8)	null
phone		VARCHAR2(20)	null
phone_ext		VARCHAR2(6)	null
transaction_no	<fk>	NUMBER(10)	not null

u_id = u_id

u_id = duplic_id

u_id = existing_id

ui_duplic

ui_duplic_id	<pk>	NUMBER(10)	not null
duplic_id	<fk>	VARCHAR2(15)	not null
existing_id	<fk>	VARCHAR2(15)	not null
weight		FLOAT	not null
description		VARCHAR2(30)	null
dummy_date		DATE	null
resolved		VARCHAR2(8)	null
transaction_no	<fk>	NUMBER(10)	null

u_id = u_id

ui_local_id

ul_id	<pk>	NUMBER(10)	not null
u_id	<fk>	VARCHAR2(15)	not null
facility	<fk>	VARCHAR2(20)	not null
local_id		VARCHAR2(25)	not null
status		VARCHAR2(8)	null
transaction_no	<fk>	NUMBER(10)	null

facility_code = facility

ui_facility

facility_code	<pk>	VARCHAR2(20)	not null
description		VARCHAR2(30)	null
status		VARCHAR2(8)	null
addr1		VARCHAR2(30)	null
addr2		VARCHAR2(30)	null
city		VARCHAR2(30)	null
state		VARCHAR2(10)	null
zip		VARCHAR2(8)	null
zipext		VARCHAR2(4)	null
county		VARCHAR2(20)	null
country		VARCHAR2(20)	null
region_code		VARCHAR2(8)	null
date_time		DATE	null
id_length		NUMBER	null
mask		CHAR(1)	null
local_id_format		VARCHAR2(35)	null
facility_mask		CHAR(1)	null
next_local_id		VARCHAR2(25)	null
allow_less_id_length		VARCHAR2(1)	null

facility_code = facility

ui_local_id_history

ul_id		NUMBER(10)	not null
u_id	<fk>	VARCHAR2(15)	not null
facility	<fk>	VARCHAR2(20)	null
local_id		VARCHAR2(25)	null
status		VARCHAR2(8)	null
transaction_no	<fk>	NUMBER(10)	not null

u_id = u_id

ui_alias

ui_alias_id	<pk>	NUMBER(10)	not null
u_id	<fk>	VARCHAR2(15)	not null
last_name		VARCHAR2(40)	not null
first_name		VARCHAR2(40)	not null
middle_name		VARCHAR2(30)	null
lname_phonetic_code		VARCHAR2(8)	null
fname_phonetic_code		VARCHAR2(8)	null
mname_phonetic_code		VARCHAR2(8)	null
transaction_no	<fk>	NUMBER(10)	null

u_id = u_id

ui_alias_history

ui_alias_id		NUMBER(10)	not null
u_id	<fk>	VARCHAR2(15)	not null
last_name		VARCHAR2(40)	null
first_name		VARCHAR2(40)	null
middle_name		VARCHAR2(30)	null
lname_phonetic_code		VARCHAR2(8)	null
fname_phonetic_code		VARCHAR2(8)	null
mname_phonetic_code		VARCHAR2(8)	null
transaction_no	<fk>	NUMBER(10)	not null

ui_ctrl_rule

ui_ctrl_rule_id	<pk>	NUMBER(10)	not null
rule_name		VARCHAR2(16)	not null
root_file		VARCHAR2(16)	not null
read_only		CHAR(1)	not null
in_use		CHAR(1)	not null
create_date		DATE	not null
create_userid		VARCHAR2(20)	not null
update_date		DATE	null
update_userid		VARCHAR2(20)	null

ui_ctrl_rule_id = ui_ctrl_rule_id

ui_ctrl_file

ui_ctrl_file_id	<pk>	NUMBER(10)	not null
ui_ctrl_rule_id	<fk>	NUMBER(10)	not null
file_type		VARCHAR2(3)	not null
file_name		VARCHAR2(18)	not null
file_ext		VARCHAR2(3)	not null
file_content		LONG	null
content_date		DATE	not null
last_synch_date		DATE	null
create_date		DATE	not null
create_userid		VARCHAR2(20)	not null
update_date		DATE	null
update_userid		VARCHAR2(20)	null

ctrl_file_id = ui_ctrl

ui_ctrl_table

ui_ctrl_table_id	<pk>	NUMBER(10)	not null
table_name		VARCHAR2(30)	not null
description		VARCHAR2(48)	null
read_only		CHAR(1)	not null
create_date		DATE	not null
create_userid		VARCHAR2(20)	not null
update_date		DATE	null
update_userid		VARCHAR2(20)	null

ui_ctrl_table_id = ui_ctrl_table_id

ui_ctrl_column

ui_ctrl_column_id	<pk>	NUMBER(10)	not null
ui_ctrl_table_id	<fk>	NUMBER(10)	not null
column_name		VARCHAR2(30)	not null
description		VARCHAR2(48)	null
read_only		CHAR(1)	not null
create_date		DATE	not null
create_userid		VARCHAR2(20)	not null
update_date		DATE	null
update_userid		VARCHAR2(20)	null

ui_misc_opt_control

control_type	<pk>	varchar2(8)	not null
description		varchar2(40)	not null

control_type = control_type

ui_misc_option

misc_option_id	<pk>	number(10)	not null
country_code	<fk>	varchar2(8)	not null
control_type	<fk>	varchar2(8)	not null
option_name		varchar2(40)	not null
value		varchar2(40)	not null

country_code = country_code

ui_misc_opt_country

country_code	<pk>	varchar2(8)	not null
country_name		varchar2(40)	null

ui_person_x_name

u_id	<pk>	varchar2(15)	not null
x_last_name		varchar2(40)	not null
x_first_name		varchar2(40)	not null
x_mother_mn		varchar2(40)	null
x_maiden		varchar2(40)	null
x_middle_name		varchar2(30)	null

ui_nickname

ui_nickname_id	<pk>	NUMBER(10)	not null
formal_name		VARCHAR2(40)	not null
nick_name		VARCHAR2(40)	not null
create_date		DATE	not null
create_userid		VARCHAR2(20)	not null
update_date		DATE	null
update_userid		VARCHAR2(20)	null

ui_ctrl_file_hist

ui_ctrl_file_hist_id		NUMBER(10)	not null
ui_ctrl_file_id	<fk>	NUMBER(10)	not null
file_type		VARCHAR2(3)	not null
file_name		VARCHAR2(18)	not null
file_ext		VARCHAR2(3)	not null
file_content		LONG	null
content_date		DATE	not null
save_date		DATE	not null
create_date		DATE	not null
create_userid		VARCHAR2(20)	not null
update_date		DATE	null
update_userid		VARCHAR2(20)	null

ui_ctrl_field

ui_ctrl_field_id	<pk>	NUMBER(10)	not null
field_name		VARCHAR2(2)	not null
field_type		VARCHAR2(2)	not null
field_length		NUMBER(10)	not null
field_missing		VARCHAR2(2)	not null
description		VARCHAR2(48)	null
ui_ctrl_column_id		NUMBER(10)	null
create_date		DATE	not null
create_userid		VARCHAR2(20)	not null
update_date		DATE	null
update_userid		VARCHAR2(20)	null

ui_table

ui_table_id	<pk>	NUMBER(10)	not null
ui_table_name		VARCHAR2(30)	not null

ui_table_id = ui_table_id

ui_table_column

table_column_id	<pk>	NUMBER(10)	not null
ui_table_id	<fk>	NUMBER(10)	not null
column_name		VARCHAR2(30)	not null
default_label		VARCHAR2(40)	not null
label		VARCHAR2(40)	not null
visible		CHAR(1)	not null
required		CHAR(1)	not null
read_only		CHAR(1)	not null

control_sec

ctrl_key	<pk>	VARCHAR2(10)	not null
description		VARCHAR2(50)	null
ctrl_value		VARCHAR2(10)	null
create_date		DATE	null

ui_no_passwd

no_passwd	<pk>	VARCHAR2(20)	not null

📄 pk_idx_ui_no_passwd

ui_alias_x_name

ui_alias_id	<pk>	number(10)	not null
u_id		varchar2(15)	not null
x_last_name		varchar2(40)	not null
x_first_name		varchar2(40)	not null
x_middle_name		varchar2(30)	null

stc_user

usersl_id	<pk>	NUMBER(10)	not null
user_id		VARCHAR2(20)	not null
last_name		VARCHAR2(30)	not null
first_name		VARCHAR2(30)	not null
middle_initial		VARCHAR2(1)	null
user_desc		VARCHAR2(100)	null
email_address		VARCHAR2(100)	null
admin		VARCHAR2(1)	not null
eff_date		DATE	not null
exp_date		DATE	null
change_pass		VARCHAR2(1)	not null
login_attempts		NUMBER(10)	not null
create_date		DATE	not null
create_userid		VARCHAR2(20)	not null

pk_stc_user2

usersl_id = usersl_id

ui_login_current

ui_login_current_id	<pk>	NUMBER(10)	not null
usersl_id	<fk>	NUMBER(10)	not null
login_time		DATE	not null
logout_time		DATE	null
update_time		DATE	null
session_id		VARCHAR2(15)	null

usersl_id = usersl_id

ui_notify_user

ui_notify_user_id	<pk>	NUMBER(10)	not null
event_code		VARCHAR2(5)	not null
usersl_id	<fk>	NUMBER(10)	not null
create_date		DATE	not null
create_userid		VARCHAR2(20)	not null

usersl_id = usersl_id

stc_user_group

usergp_id	<pk>	NUMBER(10)	not null
group_id	<fk>	NUMBER(10)	not null
usersl_id	<fk>	NUMBER(10)	not null
eff_date		DATE	not null
exp_date		DATE	null
create_date		DATE	not null
create_userid		VARCHAR2(20)	not null

stc_user_region

user_region_id	<pk>	NUMBER(10)	not null
usersl_id	<fk>	NUMBER(10)	not null
region_code		VARCHAR2(8)	not null
create_date		DATE	not null
create_userid		VARCHAR2(20)	not null

usersl_id = usersl_id

group_id = group_id

stc_group

group_id	<pk>	NUMBER(10)	not null
group_name		VARCHAR2(30)	not null
group_desc		VARCHAR2(100)	null
eff_date		DATE	not null
exp_date		DATE	null
create_date		DATE	not null
create_userid		VARCHAR2(20)	not null

ui_login

ui_login_id	<pk>	NUMBER(10)	not null
usersl_id	<fk>	NUMBER(10)	not null
login_time		DATE	not null
logout_time		DATE	null
update_time		DATE	null
session_id		VARCHAR2(15)	null

usersl_id = usersl_id

group_id = group_id

stc_group_acc

group_acc_id	<pk>	NUMBER(10)	not null
group_id	<fk>	NUMBER(10)	not null
acc_def_id	<fk>	NUMBER(10)	not null
eff_date		DATE	not null
exp_date		DATE	null
create_date		DATE	not null
create_userid		VARCHAR2(20)	not null

ui_user_passwd_hist

ui_passwd_hist_id	<pk>	NUMBER(10)	not null
usersl_id	<fk>	NUMBER(10)	not null
passwd		VARCHAR2(20)	not null
use_date		DATE	not null

ui_user_passwd_hist_1

usersl_id = usersl_id

acc_def_id = acc_def_id

stc_user_acc

user_acc_id	<pk>	NUMBER(10)	not null
usersl_id	<fk>	NUMBER(10)	not null
acc_def_id	<fk>	NUMBER(10)	not null
eff_date		DATE	not null
exp_date		DATE	null
create_date		DATE	not null
create_userid		VARCHAR2(20)	not null

acc_def_id = acc_def_id

stc_acc_def

acc_def_id	<pk>	NUMBER(10)	not null
module_id	<fk>	NUMBER(10)	not null
code		VARCHAR2(5)	not null
descr		VARCHAR2(50)	null
create_date		DATE	null
create_userid		VARCHAR2(15)	null

module_id = module_id

stc_common_detail

common_detail_id	<pk>	NUMBER(10)	not null
common_header_id	<fk>	NUMBER(10)	not null
code		VARCHAR2(20)	not null
descr		VARCHAR2(50)	not null
read_only		CHAR(1)	not null
create_date		DATE	not null
create_userid		VARCHAR2(20)	not null

stc_module

module_id	<pk>	NUMBER(10)	not null
appl_id	<fk>	NUMBER(10)	not null
code		VARCHAR2(5)	not null
descr		VARCHAR2(30)	not null
read_only		CHAR(1)	not null
create_date		DATE	not null
create_userid		VARCHAR2(20)	not null

common_header_id = common_header_id

appl_id = appl_id

stc_common_header

common_header_id	<pk>	NUMBER(10)	not null
appl_id	<fk>	NUMBER(10)	not null
code		VARCHAR2(8)	not null
descr		VARCHAR2(50)	not null
read_only		CHAR(1)	not null
max_input_len		NUMBER(10)	not null
typ_table_code		VARCHAR2(3)	not null
create_date		DATE	not null
create_userid		VARCHAR2(20)	not null

appl_id = appl_id

stc_appl

appl_id	<pk>	NUMBER(10)	not null
code		VARCHAR2(8)	not null
descr		VARCHAR2(30)	not null
read_only		CHAR(1)	not null
create_date		DATE	not null
create_userid		VARCHAR2(20)	not null

Customizing e*Index

About this Chapter

Overview

This chapter describes the default Monk files and e*Index demo schema that you can customize for your own processing environment. It also explains how to configure your e*Ways for e*Index.

The following diagram illustrates the contents of each major topic in this chapter. For the page numbers on which specific topics appear, see the next page of this chapter.

About the e*Index Schema	Learn about the Monk files you can customize, and the schema files provided with e*Index
Configure the e*Ways	Learn how to configure your e*Ways for e*Index to process data according to your business requirements

What's Inside

This chapter provides information related to the topics listed below.

Learning About e*Index Schema Components

Overview

When you install e*Index, you install several files that you can customize in order to make sure your data is translated and formatted appropriately. You can install a sample schema on which you can base your production e*Ways. This section of the chapter discusses the schema components you install, and the files that you can modify to create your e*Index schemas.

Schema Component Distribution

When you install the e*Index schema, certain components are placed in the default directory, and some are placed in the schema director of the registry. The files in the default directory include the binary files that you cannot modify, **ui-fns.monk**, and **eiEvent.ssc**. The remaining files are installed in the demo schema directory in the registry. The files are distributed this way so you can install e*Index upgrades without overwriting you customized schema files.

About the Sample Schema

SeeBeyond provides a sample e*Gate schema to help you determine how to set up your own schema for your e*Index environment. You can name the sample schema when you install it. The sample schema includes an executable file for your e*Ways, and sample Event Type Definition (ETD), Collaboration, and configuration files.

■ The executable file is named **stcewgenericmonk.exe**, and is located in the **/default/bin/<os>** directory on the registry.

■ The sample Event Type Definition file is named **eiEvent.ssc**, and is located in the **/default/monk_scripts/ui** directory on the registry.

---

*Tip: The ETD file is placed in the default directory of the registry. If you plan to customize the ETD, you should rename the file, and copy and commit it to the e*Index schema directory (in **/<schema_name>/monk_scripts/ui**). This ensures that your customizations are not overwritten if **eiEvent.ssc** is modified in future versions of e*Index.*

■ The sample Collaboration file is named **uidb.dsc**, and is located in the **/<schema_name>/runtime/monk_scripts/ui** directory on the registry.

■ The sample configuration file for the sending e*Way is named **ewUIDB.cfg,** and the sample configuration file for the polling e*Way is named **ewUIPOLL.cfg**. These files are located in the

/<schema_name>/runtime/configs/stcewgenericmonk directory on the registry.

If you are new to e*Index, you should review the sample schema before creating your production schema.

About the Collaboration Script

The sample Collaboration script, **uidb.dsc**, is referenced from the configuration files in the e*Ways for e*Index. This file specifies the input and output event type definitions, and calls the Monk function **ui-process-person**, which is described in Chapter 4, "e*Index Monk APIs". The function **ui-process-person** provides the basic data processing rules for incoming data, and you can customize the function to process incoming records in a way that best meets your processing requirements. You can insert additional Monk APIs into **uidb.dsc** in order to further customize how incoming events are processed. Currently, this file cannot be edited in the e*Gate Collaboration Editor. Make sure to use the **stcregutil** command to commit the files to the registry if you modify **uidb.dsc**. See "e*Index API Descriptions" in Chapter 4 for complete information about the e*Index Monk APIs you can use in this file.

What is the e*Index Monk Library?

The e*Gate environment includes a library of Monk functions. When you install e*Index, three Monk files are loaded into the Monk library of the e*Index schema, and one is loaded into the default schema on the registry. You can customize these files as needed to meet your business requirements.

- **ui-fns.monk**
  This file is loaded into the **/default/monk_library/ui** directory in the registry. It contains the Monk functions that allow you to reformat the data contained in the Events that are passed through the e*Ways for e*Index. These functions allow you to standardize the way data is presented. For example, if your incoming data includes telephone numbers in the format **(xxx)yyy-zzzz** or **xxx-yyy-zzzz**, you can use the strip-phone function to reformat the data into the format **xxxyyyzzzz**, which is the format used by the e*Index database. The available functions are all described in Chapter 5, "e*Index Monk Functions".

- **ui-process-person.monk**
  This file is loaded into the **/<schema_name>/monk_library/ui** directory in the registry. It contains a sample script that you can use to process records through the e*Index database. It provides basic data processing rules for data coming into the e*Index database. This file defines a Monk function named **ui-process-person**, which you can call in the collaboration scripts for your e*Ways. e*Index provides additional Monk APIs that you can use to modify **ui-process-person.monk** and to

customize how incoming records are processed. The available Monk functions are described in chapter 4 of this guide. Changes to **ui-process-person.monk** can be made using any text editor. Remember to commit the changed file to the e*Gate Registry or your changes will not be implemented.

■ **ui-custom.monk**
This file is loaded into the **/<schema_name>/runtime/monk_library/ui** directory in the registry. It contains the commands that create the Monk lists used as parameters for the Monk APIs in Collaboration scripts. These commands include get-demographics, get-transaction, get-alias, get-address, and get-phone. Just as they sound, these commands retrieve lists of demographic, transaction, alias, address, or telephone information from an incoming Event. You can customize this file using the Monk functions in **ui-fns.monk** to reformat the data in an Event. Examples of how this file can be customized are included in the examples in Chapter 5, "e*Index Monk Functions". You can modify **ui-custom.monk** can be made using any standard text editor, such as Microsoft WordPad or Unix vi. Remember to commit the changed file to the e*Gate Registry or your changes will not be implemented.

■ **ui-stdver-eway-funcs.monk**
This file is loaded into the **/<schema_name/runtime/monk_library/ui** directory in the registry. The configuration files for your e*Index e*Ways call **ui-stdver-eway-funcs.monk** to initialize files, connect to the database, verify the connection, shut down the connection, process event handling, and so on. This file contains commands for processing both inbound and outbound events. The commands included in the **ui-stdver-eway-funcs.monk** file are:

• ui-stdver-init (Monk initialization file)

• ui-stdver-startup (startup function for the sending e*Way)

• ui-stdver-conn-estab (external connection establishment function)

• ui-stdver-conn-ver (external connection verification function)

• ui-stdver-conn-shutdown (external connection shutdown function)

• ui-stdver-pos-ack (positive acknowledgment function for the sending e*Way)

• ui-stdver-pos-neg-ack (negative acknowledgment function for the sending e*Way)

• ui-stdver-shutdown (shutdown command notification function)

• ui-stdver-proc-outgoing (process outgoing message function)

• ui-stdver-proc-outgoing-stub (place holder for process outgoing message function)

• ui-poll (exchange data with external function)

• ui-stdver-data-exchg (place holder for exchange data with external function)

- ui-stdver-data-exchg-stub (place holder for exchange data with external function)
- ui-poll-startup (startup function for the polling e*Way)
- ui-poll (exchange data with external function)
- ui-poll-pos-ack (positive acknowledgment function for the polling e*Way)
- ui-poll-neg-ack (negative acknowledgment function for the polling e*Way)

For more information, see "Standard Monk API Descriptions" in Chapter 4 of this guide.

About e*Ways

e*Ways provide the points of contact between the e*Gate system and external applications. They handle the communication details necessary to send and receive information including:

- responding to or generating positive and negative acknowledgments
- rules that govern resend and/or reconnect criteria
- timeout logic
- data envelope parsing and reformatting
- buffer size
- retrieval/transmission schedules

In addition to handling communications, e*Ways can also apply business logic within Collaboration Rules to perform any of e*Gate's data identification, manipulation, and transformation operations. e*Ways are tailored to meet the communication requirements of a specific application or protocol. You can extend the capabilities of the e*Index e*Ways using the Monk programming language to handle custom communications requirements.

*Note: For more a more thorough description about how to work with and configure e*Ways, see "Working with e*Ways" in the* e*Gate Integrator User's Guide.

About Monk Configuration Functions

e*Ways use Monk functions to start and stop scheduled operations, exchange data with the external system, package data as e*Gate Events, send those Events to Collaborations, and manage the connection between the e*Way and the external systems. The Monk Configuration options, discussed later in this chapter, control the Monk environment and define the functions used to perform these basic e*Way operations. You can create and modify these functions using the e*Gate Collaboration Rules Editor or a text editor (such as Notepad or UNIX vi). Remember to commit the files to the registry so the changes to be recognized.

The Monk functions you use for the Monk Configuration options fall into the following groups:

Type of Operation	Name
Initialization	Startup Function (also Monk Environment Initialization File)
Connection	External Connection Establishment Function Connection Verification Function External Connection Shutdown Function
Schedule-driven data exchange	Exchange Data with External Function Positive Acknowledgment Function Negative Acknowledgment Function
Shutdown	Shutdown Command Notification Function
Event-driven data exchange	Process Outgoing Message Function

The series of illustrations on the following pages illustrate how these functions work together to perform startup, data exchange, and shutdown functions within an e*Way.

Initialization and Startup Functions

Figure 3-1 below illustrates how an e*Way executes the initialization and startup functions for e*Index.

Figure 3-1: Initialization Process
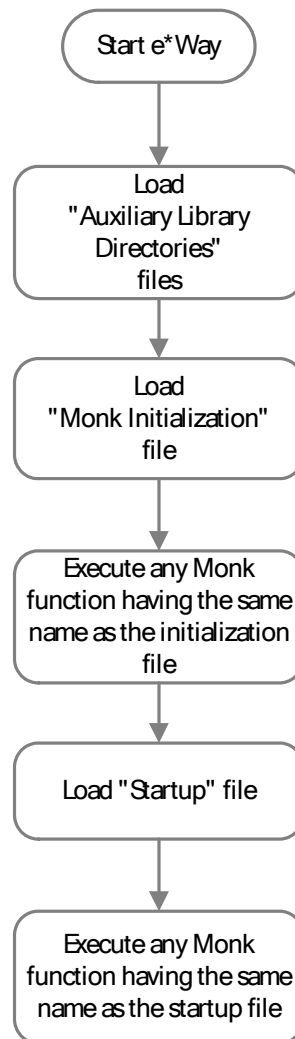
Connectivity Functions

Figure 3-2 below illustrates how the e*Ways execute the connectivity and verification functions. You can schedule when these functions are called by specifying schedules for re-establishing the connection when the connection is inactive, and for rechecking the connection after it is found to be active.

Figure 3-2: Connection Processes

```
        ┌──────────────────────┐
        │ Connect e*Way to      │
        │ external system       │
        └──────────────────────┘
                   │
                   ▼
            ╱─────────────╲
           ╱   Internal     ╲          ┌─────────────────────┐
          ╱ flag shows        ╲──No──▶ │ Wait for "Down       │
          ╲ connection active? ╱       │ Timeout" schedule    │
           ╲                  ╱         └─────────────────────┘
            ╲────────────────╱                    │
                   │                              ▼
                  Yes                  ┌─────────────────────┐
                   │                   │ Call **External      │
                   ▼                   │ Connection           │
        ┌──────────────────────┐       │ Establishment** func │
        │ Wait for "Up Timeout" │      └─────────────────────┘
        │ schedule              │
        └──────────────────────┘
                   │
                   ▼
        ┌──────────────────────┐
        │ Call **External       │
        │ Connection            │
        │ Verification** func   │
        └──────────────────────┘
```
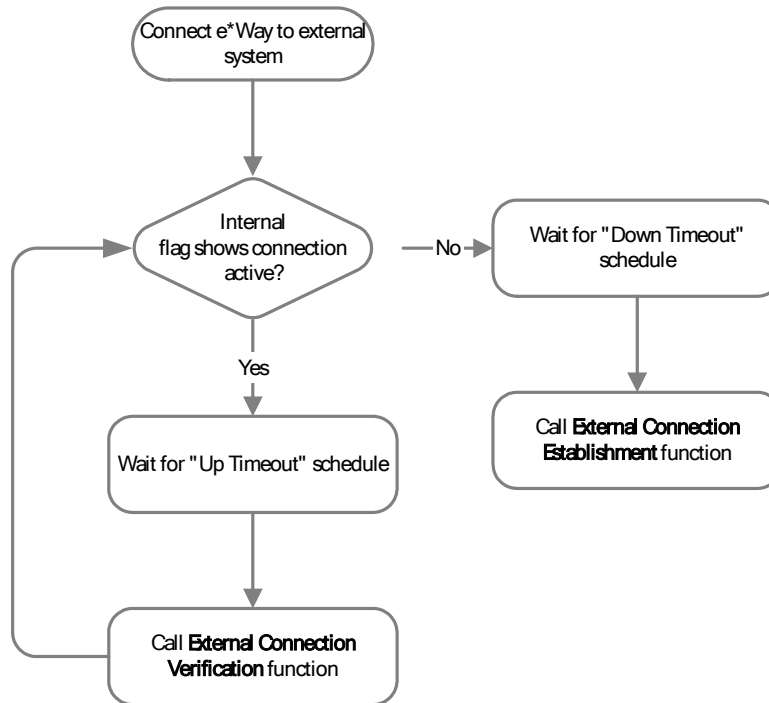
Schedule-Driven Data Exchange Functions

Figure 3-3 below illustrates how the e*Way executes the data exchange Monk functions when the e*Way is configured to be schedule-driven. For more information about scheduling data exchanges, see "Start Exchange Data Schedule" on page 3-16.

Figure 3-3: Schedule-driven Processes
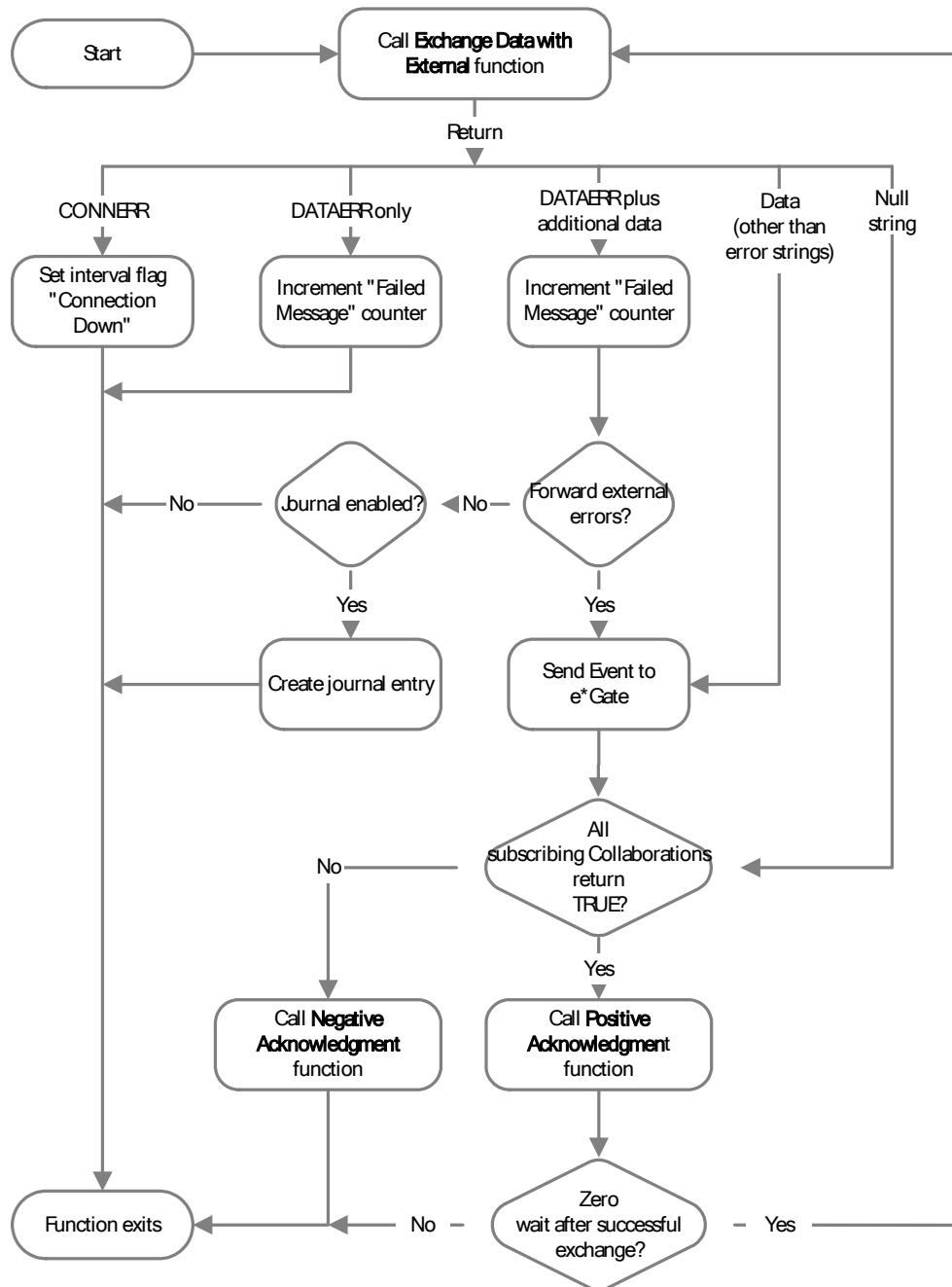
Event-Driven Data Exchange Functions

Figure 3-4 below illustrates how the e*Way executes the data exchange Monk functions when the e*Way is configured to be event-driven. For more information about event-driven data exchanges, see "Process Outgoing Message Function" on page 3-22.

Figure 3-4: Event-driven Processes
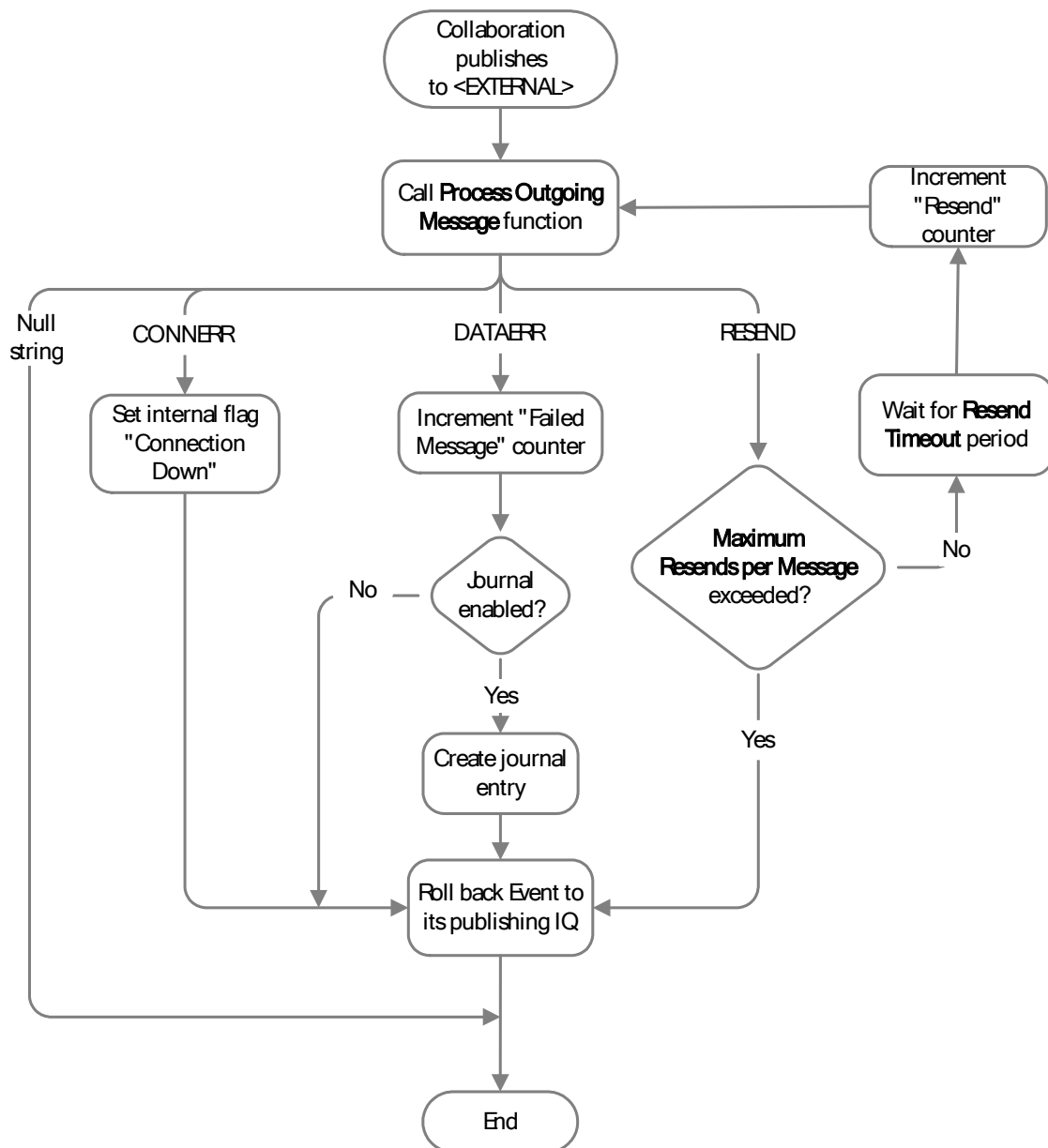
Shutdown Functions

Figure 3-5 below illustrates how the e*Way executes the functions that shutdown the e*Way when the Control Broker issues a shutdown command.

Figure 3-5: Shutdown Process
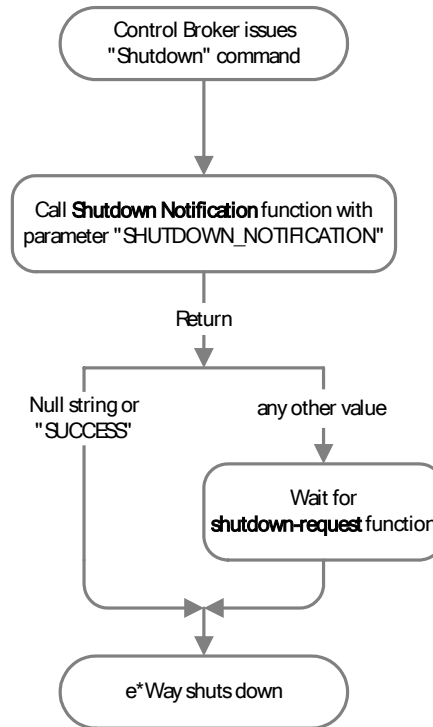
Learning About the e*Way Configuration Parameters

Overview

Before you can run e*Index, you must configure the e*Ways using the e*Way Edit Settings window, which is accessed from the e*Gate Enterprise Manager GUI. e*Index provides a default configuration file that you can modify using this window. This chapter describes the procedure for configuring the sample e*Ways. You can edit an existing e*Way and rename an e*Way in the e*Gate Enterprise Manager. Procedures for creating and editing e*Gate components are provided in the Enterprise Manager's online help.

Modifying e*Way Configuration Parameters

You can change the settings for your e*Way configuration parameters using the e*Way Editor.

Use the Edit Settings window to modify your e*Way configuration parameters
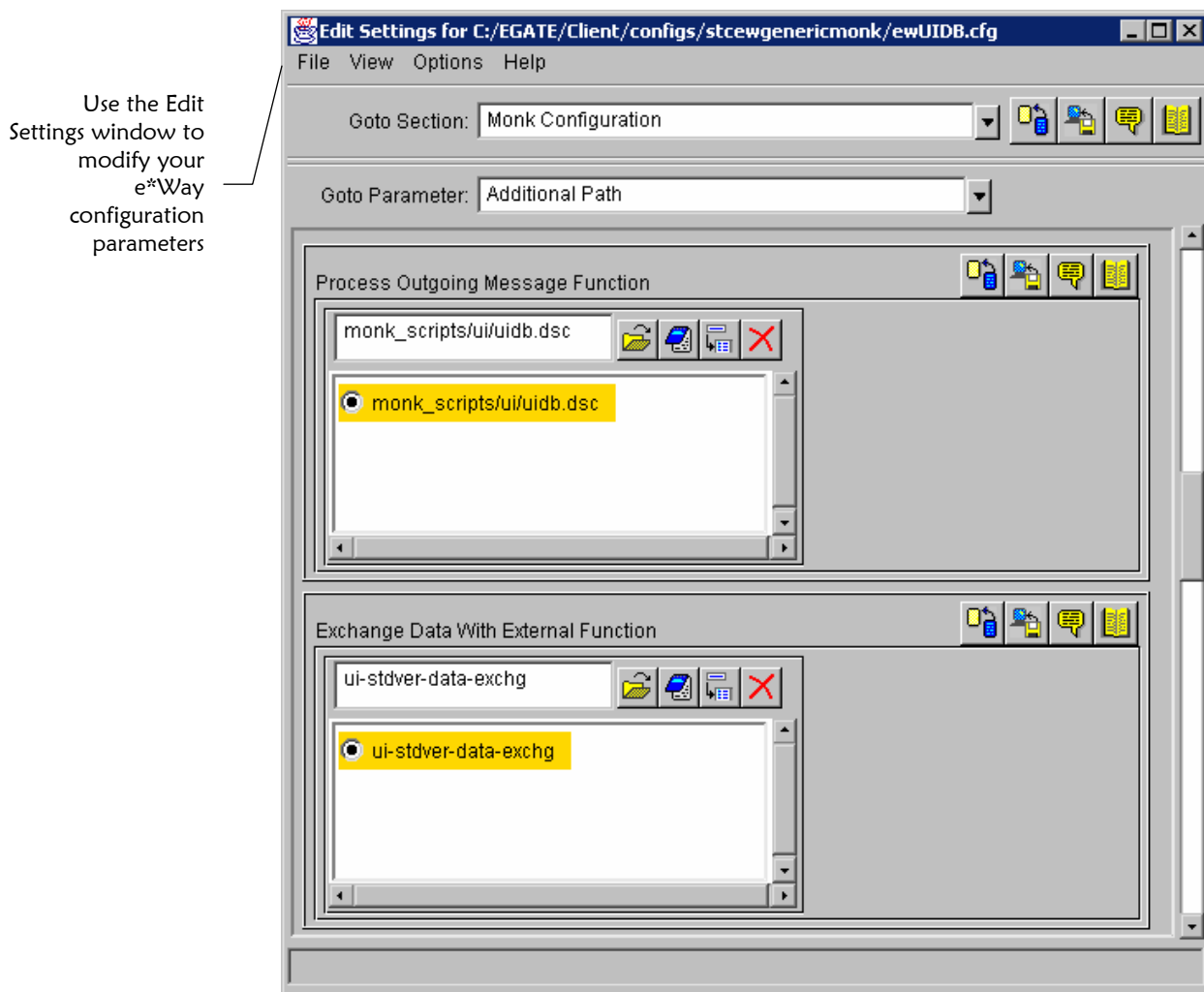
▶ **To change e*Way configuration parameters:**

Before you begin:

    ✓ Obtain information about the e*Way you need to configure, such as the names of the Monk functions you need to enter as parameters for the configuration file

    ✓ Open the e*Gate Enterprise Manager

1 In the Enterprise Manager'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. Do not to change any of the default arguments unless you have a specific need to do so.

4 Modify the e*Ways configuration parameters. The parameters for the sending and polling e*Ways are described on the following pages. The parameters are organized into the following sections (corresponding to the sections listed in the **Goto Section:** field at the top of the Edit Settings window):

- General Settings
- Communication Setup
- Monk Configuration
- Database Setup

Note: For more information about how to use the e*Way Editor, see the e*Way Editor's online help or "Working with e*Ways" in the e*Gate Integrator User's Guide*.*

General Settings

The General Settings section controls basic operational parameters, such as error handling information and the name and path of the journal files.

Journal File Name

Description

In this parameter, you can specify the name and path of the journal files for the displayed e*Way. The Journal file logs information in the following conditions:

■   When the number of resends is exceeded (see "Max Resends Per
    Message" on page 3-15).

■   When an external error occurs, but **Forward External Errors** is set to **No**
    (for more information, see "Forward External Errors" on page 3-15).

Required Values

A valid filename, optionally including an absolute path (for example,
c:\temp\filename.txt). If the path you specify does not exist, the e*Way will
create it. If you do not specify an absolute path, the file is stored in e*Gate's
SystemData directory. See the *e*Gate Integrator System Administration and
Operations Guide* for more information about file locations. There is no
default for this field.

Max Resends Per Message

Description

The value you specify for the **Max Resends Per Message** parameter
determines the maximum number of times the e*Way attempts to resend an
Event to the external system after receiving an error. When this maximum
number is reached, the Event fails and is written to the journal file.

Required Values

An integer between 1 and 1,024. The default for the sending e*Way is **1**; the
default for the polling e*Way is **5**.

Max Failed Messages

Description

The value you specify for **Max Failed Messages** determines the maximum
number of failed Events the e*Way allows. When the e*Way reaches the
specified number of failed Events, the e*Way shuts down.

Required Values

An integer between 1 and 1,024. The default for the sending e*Way is **1,024**;
the default for the polling e*Way is **3**.

Forward External Errors

Description

The value specified for the **Forward External Errors** parameter determines
whether error messages that begin with the string **DATAERR** and are
received from the external system are queued to the e*Way's configured

queue. See "Exchange Data with External Function" on page 3-23 for more information.

Required Values

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

Communication Setup

The Communication Setup parameters control the schedule by which the e*Way obtains data from the external system.   Typically, in an e*Index implementation, you want the e*Ways to continually check for data from external systems or from the e*Index database.

Note: The schedule you set in the e*Way's properties in the Enterprise Manager determines when the e*Way executable runs. The schedule you set within the parameters discussed in this section (using the e*Way Editor) determines when data is exchanged. Be sure the schedule you set for the data exchange falls within the schedule that was set up for the executable in the Enterprise Manager.

Start Exchange Data Schedule

Description

The **Start Exchange Data Schedule** parameter allows you to establish a schedule to invoke the **Exchange Data with External** function (for more information, see "Exchange Data with External Function" on page 3-23).

Required Values

One of the following:

- One or more specific dates and times

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

There are no defaults for this parameter.

If you set a schedule using **Start Exchange Data Schedule**, you must also define the following:

- Exchange Data with External Function (described on page 3-23)

- Positive Acknowledgment Function (described on page 3-24)

- Negative Acknowledgment Function (described on page 3-25)

If you do not define these three parameters, 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 NACK to the external system (using the **Positive Acknowledgment Function** or **Negative Acknowledgment Function**). It also determines whether the connection to the external system is active. If no ACK/NACK is pending and the connection is active, the e*Way immediately executes the function specified in the **Exchange Data with External Function** parameter. The function will continue to 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 3-23, "Exchange Data Interval" on page 3-17, and "Stop Exchange Data Schedule" below for more information.

Stop Exchange Data Schedule

Description

The **Stop Exchange Data Schedule** parameter establishes the schedule to stop data exchanges. This field is optional, and if you leave it blank, the e*Way will continually search for data to process. This parameter is not used by the sending e*Way.

Required Values

One of the following:

■ One or more specific dates and times

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

There are no defaults for this parameter.

Exchange Data Interval

Description

The value you specify for the **Exchange Data Interval** parameter determines the number of seconds the e*Way waits between calls to **Exchange Data with External Function** during scheduled data exchanges. This parameter is not used by the sending e*Way, which only uses a stub function for the **Exchange Data with External Function** parameter. The interval can be used by the polling e*Way.

Required Values

An integer between 0 and 86,400. The default for the sending e*Way is **120**.
The default for the polling e*Way is **10**.

Additional Information

If the **Zero Wait Between Successful Exchanges** parameter is set to **Yes** and
the **Exchange Data with External** function returns data, then the **Exchange
Data Interval** setting is ignored and the e*Way invokes the **Exchange Data
with External** function immediately. When **Exchange Data with External**
does not return data, then the **Exchange Data Interval** setting is used.

If the **Exchange Data Interval** parameter is set to zero, there is no exchange
data schedule set and the **Exchange Data with External** function is never
called.

See "Down Timeout" below and "Stop Exchange Data Schedule" on page 3-17
for more information about the data-exchange schedule.

Down Timeout

Description

The **Down Timeout** parameter specifies the number of seconds that the
e*Way waits between calls to **External Connection Establishment Function**
(for more information, see "External Connection Establishment Function" on
page 3-23).

Required Values

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

Up Timeout

Description

The value you specify for the **Up Timeout** parameter determines the number
of seconds that the e*Way waits between calls to **External Connection
Verification Function** to verify that the connection is still up (for more
information, see "External Connection Verification Function" on page 3-24).

Required Values

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

Resend Timeout

Description

Use this parameter to specify the number of seconds the e*Way should wait between attempts to resend an Event to the external system after receiving an error message from an external system.

Required Values

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

Zero Wait Between Successful Exchanges

Description

The **Zero Wait Between Successful Exchanges** parameter allows you to select whether to initiate the exchange immediately after a successful previous exchange or to initiate data exchange after the amount of time specified in **Exchange Data Interval** has passed.

Required Values

Yes or **No**. If this parameter is set to **Yes**, the e*Way immediately invokes the **Exchange Data with External** function if the previous exchange function returned data. If this parameter is set to **No**, the e*Way always waits the number of seconds specified by **Exchange Data Interval** before invoking **Exchange Data with External Function**. The default is **No**, but you may want to set it to **Yes** for the polling e*Way.

See "Exchange Data with External Function" on page 3-23 for more information.

Monk Configuration

The Monk Configuration parameters discussed in this section control the Monk environment and define the Monk functions that are used to perform basic e*Way operations.   These parameters help you set up the information required by the e*Way to use Monk to communicate with external systems. The e*Way uses Monk functions to start and stop scheduled operations, exchange data with external systems, package and send Events to e*Gate, send Events to Collaborations, and manage connections between the e*Way and external systems. Figures 3-1 through 3-5 earlier in this chapter illustrate how the Monk Configuration parameters are executed.

You can create and modify these functions using any standard text editor (such as Microsoft WordPad, Notepad, or UNIX vi). For more information

about the Monk functions used for the Monk Configuration parameters, see "Standard Monk API Descriptions" in Chapter 4 of this guide.

Parameters that require the name of a Monk function 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

The **Additional Path** parameter allows you to specify a path to append to the load path, which is the path Monk uses to locate files and data. The directory specified for **Additional Path** is searched before the default load path.

Required Values

A pathname, or a series of paths separated by semicolons. This parameter is optional and you may leave it blank. The default for both e*Index e*Ways is **monk_scripts/ui**.

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 starts up.

Auxiliary Library Directories

Description

The **Auxiliary Library Directories** parameter allows you to specify a path to directories that contain additional libraries. Any **.monk** files found within those directories are automatically loaded into the e*Way's Monk environment.

Required Values

A pathname, or a series of paths separated by semicolons. The default is **monk_library/dart;monk_library/ui**.

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 starts up.

This parameter is optional and you may leave it blank.

Monk Environment Initialization File

The **Monk Environment Initialization File** parameter allows you to specify a file that contains environment initialization functions, which are 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 or command within the load path, or filename plus path information (relative or absolute). If you specify path information, the path you specify is appended to the load path. The default is **ui-stdver-init**, which is described in the API list in Chapter 4 of this guide. For more information about the load path, see "Additional Path" on page 3-20.

Additional information

When you specify a Monk environment initialization file, the e*Way loads the file and tries 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 only once, when the e*Way starts up.

Startup Function

Description

Use the **Startup Function** parameter to specify a Monk function that the e*Way loads and invokes upon startup or whenever the e*Way's configuration changes before it enters into its initial communication state. This function allows the external system to be initialized before information exchange starts.

Required Values

The name of a Monk function, or the name of a file (optionally including path information) containing a Monk function. The default is **ui-stdver-startup** for the sending e*Way, and **ui-poll-startup** for the polling e*Way. These commands are described in the API list in Chapter 4 of this guide.

Additional information

This function is called after the e*Way loads the specified Monk environment initialization file and any files within the specified auxiliary directories.

If you specify a file name instead of a command, the e*Way loads the file and tries to invoke a function of the same base name as the file name. 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

The function you specify as your **Process Outgoing Message Function** indicates the Monk function responsible for sending outgoing messages (Events) from the e*Way to the external system. This function is event-driven (unlike the **Exchange Data with External** function, which is schedule-driven).

Required Values

The name of a Monk function, or the name of a file (optionally including path information) containing a Monk function. You may not leave this field blank. The default function for the sending e*Way is **uidb.dsc**. The default function for the polling e*Way is **ui-stdver-proc-outgoing**.

Exchange Data with External Function

Description

The function you specify for **Exchange Data with External Function** is the function that initiates an exchange of data with an external system. This function is called according to a schedule (unlike the **Process Outgoing Message Functio**n, 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. The default is **ui-stdver-data-exchg** for the sending e*Way. **ui-stdver-data-exchg** is a placeholder for the file that controls the exchange of data, since the e*Index sending e*Way is not schedule-driven. For the polling e*Way, the default is **ui-poll**.

External Connection Establishment Function

Description

The **External Connection Establishment Function** parameter specifies the Monk function that the e*Way calls 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 is required. The default is **ui-stdver-conn-estab**, which is described in the API list in Chapter 4 of this guide.

Additional Information

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

The **External Connection Verification Function** specifies the Monk function that the e*Way calls to confirm that the external system is operating and available.

Required Values

The name of a Monk function. This function is optional; if no function is specified, the e*Way executes the **External Connection Establishment Function** in its place. The default is **ui-stdver-conn-ver**, which is described in the API list in Chapter 4 of this guide.

Additional Information

This function is executed according to the interval specified within the **Up Timeout** parameter, and is only called according to this schedule. **External Connection Establishment Function** (see previous page) is called when the e*Way has determined that its connection to the external system is down.

External Connection Shutdown Function

Description

This parameter specifies the Monk function that the e*Way calls to shut down the connection to the external system.

Required Values

The name of a Monk function. The default is **ui-stdver-conn-shutdown**, which is described in the API list in Chapter 4 of this guide.

Additional Information

Include in this function any required "clean up" that must be performed as part of the shutdown procedure before the e*Way exits.

Positive Acknowledgment Function

Description

The **Positive Acknowledgment Function** parameter specifies the Monk function that the e*Way calls 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 **Exchange Data with External Function** is defined. The default for the sending e*Way is **ui-stdver-pos-ack**. The default for the polling e*Way is **ui-poll-pos-ack**. These commands are described in the API list in Chapter 4 of this guide.

Negative Acknowledgment Function

Description

The **Negative Acknowledgment Function** parameter specifies a Monk function that the e*Way calls when the e*Way fails to process and queue data 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 **Exchange Data with External Function** is defined. The default for the sending e*Way is **ui-stdver-neg-ack**. The default for the polling e*Way is **ui-poll-neg-ack**. These commands are described in the API list in Chapter 4 of this guide.

Shutdown Command Notification Function

Description

The **Shutdown Command Notification Function** specifies the Monk function that is 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. The default is **ui-stdver-shutdown**, which is described in the API list in Chapter 4 of this guide.

Additional Information

If you postpone a shutdown using this function, be sure to use the **shutdown-request** function to complete the process in a timely manner.

Database Setup

The parameters in this section allow you to specify information about your database, such as the name and type of database, and the user name and password with which to log on to the database.

Database Type

Description

Specify the type of database to connect to in the **Database Type** parameter.

Required Values

ORACLE9i, ORACLE8i, ODBC, or SYBASE. Any other value is effectively equal to ODBC. Select one of the following, depending on the database platform you are using: SYBASE, ORACLE8i, ORACLE9i, or ODBC.

Database Name

Description

Specify the name of the database to connect to in the **Database Name** parameter.

Required Values

You can enter any valid string for this parameter.

User Name

Description

In the **User Name** parameter, specify the user log on name with which the database can be accessed.

Required Values

You can enter any valid string for this parameter. The default for this field is **ui**.

Encrypted Password

Description

In the **Encrypted Password** parameter, enter the password that provides access to the database for the **User Name** you entered.

Required Values

Any valid string.

Important! *Changes to Monk files can be made using the Collaboration Rules Editor (except to **ui-custom.monk**) or with a text editor. However, if you use a text editor to edit Monk files directly, you must commit these changed files to the e*Gate Registry or your changes will not be implemented.*

*For more information about committing files to the e*Gate Registry, see the Enterprise Manager's online Help system, or the **stcregutil** command-line utility in the* e*Gate Integrator System Administration and Operations Guide.

---

e*Index Monk APIs

About this Chapter

Overview

This chapter presents the background information you need to create Monk scripts using the APIs provided in the e*Index Monk library.

The following diagram illustrates the contents of each major topic in this chapter. For the page numbers on which specific topics appear, see the next page of this chapter.

About e*Index Monk APIs — Learn about the standard Monk APIs and Monk lists defined for e*Index

e*Index Monk APIs — Learn about the usage, syntax, and parameters of the Monk APIs in the e*Index Monk API library

Standard e*Index Monk APIs — Learn about the implementation, syntax, and parameters of the external Monk APIs found in ui-stdver-funcs-eways.monk

What's Inside

This chapter provides information related to the topics listed below.

Learning About e*Index Monk APIs

Overview

This section of the chapter provides the background information you should know before using the e*Index Monk APIs.

What are e*Index Monk APIs?

*e*Index Monk APIs* are commands that you can call in your Collaboration scripts for e*Index in order to perform functions specific to the e*Index database. The Monk APIs include commands to perform tasks such as finding a person based on their UID or local ID, inserting demographic or alias information into a person's records, merging and unmerging records, queuing and dequeuing Events, and so on.

What are Standard Monk APIs for e*Index?

Standard Monk APIs are commands that you can call in your configuration file to perform certain database activities. Using these functions you can establish and verify a connection to a database, shutdown a database connection, initialize Monk files, read the *ui_control* table of the e*Index database, and so on. These functions are defined in the file **ui-stdver-eway-funcs.monk**, and are described under "Standard Monk API Descriptions" later in this chapter.

What Monk Lists are Defined for e*Index?

Monk lists contain specific information associated with a record that exists in the e*Index database, or that is being converted into the e*Index database. Monk lists are frequently used as parameters for e*Index Monk APIs. The default Monk lists for e*Index are defined in **ui-custom.monk** (located in the Monk library), and you can customize these lists for your business requirements. Use any standard editor to modify this file, and make sure to commit the file to the e*Gate registry using the **stcregutil** command. The defined lists include **demographics-list**, **transaction-list**, **alias-list**, **address-list**, and **phone-list**.

For fields in Monk lists that have no value or are unknown, you can specify null (""). Date fields should be of the format **YYYY-MM-DD** and time fields should be of the format **HH:mm:SS** using a 24-hour clock. The remaining fields do not have formatting constraints, though some are validated against the affected database table. The fields **class1** through **class5** are reserved and should not be used.

How do Control Keys Affect APIs?

The e*Index control keys, maintained in the e*Index Administrator GUI, allow you to configure how e*Index processes data. Most of the control keys affect the e*Index GUI, but some influence how data from external systems is processed as well.

1XACTMTCH

The *1XACTMTCH* control key allows you to specify whether all records that match your search criteria and have weights above the match threshold (MATCHTHRES) are treated as potential duplicates. Step 5 on page 2-11 of this guide describes how exact matching affects the **ui-process-person** API.

BLNKONUPDT

Using the *BLNKONUPDT* control key, you can specify whether to treat a blank field as null when performing an update from the e*Index e*Ways. For records entered through the back-end, a blank field is indicated by double quotes ("") and not by an empty string. The BLNKONUPDT control key determines whether the API interprets double quotes literally as double quotes (when the key is set to **No**) or whether it interprets double quotes as a null field, thus updating existing information with a null field or populating the database with an empty field instead of double quotes.

You can use double quotes to blank out any field in a record during an update if BLNKONUPDT is set to **Yes**. If you are inserting a new record and BLNKONUPDT is enabled, then any string consisting of double quotes is automatically converted to an empty string. If BLNKONUPDT is not enabled, and an incoming record has double quotes in any field, the APIs will populate that field in the database with double-quotes for both updates and inserts. If you pass in an empty string, the APIs ignore it during an update regardless of the value of BLNKONUPDT**.**

The BLNKONUPDT functionality does *not* apply to address and telephone number updates. When an empty string occurs in an address or telephone field, the empty string nulls out the field in the database. For example, if the original database in the record is:

> address1: 2505 Fifth St.
> address2: Apt. 102

And the address of the incoming record is:

> address1: 2505 Fifth St. Apt.102
> address2:

The API updates the record in the database like this, regardless of the BLNKONUPDT setting:

    address1: 2505 Fifth St. Apt.102
    address2:

Notes:

■ *The **ui-process-address** API has an overwrite flag that determines whether existing address is overwritten. For more information, see "**ui-process-address**" on page 4-80.*

■ *The address and telephone number updates only apply to the address or phone type specified. For example, if an existing record contains a home address and a business address, and is updated by an incoming record containing only a home address, then only the home address is updated.*

DUPTHRES

The *DUPTHRES* control key allows you to specify a minimum matching weight for which a profile is considered a potential duplicate of the profile being added. All profiles that have a higher matching probability weight than this threshold are considered potential duplicates of the new profile. Step 7 on page 2-12 of this guide discusses how the duplicate threshold affects the **ui-process-person** API.

EXTNSVSRCH

The *EXTNSVSRCH* control key does not affect back-end searches, but does enable extensive searching for the GUI. When extensive searching is enabled, a demographic alphanumeric search for member profiles also searches through alias names. To enable extensive searching for e*Way transactions, you need to modify the configurable query by adding the *ui_alias* table to the query.

MATCHTHRES

Use the *MATCHTHRES* control key to specify the matching probability weight at which e*Index automatically merges a new profile with an existing potential duplicate profile. Step 5 on page 2-11 of this guide describes how exact matching affects the **ui-process-person** API.

UIDLENGTH

The *UIDLNGTH* control key allows you to specify the length of the UIDs that e*Index assigns to each member profile. This only affects inbound Events in that the UID is inserted into each Event as it is processed through the database.

What Monk APIs are Available?

Several Monk APIs are defined in the e*Index Monk library, including functions to process demographic records, perform searches, create date/time stamps, display error messages, and so on. If you know the name of a Monk API, you can use Table 4-1 on the following pages to identify the purpose of the API.

Table 4-1: Standard e*Index Monk APIs

Use this Monk API …	to perform this action …
db-get-error-str	Display any database error messages after an API has processed.
make-connection-handle	Establish a connection handle to the database server.
ui-address-search-close	Close the address search cursor, and de-allocate the memory.
ui-address-search-next	Return the next address record from the address search cursor, and increment the search cursor position. You must call **ui-address-search-open** before calling **ui-address-search-next**.
ui-address-search-open	Search for existing address records based on an address list and open a cursor of weighted records returned from the search. This function returns the number of records in the result set.
ui-commit-transaction	Commit a transaction to the database and reset the transaction structure.
ui-config	Read the parameters defined in the *ui_control* table to configure certain system attributes, such as local ID length, UID format, and so on.
ui-deactivate-person	Deactivate an active UID record given the UID of the record to deactivate.
ui-deactivate-local-id	Deactivate an active local ID given the local ID number, the associated system, and a UID.
ui-delete-address	Delete an existing address record from the database. A transaction must be started before calling **ui-delete-address**.
ui-delete-aux-id	Delete a non-unique ID given the ID type and the ID code. Before calling **ui-delete-aux-id**, you must call **ui-start-transaction** to designate which UID record to modify.
ui-delete-queue-msg	Remove a message from the outgoing queue once it has been successfully dequeued and sent.

Use this Monk API …	to perform this action …
ui-delete-unresolved-duplicates	Remove a record's existing potential duplicate entries from *ui_duplic* during a person update so potential duplicates can re-evaluated for that record.
ui-dequeue	Retrieve and remove an active Event from the *msg_id_detail* table so the polling e*Way can send the Event to e*Gate.
ui-exists-aux-id	Searches for a specific non-unique ID type for a member given the member's UID, the ID type, and the identification code.
ui-get-alias	Retrieve a person's alias records based on that person's UID.
ui-get-all-local-id	Search for local IDs in a specific system (and their status) based on a member's UID. The status of the local IDs is ignored.
ui-get-assumed-match-enabled	Check to see if the ASSMTCH control is enabled in the Administrator. If ASSMTCH is enabled, all assumed matches made by the application are written to the *ui_asssumed_match* table.
ui-get-aux-id	Retrieve non-unique IDs given the type of ID and the member's UID.
ui-get-db-date-time	Retrieve the date and time on the database server in the following format: YYYY/MM/DD hh:mm:ss.
ui-get-demographic-changed	Check to see if a demographic record was changed as a result of the previous actions against the database.
ui-get-dupchk-enabled	Check to see if the DUPCHK control key is enabled in the Administrator. If DUPCHK is enabled, potential duplicates for a specific record are re-evaluated after that record is updated.
ui-get-error-string	Display e*Index error messages after an API has been called.
ui-get-id-system	Retrieve all local ID and system pairs associated with an individual based on their UID.
ui-get-local-id	Retrieve the member's local ID in a specific system given the system code and UID.
ui-get-person	Retrieve a person's demographic record based on that person's UID.
ui-get-transaction-date-time	Retrieve the time that the transaction that is currently in progress began.

Use this Monk API …	to perform this action …
ui-get-uid	Find a person's UID using that person's system and local ID as search criteria.
ui-get-vip	Retrieve a person's VIP flag based on that person's local ID in the specified system.
ui-insert-address	Insert a new address into the database. A transaction must be started before calling **ui-insert-address**.
ui-insert-alias	Insert alias information into the *ui_alias* table based on that person's UID.
ui-insert-assumed-match	Insert an assumed match record by assuming the first record in the search cursor (the one with the highest matching weight) is the assumed match of the incoming record. You can only call this function after **ui-search-open**.
ui-insert-aux-id	Insert a non-unique ID and ID type. Before calling **ui-delete-aux-id**, you must call **ui-start-transaction** to designate which UID record is being modified.
ui-insert-local-id	Insert a record into the *ui_local_id* table, giving the specified member a new local ID and system record.
ui-insert-person	Insert a new person record into the database using a demographic list. You must call ui-start-transaction before inserting a person record.
ui-local-id-merge	Merge two member profiles based on local IDs in a specific system.
ui-local-id-status	Return the status of a local ID record.
ui-lookup	Find a person's local ID in a specified system using that person's local ID in another system as search criteria.
ui-lookup-address-id	Search for the address ID of an existing address record based on the corresponding UID and address type.
ui-lookup-local-id	Look up a local ID associated with the specified system based on a member's local ID in another system. This API searches by local ID status.
ui-merge	Merge two individuals' records together using either their local IDs or their UIDs.
ui-poll	Define the function that is called to process messages being transmitted from the database to external systems through e*Gate.
ui-poll-neg-ack	Send a negative acknowledgment to the polling e*Way to verify that an Event was not received successfully.

Use this Monk API …	to perform this action …
ui-poll-pos-ack	Send a positive acknowledgment to the polling e*Way to verify that an Event was received successfully.
ui-poll-startup	Invoke setup and specify instance-specific function loads.
ui-process-address	Perform an address update or insert, depending on whether an address record already exists for the given address type and UID.
ui-process-person	Process messages coming into the database. This function is actually composed of several Monk APIs. You can customize this function to process records in the manner that best suits your processing requirements.
ui-process-phone	Perform a telephone number update or insert, depending on whether a telephone record already exists for the given address type and UID.
ui-rollback-transaction	Roll back the transaction in the database and reset the transaction structure.
ui-search-close	Close the search cursor, and de-allocate the memory.
ui-search-get-exact-match-score	Return the weight of an exact match to a new record if one exists in the database. This function checks to see if 1EXACTMTCH is enabled before returning the weight. Before calling **ui-search-get-exact-match-score**, you need to call **ui-search-open**.
ui-search-get-exact-match-uid	Return the UID of the record that is an exact match of a new record if an exact match exists in the database. Before calling **ui-search-get-exact-match-uid**, you need to call **ui-search-open**.
ui-search-insert-duplicate	Allow the records in the search cursor to be added to the *ui_duplic* table.
ui-search-local-id	Search for local IDs in a specific system based on a member's UID and the status of the local IDs.
ui-search-open	Open a cursor of weighted records returned from a search that is based on the demographic query list. This function returns the number of records in the result set.
ui-set-dup-threshold	Specify the minimum matching probability weight at which two records are considered potential duplicates of each other.

Use this Monk API …	to perform this action …
ui-set-match-threshold	Specify the minimum matching probability weight at which two records will be automatically merged.
ui-set-queue-id	Change the status of a queued message if the dequeued message is not sent successfully (as determined by a nack event in the polling e*Way).
ui-start-transaction	Start a transaction for a specific UID (if the UID is left blank, then the next available UID is assigned). A transaction is only initiated if a database insert, update, or delete is performed.
ui-stdver-conn-estab	Establish a connection to external applications.
ui-stdver-conn-shutdown	Request that the interface disconnect from the external application in preparation for a suspend/reload cycle.
ui-stdver-conn-ver	Verify whether external application connection has been established.
ui-stdver-data-exchg-stub	Create a placeholder for the function entry point for sending an Event from the external application to e*Gate. When the interface is configured as an outbound only connection, this function should not be called.
ui-stdver-init	Begin the initialization process for an e*Way and load all of the monk extension library files that are accessed by the other e*Way functions.
ui-stdver-neg-ack	Send a negative acknowledgment to the sending application to verify that an Event was not received successfully.
ui-stdver-pos-ack	Send a positive acknowledgment to the sending application to verify that an Event was received successfully.
ui-stdver-proc-outgoing	Send a received message (Event) from e*Gate to an external application.
ui-stdver-proc-outgoing-stub	Create a place holder for the function entry point for sending an Event received from e*Gate to the external application. If an interface is configured as an inbound only connection, this function should not be used.
ui-stdver-shutdown	Request that the connection to the external application shutdown.
ui-stdver-startup	Invoke setup and specify instance-specific function loads.

Use this Monk API …	to perform this action …
ui-unmerge	Unmerge two individuals' records that were previously merged. This is based on the active record's UID.
ui-update-address	Update an existing address in the database. A transaction must be started before calling **ui-update-address**.
ui-update-aux-id	Update a non-unique ID given the ID type and the new and old IDs. Before calling **ui-update-aux-id**, you must call **ui-start-transaction** to designate which UID record is being modified.
ui-update-person	Update a record using a demographic list. You must call ui-start-transaction before updating a person record.

Which Monk API Should I Use?

If you know what you want to do, but you are not sure which e*Index Monk API to use, refer to Table 4-2 on the following pages to look up a task that falls into one of the following categories:

- Connectivity and Configuration
- Performing Searches
- Displaying Error Messages
- Processing Data
- Processing Events
- Creating Date/Time Stamps
- e*Way Initialization

Table 4-2: e*Index Monk APIs by Functionality

To perform this action …	use this API …
Connectivity and Configuration	
Commit a transaction to the database and reset the transaction structure.	**ui-commit-transaction**
Read the parameters defined in the *ui_control* table to configure certain system attributes, such as local ID length, UID format, and so on.	**ui-config**
Establish a connection handle to the database server.	**make-connection-handle**

To perform this action …	use this API …
Roll back the transaction in the database and reset the transaction structure.	**ui-rollback-transaction**
Start a transaction for a specific UID (if the UID is left blank, then the next available UID is assigned). A transaction is only initiated if a database insert, update, or delete is performed.	**ui-start-transaction**
Specify the minimum matching probability weight at which two records are considered potential duplicates of each other.	**ui-set-dup-threshold**
Specify the minimum matching probability weight at which two records will be automatically merged.	**ui-set-match-threshold**
Performing Searches	
Close the address search cursor, and de-allocate the memory.	**ui-address-search-close**
Return the next address record from the address search cursor, and increment the search cursor position. You must call **ui-address-search-open** before calling **ui-address-search-next**.	**ui-address-search-next**
Search for existing address records based on an address list and open a cursor of weighted records returned from the search. This function returns the number of records in the result set.	**ui-address-search-open**
Searches for a specific non-unique ID type for a member given the member's UID, the ID type, and the identification code.	**ui-exists-aux-id**
Retrieve a person's alias records based on that person's UID.	**ui-get-alias**
Search for local IDs in a specific system (and their status) based on a member's UID. The status of the local IDs is ignored.	**ui-get-all-local-id**
Retrieve non-unique IDs given the type of ID and the member's UID.	**ui-get-aux-id**
Retrieve all local ID and system pairs associated with an individual based on their UID.	**ui-get-id-system**
Retrieve the member's local ID in a specific system given the system code and UID.	**ui-get-local-id**
Retrieve a person's demographic record based on that person's UID.	**ui-get-person**
Find a person's UID using that person's system and local ID as search criteria.	**ui-get-uid**

To perform this action …	use this API …
Retrieve a person's VIP flag based on that person's local ID in the specified system.	**ui-get-vip**
Search for the address ID of an existing address record based on the corresponding UID and address type.	**ui-lookup-address-id**
Find a person's local ID in a specified system using that person's local ID at another system as search criteria.	**ui-lookup**
Look up a local ID associated with the specified system based on a member's local ID in another system. This API searches by local ID status.	**ui-lookup-local-id**
Close the search cursor, and de-allocate the memory.	**ui-search-close**
Return the weight of an exact match to a new record if one exists in the database. This function checks to see if 1EXACTMTCH is enabled before returning the weight. Before calling **ui-search-get-exact-match-score**, you need to call **ui-search-open**.	**ui-search-get-exact-match-score**
Return the UID of the record that is an exact match of a new record if an exact match exists in the database. Before calling **ui-search-get-exact-match-uid**, you need to call **ui-search-open**.	**ui-search-get-exact-match-uid**
Allow the records in the search cursor to be added to the *ui_duplic* table.	**ui-search-insert-duplicate**
Search for local IDs in a specific system based on a member's UID and the status of the local IDs.	**ui-search-local-id**
Open a cursor of weighted records returned from a search that is based on the demographic query list. This function returns the number of records in the result set.	**ui-search-open**
Displaying Error Messages	
Display any database error messages after an API has processed.	**db-get-error-str**
Display e*Index error messages after an API has been called.	**ui-get-error-string**

To perform this action …	use this API …
Processing Data	
Deactivate an active UID record given the UID of the record to deactivate.	**ui-deactivate-person**
Deactivate an active local ID given the local ID number, the associated system, and a UID.	**ui-deactivate-local-id**
Delete an existing address record from the database. A transaction must be started before calling **ui-delete-address**.	**ui-delete-address**
Delete a non-unique ID given the ID type and the ID code. Before calling **ui-delete-aux-id**, you must call **ui-start-transaction** to designate which UID record is being modified.	**ui-delete-aux-id**
Remove a record's existing potential duplicate entries from ui_duplic during a person update so potential duplicates can re-evaluated for that record.	**ui-delete-unresolved-duplicates**
Check to see if the ASSMTCH control is enabled in the Administrator. If ASSMTCH is enabled, all assumed matches made by the application are written to the *ui_asssumed_match* table.	**ui-get-assumed-match-enabled**
Check to see if a demographic record was changed as a result of the previous actions against the database.	**ui-get-demographic-changed**
Check to see if the DUPCHK control key is enabled in the Administrator. If DUPCHK is enabled, potential duplicates for a specific record are re-evaluated after that record is updated.	**ui-get-dupchk-enabled**
Insert a new address into the database. A transaction must be started before calling **ui-insert-address**.	**ui-insert-address**
Insert alias information into the *ui_alias* table based on that person's UID.	**ui-insert-alias**
Insert an assumed match record by assuming the first record in the search cursor (the one with the highest matching weight) is the assumed match of the incoming record. You can only call this function after ui-search-open.	**ui-insert-assumed-match**
Insert a non-unique ID and specific ID type. Before calling ui-delete-aux-id, you must call ui-start-transaction to designate which UID record is being modified.	**ui-insert-aux-id**

To perform this action …	use this API …
Insert a record into the *ui_local_id* table, giving the specified person a new local ID and system record.	**ui-insert-local-id**
Insert a new person record into the database using a demographic list. You must call ui-start-transaction before inserting a person record.	**ui-insert-person**
Merge two member profiles based on local IDs in a specific system.	**ui-local-id-merge**
Return the status of a local ID record	**ui-local-id-status**
Merge two individuals' records together using either their local IDs or their UIDs.	**ui-merge**
Perform an address update or insert, depending on whether an address record already exists for the given address type and UID.	**ui-process-address**
Process messages coming into the database. This function is actually composed of several Monk APIs. You can customize this function to process records in the manner that best suits your processing requirements.	**ui-process-person**
Perform a telephone number update or insert, depending on whether a telephone record already exists for the given address type and UID.	**ui-process-phone**
Unmerge two individuals' records that were previously merged. This is based on the active record's UID.	**ui-unmerge**
Update a record using a demographic list. You must call ui-start-transaction before updating a person record.	**ui-update-person**
Update an existing address in the database. A transaction must be started before calling **ui-update-address**.	**ui-update-address**
Update a non-unique ID given the ID type and the new and old IDs. Before calling **ui-update-aux-id**, you must call **ui-start-transaction** to designate which UID record is being modified.	**ui-update-aux-id**
Processing Outgoing Events	
Retrieve and remove an active Event from the *ui_msg_detail* table so the polling e*Way can send the Event to e*Gate.	**ui-dequeue**
Remove a message from the outgoing queue once it has been successfully dequeued and sent.	**ui-delete-queue-msg**

To perform this action …	use this API …
Change the status of a queued message if the dequeued message is not sent successfully (as determined by a nack event in the polling e*Way).	**ui-set-queue-id**
Creating Date/Time Stamps	
Retrieve the date and time on the database server in the following format: YYYY/MM/DD hh:mm:ss.	**ui-get-db-date-time**
Retrieve the time that the transaction that is currently in progress began.	**ui-get-transaction-date-time**
e*Way Initialization	
Define the function that is called to process messages being transmitted from the database to external systems through e*Gate.	**ui-poll**
Send a negative acknowledgment to the polling e*Way to verify that an Event was not received successfully.	**ui-poll-neg-ack**
Send a positive acknowledgment to the polling e*Way to verify that an Event was received successfully.	**ui-poll-pos-ack**
Invoke setup and specify instance-specific function loads.	**ui-poll-startup**
Begin the initialization process for an e*Way and load all of the monk extension library files that are accessed by the other e*Way functions.	**ui-stdver-init**
Invoke setup and specify instance-specific function loads.	**ui-stdver-startup**
Establish a connection to external applications.	**ui-stdver-conn-estab**
Verify whether external application connection has been established.	**ui-stdver-conn-ver**
Request that the interface disconnect from the external application in preparation for a suspend/reload cycle.	**ui-stdver-conn-shutdown**
Send a positive acknowledgment to the sending application to verify that an Event was received successfully.	**ui-stdver-pos-ack**
Send a negative acknowledgment to the sending application to verify that an Event was not received successfully.	**ui-stdver-neg-ack**
Request that the connection to the external application shutdown.	**ui-stdver-shutdown**

To perform this action …	use this API …
Send a received message (Event) from e*Gate to an external application.	**ui-stdver-proc-outgoing**
Create a place holder for the function entry point for sending an Event received from e*Gate to the external application. If an interface is configured as an inbound only connection, this function should not be used.	**ui-stdver-proc-outgoing-stub**
Create a placeholder for the function entry point for sending an Event from the external application to e*Gate. When the interface is configured as an outbound only connection, this function should not be called.	**ui-stdver-data-exchg-stub**

For More Information

Other SeeBeyond publications may help you to learn how to perform tasks associated with creating Monk API scripts.

To learn more about …	See …
The Monk programming language	Your *Monk Developer's Reference*
e*Index Control Keys	Your *e*Index Administrator User's Guide*
e*Index Functions	Your *e*Index Global Identifier User's Guide*
Database e*Way APIs that you can use in your Monk scripts for e*Index	Your *user's guide for the database e*Way you are using (Oracle, Sybase, or ODBC)*
Configuring e*Ways	"Working with e*Ways" in your *e*Gate Integrator User's Guide*

e*Index Monk API Descriptions

Overview

This section of the chapter lists all of the Monk APIs that are included in the e*Index Monk library. Descriptions, syntax, parameters, return values, and examples for each API are provided.

db-get-error-str

The **db-get-error-str** function returns a database-related error message if a Monk API returns a value of false or MONK_EXCEPTION. Call this function after any API for which you want to view database error messages. This API is defined in the Monk library for the Database e*Way. For more information about this function, see the user's guide for the Database e*Way you are using.

Syntax

```
(db-get-error-str connection-handle)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database

Return Value

The **db-get-error-str** API returns one of the following values:

This value is returned …	if this occurs …
An error message	The error message for the specified API is retrieved successfully.

Example

In the following example, database and e*Index-related error messages, if any, are displayed for the previously called e*Index Monk API **ui-process-person**, where **connection-handle** is the connection handle defined in *ui-stdver-eway-funcs.monk*.

```
...
(set! demo (get-demographics ~input%eiEvent.REC[O]))
(set! trans (get-transact ~input%eiEvent))
(set! local-id (get ~input%eiEvent.REC[O].ID.local_id.LID.id))
(set! system (get ~input%eiEvent.REC[O].ID.local_id.LID.system))
(set! uid (ui-process-person connection-handle local-id system demo
          trans))
   (if uid
    (begin
     ...
    )
    (begin
     (let ((err_msg (db-get-error-str connection-handle))
                   (ui_err (ui-get-error-string)))
      (display (string-append "Rejecting!\n"))
      (display (string-append err_msg "\n" ui_err "\n"))
...
```

make-connection-handle

The **make-connection-handle** function allocates memory space to hold the connection handle. The connection handle must be defined before connecting to a database. When you start your e*Ways, the file *ui-stdver-eway-funcs.monk*, which defines the connection handle for you, is called. This function is defined in the Database e*Way Monk library.

Syntax

```
(make-connection-handle)
```

Parameters

Parameter	Type	Description
None		

Return Value

The **make-connection-handle** API returns one of the following values:

This value is returned …	if this occurs …
A connection handle	The connection handle was made successfully.
#f	The connection handle was not made successfully. Use the **db-get-error-str** API to retrieve the corresponding error message.

Example

The following example creates a connection handle named **connection-handle** that can now be passed through other statements. This is excerpted from the file *ui-stdver-eway-funcs.monk*.

```
(define connection-handle 0)
(set! connection-handle (make-connection-handle))
(if (connection-handle? connection-handle)
 (begin
  (set! result "FAILURE")
  (display "Failed to create connection handle.")
  (event-send "ALERTCAT_OPERATIONAL" "ALERTSUBCAT_UNUSABLE"
   "ALERTINFO_FATAL" "0" "database connection handle creation error"
   "Failed to create database connection handle" 0 (list))
 )
)
```

ui-address-search-close

The **ui-address-search-close** function closes the address search cursor, and de-allocates the memory. Use this function at the end of each **ui-address-search-open** function.

Syntax

```
(ui-address-search-close)
```

Parameters

Parameter	Type	Description
None		

Return Value

The **ui-address-search-close** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The address search cursor was closed and the memory was de-allocated successfully.
MONK_EXCEPTION	The address search cursor was not closed or the memory was not de-allocated successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

ui-address-search-next

The **ui-address-search-next** returns the next address record from the address search cursor, and increments the search cursor position by one. You must call **ui-address-search-open** to open the search cursor before calling **ui-address-search-next**. The search cursor must not be at the end of its record set or this function returns an exception.

Syntax

```
(ui-address-search-next)
```

Parameters

Parameter	Type	Description
None		

Return Value

The **ui-address-search-next** API returns one of the following values:

This value is returned …	if this occurs …
A Monk list of the address information	The function retrieves the next address record in the search cursor successfully.
MONK_EXCEPTION	The function does not retrieve the next record in the search cursor successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

ui-address-search-open

The **ui-address-search-open** function opens a cursor of weighted address records returned from a search that is based on the address information list. This function returns the number of records in the result set. The search cursor remains open until **ui-address-search-close** is called.

Syntax

```
(ui-address-search-open connection-handle address)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
address	list	A Monk list containing address information about the addresses for which you are searching.

Return Value

The **ui-address-search-open** API returns one of the following values:

This value is returned …	if this occurs …
The number of resulting records	The search is performed successfully, and the number of matching records is counted.
#f	The search is performed successfully, and there are no matching records in the search cursor.
MONK_EXCEPTION	The search is not performed successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

ui-commit-transaction

The **ui-commit-transaction** function commits the current transaction to the database, and resets the transaction structure. This API can only be called after a transaction has been initiated by **ui-start-transaction** or by **ui-merge**. In the default schema, **ui-commit-transaction** is called after each record is processed.

Syntax

```
(ui-commit-transaction connection-handle)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.

Return Value

The **ui-commit-transaction** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The transaction was successfully committed to the database.
MONK_EXCEPTION	The transaction was not successfully committed to the database. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The following example calls **ui-process-person** to process an incoming Event, and then commits the changes to the database after the Event is processed. In the default configuration, **ui-start-transaction** is called in **ui-process-person**. This example is excerpted from the file *uidb.dsc*.

```
...
(set! demo (get-demographics ~input%eiEvent.REC[O]))
(set! trans (get-transact ~input%eiEvent))
(set! local-id (get ~input%eiEvent.REC[O].ID.local_id.LID.id))
(set! system (get ~input%eiEvent.REC[O].ID.local_id.LID.system))
(set! uid (ui-process-person connection-handle local-id system demo
           trans))

   ... ;processing person record

(event-send-to-egate (message->string output))
(ui-commit-transaction connection-handle)
...
```

ui-config

The **ui-config** function reads the values defined in the *ui_control* table to configure system parameters, such as local ID length, search parameters, UID format, and so on. This function should be called after **db-login**.

For more information about the control keys that affect API processing, see "How do Control Keys Affect APIs?" earlier in this chapter.

For more information on the parameters defined in the *ui_control* table, see "Configuring e*Index" in your *e*Index Administrator User's Guide*.

Syntax

```
(ui-config connection-handle)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.

Return Value

The **ui-config** API returns one of the following values:

This value is returned …	if this occurs …
#t	The configuration was completed successfully.
MONK_EXCEPTION	The configuration was not completed successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The following example attempts to read the configuration defined in the *ui_control* table. **ui-config** is called within a try-catch block. Use the catch statement to define error handling for instances when the call to **ui-config** fails. This example is excerpted from *ui-stdver-eway-funcs.monk*.

```
...
   (display "The result of ui-config is ")
   (try (ui-config connection-handle)
     (begin
       (display "OK\n")
       (set! result "UP")
     )
     (catch
       ...
```

ui-deactivate-person

The **ui-deactivate-person** function deactivates an active UID record in the e*Index database by setting its status to **D** in the *ui_person* table. You must call **ui-start-transaction** before you can call **ui-deactivate-person**.

Syntax

```
(ui-deactivate-person connection-handle uid)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
uid	string	The UID of the record you want to deactivate.

Return Value

The **ui-deactivate-person** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The specified UID record was successfully deactivated.
MONK_EXCEPTION	The specified UID record was not successfully deactivated. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The following example reads the UID from the incoming message, and uses that UID to retrieve a record to deactivate.

```
...
 (begin
    (let (uid (get ~input%eiEvent.REC[O].ID.eid_1)))
    (ui-start-transaction connection-handle uid trans)
    (display (ui-deactivate-person connection-handle uid))
...
```

ui-deactivate-local-id

The **ui-deactivate-local-id** function deactivates an active local ID in the e*Index database by setting its status to **D** in the *ui_local_id* table. You must call **ui-start-transaction** before you can call **ui-deactivate-local-id**.

Syntax

```
(ui-deactivate-local-id connection-handle uid system local-id)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
uid	string	The UID of the member associated with the local ID and system pair you want to deactivate.
system	string	The code of the system associated with the local ID you want to deactivate.
local-id	string	The identification number you want to deactivate.

Return Value

The **ui-deactivate-local-id** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The specified local ID was successfully deactivated.
MONK_EXCEPTION	The specified local ID was not successfully deactivated. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

ui-delete-address

The **ui-delete-address** function deletes a member address from the *ui_address* table given the unique identification code of the address in the database. You can obtain the unique ID of the address by calling **ui-lookup-address-id**. Before calling **ui-delete-address**, you must call **ui-start-transaction**.

Syntax

```
(ui-delete-address connection-handle address-id)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
address-id	string	The unique ID code of the address to be deleted. The unique ID is assigned by e*Index.

Return Value

The **ui-delete-address** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The specified address record was successfully removed.
MONK_EXCEPTION	The specified address record was not successfully removed. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

ui-delete-aux-id

The **ui-delete-aux-id** function deletes a member's non-unique ID from the *ui_aux_def_id* table given the ID type and the member's UID. If the non-unique ID does not exist, this function does nothing. Before calling **ui-delete-aux-id**, you must call **ui-start-transaction** to designate the UID of the record being modified.

Syntax

```
(ui-delete-aux-id connection-handle id-type id)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
id-type	string	The type of non-unique ID to be deleted from the member's record.
id	string	The non-unique ID number that you want to delete.

Return Value

The **ui-delete-aux-id** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The member's non-unique ID was successfully removed.
MONK_EXCEPTION	The non-unique ID was not successfully removed. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

ui-delete-queue-msg

Use the **ui-delete-queue-msg** function to remove a message from the outgoing queue (the *ui_msg_detail* table) once it has been successfully dequeued and sent. The sample schema for the polling e*Way places the call to **ui-delete-queue-msg** in the ack event, ensuring that if a message is successfully sent, it is removed from the queue.

Syntax

```
(ui-delete-queue-msg connection-handle msg-id)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
msg-id	string	The identification code of the outgoing message.

Return Value

The **ui-delete-queue-msg** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The message was successfully removed from the outgoing queue.
MONK_EXCEPTION	The message was not successfully removed from the outgoing queue. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The following example from the **ui-poll-pos-ack** command in *ui-stdver-eway-funcs.monk* defines the variable **msg-id** as a data element in the outgoing message. It then deletes the specified message from the queue because the message was successfully processed.

```
...
 (display "[++] Executing e*Way positive acknowledgement function.")
 (if ($event-parse input message-string)
   (begin
    (set! msg-id (get ~input%eiEvent.EVNT.EVN.msg_id))
    (display (format "Deleting message ID [%s] from the queue\n" msg-id))
    (ui-delete-queue-msg connection-handle msg-id)
   )
 )
...
```

ui-delete-unresolved-duplicates

Use the **ui-delete-unresolved-duplicates** function to remove a record's existing potential duplicate entries from the *ui_duplic* database table during a person update. You may want to do this in order to re-evaluate potential duplicates for that record after the update. In order to re-evaluate a record's potential duplicate after deleting its potential duplicate entries, you need to call **ui-search-insert-duplicate** (see the example on the following page).

Syntax

```
(ui-delete-unresolved-duplicates connection-handle uid)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
uid	string	The unique global identifier assigned to the member by e*Index.

Return Value

The **ui-delete-unresolved-duplicates** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The duplicate record pairs were removed from *ui_duplic* successfully.
MONK_EXCEPTION	The duplicate record pairs were not successfully removed from *ui_duplic*. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The following example from the default file *ui-process-person.monk*, defines the variable **uid**, which holds the UID of the member found by **ui-get-uid**. A transaction is then started (by **ui-start-transaction**) using that UID and the incoming transaction information, and the member's information is updated (by the call to **ui-update-person**). It then checks to see if the DUPCHECK control key is enabled and whether the member's demographic information changed. If both of these are true, then the unresolved potential duplicate pair entries associated with that member are removed from *ui_duplic*, and a new set of potential duplicate records are evaluated and placed in *ui_duplic*.

---

*Note: In the default configuration, the variables **demo** and **trans** are defined in* uidb.dsc.

```
...
 (begin
  (set! demo (get-demographics ~input%eiEvent))
  (set! trans (get-transact ~input%eiEvent))
  (set! local-id (get ~input%eiEvent.REC[0].ID.local_id.LID.id))
  (set! system (get ~input%eiEvent.REC[0].ID.local_id.LID.system))
  (set! uid (ui-get-uid connection-handle system local-id))
  (if uid
    (begin
      (ui-start-transaction connection-handle uid trans)
      (ui-update-person connection-handle demo)
      (if (ui-get-dupchk-enabled)
        (if (string=? (ui-get-demographic-changed) "keychanged")
           (begin
            (ui-delete-unresolved-duplicates connection-handle uid)
                (set! search-count (ui-search-open connection-handle
demo))
                 (if search-count
                   (ui-search-insert-duplicate connection-handle
                        "POTENTIAL DUPLICATE" 1 search-count)
                   (ui-search-close)
                 )
            )
         )
       )
     )
...
```

ui-dequeue

The **ui-dequeue** function retrieves an active Event from the outbound queue table (*ui_msg_detail*), enabling the polling e*Way to pick up the Event and route it through e*Gate. This function removes active Events one at a time. Use **ui-dequeue** in the Monk script for your polling e*Way.

Syntax

```
(ui-dequeue connection-handle)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.

Return Value

The **ui-dequeue** API returns one of the following values:

This value is returned …	if this occurs …
An Event	The Event was removed from the queue successfully and an active Event exists in the queue.
An empty Event	The dequeue was successful, but an active Event does not exist in the queue.
MONK_EXCEPTION	The Event was not removed from the queue successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The following example from the **ui-poll** command in *ui-stdver-eway-funcs.monk* sets the variable **pollmsg** to be the result of the **ui-dequeue** call. If **ui-dequeue** is successful, a message is retrieved and sent to e*Gate. Use the **catch** statement for exception and error handling mechanisms.

```
...
   (let ( (pollmsg ""))
     (try
      (display "[++] Executing e*Index poll function.\n")
       (set! pollmsg (ui-dequeue connection-handle))
     )
      (catch
         ... ; error handling statements
       )
     (display (format "Returning: [%s]\n" pollmsg))
     pollmsg
   )
...
```

ui-exists-aux-id

The **ui-exists-aux-id** function searches for a member's non-unique ID from the *ui_aux_def_id* table given the ID type, ID number, and the member's UID. This function returns #t if the specified ID and type are found.

Syntax

```
(ui-exists-aux-id connection-handle uid id-type id)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
uid	string	The UID of the member associated with the non-unique ID for which you are searching.
id-type	string	The non-unique ID type associated with the specified ID number.
id	string	The non-unique ID for which you are searching.

Return Value

The **ui-exists-aux-id** API returns one of the following values:

This value is returned …	if this occurs …
#t	The specified non-unique ID was found in the database.
#f	The search was performed successfully but the specified non-unique ID was not found.
MONK_EXCEPTION	The non-unique ID was not successfully removed. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

In the example below, excerpted from the default file *uidb.dsc*, **ui-process-person (**as defined by the default *ui_process_person* file) is called to begin processing the demographic information of an incoming Event. It then calls **ui-exists-aux-id** to check the database table *ui_aux_def_id* to see if the non-unique ID type and number are already associated with the record being updated. If the pair exists in the member record, no changes are made. If the pair does not exist in the member record, then a new ID and type pair is inserted into *ui_aux_def_id*.

```
...
(begin
 (set! demo (get-demographics ~input%eiEvent))
 (set! trans (get-transact ~input%eiEvent))
 (set! local-id (get ~input%eiEvent.REC[O].ID.local_id.LID.id))
 (set! system (get ~input%eiEvent.REC[O].ID.local_id.LID.system))
 (set! uid (ui-process-person connection-handle local-id system demo trans))
  ...;  processing person information

  do ((j O (+ j 1))) ((>= j (count ~input%eiEvent.REC[O].ID.non_unique_id)))
  (display (string-append "AUX_ID" "[" (number->string j) "] "))
  (let ((id-type (get ~input%eiEvent.REC[O].ID.non_unique_id[<j>].NID.type))
        (id (get ~input%eiEvent.REC[O].ID.non_unique_id[<j>].NID.id))
        (id-orig #f)
      )
     (begin
      (display (string-append id-type " " id)) (newline)
      (if (not (ui-exists-aux-id connection-handle uid id-type id))
        (begin
          (display "Inserting new aux-id...")(newline)
          (ui-insert-aux-id connection-handle id-type id)
        )
      )
     )
  )
)
...
```

ui-get-alias

The **ui-get-alias** function checks the *ui_alias* table and retrieves a person's alias records based on the person's UID. Each field in the returned list is delimited by ",", and each record is delimited by "|". Use this function to check a member's alias names to determine whether an alias name needs to be updated or inserted into the database.

Syntax

```
(ui-get-alias connection-handle uid)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
uid	string	The UID of the person whose alias records you want to retrieve.

Return Value

The **ui-get-alias** API returns one of the following values:

This value is returned …	if this occurs …
A list of alias records	The alias information for the specified person was retrieved successfully.
MONK_EXCEPTION	The alias information for the specified person was not retrieved successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The following example finds a member's UID using the **ui-get-uid** API, and stores the UID in the Monk variable **uid**. It then searches for a member's alias names using the Monk variable **uid** to identify the member. A list of alias records for the member is returned, and is stored in the Monk variable **alias**.

```
...
(set! local-id (get ~input%eiEvent.REC[0].ID.local_id.LID.id))
(set! system (get ~input%eiEvent.REC[0].ID.local_id.LID.system))
(set! uid (ui-get-uid connection-handle system local-id))

(set! alias (ui-get-alias connection-handle uid))
(display alias)(newline)
...
```

ui-get-all-local-id

The **ui-get-all-local-id** function retrieves all of a member's local IDs in a specific system given the system code and member's UID. This function returns local IDs of any status (active, merged, or deactivated), along with the status of each local ID.

Syntax

```
(ui-get-all-local-id connection-handle uid system)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
uid	string	The UID of the member for which you want to retrieve the local ID.
system	string	The code for the system with which the local ID you want to retrieve is associated.

Return Value

The **ui-get-all-local-id** API returns one of the following values:

This value is returned …	if this occurs …
A vector containing local ID and status pairs	The member's local ID and status pairs for the specified system were retrieved successfully.
#f	No local IDs for the specified system exist for the specified member (this indicates a flaw in data integrity).
MONK_EXCEPTION	The member's local IDs were not retrieved successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The example below searches for person records in the database that closely match the incoming Event, and, if an exact match is found, retrieves the UID of the exact match. It then calls **ui-get-all-local-id** to check the entries in the *ui_local_id* table that are associated with the exact match record. If local ID and system pairs are found where the system matches the system in the incoming Event, the sample scrolls through the local IDs using the **ui-get-next-element** and **ui-has-next-element** functions.

```
(set! demo (get-demographics ~input%eiEvent))
(set! trans (get-transact ~input%eiEvent))
(set! system (get ~input%eiEvent.REC[0].ID.local_id.LID.system))
(begin
  (set! search-count (ui-search-open connection-handle demo))
  (if search-count
    (set! uid (ui-search-get-exact-match-uid))
  )
  (if uid
   (begin

     (set! lids (ui-get-all-local-id connection-handle uid system))

     (if lids
        (begin
          (do
            ((i 0 (+ i 1)))
            ((not (ui-has-next-element lids)))
            (display (format "%d: %a\n" i (ui-get-next-element lids)))
            )
        )
     )
   (display "No LIDs\n")
  ....
```

ui-get-assumed-match-enabled

The **ui-get-assumed-match-enabled** function checks the *ui_control* table to see if the ASSMTCH control key is enabled in e*Index Administrator. In the default file *ui-process-person.monk*, if ASSMTCH is enabled all assumed matches made by the application are written to the *ui_asssumed_match* table. Use this function to check the control key prior to inserting a record into *ui_assumed_match*.

Syntax

```
(ui-get-assumed-match-enabled)
```

Parameters

Parameter	Type	Description
None		

Return Value

The **ui-get-assumed-match-enabled** API returns one of the following values:

This value is returned …	if this occurs …
#t	The assumed match control key was checked successfully and the control key is enabled.
#f	The assumed match control key was checked successfully and it is not enabled
MONK_EXCEPTION	The control key was not checked successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

This sample, taken from the default *ui-process-person.monk* file, finds a set of records in the database that could potentially match the incoming record. If an exact match record is found, then the record is updated and an entry is written to the *ui_assumed_match* table to let you know that an automatic merge occurred.

*Note: In the default configuration, the variables **demo** and **trans** are defined in uidb.dsc.*

```
...
(set! demo (get-demographics ~input%eiEvent))
(set! trans (get-transact ~input%eiEvent))
...
(set! search-count (ui-search-open connection-handle demo))
(if search-count
 (set! uid (ui-search-get-exact-match-uid))
)
 (if uid
   (begin
       ...  ;checking for same facility match
   )
   (begin
     (ui-start-transaction connection-handle uid trans)
     (ui-update-person connection-handle demo)
     (ui-insert-local-id connection-handle local-id system)
     (if (ui-get-assumed-match-enabled)
       (ui-insert-assumed-match connection-handle)
     )
     (if (> search-count 1)
       ... ;processing potential duplicates
     )
   )
 )
...
```

ui-get-aux-id

The **ui-get-aux-id** function checks the *ui_aux_id_def* table and retrieves any matching non-unique IDs for a member given the type of ID to retrieve and the member's UID. Use this function to check an existing member record in the database for non-unique IDs of a specific type. The return value of **ui-get-aux-id** can determine whether a non-unique ID record should be updated or unchanged, or if a new non-unique ID record should be inserted.

Syntax

```
(ui-get-aux-id connection-handle uid id-type)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
uid	string	The UID of the member for which you want to retrieve the non-unique ID.
id-type	string	The type of non-unique ID to be retrieved from the member's record.

Return Value

The **ui-get-aux-id** API returns one of the following values:

This value is returned …	if this occurs …
The non-unique IDs, delimited by commas	The member's non-unique IDs for the specified ID type were retrieved successfully.
#f	The member's non-unique ID for the specified ID type does not exist.
MONK_EXCEPTION	The control key was not checked successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The following example shows one way that non-unique IDs can be processed differently from the default Collaboration script, **uidb.dsc** (for more information about how non-unique IDs are processed by default, see the example for **ui-exists-aux-id** on page 4-38).   In this example, we assume that a person record should not have more than one non-unique ID of a given type.

In this example, **ui-process-person (**as defined by the default *ui_process_person* file) is called to begin processing the demographic information of an incoming Event. **ui-get-aux-id** checks the database table *ui_aux_def_id* to see if the non-unique ID type exists in the person record being updated. If the ID type exists in the record and the existing ID and incoming ID do not match, the existing ID is updated; if the IDs match, no changes are made. If the ID type does not exist in the record, then a new ID and type are inserted into *ui_aux_def_id*.

```
...
(begin
 (set! demo (get-demographics ~input%eiEvent))
 (set! trans (get-transact ~input%eiEvent))
 (set! local-id (get ~input%eiEvent.REC[0].ID.local_id.LID.id))
 (set! system (get ~input%eiEvent.REC[0].ID.local_id.LID.system))
 (set! uid (ui-process-person connection-handle local-id system demo
           trans))
  ...;  processing person information

 do ((j 0 (+ j 1))) ((>= j (count ~input%eiEvent.REC[0].ID.non_unique_id)))
  (display (string-append "AUX_ID" "[" (number->string j) "] "))
  (let ((id-type (get ~input%eiEvent.REC[0].ID.non_unique_id[<j>].NID.type))
        (id (get ~input%eiEvent.REC[0].ID.non_unique_id[<j>].NID.id))
        (id-orig #f)
      )
     (begin
      (display (string-append id-type " " id)) (newline)
      (set! id-orig (ui-get-aux-id connection-handle uid id-type))
      (if id-orig
        (begin
          (if (not (string=? id id-orig))
            (begin
              (display "Updating existing aux-id...")(newline)
              (ui-update-aux-id connection-handle id-type old-id new-id)
            )
            (begin
              (display "Existing aux-id does not require
update...")(newline)
            )
          )
        )
        (begin
          (display "Inserting new aux-id...")(newline)
          (ui-insert-aux-id connection-handle id-type id)
...
```

ui-get-db-date-time

The **ui-get-db-date-time** function retrieves the date and time on the database server in the following format: YYYY/MM/DD hh:mm:ss.

Syntax

```
(ui-get-db-date-time connection-handle)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.

Return Value

The **ui-get-db-date-time** API returns one of the following values:

This value is returned …	if this occurs …
A date/time stamp	The date and time were retrieved from the database server successfully.
MONK_EXCEPTION	The date and time were not retrieved from the database server successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The following example from *ui-custom.monk* uses the date and time of the server where the e*Index database resides to get the date and time of the Event in the Monk list **transaction_info** (for more information, see "What Monk Lists are Defined?" earlier in this chapter).

```
(define get-transact
  (lambda (msg)
    (list (get ~<msg>.EVNT.EVN.event_type_code) ;function
          (get ~<msg>.EVNT.EVN.assigning_system) ;system
          (get ~<msg>.EVNT.EVN.department) ; dept
          (get ~<msg>.EVNT.EVN.source) ; source
          (get ~<msg>.EVNT.EVN.terminal_id) ; term id
          (if (empty-string? ~<msg>.EVNT.EVN.user_id)
           (string-upcase DATABASE_SETUP_USER_NAME) ; default user id
           (get ~<msg>.EVNT.EVN.user_id)        ; user id
          )
          (ui-get-db-date-time connection-handle)
                  ; use database server date/time stamp
    )
  )
)
```

ui-get-demographic-changed

The **ui-get-demographic-changed** function checks the *ui_person* table to determine if there were any changes made to the key fields in a person's demographic information after **ui-update-person** was called. The result of **ui-get-demographic-changed** can be used to determine how the information will be processed. Key fields are those fields defined by the configurable query.

Syntax

```
(ui-get-demographic-changed)
```

Parameters

Parameter	Type	Description
None		

Return Value

The **ui-get-demographic-changed** API returns one of the following values:

This value is returned …	if this occurs …
A value of "unchanged"	The demographic information was verified successfully and there were no changes to an existing member record.
A value of "changed"	The demographic information was verified successfully and there were changes to an existing member record, but not to the key fields.
A value of "keychanged"	The demographic information was verified successfully and key demographic fields were changed. Key fields include last name, first name, middle name, date of birth, gender, and SSN.
A value of "new"	The demographic information was verified successfully and the Event was inserted as a new member record.
MONK_EXCEPTION	The demographic information was not verified successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The following example searches the *ui_local_id* table of the database for a record matching the local ID and system pair in the incoming Event. If a record is found, the associated member record is updated with the new demographic information from the incoming event. If the DUPCHECK control key is enabled, the record is checked for specific demographic changes resulting from the call to **ui-update-person**. If **ui-get-demographic-changed** returns a value of "**keychanged**" potential duplicates for the record are re-evaluated.

```
(set! demo (get-demographics ~input%eiEvent))
(set! trans (get-transact ~input%eiEvent))
...
 (begin
  (set! local-id (get ~input%eiEvent.REC[0].ID.local_id.LID.id))
  (set! system (get ~input%eiEvent.REC[0].ID.local_id.LID.system))
  (set! uid (ui-get-uid connection-handle system local-id))
  (if uid
    (begin
      (ui-start-transaction connection-handle uid trans)
      (ui-update-person connection-handle demo)
      (if (ui-get-dupchk-enabled)
        (if (string=? (ui-get-demographic-changed) "keychanged")
          (begin
             ...; processing potential duplicates
```

ui-get-dupchk-enabled

The **ui-get-dupchk-enabled** function checks the *ui-control* table to see if the DUPCHK control key is enabled in e*Index Administrator. DUPCHK should be enabled if you want to re-evaluate potential duplicates for a specific record after that record is updated. This function would normally be called after a call to **ui-update-person** and before deleting or re-inserting potential duplicates.

Syntax

```
(ui-get-dupchk-enabled)
```

Parameters

Parameter	Type	Description
None		

Return Value

The **ui-get-dupchk-enabled** API returns one of the following values:

This value is returned …	if this occurs …
#t	The duplicate checking control key was read successfully, and the control key is enabled.
#f	The duplicate checking control key was read successfully and it is not enabled.
MONK_EXCEPTION	The control key was not read successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

For an example of how **ui-get-dupchk-enabled** can be used, see the example for **ui-get-demographic-changed** on page 4-50.

ui-get-error-string

The **ui-get-error-string** function returns an e*Index-related error message if an e*Index Monk API returns a value of false or MONK_EXCEPTION. Call this function after any API for which you want to view e*Index error messages.

Syntax

```
(ui-get-error-string)
```

Parameters

Parameter	Type	Description
None		

Return Value

The **ui-get-error-string** API returns one of the following values:

This value is returned …	if this occurs …
An error message	The error message is successfully called when the specified Monk API returns a value of #f.

Example

In the following example, database and e*Index-related error messages, if any, are displayed for the previously called e*index Monk API **ui-process-person**, where **connection-handle** is the connection handle defined in *ui-stdver-eway-funcs.monk*.

```
...
(set! demo (get-demographics ~input%eiEvent.REC[O]))
(set! trans (get-transact ~input%eiEvent))
(set! local-id (get ~input%eiEvent.REC[O].ID.local_id.LID.id))
(set! system (get ~input%eiEvent.REC[O].ID.local_id.LID.system))
(set! uid (ui-process-person connection-handle local-id system demo
          trans))
   (if uid
    (begin
     ...
    )
    (begin
     (let ((err_msg (db-get-error-str connection-handle))
                 (ui_err (ui-get-error-string)))
      (display (string-append "Rejecting!\n"))
      (display (string-append err_msg "\n" ui_err "\n"))
...
```

ui-get-id-system

The **ui-get-id-system** function checks the *ui_local_id* table and retrieves all local ID and system pairs that are associated with the specified UID. The local ID and system in each pair are delimited by a comma, and each pair is also delimited by a comma. Use this function to check a member's local ID and facility pairs to determine whether a new local ID needs to be inserted into the database.

Syntax

```
(ui-get-id-system connection-handle u-id)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
u-id	string	The UID of the person whose local ID and system information you want to retrieve.

Return Value

The **ui-get-id-system** API returns one of the following values:

This value is returned …	if this occurs …
A list of local IDs and system pairs	The local ID and system information was retrieved successfully.
MONK_EXCEPTION	The local ID and system information for the specified person was not retrieved successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The following example finds a member's UID using the **ui-get-uid** API, and
stores the UID in the Monk variable **uid**. It then searches for a member's
local ID and system pairs using the Monk variable **uid** to identify the
member. A list of local ID and system pairs for the member is returned, and
is stored in the Monk variable **system**.

```
...
(set! local-id (get ~input%eiEvent.REC[0].ID.local_id.LID.id))
(set! system (get ~input%eiEvent.REC[0].ID.local_id.LID.system))
(set! uid (ui-get-uid connection-handle system local-id))
   (if uid
    (begin
      (set! id-system (ui-get-id-system connection-handle uid))
      (display system)(newline)
    )
   ...
```

ui-get-local-id

The **ui-get-local-id** function retrieves a member's local ID in a specific system given the system code and member's UID. This function only returns local IDs with a status of **A** (active).

*Note: This API only returns the first matching local ID. To retrieve a complete list of local IDs matching the criteria, use **ui-search-local-id** or **ui-get-all-local-id**.*

Syntax

```
(ui-get-local-id connection-handle uid system)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
uid	string	The UID of the member for which you want to retrieve the local ID.
system	string	The code for the system with which the local ID you want to retrieve is associated.

Return Value

The **ui-get-local-id** API returns one of the following values:

This value is returned …	if this occurs …
A local ID number	The member's local ID for the specified system was retrieved successfully.
#f	A local ID for the specified system does not exist for the specified member.
MONK_EXCEPTION	The member's local ID was not retrieved successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The example below searches for person records in the database that closely match the incoming Event, and, if an exact match is found, retrieves the UID of the exact match. It then calls **ui-get-local-id** to check for entries in the *ui_local_id* table that are associated with the exact match record. Depending on whether a local ID and system pair is found, either a new person record is inserted into the database (by **ui-insert-person**) or an existing record is updated (by **ui-update-person**).

```
(set! demo (get-demographics ~input%eiEvent))
(set! trans (get-transact ~input%eiEvent))
(set! system (get ~input%eiEvent.REC[O].ID.local_id.LID.system))
(begin
  (set! search-count (ui-search-open connection-handle demo))
  (if search-count
  (set! uid (ui-search-get-exact-match-uid))
  )
  (if uid
   (begin
     (if (ui-get-local-id connection-handle uid system)
       (begin
         (ui-start-transaction connection-handle "" trans)
         (set! uid (ui-insert-person connection-handle demo))
         (ui-insert-local-id connection-handle local-id system)
         (ui-search-insert-duplicate connection-handle "SAME SYSTEM MATCH" 1 1)
           (if (> search-count 1)
            (ui-search-insert-duplicate connection-handle "POTENTIAL DUPLICATE"
2
             search-count)
          )
       )
       (begin
         (ui-start-transaction connection-handle uid trans)
         (ui-update-person connection-handle demo)
         (ui-insert-local-id connection-handle local-id system)
         (if (ui-get-assumed-match-enabled)
           (ui-insert-assumed-match connection-handle)
         )
         ...
```

ui-get-person

The **ui-get-person** function retrieves a person's current demographic record based on the specified UID. The fields in the record are delimited by "|".

Syntax

```
(ui-get-person connection-handle uid)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
uid	string	The UID of the person whose demographic information you want to retrieve.

Return Value

The **ui-get-person** API returns one of the following values:

This value is returned …	if this occurs …
A demographic record	The demographic information for the specified person was retrieved successfully.
MONK_EXCEPTION	The demographic information for the specified person was not retrieved successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

*Note: The demographic record that is returned by **ui-get-person** is delimited by a pipe ("|").*

Example

The following example finds a member's UID using the **ui-get-uid** API, and stores the UID in the Monk variable **uid**. It then searches for the current demographic record of the member identified by the Monk variable **uid**. A demographic record for the member is returned, and is stored in the Monk variable **demo**. The output would be similar to

Smith|Joe|M|11234 Mission Way||Tarzana|CA| ...

with all fields delimited by "|".

```
...
(set! local-id (get ~input%eiEvent.REC[0].ID.local_id.LID.id))
(set! system (get ~input%eiEvent.REC[0].ID.local_id.LID.system))
(set! uid (ui-get-uid connection-handle system local-id))
   (if uid
    (begin
     (set! demo (ui-get-person connection-handle uid))
     (display demo)(newline)
    )
   ...
```

ui-get-transaction-date-time

The **ui-get-transaction-date-time** function retrieves the time that the transaction that is currently in progress began. This API can only be called after **ui-start-transaction**.

Syntax

```
(ui-get-transaction-date-time)
```

Parameters

Parameter	Type	Description
None		

Return Value

The **ui-get-transaction-date-time** API returns one of the following values:

This value is returned …	if this occurs …
A date/time stamp	The time that the transaction began was retrieved successfully.
MONK_EXCEPTION	The time that the transaction began was not retrieved successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The following example from *ui-custom.monk* uses the date and time that the current open transaction began to get the date and time of the Event in the Monk list **trans_info** (for more information, see "What Monk Lists are Defined?" earlier in this chapter).

```
(define get-transact
  (lambda (msg)
    (list (get ~<msg>.EVNT.EVN.event_type_code) ;function
          (get ~<msg>.EVNT.EVN.assigning_system) ;system
          (get ~<msg>.EVNT.EVN.department) ; dept
          (get ~<msg>.EVNT.EVN.source) ; source
          (get ~<msg>.EVNT.EVN.terminal_id) ; term id
          (if (empty-string? ~<msg>.EVNT.EVN.user_id)
           (string-upcase DATABASE_SETUP_USER_NAME) ; default user id
           (get ~<msg>.EVNT.EVN.user_id)        ; user id
          )
          (ui-get-transaction-date-time)
    )
  )
)
```

ui-get-uid

The **ui-get-uid** function looks up a person's UID by cross-referencing the person's local ID from a specific system (source-system) in the *ui_local_id* table. This API only retrieves records with a status of **A**, meaning only active records are returned.

Syntax

```
(get-uid connection-handle system local-id)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
system	string	A system for which the person's local ID is known.
local-id	string	The local ID of the person in the source system.

Return Value

The **get-uid** API returns one of the following values:

This value is returned …	if this occurs …
A UID	The person's UID was found in the *ui_local_id* table.
#f	There is no corresponding UID for the specified system and local ID pair in the e*Index database.
MONK_EXCEPTION	The person's UID was not found in the *ui_local_id* table. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

Use the **ui-get-uid** function to find a member's UID and store the UID in a variable for use with other Monk APIs. To see how **ui-get-uid** can be used with other APIs, see the example for **ui-delete-unresolved-duplicates** on page 4-34.

```
...
(set! local-id (get ~input%eiEvent.REC[0].ID.local_id.LID.id))
(set! system (get ~input%eiEvent.REC[0].ID.local_id.LID.system))
(set! uid (ui-get-uid connection-handle system local-id)
   (if uid
    (begin
    ...
```

ui-get-vip

The **ui-get-vip** function retrieves a person's VIP flag using a person's local ID in a specific system.

Syntax

```
(ui-get-vip connection-handle system local-id)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
system	string	The system associated with the local ID you specified.
local-id	string	The local ID of the person whose VIP flag you want to retrieve.

Return Value

The **ui-get-vip** API returns one of the following values:

This value is returned …	if this occurs …
The value in the VIP field	The VIP information for the specified person was retrieved successfully.
MONK_EXEPTION	The VIP information for the specified person was not retrieved successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The following example returns the VIP flag for the person whose local ID and system are specified in the incoming Event. The value of the VIP flag is saved in the Monk variable **vip**.

```
...
(set! local-id (get ~input%eiEvent.REC[O].ID.local_id.LID.id))
(set! system (get ~input%eiEvent.REC[O].ID.local_id.LID.system))
(set! vip (ui-get-vip connection-handle system local-id))
 (display vip)(newline)
...
```

ui-insert-address

The **ui-insert-address** function inserts address information into the *ui_alias* table for the specified person. The person whose information you are inserting is specified by their UID. A transaction must be started before calling **ui-insert-address**.

Syntax

```
(ui-insert-address connection-handle address)
```

Parameters

Parameters	Type	Description
connection-handle	connection handle	A handle to the database.
address	list	A Monk list containing the address information to insert into the *ui_address* table.

Return Value

The **ui-insert-address** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The address information was successfully inserted into the *ui_address* table.
MONK_EXCEPTION	The address information was not successfully inserted into the *ui_address* table. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

ui-insert-alias

The **ui-insert-alias** function inserts alias information into the *ui_alias* table for the specified person. The person whose information you are inserting is specified by their UID.

Syntax

```
(ui-insert-alias connection-handle alias)
```

Parameters

Parameters	Type	Description
connection-handle	connection handle	A handle to the database.
alias	list	A Monk list containing the alias information to insert into the *ui_alias* table.

Return Value

The **ui-insert-alias** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The alias information was successfully inserted into the *ui_alias* table.
MONK_EXCEPTION	The alias information was not successfully inserted into the *ui_alias* table. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The following example uses the connection handle **connection-handle** as defined in *ui-stdver-eway-funcs.monk*. It calls **ui-process-person** to process the demographic information from the incoming message. It then retrieves a list of alias information for the record and inserts the new alias information into the *ui_alias* table. The alias information is associated with the UID specified in **ui-process-person**.

```
...
(set! demo (get-demographics ~input%eiEvent.REC[O]))
(set! trans  (get-transact ~input%eiEvent))
(set! local-id (get ~input%eiEvent.REC[O].ID.local_id.LID.id))
(set! system (get ~input%eiEvent.REC[O].ID.local_id.LID.system))
(set! uid (ui-process-person connection-handle local-id system demo trans))


...

(do ((j O (+ j 1)))
    ((>= j (count ~input%eiEvent.REC[O].DEMO.person_alias)))
  (set! alias (get-alias ~input%eiEvent.REC[O] j))
  (display (string-append "ALIAS" "[" (number->string j) "]"))
  (display alias) (newline)
  (display "PROCESS ALIAS RESULT: " )
  (display (ui-insert-alias connection-handle alias))(newline)
...
```

ui-insert-assumed-match

The **ui-insert-assumed-match** function inserts an assumed match record into *ui_assumed_match* by assuming that the first record in the search cursor (that is, the record with the highest matching weight) was used to perform an assumed match. You can only call this function after **ui-update-person**. You should also call **ui-get-assumed-match-enabled** before **ui-insert-assumed-match** to ensure that records are only being written to the *ui_assumed_match* table if the ASSMTCH control key is enabled.

Syntax

```
(ui-insert-assumed-match connection-handle)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.

Return Value

The **ui-insert-assumed-match** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The assumed match record was successfully inserted into *ui_assumed_match*.
MONK_EXCEPTION	The assumed match record was not successfully inserted into *ui_assumed_match*. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

See the example for **ui-get-assumed-match-enabled**, on page 4-44.

ui-insert-aux-id

The **ui-insert-aux-id** function adds a non-unique ID to a member's record given a specific ID type and the member's UID. Before calling **ui-insert-aux-id**, you must call **ui-start-transaction** to designate the UID of the record that is being modified.

Syntax

```
(ui-insert-aux-id connection-handle id-type id)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
id-type	string	The type of non-unique ID you want to add to the specified member's record.
id	string	The non-unique ID number to be added to the specified member's record.

Return Value

The **ui-insert-aux-id** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The new non-unique ID and ID type were successfully added to the member record.
MONK_EXCEPTION	The new non-unique ID and ID type were not successfully added to the member record. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

See the example for **ui-exists-aux-id** on page 4-38 and for **ui-get-aux-id** on page 4-46.

ui-insert-local-id

The **ui-insert-local-id** function inserts a local ID for the system defined by the current transaction. Before calling **ui-insert-local-id**, you must call **ui-start-transaction** to designate the UID of the record that is being modified.

Syntax

```
(ui-insert-local-id connection-handle local-id system)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
local-id	string	The local ID number you want to add to the member record.
system	string	The code of the system that is associated with the specified local ID.

Return Value

The **ui-insert-local-id** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The local ID and system pair was inserted into the member record successfully.
MONK_EXCEPTION	The local ID and system pair was not inserted into the member record successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

See the example for **ui-get-local-id** on page 4-56.

ui-insert-person

The **ui-insert-person** function inserts a new person record into the database using a demographic list. You must call **ui-start-transaction** before inserting a person record.

Syntax

```
(ui-insert-person connection-handle demo)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
demo	list	A Monk list containing the demographic information to be inserted into the new member record.

Return Value

The **ui-insert-person** API returns one of the following values:

This value is returned …	if this occurs …
A UID	The new person record was successfully inserted into the database.
MONK_EXCEPTION	The new person record was not successfully inserted into the database. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message, if any.

Example

See the example for **ui-get-local-id** on page 4-56.

ui-local-id-merge

The **ui-local-id-merge** function merges two member records based on their local IDs in a specific system. The record associated with the source local ID (kept-lid) is merged into the record associated with the destination local ID (merged-lid), retaining only the information from the destination local ID. If two UID records are merge, the **ui-local-id-merge** API initiates two application transactions, one for each record. If either application transaction fails, the database transaction is rolled back. If only a local ID merge occurs, then a single transaction is initiated and the status of the merged local ID changes to **M**. You do not need to call **ui-local-id-merge** within a **ui-start-transaction** and **ui-commit-transaction** pair. After you call **ui-local-id-merge**, be sure to call **db-commit** to commit the transaction.

Syntax

```
(ui-local-id-merge connection-handle merged-lid kept-lid
system source dept term-id user-id)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
merged-lid	string	The local identifier associated with the record whose information will *not* be retained in the final merge result record.
kept-lid	string	The local identifier associated with the record whose information *will* be retained in the final merge result record.
system	string	The system code of the system associated with the local IDs you want to merge.
source	string	The source code of the application from which the merge information originated.
dept	string	The department code for the department in which the merge transaction occurred.
term-id	string	The terminal ID for the terminal at which the merge transaction occurred.
user-id	string	The login ID of the user who performed the merge transaction.

Return Value

The **ui-local-id-merge** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The merge is performed using the member's local IDs, and is completed successfully.
MONK_EXCEPTION	The merge is not performed successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

This example specifies two local IDs in one system in order to merge the records. First it finds the local IDs to be merged in the incoming message, and then calls the **ui-local-id-merge** API to merge the two records together. The information in the record specified by **kept-lid** is retained in the new record, and the information in the record specified by **merged-lid** is retained for history information only.

```
...
 (begin
  (if (empty-string? ~input%eiEvent.REC[0].ID.prior_local_id.LID.id)
   (begin
    (copy-strip ~input%eiEvent ~output%eiEvent.out "")
   )
   (begin
    (let ((kept-lid (get ~input%eiEvent.REC[0].ID.local_id.LID.id))
          (merged-lid (get
~input%eiEvent.REC[0].ID.prior_local_id.LID.id))
          (system (get ~input%eiEvent.REC[0].ID.local_id.LID.system))
          (source (get ~input%ieEvent.REC[0].EVN.source))
          (dept (get ~input%ieEvent.REC[0].EVN.department))
          (term-id (get ~input%ieEvent.REC[0].EVN.terminal_id))
          (user-id (get ~input%ieEvent.REC[0].EVN.user_id))
        )
    )

    (if (ui-local-id-merge connection-handle merged-lid kept-lid system
        source dept term-id user-id)
     (begin
       (begin (newline))
   (begin (db-commit connection-handle))
...
```

ui-local-id-status

The **ui-local-id-status** function retrieves the status code of the specified local ID and system pair. If a local ID has more than one status (which would occur with a local ID that was deactivated and then re-entered), this function finds all statuses for the ID and returns one status based on the following order: A (active), M (merged), and D (deactivated).

Syntax

```
(ui-local-id-status connection-handle system lid)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
system	string	The system code of the system associated with the local identifier whose status you want to retrieve.
lid	string	The local identifier whose status you want to retrieve.

Return Value

The **ui-local-id-status** API returns one of the following values:

This value is returned …	if this occurs …
Status code	The status of the local ID was retrieved successfully. The possible status codes are: **A** for active, **D** for deactivated, and **M** for merged.
MONK_EXCEPTION	The status of the specified local ID was not retrieved successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The example below sets the variables **system** and **lid** as the system and local ID of an incoming Event. It calls **ui-local-id-status** to check the status of the system and local ID pair.

```
(set! system (get ~input%eiEvent.REC[0].ID.local_id.LID.system))
(set! lid (get ~input%eiEvent.REC[0].ID.local_id.LID.id))
(begin
  (set! status (ui-local-id-status connection-handle system lid))
  ...
```

ui-lookup

The **ui-lookup** function searches for a person's local ID in a specific system (dest-system) by using that person's local ID from another system (source-system). This information is stored in the *ui_local_id* table.

---

*Note: This API only returns the first matching local ID. To retrieve a complete list of local IDs matching the criteria, use **ui-lookup-local-id**.*

Syntax

```
(ui-lookup connection-handle source-system source-local-id
dest-system)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
source-system	string	The system code of the system for which the person's local ID is known.
source-local-id	string	The local ID of the person in the source system.
dest-system	string	The system code of the system for which the person's local ID is not known.

Return Value

The **ui-lookup** API returns one of the following values:

This value is returned …	if this occurs …
A local ID	A local ID associated with the specified system was found in the *ui_local_id* table.
#f	The member record does not have a local ID in the specified destination system.
MONK_EXCEPTION	The lookup was not successfully processed. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The following example uses the connection handle **connection-handle** as defined in *ui-stdver-eway-funcs.monk*. It searches for a person's local ID for system **BDG** by looking up the local ID and system pair from the incoming Event. If a local ID for system **BDG** is found, it is saved in the Monk variable **local_id**.

```
...
(set! source-id (get ~input%eiEvent.REC[0].ID.local_id.LID.id))
(set! dest-system (get ~input%eiEvent.REC[0].ID.local_id.LID.system))
(set! local_id (lookup connection-handle BDG source-id dest-system)
  (if local_id
   (begin
    ...
```

ui-lookup-address-id

The **ui-lookup-address-id** function searches for the identification code of a person's address of a specific type using that person's UID. This information is stored in the *ui_address* table.

Syntax

```
(ui-lookup-address-id connection-handle uid addr-type)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
uid	string	The UID of the person associated with the address ID you want to retrieve.
addr-type	string	The address type code of the address whose ID you want to retrieve.

Return Value

The **ui-lookup-address-id** API returns one of the following values:

This value is returned …	if this occurs …
Address ID	An identification code associated with the specified address type and UID was found in the *ui_address* table.
#f	There is not an address of the specified address type associated with the specified UID.
MONK_EXCEPTION	The lookup was not successfully processed. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

ui-lookup-local-id

The **ui-lookup-local-id** function retrieves all of a member's local IDs in a specific system (the destination system) given a local ID and system pair from a different system (the source system). This function only returns local IDs with the status specified (active, merged, or deactivated).

Syntax

```
(ui-lookup-local-id connection-handle src-system src-lid
dest-system status)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
src-system	string	The system code of the system associated with the known local identifier.
src-lid	string	The member's local identifier in the source system.
dest-system	string	The system code of the system associated with the local ID you want to find.
status	string	The status code of the status of the local IDs you want to retrieve. This value can be **A**, **D**, or **M**, for active, deactivated, and merged, respectively.

Return Value

The **ui-lookup-local-id** API returns one of the following values:

This value is returned …	if this occurs …
A vector containing local IDs	The member's local IDs for the specified system were retrieved successfully.
#f	No local IDs for the specified system exist for the member associated with the source local ID and system.
MONK_EXCEPTION	The member's local IDs were not retrieved successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The example below calls **ui-lookup-local-id** to check the local IDs for the member associated with the local ID and system pair represented by **src-system** and **src-lid**. **ui-lookup-local-id** looks for a local ID associated with the system **RDW** with a status of **A** (active). If local ID and system pairs are found where the system is **RDW**, the sample scrolls through the local IDs using the **ui-get-next-element** and **ui-has-next-element** functions.

```
(set! src-system (get ~input%eiEvent.REC[0].ID.local_id.LID.system))
(set! src-lid (get ~input%eiEvent.REC[0].ID.local_id.LID.id))
(begin
  (set! lids (ui-lookup-local-id connection-handle src-system src-lid "RDW"
"A"))
  (if lids
    (begin
      (do
        ((i 0 (+ i 1)))
        ((not (ui-has-next-element lids)))
        (display (format "%d: %a\n" i (ui-get-next-element lids)))
      )
    )
  )
  (display "No LIDs\n")
)
```

ui-merge

The **ui-merge** function merges two records, specified by their UID numbers. The record associated with the source UID (source-uid) is merged into the record associated with the destination UID (dest-uid), retaining only the information associated with the destination UID and the local identifiers and aliases from both UIDs. The **ui-merge** API initiates two application transactions, one for each UID record, so you do not need to call **ui-merge** within a **ui-start-transaction** and **ui-commit-transaction** pair. If either application transaction fails, the database transaction is rolled back. After you call **ui-merge**, be sure to call **db-commit** to commit the transaction.

Syntax

```
(ui-merge connection-handle source-uid dest-uid system source
dept terminal-id user-id)
```

Parameters

Parameter	Type	Description
connection- handle	connection handle	A handle to the database.
source-uid	string	The UID of the person whose records will not be retained after the merge.
dest-uid	string	The UID of the person whose records will be retained after the merge.
system	string	The system from which the incoming record was sent.
source	string	The application from which the merge information originated.
dept	string	The department code of the department from which the merge information originated.
terminal-id	string	The ID of the terminal on which the merge transaction was performed.
user-id	string	The login ID of the user who performed the merge transaction.

Return Value

The **ui-merge** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The merge is performed using the member's UIDs, and is completed successfully.
MONK_EXCEPTION	The merge is not performed successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

This example uses the UIDs from two member profiles to merge the records. First it finds the members' local IDs from the incoming message, and then uses the **ui-get-uid** API to find the UIDs of the member records being merged. The information from the record specified by **dest-local-id** and **dest-uid** is retained in the new record.

```
...
 (begin
  (if (empty-string? ~input%eiEvent.REC[0].ID.prior_local_id.LID.id)
   (begin
    (copy-strip ~input%eiEvent ~output%eiEvent.out "")
   )
   (begin
    (let ((dest-lid (get ~input%eiEvent.REC[0].ID.local_id.LID.id))
        (source-lid (get ~input%eiEvent.REC[0].ID.prior_local_id.LID.id))
        (system (get ~input%eiEvent.REC[0].ID.local_id.LID.system))
        (source (get ~input%ieEvent.REC[0].EVN.source))
        (dept (get ~input%ieEvent.REC[0].EVN.department))
        (term-id (get ~input%ieEvent.REC[0].EVN.terminal_id))
        (user-id (get ~input%ieEvent.REC[0].EVN.user_id))
      )
    )
    (set! source-uid (ui-get-uid connection-handle system source-lid))
    (set! dest-uid (ui-get-uid connection-handle system dest-lid))

    (comment "Now merge the records" "")

    (if (ui-merge connection-handle source-uid dest-uid system source dept term-
id user-id)
      (begin
        (begin (newline))
   (begin (db-commit connection-handle))
...
```

ui-process-address

The **ui-process-address** function uses an address list to perform an address update or insert, depending on whether an address record already exists in the person record for the given address type and UID. This function contains an overwrite flag to indicate whether the function should update an existing record. If the flag is turned off and there is an existing address record of the specified type in the person record, then this function will not update the address information. However, if the flag is turned off and the address type does not exist, the function will still insert a new address. This function must be used inside a loop in order to process all addresses in the message.

Syntax

```
(ui-process-address connection-handle address overwrite)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
address	list	A Monk list containing the address information to be updated or inserted into the person record. This list contains the address type of each address.
overwrite	Boolean	An indicator of whether or not to update existing address records. If this flag is set to **#t**, then new address information overwrites existing information. If this set to **#f**, then no existing information is overwritten.

Return Value

The **ui-process-address** API returns one of the following values:

This value is returned …	if this occurs …
The string 'NEW'	The new address information was successfully inserted in the person record.
The string 'UPDATE'	The new address information successfully updated existing address information in the person record.
#f	The new information was not inserted or updated in the person record because the record already exists and no changes were required. This value is also returned if the overwrite flag is set to **#f** and an address of the specified type already exists in the person record (even if the address content is different).

This value is returned …	if this occurs …
MONK_EXCEPTION	The address record was not successfully processed. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The following example is excerpted from the default file *uidb.dsc*. It first calls **ui-process-person** to process the demographic information in the incoming message. After processing the demographic data, it then retrieves the address list from the incoming Event, and compares it with the addresses in the record processed by **ui-process-person**. Depending on whether the overwrite flag for **ui-process-address** is set to true or false, the address information from the incoming Event is updated to the record in the database.

```
...
(set! demo (get-demographics ~input%eiEvent))
(set! trans (get-transact ~input%eiEvent))
(set! local-id (get ~input%eiEvent.REC[0].ID.local_id.LID.id))
(set! system (get ~input%eiEvent.REC[0].ID.local_id.LID.system))
(set! uid (ui-process-person connection-handle local-id system demo trans))
(duplicate-strip ~input%eiEvent ~output%eiEvent "")
(insert-hard uid ~output%eiEvent.REC[0].ID.eid_1)
 (display (string-append "PROCESS PERSON RESULT: " ))
 (display (string-append uid " " (ui-get-demographic-changed)))(newline)

...  ;processing alias records

(do ((j 0 (+ j 1))) ((>= j (count ~input%eiEvent.REC[0].DEMO.address)))
  (set! address (get-address ~input%eiEvent.REC[0] j))
  (display (string-append "ADDRESS" "[" (number->string j) "]"))
  (display address) (newline)
  (display "PROCESS ADDRESS RESULT: " )
  (display (ui-process-address connection-handle address overwrite))
  (newline)
)
...
```

ui-process-person

The **ui-process-person** function is used to add, update, or delete member information in the e*Index database. A sample function is defined in the file *ui-process-person.monk*, and you can customize this file with additional APIs as required. In the default configuration, this function initiates a search through the database, using the search algorithm to determine if a person already exists in the database. If the person does not exist, then a new UID is returned. If the person does exist, then the UID of the existing record is returned. The processing logic that is used for the default function is described in "About Inbound Event Processing Logic" in Chapter 2 of this guide.

*Note: The syntax, parameters, return values, and example in this section describe **ui-process-person** as it is defined in your standard installation. This command can be customized for your processing needs.*

Syntax

```
(ui-process-person connection-handle local-id system demo
trans)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
local-id	string	The local ID of the person you are processing.
system	string	The identification code for the system associated with the specified local ID.
demo	list	A Monk list that contains the demographic information of the person identified by the local ID.
trans	list	A Monk list that contains the transaction information of the person identified by the local ID.

Return Value

The **ui-process-person** API returns one of the following values:

This value is returned …	if this occurs …
A new or existing UID	The member information was processed successfully.
MONK_EXCEPTION	The member information was not processed successfully. This would occur if a UID was not found and a new UID could not be assigned, or if the database server is down. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The example below calls **ui-process-person** to process the demographic data in the incoming record. If a matching record is found in the database or a new record is inserted, then a UID is returned and processing continues. To see how **ui-process-person** is used in the default Collaboration script, **uidb.dsc**, see the example for **ui-process-address** on page 4-81.

```
...
(set! demo (get-demographics ~input%eiEvent))
(set! trans (get-transact ~input%eiEvent))
(set! local-id (get ~input%eiEvent.REC[O].ID.local_id.LID.id))
(set! system (get ~input%eiEvent.REC[O].ID.local_id.LID.system))
(set! uid (ui-process-person connection-handle local-id system demo
trans))

(if uid
  (begin
   ...
```

ui-process-phone

The **ui-process-phone** function uses a telephone number list to perform a telephone number update or insert, depending on whether a record already exists in the person record for the given telephone number type and UID. This function contains an overwrite flag to indicate whether the function should update an existing record. If the flag is turned off and a telephone record of the same type exists in the person record, then this function will not update the existing record. However, if the flag is turned off and the telephone type does not exist, the function will still insert a new telephone record. This function must be used inside a loop in order to process all telephone numbers in the message.

Syntax

```
(ui-process-address connection-handle phone overwrite)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
phone	list	A Monk list containing the telephone information to be updated or inserted into the person record. This list includes the telephone number type.
overwrite	string	An indicator of whether or not to update existing telephone records. If this flag is set to **#t**, then the new telephone information overwrites existing information. If this set to **#f**, then no existing information is overwritten.

Return Value

The **ui-process-phone** API returns one of the following values:

This value is returned …	if this occurs …
The string 'ADD'	The new telephone information was successfully inserted in the person record.
The string 'UPDATE'	The new telephone information successfully updated existing telephone information in the person record.

This value is returned …	if this occurs …
#f	The new information was not inserted or updated in the person record because the record already exists and no changes were required. This value is also returned if the overwrite flag is set to **#f** and a telephone number of the specified type already exists in the person record (even if the content is different).
MONK_EXCEPTION	The address record was not successfully processed. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The following example is taken from the default file *uidb.dsc*. It first calls **ui-process-person** to process the demographic information in the incoming message. After processing the demographic data, it then retrieves the telephone number list from the incoming Event, and compares it with the numbers in the record processed by **ui-process-person**. Depending on whether the overwrite flag for **ui-process-phone** is set to true or false, the address information from the incoming Event is updated to the record in the database.

```
...
(set! demo (get-demographics ~input%eiEvent))
(set! trans (get-transact ~input%eiEvent))
(set! local-id (get ~input%eiEvent.REC[O].ID.local_id.LID.id))
(set! system (get ~input%eiEvent.REC[O].ID.local_id.LID.system))
(set! uid (ui-process-person connection-handle local-id system demo trans))
(duplicate-strip ~input%eiEvent ~output%eiEvent "")
(insert-hard uid ~output%eiEvent.REC[O].ID.eid_1)
 (display (string-append "PROCESS PERSON RESULT: " ))
 (display (string-append uid " " (ui-get-demographic-changed)))(newline)

...  ; processing alias records
...  ; processing address records

(do ((j O (+ j 1))) ((>= j (count ~input%eiEvent.REC[O].DEMO.phone)))
 (set! phone (get-phone ~input%eiEvent.REC[O] j))
 (display (string-append "PHONE" "[" (number->string j) "]"))
 (display phone) (newline)
 (display "PROCESS PHONE RESULT: " )
 (display (ui-process-phone connection-handle phone overwrite))(newline)
)
...
```

ui-rollback-transaction

The **ui-rollback-transaction** function rolls back the current transaction in the database and resets the transaction structure. You can use this function to undo any changes made to a record in the database in the case that an error occurs during processing.

Syntax

```
(ui-rollback-transaction connection-handle)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.

Return Value

The **ui-rollback-transaction** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The transaction was successfully rolled back.
MONK_EXCEPTION	The transaction was not successfully rolled back. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The following example displays the "catch" section of a try-catch block. The **catch** command is called when an error occurs during processing and defines how to handle the error. It then rolls back the transaction during which the error occurred.

```
(catch
 (otherwise
  (display (exception-string-all))(newline)
  (display (ui-get-error-string))(newline)
  (display (db-get-error-str connection-handle))(newline)
   (if (db-alive connection-handle)
    (begin
     (set! result "")
    )
    (begin
     (set! result "CONNERR")
    )
   )
   (ui-rollback-transaction connection-handle)
 ))
```

ui-search-close

The **ui-search-close** function closes the search cursor, and de-allocates the memory. Use this function at the end of each **ui-search-open** function. Note that you only need to call **ui-search-close** in cases where records are returned from a search (that is, **ui-search-open** does not return **#f**). If **ui-search-open** returns **#f**, no records were found and there is no cursor to close.

Syntax

```
(ui-search-close)
```

Parameters

Parameter	Type	Description
None		

Return Value

The **ui-search-close** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The search cursor was closed and the memory was de-allocated successfully.
MONK_EXCEPTION	The search cursor was not closed successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The following example, taken from the default *ui-process-person.monk* file, opens a search cursor and looks for possible matches to the incoming record based on the demographic information from the incoming record. If records are found, it inserts all the records in the search cursor into *ui-duplic* and flags them as potential duplicates of the incoming record. After inserting the records into *ui_duplic*, the search cursor is closed and the resources allocated by **ui-search-open** are de-allocated.

```
(set! demo (get-demographics ~input%eiEvent))
...
(set! search-count (ui-search-open connection-handle demo))
 (if search-count
  (ui-search-insert-duplicate connection-handle "POTENTIAL DUPLICATE"
   1 search-count)
  (ui-search-close)
 )
)
...
```

ui-search-get-exact-match-score

The **ui-search-get-exact-match-score** function retrieves the weight of an exact match to a new record if one exists in the database. This function checks to see if 1EXACTMTCH is enabled before returning the weight. If this control key is set to **Yes** and there are two records above the match threshold in the result set, then neither record is considered an exact match of the new record. This function can only be called within a **ui-search-open** function.

Syntax

```
(ui-search-get-exact-match-score)
```

Parameters

Parameter	Type	Description
None		

Return Value

The **ui-search-get-exact-match-score** API returns one of the following:

This value is returned …	if this occurs …
A matching weight	One exact match exists in the search results set, and the matching weight of that record is returned.
#f	An exact match does not exist in the search results set, or more than one record in the results set is above the match threshold and 1EXACTMTCH is enabled.
MONK_EXCEPTION	The search cursor is not open. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The following example opens a search cursor and looks for any records in the database that are possible matches of the incoming Event based on the Event's demographic information. If possible matches are found and one of those matches is above the match threshold, the UID and the matching probability weight of that record is returned.

```
(set! demo (get-demographics ~input%eiEvent))
...
(begin
    (set! search-count (ui-search-open connection-handle demo))
    (if search-count
      (set! uid (ui-search-get-exact-match-uid))
      (set! weight (ui-search-get-exact-match-score))
    )
...
```

ui-search-get-exact-match-uid

The **ui-search-get-exact-match-uid** function retrieves the UID of a person record that is an exact match of a new record (if an exact match exists in the database). This function can only be called within a **ui-search-open** function.

Syntax

```
(ui-search-get-exact-match-uid)
```

Parameters

Parameter	Type	Description
None		

Return Value

The **ui-search-get-exact-match-uid** API returns one of the following values:

This value is returned …	if this occurs …
A UID	An exact match of the new record exists in the database, and the UID was retrieved successfully.
#f	An exact match does not exist in the search results set, or more than one record in the results set is above the match threshold and 1EXACTMTCH is enabled.
MONK_EXCEPTION	The search cursor is not open. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The following example opens a search cursor and looks for any records in the database that are possible matches of the incoming Event based on the Event's demographic information. If possible matches are found and one of those matches is above the match threshold, the UID of that record is returned. This record is known as the exact match.

```
(set! demo (get-demographics ~input%eiEvent))
...
(begin
    (set! search-count (ui-search-open connection-handle demo))
    (if search-count
      (set! uid (ui-search-get-exact-match-uid))
    )
...
```

ui-search-insert-duplicate

The **ui-search-insert-duplicate** function allows the records in the search cursor to be added to the *ui_duplic* table along with their matching probability weights. The start and end indexes determine the records in the search cursor that are sent to the duplicate table. This function can only be called within a **ui-search-open** function.

Syntax

```
(ui-search-insert-duplicate dup-msg start-index end-index)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
dup-msg	string	The text to be inserted into the description column of the ui_duplic table (e.g., SAME SYSTEM:). The description is typically "Same System" or "Potential Duplicate."
start-index	integer	The index of the search cursor to begin inserting duplicate records. The first record in the search cursor has an index of 1.
end-index	integer	The index of the search cursor to stop inserting duplicate records. The last record in the search cursor has an index equal to the search count.

Return Value

The **ui-search-insert-duplicate** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The duplicate records were successfully inserted into the *ui_duplic* table.
MONK_EXCEPTION	The duplicate records were not successfully inserted into the *ui_duplic* table. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

See the example for **ui-search-close** on page 4-88.

ui-search-local-id

The **ui-search-local-id** function retrieves all of a member's local IDs in a specific system given the system code and member's UID. This function only returns local IDs with the status specified (active, merged, or deactivated).

Syntax

```
(ui-search-local-id connection-handle uid system status)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
uid	string	The UID of the member for which you want to retrieve the local ID.
system	string	The code for the system with which the local ID you want to retrieve is associated.
status	string	The status code of the status of the local IDs you want to retrieve. This value can be **A**, **D**, or **M**, for active, deactivated, and merged, respectively.

Return Value

The **ui-search-local-id** API returns one of the following values:

This value is returned …	if this occurs …
A vector containing local IDs	The member's local IDs for the specified system were retrieved successfully.
#f	No local IDs for the specified system exist for the specified member (this indicates a flaw in data integrity).
MONK_EXCEPTION	The member's local IDs were not retrieved successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The example below searches for person records in the database that closely match the incoming Event, and, if an exact match is found, retrieves the UID of the exact match. It then calls **ui-search-local-id** to check the entries in the *ui_local_id* table that are associated with the exact match record that have a status of **A** (active). If local ID and system pairs are found where the system matches the system in the incoming Event, the sample scrolls through the local IDs using the **ui-get-next-element** and **ui-has-next-element** functions.

```
(set! demo (get-demographics ~input%eiEvent))
(set! trans (get-transact ~input%eiEvent))
(set! system (get ~input%eiEvent.REC[O].ID.local_id.LID.system))
(begin
  (set! search-count (ui-search-open connection-handle demo))
  (if search-count
     (set! uid (ui-search-get-exact-match-uid))
  )
  (if uid
   (begin
     (set! lids (ui-search-local-id connection-handle uid system "A"))
     (if lids
       (begin
         (do
           ((i O (+ i 1)))
           ((not (ui-has-next-element lids)))
           (display (format "%d: %a\n" i (ui-get-next-element lids)))
         )
       )
     )
     (display "No LIDs\n")
  ))
```

ui-search-open

The **ui-search-open** function opens a cursor of weighted records returned from a search based on a demographic information query list. This function returns the number of records in the result set. If records are returned from the search, the search cursor remains open until **ui-search-close** is called. If no records are returned, **ui-search-open** returns **#f** and no search cursor is opened.

Syntax

```
(ui-search-open connection-handle demo)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
demo	list	A Monk list containing demographic information about the person for whom you are searching.

Return Value

The **ui-search-open** API returns one of the following values:

This value is returned …	if this occurs …
The number of resulting records	The search is performed successfully, and the number of matching records is counted.
#f	The search is performed successfully, and there are no matching records in the search cursor.
MONK_EXCEPTION	The search is not performed successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

See the example for **ui-search-close** on page 4-88.

ui-set-dup-threshold

Use the **ui-set-dup-threshold** function to specify the minimum matching probability weight at which two records are considered potential duplicates of one another. If you specify a duplicate threshold, it overrides the value read by **ui-config** for the control key DUPTHRES. If you do not call **ui-set-dup-threshold**, then the control key value is used.

Syntax

```
(ui-set-dup-threshold threshold)
```

Parameters

Parameter	Type	Description
threshold	real	A number indicating the duplicate threshold.

Return Value

The **ui-set-dup-threshold** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The threshold was created successfully.
MONK_EXCEPTION	The threshold was not created successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The following example sets the duplicate threshold to 8.5 and the match threshold to 13.0. You can then call ui-process-person, which uses those thresholds to process member records.

```
(ui-set-dup-threshold 8.5)
(ui-set-match-threshold 13.0)
```

ui-set-match-threshold

Use the **ui-set-match-threshold** function to specify the minimum matching probability weight at which two records are considered to represent the same person, and are automatically merged. If you specify a match threshold, it overrides the value read by **ui-config** for the control key MATCHTHRES. If you do not call **ui-set-match-threshold**, then the control key value is used.

Syntax

```
(ui-set-match-threshold threshold)
```

Parameters

Parameter	Type	Description
Threshold	real	A number indicating the match threshold.

Return Value

The **ui-set-match-threshold** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The threshold was created successfully.
MONK_EXCEPTION	The threshold was not created successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

The following example sets the duplicate threshold to 8.5 and the match threshold to 13.0. You can then call ui-process-person, which uses those thresholds to process member records.

```
(ui-set-dup-threshold 8.5)
(ui-set-match-threshold 13.0)
```

ui-set-queue-id

Use the **ui-set-queue-id** function to change the status of a queued message if the dequeued message is not sent successfully (as determined by a nack event in the polling e*Way). In the sample poll schema, this command is executed within **ui-poll-neg-ack**.

Syntax

```
(ui-set-queue-id connection-handle msg-id queue-id)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
msg-id	string	The identification code of the outgoing message (from the table *ui_msg_detail*).
queue-id	string	The current status of the message in *ui_msg_detail*. The possible statuses are:
		O – This stands for "output" and means the message is ready to be sent
		E – This means there was an error when sending the message (in the sample schema).
		B – This stands for "buffer" and means the message has been dequeued and is being sent.

Return Value

The **ui-set-queue-id** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The status of the message was changed successfully.
MONK_EXCEPTION	The status of the message was not changed successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

This sample from **ui-poll-neg-ack** in *ui-stdver-eway-funcs.monk* defines the variable **msg-id** as the msg_id field in the outgoing event, and then changes the status of the message to **E** to indicate there was a negative acknowledgement when the record was sent.

```
...
 (begin
  (set! msg-id (get ~input%eiEvent.EVNT.EVN.msg_id))
  (display (format "Setting error status of message ID [%s]\n" msg-id))
  (ui-set-queue-id connection-handle msg-id "E")
 )
...
```

ui-start-transaction

The **ui-start-transaction** function begins a transaction for a new Event. If the UID parameter is left blank, the incoming Event is a new record and the next available UID is assigned. If the UID parameter is not blank, then a transaction is started for the specified UID. An actual transaction is only initiated if a database insert, update, or delete is performed during the call to **ui-start-transaction**.

Syntax

```
(ui-start-transaction connection-handle uid trans)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
uid	string	The UID number of the person record for which you want to start a transaction. If this parameter is left blank, the next available UID is used.
trans	list	A Monk list containing information about the transaction currently being processed.

Return Value

The **ui-start-transaction** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The transaction was started successfully.
MONK_EXCEPTION	The transaction was not started successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

See the example for **ui-delete-unresolved-duplicates** on page 4-34.

ui-unmerge

The **ui-unmerge** function unmerges two records, specified by the active UID number. The records are returned to their statuses prior to the merge transaction. The **ui-unmerge** API initiates two application transactions, one for each UID record, so you do not need to call **ui-unmerge** within a **ui-start-transaction** and **ui-commit-transaction** pair. If either application transaction fails, the database transaction is rolled back. Before calling ui-unmerge, you should verify the status of the UID you want to unmerge. After you call **ui-unmerge**, be sure to call **db-commit** to commit the transaction.

Syntax

```
(ui-unmerge connection-handle kept-uid system source dept
terminal-id user-id)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
kept-uid	string	The UID of the active record to unmerge.
system	string	The system from which the unmerge record originated.
source	string	The application from which the unmerge record originated.
dept	string	The processing code of the department in which the unmerge record originated.
terminal-id	string	The ID of the terminal on which the unmerge transaction was performed.
user-id	string	The login ID of the user who performed the unmerge transaction.

Return Value

The **ui-unmerge** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The unmerge is performed using the member's UID, and is completed successfully.
MONK_EXCEPTION	The unmerge is not performed successfully. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

This example uses the UID from an active, merged member profile to unmerge two records. First it finds the UID and other pertinent information from the incoming message, and then calls **ui-unmerge** to reverse the record's most recent merge transaction.

```
...
 (begin
    (let ((kept-uid (get ~input%eiEvent.REC[O].ID.eid_1))
       (system (get ~input%eiEvent.REC[O].ID.local_id.LID.system))
       (source (get ~input%ieEvent.REC[O].EVN.source))
       (dept (get ~input%ieEvent.REC[O].EVN.department))
       (term-id (get ~input%ieEvent.REC[O].EVN.terminal_id))
       (user-id (get ~input%ieEvent.REC[O].EVN.user_id))
      )
    )
    (if (ui-unmerge connection-handle kept-uid system source dept term-
id user-id)
      (begin
        (begin (newline))
  (begin (db-commit connection-handle))
...
```

ui-update-address

The **ui-update-address** function updates an address given the unique address ID. You can call **ui-exists-address-id** to obtain the address ID. Before calling **ui-update-address**, you must call **ui-start-transaction**.

Syntax

```
(ui-update-address connection-handle address-id address)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
address-type	string	The address ID of the existing address you want to update.
address	list	A Monk list containing the new address information with which to update the existing address.

Return Value

The **ui-update-address** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The address information for the specified address record was successfully updated.
MONK_EXCEPTION	The address information for the specified address ID was not successfully updated. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

ui-update-aux-id

The **ui-update-aux-id** function updates a non-unique ID given the ID type, the old ID, and the new ID. Before calling **ui-update-aux-id**, you must call **ui-start-transaction** to designate the UID of the record that is being modified.

Syntax

```
(ui-update-aux-id connection-handle id-type old-id new-id)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
id-type	string	The type of non-unique ID you want to update in the specified person record.
old-id	string	The existing non-unique ID that you want to update.
new-id	string	The new non-unique ID to replace the existing ID.

Return Value

The **ui-update-aux-id** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The non-unique ID in the specified person record was successfully updated.
MONK_EXCEPTION	The non-unique ID in the specified person record was not successfully updated. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

See the example for **ui-get-aux-id** on page 4-46.

ui-update-person

The **ui-update-person** function updates a person record using the specified demographic list. Before calling **ui-update-person**, you must call **ui-start-transaction** to designate the UID of the record that is being modified.

Syntax

```
(ui-update-person connection-handle demo)
```

Parameters

Parameter	Type	Description
connection-handle	connection handle	A handle to the database.
demo	list	A Monk list containing demographic information about the person whose record you want to update.

Return Value

The **ui-update-person** API returns one of the following values:

This value is returned …	if this occurs …
MONK_UNINITIALIZED	The person record was successfully updated with the new demographic information.
MONK_EXCEPTION	The person record was not successfully updated with the new demographic information. Use the **db-get-error-str** or **ui-get-error-string** API to retrieve the corresponding error message.

Example

See the example for **ui-delete-unresolved-duplicates** on page 4-34.

Standard Monk API Descriptions

Overview

This section of the chapter lists all of the Monk APIs that are included in the file *ui-stdver-eway-funcs.monk*. These functions are called by the configuration file when the e*Way starts up.

ui-stdver-init

The **ui-stdver-init** function begins the initialization process for an e*Way. The function loads all of the monk extension library files that are accessed by the other e*Way functions and defines a connection handle for the database. In the default configuration, this function calls the Monk script **ui-set-vticfg**, which sets the Vality environment variable to /<eGate>/client/bin (the location of the Vality rule set files for the e*Way).

Syntax

```
(ui-stdver-init)
```

Parameters

Parameter	Type	Description
None		

Return Values

The **ui-stdver-init** API returns one of the following values:

This value is returned …	if this occurs …
A string (other than "FAILURE")	The initialization occurred successfully.
A "FAILURE" string	The initialization did not occur successfully.
	*Note: In this instance, the e*Way will shut down.*

Standard Implementation

The example below illustrates how **ui-stdver-init** is defined in *ui-stdver-eway-funcs.monk*.

```
(define ui-stdver-init
 (lambda (  )
  (let ((result "SUCCESS"))
 (display "[++] Executing dart e*Way external init function.")
 (display "[++] Loading dart-eWay-stdver-funcs.monk  ")
 (display "DATABASE TYPE = ")
```

```
(display DATABASE_SETUP_DATABASE_TYPE)
(newline )
(define STCDB DATABASE_SETUP_DATABASE_TYPE)
(define DART_NULL "_NULL_")
(define DART_NULL_MODE "INOUT")
(define STCDATADIR (get-data-dir))
(define dg-shutdown #f)
(ui-set-vticfg)
(if (not (load-extension "stc_monkutils.dll"))
 (begin
  (set! result "FAILURE")
  (display "Failed to load stc_monkutils.dll.")
  )
  (begin
  (display " Loaded stc_monkutils.dll ")
))
(if (not (load-extension "stc_dbmonkext.dll"))
  (begin
  (set! result "FAILURE")
  (display "Failed to load stc_dbmonkext.dll.")
  (event-send "ALERTCAT_OPERATIONAL" "ALERTSUBCAT_UNUSABLE"
   "ALERTINFO_FATAL" "O" "stc_dbmonkext.dll load error" "Failed to load
   stc_dbmonkext.dll" O (list))
  )
  (begin
  (display "Loaded stc_dbmonkext.dll")
))
(if (not (load-extension "stc_uimonkext.dll"))
  (begin
  (set! result "FAILURE")
  (display "Failed to load stc_uimonkext.dll.")
  (event-send "ALERTCAT_OPERATIONAL" "ALERTSUBCAT_UNUSABLE"
   "ALERTINFO_FATAL" "O" "stc_uimonkext.dll load error" "Failed to load
   stc_uimonkext.dll" O (list))
  )
  (begin
  (display "Loaded stc_uimonkext.dll")
))
(define connection-handle O)
(set! connection-handle (make-connection-handle))
(if (connection-handle? connection-handle)
 (begin
  )
  (begin
  (set! result "FAILURE")
  (display "Failed to create connection handle.")
```

```
    (event-send "ALERTCAT_OPERATIONAL" "ALERTSUBCAT_UNUSABLE"
"ALERTINFO_FATAL" "O" "database connection handle creation error" "Failed
to create database connection handle" O (list))
 ))
result
   )
))
```

ui-stdver-startup

The **ui-stdver-startup** function is used for instance-specific function loads, and invokes setup.

Syntax

```
(ui-stdver-startup)
```

Parameters

Parameter	Type	Description
None		

Return Values

The **ui-stdver-startup** API returns one of the following values:

This value is returned …	if this occurs …
A string (other than FAILURE)	The function load and setup occurred successfully.
A FAILURE string	The function load and setup did not occur successfully.
	Note: In this instance, the e*Way will shut down.

Standard Implementation

The example below illustrates how **ui-stdver-startup** is defined in *ui-stdver-eway-funcs.monk*.

```
(define ui-stdver-startup
 (lambda (  )
  (let ((result "SUCCESS"))
  (display "[++] Executing e*Way external startup function.")
 result
  )
))
```

ui-stdver-conn-estab

The **ui-stdver-conn-estab** function establishes a connection to external systems.

Syntax

```
(ui-stdver-conn-estab)
```

Parameters

Parameter	Type	Description
None		

Return Values

The **ui-stdver-conn-estab** API returns one of the following values:

This value is returned …	if this occurs …
An "UP" string	The connection is established successfully.
A "SUCCESS" string	The connection is established successfully.
Any other string	The connection is not established successfully.

Additional Information

To use standard database time format, add the following function call to this function after the (db-bind) call:

```
(db-std-timestamp-format connection-handle)
```

Standard Implementation

The example below illustrates how **ui-stdver-conn-estab** is defined in *ui-stdver-eway-funcs.monk*.

```
(define ui-stdver-conn-estab
 (lambda (  )
 (let ((result "DOWN")(last-dberr ""))
 (display "[++] Executing e*Way external connection establishment
 function.")
 (display "db-retry-conn: logging into the database with:\n")
 (display "DATABASE NAME = ")
 (display DATABASE_SETUP_DATABASE_NAME)
 (newline )
 (display "USER NAME = ")
 (display DATABASE_SETUP_USER_NAME)
 (newline )
 (if (db-login connection-handle DATABASE_SETUP_DATABASE_NAME
      DATABASE_SETUP_USER_NAME DATABASE_SETUP_ENCRYPTED_PASSWORD)
  (begin
   (display "The result of ui-config is ")
   (try (ui-config connection-handle)
     (begin
       (display "OK\n")
       (set! result "UP")
     )
     (catch
      (otherwise
       (display "FAILED")
       (newline)
       (display (exception-string-all))
       (newline)
       (display (ui-get-error-string))
       (newline)
       (display (db-get-error-str connection-handle))
       (db-logout connection-handle)
       (set! connection-handle (make-connection-handle))
       "DOWN"
      )
     )
   )
  )
```

```
  (begin
   (set! last_dberr (db-get-error-str connection-handle))
   (display last_dberr)
   (event-send "ALERTCAT_OPERATIONAL" "ALERTSUBCAT_CANTCONN"
    "ALERTINFO_FATAL" "O" "Cannot connect to database" (string-append
    "Failed to connect to database: " DATABASE_SETUP_DATABASE_NAME "with
    error" last_dberr) O (list))
   (newline )
   (db-logout connection-handle)
   (set! connection-handle (make-connection-handle))
   (set! result "DOWN")
  )
 )
 result
)))
```

ui-stdver-conn-ver

The **ui-stdver-conn-ver** function is used to verify whether external system connection has been established.

Syntax

```
(ui-stdver-conn-ver)
```

Parameters

Parameter	Type	Description
None		

Return Values

The **ui-stdver-conn-ver** API returns one of the following values:

This value is returned …	if this occurs …
An "UP" string	The connection is established successfully.
A "SUCCESS" string	The connection is established successfully.
Any other string	The connection is not established successfully.

Additional Information

To use standard database time format, add the following function call to this function after the (db-bind) call:

```
(db-std-timestamp-format connection-handle)
```

Standard Implementation

The example on the following page illustrates how **ui-stdver-conn-ver** is defined in *ui-stdver-eway-funcs.monk*.

```
(define ui-stdver-conn-ver
 (lambda (  )
  (let ((result "DOWN")(last_dberr ""))
 (display "[++] Executing e*Way external connection verification
     function.")
 (display "ui-stdver-conn-ver: checking connection status...\n")
 (cond ((string=? STCDB "SYBASE") (db-sql-select connection-handle
         "verify" "select getdate()"))
       ((string=? STCDB "SYBASE11") (db-sql-select connection-handle
         "verify" "select getdate()"))
       ((string=? STCDB "SYBASE12") (db-sql-select connection-handle
         "verify" "select getdate()"))
       ((string=? STCDB "ORACLE9i") (db-sql-select connection-handle
         "verify" "select sysdate from dual"))
       ((string=? STCDB "ORACLE8i") (db-sql-select connection-handle
         "verify" "select sysdate from dual"))
       ((string=? STCDB "ORACLE8") (db-sql-select connection-handle
         "verify" "select sysdate from dual"))
       ((string=? STCDB "ORACLE7") (db-sql-select connection-handle
         "verify" "select sysdate from dual"))
       (else (db-sql-select connection-handle "verify" "select {fn
         NOW()}")))
  (if (db-alive connection-handle)
   (begin
    (db-sql-fetch-cancel connection-handle "verify")
    (set! result "UP")
   )
   (begin
    (set! last_dberr (db-get-error-str connection-handle))
    (display last_dberr)
    (event-send "ALERTCAT_OPERATIONAL" "ALERTSUBCAT_LOSTCONN"
        "ALERTINFO_FATAL" "O" "Lost connection to database" (string-
        append "Lost connection to database: "
        DATABASE_SETUP_DATABASE_NAME "with error" last_dberr) O
        (list))
    (set! result "DOWN")
   )
  )
  result
  )
))
```

ui-stdver-conn-shutdown

The **ui-stdver-conn-shutdown** function is called by the system to request that the interface disconnect from the external system in preparation for a suspend/reload cycle. Any return value indicates that the suspend can occur immediately, and the interface will be placed in the down state.

Syntax

```
(ui-stdver-conn-shutdown suspend-string)
```

Parameters

Parameter	Type	Description
suspend-string	string	When the Control Broker issues a shutdown command to the e*Way, the e*Way calls this function and passes the string "SUSPEND_NOTIFICATION" as the parameter.

Return Values

The **ui-stdver-conn-shutdown** API returns one of the following values:

This value is returned …	if this occurs …
A "SUCCESS" string	The interface is ready to suspend.
Any other string	The connection is not established successfully.

Standard Implementation

The example below illustrates how **ui-stdver-conn-shutdown** is defined in *ui-stdver-eway-funcs.monk*.

```
(define ui-stdver-conn-shutdown
 (lambda ( message-string )
  (let ((result "SUCCESS"))
  (display "[++] Executing e*Way external connection shutdown
   function.")
  (display message-string)
  (db-logout connection-handle)
  result
  )
))
```

ui-stdver-pos-ack

The **ui-stdver-pos-ack** function sends a positive acknowledgment to the sending system to verify that an Event was received successfully. This function is also used for post processing after data is successfully sent to e*Gate.

Syntax

```
(ui-stdver-pos-ack message-string)
```

Parameters

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

Return Values

The **ui-stdver-pos-ack** API returns one of the following values:

This value is returned …	if this occurs …
empty string	The positive acknowledgment is sent successfully. The e*Way is then able to proceed with the next request.
CONNERR	Connection with the external is lost.
	Note: In this instance, the e*Way changes to a down state and attempts to reconnect. Upon reconnecting, the pos-ack function is re-executed.

Standard Implementation

The example below illustrates how **ui-stdver-pos-ack** is defined in *ui-stdver-eway-funcs.monk*.

```
(define ui-stdver-pos-ack
 (lambda ( message-string )
  (let ((result ""))
  (display "[++] Executing e*Way external positive
          acknowledgement function.")
  (display message-string)
  result
  )
))
```

ui-stdver-neg-ack

The **ui-stdver-neg-ack** function sends a negative acknowledgment to the sending system to verify that an Event was not received successfully. This function is also used for post processing after failing to send data to e*Gate.

Syntax

```
(ui-stdver-neg-ack message-string)
```

Parameters

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

Return Values

The **ui-stdver-neg-ack** API returns one of the following values:

This value is returned …	if this occurs …
empty string	The negative acknowledgment is sent successfully.
CONNERR	Connection with the external is lost.
	*Note: In this instance, the e*Way changes to a down state and attempts to reconnect. Upon reconnecting, the neg-ack function is re-executed.*

Standard Implementation

The example below illustrates how **ui-stdver-neg-ack** is defined in *ui-stdver-eway-funcs.monk*.

```
(define ui-stdver-neg-ack
 (lambda ( message-string )
  (let ((result ""))
  (display "[++] Executing e*Way external negative
           acknowledgement function.")
  (display message-string)
  result
  )
))
```

ui-stdver-shutdown

The **ui-stdver-shutdown** function is called by the system to request that the connection to the external system shutdown. After calling this function, you need to execute a shutdown-request call from within a Monk function to allow the requested shutdown process to continue.

Syntax

```
(ui-stdver-shutdown shutdown-notification)
```

Parameters

Parameter	Type	Description
shutdown-notification	string	When the Control Broker issues a shutdown command to the e*Way , the e*Way will call this function and pass the string "SHUTDOWN_NOTIFICATION" as the parameter.

Return Values

The **ui-stdver-shutdown** API returns one of the following values:

This value is returned …	if this occurs …
SUCCESS	The shutdown request is received successfully, and shutdown can occur immediately.
Any other string	The shutdown request is not received successfully, and shutdown is delayed until the request is executed successfully.

Standard Implementation

The example below illustrates how **ui-stdver-shutdown** is defined in *ui-stdver-eway-funcs.monk*.

```
(define ui-stdver-shutdown
 (lambda ( message-string )
  (let ((result ""))
  (display "[++] Executing e*Way external shutdown command notification
   function.")
  result
  )
))
```

ui-stdver-proc-outgoing

The function **ui-stdver-proc-outgoing** is used for sending a received message (Event) from e*Gate to an external system.

Syntax

```
(ui-stdver-proc-outgoing message-string)
```

Parameters

Parameter	Type	Description
message-string	string	The Event to be processed.

Return Values

The **ui-stdver-proc-outgoing** API returns one of the following values:

This value is returned …	if this occurs …
empty string	The Event is sent successfully.
RESEND	The Event is not sent successfully, and is immediately resent. The e*Way compares the number of retry attempts to the parameter **Max Resends per Message**, and does one of the following:
	• If the number of attempts does not exceed the maximum, the e*Way pauses the number of seconds specified by the **Resend Timeout** parameter, increments the "resend attempts" counter for that message, then repeats the attempt to send the message.
	• If the number of attempts exceeds the maximum, the function returns **false** and rolls back the message to the e*Gate IQ from which it was obtained.
CONNERR	There is a problem communicating with the external system. In this case, the following occurs:
	• The e*Way pauses the number of seconds specified by the Resend Timeout parameter.
	• The e*Way then calls the External Connection Establishment function according to the Down Timeout schedule, and rolls back the Event to the IQ from which it was obtained.

This value is returned …	if this occurs …
DATAERR	There is a problem with the Event data itself. In this case, the following occurs:
	• The e*Way pauses the number of seconds specified by the Resend Timeout parameter.
	• The e*Way then increments its "failed message" counter, and rolls back the Event to the IQ from which it was obtained. If the e*Way's journal is enabled, the Event is journaled.
A string other than the above	The function is unsupported. The e*Way creates an entry in the log file indicating that an attempt has been made to access an unsupported function.

Standard Implementation

The example below illustrates how **ui-stdver-proc-outgoing** is defined in *ui-stdver-eway-funcs.monk*.

```
(define ui-stdver-proc-outgoing
 (lambda ( message-string )
  (let ((result ""))
  (display "[++] Executing e*Way external process outgoing
          message function.")
  (display message-string)
  result
  )
))
```

ui-stdver-proc-outgoing-stub

The **ui-stdver-proc-outgoing-stub** function is used as a place holder for the function entry point for sending an Event received from e*Gate to the external system. If an interface is configured as an inbound only connection, this function should not be used. This function is used to catch configuration problems.

Syntax

```
(ui-stdver-proc-outgoing-stub message-string)
```

Parameters

Parameter	Type	Description
message-string	string	The Event to be processed.

Return Values

The **ui-stdver-proc-outgoing-stub** API returns one of the following values:

This value is returned …	if this occurs …
empty string	The Event is sent successfully.
RESEND	The Event is not sent successfully, and is immediately resent. The e*Way compares the number of retry attempts to the parameter **Max Resends per Message**, and does one of the following:
	• If the number of attempts does not exceed the maximum, the e*Way pauses the number of seconds specified by the **Resend Timeout** parameter, increments the "resend attempts" counter for that message, then repeats the attempt to send the message.
	• If the number of attempts exceeds the maximum, the function returns **false** and rolls back the message to the e*Gate IQ from which it was obtained.
CONNERR	There is a problem communicating with the external system. In this case, the following occurs:
	• The e*Way pauses the number of seconds specified by the Resend Timeout parameter.
	• The e*Way then calls the External Connection Establishment function according to the Down Timeout schedule, and rolls back the Event to the IQ from which it was obtained.

This value is returned …	if this occurs …
DATAERR	There is a problem with the Event data itself. In this case, the following occurs:
	• The e*Way pauses the number of seconds specified by the Resend Timeout parameter.
	• The e*Way then increments its "failed message" counter, and rolls back the Event to the IQ from which it was obtained. If the e*Way's journal is enabled, the Event is journaled.
A string other than the above	The function is unsupported. The e*Way creates an entry in the log file indicating that an attempt has been made to access an unsupported function.

Standard Implementation

The example below illustrates how **ui-stdver-proc-outgoing-stub** is defined in *ui-stdver-eway-funcs.monk*.

```
(define ui-stdver-proc-outgoing-stub
 (lambda ( message-string )
  (let ((result ""))
  (display "[++] Executing e*Way external process outgoing
           message function stub.")
  (display message-string)
  (event-send "ALERTCAT_OPERATIONAL" "ALERTSUBCAT_INTEREST"
      "ALERTINFO_NONE" "O" "Possible configuration error."
      (string-append "Default eway process outgoing msg
      function passed following message: " msg) O (list))
  result
  )
))
```

ui-poll-startup

The **ui-poll-startup** function is used for instance-specific function loads for the polling e*Way, and for invoking setup.

Syntax

```
(ui-poll-startup)
```

Parameters

Parameter	Type	Description
None		

Return Values

This value is returned …	if this occurs …
A string (other than FAILURE)	The function load and setup occurred successfully.
A FAILURE string	The function load and setup did not occur successfully.
	Note: *In this instance, the e*Way will shut down.*

Standard Implementation

The example below illustrates how **ui-poll-startup** is defined in *ui-stdver-eway-funcs.monk*.

```
(define ui-poll-startup
 (lambda (  )
  (let ((result "SUCCESS"))
   (display "[++] Executing e*Way external startup function.")
   (load "eiEvent.ssc")
   result
  )
 )
)
```

ui-poll

The **ui-poll** function is used for retrieving Events from the *ui_msg_detail* table in the e*Index database, and then transmitting those Events to external systems through e*Gate.

*Note: By default, **ui-poll** is configured to return a DATAERR when there is no data in* ui_msg_detail. *This causes the polling e*Way to shut down after a number of retries. You may need to modify this function so it returns NULL when there is no data in the outbound table.*

Syntax

```
(ui-poll)
```

Parameters

Parameter	Type	Description
None		

Return Values

This value is returned …	if this occurs …
empty string	The send occurs successfully, and the Event is sent to e*Gate.
CONNERR	Connection with the external is lost. In this case, the e*Way moves to a down state, and attempts to connect. Upon reconnection, this function is re-executed with the same Event.
DATAERR	There is a problem with the Event data itself. In this case, the e*Way pauses the number of seconds specified by the Resend Timeout parameter. The e*Way then increments its "failed message" counter, and rolls back the Event.

Standard Implementation

The example below illustrates how **ui-poll** is defined in *ui-stdver-eway-funcs.monk*. You may want to customize this function to suit the specific processing requirements for the data in the *ui_msg_detail* table of your database.

```
(define ui-poll
  (lambda ()
   (let ( (pollmsg ""))
      (try
       (display "[++] Executing e*Index poll function.\n")
        (set! pollmsg (ui-dequeue connection-handle))
     )
     (catch
       (otherwise
         (begin
           (if (db-alive connection-handle)
             (begin
              (set! pollmsg "DATAERR")
             )
             (begin
              (set! pollmsg "CONNERR")
             )
           )
         )
       )
     )
     (display (format "Returning: [%s]\n" pollmsg))
     pollmsg
    )
  )
)
```

ui-poll-pos-ack

The **ui-poll-pos-ack** function sends a positive acknowledgment to the polling e*Way to verify that an Event was received successfully by an external system. This function is also used for post processing after data is successfully sent to e*Gate. The default configuration for **ui-poll-pos-ack** includes a call to **ui-delete-queue-msg** to ensure than any Events that were successfully received by the external systems are removed from the outbound queue. Events to be removed are identified by their msg_id field, which corresponds to the ui_msg_header_id column of the *ui_msg_detail* table.

Syntax

```
(ui-poll-pos-ack message-string)
```

Parameters

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

Return Values

The **ui-poll-pos-ack** API returns one of the following values:

This value is returned …	if this occurs …
empty string	The positive acknowledgment is sent successfully. The e*Way is then able to proceed with the next request.
CONNERR	Connection with the external is lost.
	*Note: In this instance, the e*Way changes to a down state and attempts to reconnect. Upon reconnecting, the pos-ack function is re-executed.*

Standard Implementation

The example below illustrates how **ui-poll-pos-ack** is defined in *ui-stdver-eway-funcs.monk*.

```
(define ui-poll-pos-ack
 (lambda ( message-string )
  (let ((result "")(msg-id "")
        (input ($make-event-map eiEvent-delm eiEvent-struct)))
    (display "[++] Executing e*Way external positive acknowledgement
      function.")
    (if ($event-parse input message-string)
      (begin
       (set! msg-id (get ~input%eiEvent.EVNT.EVN.msg_id))
       (display (format "Deleting message ID [%s] from the queue\n" msg-id))
       (ui-delete-queue-msg connection-handle msg-id)
      )
    )
     result
   )
))
```

ui-poll-neg-ack

The **ui-poll-neg-ack** function sends a negative acknowledgment to the sending system to verify that an Event was not received successfully. This function is also used for post processing after an Event transmission fails. This function flags the failed Events in the *ui_msg_detail* table. Failed events are identified by their msg_id field, which corresponds to the ui_msg_header_id column of the *ui_msg_detail* table.

Syntax

```
(ui-poll-neg-ack message-string)
```

Parameters

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

Return Values

The **ui-poll-neg-ack** API returns one of the following values:

This value is returned …	if this occurs …
empty string	The negative acknowledgment is sent successfully.
CONNERR	Connection with the external is lost.
	*Note: In this instance, the e*Way changes to a down state and attempts to reconnect. Upon reconnecting, the neg-ack function is re-executed.*

Standard Implementation

The example below illustrates how **ui-poll-neg-ack** is defined in *ui-stdver-eway-funcs.monk*.

```
(define ui-poll-neg-ack
 (lambda ( message-string )
  (let ((result "")(msg-id "")
        (input ($make-event-map eiEvent-delm eiEvent-struct)))
    (display "[++] Executing e*Way external negative acknowledgement
      function.")
    (if ($event-parse input message-string)
      (begin
       (set! msg-id (get ~input%eiEvent.EVNT.EVN.msg_id))
       (display (format "Setting error status of message ID [%s]\n" msg-id))
       (ui-set-queue-id connection-handle msg-id "E")
    ) )
     result
)))
```

ui-stdver-data-exchg-stub

The **ui-stdver-data-exchg-stub** function is used as a placeholder for the function entry point for sending an Event from the external system to e*Gate. When the interface is configured as an outbound only connection, this function should not be called.

Syntax

```
(ui-stdver-data-exchg-stub)
```

Parameters

Parameter	Type	Description
None		

Return Values

A string

This value is returned …	if this occurs …
empty string	The send occurs successfully, and nothing is sent to e*Gate.
message string	The send occurs successfully, and the Event is sent to e*Gate.
CONNERR	Connection with the external is lost. In this case, the e*Way moves to a down state, and attempts to connect. Upon reconnection, this function is re-executed with the same Event.

Standard Implementation

The example below illustrates how **ui-stdver-data-exchg-stub** is defined in *ui-stdver-eway-funcs.monk*.

```
(define ui-stdver-data-exchg-stub
 (lambda (  )
  (let ((result ""))
  (display "[++] Executing e*Way external data exchange
           function stub.")
  (event-send "ALERTCAT_OPERATIONAL" "ALERTSUBCAT_INTEREST"
   "ALERTINFO_NONE" "O" "Possible configuration error." "Default eway
   data exchange function called." O (list))
  result
)))
```

e*Index Monk Functions

About this Chapter

Overview

This chapter presents background information you need to use the e*Index Monk functions in *ui-fns.monk* to reformat your data.

The following diagram illustrates the contents of each major topic in this chapter. For the page numbers on which specific topics appear, see the next page of this chapter.

About e*Index Monk Functions — Learn about the standard e*Index Monk functions provided by SeeBeyond

Monk Function Descriptions — Learn about each function provided in the standard e*Index Monk library

What's Inside

This chapter provides background information and instructions related to the topics listed below.

Learning About e*Index Monk Functions

Overview

This section of the chapter provides background information about the e*Index Monk functions that you can use to reformat your data. The functions are located in the files *ui-fns.monk*.

What are e*Index Monk Functions?

*e*Index Monk functions* are specialized commands that allow you to transform your data into a useable format through a Monk script. For example, if you are transferring data from a system that stores telephone numbers with any non-numeric characters, you can use a function named strip-phone to remove the unwanted characters before transferring the data. Use these functions to modify *ui-custom.monk*, which contains the commands that define the fields in your Monk lists such as **get-demographics**, **get-alias**, **get-transaction**, **get-phone**, and **get-address**. The Monk lists created by *ui_custom.monk* are passed through Monk APIs as parameters, so the list data must be in the correct format in order for Events to be processed correctly.

The Monk functions are defined in your *ui_fns.monk* file. Descriptions of these functions begin on page 5-5.

Can I Modify e*Index Monk Functions?

You can customize the Monk functions in *ui-fns.monk* to match the data requirements of your business. If necessary, you can also add new functions to *ui-fns.monk* to perform additional modifications to the data in the Monk lists. These commands are written in the Monk scripting language. For more information about Monk, see your *Monk Developer's Reference*.

What e*Index Monk Functions are Defined?

Several e*Index Monk functions are defined in the Monk library to help you tailor demographic data into the format required by your business systems. Use the table on the following page to identify each function and its purpose.

Table 4-1: Standard e*Index Monk Functions

Use this Monk function …	to perform this action …
strip-ssn	Remove any non-numeric characters from a social security number.
strip-phone	Remove any non-numeric characters from a telephone number.
filter-dob	Remove non-numeric characters from a date. This function does not remove dashes ("-") from a date.
filter-paren	Remove the parenthesis from the specified data string.
string-all-char?	Verify that a string consists only of a specified character. Use this function to determine if a data string consists solely of spaces.
convert-sp-nul-zero	Convert a string that contains spaces, or null or zero values into double quotes. This function does not alter an empty string.
convert-empty2quotes	Convert a string that contains spaces, or null or zero values into double quotes. This function converts an empty string to double quotes.
trim-lead-space	Remove any spaces at the beginning of a string.
ui-get-next-element	Returns the next element in a vector.
ui-has-next-element	Checks if there is a next element in a vector, and returns #t if a next element is found.

For More Information

Other e*Index publications may help you to learn how to perform tasks associated with using Monk functions.

To learn more about …	See …
The Monk programming language	Your *Monk Developer's Reference*
e*Index Monk APIs and Lists	Chapter 4 of this guide
e*Index Features and Functions	Your *e*Index Global Identifier User's Guide*

e*Index Monk Function Descriptions

Overview

This section of the chapter describes the Monk functions that are defined in *ui_fns.monk*. Use these functions to alter the data you are working with into a useable format.

strip-ssn

The **strip-ssn** function removes the non-numeric characters from a given social security number, and changes the format from xxx-yy-zzzz to xxxyyzzzz. This function also checks for invalid SSNs, such as 111-11-1111, 222-22-2222, 999-99-9999, and so on.

Syntax

```
(strip-ssn <ssn_no>)
```

Parameter

Parameter	Type	Description
ssn_no	string	The social security number to be stripped.

Return Value

The **strip-ssn** function returns one of the following values:

This value is returned …	if this occurs …
<empty string>	The SSN field is empty.
" "	The original SSN is non-numeric or is " ", or the SSN is less than 9 digits in length.
An SSN in the format **xxxyyzzzz**	The original SSN is in **xxx-yy-zzzz** or **xxxyyzzzz** format.

Example

The following example modifies the social security number in a demographic list by removing the dashes. If DEMO.SSN_number equals 948-88-8884, the return value is 948888884.

```
(define get-demographics
  (lambda (msg)
    (begin
     (list (get ~<msg>.DEMO.person_name.NM.last_name)
           (get …
           )
           (strip-ssn (get ~<msg>.DEMO.SSN_number))
           (…
           )
```

strip-phone

The **strip-phone** function removes the non-numeric characters from a phone number, and verifies that the number is valid. Invalid numbers include numbers such as (000)000-0000, (111)111-1111, 999-999-9999, and so on. Strip phone also verifies the length of the phone number.

Syntax

```
(strip-phone <phone_no>)
```

Parameters

Parameter	Type	Description
phone_no	string	The telephone number to be stripped.

Return Value

The **strip-phone** function returns one of the following values:

This value is returned …	if this occurs …
<empty string>	The telephone number field is empty.
An telephone number in the format **xxxyyyzzzz**	The original telephone number is in **xxxyyyzzzz**, **xxx-yyy-zzzz**, or (**xxx)yyy-zzzz** format.

Example

The following example removes any parenthesis or dashes from the telephone numbers included in the telephone list. If PH.phone_number equals (050)551-5551, then the return value is 0505515551.

```
(define get-phone
 (lambda (msg index)
    (list (get ~<msg>.DEMO.phone[<index>].PH.type)
          (strip_phone
           (get ~<msg>.DEMO.phone[<index>].PH.phone_number)
          )
          (get ~<msg>.DEMO.phone[<index>].PH.phone_ext)
      )
   )
)
```

filter-zip

The **filter-zip** function separates a zip code and zip extension when the original data includes both in the same field.

Syntax

```
(filter-zip  <arg1> <arg2>)
```

Parameters

Parameter	Type	Description
arg1	string	The zip code or zip code plus extension.
arg2	string	The value to be returned. Enter **zip** to return the zip code, or enter **ext** to return the zip code extension.

Return Value

The **filter-zip** function returns one of the following values:

This value is returned …	if this occurs …
" "	The zip code field does not contain a zip code, and the value you enter for <arg2> is **zip**.
" "	The zip code field does not contain a zip extension, and the value you enter for <arg2> is **ext**.
A zip code without the zip extension	The zip code field contains a zip code, and the value you enter for <arg2> is **zip**.
A zip extension without the zip code	The zip code field contains a zip extension, and the value you enter for <arg2> is **ext**.

Example

The following example separates the zip code and zip extension into two separate fields. In the original data, the zip code and extension were included together in one field. If the original zip code and extension is **58623-1825**, the return value for the first filter-zip function is **58623**, and the return value for the second filter-zip function is **1825**.

```
(define get-address
  (lambda (msg index)
   (list (get ~<msg>.DEMO.address[<index>].AD.type)
         (get...
         )
         (filter-zip (get ~<msg>.DEMO.address[<index>].AD.zip)
         )
         (filter-zip (get
~<msg>.DEMO.address[<index>].AD.zip_ext)
         )
         (get...
         )
   )
  )
)
```

filter-paren

The **filter-paren** function removes any parentheses contained in the given input string.

Syntax

```
(filter-paren <arg1>)
```

Parameters

Parameter	Type	Description
arg1	string	The data from which you want to remove the parenthesis.

Return Value

The **filter-paren** function returns one of the following values:

This value is returned …	if this occurs …
The original input string without the parenthesis	The original input string contains parenthesis.
The original input string	The original input string does not contain any parenthesis.

Example

The following example checks for and removes parenthesis in the last name field of the demographic list. For example, if the value of DEMO.person_name.NM.last_name is **(Thorenson)**, the return value is **Thorenson**.

```
(define get-demographics
  (lambda (msg)
    (begin
     (list (filter_paren
              (get ~<msg>.DEMO.person_name.NM.last_name)
            )
            (get …
            )
     )
    )
  )
)
```

string-all-char?

The **string-all-char?** function tests a string to see if it consists only of the input character. Typical usage of this function is to test if a string is all spaces.

Syntax

```
(string-all-char #\<arg1> <arg2>)
```

Parameters

Parameter	Type	Description
arg1	character	The character that you want the function to check for in the given string. You are verifying that the string consists only of this character. To check if the given string is all spaces, use the text **space** for arg1.
arg2	string	The data you want to verify.

Return Value

The **string-all-char?** function returns one of the following values:

This value is returned …	if this occurs …
#t	The input string consists only of the character you specify in \<arg1>.
#f	The input string includes characters other than the character you specify in \<arg1>.

Example

In the example below, **string-all-char?** is used to determine whether the phone number field is all spaces. If the field is all spaces, then **string-all-char?** returns #t. If the field is not all spaces, then **string-all-char?** returns #f.

```
(define get-phone
 (lambda (msg index)
    (list (get ~<msg>.DEMO.phone[<index>].PH.type)
          (string-all-char? #\space
                (get ~<msg>.DEMO.phone[<index>].PH.phone_number)
          )
          (get ~<msg>.DEMO.phone[<index>].PH.phone_ext)
      )
  )
)
```

convert-sp-nul-zero

The **convert-sp-nul-zero** function interprets spaces, null values, or zero characters within a field by translating the string into double quotes. Usage of this function is coordinated with the e*Index rule that double quotes passed into a field cause the field to be nullified.

Syntax

```
(convert-sp-nul-zero <arg1> <#t if check for spaces> <#t if
check for nulls> <#t if check for zeros>)
```

Parameters

Parameter	Type	Description
arg1	string	The data you want to check for spaces, and null or zero characters.
if check for spaces	Boolean	An indicator that specifies whether to check for spaces in the string. Enter **#t** to check for spaces; enter **#f** if you do not want to check for spaces.
if check for nulls	Boolean	An indicator that specifies whether to check for null values in the string. Enter **#t** to check for nulls; enter **#f** if you do not want to check for nulls.
if check for zeros	Boolean	An indicator that specifies whether to check for zero values in the string. Enter **#t** to check for zero values; enter **#f** if you do not want to check for zero values.

Return Value

The **convert-sp-nul-zero** function returns one of the following values:

This value is returned …	if this occurs …
The original input string	The input string does not contain the values you specify (space, null, or zero).
" "	The input string contains any of the values you specify (space, null, or zero).
<empty string>	The input string is an empty string.

Example

The following example checks the VIP field for any spaces, and if any are found, returns double quotes. If no spaces are found, then the return value is the original value of the field.

```
(define get-demographics
  (lambda (msg)
    (begin
     (list (get ~<msg>.DEMO.person_name.NM.last_name)
           (get …
           )
           (convert-sp-nul-zero (get ~<msg>.DEMO.vip) #t #f #f)
           (get …
           )
     )
    )
  )
)
```

convert-empty2quotes

The **convert-empty2quotes** function is identical to **convert-sp-nul-zero** (described above) except that it returns double quotes rather than an empty string when the input is an empty string.

Syntax

```
(convert-empty2quotes <arg1> <#t if check for spaces> <#t if
check for nulls> <#t if check for zeros>)
```

Parameters

Parameter	Type	Description
arg1	string	The data you want to check for spaces, null, or zero characters.
if check for spaces	Boolean	An indicator that specifies whether to check for spaces in the string. Enter **#t** to check for spaces; enter **#f** if you do not want to check for spaces.
if check for nulls	Boolean	An indicator that specifies whether to check for null values in the string. Enter **#t** to check for nulls; enter **#f** if you do not want to check for nulls.
if check for zeros	Boolean	An indicator that specifies whether to check for zero values in the string. Enter **#t** to check for zero values; enter **#f** if you do not want to check for zero values.

Return Value

The **convert-empty2quotes** function returns one of the following values:

This value is returned …	if this occurs …
The original input string	The input string does not contain the values you specify (space, null, or zero).
" "	The input string contains any of the values you specify (space, null, or zero).
" "	The input string is an empty string.

Example

The following example checks the VIP field to see if it is an empty string, or contains null characters. If it is an empty string or if there are any null values in the string, then the output is double quotes. If the string is **VIP**, then the output is **VIP**.

```
(define get-demographics
  (lambda (msg)
    (begin
     (list (get ~<msg>.DEMO.person_name.NM.last_name)
           (get …
           )
           (convert-empty2quotes (get ~<msg>.DEMO.vip) #f #t #f)
           (get …
           )

     )
    )
  )
)
```

trim-lead-space

The **trim-lead-space** function returns the input string with any leading spaces removed. All other spaces are untouched.

Syntax

```
(trim-lead-space <arg1>)
```

Parameters

Parameter	Type	Description
arg1	string	The string from which you want to remove any spaces that exist before the text.

Return Value

The **trim-lead-space** function returns one of the following values:

This value is returned …	if this occurs …
The original input string	The input string does not contain any spaces at the beginning of the text.
The original string with no leading spaces	The input string contains spaces at the beginning of the text.

Example

In the example below, any leading spaces that exist before the last name in the demographic list are removed. If DEMO.person_name.NM.last_name is "   **Johnson**", then the return value is "**Johnson**".

```
(define get-demographics
  (lambda (msg)
    (begin
     (list (trim-lead-space
             (get ~<msg>.DEMO.person_name.NM.last_name)
            (get …
            )
            (convert-empty2quotes (get ~<msg>.DEMO.vip) #f #t #f)
            (get …
            )
      )
    )
  )
)
```

ui-get-next-element

The **ui-get-next-element** function retrieves the next element in a vector. Use this function in conjunction with **ui-has-next-element** to scroll through the local IDs returned by the APIs **ui-lookup-local-id**, **ui-search-local-id**, and **ui-get-all-local-id**.

Syntax

```
(ui-get-next-element vector-obj)
```

Parameter

Parameter	Type	Description
vector-obj	vector	A vector containing the elements through which you want to scroll.

Return Value

The **ui-get-next-element** function returns one of the following values:

This value is returned …	if this occurs …
The next element in the vector	The next element exists in the vector.
Exception	There is no next element to retrieve. This also causes an abort.

Example

The following example searches for active local ID records associated with the UID represented by the variable **uid**. If local ID records are returned, it calls **ui-has-next-element** and **ui-get-next-element** to scroll through the records.

```
...
  (set! lids (ui-search-local-id uid system "A"))
  (if lids
    (begin
      (do
      ((i O (+ i 1)))
      ((not (ui-has-next-element lids)))
      (display (format "%d: %a\n" i (ui-get-next-element lids)))
      )
    )
  )
  (display "No LIDs\n")
   ...
```

ui-has-next-element

The **ui-has-next-element** function checks whether there is a next element in a vector, and returns Boolean true if a next element is found. Use this function in conjunction with **ui-get-next-element** to scroll through the local IDs returned by the APIs **ui-lookup-local-id**, **ui-search-local-id**, and **ui-get-all-local-id**.

Syntax

```
(ui-has-next-element vector-obj)
```

Parameter

Parameter	Type	Description
vector-obj	vector	A vector containing the elements through which you want to scroll.

Return Value

The **ui-has-next-element** function returns one of the following values:

This value is returned …	if this occurs …
#t	The vector list contains another element.
#f	The vector list does not contain any more elements.

Example

The following example searches for active local ID records associated with the UID represented by the variable **uid**. If local ID records are returned, it calls **ui-has-next-element** and **ui-get-next-element** to scroll through the records.

```
...
  (set! lids (ui-search-local-id uid system "A"))
  (if lids
    (begin
      (do
      ((i 0 (+ i 1)))
      ((not (ui-has-next-element lids)))
      (display (format "%d: %a\n" i (ui-get-next-element lids)))
      )
    )
  )
  (display "No LIDs\n")
   ...
```