Guide to the Application Event System
Copyright © 2010 Infor

All rights reserved. The word and design marks set forth herein are trademarks and/or registered trademarks of Infor and/or related affiliates and subsidiaries. All rights reserved. All other trademarks listed herein are the property of their respective owners. www.infor.com.

Important Notices

The material contained in this publication (including any supplementary information) constitutes and contains confidential and proprietary information of Infor.

By gaining access to the attached, you acknowledge and agree that the material (including any modification, translation or adaptation of the material) and all copyright, trade secrets and all other right, title and interest therein, are the sole property of Infor and that you shall not gain right, title or interest in the material (including any modification, translation or adaptation of the material) by virtue of your review thereof other than the non-exclusive right to use the material solely in connection with and the furtherance of your license and use of software made available to your company from Infor pursuant to a separate agreement ("Purpose").

In addition, by accessing the enclosed material, you acknowledge and agree that you are required to maintain such material in strict confidence and that your use of such material is limited to the Purpose described above.

Although Infor has taken due care to ensure that the material included in this publication is accurate and complete, Infor cannot warrant that the information contained in this publication is complete, does not contain typographical or other errors, or will meet your specific requirements. As such, Infor does not assume and hereby disclaims all liability, consequential or otherwise, for any loss or damage to any person or entity which is caused by or relates to errors or omissions in this publication (including any supplementary information), whether such errors or omissions result from negligence, accident or any other cause.

Publication Information

Release: SyteLine 8.02
Publication Date: June 2010

Changes made to this document since the previous version

<table>
<thead>
<tr>
<th>Date</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>June 2010</td>
<td></td>
<td>Many revisions for SyteLine 8.02.</td>
</tr>
</tbody>
</table>
## Contents

### About This Guide
- Additional Infor ERP SyteLine Documentation
- Prerequisite Knowledge
- Contacting Infor

### About the Application Event System
- Basic Components of the Application Event System
- Advantages of the Application Event System
- Examples of Ways to Use the Application Event System
  - Example 1: Sending a Notification When a Record Is Added
  - Example 2: Getting Approval for a Credit Limit Change
  - Example 3: Complex Approval of a Purchase Order Status Change
  - Example 4: Automatically Shipping a Customer Order
- Workflow Event Handlers

### How the Application Event System Works
- How Events Are Handled
  - When Events Can Be Generated
  - Where Events Can Be Generated
- Controlling the Sequence of Event Handlers and Actions
- Restricting Which Handlers Run
  - Using the Session Access As Form
  - Using Event Handler Settings
- Synchronicity
  - Synchronous Events
  - Asynchronous Events
  - Event Handlers
- Suspension
  - When an Event Is Suspended
  - When an Event Is Not Suspended
- Payload
- Adjournment and Resumption
- Success, Failure, and Retries
  - Success
  - Failure
## Event Messages

- Event Message-Related Forms
  - The Inbox Form
  - The Saved Messages Form
  - The Send Message Form

- Send E-mail to External E-mail Inbox for Prompts

- Prompts and Responses
  - Voting Rules
  - Dealing with Indeterminate Voting Results
  - Quorums

## Sample Scenarios

- Sending Notifications
- Requesting Approvals
- Modifying Records
- Voting
- Localizing Message Contents
- More Advanced Scenarios

## Reference Tables

- Firing Events
- Summary of Synchronous Functionality
- Framework Events
- Framework Event Attributes
- Application Events
- Event Action Types
- Event Action Parameters
- Expression Functions
  - Standard Functions
  - Pre-parser Functions
- Expression Operators

## Expression Grammar

- Restrictions
- Start Symbol
- Character Sets
- Terminals
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rules</td>
<td>177</td>
</tr>
<tr>
<td>Variable, Constant, and Event Parameter References</td>
<td>177</td>
</tr>
<tr>
<td>Expressions</td>
<td>177</td>
</tr>
<tr>
<td>Boolean Rules</td>
<td>177</td>
</tr>
<tr>
<td>Typeless Rules</td>
<td>180</td>
</tr>
<tr>
<td>String Rules</td>
<td>182</td>
</tr>
<tr>
<td>Numeric Rules</td>
<td>185</td>
</tr>
<tr>
<td>Date Rules</td>
<td>188</td>
</tr>
<tr>
<td>Restricted Arguments</td>
<td>195</td>
</tr>
<tr>
<td>Keyword Paren Lists</td>
<td>195</td>
</tr>
<tr>
<td>Enumerations</td>
<td>196</td>
</tr>
<tr>
<td>Sample Stored Procedures</td>
<td>199</td>
</tr>
<tr>
<td>Passing Parameters to a Synchronous Event</td>
<td>199</td>
</tr>
<tr>
<td>Calling a Synchronous Event within a Transaction and Handling Failure</td>
<td>199</td>
</tr>
<tr>
<td>Synchronization of Metadata</td>
<td>201</td>
</tr>
<tr>
<td>Overview and Rationale</td>
<td>201</td>
</tr>
<tr>
<td>The Inherent Hierarchy of Metadata</td>
<td>202</td>
</tr>
<tr>
<td>Maintaining Handler IDs Through Metadata Updates</td>
<td>203</td>
</tr>
<tr>
<td>Protecting Running Events from Metadata Changes</td>
<td>203</td>
</tr>
<tr>
<td>Detailed Examples</td>
<td>204</td>
</tr>
<tr>
<td>Using Specific Chronology</td>
<td>204</td>
</tr>
<tr>
<td>Using Non-specific Chronology</td>
<td>207</td>
</tr>
<tr>
<td>Performing Upgrades</td>
<td>216</td>
</tr>
<tr>
<td>Overriding Others' Handlers</td>
<td>221</td>
</tr>
<tr>
<td>Non-exclusive Overrides</td>
<td>221</td>
</tr>
<tr>
<td>Exclusive Overrides</td>
<td>223</td>
</tr>
<tr>
<td>Disabling the Ability to Override</td>
<td>225</td>
</tr>
<tr>
<td>Dealing with Obsolete Handlers</td>
<td>226</td>
</tr>
<tr>
<td>Event Flow Options</td>
<td>229</td>
</tr>
<tr>
<td>Glossary</td>
<td>231</td>
</tr>
<tr>
<td>Index</td>
<td>237</td>
</tr>
</tbody>
</table>
About This Guide

This guide describes the application event system, which developers and system administrators can use to customize their systems. In essence, this new functionality—among other things—replaces and expands upon the functionalities provided by Workflow in earlier versions of SyteLine. With it, developers and system administrators can create events and event handlers to automate many tasks that would otherwise have to be done manually, with less effort and less risk of error.

This guide is provided in PDF format, but it has been optimized for online use, so the links in this guide work. You can also print all or part of it, depending on your preferences.

This version is current with WinStudio 6.1.0.

Additional Infor ERP SyteLine Documentation

In addition to this guide, the Infor ERP SyteLine online help includes a significant number of online Help topics designed to help you use this new functionality most effectively.

The most current version of all documentation is available on the Infor support web pages (see Contacting Infor Support).

Prerequisite Knowledge

This publication assumes that you have at least some knowledge in the following areas:

- System architecture and function (including the tier structure) for your Infor system
- SQL Server database

If you do not have this prerequisite knowledge, we advise you to obtain it before going any further. For the most up-to-date list of software and hardware requirements for Infor products, see the documentation for your system.

You can also refer to the Guide to Technology on our support website. This document lists typical system administration tasks you should be familiar with before attempting to install and administer Infor products.
Contacting Infor

Contacting Infor Support

If you cannot find the answer to a question in this document or in the product’s online help, contact Infor Global Support at our support website our support website. Simply log in and select your product area. From this site, you can do the following:

- Gain easy access to critical support resources like the Knowledge Center, software updates, and release notes.
- Obtain documentation for your product.
- Log and track incidents.
- Access the tools you need to keep your software running efficiently.
- Link to additional Infor resources.

Planning Your Communication

To make sure the correct analyst is assigned to your case and to expedite the resolution of your questions, please have the following information available when you call us:

- Your company name and phone number
- Infor ERP SyteLine version release and point release
- Database software version and release, if applicable
- Platform or environment (Example: Windows® 2003)
- Functional area (Examples: Production, Administration, etc.)
- What you were doing (Example: Creating an event handler)
- What type of data you were accessing or trying to access (Example: Customer data)
- If you received an error message, the full message text and error number
- If you are calling back on an existing case, the case number

Signing Up for Support

If you are not currently on support and would like more information on your support options, please call your customer account representative. If you are not sure who your account representative is, contact Infor Customer Service.
The *application event system* is a powerful and flexible set of tools designed to help extend and customize how the system works. You can use these tools to define and monitor events across the system.

The following topics will help you get a good understanding of the concepts involved in this system.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Components of the Application Event System</td>
<td>10</td>
</tr>
<tr>
<td>Advantages of the Application Event System</td>
<td>13</td>
</tr>
<tr>
<td>Examples of Ways to Use the Application Event System</td>
<td>14</td>
</tr>
</tbody>
</table>
Basic Components of the Application Event System

In its most basic form, the application event system consists of three interrelated components:

- **Events** are uniquely named incidents that can occur during the use of an application. Events can have multiple triggers and can be generated by user actions, conditions in a database, other events, or other situations.

- **Event handlers** consist of data that specifies:
  - The event to which it is to respond
  - Any conditions, situations, or attributes that determine when and why the handler executes
  - One or more event actions to take place during the handler’s execution

Each event can have multiple handlers but each handler can be associated with only one event.

- **Event actions** consists of data that specifies the individual tasks or bits of work that are performed by the event handler. Each event handler must have at least one action and can have multiple actions.

For information about how these basic constituents work together, see Chapter 2, [How the Application Event System Works](#).

The following diagram is an example of one possible set of events, event handlers, and event actions.
For example, you might want the system to automatically notify you whenever someone adds a customer order to the system and to request your approval if the order is $1000 or more. In this example:

- The event in this case gets triggered whenever someone enters a new customer order.
- The event and the situation that tells the event handler to run are set up as part of the event handler definition.
- The event action that you want to take place is a notification that an order has been placed. You also want that event action to request approval for orders $1000 or more.
- Only a single event handler is needed here. But that event handler requires several actions to complete all the requirements:
  - The first action is to check the amount of the order and determine, on the basis of the amount, what additional actions are required. If the order is $1000 or more, then it must direct the flow to another action that requests a manager’s approval. If the order is less than $1000, then it directs the flow to another action that simply sends out a notification that the order has been placed.
  - The second action is used only if the order is $1000 or more. If this action is used, the system generates a prompt message to the manager, requesting approval for the order. The system then waits until the manager responds. If the manager approves the order, the system then proceeds to the next action. If the manager does not approve the order, the event handler fails and no further action is taken. (Though, as good general practice, you should provide for this eventuality as well; but, in the interest of keeping this example simple, we will not provide for the case where the order is not approved.)
  - The third action is used only if the order is less than $1000. This action sends out a notification to let the manager know when an order has been completed.
Conceptually, the event and associated handlers might look like this:

**Event:** OrderNotification

This event is generated whenever a customer order is added.

**Event Handler**

- **Event Action 1:** The system checks to see if the amount of the order is greater than $1000.
  - If the amount is $1000 or more, go to Event Action 2.
  - If the amount is less than $1000, go to Event Action 3.

- **Event Action 2:** The system sends a message requesting your approval. The system then waits for your approval before moving on.

- **Event Action 3:** The system sends a message to your Inbox (if applicable, after approval), notifying you that the transaction has been completed.

For more examples, including the processes to create them, see Appendix A, Sample Scenarios.
Advantages of the Application Event System

Events and event handlers are tagged at the time of their creation with a special identifier (Access As) that prevents them from being modified or deleted by other development organizations. Among other things, this prevents your custom events and handlers from being overwritten when Infor or another developer issues updates to their events and handlers.

The application event system provides an infrastructure that allows both the framework and application code to generate events, and for developers and system administrators to create handlers for those events. Event handlers can augment or replace the default framework or application behavior associated with the event. You can design event handlers so that maximal work can be performed without requiring new procedural code.

This is possible because event handlers are defined using metadata. Metadata, in this context, refers to the practice of using uncompiled code and information about data formats that are interpreted during run time, rather than compiled code (called here "procedural code").

For more information about the components and elements of the application event system, see Chapter 3, Designing and Using Events and Handlers.

The application event system offers several significant advantages over hard-coded systems:

- The application event system provides the power and flexibility to customize the way the system performs without having to modify the basic code. Business processes are, therefore, "softer" and easier to modify. You can modify application processes and policies without having to directly modify the application code and, in many cases, without having to write any procedural code at all. This means that the amount of procedural code required to implement functionality can be greatly reduced.
- You can use your event handlers with events created by others to gain control of the application flow and take appropriate action, rather than being forced to call into the application using APIs.
- Because event handlers are defined using metadata, there is no upgrade problem, and no collision problem if other developers also have event handlers for the same event.
Examples of Ways to Use the Application Event System

The following subsections offer examples of how to use the event system, including explanations of how this can benefit you. For more sample scenarios, including the code required to implement them, see Appendix A, Sample Scenarios.

Example 1: Sending a Notification When a Record Is Added

Suppose you have a sales manager who wants to be notified whenever someone adds a new customer to the system. You could rely on personnel compliance by anyone who has the ability to add customers to the system and trust them to remember to send the sales manager an e-mail.

You could, however, use the application event system instead to automate compliance with this request. You can set up an event that is generated whenever anyone adds a new customer into the system, and use event handlers and actions to automatically generate a notification that is sent to the sales manager.

For a more detailed example, including the procedure to create and use the required handlers, see Sending Notifications on page 77.

Example 2: Getting Approval for a Credit Limit Change

Suppose your company requires that any change to a customer’s credit limit be approved by a designated credit manager. You could require any customer representative to contact that credit manager whenever a change to a credit limit is requested.

However, you could instead use the application event system to automatically send the credit manager a message requesting approval of the change. To speed up the process, you also send an e-mail to the credit manager with links the manager can click to approve or deny the request.

When the credit manager responds to the message and approves the request, the system can automatically change the credit limit amount.

For a more detailed example, including the procedure to create and use the required handlers, see Requesting Approvals on page 93.

Example 3: Complex Approval of a Purchase Order Status Change

Suppose your company has a business process in place that:

- Requests approval from a purchasing manager when the status of any purchase order (PO) is changed to **Ordered**.
- If the purchasing manager approves the request and the total cost of the PO exceeds a certain amount, requests consensus from a group of higher level managers on whether the order should be sent to the vendor.
- If those managers also approve and the total cost of the PO exceeds another, higher amount, requests unanimous approval from a group of top-level directors.
- If the request is approved at all the required levels for the total cost of the PO, the transaction is approved and completed.
As with the previous two examples, this process could all be taken care of manually, involving actions by many employees. However, the risk of something getting missed somewhere increases with each approval required.

So, you could use events and handlers to automate this entire process, evaluating the PO at each step of the way, requesting only the necessary approvals, and completing the transaction.

Example 4: Automatically Shipping a Customer Order

Suppose you have a system that, by default, requires someone to manually set the system to ship a customer order line when the status of the line is set to **Filled**. This can cause delays in orders, if/when the responsible employee does not set the order to ship in a timely manner.

You can use the application event system to "cut out the middle-man" and automatically set the order to ship as soon as the status is set to **Filled**.

Workflow Event Handlers

The application event system is extremely powerful and flexible. However, for some non-developer users, the system can be somewhat overwhelming at first. To help you with the learning curve, Infor provides a set of predefined workflow event handlers. These handlers are almost ready to use; in most cases, all you need to do is define the user names or e-mail addresses of the people who should be notified when a certain event happens, and activate the handler.

For example, if a customer's credit limit is changed, the "Customer Credit Limit Change and Approval" workflow event handler can automatically send an e-mail to the Accounts Receivable manager and request approval of the change. To set this up, just specify the manager's e-mail address and activate the handler on the Workflow Event Handler Activation form.
Events are generated (or triggered) in response to an action or condition that occurs in the system. When the event is generated, it can execute one or more event handlers with the event actions associated with that event handler.

This section includes the following major topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>How Events Are Handled</td>
<td>18</td>
</tr>
<tr>
<td>Controlling the Sequence of Event Handlers and Actions</td>
<td>19</td>
</tr>
<tr>
<td>Restricting Which Handlers Run</td>
<td>20</td>
</tr>
<tr>
<td>Synchronicity</td>
<td>22</td>
</tr>
<tr>
<td>Suspension</td>
<td>24</td>
</tr>
<tr>
<td>Adjournment and Resumption</td>
<td>27</td>
</tr>
<tr>
<td>Success, Failure, and Retries</td>
<td>28</td>
</tr>
<tr>
<td>Transactions</td>
<td>31</td>
</tr>
<tr>
<td>The Framework Event Service</td>
<td>34</td>
</tr>
<tr>
<td>Event and Event Handler Revisions</td>
<td>36</td>
</tr>
</tbody>
</table>
How Events Are Handled

Essentially, when an event is generated, it requires an event handler to do something in response. Otherwise, the event serves no practical purpose. That is, if there is no handler for an event, the event actually accomplishes nothing when it is generated.

TIP: That does not mean that an event should never exist unless it has an event handler. Infor includes some framework events as part of the system. Infor does not include handlers for these events. These events were created to provide events that other developers can use with their custom event handlers.

When Events Can Be Generated

An event can be generated when:

- A system user performs a particular action, perhaps only on a given form and/or when a particular business process is involved.
- A database calculation is performed, perhaps resulting in a certain value.
- Another event results in generating this event.
- A certain amount of time has passed.

The following are all examples of situations and conditions that can trigger events:

- A sales representative saves a record in the Customers form.
- A manager changes the credit hold status of a customer.
- A factory manager adds a new item to the list of those being manufactured in that facility.
- The first day of each month arrives. (An event can be used to generate a monthly report, for instance.)
- The quantity-on-hand of a particular item becomes less than zero.

Where Events Can Be Generated

In the application event system, events can be generated from any tier.

- In the client tier, the event can be generated by using a form that has a form event handler with a response type of Generate Application Event.
- In the middle tier, the event can be generated by invoking the appropriate .NET method.
- In the database tier, the event can be generated by using the appropriate stored procedure.
- In any tier, an event can generate another event by using the GenerateEvent action type.

For details on how to generate an event from any of these locations, see Firing Events on page 126.
Controlling the Sequence of Event Handlers and Actions

Two types of settings control the order in which event handlers and their actions execute:

- Each event handler has a handler sequence number. Handlers execute in the order of their sequence numbers.

  To modify the flow and change handler sequence order, you can use the Keep With and Chronology options on the Event Handlers form, or the Event Handler Sequence form. For more information, see About Event Handling and Order on page 55.

- The event actions associated with each event handler also have their own action sequence numbers. These execute in numeric order, unless:
  - You use the Initial Action option on the Event Handlers form to designate that a particular action should execute first.
  - You use certain action types to modify the flow.

For more information, see About Event Actions on page 44.
Restricting Which Handlers Run

There might be times when you want or need to disable the event handlers created by one or more development organizations, including yours, at least temporarily. This would occur typically when you are troubleshooting problems with the event system. The system offers two basic ways to disable event handlers.

Using the Session Access As Form

When you want to disable (or enable) all the event handlers created by a particular developing organization all at once, you can use the Session Access As form to accomplish this.

**TIP:** An alternate way to accomplish the same thing for individual handlers with your Access As value is to open the Event Handler form and for each event handler that you do not want to include in testing or debugging, clear the Active check box.

**An Example**

Suppose, for example, that a customer is having a problem that he suspects is being caused by something he did in the event system, but he is not sure what. He places a call to Infor technical support, and the technical support representative wants to verify that the customer’s custom event handlers are not causing the problem.

In this case, the technical support representative might ask the customer to temporarily disable all custom event handlers so that the operation can be tested with only standard functionality in place.

The technical support representative might instruct the customer to use the Session Access As form to perform either of the following actions:

- In the **Include Access As** field, enter `Core,BaseAppAccessAs` where `BaseAppAccessAs` is the Access As identifier associated with the base application installed on the system.
  
  With this setting, only the Infor core (framework) and base application event handlers will execute. Custom event handlers created by the end-customer will not execute. For more information, see Session Access As Form Options below.

- Leave the **Exclude Access As** field blank, and select the Exclude Blank Access As check box.
  
  This option allows all Infor and business partners event handlers to operate. Only the customer’s event handlers are ignored. For more information, see Session Access As Form Options below.

**Session Access As Form Options**

To disable or enable event handlers using the Session Access As form, use any of the following options:

**NOTE:** You are not obligated to use both the Include Access As and the Exclude Access As fields. You can, however, use any combination of the options available on this form.
With the **Session Access As** form open, in the **Include Access As** field, enter the Access As identifiers for event handlers that are to be recognized during this session.

To include multiple Access As identifiers, list them separated only by commas (no spaces).

If this field is left blank, the system can recognize and execute all event handlers.

- In the **Exclude Access As** field, enter the Access As identifiers for event handlers that are to be ignored during this session.

To exclude multiple Access As identifiers, list them separated only by commas (no spaces).

If this field is left blank, the system can recognize and execute all event handlers. The exception to this occurs only if the **Exclude Blank Access As** check box is selected. In this case, all event handlers are recognized *except* event handlers for which the Access As identifier is null (blank).

- To exclude those event handlers that have a blank Access As identifier, select the **Exclude Blank Access As** check box.

### Using Event Handler Settings

When you want to disable only certain event handlers temporarily, use the **Active** check box on the **Event Handlers** form.

You can only disable (or enable) event handlers that have the same Access As identifier that you have. You cannot, for instance, use this technique to disable **Core** event handlers.
Synchronicity

Synchronous Events

Events are synchronous when they are generated by:

- Core (framework) events
- Calling the FireApplicationEvent() .NET method with the Synchronous parameter set to True
- Calling the dbo.FireEventSp stored procedure
- Using a form event handler with a response type of Generate Application Event and the Synchronous option selected
- Using an event action of the type Generate Event and a parameter of Synchronous(True)

Unless the event is destined for suspension, the expectation is that the event will complete synchronously. Therefore, its synchronous event handlers execute in sequence in the same thread that generated it and block that thread until they have all been executed or until a handler exits with a failure status.

Asynchronous Events

Events are asynchronous when they are generated by:

- Calling the FireApplicationEvent() .NET method with the Synchronous parameter set to False
- Calling the dbo.PostEventSp stored procedure
- Using a form event handler with a response type of Generate Application Event and the Synchronous option cleared
- Using an event action of the type Generate Event and a parameter of Synchronous with a value of False

This means that the event runs in a different thread than the one that posted it, usually on a utility server. In this case, the event does not block the generating thread, running independently of it. As soon as an acknowledgment is received that the event was successfully queued, the thread that generated the event continues on.

For more information, see The Framework Event Service on page 34.

Event Handlers

A synchronous event handler is one that must complete before the system continues to the next event handler in a sequence. For the entire event to be handled successfully, all synchronous handlers in the sequence must complete successfully. The exception to this rule is that, if one event handler fails, other than for an illegal operation, and the system is set to ignore failures for that handler, the system can continue to the next synchronous event handler. Otherwise, the system returns a failure error, and no more event handlers in the sequence execute.

By contrast, an asynchronous event handler runs independently of other event handlers.
Event handlers can be designated as synchronous or asynchronous at the time they are created, using the **Synchronous** check box on the **Event Handlers** form.

Any event, either synchronous or asynchronous, can execute an asynchronous event handler when execution reaches an event handler designated as an asynchronous event handler; that is, for which the **Synchronous** check box is cleared. At this point, the following occurs:

- The system sends the asynchronous event handler to the event handler queue.
  
  Called queueing, the system does this by means of the PostEventHandlerSp stored procedure, to which it passes the configuration name from the event state.

- If queueing was successful, the event's thread continues on to the next event handler or, if no subsequent handlers are defined, completes the event.

- If queueing was unsuccessful, the event stops with a failure condition.

For a detailed view of synchronicity functionality, see Summary of Synchronous Functionality on page 127.
Suspension

Suspension occurs when a requested operation is sent to the event system for completion at a later time. This occurs when the event handler has the **Suspend** option selected and contains an adjourning action.

Suspension is possible only with certain framework events. You cannot create custom events that can be suspended. Currently, there are only three events that can be suspended:

- IdoOnItemInsert
- IdoOnItemUpdate
- IdoOnItemDelete

Suspension occurs when the system generates an event which:

- Is one of these three framework events that can be suspended.
- Has the **Suspend** check box selected (on the form) for at least one handler that applies to the generated event’s object and initiator.

When an Event Is Suspended

When both of the above conditions are met, the system passes control of the requested update (insertion/update/deletion) to the event system. The event system then uses the following process to try to make sure the event handlers can all execute successfully before actually committing the system (and the data) to execution of the event.

**Suspend-validating Mode**

When an event is suspended, the system does the following, in what is known as **suspend-validating mode**:

1. Begins a database transaction.
2. Performs an update to the database, to validate pertinent data against any SQL constraints or triggers.
3. Validates the data by running all effective event handlers that are synchronous and not suspended in the current transaction.

This involves both application and event system changes.

- At the beginning of this process, in a separate event transaction, the system copies the event handler to a new revision, if necessary (for information about revisions, see [Event and Event Handler Revisions](#) on page 36.

Both the **suspend-validating mode** and the **suspend-committing mode** (see following section) must use the same metadata. However, the validating transaction will be rolled back, and the master transaction *might* be rolled back, so the current event handler revision must be available, in case someone edits the handlers between execution of the two modes.

- During this process, a non-output event parameter named **Suspend_ValidatingMode** with a value of 1 is made available to handlers.
TIP: You can test this on any synchronous, non-suspended handlers that you do not want to execute at this time (perhaps because its actions would irreversibly affect something not controlled by our database transaction), using the expression
\[
\text{CONDITION( } E(\text{Suspend_VvalidatingMode})<>1)\text{).}
\]

NOTE: At this point, the event system treats any failure as an error and exits the execution of the event.

4. Rolls back the transaction.

5. Places a task on the event queue to run all effective handlers in suspend-committing mode (see Suspend-committing Mode below).

   If the collection update is rolled back due to a later error, this suspension is also rolled back with it, resulting in a state identical to one in which this update was never requested.

   When the event service picks up the queued event, it knows that a suspension is in effect by checking the EventQueue.Suspend attribute.

6. The system marks the record involved in the update/deletion request to have the InWorkflow property set to 1. This prevents any user from attempting another change to the suspended record. Note that suspended insertions do not reach the database at all unless and until the suspended event finishes successfully.

7. The thread in which the event was generated continues on as if the change request succeeded.

   This occurs even if the thread originated from an event handler executing an UpdateCollection or ExecuteIDORequest type event action. In other words, suspension affects only one level of event execution.

Suspend-committing Mode

Once the suspended event completes the suspend-validating mode successfully and is placed on the event queue, the system processes the event using the following rules. This is known as the suspend-committing mode.

- Any failure or error condition prevents further event handlers from executing and prevents the standard processing (that is, the requested insertion, update, or deletion) from occurring. However, the InWorkflow attribute is cleared.

- After each event handler finishes successfully, execution proceeds to the next event handler.

- The system queues any asynchronous event handler, and execution proceeds to the next event handler.

- When the last event handler finishes or is queued successfully, the following occurs:
  - The standard processing is committed.
    - That is, the requested insertion, update, or deletion occurs.
      - For an update request, this also includes clearing the InWorkflow attribute.
  - The event finishes.
When an Event Is Not Suspended

When an event that can be suspended is not suspended (that is, no event handler for that event that applies to that event’s object and initiator has the Suspend check box selected), the system executes the event using the following process:

1. The system performs a database update.
2. The event system executes all effective event handlers, returning either success or failure.
3. The system handles failures as errors and exits the event.

**NOTE:** Each insertion/update/deletion request in the process—which could be from an external XML request or from a Save action in WinStudio—participates in the overall transaction. In this case, all complete or nothing completes. So, each handler chain (the same chain executes for each insertion/update/deletion in the group) also participates in that overall transaction. A failure of any of those handler chains results in the entire transaction being rolled back.

Payload

In either of the above cases, an event that can be suspended carries a payload that represents, and that allows handlers to access and modify, the requested change (that is, insertion, update, or deletion).

Each named property value that forms part of the change is passed into and out of the event system as a parameter with a name of the form "Row.Property", where Property is the name of the IDO property. Modified property values are accompanied by another parameter with a name of the form "Row.Property.Modified" and a value of “1”.

During notify and prompt actions, these parameters

- are temporarily converted to event variables, which overrides any value that may have been set in preceding actions or specified on the Event Variable Groups form
- are available for display (and update, for prompts) on the Variables tab of the Inbox form (and for display at the end of an external e-mail prompt), and
- are then converted back to parameters for subsequent actions.

During other actions such as Set Values, these parameters can be modified to affect the final result of the change when it is applied to the database. This can be done by using the following syntax:

- SETPROPVALUES("Property"=expression); if the new value is different than the original framework event parameter received by the event, the PROPERTYMODIFIED(PropertyName) framework event parameter is also set to TRUE automatically.
- SETPARMVALUES(Row.Property=expression, Row.Property.Modified="1"); note that you should also indicate to the framework that the property is now modified, as shown.
Adjournment and Resumption

Some chains of event actions might need to be temporarily stopped, while waiting for some condition to be met. For instance, a customer order might need a manager’s approval before it can be entered into the system. In this case, the system temporarily halts the execution of the event handler chain until the condition is met; for instance, until the manager responds with an approval (or not). This is known as adjournment.

Certain event actions, called *adjourning actions*, must wait for an external stimulus before the event handler can proceed with the next action. An event handler containing such an action must be one of the following:

- Asynchronous
- Part of a framework event that can be suspended and is marked to suspend (see Suspension on page 24)

When execution reaches an adjourning action, the event handler state is set to retest or time out at a specified time. At this point, the event becomes an adjourned event.

The event service then processes it at the next opportunity after the Time Out and/or Retest time. This is called *resumption*. 
Success, Failure, and Retries

Each event handler can exit with a status of **Success** or **Failure**. When an event handler fails, if it is not set to ignore failures, and if it has not attempted an illegal operation, the system can retry the event handler’s actions.

**Success**

An event handler exits with a status of **Success** only when it does one of the following:

- Reaches an event action of the type **Finish**.
- Completes all event actions successfully without reaching an event action of type **Fail**.

**Failure**

An event handler exits with a status of **Failure** when it does any of the following:

- Reaches an event action of the type **Fail**.
- Attempts an illegal operation.
- Experiences an unexpected failure of an event action or something launched by an event action.

**When Failures Occur**

When a failure occurs:

- The **Event Handler Status** form’s **Result** field displays an appropriate error message, which includes an error message passed from a failing stored procedure in a message type parameter (@Infobar), where applicable.
- The current event action state’s **Times Failed** counter is incremented. This value displays on the **Actions** tab of the **Event Handler Status** form.
- The event handler state’s **Last Activity Date** is set to the date and time of the failure. This value displays on the **Handlers** tab of the **Event Handler Status** form.
- Any event variables are maintained at their current values for inspection on the **Variables** tab of the **Event Handler Status** form.

**Ignoring Failures**

By default, if any synchronous event handler reports a failure, the system skips any remaining event handlers for that event and exits with a status of **Failure**.

You can override this default behavior by using the **Ignore Failure** option for the event handler (on the **Event Handlers** form). If you select this option, the event handler is always treated as successful unless it deliberately attempts an illegal operation. (The **Status** field on the **Event Handler Status** form, however, still shows a status of **Failure**, even though the **Any Handlers Failed** check box remains cleared.)
The following table shows what conditions can result in failures for various action types and whether the failure can be ignored.

<table>
<thead>
<tr>
<th>Action type for the current event action</th>
<th>Condition</th>
<th>Ignore Failure option selected?</th>
<th>Result (Event Status)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set Values</td>
<td>Illegal operation: Any attempt to set a non-output event parameter</td>
<td>N/A</td>
<td>Failure</td>
</tr>
<tr>
<td>Some</td>
<td>Illegal operation: Any attempt to use an action type that is not supported on the current tier</td>
<td>N/A</td>
<td>Failure</td>
</tr>
<tr>
<td>Any</td>
<td>Illegal operation: Syntax error in parameter</td>
<td>N/A</td>
<td>Failure</td>
</tr>
<tr>
<td>Generate Event</td>
<td>Illegal operation: Any attempt to generate a framework event</td>
<td>N/A</td>
<td>Failure</td>
</tr>
<tr>
<td>Any</td>
<td>Unexpected error</td>
<td>No</td>
<td>Failure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Yes</td>
<td>Success</td>
</tr>
</tbody>
</table>

A **Fail** action whose condition evaluates to **True** on an event handler for which the **Ignore Failure** option is selected is equivalent to a **Finish** action.

An asynchronous event handler's failure does not affect any remaining event handlers for that event. However, setting the **Ignore Failure** option in this case avoids affecting the **AnyHandlersFailed()** attribute of that event, unless the failure was caused by deliberately attempting an illegal operation, as shown in the table above.

**Retrying Event Handlers**

In the Event Status and Event Handler Status forms, you can retry a failed event handler, if it is retryable. Whether a handler can be retried is determined by the cause of the failure and whether the event handler is marked transactional, as shown here:

<table>
<thead>
<tr>
<th>Cause of Failure</th>
<th>Event transactional</th>
<th>Handler transactional</th>
<th>Retryable</th>
<th>Retry point</th>
<th>Event Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Illegal operation</td>
<td>N/A</td>
<td>N/A</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Any legal operation</td>
<td>Yes</td>
<td>N/A</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Any legal operation</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>InitialAction</td>
<td>Re-initialized</td>
</tr>
<tr>
<td>Unconditional Fail Event Action</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>InitialAction</td>
<td>Re-initialized</td>
</tr>
<tr>
<td>Conditional Fail Event Action</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>CurrentAction</td>
<td>Maintained</td>
</tr>
<tr>
<td>Unexpected Error</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>CurrentAction</td>
<td>Maintained</td>
</tr>
</tbody>
</table>
When retrying a single handler, whether the handler runs synchronously or asynchronously depends on the parameter passed to the API, not the handler’s definition. When retrying an event, the handlers are run as shown in this table:

<table>
<thead>
<tr>
<th></th>
<th>Failed Synch Handler</th>
<th>Failed Asynch Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restarted synchronously</td>
<td>Synchronous</td>
<td>Asynchronous</td>
</tr>
<tr>
<td>Restarted asynchronously</td>
<td>Asynchronous</td>
<td>Asynchronous</td>
</tr>
</tbody>
</table>

An event can have a synchronous handler that is not retryable, but it will still retry any asynchronous handlers that can be retried. In this case, any synchronous handlers that would have come after the failed handler will not be run.

The Event or EventHandler Result must be cleared when retrying the handler. However, the previous handler result is saved in a handler variable called PreviousResult, and the event results are saved in an event parameter also called PreviousResult.
Transactions

By definition, a *transactional* event is an event or set of event handlers in which all must complete successfully before any data is committed in the system. If any handler fails, then the system rolls back (reverts) all data to its initial state.

If the transactional event contains any adjourning event actions, the system modifies this behavior somewhat: Any actions taken up to the first adjournment, between each resumption of an adjournment and the beginning of the next adjournment, and after the last resumption, are each committed separately.

The system treats any asynchronous event handlers as if they belong to an asynchronously generated non-transactional event. In other words, because they are outside the synchronous flow, these event handlers are not treated as part of the transaction.

For the specifications of synchronously and asynchronously generated events, see the table for *Summary of Synchronous Functionality* on page 127.

Transactions with Synchronous Events

An event generated synchronously without the **Transactional** check box selected (on the **Event Handlers** form) either:

- Runs in the current transaction state of the firing thread (if no lower-level transactioning is specified).
- Handles transactions at the event handler or event action level, as explained in the following points:
  - In the database tier, this occurs naturally with no special syntax.
  
  When `FireEventSp` is called within a SQL transaction, any changes to the non-event system portions of the database (using **Audit**, **Call Database Method**, and **Run Background Task** event actions) become a part of that transaction, which can be rolled back after `FireEventSp` returns.

  When `FireEventSp` is called outside a SQL transaction, each event action that might change non-event system portions of the database is wrapped in its own transaction, which is rolled back in case of failure, except when the containing event handler is marked **Transactional**. Changes to event system state data are performed on a separate connection, so they survive any rollback of the encompassing or wrapping transactions.
In the middle tier, any execution methods for database-related event actions must enlist in an existing transaction or create a new one.

To include the entire event in a single transaction, call the FireApplicationEvent method from a hand-coded IDO method that is marked **Transactional** in the IDO metadata. In this case, the execution methods for the database-related event action enlist in the transaction created by the IDO Runtime for that transactional IDO method.

When the FireApplicationEvent method is called from a hand-coded IDO method that is not marked **Transactional** and whose caller is not in a transaction state, each event action execution method creates a new transaction, which is rolled back in case of failure, except when the containing event handler is marked **Transactional**. Changes to event system state data are performed in a separate connection (except in **Suspend-validating Mode**), so they survive any rollback of the encompassing or wrapping transactions.

The client is not permitted to control transactions that span form event handler calls. Client-tier event-firing requests are passed directly on to the MGCore.Events.FireEvent IDO method.

A failure of a synchronous event returns an error condition to the firing thread. For framework (**Core**) events, this causes the entire current transaction to be rolled back. For application events, whatever generated the event (stored procedure/trigger, .NET method, or VB.NET script) is responsible for trapping the error condition, accomplishing a rollback of the current transaction, and throwing execution to an appropriate recovery point in or out of that code.

### Event Handlers Marked Transactional

An event handler marked **Transactional** is wrapped in:

- A new SQL transaction when executed in the database tier.
- A new transaction when executed from the middle tier, if no encompassing transaction is active at the trigger point of the outermost enclosing event.

This event handler cannot contain any **adjourning** event action types (for example, **Prompt**, **Wait**, or **Sleep**), but it can be marked **Asynchronous** itself. If this event handler ends in a failure, the wrapping transaction is rolled back (assuming it was created).

### Event Handler Not Marked Transactional

An event generated asynchronously without the **Transactional** setting does not run in a transaction state. For event handlers that are not marked **Transactional**, for a synchronous event generated from a middle-tier non-transactional custom IDO method, are handled as follows:

- An event can be generated asynchronously with a **Transactional** setting that indicates that a new transaction is begun and the entire database-related effects of its synchronous event handlers are committed if they all succeed or rolled back if any fail.
- Any asynchronous event handlers are treated as if they belong to an asynchronously generated non-transactional event.
Rolling Back Transactions

The following conditions cause the system to set the **Transactional** setting on a new event state row:

- FireEventSp or FireApplicationEvent() detects an enclosing transaction.
- FireEventSp, FireApplicationEvent(), PostEventSp, or PostEvent() receives a **Transactional** setting.

This event state setting signals the system administrator that when this event state is finished and not rolled back, everything happened that was expected to happen.

In a synchronous multi-event situation, to encode the proper information in the **Rolled Back** setting, an enclosing transaction that contains one or more FireEventSp or FireApplicationEvent() calls must do the following:

1. Once, before calling FireEventSp or FireApplicationEvent(), call EventBeginTransactionSp or EventBeginTransaction().
   This uses a separate transaction to clear a storage area to record the row pointers of the event state rows to be created and returns a transaction identifier to be passed to FireEventSp or FireApplicationEvent().

2. Pass that transaction identifier to FireEventSp or FireApplicationEvent().
   This, in a separate transaction, records the row pointer of the new event state row.

3. Upon rolling back or committing the transaction (it does not matter whether before or after), call EventEndTransactionSp or EventEndTransaction() and pass the transaction identifier and whether a commit or rollback will be/was performed.
   This uses a separate transaction to set the **Rolled Back** setting on the stored event state rows (if rolling back), and clears the storage area.

If this is neglected, the system administrator will not be able to determine why database actions that appear to have finished successfully, according to the status forms, are not reflected in the database.

In asynchronous situations, and when a failure is detected by the event system, the **Rolled Back** setting for the failing transactional event is set by the event system.
The Framework Event Service

The Framework Event Service is an independent process that can run on any utility server. Each instance of the Framework Event Service monitors the event queue for any number of configurations defined locally.

For load-balancing purposes, this service safely acquires work from the database queue of each configuration’s application, so that multiple event services can run on one or multiple servers. The event service monitors the event queue for:

- New events that have been queued by means of asynchronous generation methods or suspending framework events, as found in the Queued Event table.
- New event handlers that have been queued by means of the firing of asynchronous event handlers, as found in the Queued Event Handler table.

Setting Up the Framework Event Service

To set up the Framework Event Service to monitor specified configurations, you must specify each configuration individually on the Event Service tab of the Service Configuration Manager utility.

NOTE: If you do not do this setup, any event handler you create that requires the event service will not work until you do.

For more information, including the procedure, see the online Help for the Service Configuration Manager utility.

If you want to use the IDO Runtime Development Server (IDORuntimeHost.exe) in your development work, you must also temporarily remove the dependency that the Event Service has on the IDO Runtime service. For more information, including the procedure to do this, see the online Help for WinStudio developers: In SyteLine, from the Help menu, select Customizing Forms; then, when the Help window opens, select the topic “Running IDO Runtime Development Server”.

Processing Order in the Framework Event Service

The Framework Event Service processes any queued events in “first in, first out” (FIFO) order.

Because the event service could receive a request to run something while it is executing a prior request, all requests are queued for execution in the order requested. If the new request happens to be the only one in the queue and the event service is not busy, the request is executed at the next polling.

When the event queue is empty, and immediately after processing each queued event or event handler, the event service checks the Event Handler State and Event Trigger tables for any items whose Retest At time setting has arrived or passed.

The event service then checks the Event Handler State table for any items for which the timeout has expired, that is, the Times Out At time setting has arrived or passed. The event service processes these in order (that is, oldest first), using the Retest At time or Times Out At time to determine the order.
After this, the event service checks the event queue again for any incoming items.

Administrative Details

To eliminate the possibility of deadlocks when manipulating event data, a second database connection is used for event data modifications, that is, separate from the one used by the event action execution methods. Also, any event data modifications that do not need to be available to other processes—that is, any tier of any interactive user session or other event service instances—are made in memory and written only when required.

To avoid overtaxing the utility server, the Framework Event Service administrator has control over the minimum interval for the following attributes:

- **RetestInterval** parameter on Wait event actions
- **Interval** parameter on Sleep event actions
- **RetestInterval** attributes on event triggers

When any of these attributes is referenced, if the designer-specified value is greater than zero but lower than the minimum interval for the Framework Event Service, the minimum value is used.

The Framework Event Service logs various levels of messages to the Infor framework Log Monitor.
Event and Event Handler Revisions

The system creates a set of event handler revisions the first time:

- The event is generated or any of its handlers executes.
- The event is generated after any of its constituent handlers or their actions has been modified in any way.
- Any of the event’s handlers executes after any of its constituent actions has been modified in any way.

Modifications can include additions, changes, or deletions to the handlers or actions associated with the event or handler.

When the revision is created, the system copies all of the event's handlers and actions to a read-only table. The event or handler then uses the data in this read-only table when executing the handlers and actions, until a new revision is created.

Revisions are used to help ensure that the metadata used by an event and its handlers and actions is not altered while the event is executing. In some cases, it can take the system a period of time to finish executing an event's handlers, either because of the processing time involved, or because the system is waiting for some input or response before it can continue. It is possible that someone could make changes to the event's handlers and actions while the event is processing or waiting, and that these changes could affect the execution of the event that is in progress.

Revisions were developed to address this kind of potential problem. When an event is triggered, the event and its handlers use the revisions that are in effect at the time they start executing until they are finished.

Example:

Suppose you are using an event to send notices to a manager when a customer order is more than $10,000. The manager must approve the order before it can be processed.

During a transfer of responsibilities from one manager to another, a new manager is assigned the responsibility of approving such orders. The system administrator makes the change of notice to the appropriate event action.

However, the original manager still has a few orders pending approval. These orders continue to use the metadata for the revision in effect at the time they were created, awaiting that manager's approval. In the meantime, the new manager receives notices for any subsequent orders, because the first new order generates a new revision using the information for the new manager.
Infor has built in to the system a number of events and handlers that are available for immediate use. In addition, if you have an add-on products distributed by one of Infor’s business partners, they might have also added their own events and handlers that you can use.

You can also design and create your own custom events and handlers to automate tasks for your particular needs. The following major topics provide information about the key elements of the application event system and the procedures for creating and using custom events and handlers.

TIP: If you are creating and using your own custom events and handlers, we recommend that you discard the metadata cache periodically. This should be done after doing development work, before testing, and after synchronizing your metadata on your system. For more information, see Discarding the Metadata Cache on page 58.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Elements of the Application Event System</strong></td>
<td>39</td>
</tr>
<tr>
<td>- About the Access As Identifier</td>
<td>39</td>
</tr>
<tr>
<td>- About Events</td>
<td>39</td>
</tr>
<tr>
<td>- About Event Triggers</td>
<td>41</td>
</tr>
<tr>
<td>- About Event Handlers</td>
<td>43</td>
</tr>
<tr>
<td>- About Event Actions</td>
<td>44</td>
</tr>
<tr>
<td>- About Event Action Parameters</td>
<td>45</td>
</tr>
<tr>
<td>- About Event Action Parameter Forms</td>
<td>48</td>
</tr>
<tr>
<td>- About Event Variables and Initial States</td>
<td>49</td>
</tr>
<tr>
<td>- About Event Global Constants</td>
<td>49</td>
</tr>
<tr>
<td><strong>Application Event System Design Forms</strong></td>
<td>51</td>
</tr>
<tr>
<td><strong>Setting Up Custom Events and Handlers</strong></td>
<td>54</td>
</tr>
<tr>
<td>- Designing a Custom Event</td>
<td>54</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>About Event Handling and Order</td>
<td>55</td>
</tr>
<tr>
<td>Determining Names of IDO Collections and Components</td>
<td>57</td>
</tr>
<tr>
<td>Discarding the Metadata Cache</td>
<td>58</td>
</tr>
</tbody>
</table>
Elements of the Application Event System

This section identifies and describes the key elements of the application event system. Where applicable, procedures for creating and using various elements are also provided.

About the Access As Identifier

One key element of the application event system is the Access As identifier. The Access As identifier is used to:

- Indicate who (what organization) created and "owns" an event or related event system object.
- Prevent unauthorized developers from modifying or deleting event system objects that they do not own.

The Access As identifier is also used to indicate ownership and modification rights for certain IDO metadata. It might also be used in the future for other, similar metadata.

Generally, the Access As identifier falls into one of three classifications:

- **Core** – Indicates that the object is one that Infor created and owns. These event system objects are used for the framework forms and operations.
- **Any other name** – Indicates that the object was created by and belongs to an application created either by Infor or by one of Infor’s business partners or other authorized vendors.
- **Blank** – Indicates that the object was created by and belongs to the end customer.

Within WinStudio, several forms include an Access As field. On the Access As form, this field indicates the "current Access As" value. This is the value assigned to any new event system objects you might create. On all other forms, this field indicates who has ownership of the pertinent object metadata, in other words, who created and owns it.

You can only modify or delete metadata for event system objects that have the same value as on the Access As form, in other words, event system objects that your organization has created and owns.

**NOTE:** If you are an Infor business partner or authorized vendor developer, and you need to change the Access As identifier, contact your Infor Professional Services Organization (PSO) representative for help.

With a few exceptions (noted where applicable), you can attach your own event triggers and handlers to event system objects owned by other organizations (that is, with a different Access As identifier than yours), but you cannot directly modify or delete those event system objects.

About Events

An event is defined as a uniquely named situation that can be triggered by:

- An action performed by somebody working in the system.
- A particular condition that occurs while the system is running.
- A certain value that is exceeded in a database record.
- Another event’s handler.
Other, similar occurrences.

Event Types

Events can be one of three general types:

- **Core, or framework, events** – These are events that Infor has defined and built in to the system. They are tagged with an Access As identifier of Core.
  
  These events generally fall into one of two categories:
  
  - IDO (business process-related) events that are generated when certain IDOs are invoked. These IDOs include IdoOnItemInsert, IdoOnLoadCollection, IdoOnInvoke, and others. You can identify these events easily by their names, which begin with the letters Ido.
  
  - Session events that are generated when certain session activities take place. These include SessionOnLogin, SessionOnLogout, and SessionOnVarChanged.
  
  These events are always synchronous (see Synchronicity on page 22) and transactional (see Transactions on page 31). Some can be optionally suspended to await user responses (see Suspension on page 24).

- **Application-specific events** – These are events that typically have been created by Infor, its business partners, and authorized vendors. They are tagged with an Access As identifier that indicates what application or development organization they belong to.

- **Customer-defined events** – These are events that a developer in an end-customer organization has created. They are tagged with a blank Access As identifier, which indicates that they were created by and belong to the customer.

For more information about Access As identifiers, see About the Access As Identifier on page 39.

Defining Events

Events can be defined (named) on the following forms:

- **Events**
- **Event Triggers**
- **Event Handlers**

To define an event, enter the name of the event in the Event Name field of one of these forms.

**NOTE:** If you do not name the event on the Events form, it will still be available to the drop-down lists on other forms. Events named on those forms, however, do not display on the Events form. So, if you want the event to display on the Events form, you should name the event on that form.

When an event is defined, or named, it is really just that: a name. Until you define a way for it to be triggered, the event remains just a name.

For more information about any of these forms, see the online Help for that form.
Modifying Events

Once an event has been created and saved, the only thing you can modify is the event's description. The event name and other attributes are locked.

NOTE: You can modify an event's description only if the Access As field has the value of the current Access As value, as displayed on the Access As form.

To modify an event's description:
1. Open the Events form and select the event you want to modify.
2. In the Description field, modify the description text as desired.
3. Save.

Deleting Events

If you are certain you no longer need an event and you want to delete it, you can.

You can delete an event only if the Access As field has the value of the current Access As value, as displayed on the Access As form.

To delete an event:
1. Open the Events form and select the event you want to delete.
2. From the Actions menu, select Delete.
3. Save.

About Event Triggers

An event trigger is defined as a condition that causes a particular event to be generated, more-or-less independent of anything that might be happening in the user interface. The event trigger carries a set of event trigger parameters for use when the event is triggered.

An event trigger can be set to generate the event only once, or it can be set to retest for its condition after waiting a certain amount of time since either of the following situations was true:

- The trigger last successfully generated the event.
- The trigger last tested unsuccessfully for its condition.

In both cases, the event trigger’s designer can set the interval to wait, both for the successful firing of the trigger and for the unsuccessful test for the trigger conditions (using separate settings). Testing and retesting is accomplished by means of polling; this is not a true interruptive trigger.

An event trigger carries with it the user name and configuration in effect at the time it was defined. This data is passed on to the event state when the trigger generates the event.

Defining Event Triggers

Event triggers are defined using the Event Triggers form. Use this form to determine the condition that will generate the event, set the parameters to be passed to the event when it is generated, and enter retest intervals.
To create an event trigger:

1. Open the **Event Triggers** form.
2. Press F3.
3. From the **Actions** menu, select **New**.
4. From the **Event Name** drop-down list, select the event for which you want to define a trigger.

**NOTE:** You cannot define a trigger for a framework (**Core**) event.

5. On the **Trigger** tab, in the **Condition** field, enter the condition for which the event is to be generated.

   For more information about defining conditions, see Triggers and Conditions below.

6. On the **Parameters** tab, enter the name and values for any event parameters for which you need to pass values to the event handlers when the event is generated.

7. Set other options on this form as desired.

   For more information on other form options see the online Help for each field.

8. Save the form.

**Triggers and Conditions**

Each event trigger must contain a condition consisting of one of the following:

- A Boolean expression
- Two non-Boolean expressions separated by a comparison operator

**Examples:**

- The following example causes the event to be generated when seven days have elapsed since the current result of the database function `dbo.LastEntryDate()`:
  
  `DATEDIFF(day, DBFUNCTION("LastEntryDate"), CURDATETIME()) > 7`

- The following example causes the event to be generated when the balance on a certain customer's order is greater than $10,000:
  
  `DBFUNCTION("OrderBalance", GC(BigCustNum)) > 10000`

- The following example causes the event to be generated on the first day of each month:
  
  `DATEPART(day, CURDATETIME()) = 1`

**NOTE:** The condition should generally involve either a time operation or a database calculation, or both. This is because time and the database are the only known factors that can undergo change from external stimuli (that is, by the forward movement of time or by the actions of other application users, respectively).

**Retesting Triggers**

When the event trigger's **Retest At Date** setting becomes older than the current system clock time, the event trigger is available to be processed by the event service. This happens when the event service is free from processing any waiting queued events and handlers and any already-waiting triggers that need to be tested or retested.
To process the event trigger, the system parses the condition, evaluates it, and, if the condition evaluates to True, then the event is generated. At that point, the Retest At Date setting is set to the current time plus the amount of time set for the Trigger Reset Interval. If the Trigger Reset Interval is set to 0 (zero), then the Active flag is turned off, which indicates that the trigger will not be retested.

If the condition evaluates to False, then the Retest At Date setting is set to the current time plus the amount of time set for the Condition Retest Interval. If the Condition Retest Interval is set to 0 (zero), then the Active flag is turned off, which indicates that the trigger will not be retested.

About Event Handlers

An event handler defines the work to be performed upon the firing of a particular event. Each event handler is comprised of one or more event actions and, optionally, an initial state.

Each event can have multiple event handlers that execute when the event is generated. In such cases, the handler sequence number and other factors determine the order in which event handlers are actually processed.

For more information about event actions, see About Event Actions on page 44.

For more information about initial states, see About Event Variables and Initial States on page 49.

Defining Event Handlers

Each event handler is uniquely defined in the system by the combination of an event name and a handler sequence number. Both of these are set on the Event Handlers form. Event handlers also must have one or more associated event actions. For more information, see About Event Actions on page 44.

To create an event handler

1. Open the Event Handlers form.
2. Press F3.
3. From the Actions menu, select New.
4. In the Event Name field, do one of the following:
   - Select the event for which you want to create a handler.
     This sets up the event handler for an already-defined event.
   - Enter the name for an event which has not been previously defined.
     This effectively defines a new event as well.
5. (Optional) To control the order in which the event handler executes (especially with respect to existing event handlers), use the Keep With and Chronology fields, or the Event Handler Sequence form. For more information, see About Event Handling and Order on page 55.
6. Save the new event handler.
7. Use the **Event Actions** button to open the **Event Actions** form and create the actions to be performed when this event handler is executed.

   For more information, including the procedure, see **About Event Actions** on page 44.

8. After closing the **Event Actions** form, set the other options on the **Event Handlers** form as desired.

   An event handler can be restricted to execute only in relation to a specific set of conditions. For example, a particular event handler might be defined to execute only when the associated event is triggered by an action on a particular form or when a particular IDO is involved.

   An event handler can also be set to execute synchronously or asynchronously, or as part of a transactional event or set of event handlers. For more information, see **Synchronicity** on page 22 or **Transactions** on page 31.

   For information about the other options, see the online help for each option.

9. Save the event handler.

### About Event Actions

An **event action** is defined as a unit of work to be performed during the execution of an event handler.

A single event handler can have multiple event actions, but each action is assigned to a single event handler.

Depending on its action type, an event action can do such things as:

- Evaluate and compare expressions, using the results to select which event action of its event handler to perform next.
- Affect the event’s visual state.
- Complete the event handler.
- Set event variables.
- Call methods or Web services.
- Perform other predefined tasks.

### Defining Event Actions

To define new event actions, open the **Event Actions** form using the **Event Actions** button on the **Event Handlers** form. If you open the **Event Actions** form from the Explorer or the **Select Form** dialog box, you can only view and modify existing actions.

To create an event action:

1. Open the **Event Handlers** form.
2. Create a new event handler; or, in the grid view, select the event handler for which you want to create the action.
3. Click the **Event Actions** button.
4. From the **Actions** menu, select **New**.
5. In the **Action Sequence** field, enter a number.

   This number determines the order in which actions for a particular handler are processed.

6. In the **Event Actions** form, **Action Type** field, select the type of action to be performed.

   For a complete list of action types and what they do, see [Event Action Types](#) on page 135.

7. Click **Edit Parameters** and use the associated event actions parameter form to define the parameters for that action.

8. To verify that the syntax is correct, click **Check Syntax**.

   If you have any syntax errors, fix them before proceeding.

9. If the action involves a variable to be used in event messages and you want to restrict the variable's accessibility on the target form, set those restrictions on the **Variable Access** tab.

   For more information about setting variable access, see the online Help for the **Variable Access** tab.

10. Save your changes and close the form.

### About Event Action Parameters

Depending on the action type, you also specify optional parameters for each event action type. You can specify parameters in any order for a particular action. To list multiple parameters, enter them one after another, entering either a space or nothing between them. Event action parameters are defined on the **Parameters** tab of the **Event Actions** form.

**TIP:** You can use the event action parameter forms to define the parameters. When you do, the parameters are returned to the **Event Actions** form properly formatted and free of syntax errors. For more information, see [About Event Action Parameter Forms](#) on page 48.

The syntax for each event action parameter is as follows:

```
FUNCTION(value)
```

**TIP:** Although it is not a requirement that function names be entered using all uppercase letters, we recommend the practice, as it leads to greater ease of recognition and readability.

The value enclosed in parentheses can consist of:

- A constant number.
- A constant string enclosed in quotation marks.
- A constant Boolean value: TRUE or FALSE.
- An event function call.

These can be nested. For more information, see [Nesting Function Calls](#) on page 47.

- An expression consisting of a number of these elements combined using operators.
You can also use the parentheses following the function to wrap expressions that signify operations to be performed on the results of the expression.

For example, the function \texttt{V} takes as a parameter the name of a variable. This function can be placed in the parameters for other functions, as in the following example:

\texttt{METHOD(V(FuncNameVar))}

For more information about event action parameters, including a list of all acceptable parameters, their meanings, and examples, see Event Action Parameters on page 140.

For the complete expression grammar for constructing event action parameters, see Appendix C, Expression Grammar.

Function Types

Functions can be any of three basic types:

- \textit{Parameter} functions — These are functions whose parentheses wrap a "parameter" to the event action. For example, the following are all typical parameter functions:
  - SETVARVALUES
  - METHOD
  - INTERVAL
  - EVENTNAME

  These functions must always appear at the "root" level and can never be nested inside any other type of function.

  These functions are identified in this documentation generically as PARAMS(…).

- \textit{Value} functions — These are functions that call event values such as:
  - SUBSTITUTE
  - DATE
  - ABS
  - CEILING

  These types of functions can never appear at the "root" level but must be nested within another function construct (either a parameter or another function).

  These functions are identified in this documentation generically as FUNCTION(…).

- \textit{Word} functions — These are verbatim words used inside certain event function calls, such as:
  - AS
  - STRING
  - NUMBER
  - DAY
DATE

Note that some of these can also be used as event function calls. These functions always appear within an event function call, however.

These functions are identified in this documentation generically as …WORD…

Nesting Function Calls

When defining event action parameters, keep the following rules (as specified in the previous section) in mind:

- PARAMS(…)-type functions can never be nested.
- FUNCTION(…)-type functions must be nested and can be nested either within PARAMS(…) functions or within other FUNCTION(…) functions.
- …WORD…-type functions must be nested within FUNCTION(…)-type functions.

The following is an example of an event action parameter that uses nesting of functions:

\[ \text{PARAMS}(...\text{FUNCTION1}(...\text{FUNCTION2}(...\text{WORD1}...).\text{WORD2}...)...) \]

Passing Parameters from Actions

Parameter lists to methods, scripts, Web services, and generated events are always enclosed in a PARAMS(…) function and delimited by commas. Each parameter is specified in one of the following ways:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Direction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>V(var)</td>
<td>Input</td>
<td>Pass in the value of the variable var.</td>
</tr>
<tr>
<td>expression</td>
<td>Input</td>
<td>Pass in the value obtained by evaluating the expression.</td>
</tr>
<tr>
<td>RV(var)</td>
<td>Input and output</td>
<td>Pass in the value of the variable var, and place the output value into the same variable.</td>
</tr>
</tbody>
</table>

Setting Variable and Parameter Values

You can set values for event variables and parameters using the following syntaxes:

<table>
<thead>
<tr>
<th>Event action type</th>
<th>Type of storage</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call Database Method</td>
<td>Variable</td>
<td>\text{PARMS}(RV(var))</td>
</tr>
<tr>
<td>Call IDO Method</td>
<td>Parameter</td>
<td>\text{PARMS}(RE(param))</td>
</tr>
<tr>
<td>Generate Event</td>
<td>Variable</td>
<td>\text{SET}(RV(var)=name)</td>
</tr>
<tr>
<td>Load IDO Row</td>
<td>Parameter</td>
<td>\text{SET}(RE(param)=name)</td>
</tr>
<tr>
<td>Set Values</td>
<td>Variable</td>
<td>\text{SETVARIABLES}(var=expr)</td>
</tr>
<tr>
<td></td>
<td>Parameter</td>
<td>\text{SETPARMVALUES}(param =expr)</td>
</tr>
</tbody>
</table>
About Event Action Parameter Forms

To make it easier to create event action parameters that are properly defined and formatted, each event action type has a corresponding event action parameter form. These forms allow you to build appropriate sets of parameters for each action type by providing options that pertain to that action type.

For example, if you are creating an event action of the type Notify, when you open the associated event action parameter form, you will see something like the following:

![EVENT ACTION NOTIFY (MODAL)](image)

Notice that each editable text field is labeled with a button on the left. This is a fairly typical pattern for event action parameter forms. With each button/field pair, you have the option to either type the value for that parameter in the field or click the button. Clicking the button opens another form that pertains to the parameter associated with that button and that allows you to build just that parameter.

Some options are presented using combo boxes that also have drop-down lists from which you can select appropriate values for that field. Some options are not accompanied by buttons at all. Other options are selected (or not) using check boxes, as with the **Save in Sent Items** in the foregoing example.

Very few options on any given form are considered required. As a rule, only those fields that you populate return parameter values to the Event Actions form. Where strings require quotation marks, the system inserts them automatically. Where expressions are nested inside other expressions or parameter statements, the system automatically places parentheses where they are required.

The net result is that, by using the event action parameter forms, you can eliminate most, or more likely all, syntax errors in your parameter statements.
About Event Variables and Initial States

If an event handler has variables associated with it, those variables can be assigned an initial state, that is, a set of values they are assigned when the event handler starts to execute. Each initial state consists of:

- A name that identifies it in the system
- Any number of event variables with the initial values they are to have when the event handler starts to execute

For example, you might want to use with your event handlers variables with the following initial values:

<table>
<thead>
<tr>
<th>Name of variable</th>
<th>Initial value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment</td>
<td>250</td>
<td>—</td>
</tr>
<tr>
<td>ItemTypes</td>
<td>AB</td>
<td>—</td>
</tr>
<tr>
<td>TimesRemaining</td>
<td>GC(MaxTimes)</td>
<td>In this case, the value is determined by the value of the MaxTimes global constant.</td>
</tr>
</tbody>
</table>

Initial states are defined using the Event Variable Groups form. Once created, you can use a defined initial state with any other event handler by selecting it in the Initial State field on the Event Handlers form.

About Event Global Constants

An event global constant is defined as a named static value that event expressions can reference during processing of the associated event handlers.

An Example

For example, you could use a global constant to create a list of managers who are authorized to control customer credit limits. You could then reference this global constant whenever you create an event handler that sends a notification (Notify event action) regarding a customer’s credit limit. This practice allows you to use the list for multiple event handlers without having to hard-code the list of names in each event handler’s actions. It also allows you to change the list in one place and have that one change affect every event handler that uses it.

For more examples of how global constants can be used, see Appendix A, Sample Scenarios.

Defining and Using Event Global Constants

Event global constants are defined using the Event Global Constants form.

The system references a global constant using a function mechanism that allows dynamic evaluation at each reference.

Event global constants can be especially useful when defining a set of choices to offer the recipients of a prompt message. For example, a global constant named
PromptChoicesYesNo can be defined with the value 1,sYes,0,sNo. You can then reference this constant for any prompt event action, using the expression:

CHOICES(GC(PromptChoicesYesNo))
Application Event System Design Forms

The system includes a set of specialized forms created to enable you to create and use your own custom application events. With the exception of the Access As form and the System Configuration Parameters form in the following table, these forms are located in the Explorer under Master Explorer > System > Event System. The Access As and System Configuration Parameters forms are located one level up, at Master Explorer > System.

Access to these forms is controlled by the System Administration authorization group.

The following table lists and briefly describes the use of the event system design forms.

<table>
<thead>
<tr>
<th>Form name</th>
<th>Description/Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access As</td>
<td>Although not directly used in the creation and customization of application events and handlers, this form displays the current Access As setting. This, in turn, is an indicator of which event system elements you are authorized to modify and/or delete. For more information about the Access As setting, see About the Access As Identifier on page 39.</td>
</tr>
<tr>
<td>Events</td>
<td>Used to name events. Once named here, events are available on other forms as well, particularly the Event Triggers and Event Handlers forms. For more information, see About Events on page 39.</td>
</tr>
<tr>
<td>Event Triggers</td>
<td>Used to define event triggers, which set conditions that cause a named event to be generated. For more information, see About Event Triggers on page 41.</td>
</tr>
<tr>
<td>Event Handlers</td>
<td>Used to display and define event handlers, which determine the work to be done when an event is generated. For more information, see About Event Handlers on page 43.</td>
</tr>
<tr>
<td>Event Handler Diagram</td>
<td>Used to present a graphical representation of an event handler flow. You can also use this form to access the Event Actions form for selected event actions, to view or modify them. Finally, you can also add event actions to an event handler flow using this form. For more information, see the online help topic &quot;Using the Event Handler Diagram Form.&quot;</td>
</tr>
<tr>
<td>Event Handler Sequence</td>
<td>Used to change the order of any handlers that have the current Access As identifier (as indicated on the Access As form). For more information, see the online help for this form.</td>
</tr>
<tr>
<td>Event Actions</td>
<td>Used to display and define the actions to be performed by a particular event handler during its execution. These are the individual tasks accomplished by the event handlers. For more information, see About Event Actions on page 44.</td>
</tr>
<tr>
<td>Event action parameter forms</td>
<td>Used to define the event action parameters for each action type. For more information, see About Event Action Parameter Forms on page 48.</td>
</tr>
<tr>
<td>Event Variable Groups</td>
<td>Used to display and define initial states, which are sets of event variables and the initial values they are to pass to the event handler when it starts to run. For more information, see About Event Variables and Initial States on page 49.</td>
</tr>
<tr>
<td>Form name</td>
<td>Description/Use</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Event Global Constants</td>
<td>Used to display and define event global constants, which are static values that can be accessed and used by expressions during the running of an event handler. For more information, see About Event Global Constants on page 49.</td>
</tr>
<tr>
<td>System Configuration Parameters</td>
<td>Although not directly used in the creation and customization of application events and handlers, you can use this form to control some aspects of how events and handlers behave on the system. These aspects are concerned mostly with retest and reset intervals that globally govern when and how often various conditions can be retested or events can retry. For more information, see the online help for this form.</td>
</tr>
<tr>
<td>Workflow Event Handler Activation</td>
<td>Used to activate predefined event handlers that represent common workflows. You must specify some information, such as users or e-mail addresses that will be notified when an event occurs. You can also copy and modify these event handlers. For more information, see the online help for this form.</td>
</tr>
</tbody>
</table>
The following diagram shows the functional relationship between the design forms and elements for the application event system.
Setting Up Custom Events and Handlers

To create and use your own custom application events, there are several important steps and considerations to keep in mind. You must:

- Design and define your custom event (see Designing a Custom Event).
- Carefully plan and set up the order in which event handlers and actions execute (see About Event Handling and Order on page 55).
- Periodically discard the metadata cache, to ensure that you are working with the most current version of the event metadata (see Discarding the Metadata Cache on page 58).

Designing a Custom Event

When you create a custom event, you must also define what will generate the event. In the current WinStudio system, there are four ways to generate a custom event:

- Create an event trigger using the Event Trigger form. This is probably the easiest and most common way to generate a custom event.
- Use an action of type Generate Event in another event handler.
- Use a form (WinStudio) event handler with a Response Type of Generate Application Event. (Do not confuse the form, or WinStudio event handlers with the application event system handlers described in this guide.)
- Use the WinStudio API to write a custom script that will generate an event.

The following steps represent a typical process for creating custom application events:

1. (Optional) If you want to name the event before defining how it should be triggered and/or handled, use the Events form.
   If you do not name the event on the Events form, it will still be available to the drop-down lists on other forms (specifically, the Event Triggers and Event Handlers forms). Events named on those forms, however, do not display on the Events form. So, if you want the event to display on the Events form, you should name the event on that form.
   For the procedure to create an event, see Defining Events on page 40.

2. (Required only if you want to generate the event using an event trigger) To define one or more triggers that will generate the event, use the Event Triggers form.
   For more information about event triggers, including the procedure to create them, see About Event Triggers on page 41.

3. To define one or more event handlers that will execute when the event is generated, use the Event Handlers form.
   For more information about event handlers, including the procedure to create them, see About Event Handlers on page 43.

   Each event can have multiple handlers that execute when the event is generated. The order in which multiple handlers execute is controlled by a number of factors. For more information and examples, see About Event Handling and Order on page 55.
4. To define one or more event actions for each event handler, use the Event Actions form.
   For more information about event actions, including the procedure to create them, see About Event Actions on page 44.

5. If required, to name and define an initial state for the event handler to use, use the Event Variable Groups form.
   For more information about event variables and initial states, see About Event Variables and Initial States on page 49.

6. If required, to name and define any global constants for the event handler to use, use the Event Global Constants form.
   For more information about event global constants, see About Event Global Constants on page 49.

7. Test the event and its triggers and handlers on a test system before implementing them on your live system.
   For specific examples of how to create and use custom events, see Appendix A, Sample Scenarios.

About Event Handling and Order

There are two basic ways to arrange the order in which events that have multiple handlers and actions execute those handlers and actions.

Ordering Event Handlers

When an event handler is first defined and saved, the system automatically assigns it a handler sequence number. When the event handler is first saved, the system checks to see if there are already any other handlers associated with the named event. Depending on the results of that check, the system then assigns 1 to the handler (if there are no other handlers associated with the event) or the next available integer (if there are other handlers associated with the event).

In general, then, if an event uses multiple event handlers, by default, the system uses the handler sequence numbers to determine the order in which the handlers execute.

It is possible, however, to indirectly alter this default order. This is done either by:

- Using the Keep With and Chronology fields on the Event Handlers form.
- Moving your own adjacent event handlers up or down in the sequence, using the Event Handler Sequence form.

To alter the default order in which event handlers execute using the Keep With and Chronology fields on the Event Handlers form:

1. (Optional) From the Keep With drop-down list, select the event handler you want to use as a reference point and anchor for the current handler.

   This step is not required if you are using the First or Last option in the Chronology field.
2. From the Chronology drop-down list, select the option you want the current handler to use with respect to the handler you selected in Step 1. Options include:
   - **First** – Executes the current handler before any other handlers.
   - **Before** – Executes the current handler just before the referenced handler.
   - **Instead** – Executes the current handler in place of the referenced handler. In this case, the referenced (original) handler does not execute at all.
     For exceptions to this rule, see Overriding Others’ Handlers on page 221.
   - **Exclusively Instead** – Executes the current handler instead of the referenced handler and any other handler that may be referenced to execute instead of (Instead) that same handler.
   - **After** – Executes the current handler just after the referenced handler.
   - **Last** – Executes the current handler after all other handlers.

For more information about reordering event handlers, including some very detailed examples, see Appendix E, Synchronization of Metadata.

### Resequencing Event Handlers

Use the **Event Handler Sequence** form to change the sequence in which your adjacent event handlers execute for a specified event.

**NOTES:**

- This form is intended to be accessed as a linked form, from the **Event Handlers** form only, using the **Resequence** button. If you open it in standalone mode, the results can be unpredictable.
- You can change the order only of event handlers that have the same Access As value as displayed on the **Access As** form, and then only if they are grouped together (that is, adjacent to one another in the sequence), and within the group. You cannot use this form to change the sequence of event handlers with other Access As values.
- To change the order of your event handlers with respect to those of others (that is, with different Access As values), use the **Keep With** and **Chronology** fields on the Event Handlers form.

To change the sequence of an event handler:

1. In the grid view, select an event handler that has your Access As value.
2. Click the **Up** or **Down** button to move the selected handler up or down in the sequence, keeping in mind the restrictions mentioned above.
   - If you try to violate those restrictions, the system generates an error and you cannot complete the move.
3. Save.

### Ordering Event Actions

A single event handler can contain multiple event actions. When you define an event action, you must assign an action sequence number to it (in the **Action Sequence** field of the **Event Actions** form). You can assign any number you want in that field, and the
system automatically sorts the actions in the correct sequence on the Event Handlers form. When the event handler executes, the system uses this Action Sequence number to determine the order in which the actions execute.

The only exception to this rule is that, if you select a particular action (other than 1) in the Initial Action field of the Event Handlers form, then processing of the event actions begins with the designated action and proceeds from there. So, for example, if you had four actions associated with a handler, and you later decided you wanted action number 3 to be the starting point, you would select 3 in the Initial Action field. In this case, only actions 3 and 4 would execute. The system would skip over actions 1 and 2. (You could later execute these actions by using an action type of Branch or Goto, with one of these actions as the destination.)

Determining Names of IDO Collections and Components

To create custom event handlers and actions, you are often required to know the internal names of the collections (known internally as "IDO collections") and the components you want to refer to. For example, to set up a handler, you often need the name of the IDO collection associated with a particular form. To include dynamic content in the subject or body of a message, you often must know the internal name of a column or component within that IDO table.

But what if you do not know those names? How can you find what you need?

Determining the Name of an IDO Object

To determine the name of an IDO object for an event handler definition:

1. Open the form that uses the IDO collection you require.
   
   For example, if you are setting up an event handler to work with customer records, you would open the Customers form.

2. Enter Design mode for that form.

3. Verify that the Form properties sheet is showing.
   
   If it is not, from the View menu, select the Form Properties option.

4. Select the Collections tab.

   The names of all IDO collections associated with that form are displayed in the Collections list at the top of the form. Usually, there is only collection, which makes it easy to figure out. If more than one collection is listed, you might have to take further steps to determine which one is the one you need.

   The internal name of the IDO is that part of the IDO name that displays after the period. For example, the Customers form uses the SL.SLCustomers IDO. The name of the IDO collection as you need it to create an event handler, is SLCustomers.
Determining the Name of a Form Component

To determine the name of a form component:

1. Open the form that has the field or other component that displays the data to which you want to refer.
   
   For example, to create a subject line that includes a customer’s ID number and name, you open the Customers form.

2. Enter Design mode for that form.

3. Verify that the Component properties sheet is showing.
   
   If it is not, from the View menu, select the Component Properties option.

4. On the form, select the component for which you require the name.
   
   For example, to determine the customer ID number, select the drop-down list box to the right of the Customer field label.

5. On the Component properties sheet, locate the Data Source section, Binding field.
   
   The component’s internal name displays in that field, after the period. For example, for the customer ID number, the Binding field displays object.CustNum. Thus, the customer ID field name is CustNum.

Discarding the Metadata Cache

Because certain summary event metadata is cached in memory for faster performance, the IDO metadata cache should be discarded periodically, after changes to event metadata (that is, after making changes to events, handlers, actions, triggers, and global constants). This should be done, at a minimum, after doing development work, before testing, and after synchronizing your metadata on your system.

NOTES:

- If you have multiple utility servers in your system, you must discard cached metadata for each utility server on which metadata might have been cached. The best way to do this is to use the second option described below.

- Any event metadata that is not referenced within two minutes is automatically discarded from the cache. That is why you might notice that things work the way you expect, even without manually discarding the cached metadata. We still recommend that, as a precaution, you manually discard the cached metadata to be sure.

There are four ways you can discard the cached metadata:

- By using the Discard Cache button on the Utilities tab of the Configuration Manager. For more information, see the Configuration Manager online Help.

- By unloading global objects from your system (requires that certain settings be made; see Discarding Cached Metadata by Unloading Global Objects, below).

- Be restarting the IDO Runtime Service on the utility server (see Discarding Cached Metadata by Restarting the IDO Runtime Service on page 59).
In the IDO Runtime Development Server (requires that you be running a copy of this software locally; see Discarding Cached Metadata in the IDO Runtime Development Server on page 59).

Discarding Cached Metadata by Unloading Global Objects

With the correct system settings, you can have the system discard the cached metadata automatically every time you unload all global objects.

To discard cached metadata by unloading global objects:

1. In WinStudio, from the View menu, select Settings.
2. Select the Runtime tab.
3. Select the check box labeled Unload IDO metadata along with global objects.

While working on event metadata, periodically unload global objects, and the cached metadata is automatically discarded with the global objects.

Discarding Cached Metadata by Restarting the IDO Runtime Service

If you do not want to use the first option and you are not using the IDO Runtime Development Server on your local machine, you can discard the cached metadata by manually stopping and restarting the IDO Runtime Service on the utility server. This works because the metadata cache is not saved when the service is stopped.

To discard cached metadata by restarting the IDO Runtime Service:

1. On the utility server, open Control Panel.
2. Select Administrative Tools > Services.
3. From the list double click Infor Framework IDO Runtime Service.
4. In the Infor Framework IDO Runtime Services Properties dialog box, click Stop.
5. When the IDO Runtime Services Properties dialog box indicates that the service has stopped, click Start.

Discarding Cached Metadata in the IDO Runtime Development Server

If you are using a local installation of the IDO Runtime Development Server, you can discard the cached metadata manually.

To manually discard cached metadata in the IDO Runtime Development Server:

1. In the IDO Runtime Development Server, select the configuration for which you want to discard the cached IDO and event metadata.
2. From the Configuration menu, select Discard IDO Metadata Cache.
Tracking Event System Status

Some application events can take a considerable amount of time to process, especially if they involve event messages that require responses from the recipients. The system provides a number of tools and forms that allow you to track the status of application events as they execute and after they have finished executing. Some of these forms also allow you to temporarily adjust the behavior of handler execution.

These forms are located in the Explorer under Master Explorer > System > Event System.

Initially, these forms can be accessed only by members of the System Administration authorization group.

The following forms can all be used to track various aspects of event system status:

<table>
<thead>
<tr>
<th>Form Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Status Form</td>
<td>62</td>
</tr>
<tr>
<td>Event Handler Status Form</td>
<td>62</td>
</tr>
<tr>
<td>Event Queue Form</td>
<td>62</td>
</tr>
<tr>
<td>Event Revisions Form</td>
<td>63</td>
</tr>
<tr>
<td>Event Handler Revisions Form</td>
<td>63</td>
</tr>
<tr>
<td>Suspended Updates Form</td>
<td>63</td>
</tr>
</tbody>
</table>
Event Status Form

Use the **Event Status** form to view:

- The status of events that are currently running.
- The history of events that have finished running.

This form has four tabs:

- **Event** tab – Provides options to filter for events and handlers in various states and make it easier to locate specific ones.
  
  For example, suppose you want to find data about all events that are currently suspended. With Filter-in-Place, you would select the **Suspended** check box and then execute Filter-in-Place.

- **Handlers** tab – Displays data about handlers that are currently processing or have finished processing.

- **Parameters** tab – Displays data about the parameters associated with an event that is currently running. These include input parameters passed into the event, as well as output parameters created by handlers’ actions.

- **Output Parameters** tab – Displays data about the output parameters of an event that has finished running.

For more information about this form and its options, see the online Help.

Event Handler Status Form

Use the **Event Handler Status** form to view:

- The status of event handlers that are currently running.
- The history of event handlers that have finished running.

This form has three tabs:

- **Handler** tab – Displays data about event handler itself.

- **Actions** tab – Displays only data about actions of the handler that have started.

- **Variables** tab – Displays information about variables associated with the event handler.

For more information about this form and its options, see the online Help.

Event Queue Form

The **Event Queue** form displays a list of all asynchronous events and event handlers that the system has queued for processing. All information on this form is display-only. (For more information about synchronous and asynchronous events and handlers, see **Synchronicity** on page 22.)
Events on the queue are processed in FIFO (first in, first out) order. The ID number on this form indicates the order in which events are queued for processing.

This form also displays other information about events and event handlers that have been queued for processing. Each event or event handler that has been queued is displayed as a separate record in the grid view.

For more information about this form and its options, see the online Help.

Event Revisions Form

The Event Revisions form displays information about the event revisions associated with events.

For more information about event revisions and how they work, see Event and Event Handler Revisions on page 36.

Event Handler Revisions Form

The Event Handler Revisions form shows information about event handler revisions associated with event handlers. All information on this form is display-only. The reason for this is that some running or finished handlers are using the metadata in this revision to complete their processing. To change the data for this event handler, you must use the Event Handlers form.

For more information about event handler revisions and how they work, see Event and Event Handler Revisions on page 36.

Suspended Updates Form

The Suspended Updates form displays a list of all the update actions that are currently in a suspended state. This form also allows you to manually take selected updates out of suspension.

For more information about this form, see the online Help for the form.
Event Messages

*Event messages* in the application event system can be generated by:

- The system, as part of an event handler’s actions.
  
  Only Notify and Prompt action types can generate messages.

- Other users on the system, much like e-mail.

Each message is visible only by the recipients and, optionally, the sender of that message.

This section includes information about the following topics:

<table>
<thead>
<tr>
<th>Topics</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Message-Related Forms</td>
<td>66</td>
</tr>
<tr>
<td>Send E-mail to External E-mail Inbox for Prompts</td>
<td>70</td>
</tr>
<tr>
<td>Prompts and Responses</td>
<td>71</td>
</tr>
</tbody>
</table>
Event Message-Related Forms

Event message-related forms are used to view, sort, file, respond to, and send messages generated within the application event system. These message forms reside in the Master Explorer > System > Messages folder.

The application event system uses the following message-related forms:

- **The Inbox Form**
- **The Saved Messages Form**
- **The Send Message Form**

The **Inbox** form can be accessed in a number of different ways:

- From the View menu, by selecting the **Inbox Form** option.
- In the Windows taskbar (notifications area), by double clicking the Infor ERP SyteLine Inbox notification icon ().
- In the same ways that any other forms are accessed.

You can also access a special Web-based version of this form to check messages. For more information, see **The Web-based Inbox** on page 68.

The **Inbox** form displays system messages that can come from two possible sources:

- Application events that employ a Notify or Prompt action type.
- Communications initiated and sent by individual system users to other users on the system.

This form displays only those messages which are still in the recipient's **Inbox** “folder” and not messages that have been moved to other folders. To view those messages, recipients must use the **Saved Messages** form. The **Inbox** form also does not allow recipients to move messages to other "folders." To do that, recipients must use the **Saved Messages** form. For more information, see **The Saved Messages Form** on page 68.

When a message is received, if the recipient has set options to be notified, the system alerts the recipient with the selected notifications. For more information, see "Notifications Settings" in the online Help.

Recipients can mark messages as read and, depending on how and why the message was sent, make other responses.

**Responding to System-Generated Messages**

If the message is the result of a system-generated prompt, each recipient can respond to the prompt, usually by means of a set of voting buttons. For more information about these buttons, see **Prompts and Responses** on page 71.

If the message was also sent to the recipient's external e-mail inbox, and the recipient responded to the e-mail message, the message in the refreshed SyteLine inbox is marked as expired and the buttons are inactive, so the recipient cannot respond twice.
If the message is system-generated and involves variables, for each variable, depending on the Variable Access setting for the event variable (on the Event Actions form) or initial state (on the Event Variable Groups form) or payload status (on the Event Action Notify or Event Action Prompt form), recipients might be able to:

- Provide an optional response.
- Provide a mandatory response.
- Only read the variable value.
- Not see the variable value at all.

Setting Variable Access

The effective visibility and writability of each variable displayed on the Inbox form or Web page is determined by two things:

- What type of action generated the message (Notify or Prompt)
- The (optional) variable access level as specified on the Event Action form for the action itself and/or on the Event Variable Groups form for the event handler’s initial state and/or on the Event Action Notify or Event Action Prompt form for the action with regard to the payload status of the property that corresponds to the variable, when the handler is associated with an IDO event.

For Notify type messages: The default for variable access is Read-Only. You can use the variable access options to override this to Hidden for each variable.

For Prompt type messages: The default for variable access is Writable. You can use the variable access options to override this to Hidden, Read-Only, or Mandatory for each variable.

To change the value of a variable that might appear with a message in the Inbox form, you can do any of the following actions:

- Entering the data using the Variables tab on the Inbox form or Web page as part of a response to a Prompt action.
- Using a Set Values event action with the function syntax SETVARVALUES(PropertyName=name).
- Using any of various event actions that can set a variable on output using the function syntax RV(PropertyName).

Setting Translatable Captions for Variables

In the Inbox form’s Variable tab, the Caption column displays the contents of the notify or prompt message's variable captions in the current user's language. This assumes that 1) the Caption component attributes are set to interpret the bound contents and 2) the caption contains a translatable string name. For payload variables resulting from an IDO event, this string name may come from an IDO property's label string ID. For non-payload variables that are created by an event action, you define this string name through the Event Actions form's Variable Access tab.
The Web-based Inbox

System users can also access their inboxes by means of a Web-based Inbox form. Using the Web-based version, users are not required to log on to the system, but they can perform any tasks in the Web-based version that they can perform in the regular system version.

To access the Web-based Inbox, the following URL must be used:

   http://NameOfUtilityServer/InforInbox/Inbox.aspx

where NameOfUtilityServer is the machine name of the utility (Web) server for the configuration to which the user normally logs on.

The Saved Messages Form

The Saved Messages form displays messages that you have saved or at least have not yet deleted for a selected "folder." Within this folder, you can sort messages by any one of a number of criteria. For more information, see the online Help for this form.

You can also use this form to:
- Create your own "folders."
- Move messages from one folder to another.

For more information, including the procedure to move messages, see the online Help for the Folder Name field on this form.

Folders, in this system, are not represented visually as they typically are in e-mail programs such as Outlook. The only place you can actually view your personal folders is in the Folder Name field on this form. For more information, including the procedure to create your own folders, see the online Help for the Folder Name field.

You can also view messages on this form for information about any responses you might have made to messages generated by the system.

Moving Messages Between Folders

You can use the Saved Messages form to move messages from one folder to another. If the folder does not already exist, you can create the folder at the same time.

To move a message from one folder to another folder

1. Open the Saved Messages form.
2. From the Folder Name drop-down list, select the folder in which the message you want to move is currently placed.
   - The system displays in the grid view all messages in that folder.
3. In the grid view, select the message you want to move.
4. In the Folder Name field, do one of the following actions:
   - For an existing folder, type in the name of the folder or select the folder from the drop-down list.
To create a new folder and move the message to that folder, type in the name of the new folder.

5. Save the message.

The Send Message Form

The Send Message form is used to send system messages, similar to e-mail messages, to other users in the system. You can designate multiple recipients, "carbon-copy" recipients, and instruct the system whether to save a copy in your Sent Items "folder."

For more information, including specific procedures and instructions for using this form, see the online help for this form.
Send E-mail to External E-mail Inbox for Prompts

If a SyteLine user is set up in the Users form to allow the system to "send external prompts," then a prompt action sends the message both to the SyteLine Inbox form and to the recipient's inbox in an external e-mail system such as Microsoft Outlook. The message that is sent to the external e-mail system is an HTML-formatted e-mail that consists of these parts:

- Original subject in Subject line
- Category
- List of internal recipients
- List of internal Cc line
- Original Body
- Original Question
- Choices, as individual hyperlinks to a .NET active server page (ASP) that records the vote
- Payload, which is the contents of the Variables grid from the SyteLine inbox.

Text in the Subject, Category, Body, Question, and Choices, as well as payload captions, can be translated and formatted based on the Default Language specified for the recipient user in the Users form.

When the recipients vote by clicking the link in the e-mail, their Windows-default Web browser opens, or opens a new tab if applicable, and sends the information from the link they clicked to the ASP URL that was included (and hidden) in the email. This URL is built by the system based on the web server list.

The ASP then registers the vote programmatically, as if the recipient had logged into SyteLine, displayed the same message in the SyteLine Inbox, and selected the corresponding Choice button on the Response tab.

The Web page then displays a success or failure message.

The message displays both in e-mail and in the SyteLine Inbox form, but only one response is allowed:

- If users respond from e-mail first, when they display the SyteLine Inbox (properly refreshed), the message is expired and the buttons are inactive.
- If users respond from the SyteLine Inbox first, then clicking a link in the external e-mail brings up the Web page with a message that the message has expired and they already voted for the previously selected choice.
Prompts and Responses

If a message is the result of a Prompt action, the sender of that message can request a specific response from each recipient, usually in the form of a button-vote mechanism. In such cases, the system must be set first to wait for responses, then to know how to handle responses as they are received, and finally, be instructed what to do when responses are not received within a specified timeout period.

Incoming prompts request a response from recipients (using the Question field on the Inbox form or the external e-mail) and display a set of choices. The choices are displayed in the form of voting buttons in the Response tab area of the SyteLine Inbox form, or in the form of links in an external e-mail. For example, a prompt might include buttons or links labeled:

- Approve / Disapprove (default option)
- Yes / No
- OK / Send More Info / Cancel

To customize the choices for a prompt, you must include a Prompt action with a Choices parameter as part of the event action definition. This Choices parameters consists of the CHOICES function followed by a string expression that evaluates to a comma-separated list that contains an even number of elements (value/label pairs). For example, if you want the voting buttons to be labeled Yes and No, with corresponding values returned to the action to be 1 and 0, you could include the following parameter:

```
CHOICES("1,sYes,0,sNo")
```

In this example, the strings "sYes" and "sNo" are WinStudio form strings. These have already been defined for the system as Yes and No, respectively.

If you want your button labels to be localized, for the recipient, you must:

- Use names of existing form strings (as found in the Strings table); or
- Add your own form strings, using the Strings form, and provide the necessary translations. (To open the Strings form, you must be in design mode and, from the Edit menu, select Strings.)

If localization is not an issue, you can also use a literal value that displays on the button verbatim. To enter the string as a literal value here, simply enter it as a list value. If the system does not find the string in the Strings table, the system treats it as a literal value.

For more information about the Choices parameter, see Event Action Parameters on page 140.

Voting Rules

When a prompt is sent to a single recipient, the result of the prompt is the return value from that recipient's choice. However, when a prompt is sent to multiple recipients, you must select a vote-counting method to determine the result of the prompt and include a Voting Rule parameter in your event action definition.
The following table lists and describes the available voting rules.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Majority</td>
<td>A choice must receive more than 50% of the vote to win. As soon as more than 50% of the recipients respond with a particular choice, that choice wins. If you use this voting rule, you should use a Voting Tie parameter (VOTINGTIE) to tell the system how to handle a tied vote. For more information, see Dealing with Indeterminate Voting Results on page 74.</td>
</tr>
<tr>
<td>Plurality</td>
<td>The choice with the highest number of votes wins, even if it does not receive more than 50% of the vote. For example, if three choices are offered, and The first choice receives 24% of the vote; The second choice receives 43% of the vote; and The third choice receives the remaining 33% of the vote; the second choice wins, even though it received less than 50% of the total vote. If you use this voting rule, you should use a Voting Tie parameter (VOTINGTIE) to tell the system how to handle a tied winning vote. For more information, see Dealing with Indeterminate Voting Results on page 74.</td>
</tr>
<tr>
<td>ConditionalPlurality</td>
<td>The choice with the highest number of votes wins, but only if a specified minimum percentage of votes is reached. If you use this rule, you must also include a Minimum Percentage (MINIMUM) parameter. For example, if three choices are offered to 19 recipients, and you specify a minimum of 40% to win, then: In an 8-7-4 split, the choice with 8 votes would win because it meets the minimum percentage. In a 7-6-6 split, there would be no winner, because no choice meets the minimum percentage. In this case, the system must deal with the vote as an indeterminate result. For more information on indeterminate results, see Dealing with Indeterminate Voting Results on page 74. (With a simple Plurality vote, the choice that reaches 7 votes in a 7-6-6 split would win.) If you use this voting rule, you should use a Voting Tie parameter (VOTINGTIE) or Voting Disparity parameter (VOTINGDISPARITY) to tell the system how to handle the vote. For more information, see Dealing with Indeterminate Voting Results on page 74.</td>
</tr>
<tr>
<td>MinimumCount</td>
<td>The first choice to reach a specified minimum number of votes wins. If you use this rule, you must also include a Minimum Count (MINIMUM) parameter. For example, if three choices are offered to 13 recipients, and you specify a minimum of 5 votes to win, the first choice to receive 5 votes automatically wins. Note that, as soon as the minimum count is reached, event execution moves immediately to the next action. In this case, the system expires any responses not yet received, and no further voting can take place.</td>
</tr>
</tbody>
</table>
### MinimumPercentage
- **Description**: The first choice to receive a specified percentage of the vote wins. The percentage is based on the number of recipients of the prompt, not the number of respondents.
- **Parameters**:
  - Minimum Percentage (MINIMUM)
- **Note**: If you use this rule, you must also include a Minimum Percentage (MINIMUM) parameter. Note that, as soon as the minimum percentage is reached for a choice, event execution moves immediately to the next action. In this case, the system expires any responses not yet received, and no further voting can take place.

### EarliestResponse
- **Description**: The first response to the prompt wins, regardless of the choice.
- **Note**: Note that, as soon as the first response is received, event execution moves immediately to the next action. In this case, the system expires any responses not yet received, and no further voting can take place.

### PreferredChoice
- **Description**: If any one respondent votes for the preferred choice, that choice wins. In a case where none of the respondents select the preferred choice, then this rule behaves like the Plurality rule for the remaining choices.
- **Parameters**:
  - Preferred Choice (PREFCHOICE)
- **Note**: If you use this rule, you must also include a Preferred Choice (PREFCHOICE) parameter to specify which choice is the preferred choice. For example, if you have three choices, and you specify the first choice as the preferred choice, then:
  - If anyone votes for the first choice, that choice wins.
  - If the end vote is a 0-6-5 split, the second choice wins.
- **Note**: Note that, as soon as the preferred choice receives a vote, event execution moves immediately to the next action. In this case, the system expires any responses not yet received, and no further voting can take place.

### MinimumCountPreferredChoice
- **Description**: If a specified number of votes for a specified choice is cast, that choice wins. If you use this rule, you must also include a Minimum (MINIMUM) parameter to specify the minimum count, and a Preferred Choice (PREFCHOICE) parameter to specify which choice is the preferred choice. For example, if you set the Minimum to 3 for a Preferred Choice of "Approve," and 3 recipients respond with "Approve," the preferred choice wins. If less than that number of votes are cast for that choice after all recipients have responded, the vote reverts to Plurality. (In that case, the preferred choice may still win.) Note that when you set the Minimum to 1, this rule behaves exactly like Preferred Choice.

### MinimumPercentagePreferredChoice
- **Description**: If a specified percentage of votes for a specified choice is cast, that choice wins. If you use this rule, you must also include a Minimum (MINIMUM) parameter to specify the minimum percentage, and a Preferred Choice (PREFCHOICE) parameter to specify which choice is the preferred choice. For example, if you set the Minimum to 25% for a Preferred Choice of "Approve," and 2 of 8 of recipients respond with "Approve," the preferred choice wins. If less than that percentage of votes are cast for that choice after all recipients have responded, the vote reverts to Plurality. (In that case, the preferred choice may still win.)
Dealing with Indeterminate Voting Results

The following situations create "indeterminate" voting results and set action attributes that are exposed as event functions that can be evaluated by subsequent event actions:

- Any disagreement among multiple recipients, registered as soon as a disagreement is detected. This can include a vote like the example offered in the Plurality description above.
  
  The associated event function is the VOTINGDISPARITY( ) event function, which is a Boolean function that indicates only that there was a disagreement.

- A tie in the case of a Plurality or Majority vote, registered at the point when all responses have been received or when the timeout period has expired.

  The associated event function is the VOTINGTIE( ) event function, which is a Boolean function that indicates only that there was a tie.

You can use the returns from these functions, along with the functions RECIPIENTS( ), RESPONDERS( ), RECIPIENTLIST( ), RESPONDERLIST( ), and NONRESPONDERLIST() to take further actions, such as:

- Reprompting all the recipients and try to get a consensus.
- Reprompting only a select group of the respondents and urge them to adopt a different choice.
- Reprompting only recipients who have not yet responded.
- Take some other predetermined action.

Quorums

On a prompt action, a quorum is automatically calculated based on the number of recipients, the voting rule, and voting parameters such as Minimum. If there is a number of votes by whose tally a voting result can be determined unambiguously, that number is the quorum. Otherwise, the quorum is the number of recipients, that is, everyone has a chance to vote unless a timeout expires. As soon as the quorum is reached, voting is closed, any remaining unvoted messages are expired, and the event continues to the subsequent event action.

However, if you specify a Quorum value, that overrides the automatic calculation. For example, if a message requiring a response is sent to 10 people, but you want a quorum to be reached when only 4 have voted, then specify 4 as the Quorum value.

By default, if Quorum is not specified or is specified with a positive value, Wait for Quorum is true; that is, the event waits until the quorum is reached before it continues with the next event action. If Quorum is specified with a non-positive value, the Wait for Quorum default value is False. If these settings conflict, for example Quorum = 3 and Wait for Quorum is False, the system displays an error message.

If Wait for Quorum is false, the event does not wait for a quorum to be reached. As soon as the messages are sent, execution continues with the next event action. If the system is not waiting for a quorum, the event designer needs to determine when a quorum is reached and what further actions to take. This can be done using VOTINGRESULT(), RESPONDERS(), RECIPIENTS(), etc., in combination with the Wait or Sleep actions.
Sample Scenarios

This appendix presents a number of typical scenarios in which you might want to use the application event system to automate various tasks in response to various situations. In each case, the situation is described and then a proposed solution involving events, handlers, and/or triggers. These solutions are presented in a step-by-step format, as examples that you can learn from and possibly modify for your own use.

NOTES:

- This appendix is still under development, and more scenarios will be provided as they become available. We recommend that you frequently check the Infor global support Web site, for updates to this guide.
- To a certain extent, each scenario builds on the concepts and practices of previous scenarios, so the most effective way to use them is to work through them sequentially. However, each scenario is also more-or-less "self-contained" and can be used independently of the others.

TIP: To see a graphical representation for each flow as you work on it, you can use the Diagram button on the Event Handlers form. This button opens the Event Handler Diagram form, which you can use to view the flow of the event handler as well as access the Event Actions form to edit individual actions. For more information, see the online Help for the Event Handler Diagram form.

The following is a list of the scenarios included in this appendix:

Sending Notifications

- **Scenario 1: Notification of a New Record—Adding a Customer** on page 77 – A simple notification is sent to a credit manager when a new customer is added to the database.
- **Scenario 2: Notification of Changes to an Existing Record—Changing the Credit Limit** on page 82 – The credit manager is notified by e-mail that a customer’s credit limit has been changed and is told what the new credit limit is.
- **Scenario 3: Notification That Includes an "Old" Value** on page 86 – A group of inventory stockers are automatically notified whenever an item’s lot size changes. In the message that is sent, both the previous lot size and the new lot size are included.

Requesting Approvals

- **Scenario 4: Approval for a New Record** on page 93 – A purchasing manager is prompted for approval whenever a new purchase order is requested.
Scenario 5: Requesting Approval by External E-mail for Changes to an Existing Record on page 98 – A credit manager is prompted through an external e-mail for approval of a change to a customer’s credit limit.

Scenario 6: Requesting Multiple and Complex Approvals on page 103 – A purchasing manager is prompted for approval on a purchase order (PO) both of a change in status to Ordered and for the amount of the PO. If the PO is for an amount greater than $100,000, a supervisor is also prompted for approval. If the PO is for an amount greater than $1,000,000, two senior-level executives must also approve it.

Modifying Records

Scenario 7: Adding Information to a Record on page 114 – A credit manager is prompted to provide a credit limit for a new customer, by means of a response to a message.

Voting

Scenario 8: Voting for Various Choices on page 117– Several managers are prompted to approve an engineering change, by means of a response to a message.

Localizing Message Contents

Scenario 9: Translating Captions in a Purchase Request on page 120 – A message containing localizable strings is created.

More Advanced Scenarios

Scenario 10: Opening a Session in a Remote Environment on page 122 – A remote site or SyteLine environment is accessed to retrieve data. The details of and procedure for this scenario are in the Integrating IDOs with External Applications guide.

Scenario 11: Cross-Site Event Firing - Adding a Message to Another Site’s Inbox on page 122 – A message is sent to another SyteLine site’s Inbox form by using a GenericNotify event.
Sending Notifications

One of the simplest uses of the event system is to set up situations in which message notifications are sent out automatically to specified individuals whenever a situation occurs or a condition is met. The scenarios in this section illustrate this kind of situation.

**Scenario 1: Notification of a New Record—Adding a Customer**

Suppose you have a credit manager who wants to be notified whenever a new customer is added to the system, regardless of who adds the customer. Of course, you can simply require each employee who adds customers to the system to manually send a notice whenever a customer is added. But that places an additional burden on the employee and is prone to possible oversight.

A Simple Event and Handler

You can use the application event system to automatically create a notice whenever a new customer is added. In this example, you do not need to create an event, because Infor provides an event named `IdoPostItemInsert` that you can use. All you need to do is create an event handler for that event and assign an action to it that generates and sends the message to the credit manager.

We could use the `IdoOnItemInsert` framework event instead of the `IdoPostItemInsert` event. This will be significant when we get to the section on Refining the Message on page A-3. The advantage of using the `IdoPostItemInsert` event is that, if you allow the Customers form to auto-assign the customer number (instead of entering the customer ID number yourself), the system waits until the ID number has been assigned before filling in the "CustNum" data in the message. If we use the `IdoOnItemInsert` event, the system does not wait, which means that, if you auto-assign the customer ID number, the resulting message has "TBD" in place of the actual customer number.

The following process describes how to set this up:

1. Create the event handler:
   a. Open the **Event Handlers** form.
   b. Press F3.
   c. Press CTRL+N.
d. Create the handler with the following settings:

<table>
<thead>
<tr>
<th>Field or Option</th>
<th>Setting / Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Name</td>
<td>From the drop-down list, select <em>IdoPostItemInsert</em>. <em>NOTE:</em> For details about this and the other framework events included with the system, see Framework Events on page B-4.</td>
</tr>
<tr>
<td>Applies to Initiators</td>
<td>Leave blank.</td>
</tr>
<tr>
<td>Applies to Objects</td>
<td>Enter <em>SLCustomers</em>. To determine what object you need, see the procedure provided in the online Help for this field.</td>
</tr>
<tr>
<td>Keep With</td>
<td>Leave blank.</td>
</tr>
<tr>
<td>Chronology</td>
<td>Leave blank.</td>
</tr>
<tr>
<td>Ignore Failure</td>
<td>Cleared.</td>
</tr>
<tr>
<td>Suspend</td>
<td>Cleared.</td>
</tr>
<tr>
<td>Synchronous</td>
<td>Cleared. Because this notification does not require any response from the credit manager, it can run asynchronously. For more information, see Synchronicity on page 2-4.</td>
</tr>
<tr>
<td>Active</td>
<td>Selected.</td>
</tr>
<tr>
<td>Can Override</td>
<td>Selected.</td>
</tr>
<tr>
<td>Transactional</td>
<td>Cleared.</td>
</tr>
<tr>
<td>Obsolete</td>
<td>Cleared.</td>
</tr>
<tr>
<td>Initial State</td>
<td>Leave blank.</td>
</tr>
<tr>
<td>Initial Action</td>
<td>Leave blank.</td>
</tr>
</tbody>
</table>

For more information about any of these fields and options, see the Help for the Event Handlers form.

e. Save.

2. Define the action for the event handler you just created:
   a. In the Event Handlers form, select the handler you created in Step 1.
   b. Click Event Actions.
   c. In the Event Actions form, in the Action Sequence field, enter 10.

   NOTE: Technically, you can enter any integer you want in this field, and the system treats them sequential order. We recommend using multiples of ten, initially at least, just in case you later need to add more action steps between existing steps, so you do not need to renumber all existing steps.

d. From the Action Type drop-down list, select Notify.
   e. Click Edit Parameters.
   f. In the Event Action Notify form, click the To button.
g. In the **Event Action Parameter Recipients** form, from the list of recipients, select the user ID of the credit manager (or whoever is serving in that role).

   TIP: You can select more than one recipient. Also, to deselect a recipient, click the user ID again.

h. Click **Update** and then **OK**.

i. In the **Subject** field, type **New Customer!**

j. In the **Category** field, type **Change Notification**.

k. In the **Body** field, type **We have a new customer!**

l. Select the **Save in Sent Items** check box.
   This parameter tells the system to save a copy of the notification in the **Sent Items** folder of the person who added the new customer.

m. Click **OK**.
   On the **Event Actions** form, in the editable field on the **Parameters** tab, you should see the following:

   ```
   TO("userID")
   SUBJECT("New Customer!")
   CATEGORY("Change Notification")
   BODY("We have a new customer!")
   SAVEMESSAGE(TRUE)
   ```

   where **userID** is the sign-in user ID for the credit manager.

   TIP: Note where double quotation marks and parentheses are inserted. Because it can be confusing to know where and when to use these punctuation marks, we recommend that you use the event action parameter forms as described in these scenarios. They put the correct punctuation marks in automatically where and when needed and can help you avoid many time-consuming errors in syntax.

n. Save the action and close the **Event Actions** form.

3. (Optional, but recommended) To verify that there are no syntax errors, click **Check Syntax**.

4. Discard the cached metadata.

   For more information, including the procedure, see [Discarding the Metadata Cache](#) on page 58.

Test the event by using the **Customers** form to create a new customer. Then, sign in as the user ID specified in the **TO** parameter, open the **Inbox** form, and verify that the message was received. The message should appear there, along with the properties associated with the new customer, on the **Variables** tab. Note that, because you did not specify any **variable access** rules, all properties (variable values) are display-only.

**Refining the Message**

Suppose you now want to refine the message, to make it even more informative and useful to the recipient. Not only do you want the recipient to get a message, but you want that
message to include the customer number and name for the new customer, so the recipient can look up the customer profile more easily.

To include the customer number and name in the message:

1. In the Event Handlers form, select the handler you just created and then click Event Actions.
2. In the Event Actions form, click Edit Parameters.
3. In the Event Action Notify form, click the Body button.
4. In the Event Action Expression Editor form, from the Select a function drop-down list, select SUBSTITUTE.

The SUBSTITUTE function allows you to specify the basic text of a message, with "replacement markers" embedded in the message. At run time, the system substitutes specified values for these replacement markers. This effectively allows you to create messages with dynamic content.

5. In the Argument 1 field, enter We have added a customer, {0}, customer ID {1}, to our family of customers.

The numbers enclosed in curly braces ( {0} and {1} ) are the replacement markers for which values will be substituted at run time.

TIP: Replacement markers must be enclosed in curly braces { }. They must begin with zero (0) and increment sequentially. If you do not begin with zero or you skip integers, they do not work.

6. Create the expression that will be used to supply the value for replacement marker {0}:
   a. Place the cursor in row 1 of the Arguments grid and then click Build Expression.
   b. In the Event Action Expression Editor form, from the Select a function drop-down list, select PROPERTY.

      The PROPERTY function picks up the value of the CustNum (Customer Number) field.

      TIP: What if you do not know the name of the property for which you want to retrieve a value? How do you find that property name. For an easy method to find the property name, see Determining Names of IDO Collections and Components on page 57.

   c. In the Argument 1 field, type Name and then click OK.
   7. Repeat Step 6 for row 2, using CustNum for the PROPERTY argument (propertyname) in substep d.
   8. In the Event Action Notify form, click OK.
   9. (Optional) On the Event Actions form, click Check Syntax.
   10. Save the action and close the Event Actions form.
   11. Discard the cached metadata.

   For more information, including the procedure, see Discarding the Metadata Cache on page 58.
Test by creating a new customer record and verifying that the intended recipient receives a notification message that includes the correct new customer name and number.

**Refining the Recipient**

When defining the recipients for this message, it can be a good idea to use a global constant value, rather than a hard-coded user ID. This allows you to use the same global constant value in other places in your application. Then, if the name of the credit manager changes, for instance, it is possible to change the recipients by simply changing the global constant value. It also allows you to add multiple recipients, for instance, if you have co-credit managers or you have a trainee you want to also receive the messages.

For more information about global constants, see [About Event Global Constants](#) on page 49.

To redefine the recipients as a global constant:

1. Create the global constant:
   a. Open the **Event Global Constants** form and take it out of filter-in-place mode.
   b. In the **Name** field, enter the name to assign to the constant.
      In this case, you might use **CreditMgr**.
   c. In the **Value** field, enter the logon user ID for the credit manager.
      TIP: To add multiple recipients, enter the user IDs separated by semi-colons only—no spaces.
   d. Save the global constant and close the form.

2. Incorporate the global constant in the event handler action:
   a. In the **Event Handlers** form, select the handler and then click **Event Actions**.
   b. In the **Event Actions** form, click **Edit Parameters**.
   c. In the **Event Action Notify** form, click the **To** button.
   d. In the **Event Action Parameter Recipients** form, click the **Recipients** button.
   e. In the **Event Action Expression Editor** form, from the **Select a function** drop-down list, select **GC**.
      The GC function calls a specified global constant and uses its value at run time. In this case, you want the global constant you created in Step 1.
   f. From the **Argument 1** drop-down list, select the global constant you created in Step 1 (**CreditMgr**) and then click **OK**.
      Notice that the global constant name is not enclosed in double quotation marks. Generally, only literal strings and property names must be enclosed in double quotation marks.
   g. In the **Event Action Parameter Recipients** form, click **OK**.
   h. In the **Event Action Notify** form, click **OK**.
   i. To verify that there are no syntax errors, click **Check Syntax**.
j.  (Optional) On the Event Actions form, click the Substituted Parameters tab and notice that the TO parameter indicates the actual recipient; in other words, the value of the global constant.
k.  Save the action and close the Event Actions form.

Test by creating a new customer record and verifying that all designated recipients receive the notification message. You can also check the Saved Messages form for the user ID from which you were signed in when the message was sent.

Points to Note and Remember
In creating this kind of event handler, keep the following in mind:

- If you do not require a response from the recipient, create the handler as an asynchronous handler, to avoid system slow-downs.
- To be able to use a recipient in other handlers and be able to change that recipient when necessary in only one place, use an event global constant for the recipient.
- To use active data in a message, use the SUBSTITUTE and PROPERTY (or P) function constructs.

Scenario 2: Notification of Changes to an Existing Record—Changing the Credit Limit
This scenario is similar to the first one, except that, instead of notifying the credit manager when a customer is added to the database, we want to create an event and handler that notifies the credit manager whenever a customer’s credit limit is changed. Because the credit manager prefers email and is not always signed in to SyteLine, we also want to send the notification as an e-mail.

NOTE: For this scenario to work properly, you must have SMTP enabled and configured on the Intranets form of the utility server. You must also have your SMTP server set up to relay the e-mails that are sent. For information on how to do this, consult your Windows operating system documentation. Finally, any recipients must also have e-mail addresses saved as part of their user profiles.

As with the first scenario, we can use an existing framework event, IdoOnItemUpdate, and create our own handler for it. And again, because we are simply sending out a notification and the system is not waiting for a response from the credit manager, we can make it an asynchronous event handler.

This event handler requires two actions: one to check whether the Credit Limit field has been changed and one to send the e-mail notification.

To accomplish this scenario:

1. Create the event handler:
   a. Open the Event Handlers form.
   b. Press F3.
   c. Press CTRL+N.
d. Create the handler with the following settings:

<table>
<thead>
<tr>
<th>Field or Option</th>
<th>Setting / Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Name</td>
<td>From the drop-down list, select IdoOnItemUpdate.</td>
</tr>
<tr>
<td></td>
<td>NOTE: For details about this and the other framework events included with the system, seeFramework Events on page B-4.</td>
</tr>
<tr>
<td>Applies to Initiators</td>
<td>Leave blank.</td>
</tr>
<tr>
<td>Applies to Objects</td>
<td>Enter SLCustomers.</td>
</tr>
<tr>
<td></td>
<td>To determine what object you need, see the procedure provided in the online Help for this field.</td>
</tr>
<tr>
<td>Keep With</td>
<td>Leave blank.</td>
</tr>
<tr>
<td>Chronology</td>
<td>Leave blank.</td>
</tr>
<tr>
<td>Ignore Failure</td>
<td>Cleared.</td>
</tr>
<tr>
<td>Suspend</td>
<td>Cleared.</td>
</tr>
<tr>
<td>Synchronous</td>
<td>Cleared.</td>
</tr>
<tr>
<td></td>
<td>Because this notification does not require any response from the credit manager, it can run asynchronously. For more information, see Synchronicity on page 2-4.</td>
</tr>
<tr>
<td>Active</td>
<td>Selected.</td>
</tr>
<tr>
<td>Can Override</td>
<td>Selected.</td>
</tr>
<tr>
<td>Transactional</td>
<td>Cleared.</td>
</tr>
<tr>
<td>Obsolete</td>
<td>Cleared.</td>
</tr>
<tr>
<td>Initial State</td>
<td>Leave blank.</td>
</tr>
<tr>
<td>Initial Action</td>
<td>Leave blank.</td>
</tr>
</tbody>
</table>

For more information about any of these fields and options, see the online help for the Event Handlers form.

e. Save the handler.

2. Create the first action, which checks the condition of the Credit Limit field when the customer record is saved:

a. In the Event Handlers form, select the handler you just created.

b. Click Event Actions.

c. In the Event Actions form, in the Action Sequence field, enter 10.

d. From the Action Type drop-down list, select Finish.

This action type tells the system to finish executing the handler when a particular condition has been met and exit.

e. Click Edit Parameters.

f. In the Event Action Finish form, click the Condition button.

g. In the Event Action Parameter Condition form, click the Expression 1 button.
h. In the Event Action Expression Editor form, from the Select a function drop-down list, select the function PROPERTYMODIFIED.

The PROPERTYMODIFIED function checks to see whether the named property has been modified since the last save. If the property has been modified, the expression returns a value of TRUE.

TIP: What if you do not know the name of the property for which you want to retrieve a value? How do you find that property name. For an easy method to find the property name, see Determining Names of IDO Collections and Components on page 57.

i. In the Argument 1 field, enter CreditLimit, which is the name of the property that we want to check.

j. Click OK.

Notice in the Event Action Parameter Condition form that the expression has been returned and that double quotation marks have been automatically inserted around the name of the property.

Notice also that the Operator and Expression 2 fields have been disabled. This is because the PROPERTYMODIFIED function is a Boolean expression; thus, no comparison is needed to return a Boolean value.

k. Select the NOT check box.

The reason you select this option is because, if you did not, the expression would return a value of TRUE whenever the CreditLimit property has been modified and the handler would finish. But you want the system to continue to the next action when the CreditLimit property has been modified; you want the system to finish at this point only if the CreditLimit property has not been modified.

l. Click OK.

Notice that the system returns the expression to the Event Action Finish form correctly formatted.

m. Click OK.

Notice that the system returns the entire parameter to the Event Actions form with the syntax correctly formatted.

n. To verify that the syntax is correct, click Check Syntax.

o. Save the action.

3. Create the second action, which sends the e-mail notification:

a. To create the action, press CTRL+N.

b. In the Action Sequence field, enter 20.

c. From the Action Type drop-down list, select Send Email and then click Edit Parameters.

d. In the Event Action Send Email form, click the To button.
e. In the **Event Action Parameter Recipients** form, select the user ID for the credit manager.

   If the credit manager has an e-mail address set up as part of the user profile, the e-mail address displays to the right of the user ID. If the credit manager’s user ID does not display an e-mail address, you must add the e-mail address to the credit manager’s user profile on the **Users** form.

   Notice that you could again use a global constant for the credit manager’s e-mail address. However, because we are sending e-mail, we cannot reuse the existing **CreditMgr** global constant, but must create a new global constant for the credit manager’s e-mail address. The reason for not using an event global constant in this case was simply to give you some experience with the **Event Action Parameter Recipients** form’s other capabilities. (In most scenarios, a global constant will be used.)

f. Click **Update**.

   Notice that the system places the e-mail address for the credit manager in the **Recipients** field.

g. Click **OK**.

h. In the **Subject** field, type: **Credit limit change**

i. In the Category field, type: **Financial**

j. Click the **Body** button.

k. In the **Event Action Expression Editor**, from the **Select a function** drop-down list, select **SUBSTITUTE**.

l. In the **Argument 1** field, type: **The credit limit has been changed to ${0} for customer {1}, customer number {2}**.

m. Place the cursor in the first row of the **Arguments** grid, and click **Build Expression**.

n. In the **Event Action Expression Editor**, from the **Select a function** drop-down list, select **FILTERPROPERTY**.

o. In the **Argument 1** field, type **CreditLimit** and click **OK**.

p. Place the cursor in the second row of the **Arguments** grid, and click **Build Expression**.

q. In the **Event Action Expression Editor**, from the **Select a function** drop-down list, select **FILTERPROPERTY**.

r. In the **Argument 1** field, type **Name** and click **OK**.

s. Place the cursor in the third row of the **Arguments** grid, and click **Build Expression**.

t. In the **Event Action Expression Editor**, from the **Select a function** drop-down list, select **FILTERPROPERTY**.

u. In the **Argument 1** field, type **CustNum** and click **OK**.
v. Click OK.

Notice that the system returns the entire SUBSTITUTE expression to the Event Action Send Email form, correctly formatted.

Notice also that there is no option to save the message to the user’s Sent Items folder. This is because this notification is being sent as an e-mail. That being the case, we cannot use the SAVEMESSAGE parameter to have the system save a copy of the notification in the Sent Items folder of the person who added the new customer.

w. Click OK.

x. Save the action and close the Event Actions form.

4. Verify that you have no syntax errors by clicking Check Syntax.

5. Discard the cached metadata.

For more information, including the procedure, see Discarding the Metadata Cache on page 58.

Test this event handler by changing a customer’s credit limit and saving the record. The system should generate an e-mail message that gets sent to the credit manager.

Points to Note and Remember

In creating this kind of event handler, keep the following in mind:

- To create an event handler that sends an e-mail, you must have the SMTP set up on the Intranets form. Also, the e-mail service on that computer must be set up to enable the relaying of e-mail automatically.
- To have the handler do something only when certain conditions are met, use the Finish action type and the CONDITION(NOT PROPERTYMODIFIED) parameter and function.
- To eliminate the single quotes that appear around replacement values in the generated messages, use the PROPERTY function in place of the FILTERPROPERTY function we used in this scenario.

Scenario 3: Notification That Includes an "Old" Value

In this scenario, we want to notify a group of inventory stock clerks automatically whenever an item’s lot size changes. In the message that is sent, we want to include both the previous lot size and the new lot size. We also want to let them know who initiated the change.

Once again, we will use an existing framework event, IdoOnItemUpdate, and create our own handler for it. Because this handler needs to retrieve the "before" property values, we must make it synchronous, so that during handler execution we can retrieve from the database the original row that is being updated, before it is updated by the IDO request.

This event handler requires three actions:

- One to check whether the Lot Size field has been changed and, if not, finish.
- One to retrieve the row being updated and both the original and new values for the Lot Size field.
A. One to send the notification to the inventory stock clerks.

To accomplish this scenario:

1. Create the event handler:
   a. Open the **Event Handlers** form.
   b. Press F3.
   c. Press CTRL+N.
   d. Create the handler with the following settings:

<table>
<thead>
<tr>
<th>Field or Option</th>
<th>Setting / Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Name</td>
<td>From the drop-down list, select <strong>IdoOnItemUpdate</strong>. <strong>NOTE:</strong> For details about this and the other framework events included with the system, see Framework Events on page B-4.</td>
</tr>
<tr>
<td>Applies to Initiators</td>
<td>Leave blank.</td>
</tr>
<tr>
<td>Applies to Objects</td>
<td>Enter <strong>SLItems</strong>. To determine what object you need, see the procedure provided in the online Help for this field.</td>
</tr>
<tr>
<td>Keep With</td>
<td>Leave blank.</td>
</tr>
<tr>
<td>Chronology</td>
<td>Leave blank.</td>
</tr>
<tr>
<td>Ignore Failure</td>
<td>Cleared.</td>
</tr>
<tr>
<td>Suspend</td>
<td>Cleared.</td>
</tr>
<tr>
<td>Synchronous</td>
<td>Selected. Because this notification requires both the original value and the new value, it must be run synchronously. For more information, see Synchronicity on page 2-4.</td>
</tr>
<tr>
<td>Active</td>
<td>Selected.</td>
</tr>
<tr>
<td>Can Override</td>
<td>Selected.</td>
</tr>
<tr>
<td>Transactional</td>
<td>Cleared.</td>
</tr>
<tr>
<td>Obsolete</td>
<td>Cleared.</td>
</tr>
<tr>
<td>Initial State</td>
<td>Leave blank.</td>
</tr>
<tr>
<td>Initial Action</td>
<td>Leave blank.</td>
</tr>
</tbody>
</table>

   e. Save the handler.

2. Create an event global constant for the group of stock clerks.

Putting the user IDs for the entire group into a global constant allows us to change the list, which might be used in other places as well, in a single place easily.

   a. Open the **Event Global Constants** form.
   b. Press F3 or F4.
   c. Press CTRL+N.
   d. In the Name field, enter the name for the global constant, in this case, **StockClerks**.
   e. In the Value field, enter the user IDs for the stock clerks, separated only by semi-colons (;) and no spaces.
   f. Save, and close the form.
3. Create the first action, which checks the condition of the Lot Size field when the item record is saved:
   a. In the Event Handlers form, select the handler.
   b. Click Event Actions.
   c. In the Event Actions form, in the Action Sequence field, enter 10.
   d. From the Action Type drop-down list, select Finish.
   e. Click Edit Parameters.
   f. In the Event Action Finish form, click the Condition button.
   g. In the Event Action Parameter Condition form, click the Expression 1 button.
   h. In the Event Action Expression Editor, from the Select a function drop-down list, select PROPERTYMODIFIED.
   i. In the Argument 1 field, type LotSize and then click OK.
   j. In the Event Action Parameter Condition form, select the NOT check box and then click OK.
   k. In the Event Action Finish form, click OK.
      This parameter tells the system to check the LotSize property. If it has not been modified, finish handler execution and exit. If it has been modified, continue with the second action.
   l. Save the action.

4. Create the second action, which retrieves both the original and new values for the Lot Size field.

   As part of this step, the system retrieves the original value of the Lot Size field and stores the value to an event variable named OldLotSize.

   For this action type to work, you must:
   - Use the IDO( ) function to identify the same IDO that fired the event.
   - Name the same property in the PROPERTIES( ) and SET( ) functions, and both should be the same as the field/component property value the user is changing.
   - Make sure the handler is synchronous.

   To create the action:
   a. Press CTRL+N.
   b. In the Action Sequence field, enter 20.
   c. From the Action Type drop-down list, select Load IDO Row.
   d. Click Edit Parameters.
e. In the Event Action Load IDO Row form, the IDO field, type: SLItems

   TIP: You can also select the IDO you want from the drop-down list, but you might have to filter on the field or increase the record cap for drop-down lists to see this one.

   Also, the procedure for figuring out what IDO collection you need is similar to the procedure for figuring out what property name you need. For more information, see Determining Names of IDO Collections and Components on page 57.

f. In the Properties field, type LotSize.

   g. Click the Filter button.

h. In the Event Action Expression Editor, from the Select a function drop-down list, select SUBSTITUTE.

i. In the Argument 1 field, type: Item = {0}

j. Place your cursor in the first Arguments row and then click Build Expression.

k. In the Event Action Expression Editor, from the Select a function drop-down list, select FILTERPROPERTY.

l. In the Argument 1 field, type Item and then click OK.

m. Click OK.

n. Click the Output button.

o. In the first row of the Event Action Output Parameters form, from the Output Type drop-down list, select Return Variable.

p. In the Output Object Name field, type: OldLotSize

q. In the Value field, type LotSize and then click OK.

r. In the Event Action Load IDO Row form, click OK.

   Your resulting syntax statement should appear as follows in the Parameters field of the Event Actions form:

   IDO("SLItems")
   PROPERTIES("LotSize")
   FILTER(SUBSTITUTE("Item = {0}", FP("Item")))
   SET(RV(OldLotSize) = "LotSize")

s. Verify that the action has no syntax errors.

   t. Save the action.

5. Create the third action, which sends the notification:

   a. To create the action, press CTRL+N.

   b. In the Action Sequence field, enter 30.

   c. From the Action Type drop-down list, select Notify.

   d. Click Edit Parameters.

   Setting up the TO parameter:

   a. In the Event Action Notify form, click the To button.

   b. In the Event Action Parameter Recipients form, click Recipients.
c. In the Event Action Expression Editor, from the Select a function drop-down list, select GC.
   The GC function allows you to designate an event global constant to use for the recipients. In this case, we will designate the global constant we created earlier, StockClerks.

d. From the Argument 1 drop-down list, select StockClerks and then click OK.
e. In the Event Action Parameter Recipients form, click OK.

Setting up the CC, SUBJECT, and CATEGORY parameters:

a. Click Cc button.
b. In the Event Action Parameter Recipients form, click Recipients.
c. In the Event Action Expression Editor, from the Select a function drop-down list, select ORIGINATOR.
   Notice that the ORIGINATOR function takes no arguments.
d. Click OK twice.
e. In the Subject field, type: Lot size change
f. In the Category field, type: Change Notification

Setting up the BODY parameter:

a. Click the Body button.
b. In the Event Action Expression Editor, from the Select a function drop-down list, select SUBSTITUTE.
c. In the Argument 1 field, type: The lot size has been changed for item: {1} The previous lot size was {2}, the new lot size is {3}. Please take note and adjust your activities accordingly. This change was made by {0}.
d. Place your cursor in the first row of the Arguments grid and then click Build Expression.
e. In the Event Action Expression Editor, from the Select a function drop-down list, select ORIGINATOR and then click OK.
   Notice that we can place this first property (ORIGINATOR) last in the message, and, as long as we have the appropriate index number assigned ( {0} ), it will be correctly displayed in the message.
f. Place your cursor in the second row of the Arguments grid and then click Build Expression.
g. In the Event Action Expression Editor, from the Select a function drop-down list, select FP.
h. From the Argument 1 drop-down list, select Item and then click OK.
   In this substep, because we want the item number to be enclosed in single quote marks, we use the FP (alternate for the FILTERPROPERTY) function. However, we do not want single quote marks around the lot size amounts, so we will let those values evaluate as their native datatypes.
i. Place your cursor in the third row of the Arguments grid and then click Build Expression.
j. In the Event Action Expression Editor, from the Select a function drop-down list, select V from the list of functions.

k. In the Argument 1 field, type OldLotSize and then click OK.

l. Place your cursor in the fourth row of the Arguments grid and then click Build Expression.

m. In the Event Action Expression Editor, from the Select a function drop-down list, select P (or PROPERTY) from the list of functions.

n. From the Argument 1 drop-down list, select LotSize and then click OK.

o. In the Event Action Expression Editor, click OK.

The resulting syntax statement looks similar to the following:

```
BODY(SUBSTITUTE("The lot size has been changed for item: {1} The previous
lot size was {2}, the new lot size is {3}. Please take note and adjust your
activities accordingly. This change was made by {0}.", ORIGINATOR(),
FP("Item"), V(OldLotSize),
P("LotSize"))
```

TIP: If you want, you can add line returns to make your syntax statement look like this example. The system ignores white space and line returns when processing the statements.

p. In the Event Action Notify form, click OK.

6. Verify that you have no syntax errors.

7. Save the action and close the Event Actions form.

8. Discard the cached metadata.

   For more information, including the procedure, see Discarding the Metadata Cache on page 58.

**Testing the Event Handler**

To test this handler:

1. Open the Items form and locate an item that is lot-tracked.

2. On the General tab, change the value in the Lot Size field and save the record.

3. In the stockers’ Inbox forms, verify that the message was sent and contains the correct values.

**Points to Note and Remember**

In creating this kind of event handler, keep the following in mind:

- To retrieve the previous (existing) value of a field for display in a message, you must make the handler synchronous.
- To display both the original value of a field and the new one, use an event variable to temporarily store the original value.
Substituted values in a statement can be presented in any order as long as their index numbers match their positions in the list.

To prevent single quotes from being placed around a substituted value, use the PROPERTY function instead of the FILTERPROPERTY function.

Extra Challenge

Try on your own adding the item description to the body of the message. This might require you to determine the name of the item description property, if you do not already know it. For more information, see Determining Names of IDO Collections and Components on page 57.
Requesting Approvals

The following two scenarios, which are similar, build on the concepts and practices used in the first two scenarios. The big difference is that they both use the Prompt action type to ask a manager to approve some action.

TIP: Before trying these scenarios, we recommend that you go through the scenarios so far, if you have not already done so. These scenarios assume that you already know how to accomplish certain tasks without necessarily taking you through them in detail.

**Scenario 4: Approval for a New Record**

In this scenario, we want to send a message to the purchasing manager whenever a purchase order (PO) is added. This message prompts the purchasing manager for approval of the PO. The purchasing manager can indicate approval (or disapproval) by clicking a button in the message itself. If approved, the system adds the PO; if not, the system does not add the PO.

As with the previous scenarios so far, the system framework has a built-in event, **IdOnItemInsert**, which we can use with our handler. The handler itself requires two actions: one to send the prompt, and one to tell the system what to do if approval is denied.

To set the system up to handle this:

1. Create a global constant for the purchasing manager.
   a. For the **Name** of the global constant, use **PurchasingMgr**.
   b. For the **Value** field, enter the purchasing manager’s user ID.
      Remember: To add multiple recipients, enter user IDs separated by semi-colons (;) and no spaces.

2. Create and save an event handler with the following settings:

<table>
<thead>
<tr>
<th>Field or Option</th>
<th>Setting / Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Name</td>
<td>From the drop-down list, select <strong>IdOnItemInsert</strong>. <strong>NOTE:</strong> For details about this and the other framework events included with the system, see Framework Events on page B-4.</td>
</tr>
<tr>
<td>Applies to Initiators</td>
<td>Leave blank.</td>
</tr>
<tr>
<td>Applies to Objects</td>
<td>Enter <strong>SLPos</strong>. To determine what object you need, see the procedure provided in the online Help for this field.</td>
</tr>
<tr>
<td>Keep With</td>
<td>Leave blank.</td>
</tr>
<tr>
<td>Chronology</td>
<td>Leave blank.</td>
</tr>
<tr>
<td>Ignore Failure</td>
<td>Cleared.</td>
</tr>
</tbody>
</table>
For more information about any of these fields and options, see the online Help for the Event Handlers form.

3. Create the first action, to send the message:

   a. In the Event Actions form, in the Action Sequence field, enter 10.
   b. From the Action Type drop-down list, select Prompt and then click Edit Parameters.
      This action type not only sends a notification to the designated recipient, it also prompts the recipient for a response.
   c. Starting with the Event Action Prompt form, use the associated forms to create the following parameters:

<table>
<thead>
<tr>
<th>Field or Option</th>
<th>Setting / Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suspend</td>
<td>Cleared. Because this notification does require a response from the purchasing manager, it must run synchronously and be suspended. However, we cannot select this (and make it &quot;stick&quot;) until at least one adjourning action exists. So, we must leave this cleared for now and come back to it after our actions have been defined. For more information, see Suspension on page 2-5.</td>
</tr>
<tr>
<td>Synchronous</td>
<td>Doesn’t matter at this point. Because this notification does require a response from the purchasing manager, it must run synchronously and be suspended. When you later select the Suspend option, this will be automatically selected. For more information, see Synchronicity on page 2-4.</td>
</tr>
<tr>
<td>Active</td>
<td>Selected.</td>
</tr>
<tr>
<td>Can Override</td>
<td>Selected.</td>
</tr>
<tr>
<td>Transactional</td>
<td>Cleared.</td>
</tr>
<tr>
<td>Obsolete</td>
<td>Cleared.</td>
</tr>
<tr>
<td>Initial State</td>
<td>Leave blank.</td>
</tr>
<tr>
<td>Initial Action</td>
<td>Leave blank.</td>
</tr>
</tbody>
</table>

For more information about any of these fields and options, see the online Help for the Event Handlers form.

<table>
<thead>
<tr>
<th>Field / Button</th>
<th>Action</th>
<th>Result / Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>To button</td>
<td>Click.</td>
<td>The Event Action Parameter Recipients form opens.</td>
</tr>
<tr>
<td>Recipients</td>
<td>Click button.</td>
<td>The Event Action Expression Editor opens.</td>
</tr>
<tr>
<td>Select a function</td>
<td>Select GC.</td>
<td>The Argument 1 field and drop-down list display.</td>
</tr>
<tr>
<td>Argument 1 drop-down list</td>
<td>Select PurchasingMgr.</td>
<td>The system returns the expression to the Event Action Parameter Recipients form.</td>
</tr>
<tr>
<td></td>
<td>This is the global constant we created earlier.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Click OK.</td>
<td></td>
</tr>
<tr>
<td>OK button</td>
<td>Click.</td>
<td>The system returns the expression to the Event Action Prompt form.</td>
</tr>
<tr>
<td>Subject field</td>
<td>Enter: New purchase order needs your approval</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Category field</td>
<td>Enter: Order Approval</td>
<td>—</td>
</tr>
<tr>
<td>Body button</td>
<td>Click.</td>
<td>The Event Action Expression Editor form opens.</td>
</tr>
<tr>
<td>Select a function</td>
<td>Select SUBSTITUTE.</td>
<td>The Argument 1 field and drop-down list, Arguments grid, and buttons display.</td>
</tr>
<tr>
<td>Argument 1 field</td>
<td>Enter: A new purchase order has been requested for vendor {0}, {1}. Please review the details on the Variables tab and register your approval on the Response tab.</td>
<td>—</td>
</tr>
<tr>
<td>Arguments grid, row 1</td>
<td>With cursor in the field, click Build Expression.</td>
<td>The Event Action Expression Editor form opens.</td>
</tr>
<tr>
<td>Select a function</td>
<td>Select P or PROPERTY.</td>
<td>The Argument 1 field and drop-down list display.</td>
</tr>
<tr>
<td>Argument 1 field</td>
<td>Select VendNum and click OK.</td>
<td>The system returns the expression to the parent Event Action Expression Editor.</td>
</tr>
<tr>
<td>Arguments grid, row 2</td>
<td>With cursor in the field, click Build Expression.</td>
<td>The Event Action Expression Editor form opens.</td>
</tr>
<tr>
<td>Select a function</td>
<td>Select P or PROPERTY.</td>
<td>The Argument 1 field and drop-down list display.</td>
</tr>
<tr>
<td>Argument 1 field</td>
<td>Select VendorName and click OK.</td>
<td>The system returns the expression to the parent Event Action Expression Editor.</td>
</tr>
<tr>
<td>OK button</td>
<td>Click.</td>
<td>The system returns the entire BODY parameter content to the Event Action Prompt form.</td>
</tr>
<tr>
<td>Question field</td>
<td>Enter: Do you approve this new PO?</td>
<td>Note that, when the handler runs, the QUESTION parameter is presented on the Response tab of the recipient’s Inbox.</td>
</tr>
<tr>
<td>Choices button</td>
<td>Click.</td>
<td>The Event Action Prompt Choices form opens. Note that the CHOICES parameter creates, displays, and enables the voting buttons that will be required for the purchasing manager to signal approval or rejection. When the handler runs, these CHOICES buttons appear directly beneath the QUESTION in the recipient’s Inbox.</td>
</tr>
<tr>
<td>Return Value field, row 1</td>
<td>Enter: 1</td>
<td>Note that you can enter any value you want here.</td>
</tr>
<tr>
<td>Button Caption field, row 1</td>
<td>Enter: sYes</td>
<td>Note that you can use translatable strings from the Strings table. These strings appear in the drop-down list for this field.</td>
</tr>
<tr>
<td>Return Value field, row 2</td>
<td>Enter: 0</td>
<td>—</td>
</tr>
<tr>
<td>Button Caption field, row 2</td>
<td>Enter: sNo</td>
<td>—</td>
</tr>
<tr>
<td>OK button</td>
<td>Click.</td>
<td>The system returns the choices data to the Event Action Prompt form.</td>
</tr>
<tr>
<td>OK button</td>
<td>Click.</td>
<td>The system returns all defined parameters to the Event Actions form, correctly formatted.</td>
</tr>
</tbody>
</table>

d. Verify that there are no syntax errors.
e. Save the action.

If you have done everything correctly, your syntax for this action step should look like the following:

```
TO(GC(PurchasingMgr))
SUBJECT("New purchase order needs your approval")
CATEGORY("Order Approval")
BODY(SUBSTITUTE("A new purchase order has been requested for vendor {0}, {1}. Please review the details on the Variables tab and register your approval on the Response tab.", P("VendNum"), P("VendorName") )
SAVEMESSAGE(FALSE)
QUESTION("Do you approve this new PO?")
CHOICES("1,sYes,0,sNo")
```

4. Create the second action, which tells the system how to respond if approval is not granted:

a. In the **Action Sequence** field, enter **20**.

b. From the **Action Type** drop-down list, select **Fail**.

   This action type ends handler execution with an error status. This effectively aborts the process and prevents the PO from being added to the database.

c. Starting with the **Event Action Fail** form, use the associated forms to create the following parameters:

<table>
<thead>
<tr>
<th>Field / Button</th>
<th>Action</th>
<th>Result / Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition button</td>
<td>Click.</td>
<td>The <strong>Event Action Parameter Condition</strong> form opens.</td>
</tr>
<tr>
<td>Expression 1 button</td>
<td>Click.</td>
<td>The <strong>Event Action Expression Editor</strong> opens.</td>
</tr>
<tr>
<td>Select a function</td>
<td>Select <strong>VOTINGRESULT</strong>.</td>
<td>The <strong>Action</strong> drop-down list displays. This function and action evaluate the results of whatever action is selected from the list. The action refers by number to the action type. In this case, we have only one other action, and that is the correct action, the Prompt action.</td>
</tr>
<tr>
<td>Action drop-down list</td>
<td>Select <strong>10 Prompt</strong> and click <strong>OK</strong>.</td>
<td>The system returns the expression to the <strong>Expression 1</strong> field on the <strong>Event Action Parameter Condition</strong> form.</td>
</tr>
<tr>
<td>Operator drop-down list</td>
<td>Select <strong>=</strong> (equals).</td>
<td>—</td>
</tr>
<tr>
<td>Expression 2 field</td>
<td>Enter: <strong>0</strong> (zero) and click <strong>OK</strong>.</td>
<td>This value tells the system to fail the handler with an error if the recipient responds with a &quot;No&quot; (0). The system returns the entire condition statement to the <strong>Event Action Fail</strong> form.</td>
</tr>
<tr>
<td>Result field</td>
<td>Enter: <strong>The PO request was rejected by the purchasing manager.</strong></td>
<td>This message appears on the <strong>Event Status</strong> form if the purchasing manager responds with a &quot;No.&quot;</td>
</tr>
<tr>
<td>OK button</td>
<td>Click.</td>
<td>The system returns both parameters to the <strong>Event Actions</strong> form, correctly formatted.</td>
</tr>
</tbody>
</table>

d. Verify that there are no syntax errors and save the action.
5. Return to the Event Handlers form and select the Suspend check box.
6. Save the handler.
7. Discard the cached metadata.
   For more information, including the procedure, see Discarding the Metadata Cache on page 58.

Testing the Event Handler

To test this handler:

1. Open the Purchase Orders form and create a new purchase order. Save it.
   After you save the PO, when the Purchase Orders refreshes the display, the new record should disappear from the display. It remains hidden until/unless it has been approved.
   If you do not assign a PO number, the generated message displays it with a PO number of TBD.

2. Open the Inbox form for the individual designated as the Purchasing Manager and verify that the message was received and that the Response tab displays the question and choice buttons.

3. (Optional) With the Purchase Orders form selected, from the Actions menu, select View Event Status.
   This opens the Event Status form. Navigate to the last row and verify that the status for this event is Running.

4. In the Inbox form, click the button labeled Yes.
5. Refresh the collection on the Purchase Orders form and verify that the new PO now displays in the list.

You can also do a second test, clicking the button labeled No to reject the request. In this case, when you refresh the Purchase Orders form, the new PO record is never added to the database and does not appear in the list of POs.

Points to Note and Remember

When creating this kind of event handler, keep the following in mind:

- When creating a message that requires a response from the recipient (usually a Prompt action type), you must mark the handler so that it suspends when executed. This means that it is also automatically marked as a synchronous handler.
- Because these event handlers must be suspended, pending the purchasing or credit manager’s response, the Framework Event Service must be enabled for the configuration in which you are logged on.

Extra Challenge

Try changing the Subject line so that it displays the user ID of the person who created the new PO and the PO number.
Hint: You will need to use the SUBSTITUTE function.

**Scenario 5: Requesting Approval by External E-mail for Changes to an Existing Record**

This scenario is similar to the one in **Scenario 2: Notification of Changes to an Existing Record—Changing the Credit Limit** on page 82, except that in this case, we want the credit manager’s approval for the credit limit change, and we are sending the request to the manager’s external e-mail address. If the credit manager approves the change, the system writes and saves it. If the credit manager does not approve the change, the system rolls back the record to the previously approved credit limit.

As with the other scenarios so far, we can use existing framework events and IDOs to accomplish this. This time, however, because we are sending out a prompt and requiring a response from the credit manager, we must make it a synchronous and suspending event handler.

This event handler requires three actions:

- One checks whether the **Credit Limit** field has been changed. If it has *not*, it finishes with a status of Success.
- One sends the prompt message and external e-mail.
- If the credit manager does *not* approve the change, the third one fails the event and rolls back the record.

To accomplish this scenario:

1. Set up the recipient in the **Users** form to allow external e-mail and to have the appropriate default language code.
   - Select **Send E-mail Prompts**
   - Ensure that the **E-mail Address** is correct
   - Specify the **Default Language** to use for formatting text strings
2. Create an event handler with the following settings:
   - **Event Name** = **IdoOnItemUpdate**
   - **Applies to Objects** = **SLCustomers**
3. Save the handler.

   **TIP:** If you still have the handler you created for **Notification of Changes to an Existing Record—Changing the Credit Limit** on page 82 active, it is a good idea to clear the **Active** check box or mark it as **Obsolete**, so that this event handler and that one do not create duplicate and possibly confusing messages.

4. Create the first action, which checks the condition of the **Credit Limit** field when the customer record is saved:
   a. Click **Event Actions** for the handler you just created.
   b. In the **Event Actions** form, create a new action with the following settings:
      - **Action Sequence** = **10**
      - **Action Type** = **Finish**
c. Click **Edit Parameters**.

d. Starting with the **Event Action Finish** form, use the associated forms to create the following parameters:

<table>
<thead>
<tr>
<th>Field / Button</th>
<th>Action</th>
<th>Result / Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition button</td>
<td>Click.</td>
<td>The Event Action Parameter Condition form opens.</td>
</tr>
<tr>
<td>Expression 1 button</td>
<td>Click.</td>
<td>The Event Action Expression Editor opens.</td>
</tr>
<tr>
<td>Select a function</td>
<td>Select PROPERTYMODIFIED.</td>
<td>The system displays the Argument 1 button and field.</td>
</tr>
</tbody>
</table>
| Argument 1 field    | Enter: CreditLimit
Then click OK. | The system returns the expression to the parent Event Action Parameter Condition form and disables the Operator and Expression 2 options. |
| NOT check box       | Select. Then click OK. | This tells the action to finish with a status of Success if the Credit Limit field has *not* been changed. The system returns the expression to the Event Action Finish form. |
| OK button           | Click. | The system returns the entire parameter to the Event Actions form, correctly formatted. |

e. Verify that the syntax is correct.

f. Save the action.

5. Create the second action, to send the prompt message.

This action sends the prompt to the credit manager, through both the SyteLine Inbox and external e-mail, and suspends the handler until the credit manager responds to the request.

a. In the **Event Actions** form, create a new action with the following settings:

- **Action Sequence** = 20
- **Action Type** = Prompt

b. Click **Edit Parameters**.

c. Starting with the **Event Action Prompt** form, use the associated forms to create the following parameters:

<table>
<thead>
<tr>
<th>Field / Button</th>
<th>Action</th>
<th>Result / Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>To button</td>
<td>Click.</td>
<td>The Event Action Parameter Recipients form opens.</td>
</tr>
<tr>
<td>Recipients button</td>
<td>Click.</td>
<td>The Event Action Expression Editor form opens.</td>
</tr>
<tr>
<td>Select a function</td>
<td>Select GC.</td>
<td>The system displays the Argument 1 button and field.</td>
</tr>
</tbody>
</table>
| Argument 1 dropdown list | Select CreditMgr.
Then click OK. | The system returns the expression to the Event Action Parameter Recipients form. |
<p>| OK button          | Click. | The system returns the expression to the Event Action Prompt form. |
| Subject button     | Click. | The Event Action Expression Editor form opens. |
| Select a function  | Select SUBSTITUTE. | The system displays the buttons and fields associated with the SUBSTITUTE function. |</p>
<table>
<thead>
<tr>
<th>Field / Button</th>
<th>Action</th>
<th>Result / Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument 1 field</td>
<td>Enter: Credit limit change request for customer ID: {0}</td>
<td>Notice that the SUBSTITUTE function is being used to present the customer’s ID number in the Subject line, so that messages can be saved and tracked more easily.</td>
</tr>
<tr>
<td>Arguments grid, row 1</td>
<td>Place the cursor in the field and then click Build Expression.</td>
<td>The Event Action Expression Editor form opens.</td>
</tr>
<tr>
<td>Select a function</td>
<td>Select P or PROPERTY. NOTE: These are equivalent functions.</td>
<td>The system displays the Argument 1 button and field.</td>
</tr>
<tr>
<td>Argument 1 drop-down list</td>
<td>Select CustNum. Then click OK.</td>
<td>The system returns the expression to the parent Event Action Expression Editor form.</td>
</tr>
<tr>
<td>OK button</td>
<td>Click.</td>
<td>The system returns the entire expression to the Event Action Prompt form.</td>
</tr>
<tr>
<td>Category field</td>
<td>Enter: Financial</td>
<td>—</td>
</tr>
<tr>
<td>Body button</td>
<td>Click.</td>
<td>The Event Action Expression Editor form opens.</td>
</tr>
<tr>
<td>Select a function</td>
<td>Select SUBSTITUTE.</td>
<td>The system displays the buttons and fields associated with the SUBSTITUTE function.</td>
</tr>
<tr>
<td>Argument 1 field</td>
<td>Enter: You have a request for a credit limit change to ${0} for {1}, Customer ID {2}. Please respond to the question and indicate your approval on the Response tab.</td>
<td>This sets up the basic message with three replacement markers.</td>
</tr>
<tr>
<td>Arguments grid, row 1</td>
<td>Place the cursor in the field and then click Build Expression.</td>
<td>The Event Action Expression Editor form opens.</td>
</tr>
<tr>
<td>Select a function</td>
<td>Select P.</td>
<td>The system displays the Argument 1 button and field.</td>
</tr>
<tr>
<td>Argument 1 drop-down list</td>
<td>Select CreditLimit. Then click OK.</td>
<td>The system returns the expression to the first row of Arguments grid on the parent Event Action Expression Editor form.</td>
</tr>
<tr>
<td>Arguments grid, row 2</td>
<td>Place the cursor in the field and then click Build Expression.</td>
<td>The Event Action Expression Editor form opens.</td>
</tr>
<tr>
<td>Select a function</td>
<td>Select P.</td>
<td>The system displays the Argument 1 button and field.</td>
</tr>
<tr>
<td>Argument 1 drop-down list</td>
<td>Select Name. Then click OK.</td>
<td>The system returns the expression to the second row of Arguments grid on the parent Event Action Expression Editor form.</td>
</tr>
<tr>
<td>Arguments grid, row 3</td>
<td>Place the cursor in the field and then click Build Expression.</td>
<td>The Event Action Expression Editor form opens.</td>
</tr>
<tr>
<td>Select a function</td>
<td>Select P.</td>
<td>The system displays the Argument 1 button and field.</td>
</tr>
<tr>
<td>Argument 1 drop-down list</td>
<td>Select CustNum. Then click OK.</td>
<td>The system returns the expression to the third row of Arguments grid on the parent Event Action Expression Editor form.</td>
</tr>
<tr>
<td>OK button</td>
<td>Click.</td>
<td>The system returns the entire SUBSTITUTE expression to the Event Action Prompt form.</td>
</tr>
<tr>
<td>Question field</td>
<td>Enter: Do you approve this credit limit change?</td>
<td>Note that you have an 80-character limit in the Question field.</td>
</tr>
<tr>
<td>Choices button</td>
<td>Click.</td>
<td>The Event Action Prompt Choices form opens.</td>
</tr>
</tbody>
</table>
NOTE: You could incorporate other field values from the Event Action Prompt form before saving and closing, but in the interest of relative brevity, we will finish with what we have here.

d. Verify that the syntax is correct.
e. Save the action.

6. Create the third action, which tells the system how to respond if approval is not granted.

This parameters tells the system to consider the action as having failed if the credit manager rejects the credit limit change. In other words, if the credit manager votes "No" [0] on the second action (Action Sequence = 20), then this action fails.

a. In the Event Actions form, create a new action with the following settings:

- **Action Sequence = 30**
- **Action Type = Fail**

This action type ends handler execution with an error status. This effectively aborts the process and prevents the credit limit from being changed for the customer.

b. Click Edit Parameters.

c. Starting with the Event Action Fail form, use the associated forms to create the following parameters:

<table>
<thead>
<tr>
<th>Field / Button</th>
<th>Action</th>
<th>Result / Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Value, row 1</td>
<td>Enter 1.</td>
<td>TIP: Theoretically, you can use any value you want here, as long as you remember what it is and use the same value later in the Fail action step.</td>
</tr>
<tr>
<td>Button Caption, row 1</td>
<td>Enter sYes.</td>
<td>Notice that this is a translatable string from the Strings table. Also notice that you can select a string from the Strings table from the drop-down list.</td>
</tr>
<tr>
<td>Return Value, row 2</td>
<td>Enter 0.</td>
<td>—</td>
</tr>
<tr>
<td>Button Caption, row 2</td>
<td>Enter sNo.</td>
<td>—</td>
</tr>
<tr>
<td>OK button</td>
<td>Click.</td>
<td>The system returns the Choices values to the Event Action Prompt form.</td>
</tr>
<tr>
<td>Save in Sent Items</td>
<td>Select.</td>
<td>This sends a copy of the message to whomever initiated the credit limit change.</td>
</tr>
<tr>
<td>OK button</td>
<td>Click.</td>
<td>The system returns the entire set of Prompt action parameters to the Event Actions form.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field / Button</th>
<th>Action</th>
<th>Result / Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition button</td>
<td>Click.</td>
<td>The Event Action Parameter Condition form opens.</td>
</tr>
<tr>
<td>Expression 1 button</td>
<td>Click.</td>
<td>The Event Action Expression Editor opens.</td>
</tr>
<tr>
<td>Select a function</td>
<td>Select VOTINGRESULT.</td>
<td>The system displays the Action drop-down list field.</td>
</tr>
<tr>
<td>Action drop-down list</td>
<td>Select 20 Prompt.</td>
<td>Notice that the system displays only the 20 in the field. The action type name in the drop-down list is there to help you select the correct action step.</td>
</tr>
<tr>
<td>OK button</td>
<td>Click.</td>
<td>The system returns the expression to the Event Action Parameter Condition form.</td>
</tr>
</tbody>
</table>
NOTE: You could also at this point set another event action to notify the original sender by message that the change has been approved or disapproved, but that is beyond the scope for this exercise.

d. Verify that there are no syntax errors.

e. Save the action and close the Event Actions form.

7. Return to the Event Handlers form and select the Suspend check box.

8. Save the handler.

9. Discard the cached metadata.

For more information, including the procedure, see Discarding the Metadata Cache on page 58.

Testing This Event Handler

To test this handler:

1. Open the Customers form and change the credit limit for a customer. Save it.

   Notice that the entire record for this customer is now temporarily disabled, because the update has been suspended pending approval. Therefore, no further changes can be made to this customer record until this event is resolved one way or the other.

   Also notice that, at this point, all fields, including the Credit Limit field, display their original values. Anyone (including you) who views this suspended record sees the original values, until this suspended event finishes successfully, at which time the new values are saved in the database and displayed on the Customers form.

2. (Optional) Open the Saved Messages form for your current logon ID and verify that a copy of the message has been saved there.

3. Open the Inbox form for the individual designated as the credit manager and verify that the message was received and that the Response tab displays the question and choice buttons.
4. Open the credit manager’s external e-mail system and verify that the e-mail was received, and the question and choice links display.

5. (Optional) With the Customers form open and customer record that was changed selected, from the Actions menu, select View Event Status.

   This opens the Event Status form. Verify that the status for this event is Running. You can also open the Event Status form manually.

6. In the credit manager’s e-mail, click the link labeled Yes. Verify that the ASP processes the message and returns a success response.

7. In the credit manager’s SyteLine Inbox form, verify that the message is automatically marked as Expired, the Choices buttons are now disabled, and the Selected Choice is Yes.

8. Refresh the collection on the Customers form and verify that the new credit limit was saved.

   Notice too that the entire customer record is once again enabled for editing.

You can also do a second test, clicking the link labeled No to reject the request. In this case, when you refresh the Customers form, notice that the Credit Limit field has retained its original amount.

Points to Note and Remember

In creating this kind of event handler, keep the following in mind:

- You can use the SUBSTITUTE function other places than just in the body of a message. You can use it in the Subject line and other places. And, as we will see later, you can also use it for purposes other than replacing text in messages.

- When checking on a voting result, the number referred to in the syntax is the action sequence number for the action that contains the choice.

Extra Challenges

- Change the body of the message to include both the original credit limit and the proposed new limit.

  Hint: You will need to save the old credit limit in an event variable.

- Create another event action to notify the original sender by message that the change has been approved or disapproved.

Scenario 6: Requesting Multiple and Complex Approvals

In this scenario, when a purchase order (PO) status is changed to Ordered, it requires the approval of the purchasing manager. At the same time, if the PO is for more than $100,000, the PO also requires the approval of the purchasing manager’s supervisor. Finally, if the PO is for more than $1,000,000, the PO requires the further approval of two senior executives.

If the PO is disapproved at any level, the PO is rolled back to the previous values, and any changes made to it are lost. While it is in the process of being approved, the PO remains
suspended until approval or disapproval is determined. This means, among other things, that if one or more approvers fail to respond to the request, the PO is locked and cannot be changed until all required approvers respond.

The following flow diagram illustrates what must happen with this handler:
For this scenario we will:

- Use the same global constant for the purchasing manager (PurchasingMgr) that we used for a previous scenario (Scenario 4: Approval for a New Record on page A-10).
- Use the IdoOnItemUpdate framework event for the SLPos IDO. This will cause the event to be generated whenever a PO record is updated either in the Purchase Orders form or by another process that attempts to update that record.
- Pass property values and an identifying property (ItemId) to the event as input parameters so the system can store them with the event as event parameters.

To accomplish this scenario:

1. Create and save an event handler with the following settings:
   - **Event Name** = IdoOnItemUpdate
   - **Applies to Objects** = SLPos

2. Create the require event global constants to make this scenario work. In the Event Global Constants form, create the following global constants:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Comments</th>
</tr>
</thead>
</table>
| PurchasingMgr      | userID1    | This is the logon user ID for the purchasing manager required for initial approval.
   |             | This can be the same as the one we created for Scenario 4: Approval for a New Record on page A-10.
   |             | If you have multiple recipients, separate them with semi-colons (;) and no spaces. |
| PurchasingSuper    | userID2    | This is the logon user ID for the purchasing supervisor required to approve POs over $100,000. |
| PurchasingSenior   | userId3;userId4 | These are the logon user IDs for the two senior executives required to approve POs over $1,000,000. |
| SuperCost          | 100000     | This global constant represents the minimum amount that must be approved by a supervisor. We are using a global constant for it so that it can be changed globally at some future time. The value is a literal amount, no commas. |
| SeniorCost         | 1000000    | This global constant represents the minimum amount that must be approved by two senior-level executives. We are using a global constant for it so that it can be changed globally at some future time. The value is a literal amount, no commas. |
| POApprovalPrompt   | [See below.]| Note that, because we want to use the same basic prompt for all levels of approvals, we are putting it into a global constant. This is accomplished with the use of a few variables. |
Value for **POApprovalPrompt**:

```plaintext
SUBJECT(" Purchase Order Update Approval Needed")
CATEGORY("Order Approval")
TO(GC(TV(Approver)))
BODY(SUBSTITUTE("A purchase order, {0}, has been updated to Ordered
status for vendor, {1}, number: {2}. Please review the details on the Variables
tab and then indicate your approval on the Response tab.",
FP("PoNum"),
P("VendorName"),
FP("VendNum")))
QUESTION("Do you approve this PO change?")
CHOICES("1, sYes, 0, sNo")TV(CountMethod)
FILTERFORM("PurchaseOrders")
FILTER(SUBSTITUTE("PoNum={0}", FP("PoNum")))
```

3. Using the **Event Actions** form, add the first event action:

- **Action Sequence** = 1
- **Action Type** = Finish

   a. Starting with the **Edit Parameters** button, use the event action parameter forms to complete the event action:

<table>
<thead>
<tr>
<th>Field / Button</th>
<th>Action</th>
<th>Result / Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition button</td>
<td>Click.</td>
<td>The Event Action Parameter Condition form opens.</td>
</tr>
<tr>
<td>Expression 1 button</td>
<td>Click.</td>
<td>The Event Action Expression Editor opens.</td>
</tr>
<tr>
<td>Select a function</td>
<td>Select PROPERTYMODIFIED.</td>
<td>The system displays the Argument 1 button and field.</td>
</tr>
<tr>
<td>Argument 1 field</td>
<td>Enter Stat.</td>
<td></td>
</tr>
<tr>
<td>OK button</td>
<td>Click.</td>
<td>The system returns the expression to the Event Action Parameter Condition form.</td>
</tr>
<tr>
<td>NOT check box</td>
<td>Select.</td>
<td></td>
</tr>
<tr>
<td>Condition field</td>
<td>Add the following text: OR P(&quot;Stat&quot;) &lt;&gt; &quot;O&quot;</td>
<td>NOTE: The reason this is required is that the Event Action Parameter Condition form can only be used to construct simple condition statements. For complex conditions, you can start with that form, but you must then manually edit the condition statement. The &quot;O&quot; in this case is the capital letter, not a zero. The final result of this condition is: If the Stat property has not been modified, or if the value of the Stat field is not O, then finish. Or, in other words, if the Stat property has been modified and the value of the field is O, then continue to the next action.</td>
</tr>
<tr>
<td>OK button</td>
<td>Click.</td>
<td>The system returns the expression to the Event Action Finish form.</td>
</tr>
</tbody>
</table>

   b. Check the syntax and save the action.

4. Add the second event action:

- **Action Sequence** = 2
- **Action Type** = Set Values
a. Starting with the **Edit Parameters** button, use the event action parameter forms to complete the event action:

<table>
<thead>
<tr>
<th>Field / Button</th>
<th>Action</th>
<th>Result / Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables button</td>
<td>Click.</td>
<td>The Event Action Set Name/Value Pairs form opens.</td>
</tr>
<tr>
<td>Variable Name column, row 1</td>
<td>Type: Approver</td>
<td>—</td>
</tr>
<tr>
<td>Value column, row 1</td>
<td>Click Build Expression.</td>
<td>The Event Action Expression Editor opens.</td>
</tr>
<tr>
<td>Select a function drop-down list</td>
<td>Select GC.</td>
<td>The form displays the Argument 1 button and drop-down list.</td>
</tr>
<tr>
<td>Argument 1 drop-down list</td>
<td>Select PurchasingMgr and then click OK.</td>
<td>The system returns to the Event Action Set Name/Value Pairs form.</td>
</tr>
<tr>
<td>Variable Name column, row 2</td>
<td>Type: CountMethod</td>
<td>—</td>
</tr>
<tr>
<td>Value column, row 2</td>
<td>Type: VOTINGRULE(Plurality)</td>
<td>Although this looks like a function, the system in this case does not treat VOTINGRULE() like the other functions.</td>
</tr>
<tr>
<td>OK button.</td>
<td>Click.</td>
<td>The system returns to the Event Action Set Values form.</td>
</tr>
<tr>
<td>OK button.</td>
<td>Click.</td>
<td>The system returns the parameters to the Event Actions form, correctly formatted.</td>
</tr>
</tbody>
</table>

This action step sets the values of two variables as follows:

- The variable named **Approver** is set to the value of the global constant, **PurchasingMgr**, which is the user ID for the purchasing manager.
- The variable named **CountMethod** is set to count the votes using the **Plurality** rule, which simply says that the choice with the greatest number of votes wins. Since we have only one individual set to vote at this point, the purchasing manager’s vote alone determines what happens next.

**TIP:** If you have more than one user ID in the **PurchasingMgr** global constant, you might want to use a different voting rule. For more information about voting rules, see Voting Rules on page 5-5.

b. Check the syntax and save the action.

5. Add the third event action:

- **Action Sequence = 3**
- **Action Type = Prompt**
In the **Parameters** text input field, enter the following:

```sql
TGC(POApprovalPrompt)
```

NOTE: Because TGC( ) is a pre-parser function, you cannot use the event action parameter forms to create or set it. You must enter this statement directly in the **Parameters** field of the **Event Actions** form. This also means that you cannot use the **Check Syntax** button to check the syntax.

This statement performs a text evaluation of the **POApprovalPrompt** global constant to obtain the parameters for a prompt action. Part of this text evaluation includes an evaluation and insertion of the values for the two variables we set in the previous step.

After evaluating the **POApprovalPrompt** global constant, this action also sends out the prompt message to the purchasing manager and suspends the handler pending the manager’s response. Because we did not specify any Variable Access rules, the message allows the purchasing manager to modify any variable values before approving it.

b. Save the action.

6. Add the fourth event action:

   - **Action Sequence** = 4
   - **Action Type** = Branch

   a. Starting with the **Edit Parameters** button, use the event action parameter forms to complete the event action:

<table>
<thead>
<tr>
<th>Field / Button</th>
<th>Action</th>
<th>Result / Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Condition</strong> button</td>
<td>Click.</td>
<td>The <strong>Event Action Parameter Condition</strong> form opens.</td>
</tr>
<tr>
<td><strong>Expression 1</strong> button</td>
<td>Click.</td>
<td>The <strong>Event Action Expression Editor</strong> opens.</td>
</tr>
<tr>
<td><strong>Select a function</strong></td>
<td>Select VOTINGRESULT.</td>
<td>The <strong>Action</strong> drop-down list appears.</td>
</tr>
<tr>
<td><strong>Action</strong> drop-down list</td>
<td>Select 3 Prompt.</td>
<td>The <strong>Action</strong> field displays 3.</td>
</tr>
<tr>
<td><strong>OK</strong> button</td>
<td>Click.</td>
<td>The system returns to the <strong>Event Action Parameter Condition</strong> form.</td>
</tr>
<tr>
<td><strong>Operator</strong> drop-down list</td>
<td>Select &lt;&gt;.</td>
<td>—</td>
</tr>
<tr>
<td><strong>Expression 2 field</strong></td>
<td>Type: 1</td>
<td>—</td>
</tr>
<tr>
<td><strong>OK</strong> button</td>
<td>Click.</td>
<td>The system returns to the <strong>Event Action Branch</strong> form.</td>
</tr>
<tr>
<td><strong>Destination</strong> drop-down list</td>
<td>Type: 14</td>
<td>Even though we have not yet created action Sequence 14, we can enter the number here. This is the action to which we will eventually be jumping if the purchasing manager rejects the request.</td>
</tr>
<tr>
<td><strong>OK</strong> button</td>
<td>Click.</td>
<td>The system returns the action parameters to the <strong>Event Actions</strong> form, correctly formatted.</td>
</tr>
</tbody>
</table>
This action step evaluates the purchasing manager’s response (from action Sequence 3) and directs the handler to the next action depending on that response. If the manager approves the request, the handler continues to the next action. If the manager rejects the request, the system goes to the destination, action Sequence 14, and continues from there.

b. Check the syntax and save the action.

7. Add the next event action:
   - **Action Sequence = 5**
   - **Action Type = Finish**
   a. Starting with the **Edit Parameters** button, use the event action parameter forms to complete the event action:

<table>
<thead>
<tr>
<th>Field / Button</th>
<th>Action</th>
<th>Result / Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition button</td>
<td>Click.</td>
<td>The Event Action Parameter Condition form opens.</td>
</tr>
<tr>
<td>Expression 1 button</td>
<td>Click.</td>
<td>The Event Action Expression Editor opens.</td>
</tr>
<tr>
<td>Select a function drop-down list</td>
<td>Select P.</td>
<td>The Argument 1 button and field display.</td>
</tr>
<tr>
<td>Argument 1 field</td>
<td>Type: POCost.</td>
<td>This property name is derived from the Purchase Orders form.</td>
</tr>
<tr>
<td>OK button</td>
<td>Click.</td>
<td>The system returns the statement to the Event Action Parameter Condition form.</td>
</tr>
<tr>
<td>Operator drop-down list</td>
<td>Select &lt;.</td>
<td>—</td>
</tr>
<tr>
<td>Expression 2 button</td>
<td>Click.</td>
<td>The Event Action Expression Editor opens.</td>
</tr>
<tr>
<td>Select a function drop-down list</td>
<td>Select GC.</td>
<td>The Argument 1 button and field display.</td>
</tr>
<tr>
<td>Argument 1 drop-down list</td>
<td>Select SuperCost.</td>
<td>—</td>
</tr>
<tr>
<td>OK button</td>
<td>Click.</td>
<td>The system returns the statement to the Event Action Parameter Condition form.</td>
</tr>
<tr>
<td>OK button</td>
<td>Click.</td>
<td>The system returns the condition parameter to the Event Action Finish form.</td>
</tr>
<tr>
<td>Result field</td>
<td>Type: Approved by Purchasing Manager.</td>
<td>—</td>
</tr>
<tr>
<td>OK button</td>
<td>Click.</td>
<td>The system returns to the Event Actions form with the parameters correctly formatted.</td>
</tr>
</tbody>
</table>

This action step determines whether the cost of the PO is less than $100,000, the value of the **SuperCost** global constant. If it is, then the handler commits the PO record to the database, writes the result to the event state (viewable on the **Event States** form), and finishes with a status of **Success**. If the PO cost is $100,000 or greater, then the handler continues to the next action.

8. Add the next event action:
   - **Action Sequence = 6**
- **Action Type = Set Values**

  This action step is similar to action Sequence 2, the difference being:

  - The variable named **Approver** is set to the value of the global constant, **PurchasingSuper**, which is the user ID for the purchasing manager’s supervisor.
  - The variable named **CountMethod** is set to count the votes using the **Majority** rule, which says that any choice that gets more than 50% of the vote wins. If we had three supervisors voting, for instance, whichever choice gets the first two votes determines the outcome. Since we have only one individual set to vote at this point, the purchasing supervisor’s vote alone determines what happens next.

9. Add the next event action:

- **Action Sequence = 7**
- **Action Type = Prompt**

  In the **Parameters** text input field, enter the following:

  \[ \text{TGC(POApprovalPrompt)} \]

  As in action Sequence 3, this action step performs a text evaluation of the **POApprovalPrompt** global constant to obtain the parameters for a prompt action. This time, the prompt uses the new variable values for the purchasing supervisor that were set in Step 8.

  After evaluating the **POApprovalPrompt** global constant, this action sends out the prompt message to the purchasing supervisor and suspends the handler again, pending the supervisor’s response. Again, because we did not specify any Variable Access rules, the message allows the purchasing supervisor to modify any variable values before approving it.

10. Add the next event action:

- **Action Sequence = 8**
- **Action Type = Branch**

  This action is similar to action Sequence 4, with the sole difference being that for the **VOTINGRESULT( )** expression, we look at action Sequence 7 instead of Sequence 3.

  This action, then, evaluates the purchasing supervisor’s response (from Action Sequence 7). As soon as any choice has a majority (more than 50% of the votes), the system directs the handler to the next action depending on that response.

  In this case, if the supervisor approves the request, the handler continues to the next action. If the supervisor rejects the request, the system goes to the destination, action Sequence 14, and continues from there.

11. Add the next event action:

- **Action Sequence = 9**
- **Action Type = Finish**

  This action is similar to action Sequence 5, with the following differences:

  - The global constant to use for the condition expression is **SeniorCost**, instead of **SuperCost**.
The Result statement should read: **Approved by both the purchasing manager and the purchasing supervisor.**

This action step determines whether the cost of the PO is less than $1,000,000, the value of the **SeniorCost** global constant. If it is, then the handler commits the PO record to the database, writes the result to the event state (viewable on the Event States form), and finishes with a status of **Success**. If the PO cost is $1,000,000 or greater, then the handler continues to the next action.

12. Add the next event action:

- **Action Sequence = 10**
- **Action Type = Set Values**
  
  ```plaintext
  SETVARVALUES(Approver="PurchasingSenior",
  CountMethod="VOTINGRULE(MinimumPercentage) MINIMUM(100)"
  )
  ```

  This action step is similar to action Sequences 2 and 6, the differences being:

- The variable Approver is set to the value of the global constant, **PurchasingSenior**, which is the user IDs for the senior-level executives who must approve requests over $1,000,000.

- The variable named CountMethod is set to count the votes using the **MinimumPercentage** rule, which says that the first choice to reach a minimum percentage determines the next action. In this case, the minimum percentage is 100%, so all recipients must approve for the PO to reach final approval. If any recipient rejects the request, the entire request is rejected, no matter who has approved it to that point.

  **TIP:** The **MinimumPercentage** voting rule requires that you specify a minimum percentage for passage. This means that, in addition to the **VOTINGRULE( )** keyword, you must also specify the **MINIMUM( )** keyword as part of the variable definition. The resulting declaration for the Value column of the CountMethod variable is: **VOTINGRULE(MinimumPercentage) MINIMUM(100)**

13. Add the next event action:

- **Action Sequence = 11**
- **Action Type = Prompt**

  In the Parameters text input field, enter the following:

  ```plaintext
  TGC(POApprovalPrompt)
  ```

  As in action Sequences 3 and 7, this action step performs a text evaluation of the **POApprovalPrompt** global constant to obtain the parameters for a prompt action. This time, the prompt uses the new variable values for the senior-level executives that were set in the previous step.

  After evaluating the **POApprovalPrompt** global constant, this action sends out the prompt message to the senior executives. Again, because we did not specify any Variable Access rules, the message allows the executives to modify any variable values before approving it.

14. Add the next event action:

- **Action Sequence = 12**
Action Type = Branch

This action is similar to action Sequences 4 and 8, with the difference being that for the VOTINGRESULT( ) expression, we look at action Sequence 11.

This action step evaluates the senior executives’ responses (from action Sequence 11). If both executives vote to approve the request, then the handler moves on to the next action. If either or both of them vote to reject the request, then the handler goes to the destination, action Sequence 14, and continues from there.

15. Add the next event action:
   - Action Sequence = 13
   - Action Type = Finish
     a. In the Event Action Finish form (reached by clicking Edit Parameters), in the Result field, type: Purchase order change approved by senior purchasing executives.
        This action sequence commits the PO record to the database, writes the result to the event state (viewable on the Event States form), and finishes with a status of Success.
     b. Save all actions.

16. Add the next event action:
   - Action Sequence = 14
   - Action Type = Notify
     For any result which ends up in a disapproval of the request change, this action step sends a notification message to the individual who made the original change to the PO status, letting that individual know that the change request has been disapproved at some level.
     a. Use the event action parameter forms to create the following notification message event action:
        TO(ORIGINATOR())
        SUBJECT(SUBSTITUTE("Purchase order {0} change request disapproved", FP("PoNum")))
        CATEGORY("Notification")
        BODY(SUBSTITUTE("Your purchase order change request for PO {0} for {1}, vendor number: {2} has been disapproved. If you have questions, please see the required approvers.",
                          FP("PoNum"),
                          P("VendorName") ,
                          FP("VendNum")))
        SAVEMESSAGE(FALSE)
     b. Check the syntax and save the action.

17. Add the final event action:
   - Action Sequence = 15
- **Action Type = Fail**

  In the **Event Action Fail** form, set the **Result** field to:

  **PO change not approved.**

  This action step writes the result of the rejection to the event state (viewable on the **Event States** form), and exits with a status of **Failure**.

  18. Now that all event actions (including the requisite adjourning actions) have been created and saved, go back to the **Event Handlers** form and select the **Suspend** check box.

  19. Save the handler.

  20. (Optional) Click the **Diagram** button to view the diagrammatic view of the event handler flow in the **Event Handler Diagram** form.

  21. Discard the cached metadata.

     For more information, including the procedure, see **Discarding the Metadata Cache** on page 58.

**Test 1 for This Handler**

In the first test, create a purchase order for less than $100,000 and let the Purchasing Manager disapprove it.

Expected result: The PO status is not changed to **Ordered**.

To perform this test:

1. On the **Purchase Orders** form create a new PO and save it.
2. On the **Purchase Order Lines** form, create a line for an amount of less than $100,000.
3. On the **Purchase Orders** form, change the status for the line you just created to **Ordered** and save the PO.

   Notice that the PO record is disabled because it is now in a suspended state. The status appears to revert to **Planned**, because it has not yet been approved and thus, it has not yet actually been changed in the database.

4. Logged in using the Purchasing Manager's user ID, open the **Inbox** form.
5. Read the new prompt message that the system generated, and on the **Response** tab, select the **No** option.
6. Open or refresh the **Purchase Orders** form, and verify that the new PO line status is still **Planned** and that the PO is again enabled for changes.

**Test 2 for This Handler**

This is the same as Test 1, except this time have the purchasing manager approve the change. Verify that the status is changed to **Ordered** and the PO is again enabled for change.
Test 3 for This Handler

Create a PO with a total cost of between $100,000 and $1,000,000. Have the purchasing supervisor disapprove and verify that the change is rolled back.

Test 4 for This Handler

Do the same as Test 3, except have both the purchasing manager and purchasing supervisor approve the change. Verify that the change is written to the database and that the record is again enabled for other changes.

Additional Tests

For complete thoroughness, other tests could (and should) be devised and conducted before making this event handler live on an active system.

For example, you could change the status to something other than Ordered and make sure that the PO change does not suspend. You should also test for the senior executive approvals and disapprovals.

Points to Note and Remember

- Whenever possible, use the event action parameter forms (accessible by way of the Edit Parameters button). This is your best insurance against syntax errors.
- Not all actions can be created using the event action parameter forms. This is particularly true of the pre-parser functions. In these cases, you cannot check the syntax for these actions using the Check Syntax button.
- You can create whole actions using global constants. You cannot subsequently check these for syntax errors, though, using the Check Syntax button, so proceed with caution.

Extra Challenges

- Add the user ID to the Result statements for both approvals and disapprovals.
- Have the system generate a message to the user who initiated the request notifying that user that the request has been approved.

Modifying Records

Scenario 7: Adding Information to a Record

In this scenario, we want to notify a credit manager that a new customer is being added and ask the credit manager to determine what the credit limit will be for that customer. So, we must send a notification that prompts the credit manager for a response and then uses the data from that response to add data to the pending new customer record and commit the changes to the database.

For this scenario we will:

- Use the same global constant (CreditMgr) we used for previous scenarios.
Use the IdoOnItemInsert framework event.

To accomplish this scenario:

1. Create and save the event handler with the following settings:
   - Event Name = IdoOnItemInsert
   - Applies to Objects = SLCustomers
2. Create one action, which sends a prompt to the credit manager requesting a response:
   - Action Sequence = 10
   - Action Type = Prompt
     a. Use the Event Action Prompt form and associated event action parameter forms to create the following parameters:
        - TO = GC(CreditMgr)
        - SUBJECT = SUBSTITUTE("Need credit limit for customer ID {0}, {1}", FP("CustNum"), P("Name"))
        - CATEGORY = "Financial"
        - BODY = SUBSTITUTE("Please use the Variables tab to provide a credit limit for customer ID {0}, {1}. Then use the Post button on the Response tab to register the new credit limit.", FP("CustNum"), P("Name"))
        - SAVEMESSAGE = FALSE
        - QUESTION = To post the credit limit, click the button below.
        - CHOICES = 1,sPost
        - FILTERFORM = Customers
        - FILTER = SUBSTITUTE("CustNum={0}", FP("CustNum"))
     b. On the Variable Access tab:
        - Name = CreditLimit
        - Access = Mandatory
          This forces a response from the credit manager.
     c. Save the action and close the Event Actions form.
3. Return to the Event Handlers form and select the Suspend check box.
4. Save the handler.
5. Discard the cached metadata.

   For more information, including the procedure, see Discarding the Metadata Cache on page 58.

Testing the Event Handler

1. Open the Customers form and create a new customer record. Save it.

   Notice that the newly saved record does not appear in the list of customers at this point when you refresh the Customers form.

2. Using the credit manager’s logon, open the credit manager’s Inbox.

   The new message should appear in the Inbox.
3. On the **Variables** tab, locate the **Row.CreditLimit** variable and enter an amount.
4. Save the record.
5. On the **Response** tab, click **Post**.
6. Back on the **Customers** form, refresh the form and verify that the newly created record appears in the list.

You can (and should) also devise other tests to verify that the system behaves as expected when the credit manager posts the response without entering a value in the **Row.CreditLimit** variable field.

### Points to Note and Remember

When creating a handler like this one, keep the following points in mind:

- **Because the variables are listed on the Variables tab and the question and response buttons are on the Response tab of the message, you should design your message body to include brief but detailed instructions for responding to the request. Do not assume the recipient will know or remember.**

- **Because we did not specify variable access rules to address property variables other than the CreditLimit variable, all the variable property values associated with the Customers form are displayed and writable. That means that the credit manager, if desired, can change any variable data before saving the data and posting it to the database by clicking **Post**. (To make other variables non-writable, you must set the variable access for each individually.)**

- **The fact that the prompt message was sent to a single recipient (in this case) means that only one vote is required for a quorum, and once the credit manager posts the response, the vote is final and the database is updated. If there are multiple recipients associated with the CreditMgr global constant, then you might also need to set voting rules to determine how the responses will be handled. In this case, it is not necessary, because the system assumes a Plurality voting rule, and with only one recipient, that means that the first to respond is the one whose data is committed.**

- **If the credit manager never votes, the record is never committed to the database, but it remains adjourned indefinitely.**

- **The QUESTION parameter has a limit of 80 characters.**
Voting

Scenario 8: Voting for Various Choices

In this scenario, we need several managers at the same level to approve an engineering change, by means of a response to a message. So, we must send a notification that prompts the managers for a response. If at least two of the managers send responses approving the change, we then approve the requested change in the application.

For this scenario we will:

- Assume that global constants were created for EngineeringMgr, ProjectMgr, and ProgramMgr. The creation of global constants is described in previous scenarios.
- Use the IdoOnItemUpdate framework event.

To accomplish this scenario:

1. Create and save the event handler with the following settings:
   - Event Name = IdoOnItemUpdate
   - Applies to Objects = SLECNs
   - Description = ECN Approval

2. Create one action, which sends a prompt to the managers requesting a response:
   - Action Sequence = 10
   - Action Type = Prompt
     a. Use the Event Action Prompt form and associated event action parameter forms to create the following parameters:
        - TO = GC(EngineeringMgr) + ';' + GC(ProjectMgr) + ';' + GC(ProgramMgr)
        - SUBJECT = SUBSTITUTE("Need approval for engineering change {0}, {1}", P("EcnNum"), P("ReasonCodeDescription"))
        - CATEGORY = "Engineering"
        - BODY = "Please review the proposed engineering change on the Variables tab. Then use the Approve or Reject buttons on the Response tab to register your response."
        - SAVEMESSAGE = FALSE
        - QUESTION = To approve or reject, click the buttons below.
        - CHOICES = 1,sApprove, 0,sReject
        - VOTINGRULE = Minimum Count Preferred Choice
        - PREFERREDCHOICE = 1
        - MINIMUM = 2
        - FILTERFORM = EngineeringChangeNotices
        - FILTER = SUBSTITUTE("ECNNum={0}" , FP("ECNNum"))
b. Save the action. If you have done everything correctly, your syntax for this action step should look like the following:

```plaintext
TO(' + GC(EngineeringMgr) + ';' + GC(ProjectMgr) + ';' + GC(ProgramMgr))
CATEGORY("Engineering")
SUBJECT(SUBSTITUTE("Need approval for engineering change {0}, {1}",
P("EcnNum"), P("ReasonCodeDescription")))
BODY("Please review the proposed engineering change on the Variables tab. Then use the Approve or Reject buttons on the Response tab to register your response.")
SAVEMESSAGE(FALSE)
QUESTION("To approve or reject, click the buttons below.")
CHOICES("1,sApprove,0,sReject")
VOTINGRULE(MinimumCountPreferredChoice)
MINIMUM(2)
PREFCHOICE("1")
FILTERFORM("EngineeringChangeNotices")
FILTER(SUBSTITUTE("ECNNum={0}", FP("ECNNum")))
```

3. Create the second action, which tells the system how to respond if approval is not granted:

   a. In the Action Sequence field, enter 20.
   b. In the Action Type field, select Fail.
      This action type ends handler execution with an error status. This effectively aborts the process and prevents the ECN from being changed in the database.
   c. Starting with the Event Action Fail form, use the associated forms to create the following parameters:
      - CONDITION(VOTINGRESULT(10) = "0")
      - RESULT("The ECN change request was rejected by the managers.")
   d. Save the action and close the Event Actions form.

4. Return to the Event Handlers form and select the Suspend check box.
5. Save the handler.
6. Discard the cached metadata.

   For more information, including the procedure, see Discarding the Metadata Cache on page 58.

Testing the Event Handler

To test this handler:

1. Open the Engineering Change Notices form and update an existing ECN. Save it.
   After you save the ECN, when the Engineering Change Notices refreshes the display, the record should be disabled for updating. It remains read-only until/unless it has been approved.
2. (Optional) With the Engineering Change Notices form selected, from the Actions menu, select View Event Status. This opens the Event Status form. Navigate to the last row and verify that the status for this event is Running.
3. Open the **Inbox** form for the individual designated as the Engineering Manager and verify that the message was received and that the Response tab displays the question and choice buttons. Click the button labeled **Yes**.

4. Open the **Inbox** form for the individual designated as the Project Manager and verify that the message was received and that the Response tab displays the question and choice buttons. Click the button labeled **Yes**.

5. Refresh the collection on the **Engineering Change Notices** form and verify that the ECN now displays normally (read/write) and shows your changes. As soon as two managers vote for the preferred choice, voting is closed and the change is approved. The third manager’s vote is not needed.

You can also do a second test, clicking the button labeled **No** to reject the request by all three managers. In this case, when you refresh the **Engineering Change Notices** form, the ECN record displays normally but your changes are gone.

**Points to Note and Remember**

When creating this kind of event handler, keep the following in mind:

- When creating a message that requires a response from the recipient (usually a **Prompt** action type), you must mark the handler so that it suspends when executed. This means that it is also automatically marked as a synchronous handler.

- Because these event handlers must be suspended, pending the managers’ responses, the Framework Event Service must be enabled for the configuration in which you are logged on.
Localizing Message Contents

Scenario 9: Translating Captions in a Purchase Request

This scenario uses strings in captions so the text can be read by users in different countries.

Metadata Setup

These strings must exist in the Forms database Strings tables:

<table>
<thead>
<tr>
<th>Strings.Name</th>
<th>Strings.String</th>
<th>SpainString.String</th>
</tr>
</thead>
<tbody>
<tr>
<td>sItem</td>
<td>Item</td>
<td>Prod</td>
</tr>
<tr>
<td>sWhse</td>
<td>Whse</td>
<td>Alm</td>
</tr>
<tr>
<td>sPoitemApprovalQuestion</td>
<td>Do you approve of purchasing %1 %2 of [%3: %4] for delivery to [%5: %6]</td>
<td>Usted aprueba de comprar %1 %2 del %3 &quot;%4&quot; para la entrega al %5 &quot;%6&quot;</td>
</tr>
<tr>
<td>sPoitemApproval</td>
<td>Purchase Approval</td>
<td>Aprobación de Comprar el Artículo</td>
</tr>
<tr>
<td>sOrderApproval</td>
<td>Order Approval</td>
<td>Aprobación del Documento</td>
</tr>
</tbody>
</table>

Event Message Category

In the Event Message Categories form, set up the Category `FORMAT(sOrder Approval)`, with the description Approval of an Order.

Event Action Parameters

Create an event action that includes these parameters:

```
... QUESTION (
  CLIENTSUBSTITUTE(
    "sPoitemApprovalQuestion",
    P(QtyOrdered),
    P(UMDesc),
    "STRINGS(sItem)",
    P(Item), "STRINGS(sWhse)",
    P(Whse)
  )
) )
SUBJECT( 
  CLIENTSUBSTITUTE( 
    P(PoitemApproval) 
  ) 
) ) 
CATEGORY("FORMAT(sOrder Approval)")
...
```
After the event action execution, the full components of the message in the database are as follows:

**EventMessage.Question**

FORMAT(sPoltemApprovalQuestion, ~LIT~(100.0), ~LIT~(Metric Tons), STRINGS(sItem), ~LIT~("5" screw/x03 chrome/x04 hex head/x05), STRINGS(sWhse), ~LIT~(MAIN))

**EventMessage.Subject**

FORMAT(sPoltemApproval)

**EventMessage.Category**

FORMAT(sOrderApproval)

Results (English)

An English-speaking user who refreshes the Inbox sees this message:

<table>
<thead>
<tr>
<th>Component</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>Question</td>
<td>Do you approve of purchasing 100.0 Metric Tons of [Item: 5&quot; screw, chrome (hex head)] for delivery to [Whse: MAIN]</td>
</tr>
<tr>
<td>Subject</td>
<td>Purchase Approval</td>
</tr>
<tr>
<td>Category</td>
<td>Order Approval</td>
</tr>
</tbody>
</table>

Results (Spanish)

A Spanish-speaking user who refreshes the Inbox sees this message:

<table>
<thead>
<tr>
<th>Component</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>Question</td>
<td>Usted aprueba de comprar 100.0 Metric Tons del Prod &quot;5&quot; screw, chrome (hex head) para la entrega al Alm &quot;MAIN&quot;</td>
</tr>
<tr>
<td>Subject</td>
<td>Aprobación de Comprar el Artículo</td>
</tr>
<tr>
<td>Category</td>
<td>Aprobación del Documento</td>
</tr>
</tbody>
</table>
More Advanced Scenarios

Scenario 10: Opening a Session in a Remote Environment

In this scenario, we want to call in to another site or an external ERP SyteLine environment and return a specific discrete piece of information from the remote App database. In this scenario, we want to return the **On Hand** quantity of an item in another site.

For the details of and procedure for this scenario, see the *Integrating IDOs with External Applications* guide.

Scenario 11: Cross-Site Event Firing - Adding a Message to Another Site's Inbox

This scenario illustrates the general requirements used to set up cross-site event firing. Our example adds a message to another site's Inbox form.

- An event handler called **GenericNotify** is available in every target site. You can fire this event to perform the "remote" work.

  This event has one handler with one action that performs a Notify, using event parameters for To, Subject, Category, and Body.

- A stored procedure called **dbo.FireGenericNotifySp** is available in every target site. You can use this stored procedure to fire the event. This stored procedure accepts the T-SQL parameters @To, @Subject, @Category and @Body, and it fires the GenericNotify event, passing in the information that the event needs.

  **NOTE:** You could do the same thing using a hand-coded IDO Method that calls Mongoose.EventSystem.EventHandlers.FireApplicationEvent(), in case the event needs to perform IDO-level actions. However, for this scenario, using the stored procedure is easier.

- Where you want to add a message to another site's inbox, your event handler includes a **Dispatch IDO Request** action, providing these parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL()</td>
<td>URL of an IDO Request Service that serves the target site</td>
</tr>
<tr>
<td></td>
<td><strong>Sample value:</strong> http://[server]/IDORequestService/RequestService.aspx</td>
</tr>
</tbody>
</table>
If you have multiple intranets that do not serve all sites, and you want the event to fire across intranets, you may need to set up an event global constant for each intranet (or site) URL, and select one of those at runtime.

The sample action shown here assumes that the originating user has the same password on the source and target sites, which allows it to log in for the Invoke. If this may not be the case, you can instead set up a generic remote user, with permissions only to perform this one Invoke, and specify the USERNAME() and PASSWORD() parameters on the Dispatch IDO Request action. Alternatively, you can adjust the USELOCALPASSWORD() parameter.

If you want to perform a different type of remote work, you can set up your own events with handlers and actions to address the work you want to do. Then you can create stored procedures to fire those events, and finally call those stored procedures using Dispatch IDO Request actions in the appropriate existing handlers.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONFIGNAME()</td>
<td>Name of a configuration whose App database contains the target site. (SyteLine requires a configuration that is named after each site, so you can use the site name here.) Sample value: E(SiteId)</td>
</tr>
<tr>
<td>IDOREQUEST()</td>
<td>Invoke “SPI” (or the name of an IDO) plus the name of the stored procedure (in this case FireGenericNotifySp), passing in the event parameters as method parameters to the “method.” Sample value (shown as XML to allow pasting into the form): SUBSTITUTE('&lt;RequestHeader Type=&quot;Invoke&quot;&gt; &lt;InitiatorType /&gt; &lt;InitiatorName /&gt; &lt;SourceName /&gt; &lt;TargetName /&gt; &lt;RequestData&gt; &lt;Name&gt;SP!&lt;/Name&gt; &lt;Method&gt;FireGenericNotifySp&lt;/Method&gt; &lt;Parameters&gt; &lt;Parameter&gt;{0}&lt;/Parameter&gt; &lt;Parameter&gt;{1}&lt;/Parameter&gt; &lt;Parameter&gt;{2}&lt;/Parameter&gt; &lt;Parameter&gt;{3}&lt;/Parameter&gt; ByRef=&quot;Y&quot; /&gt; &lt;/Parameters&gt; &lt;/RequestData&gt; &lt;/RequestHeader&gt;', E(ToParm), E(SubjectParm), E(CategoryParm), E(BodyParm))</td>
</tr>
</tbody>
</table>
This section of the Help provides the following reference tables containing detailed information about various aspects of the application event system. Topics include:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firing Events</td>
<td>126</td>
</tr>
<tr>
<td>Summary of Synchronous Functionality</td>
<td>127</td>
</tr>
<tr>
<td>Framework Events</td>
<td>128</td>
</tr>
<tr>
<td>Framework Event Attributes</td>
<td>129</td>
</tr>
<tr>
<td>Application Events</td>
<td>134</td>
</tr>
<tr>
<td>Event Action Types</td>
<td>135</td>
</tr>
<tr>
<td>Event Action Parameters</td>
<td>140</td>
</tr>
<tr>
<td>Expression Functions</td>
<td>155</td>
</tr>
<tr>
<td>Expression Operators</td>
<td>171</td>
</tr>
</tbody>
</table>
Firing Events

The following table provides details about:
- Where events can be generated from (Tier)
- What can be used to generate them from that location (Triggered by)
- How to set them up (Details for construction)
- Whether the event is generated as a synchronous or asynchronous event (Synchronous?)

<table>
<thead>
<tr>
<th>Tier</th>
<th>Triggered by</th>
<th>Details for construction</th>
<th>Synchronous?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>Event handler in form</td>
<td>Use a response type of <strong>Generate Application Event</strong>, select the <strong>Synchronous</strong> option.</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use a response type of <strong>Generate Application Event</strong>, clear the <strong>Synchronous</strong> option.</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Script in form</td>
<td>Generate a custom form event. Create a form event handler for that custom event with a response type of <strong>Generate Application Event</strong>, with the <strong>SYNCHRONOUS( )</strong> parameter.</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Generate a custom form event. Create a form event handler for that custom event with a response type of <strong>Generate Application Event</strong>, without the <strong>SYNCHRONOUS( )</strong> parameter.</td>
<td>No</td>
</tr>
<tr>
<td>Middle</td>
<td>Custom IDO method</td>
<td>Invoke the <strong>FireApplicationEvent( )</strong> static .NET method with the <strong>Synchronous</strong> parameter passed in with a value of <strong>True</strong>.</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Invoke the <strong>FireApplicationEvent( )</strong> static .NET method with the <strong>Synchronous</strong> parameter passed in with a value of <strong>False</strong>.</td>
<td>No</td>
</tr>
<tr>
<td>Database</td>
<td>T-SQL</td>
<td>Use the command: <strong>EXEC FireEventSp</strong></td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use the command: <strong>EXEC PostEventSp</strong></td>
<td>No</td>
</tr>
<tr>
<td>Any</td>
<td>An event action</td>
<td>Use the <strong>GenerateEvent</strong> action type with a parameter of <strong>SYNCHRONOUS(true)</strong>.</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use the <strong>GenerateEvent</strong> action type with a parameter of <strong>SYNCHRONOUS(false)</strong>.</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>An event trigger</td>
<td>Select the <strong>Synchronous</strong> option.</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Clear the <strong>Synchronous</strong> option.</td>
<td>No</td>
</tr>
</tbody>
</table>
## Summary of Synchronous Functionality

<table>
<thead>
<tr>
<th>Source</th>
<th>Synchronous?</th>
<th>Consists of</th>
<th>Requester</th>
<th>Initial executor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework</td>
<td>Yes</td>
<td>Synchronous (none marked <strong>Suspend</strong>) &amp; asynchronous event handlers</td>
<td>WinStudio</td>
<td>IDO Runtime</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IDO Runtime</td>
<td>IDO Runtime</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Database</td>
<td>Database</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Synchronous (some marked <strong>Suspend</strong>) &amp; asynchronous event handlers</td>
<td>WinStudio</td>
<td>IDO Runtime (validating mode)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IDO Runtime</td>
<td>Event Service (Committing mode)</td>
</tr>
<tr>
<td>FireEvent OR Generate Event</td>
<td>Yes</td>
<td>Synchronous and asynchronous event handlers</td>
<td>WinStudio</td>
<td>IDO Runtime</td>
</tr>
<tr>
<td>with <strong>Synchronous</strong> (True)</td>
<td></td>
<td></td>
<td>IDO Runtime</td>
<td>IDO Runtime</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Database</td>
<td>Database</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Event service</td>
<td>Event service</td>
</tr>
<tr>
<td>PostEvent OR Generate Event</td>
<td>No</td>
<td>Synchronous and asynchronous event handlers</td>
<td>WinStudio</td>
<td>Event service</td>
</tr>
<tr>
<td>with <strong>Synchronous</strong> (False)</td>
<td></td>
<td></td>
<td>IDO Runtime</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Database</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Event service</td>
<td></td>
</tr>
</tbody>
</table>

### Context Table

<table>
<thead>
<tr>
<th>Context</th>
<th>Synchronous?</th>
<th>Suspending?</th>
<th>Event’s or prior event handler’s executor</th>
<th>Initial executor</th>
<th>Can adjourn?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suspending event (always generated synchronously)</td>
<td>Yes</td>
<td>No</td>
<td>IDO Runtime (validating mode)</td>
<td>IDO Runtime</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Event service (committing mode)</td>
<td>Event service</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>Event service</td>
<td>Event service</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-suspending event generated synchronously</td>
<td>Yes</td>
<td>N/A</td>
<td>WinStudio</td>
<td>IDO Runtime</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IDO Runtime</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Database</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Event service</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-suspending event generated asynchronously</td>
<td>Yes</td>
<td>N/A</td>
<td>WinStudio</td>
<td>Event service</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IDO Runtime</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Database</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Event service</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Any event</td>
<td>No</td>
<td>N/A</td>
<td>WinStudio</td>
<td>Event service</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IDO Runtime</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Database</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Event service</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Framework Events

The following table lists and describes the framework (Core) events that ship with the system. You can create handlers that execute when these events are generated, but you cannot create your own triggers for these events.

<table>
<thead>
<tr>
<th>Event name</th>
<th>Trigger</th>
<th>Context attributes passed as event parameters, in addition to the object name</th>
</tr>
</thead>
</table>
| IdoOnLoadCollection   | When an IDO collection is loaded into the system.                       | ■ Load flags  
■ Property names  
■ Post query actions |
| IdoPostLoadCollection | After an IDO collection is loaded into the system                        | Result set                                                                                                     |
| IdoOnUpdateCollection | Before an IDO collection update is performed                             | ■ Custom insert specification  
■ Custom update specification  
■ Custom delete specification |
| IdoOnItemInsert       | When an IDO item is inserted into a collection                          | Row (IDO item) being inserted                                                                                   |
| IdoPostItemInsert     | After an IDO item is inserted into a collection                         | Row (IDO item) that was inserted                                                                               |
| IdoOnItemUpdate       | When an IDO item is updated                                             | ■ Row (IDO item) being updated  
■ Modified flags                                                             |
| IdoPostItemUpdate     | After an IDO item is updated                                             | ■ Row (IDO item) that was updated  
■ Modified flags                                                             |
| IdoOnItemDelete       | When an IDO item is deleted                                             | Row (IDO item) being deleted                                                                                   |
| IdoPostItemDelete     | After an IDO item is deleted                                             | Row (IDO item) that was deleted                                                                                |
| IdoPostUpdateCollection| After all IDO collection updates have been processed                    | —                                                                                                              |
| IdoOnInvoke           | When an IDO method is invoked                                           | ■ Name of the IDO method that was invoked  
■ Number of parameters passed to the method  
■ Values of the parameters passed to the method                               |
| IdoPostInvoke         | After an IDO method is invoked                                          | ■ Name of the IDO method that was invoked  
■ Number of parameters passed to the method  
■ Values of the parameters passed to the method                               |
| SessionOnLogin        | When a new session is requested                                          | User name                                                                                                     |
| SessionOnLogout       | When a session is closed                                                | User name                                                                                                     |
| SessionOnVarChanged   | When a session variable’s value is changed                               | The name and new value of the session variable                                                                |
## Framework Event Attributes

The following table lists and describes the expression functions associated with each framework (Core) event.

For more information about function syntax, see **Expression Functions** on page 155.

**NOTE:** Wherever property names are mentioned in the following table, `<Property#>`, as represented here, specifies the name as a string expression. To specify the property name as a constant value, enclose it in double quotes ("...")

<table>
<thead>
<tr>
<th>Event name</th>
<th>Applicable functions</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IdoOnLoadCollection</strong></td>
<td>INITIATOR( )</td>
<td>Composite initiator (period-separated):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Type (form or other client)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Name (form name or other client ID)</td>
</tr>
<tr>
<td></td>
<td>IDO( )</td>
<td>Name of the IDO collection</td>
</tr>
<tr>
<td></td>
<td>PROPERTYNAMES( )</td>
<td>List of property names to be retrieved</td>
</tr>
<tr>
<td></td>
<td>POSTQUERYACTIONS( )</td>
<td>List of post-query actions</td>
</tr>
<tr>
<td></td>
<td>FILTERSTRING( )</td>
<td>Filter string</td>
</tr>
<tr>
<td></td>
<td>RECORDCAP( )</td>
<td>Record cap</td>
</tr>
<tr>
<td><strong>IdoPostLoadCollection</strong></td>
<td>INITIATOR( )</td>
<td>Composite initiator (period-separated):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Type (form or other client)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Name (form name or other client ID)</td>
</tr>
<tr>
<td></td>
<td>IDO( )</td>
<td>Name of the IDO collection</td>
</tr>
<tr>
<td></td>
<td>PROPERTYNAMES( )</td>
<td>List of property names that were retrieved</td>
</tr>
<tr>
<td></td>
<td>PROPERTY(1, <code>&lt;Property1&gt;</code>)</td>
<td>Value of the first property from the first row retrieved</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>PROPERTY(N, <code>&lt;PropertyM&gt;</code>)</td>
<td>... Value of the Mth property from the Nth row retrieved</td>
</tr>
<tr>
<td></td>
<td>P(N, <code>&lt;Property&gt;</code>)</td>
<td>Value of the given property of the Nth row</td>
</tr>
<tr>
<td></td>
<td>FILTERPROPERTY(N, <code>&lt;Property&gt;</code>)</td>
<td>Value of the given property of the Nth row as a string</td>
</tr>
<tr>
<td></td>
<td>FP(N, <code>&lt;Property&gt;</code>)</td>
<td>Value of the given property of the Nth row as a string</td>
</tr>
<tr>
<td></td>
<td>FILTERSTRING( )</td>
<td>Filter string</td>
</tr>
<tr>
<td></td>
<td>RECORDCAP( )</td>
<td>Record cap</td>
</tr>
<tr>
<td></td>
<td>ROWS( )</td>
<td>Number of rows that were loaded</td>
</tr>
<tr>
<td><strong>IdoOnUpdateCollection</strong></td>
<td>INITIATOR( )</td>
<td>Composite initiator (period-separated):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Type (form or other client)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Name (form name or other client ID)</td>
</tr>
<tr>
<td></td>
<td>IDO( )</td>
<td>Name of the IDO collection</td>
</tr>
<tr>
<td></td>
<td>CUSTOMINSERT( )</td>
<td>Specification for a custom insert</td>
</tr>
<tr>
<td></td>
<td>CUSTOMUPDATE( )</td>
<td>Specification for a custom update</td>
</tr>
<tr>
<td></td>
<td>CUSTOMDELETE( )</td>
<td>Specification for a custom delete</td>
</tr>
<tr>
<td>Event name</td>
<td>Applicable functions</td>
<td>Contents</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------------</td>
<td>----------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>IdoOnItemInsert</strong></td>
<td>INITIATOR( )</td>
<td>Composite initiator (period-separated):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Type (form or other client)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Name (form name or other client ID)</td>
</tr>
<tr>
<td></td>
<td>IDO( )</td>
<td>Name of the IDO collection</td>
</tr>
<tr>
<td></td>
<td>PROPERTYNAMES( )</td>
<td>List of property names to be saved</td>
</tr>
<tr>
<td></td>
<td>PROPERTY(&lt;Property1&gt;)</td>
<td>Value of the first property of the row to be saved into the database</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PROPERTY(&lt;PropertyM&gt;)</td>
<td>Value of the M&lt;sup&gt;th&lt;/sup&gt; property of the row to be saved into the database</td>
</tr>
<tr>
<td></td>
<td>P(&lt;Property&gt;)</td>
<td>Value of the given property</td>
</tr>
<tr>
<td></td>
<td>FILTERPROPERTY(&lt;Property&gt;)</td>
<td>Value of the given property as a string</td>
</tr>
<tr>
<td></td>
<td>FP(&lt;Property&gt;)</td>
<td>Value of the given property as a string</td>
</tr>
<tr>
<td></td>
<td>PROPERTYMODIFIED(&lt;Property1&gt;)</td>
<td>1 if and only if the first property was updated from its original value; otherwise 0</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PROPERTYMODIFIED(&lt;PropertyM&gt;)</td>
<td>1 if and only if the M&lt;sup&gt;th&lt;/sup&gt; property was updated from its original value; otherwise 0</td>
</tr>
<tr>
<td><strong>IdoPostItemInsert</strong></td>
<td>INITIATOR( )</td>
<td>Composite initiator (period-separated):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Type (form or other client)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ Name (form name or other client ID)</td>
</tr>
<tr>
<td></td>
<td>IDO( )</td>
<td>Name of the IDO collection</td>
</tr>
<tr>
<td></td>
<td>PROPERTY(&lt;Property1&gt;)</td>
<td>Value of the first property of the row that was saved into the database</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PROPERTY(&lt;PropertyM&gt;)</td>
<td>Value of the M&lt;sup&gt;th&lt;/sup&gt; property of the row that was saved into the database</td>
</tr>
<tr>
<td></td>
<td>P(&lt;Property&gt;)</td>
<td>Value of the given property</td>
</tr>
<tr>
<td></td>
<td>FILTERPROPERTY(&lt;Property&gt;)</td>
<td>Value of the given property as a string</td>
</tr>
<tr>
<td></td>
<td>FP(&lt;Property&gt;)</td>
<td>Value of the given property as a string</td>
</tr>
<tr>
<td>Event name</td>
<td>Applicable functions</td>
<td>Contents</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| IdoOnItemUpdate        | INITIATOR( )                                                                                                                                                                                                       | Composite initiator (period-separated):  
- Type (form or other client)  
- Name (form name or other client ID)                                                                                                                                                                           |
|                        | IDO( )                                                                                                                                                                                                              | Name of the IDO collection                                                                                                                                                                                  |
|                        | PROPERTYNAMES( )                                                                                                                                                                                                   | List of property names to be saved                                                                                                                                                                          |
|                        | PROPERTY(<Property1>)                                                                                                                                                                                               | Value of the first property of the row to be saved into the database                                                                                                                                         |
|                        | ...                                                                                                                                                                                                                 | ...                                                                                                                                                                                                     |
|                        | PROPERTY(<PropertyM>)                                                                                                                                                                                               | Value of the M\textsuperscript{th} property of the row to be saved into the database                                                                                                                                 |
|                        | P(<Property>)                                                                                                                                                                                                      | Value of the given property                                                                                                                                                                                  |
|                        | FILTERPROPERTY(<Property>)                                                                                                                                                                                              | Value of the given property as a string                                                                                                                                                                      |
|                        | FP(<Property>)                                                                                                                                                                                                     | Value of the given property as a string                                                                                                                                                                      |
|                        | PROPERTYMODIFIED(<Property1>)                                                                                                                                                                                          | 1 if and only if the first property was updated from its original value; otherwise 0                                                                                                                                 |
|                        | ...                                                                                                                                                                                                                 | ...                                                                                                                                                                                                     |
|                        | PROPERTYMODIFIED(<PropertyM>)                                                                                                                                                                                          | 1 if and only if the M\textsuperscript{th} property was updated from its original value; otherwise 0                                                                                                                                 |
| IdoPostItemUpdate      | INITIATOR( )                                                                                                                                                                                                       | Composite initiator (period-separated):  
- Type (form or other client)  
- Name (form name or other client ID)                                                                                                                                                                           |
<p>|                        | IDO( )                                                                                                                                                                                                              | Name of the IDO collection                                                                                                                                                                                  |
|                        | PROPERTY(&lt;Property1&gt;)                                                                                                                                                                                               | Value of the first property of the row that was saved into the database                                                                                                                                     |
|                        | ...                                                                                                                                                                                                                 | ...                                                                                                                                                                                                     |
|                        | PROPERTY(&lt;PropertyM&gt;)                                                                                                                                                                                               | Value of the M\textsuperscript{th} property of the row that was saved into the database                                                                                                                                 |
|                        | P(&lt;Property&gt;)                                                                                                                                                                                                      | Value of the given property                                                                                                                                                                                  |
|                        | FILTERPROPERTY(&lt;Property&gt;)                                                                                                                                                                                              | Value of the given property as a string                                                                                                                                                                      |
|                        | FP(&lt;Property&gt;)                                                                                                                                                                                                     | Value of the given property as a string                                                                                                                                                                      |</p>
<table>
<thead>
<tr>
<th>Event name</th>
<th>Applicable functions</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>IdoOnItemDelete</td>
<td>INITIATOR( )</td>
<td>Composite initiator (period-separated): type (form or other client) and name (form name or other client ID).</td>
</tr>
<tr>
<td></td>
<td>IDO( )</td>
<td>Name of the IDO collection</td>
</tr>
<tr>
<td></td>
<td>PROPERTY(&lt;Property1&gt;)</td>
<td>Value of the first property of the row to be deleted from the database</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PROPERTY(&lt;PropertyM&gt;)</td>
<td>Value of the Mth property of the row to be deleted from the database</td>
</tr>
<tr>
<td></td>
<td>P(&lt;Property&gt;)</td>
<td>Value of the given property</td>
</tr>
<tr>
<td></td>
<td>FILTERPROPERTY(&lt;Property&gt;)</td>
<td>Value of the given property as a string</td>
</tr>
<tr>
<td></td>
<td>FP(&lt;Property&gt;)</td>
<td>Value of the given property as a string</td>
</tr>
<tr>
<td></td>
<td>PROPERTYMODIFIED()</td>
<td>Always returns 0</td>
</tr>
<tr>
<td>IdoPostItemDelete</td>
<td>INITIATOR( )</td>
<td>Composite initiator (period-separated): type (form or other client) and name (form name or other client ID).</td>
</tr>
<tr>
<td></td>
<td>IDO( )</td>
<td>Name of the IDO collection</td>
</tr>
<tr>
<td>IdoPostUpdate-Collection</td>
<td>INITIATOR( )</td>
<td>Composite initiator (period-separated): type (form or other client) and name (form name or other client ID).</td>
</tr>
<tr>
<td></td>
<td>IDO( )</td>
<td>Name of the IDO collection</td>
</tr>
<tr>
<td>IdoOnInvoke</td>
<td>INITIATOR( )</td>
<td>Composite initiator (period-separated): type (form or other client) and name (form name or other client ID).</td>
</tr>
<tr>
<td></td>
<td>IDO( )</td>
<td>Name of the IDO collection</td>
</tr>
<tr>
<td></td>
<td>METHOD( )</td>
<td>Name of the method to be invoked</td>
</tr>
<tr>
<td></td>
<td>METHODPARMS( )</td>
<td>Number of parameters in the following lists</td>
</tr>
<tr>
<td></td>
<td>METHODPARM(&lt;n&gt;)</td>
<td>The value of the nth parameter passed to the method, where n is an integer</td>
</tr>
<tr>
<td></td>
<td>FILTERMETHODPARM(&lt;n&gt;)</td>
<td>The value of the nth parameter passed to the method as a string</td>
</tr>
<tr>
<td>IdoPostInvoke</td>
<td>INITIATOR( )</td>
<td>Composite initiator (period-separated): type (form or other client) and name (form name or other client ID).</td>
</tr>
<tr>
<td></td>
<td>IDO( )</td>
<td>Name of the IDO collection</td>
</tr>
<tr>
<td></td>
<td>METHOD( )</td>
<td>Name of the method that was invoked</td>
</tr>
<tr>
<td></td>
<td>METHODPARMS( )</td>
<td>Number of parameters in the following lists</td>
</tr>
<tr>
<td></td>
<td>METHODPARM(&lt;n&gt;)</td>
<td>The value of the parameter referenced by n, where n is an integer</td>
</tr>
<tr>
<td></td>
<td>FILTERMETHODPARM(&lt;n&gt;)</td>
<td>The value of the nth parameter passed to the method</td>
</tr>
<tr>
<td>Event name</td>
<td>Applicable functions</td>
<td>Contents</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SessionOnLogin</td>
<td>INITIATOR( )</td>
<td>Composite initiator (period-separated):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Type (form or other client)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Name (form name or other client ID)</td>
</tr>
<tr>
<td></td>
<td>USERNAME( )</td>
<td>User requesting the session logon</td>
</tr>
<tr>
<td>SessionOnLogout</td>
<td>INITIATOR( )</td>
<td>Composite initiator (period-separated):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Type (form or other client)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Name (form name or other client ID)</td>
</tr>
<tr>
<td></td>
<td>USERNAME( )</td>
<td>User logging off the session</td>
</tr>
<tr>
<td>SessionOnVarChanged</td>
<td>INITIATOR( )</td>
<td>Composite initiator (period-separated):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Type (form or other client)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Name (form name or other client ID)</td>
</tr>
<tr>
<td></td>
<td>VARIABLENAME( )</td>
<td>Name of the session variable</td>
</tr>
<tr>
<td></td>
<td>VARIABLEVALUE( )</td>
<td>Value of the session variable</td>
</tr>
</tbody>
</table>
## Application Events

The following table lists and describes the application events that ship with the system. Event handlers already exist for these events.

<table>
<thead>
<tr>
<th>Event name</th>
<th>Trigger</th>
<th>Context attributes to be passed as event parameters, in addition to the object name</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenericNotify</td>
<td>When the Task List (or anyone else) fires this event for a reminder</td>
<td>ToVar, SubjectVar, CategoryParm, BodyVar (To, Subject, Category, and Body of the notify action)</td>
</tr>
<tr>
<td>TaskListCheck</td>
<td>When the TaskListCheck Event Trigger fires from the Event Service</td>
<td>None</td>
</tr>
</tbody>
</table>
## Event Action Types

The following table lists and describes the action types available for use when defining event actions. For details about the associated event action parameters, including their syntax and examples, see [Event Action Parameters](#) on page 140.

**NOTE:** Adjourning event action types (that is, **Prompt**, **Wait**, and **Sleep**) cannot be assigned to event actions on a transactional event handler. For more information about adjournment, see [Adjournment and Resumption](#) on page 27. For more information about transactional event handlers, see [Transactions](#) on page 31.

Mid-tier event action types (that is, **Call IDO Method**, **Dispatch IDO Request**, **Execute IDO Request**, **Load IDO Collection**, **Load IDO Row**, and **Update Collection**) cannot be executed on a synchronous event handler whose event is fired from the database layer (that is, via FireEventSp, or attached to any Session framework event). If this is attempted, the handler fails with an error.

<table>
<thead>
<tr>
<th>Action type</th>
<th>What it does</th>
<th>Associated event action parameters</th>
<th>Adjourning?</th>
</tr>
</thead>
</table>
| Achieve Milestone   | Changes the visual state of a running event when a specified condition is met. This action type can be particularly useful when you want the system to allow for fast, visual indications of what action step a handler has reached. | ■ Condition  
■ State (text)  
■ Image (file)  
■ Result                                              | No          |
| Audit               | Adds an entry to the Audit Log. This action type can be useful when you want to track various changes of values or milestones achieved, for example.                                                         | ■ Object  
■ Key value  
■ Old value  
■ New value                                            | No          |
| Branch              | Goes to a specified other step when a specified condition is met.                                                                                                                                              | ■ Condition  
■ Destination                                             | No          |
| Call Database Method| Calls a stored procedure, passing any parameters required by or returned from the stored procedure.                                                                                                               | ■ Stored procedure name  
■ Parameters                                                  | No          |
| Call IDO Method     | Invokes an IDO method, passing any parameters required by or returned from the IDO method.                                                                                                                                              | ■ IDO  
■ Method  
■ Parameters                                                  | No          |
| Call Web Service    | Invokes a Web service, passing any parameters required by or returned from the Web service.                                                                                                                                              | ■ URL  
■ Operation (Method)  
■ Parameters  
■ Output parameters                                    | No          |
<table>
<thead>
<tr>
<th>Action type</th>
<th>What it does</th>
<th>Associated event action parameters</th>
<th>Adjourning?</th>
</tr>
</thead>
</table>
| Dispatch IDO Request | Dispatches an IDO request to execute some action on a system to which the user is not currently connected or signed in. This IDO request can be, for example, to load a collection, update a collection, or invoke an IDO method on an IDO runtime service at another "enterprise" (for example, one of your customers or vendors who also uses SyteLine) or with another site in your own enterprise. This action type opens a session on the remote database server, performs the requested action, and closes the session. If appropriate, this action type also receives any response data from the remote database server and places it in a return variable or return parameter. You can then use the Transform XML action type to parse the return data. | IDO Request XML  
URL  
Output: IDO Response XML                                                                 | No          |
| Execute IDO Request  | Executes an IDO request on the system to which the user is currently connected or signed in. This IDO request can be, for example, to load a collection, update a collection, or invoke an IDO method. This action type:  
- Optionally, opens a new session (closing it when the requested action is finished).  
- Performs the requested action.  
- If appropriate, receives any response data generated by the IDO action and places it in a return variable or return parameter. You can then use the Transform XML action type to parse the return data. | IDO Request XML  
Output: IDO Response XML                                                                 | No          |
| Fail                 | Ends the handler execution with an error status when a specified condition is met and, optionally, specifies a failure or error message to be displayed on the Event Status form. | Condition  
Result (error message)                                                                 | No          |
| Finish               | Marks the event handler as being finished successfully, optionally when a specified condition is met. Optionally, specifies a failure or error message to be displayed on the Event Status form. | Condition  
State  
Image  
Result                                                                 | No          |
<table>
<thead>
<tr>
<th>Action type</th>
<th>What it does</th>
<th>Associated event action parameters</th>
<th>Adjourning?</th>
</tr>
</thead>
</table>
| Generate Event   | Generates another event, optionally when a specified condition is met, passing any parameters required by or returned from the other event.                                                                 | ■ Condition  
 ■ Event name  
 ■ Synchronous  
 ■ Parameters  
 ■ Output parameters                                     | No           |
| Goto             | Directs the handler flow to an action step other than the next one in the sequence. This can be useful when you want to redirect the flow based on some condition that has been met in another action step. | ■ Destination                                                     | No           |
| Load IDO Collection | Loads records from a specified collection associated with an IDO, up to the record cap. You can use various options on this form to limit the amount of data returned by this event action. The data retrieved by this action is held in a “result set”, which you must then refer to for further action. | ■ IDO  
 ■ Properties list  
 ■ Filter and Order By clause OR custom load method and parameters  
 ■ Record cap  
 ■ Result set identifier | No           |
| Load IDO Row     | Loads a specified record (row) from a collection associated with an IDO.                                                                                                                                 | ■ IDO  
 ■ Properties list  
 ■ Filter and Order By clause OR custom load method and parameters  
 ■ Output specification                                                                 | No           |
| Notify           | Sends a system-generated message to recipients (system users), optionally when a specified condition is met. This notification is transmitted internally within the WinStudio system, without having to use external e-mail or other messaging systems. | ■ Condition  
 ■ Subject  
 ■ Category  
 ■ Body  
 ■ To recipient list  
 ■ CC (Copy) list  
 ■ Save in Sent Items  
 ■ Filter form  
 ■ Filter spec  
 ■ Entry form                                                                 | No           |
<table>
<thead>
<tr>
<th>Action type</th>
<th>What it does</th>
<th>Associated event action parameters</th>
<th>Adjourning?</th>
</tr>
</thead>
</table>
| Prompt           | Sends a system-generated message to recipients (system users) and awaits a response from each. Also sends an e-mail to recipients for whom **Send E-mail Prompts** is specified in the **Users** form. To send e-mails for prompts, you must have an SMTP server designated on the **Intranets** form and automatic relaying of e-mail messages configured on the SMTP server. Counts votes according to a specified rule. If no response or an insufficient response is received within a certain time period, optionally goes to an alternate action, such as the following:  
  - Perform an alternate prompt to an alternate set of recipients.
  - Set a default and branch to the subsequent event action.
  - Go directly to the subsequent event action (the same as if a sufficient number of responses were received). For information about voting rules, see [Voting Rules](#) on page 71.                                                                 | ■ Condition  
■ Subject  
■ Category  
■ Body  
■ To recipient list  
■ CC (Copy) list  
■ Save in Sent Items  
■ Question  
■ Choices  
■ Voting rule  
■ Preferred choice  
■ Minimum count or percentage  
■ Timeout  
■ Timeout destination  
■ Filter form  
■ Filter specification  
■ Entry form  
■ Payload access  
■ Modified payload access  
■ Unmodified payload access | Yes          |
| Run Background Task | Places a task on the background task queue.                                                                                                                                                                    | ■ Task name  
■ Task parameters  
■ Task status  
■ Task number | No           |
| Send Email       | Sends an external e-mail to recipients (system users). To use this action, you must have an SMTP server designated on the **Intranets** form and automatic relaying of e-mail messages configured on the SMTP server.                                                                 | ■ Condition  
■ Subject  
■ Body  
■ To recipient list  
■ CC (Copy) list | No           |
| Set Attributes   | Sets text that displays in the **Event Title** field of the **Event Status** form.                                                                                                                             | ■ Title | No           |
| Set Values       | Sets event variable(s) and/or event parameter(s) to given values or expressions, optionally when a specified condition is met.                                                                                                   | ■ Condition  
■ Variable/Expression pairs  
■ Parameter/Expression pairs  
■ Session variable/Expression pairs  
■ Row number and property/Expression pairs  
■ Property/Expression pairs  
■ Value | No           |
<table>
<thead>
<tr>
<th>Action type</th>
<th>What it does</th>
<th>Associated event action parameters</th>
<th>Adjourning?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sleep</td>
<td>Waits until a specified period of time has elapsed before proceeding to the next action.</td>
<td>Interval</td>
<td>Yes</td>
</tr>
</tbody>
</table>
| Transform XML  | Takes XML as input and transforms it for use elsewhere. This action type transforms specified XML, based on transformation rules contained in the XSLT, with the resulting XML stored in a result variable or parameter. A transform XML operation can range from simple (picking out a single specific element from a large document, to be used as part of a SUBSTITUTE expression in a Notify action) to complex (translating an entire document from one format to another, to be used as input to some other process). | XML  
XSL  
Output: Result XML                                                                                   | No          |
| Update Collection | Updates an IDO collection by inserting, updating, or deleting records within that collection. Using this action type, you can either update a collection directly from a specified IDO, or you can update a collection that is already loaded into a named result set. | IDO  
Action  
Property/Expression pairs  
Collection  
Row  
Optimistic Lock  
Commit                                                                                                      | No          |
| Wait           | Instructs the system to wait for a specified condition to be met before proceeding to the next action. If the condition does not become true within a certain time period, optionally goes to an alternate action. A restest interval of zero (0) means that the condition is never retested. A timeout setting of zero (0) means that the event action never times out, that is, that it waits indefinitely. | Condition  
Retest interval  
Timeout  
Timeout destination                                                                                         | Yes         |
Event Action Parameters

The following table lists (in alphabetic order) and describes all the event action parameters and their functions. For a list of all action types with the possible parameters associated with them, see Event Action Types on page 135.

For information about constructing and using event expressions, see Expression Functions on page 155 and Expression Operators on page 171. For the complete grammar available for constructing expressions, see Expression Grammar on page 173.

When using the table, keep the following in mind:

<table>
<thead>
<tr>
<th>The following:</th>
<th>Refers to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLCAPS</td>
<td>Functions and punctuation to be entered and stored verbatim.</td>
</tr>
<tr>
<td>Parentheses ( )</td>
<td>Content that varies according to the context. For example, for a SUBJECT(expr) parameter, expr represents the actual content that is sent in the message.</td>
</tr>
<tr>
<td>Equal signs (=)</td>
<td>Content that varies according to the context. For example, for a SUBJECT(expr) parameter, expr represents the actual content that is sent in the message.</td>
</tr>
<tr>
<td>Commas</td>
<td>Content that varies according to the context. For example, for a SUBJECT(expr) parameter, expr represents the actual content that is sent in the message.</td>
</tr>
<tr>
<td>Content in italics</td>
<td>Indicates more like the preceding element, separated by commas. Parentheses indicate that the repetition of elements is optional.</td>
</tr>
<tr>
<td>Ellipses (...)</td>
<td>Indicates more like the preceding element, separated by commas. Parentheses indicate that the repetition of elements is optional.</td>
</tr>
<tr>
<td>scalarExpr</td>
<td>Scalar (numeric, string, date, or typeless) event expressions. For more information about constructing and using event expressions, see Expression Functions on page 155.</td>
</tr>
<tr>
<td>Boolean</td>
<td>A Boolean event expression. For more information about constructing and using event expressions, see Expression Functions on page 155.</td>
</tr>
</tbody>
</table>
| name var (variable) parm (parameter) collection | Identifiers used for named or contextual purposes. These identifiers can consist of:  
  - An initial uppercase or lowercase letter or underscore (_), followed by  
  - Any combination of letters, numerals, underscores, square brackets ([ ]), and/or periods. |
<p>| prop (property) | Specifies the property name as a string expression. To specify the property name as a literal value, enclose it in double quotation marks (&quot; &quot;). |
| action         | Event action sequence number (values to be adjusted automatically if actions are resequenced). |</p>
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Used with action type(s)</th>
<th>Description/Purpose</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>■ Update Collection</td>
<td>Indicates what the system is to do with the data.</td>
<td>One of the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>■ ACTION(INSERT)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>■ ACTION(UPDATE)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>■ ACTION(DELETE)</td>
</tr>
<tr>
<td>Body</td>
<td>■ Notify</td>
<td>Indicates the content that is to be generated for the body of the message, whether for an internal system message or an e-mail. The CLIENTSUBSTITUTE() function can be used to build an expression that is automatically translated locally for each Inbox user and e-mail prompt recipient (applicable only for Notify and Prompt).</td>
<td>BODY(scalarExpr)</td>
</tr>
<tr>
<td></td>
<td>■ Prompt</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ Send Email</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>■ Notify</td>
<td>Specifies a category to group the message with others of a similar purpose.</td>
<td>CATEGORY(scalarExpr)</td>
</tr>
<tr>
<td></td>
<td>■ Prompt</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cc</td>
<td>■ Notify</td>
<td>Indicates a list of secondary recipients of the message, delimited by semi-colons (;). Do not leave spaces between multiple recipients in the list. For Notify and Prompt actions, recipients are identified by their user IDs on the system. For Send Email actions, recipients are identified by their e-mail addresses.</td>
<td>CC(scalarExpr)</td>
</tr>
<tr>
<td></td>
<td>■ Prompt</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ Send Email</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Examples:**
CC("jdoe;tsmith")
CC("jdoe@provider.net")

Note that literal values for user IDs or e-mail addresses must be enclosed in double quotes ("").
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Used with action type(s)</th>
<th>Description/Purpose</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Choices</td>
<td>Prompt</td>
<td>Indicates a comma-separated list of return values and button labels (or link labels for e-mail). Note that the parser strips out whitespace, so it is OK to include spaces between items in the list. In each pair, the first item is the value to be returned if the recipient selects that option. The second item is the label that appears on the selection button/link. For the button/link labels, you can use strings defined in the Strings table or literal values. For e-mails, strings are translated and formatted based on the Default Language specified for the recipient in the Users form. On the Inbox form, strings are translated and formatted based on the Language setting specified on the Runtime Behavior Settings. The <code>scalarExpr</code> must consist of an even number of list items. The default button/link labels are <em>Approve</em> and <em>Disapprove</em>, with return values of 1 and 0 respectively.</td>
<td><code>CHOICES(scalarExpr)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Examples:</strong> <code>CHOICES( '&quot; + GC(YesValue) + &quot;,sYes,&quot; + GC(NoValue) + &quot;,sNo&quot; )</code> Note that the expression must be enclosed in double-quotes (&quot;&quot;). For more information about designating strings for this parameter, see Prompts and Responses on page 71.</td>
<td></td>
</tr>
<tr>
<td>COLLECTION</td>
<td>Update Collection</td>
<td>Indicates the name (ID) of a result set that was previously loaded using a Load IDO Collection action type.</td>
<td><code>COLLECTION(ID)</code></td>
</tr>
<tr>
<td>(Result Set ID)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td>Update Collection</td>
<td>Determines whether or not the changes made up to this point in the action flow are to be committed to the database.</td>
<td><code>COMMIT(Boolean)</code></td>
</tr>
<tr>
<td>Condition</td>
<td>Achieve Milestone, Branch, Fail, Finish, Generate Event, Notify, Prompt, Send Email, Set Values, Wait</td>
<td>Sets up the condition that determines when the action is executed.</td>
<td><code>CONDITION(Boolean)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Examples:</strong> <code>CONDITION(NOT PROPERTYMODIFIED(&quot;CreditLimit&quot;))</code> <code>CONDITION(P(&quot;Quantity&quot;) &lt; 500)</code></td>
<td></td>
</tr>
<tr>
<td>Configuration Name</td>
<td>Dispatch IDO Request</td>
<td>Determines the configuration on a remote server from which you want to make an IDO request.</td>
<td><code>CONFIGNAME(scalarExpr)</code></td>
</tr>
<tr>
<td>Parameter</td>
<td>Used with action type(s)</td>
<td>Description/Purpose</td>
<td>Syntax</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
</tbody>
</table>
| Custom Load (method)     | ● Load IDO Collection  
● Load IDO Row                                                                 | Indicates the name of the custom load method to use in loading the collection data.                                                                                                                                  | CUSTOMLOAD(scalarExpr)                     |
| Custom LoadParms (Custom load method parameters) | ● Load IDO Collection  
● Load IDO Row                                                                 | Indicates the parameters to be passed to the custom load method.                                                                                                                                                       | CUSTOMLOADPARMS(scalarExpr)                |
| Description              | ● Audit                                                                                     | Defines the description to be passed to the audit log. The CLIENTSUBSTITUTE() function can be used to build an expression that is automatically translated locally for each user viewing the Audit Log form. | DESC(scalarExpr)                           |
| Destination              | ● Branch  
● Goto                                                                                     | Indicates the action sequence number of the event action the system is to go to when executing this action.                                                                                                         | DEST(action)  
**Example:** DEST(40)                  |
| Entry Form               | ● Notify  
● Prompt                                                                                   | Indicates the name of the form to be launched to edit the event variables for the event handler.                                                                                                                   | ENTRYFORM(scalarExpr)                      |
| Event Name               | ● Generate Event                                                                            | Indicates the name of the event to be generated.                                                                                                                                                                       | EVENTNAME(scalarExpr)                      |
| Filter or Filter Spec    | ● Load IDO Row  
● Load IDO Collection  
● Notify  
● Prompt                                                                 | Identifies the filter to use in restricting data retrieval. This parameter is not valid with CUSTOMLOAD. Functionally, this is the same as the Permanent Filter Expression property in WinStudio development. | FILTER(scalarExpr)                         |
| Filter Form              | ● Notify  
● Prompt                                                                                   | Indicates the name of the form to be launched to display information about the row represented by the event variables for the event handler.                                                                       | FILTERFORM(scalarExpr)                     |
| IDO                      | ● Call IDO Method  
● Load IDO Collection  
● Load IDO Row  
● Update Collection                                                                 | Indicates the name of the IDO on which to perform the operation.                                                                                                                                                      | IDO(scalarExpr)                            |
| IDO Request XML          | ● Dispatch IDO Request  
● Execute IDO Request                                                                 | Contains the correctly formatted XML content of the IDO request.                                                                                                                                                      | IDOREQUEST(scalarExpr)                     |
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Used with action type(s)</th>
<th>Description/Purpose</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image</td>
<td>Achieve Milestone, Finish</td>
<td>Specifies the name of the graphics file to display with the event in the status forms. This path and file name must be in the format: <code>Drive:\folder\subfolder\...\myGraphicFile.ext</code> The specified graphics file must be one of the supported types. For more information, see the online Help.</td>
<td>IMAGE(scalarExpr) Example: IMAGE(&quot;C:\WINDOWS\Web\Wallpaper\Tulips.jpg&quot;)</td>
</tr>
<tr>
<td>Interval</td>
<td>Sleep</td>
<td>Specifies the amount of time (in seconds) that the system is to pause before moving to the next action.</td>
<td>INTERVAL(numericExpr)</td>
</tr>
<tr>
<td>Key value</td>
<td>Audit</td>
<td>Specifies the text that is to appear in the Primary Key column of the Audit Log form when this action runs.</td>
<td>KEY(scalarExpr)</td>
</tr>
<tr>
<td>Method</td>
<td>Call IDO Method</td>
<td>Specifies the name of the IDO method to call.</td>
<td>METHOD(scalarExpr)</td>
</tr>
<tr>
<td>Method Parameters</td>
<td>Set Values</td>
<td>Changes method parameter values during an IdoOnInvoke event. paramNumber is the 1-based position of the parameter to change. scalarExpr is the value to which the parameter is to be set. For multiple parameters, include them as a comma-separated list.</td>
<td>SETMETHODPARM-VALUES(paramNumber = scalarExpr[, ...])</td>
</tr>
</tbody>
</table>
| Minimum (Count or Percentage) | Prompt | **Count**—Indicates the minimum number of votes required for a choice to win. Must be used in conjunction with a ConditionalPlurality, or MinimumCount voting rule.  
**Percentage**—Indicates the minimum percentage of the total vote (based on recipients of the message) required for a choice to win. Must be used in conjunction with a MinimumPercentage voting rule. For more information about voting rules, see Voting Rules on page 71. | MINIMUM(numericExpr) |
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Used with action type(s)</th>
<th>Description/Purpose</th>
<th>Syntax</th>
</tr>
</thead>
</table>
| Modified Payload Access   | Prompt                   | Specify how message payload elements that represent modified properties are handled:  
  - HIDDEN - The modified parts of the payload are hidden, for example, for a secret ballot.  
  - READONLY - The modified parts of the payload are display-only, so the recipient cannot further modify the contents.  
  This specification applies to event message variables with names in the form Row.Property that have a corresponding Row.Property.Modified element with a value of 1.  
  This setting is overridden by settings for the Default Access (Event Variable Groups form) or Access (Event Actions form, Variables grid) for the same property name. | Either of the following:  
  MODIFIEDPAYLOADACCESS(HIDDEN)  
  MODIFIEDPAYLOADACCESS(READONLY) |
<p>| New Value                 | Audit                    | Specifies the text that is to appear in the New Value column of the Audit Log form when this action runs. Represents the new value of something that was changed. | NEW(scalarExpr)                                                        |
| Old Value                 | Audit                    | Specifies the text that is to appear in the Old Value column of the Audit Log form when this action runs. Represents the original value of something that was changed. | OLD(scalarExpr)                                                        |
| Operation                 | Call Web Service         | Specifies the name of the Web service to call.                                                                                                                                                                       | METHOD(scalarExpr)                                                     |
| Optimistic Lock           | Update Collection        | Determines whether optimistic locking will be employed for safety.                                                                                                                                                 | OPTIMISTICLOCKING(Boolean)                                            |</p>
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Used with action type(s)</th>
<th>Description/Purpose</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order By</td>
<td>- Load IDO Collection</td>
<td>Specifies the name of a valid property to be used for the sort order on the collection when it is loaded and displayed. For example, if you select <code>CreateDate</code> as the property for this field, the system sorts the collection according to the dates on which transactions were recorded. You can use multiple properties, and the system prioritizes multiple properties in the order in which they are listed. In other words, the first property would be the primary property for sorting; the next property would be the secondary property for sorting; and so on. Multiple properties must be specified as a comma-separated list. This parameter is not valid with CUSTOMLOAD.</td>
<td><code>ORDERBY(scalarExpr)</code>&lt;br&gt;&lt;br&gt;<strong>Examples:</strong>&lt;br&gt;<code>ORDERBY(&quot;PoLine&quot;)</code>&lt;br&gt;<code>ORDERBY(&quot;CoRelease ASC, Coltem&quot;)</code>&lt;br&gt;Note that, in the second example, there is an attribute &quot;ASC&quot; specified. This tells the system to sort the list in ascending order, which is also the default. You could instead use DESC to tell the system to sort the list in descending order.</td>
</tr>
<tr>
<td></td>
<td>- Load IDO Row</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output (SET)</td>
<td>- Load IDO Row</td>
<td>Specifies where the system is to place the IDO row data. You can use return variables (RV) or return parameters (RE). Multiple outputs can be designated, using a comma-separated list of return value-name pairs. A separate value-name pair is required for each property being returned.</td>
<td>`SET(RV(var)</td>
</tr>
<tr>
<td>Output Parameters (SET)</td>
<td>- Generate Event <img src="image.png" alt="Image" /></td>
<td>Specifies where the system is to place the output (return) data from the event or Web service. You can use return variables (RV) or return parameters (RE). Multiple output parameters can be returned, using a comma-separated list of return value-name pairs. Each name is the name of a node in the XML response document.</td>
<td>`SET( RV(var)</td>
</tr>
<tr>
<td>Parameter</td>
<td>Used with action type(s)</td>
<td>Description/Purpose</td>
<td>Syntax</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------------</td>
<td>---------------------</td>
<td>--------</td>
</tr>
</tbody>
</table>
| Parameters [1] | ■ Call Database Method  
■ Call IDO Method | Specifies a comma-separated list of parameters required by the IDO method or stored procedure. For alternate ways to pass parameters in these cases, see Passing Parameters from Actions on page 47. If an event variable is named using this function and that variable does not already exist for the current handler, the system creates it automatically as a non-persistent variable. If a parameter is set using this function, and that parameter was not passed into the current event, the system creates it automatically as an output parameter. | PARMS( scalarExpr | RV(var) | RE(param)[, ... ]) |
| Parameters [2] | ■ Call Web Service  
■ Generate Event | Specifies a comma-separated list of parameters to be passed as input to the event or Web service. These parameters must be in the form of name-value pairs. | PARMS(name=scalarExpr[ … ]) |
<p>| Parameters [3] | ■ Set Values | Specifies parameters as name-value pairs. For multiple parameters, include them as a comma-separated list of name-value pairs. If a parameter is set using this function, and that parameter was not passed into the current event, the system creates it automatically as an output parameter. | SETPARMVALUES( paramName = scalarExpr[, … ]) |
| Password | ■ Dispatch IDO Request | Indicates the password associated with the user ID specified in the User Name field. | PASSWORD(scalarExpr) |</p>
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Used with action type(s)</th>
<th>Description/Purpose</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Payload Access</td>
<td>■ Prompt</td>
<td>Sets the recipient's access to payloads for variables whose name is in the form Row.Property:&lt;br&gt;■ HIDDEN – The payload is hidden, for example, because it contains no information relevant to the voting.&lt;br&gt;■ READONLY - The entire payload is display-only, so the recipient cannot modify any of the contents.&lt;br&gt;This specification applies to event message variables with names in the form Row.Property.&lt;br&gt;This setting is overridden by settings for the Default Access (Event Variable Groups form) or Access (Event Actions form, Variables grid) for the same property name.</td>
<td>Either of the following:&lt;br&gt;PAYLOADACCESS(HIDDEN)&lt;br&gt;PAYLOADACCESS(READONLY)</td>
</tr>
<tr>
<td>Preferred Choice</td>
<td>■ Prompt</td>
<td>Specifies the preferred choice among multiple voting choices. This parameter must be used in conjunction with one of the PreferredChoice voting rules.</td>
<td>PREFCHOICE(scalarExpr)</td>
</tr>
<tr>
<td>Properties [1]</td>
<td>■ Load IDO Collection</td>
<td>Identifies a comma-separated list of properties to load from the IDO collection that is specified in the IDO field.</td>
<td>PROPERTIES(scalarExpr)</td>
</tr>
<tr>
<td>Properties [2]</td>
<td>■ Update Collection</td>
<td>Specifies the primary keys identifying the row(s) to be updated, followed by the properties for which you want to set values, both as name-value pairs, in a comma-separated list.</td>
<td>SETPROPVVALUES(propName = scalarExpr[, ...])</td>
</tr>
<tr>
<td>Properties [3]</td>
<td>■ Set Values</td>
<td>Specifies the properties for which you want to set values as name-value pairs. For multiple properties, include them as a comma-separated list. Used only with IdoOnItemInsert, IdoOnItemUpdate, and IdoOnItemDelete framework (Core) events.</td>
<td>SETPROPVVALUES(propName = scalarExpr[, ...])</td>
</tr>
<tr>
<td>Parameter</td>
<td>Used with action type(s)</td>
<td>Description/Purpose</td>
<td>Syntax</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>Question</td>
<td>Prompt</td>
<td>Specifies the content of the direct question that recipients are to respond to when they receive the message. This question appears in the Question field on the Response tab of the Inbox form. The CLIENTSUBSTITUTE() function can be used to build an expression that is automatically translated locally for each Inbox user and e-mail prompt recipient.</td>
<td>QUESTION(scalarExpr)</td>
</tr>
<tr>
<td>Quorum</td>
<td>Prompt</td>
<td>Specifies the number of votes to await before closing voting, expiring the remaining unvoted messages, and continuing to the subsequent event action. If this field contains a value, the number overrides the automatic quorum algorithm used to calculate whether further votes are required in order to make the VOTINGRESULT() non-blank. If the Quorum value is positive, it implies that Wait for Quorum is TRUE. If the Quorum value is non-positive, it implies that Wait for Quorum is FALSE.</td>
<td>QUORUM(numericExpr)</td>
</tr>
<tr>
<td>Record Cap</td>
<td>Load IDO Collection</td>
<td>Designates the maximum number of records to be returned when the collection is loaded. The value you set for this field overrides any system or personal settings when executing this handler.</td>
<td>RECORDCAP(numericExpr) Example: RECORDCAP(500)</td>
</tr>
<tr>
<td>Response (SET)</td>
<td>Dispatch IDO Request</td>
<td>Specifies the type—return variable (RV) or return parameter (RE)—and name of the container that is to hold the XML response (RESULT) data from the IDO request.</td>
<td>SET(RV(var) = RESULT)</td>
</tr>
<tr>
<td></td>
<td>Execute IDO Request</td>
<td></td>
<td>SET(RE(param) = RESULT)</td>
</tr>
<tr>
<td>Result</td>
<td>Finish</td>
<td>Specifies the content of the message that is to be displayed if the event handler fails or finishes at this point. This message displays in the Results field of the Event Status form. The CLIENTSUBSTITUTE() function can be used to build an expression that is automatically translated locally for each user viewing the Event Status form.</td>
<td>RESULT(scalarExpr) Example: RESULT(&quot;The transaction was disapproved by the purchasing manager.&quot;)</td>
</tr>
<tr>
<td></td>
<td>Fail</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Used with action type(s)</td>
<td>Description/Purpose</td>
<td>Syntax</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>--------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>Result Set Assignment</td>
<td>Load IDO Collection</td>
<td>Designates the name (ID) of the result set where the system is to put the loaded collection.</td>
<td>SET(R(id) = RESULT)</td>
</tr>
<tr>
<td>Result Set ID</td>
<td>Update Collection</td>
<td>Identifies the name (ID) of a result set that was previously loaded using a Load IDO Collection action type.</td>
<td>COLLECTION(id)</td>
</tr>
<tr>
<td>Result XML</td>
<td>Transform XML</td>
<td>Specifies the return type—return variable (RV) or return parameter (RE)—and name of the variable or parameter that is to receive the resulting XML transformation output.</td>
<td>SET(RV(var) = RESULT) SET(RE(param) = RESULT)</td>
</tr>
<tr>
<td>Retest Interval</td>
<td>Wait</td>
<td>Designates the amount of time, in seconds, that the system is to wait before retesting for the condition.</td>
<td>RETEST(numericExpr)</td>
</tr>
<tr>
<td>Row</td>
<td>Update Collection</td>
<td>Indicates the row (record) to be updated in the collection.</td>
<td>ROW(numericExpr)</td>
</tr>
<tr>
<td>Save in Sent Items</td>
<td>Notify</td>
<td>Determines whether a copy of the message is to be saved in the originator’s Sent Items folder.</td>
<td>SAVEMESSAGE(Boolean)</td>
</tr>
<tr>
<td></td>
<td>Prompt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Session Variables</td>
<td>Set Values</td>
<td>Specifies session variables as name-value pairs. For multiple variables, include them as a comma-separated list.</td>
<td>SETGLOBVALUES( varName = scalarExpr[, …])</td>
</tr>
<tr>
<td></td>
<td>Achieve Milestone</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Finish</td>
<td></td>
<td></td>
</tr>
<tr>
<td>State</td>
<td>Achieve Milestone</td>
<td>Indicates the text to display on the Event Status form, Event State field, when the action runs. The CLIENTSUBSTITUTE() function can be used to build an expression that is automatically translated locally for each user viewing the Event Status form.</td>
<td>STATE(scalarExpr)</td>
</tr>
<tr>
<td></td>
<td>Finish</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stored Procedure</td>
<td>Call Database Method</td>
<td>Specifies the name of the stored procedure to call.</td>
<td>METHOD(scalarExpr)</td>
</tr>
<tr>
<td>(METHOD)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subject</td>
<td>Notify</td>
<td>Specifies the subject line for the message, whether for an internal system message or an e-mail. The CLIENTSUBSTITUTE() function can be used to build an expression that is automatically translated locally for each Inbox user and e-mail prompt recipient (applicable only for Notify and Prompt).</td>
<td>SUBJECT(scalarExpr)</td>
</tr>
<tr>
<td></td>
<td>Prompt</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Send Email</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Used with action type(s)</td>
<td>Description/Purpose</td>
<td>Syntax</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>Synchronous</td>
<td>Generate Event</td>
<td>Instructs the system whether to treat the generated event as a synchronous event.</td>
<td>SYNCHRONOUS(Boolean)</td>
</tr>
<tr>
<td></td>
<td>TRUE</td>
<td>True = Synchronous</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td>False = Asynchronous</td>
<td></td>
</tr>
<tr>
<td>Task Name</td>
<td>Run Background Task</td>
<td>Specifies the name of the task to be placed on the background task queue, as defined on the Background Task Definitions form.</td>
<td>TASKNAME(scalarExpr)</td>
</tr>
<tr>
<td>Task Number</td>
<td>Run Background Task</td>
<td>Indicates the name of the event variable that is to receive the task number as output from the Task Manager when the task is placed on the background task queue.</td>
<td>TASKNUMBER(ID)</td>
</tr>
<tr>
<td>Task Parameters</td>
<td>Run Background Task</td>
<td>Specifies any parameters that the background task requires to run correctly.</td>
<td>TASKPARMS(scalarExpr)</td>
</tr>
<tr>
<td>Task Status</td>
<td>Run Background Task</td>
<td>Specifies the task status that the status is to have initially when it is placed on the Background Task Queue.</td>
<td>Either of the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>READY – Places the task on the Background Task Queue to be run as soon as possible.</td>
<td>TASKSTATUS(READY)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WAITING – Places the task on the Background Task Queue to be run only when the scheduling requirements for that task, as set on the Background Queue form, are</td>
<td>TASKSTATUS(WAITING)</td>
</tr>
<tr>
<td>Time Out</td>
<td>Prompt</td>
<td>The amount of time (in seconds) that the system is to wait for a response (in the case of a Prompt action) or that it must wait before retesting (in the case of a Wait action).</td>
<td>TIMEOUT(numericExpr)</td>
</tr>
<tr>
<td></td>
<td>Wait</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time Out Destination</td>
<td>Prompt</td>
<td>Specifies which action sequence step the system should move to when the time-out limit has been reached.</td>
<td>DEST(action)</td>
</tr>
<tr>
<td></td>
<td>Wait</td>
<td>The action sequence step is identified by the Action Sequence number on the Event Actions form.</td>
<td></td>
</tr>
<tr>
<td>Title (Event title)</td>
<td>Set Attributes</td>
<td>Specifies the Event Title as you want it to appear on the Event Status form, when the event is running or finished.</td>
<td>SET(EVENTTITLE = scalarExpr)</td>
</tr>
<tr>
<td>Parameter</td>
<td>Used with action type(s)</td>
<td>Description/Purpose</td>
<td>Syntax</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>To (Recipient list)</td>
<td>■ Notify</td>
<td>Designates the list of the primary recipients of the message, delimited by semi-colons (;). Do not leave spaces between multiple recipients. For Notify and Prompt actions, recipients are identified by their user IDs. For Send Email actions, recipients are identified by their e-mail addresses.</td>
<td>TO(scalarExpr)</td>
</tr>
<tr>
<td></td>
<td>■ Prompt</td>
<td></td>
<td>Examples:</td>
</tr>
<tr>
<td></td>
<td>■ Send Email</td>
<td></td>
<td>TO(“jdoe;tsmith”)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TO(“<a href="mailto:jdoe@provider.net">jdoe@provider.net</a>”)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note that literal values for user IDs or e-mail addresses must be enclosed in double quotes (“”).</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transactional</td>
<td>■ Generate Event</td>
<td>Instructs the system whether to treat the generated event as transactional. For more information about transactional event handlers, see Transactions on page 31.</td>
<td>TRANSACTIONAL(Boolean)</td>
</tr>
</tbody>
</table>
| Unmodified Payload Access     | ■ Prompt                  | Specifies how message payload elements that represent unmodified properties are handled:  
|                               |                          | ■ HIDDEN - The unmodified parts of the payload are hidden. This allows the recipient to focus on the modified parts of the payload when evaluating the question.                                                   | Either of the following:  
|                               |                          | ■ READONLY - The unmodified parts of the payload are display-only, so the voter cannot modify the contents. This specification applies to event message variables with names in the form Row.Property that do not have a corresponding Row.Property.Modified element with a value of 1.  
|                               |                          | This setting is overridden by settings for the Default Access (Event Variable Groups form) or Access (Event Actions form, Variables grid) for the same property name.                        | UNMODIFIEDPAYLOADACCESS(HIDDEN)  
|                               |                          |                                                                                                                                                                                                                     | UNMODIFIEDPAYLOADACCESS(READONLY) |
| URL                           | ■ Call Web Service       | Identifies the URL that is to receive the request. The URL must be in standard HTTP format.                                                                                                                       | URL(scalarExpr)   |
|                               | ■ Dispatch IDO Request    |                                                                                                                                                                                                                     |                   |

**Transactions**
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Used with action type(s)</th>
<th>Description/Purpose</th>
<th>Syntax</th>
</tr>
</thead>
</table>
| Use Local Password      | Dispatch IDO Request      | Instructs the system whether to automatically retrieve the password associated with the user ID (User Name) and include it in the IDO request.  
|                         |                          | **TIP:** If you want to use this option, you must make sure that the remote system has an identical login (user name and password) to the one you want to use from the current local system.       | USELOCALPASSWORD(Boolean)                  |
| User Name               | Dispatch IDO Request      | Designates a valid user ID for the remote system configuration. This allows a session to open.  
|                         |                          | **TIP:** Make sure you enter a user ID that has permissions and licensing sufficient to do whatever you are requesting from the IDO on the remote system.                                             | USERNAME(scalarExpr)                       |
| Value                   | Set Values               | Used only for the  
|                         |                          | **SessionOnVarChanged** framework event.  
|                         |                          | Specifies a value to be assigned to the session variable that, when changed, generated the event. This value replaces whatever value was requested by the process that initiated the change. In effect, this overrides the original change request behind the scenes.  
|                         |                          | For example, suppose you have an event that is set to be executed whenever a user tries to reset a particular session variable. This event can be used to override the user-initiated change and reset the variable to whatever value you designate here. | SETVALUE(scalarExpr)                       |
| Variables               | Set Values               | Specifies variable definitions as name-value pairs. For multiple variables, include them as a comma-separated list.  
|                         |                          | If a variable does not exist when the event is executed, the system creates the variable when needed.  
|                         |                          | **TIP:** Including a null value for a variable (such as `myVar =`) generates a syntax error. To designate a null value for the value of a variable, use a double set of quotation marks (`myVar = ""`). | SETVARVALUES(  
<p>|                         |                          | <code>varName = scalarExpr[, ...]</code>)                                                                                                                          |                                             |</p>
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Used with action type(s)</th>
<th>Description/Purpose</th>
<th>Syntax</th>
</tr>
</thead>
</table>
| Voting Rule        | ■ Prompt                 | Designates the method to be used for counting and evaluating votes. Possible values include:  
- Majority  
- Plurality  
- ConditionalPlurality  
- MinimumCount  
- MinimumPercentage  
- EarliestResponse  
- PreferredChoice  
- MinimumCountPreferredChoice  
- MinimumPercentagePreferredChoice  
For more information about these values, see Voting Rules on page 71. | VOTINGRULE(rule)  
Example:  
■ VOTINGRULE(Plurality) |
| Wait for Quorum   | ■ Prompt                 | Specifies whether to await a quorum of responses before continuing to the subsequent event action on a prompt action:  
- FALSE - Voting remains open, no messages are expired, and Timeout and Timeout Destination are ignored.  
- TRUE - Execution is adjourned until a quorum is attained.  
- Default - When Quorum is not specified, the default is TRUE. When Quorum is specified with a positive value, the default is TRUE. When Quorum is specified with a non-positive value, the default is FALSE. | WAITFORQUORUM(Boolean) |
| XML                | ■ Transform XML          | Specifies the XML code (content) to be transformed. The result can be retrieved using the RESULT function.                                                                                                         | XML(scalarExpr)             |
| XSLT               | ■ Transform XML          | Designates the stylesheet (XSLT) or a reference to the XSLT file containing the transformation rules (using the FILECONTENTS function).                                                                           | XSLT(scalarExpr)            |
Expression Functions

Standard Functions

To construct the expressions used in Event Action Parameters, use the standard expression functions in the following table, if necessary in conjunction with the Expression Operators on page 171.

For the complete grammar available for constructing expressions, see Expression Grammar on page 173.

When using the table, keep the following in mind:

<table>
<thead>
<tr>
<th>The following</th>
<th>Refers to</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNCTION( value )</td>
<td>Syntax for functions and values.</td>
</tr>
<tr>
<td>Content in italics</td>
<td>Content that varies according to the context. For example, for a V( varName ) function, varName represents the actual value (the variable name) to be used by the V (variable) function.</td>
</tr>
</tbody>
</table>

The identifiers listed as arguments below are not enclosed in double-quotes (" ") and should not be unless they are cast as literal string values in the resulting expression.

<table>
<thead>
<tr>
<th>Function / Syntax</th>
<th>What it does</th>
<th>Arguments</th>
<th>Return data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIONSEQ( )</td>
<td>Returns the design-time sequence of the currently executing event action.</td>
<td>—</td>
<td>Numeric</td>
</tr>
<tr>
<td>ACTIONTYPECNAME()</td>
<td>Returns the English-like name of the currently executing event action's type. This function returns the same value as the T-SQL function dbo.EventActionTypeName() when it is passed the EventAction.ActionType for this event action.</td>
<td>—</td>
<td>String</td>
</tr>
<tr>
<td>ANYHANDLERS-FAILED( )</td>
<td>Indicates whether any event handlers for the running event have exited with a status of Failure.</td>
<td>—</td>
<td>Boolean</td>
</tr>
<tr>
<td>APPNAME( )</td>
<td>Returns the name of the application to which the user is currently signed in. TIP: The name of the application is set in the Configuration Manager, Edit Application dialog box.</td>
<td>—</td>
<td>String</td>
</tr>
<tr>
<td>BEGINDATE( )</td>
<td>Returns the date and time the event began executing.</td>
<td>—</td>
<td>Date/Time</td>
</tr>
<tr>
<td>Function / Syntax</td>
<td>What it does</td>
<td>Arguments</td>
<td>Return data type</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>CAST(expr AS DATATYPE)</td>
<td>Causes the system to treat the specified expression as the designated data type. For example, CAST(P(&quot;CreditLimit&quot;) AS STRING) would cause the system to treat the value of the property &quot;CreditLimit&quot; as a string, rather than as a number.</td>
<td></td>
<td>Varies</td>
</tr>
<tr>
<td>CEILING( )</td>
<td>Returns the least integer greater than or equal to a specified numeric expression. NOTE: This function is equivalent to CEILING in T-SQL.</td>
<td></td>
<td>Numeric</td>
</tr>
<tr>
<td>CLIENTSUBSTITUTE</td>
<td>Creates a string combined of literal values and replacements for which values are substituted at run time. CLIENTSUBSTITUTE( ) allows inclusion of literal commas and parentheses in the string. You can concatenate the string that results from evaluating the CLIENTSUBSTITUTE( ) expression with other strings. You can pass the string to SUBSTITUTE in order to build a larger expression, for example, BODY( ), that contains the unchanged <del>LIT</del>( ) expressions.</td>
<td></td>
<td>String</td>
</tr>
<tr>
<td>COMPANYNAME( )</td>
<td>Returns the name of the company associated with the application to which the user is currently signed in. TIP: The name of the company is set using the Configuration Manager, Edit Application dialog box.</td>
<td></td>
<td>String</td>
</tr>
<tr>
<td>CONFIGNAME( )</td>
<td>Returns the configuration name associated with the running event.</td>
<td></td>
<td>String</td>
</tr>
<tr>
<td>Function / Syntax</td>
<td>What it does</td>
<td>Arguments</td>
<td>Return data type</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>CURDATETIME( )</td>
<td>Returns the current date and time, corrected for the server’s time zone (except for the database server).</td>
<td>—</td>
<td>Date/Time</td>
</tr>
<tr>
<td></td>
<td><strong>NOTE:</strong> This function is equivalent to dbo.GetSiteDate(getdate()) in T-SQL.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUSTOMDELETE( )</td>
<td>Returns the specification for a Custom Delete attribute.</td>
<td>—</td>
<td>String</td>
</tr>
<tr>
<td>CUSTOMINSERT( )</td>
<td>Returns the specification for a Custom Insert attribute.</td>
<td>—</td>
<td>String</td>
</tr>
<tr>
<td>CUSTOMUPDATE( )</td>
<td>Returns the specification for a Custom Update attribute.</td>
<td>—</td>
<td>String</td>
</tr>
</tbody>
</table>
| DATE(year, month, day) or DATE(year, month, day, hour, minute, second, millisecond) | Returns a date and/or time with the given attributes. You must use one of the two syntax models, or the system returns an error.                                                                          | **year** – A four-digit integer  
**month** – A one-digit or two-digit integer (1–12)  
**day** – A one-digit or two-digit integer (1–31)  
**hour** – A one-digit or two-digit integer (1–23)  
**minute** – A one-digit or two-digit integer (0–59)  
**second** – A one-digit or two-digit integer (0–59)  
**millisecond** – A one-digit to three-digit integer (0–999) | Date/Time          |
| DATEADD(datePart, number, date) | Returns a new date/time value based on adding or subtracting a number of intervals to the specified date. **NOTE:** This function is equivalent to DATEADD in T-SQL. For more information about possible values for arguments, see the online Help topic for this function. | **datePart** – The type of interval to be added to or subtracted from **date**  
**number** – number of intervals to be added to or subtracted from **date**. For addition (resulting in a value later than **date**), use a positive value; for subtraction (resulting in a value earlier than **date**), use a negative value.  
**date** – An expression that returns the Date/Time value to or from which you want to add or subtract. | Date/Time          |
<table>
<thead>
<tr>
<th>Function / Syntax</th>
<th>What it does</th>
<th>Arguments</th>
<th>Return data type</th>
</tr>
</thead>
</table>
| DATEDIFF(datePart, startDate, endDate) | Returns the number of date or time boundaries crossed between two specified dates. For example, you could use this function to find the number of days in a certain interval of dates. The startDate value is subtracted from the endDate. If the startDate is later than the endDate, then the return value is a negative number. If the result is out of range for integer values, the system returns an error. **NOTE:** This function is equivalent to DATEDIFF in T-SQL. For more information about possible values for arguments, see the online Help topic for this function. | datePart – Specifies the type of interval on which to calculate the difference  
startDate – The beginning date for the calculation  
endDate – The ending date for the calculation | Numeric |
| DATEPART( datePart, date)         | Returns a number indicating the value of a part of a date value for a specified date. For example, if you wanted the value of the month for a date, you might get a return value of 10 (for October). **NOTE:** This function is equivalent to DATEPART in T-SQL. For more information about possible values for arguments, see the online Help topic for this function. | datePart – The part of the date for which you want to return the value  
date – The date for which you want to return the part value | Numeric |
| DBFUNCTION( functionName, arg1, arg2, ... ) | Calls a database function. **NOTE:** This function is available only for database operations. | functionName – The name of the database function you want to call  
arg1, arg2, ... – Any arguments required by the database function | Typeless |
<p>| E( param )                        | Returns the value of an event parameter. | param – The name of the event parameter | Typeless |
| EVENTNAME( )                      | Returns the name of the event being processed. | — | String |
| EVENTREVISION()                   | Returns the design-time revision number of the active group of handlers of the event being processed. | — | Numeric |
| EVENTSTATE( )                     | Returns the designer-defined state of the running event. | — | String |
| EVENTTITLE( )                     | Returns the title associated with a specified event. | — | String |</p>
<table>
<thead>
<tr>
<th>Function / Syntax</th>
<th>What it does</th>
<th>Arguments</th>
<th>Return data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FE( param )</td>
<td>Retrieves the value of an event parameter as a string enclosed in quotes. If the parameter is empty, returns NULL as an unquoted string.</td>
<td>param – Name of the parameter for which the value is being returned</td>
<td>String</td>
</tr>
<tr>
<td>FGC( constant )</td>
<td>Retrieves the value of an event global constant as a string enclosed in quotes. If the global constant has no value assigned, returns NULL as an unquoted string.</td>
<td>constant – Name of the global constant for which the value is being returned</td>
<td>String</td>
</tr>
</tbody>
</table>
| FILECONTENTS( location ) | Returns the contents of a named text file at a specified location.                                                                                                                                     | location – Location of the text file using the standard format: C:\myFiles\myFolder\myTextFile.txt

**TIP:** This file is not required to be a *.txt file, necessarily, but it must be a file that is text-based to be useful. Other useful file types include (but are not limited to) *.htm, *.rtf, *.xml, and so on. For more information, see the online Help topic for this function. | String           |
<p>| FILTER( expr )    | Converts and presents an expression as a string enclosed in quotes. If the expression is empty, presents it as NULL with no quotes.                                                                       | expr – The expression to be converted and presented                                                                                         | String           |
| FILTERMETHODPARM( numExpr ) | Returns the value of the parameter for an IDO method call, at a specified index number. The value is returned as a string in quotes. If the parameter value is empty, returns NULL as an unquoted string. This function is used only with the IdoOnInvoke and IdoPostInvoke framework events. | numExpr – An integer or expression that resolves to an integer                                                                            | String           |</p>
<table>
<thead>
<tr>
<th>Function / Syntax</th>
<th>What it does</th>
<th>Arguments</th>
<th>Return data type</th>
</tr>
</thead>
</table>
| FILTERPROPERTY( [collection,] row#,] propertyName ) | Returns the value of a specified property, optionally in a designated row and collection result set, as a string enclosed in quotes. If the value is empty, returns NULL as an unquoted string. **NOTE:** Functionally, this is equivalent to the FP function. For more information, see the online Help for this function. | - *collection* – The name of a result set containing a collection you loaded earlier using a Load IDO Collection action. Optional.  
- *row#* – The one-based number of the row containing the property value you want to retrieve  
**NOTE:** This argument is used only with the IdoOnLoadCollection and IdoPostLoadCollection framework events, or with a Load IDO Collection action.  
- *propertyName* – The name of the property for which you want to retrieve the value  
**NOTE:** When using the IdoOnItem or IdoPostItem framework events to retrieve a property that is part of that event, this is the only accepted parameter. | String |
<p>| FILTERSTRING( )            | Returns filter used during an event. Applies only to the IdoOnLoadCollection and IdoPostLoadCollection framework (Core) events.                                                                                     | —                                                                                           | String |
| FLOOR( )                   | Returns the greatest integer that is less than or equal to the value of a number or numeric expression. <strong>NOTE:</strong> This function is equivalent to FLOOR in T-SQL.                                                  | —                                                                                           | Numeric |
| FP( [ [collection,] row#,] propertyName ) | See FILTERPROPERTY.                                                                                                                                                                                          | —                                                                                           | String |
| FSV( var )                 | Returns the value of a session variable as a string enclosed in quotes. If the variable is empty, returns NULL as an unquoted string.                                                                     | <em>var</em> – The name of the session variable for which you want to return the value                 | String |
| FV( var )                  | Returns the value of an event variable as a string enclosed in quotes. If the variable is empty, returns NULL as an unquoted string.                                                                     | <em>var</em> – The name of the variable for which you want to return the value                         | String |
| GC( constant )             | Retrieves the value of an event global constant.                                                                                                                                                           | <em>constant</em> – The name of the event global constant for which you want to return the value     | Typeless |</p>
<table>
<thead>
<tr>
<th>Function / Syntax</th>
<th>What it does</th>
<th>Arguments</th>
<th>Return data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>HANDLERIGNORESFACLURE()</td>
<td>Indicates whether the currently executing event handler ignores failures.</td>
<td>—</td>
<td>Boolean</td>
</tr>
<tr>
<td>HANDLERSEQ()</td>
<td>Returns the design-time sequence of the currently executing event handler.</td>
<td>—</td>
<td>Numeric</td>
</tr>
<tr>
<td>HANDLERSUSPENDS()</td>
<td>Indicates whether the currently executing event handler is part of a chain that suspends a framework IDO event.</td>
<td>—</td>
<td>Boolean</td>
</tr>
<tr>
<td>HANDLERSYNCHRONOUS()</td>
<td>Indicates whether the currently executing event handler is part of a chain that was fired synchronously and has not adjourned.</td>
<td>—</td>
<td>Boolean</td>
</tr>
<tr>
<td>HANDLERTRANSACTIONAL()</td>
<td>Indicates whether the unadjourned portion of the currently executing event handler is (or was) enclosed in a database transaction.</td>
<td>—</td>
<td>Boolean</td>
</tr>
<tr>
<td>HASBEGUN( action )</td>
<td>Indicates whether a specified action has begun executing.</td>
<td>action – Action sequence number</td>
<td>Boolean</td>
</tr>
<tr>
<td>HASFINISHED( action )</td>
<td>Indicates whether the specified action has finished executing.</td>
<td>action – Action sequence number</td>
<td>Boolean</td>
</tr>
<tr>
<td>IDO( )</td>
<td>Returns the name of the specified IDO collection for which an operation was requested.</td>
<td>—</td>
<td>String</td>
</tr>
<tr>
<td>IF( condition, true_expression, false_expression )</td>
<td>Depending on the outcome of a condition test, returns one of two expressions.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ condition – The condition to be tested for</td>
<td>■ condition – The condition to be tested for</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ true_expression – The expression to be returned if the condition tests TRUE</td>
<td>■ true_expression – The expression to be returned if the condition tests TRUE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ false_expression – The expression to be returned if the condition tests FALSE</td>
<td>■ false_expression – The expression to be returned if the condition tests FALSE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Can vary</td>
<td>Can vary</td>
<td></td>
</tr>
<tr>
<td>INITIATOR()</td>
<td>Returns an identifier indicating the object or process that initiated an event. This specification normally consists of a composite initiator type (form or other client) and name (form name or other client ID), separated by a period.</td>
<td>—</td>
<td>String</td>
</tr>
<tr>
<td>INSIDEDATABASE()</td>
<td>Indicates whether the current event action is executing inside the database.</td>
<td>—</td>
<td>Boolean</td>
</tr>
<tr>
<td>Function / Syntax</td>
<td>What it does</td>
<td>Arguments</td>
<td>Return data type</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------</td>
<td>-----------</td>
<td>-----------------</td>
</tr>
</tbody>
</table>
| INSTR( string1, string2 ) | Returns the starting position of the first character in one string (string2) within another string (string1). If string2 is not found within string1, then this function returns zero (0). | - string1 – String to be evaluated  
- string2 – The (sub)string you are looking for within string1, for which the starting position is to be returned | Integer |
| LEN( string ) | Returns the length of a specified string, as a number of characters. | - string – The literal string or an expression that resolves to a string | Integer |
| LOADFLAGS( ) | Returns a comma-separated list of the following options applicable to an event:  
- A Position/Direction indicator: First, Next, Prev, or Last  
- An optional Distinct indicator: Distinct (If load is not flagged Distinct, this indicator is not present.)  
- An optional Read Mode indicator: ReadCommitted, or ReadUncommitted (If the load is flagged Default Read Mode, this indicator is not present.)  
Indicators appear in the order shown above, if present.  
Applies only to the IdoOnLoadCollection and IdoPostLoadCollection framework (Core) events | — | String |
| LOWER( string ) | Returns a string with uppercase characters converted to lowercase. | - string – String to be converted to lowercase | String |
| MESSAGE( msgID [, msgParam1, msgParam2, ... ] ) | Constructs and returns one or more messages using MsgAppSp. For more information, see the online Help for this function. | - msgID – Message identifier containing the object name  
- msgParam# – The parameters required to construct the message | String |
| METHOD ( ) | Returns the name of the method being invoked.  
This function is used only with the IdoOnInvoke and IdoPostInvoke framework events. | — | String |
<p>| METHODPARM( numExpr ) | Returns the value of a parameter for an IDO method call, at the specified index number. This function is used only with the IdoOnInvoke and IdoPostInvoke framework events. | - numExpr – A number or numeric expression that tells the system what index number to go to for the parameter value | Typeless |</p>
<table>
<thead>
<tr>
<th>Function / Syntax</th>
<th>What it does</th>
<th>Arguments</th>
<th>Return data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>METHODPARMS( )</td>
<td>Returns a count of the number of parameters required by the method. This function is used only with the IdoOnInvoke and IdoPostInvoke framework events.</td>
<td>—</td>
<td>Integer</td>
</tr>
<tr>
<td>NEWGUID( )</td>
<td>Generates a new globally unique identifier (GUID).</td>
<td>—</td>
<td>String</td>
</tr>
<tr>
<td>NONRESPONDERLIST ( action )</td>
<td>Returns a semicolon (;) delimited list of recipients who have not responded to a specified Prompt message.</td>
<td>action – Action sequence number</td>
<td>String</td>
</tr>
<tr>
<td>ORIGINATOR( )</td>
<td>Returns the user ID for the user who initiated the running of the event handler. This function can be useful when you need to identify who initiated the action that triggered the event or when you want the user who initiated the action to receive a copy of the message, for example.</td>
<td>—</td>
<td>String</td>
</tr>
<tr>
<td>P( [ [collection,] row#,] propertyName )</td>
<td>See PROPERTY.</td>
<td>—</td>
<td>Typeless</td>
</tr>
<tr>
<td>POSTQUERY-ACTIONS( )</td>
<td>Returns the list of post-query actions to be executed (server side) for every row returned by an event. Applies only to the IdoOnLoadCollection and IdoPostLoadCollection framework (Core) events.</td>
<td>—</td>
<td>String</td>
</tr>
<tr>
<td>POWER( numExpr, power )</td>
<td>Returns the value of a specified numeric expression to a designated power. <strong>NOTE:</strong> This function is equivalent to POWER in T-SQL.</td>
<td>numExpr – A number or an expression that resolves to a numeric value, power – the exponent, or power to which the numExpr is to be raised</td>
<td>Numeric</td>
</tr>
<tr>
<td>Function / Syntax</td>
<td>What it does</td>
<td>Arguments</td>
<td>Return data type</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------</td>
<td>-----------</td>
<td>-----------------</td>
</tr>
<tr>
<td>PROPERTY( [collection,] row#,] propertyName )</td>
<td>Returns the value of a specified property, optionally in a designated row and collection result set.</td>
<td></td>
<td>Typeless</td>
</tr>
<tr>
<td>PROPERTY-MODIFIED( propertyName )</td>
<td>Indicates whether a specified property has been modified or not. This function is used only with the IdoOnItem framework events.</td>
<td>propertyName – The name of the property for which you want to know whether it has been modified</td>
<td>Boolean</td>
</tr>
<tr>
<td>PROPERTY-NAMES( )</td>
<td>Returns a list of properties to be retrieved in the current operation.</td>
<td>—</td>
<td>String</td>
</tr>
<tr>
<td>RECIPIENTLIST( action )</td>
<td>Returns a semicolon (;) delimited list of recipients who received a message sent as the result of a specified action. The target action step must be a Notify or Prompt action type.</td>
<td>action – The action sequence number for the action that sent the message (Notify or Prompt)</td>
<td>String</td>
</tr>
<tr>
<td>RECIPIENTS( action )</td>
<td>Returns the number of recipients who received the message sent by a specified action. The target action step must be a Notify or Prompt action type.</td>
<td>action – The action sequence number for the action that sent the message (Notify or Prompt)</td>
<td>Integer</td>
</tr>
<tr>
<td>RECORDCAP( )</td>
<td>Returns the record cap in effect during an event. Applies only to the IdoOnLoadCollection and IdoPostLoadCollection framework (Core) events.</td>
<td>—</td>
<td>Integer</td>
</tr>
<tr>
<td>Function / Syntax</td>
<td>What it does</td>
<td>Arguments</td>
<td>Return data type</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------</td>
<td>-----------</td>
<td>------------------</td>
</tr>
</tbody>
</table>
| REPLACE( string1, string2, string3 ) | Within a string, replaces all instances of a specified substring with a different substring. | - `string1` – String in which the replacements are to be made  
- `string2` – Substring to be replaced  
- `string3` – Replacement substring  
Any of these strings can be referenced using an expression. | String |
| RESPONDERLIST( action [,choice] ) | Returns a semicolon (;) delimited list of recipients who have responded to a specified Prompt message, optionally with a selected choice. | - `action` – Action sequence number for the prompt action that produced the messages whose responses are to be checked.  
- `choice` – The particular choice for which you want to see who selected it. Optional.  
If omitted, the list includes all responders who selected any choice.  
The value for this argument must be a valid return value for the Prompt choices as defined on the Event Action Prompt and Event Action Prompt Choices forms. | String |
| RESPONDERS( action [,choice] ) | Indicates the number of recipients who have responded to a specified Prompt message, optionally for a selected choice. | - `action` – The action sequence number for the Prompt action  
- `choice` – The particular choice for which you want to see how many selected it. Optional.  
If omitted, the return value is a total of all who responded, regardless of their choices.  
The value for this argument must be a valid return value for the Prompt choices as defined on the Event Action Prompt and Event Action Prompt Choices forms. | Integer |
<table>
<thead>
<tr>
<th>Function / Syntax</th>
<th>What it does</th>
<th>Arguments</th>
<th>Return data type</th>
</tr>
</thead>
</table>
| ROUND( numExpr, length )  | Returns a numeric expression rounded to the specified length or precision. **NOTE:** This function is equivalent to ROUND in T-SQL.                                                                         | - numExpr – The numeric expression containing the value you want to have rounded  
- length – The number of places or the precision to which the number should be rounded.  
When length is a positive number, numExpr is rounded to the number of decimal positions specified by length.  
When length is a negative number, numExpr is rounded on the left side of the decimal point, as specified by length. | Numeric          |
| ROWS( collection )        | Returns a number indicating how many records (rows) are in a specified result set (as the result of a Load IDO Collection action).                                                                       | - collection – The name of the result set for which you want to know how many records have been loaded | Integer          |
| ROWS( )                   | Returns a number indicating how many rows were loaded by the collection that generated an event. Applies only to the IdoPostLoadCollection event.                                                            | —                                                                                                                                                   | Integer          |
| SUBSTITUTE( string, arg1[, arg2, …] ) | Creates a string combined of literal values and "replacement zones" for which values are substituted at runtime. Replacement zones are indicated by the use of curly braces { } enclosing numbers. Numbers must start with zero and be consecutive, although they need not necessarily appear in the string in the correct order.  
For more information, see the online Help topic for this function. | - string – The basic literal string containing the replacement zones  
- arg# – The sources for the replacement text to go into the replacement zones. These are typically expressions. These must appear in the order they are numbered within the string. That is, the first argument (arg1) is used for (0); the second argument (arg2) is used for (1), and so forth. | String           |
<table>
<thead>
<tr>
<th>Function / Syntax</th>
<th>What it does</th>
<th>Arguments</th>
<th>Return data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBSTRING( string, start, length )</td>
<td>Returns part of a designated string.</td>
<td>- string – An expression containing the string to be processed</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- start – A number indicating the starting position from which the substring is to be returned. The first character of a string is position 1. If start indicates a position beyond the length of string, the system returns an empty string.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- length – (Optional) A number indicating how many characters are to be returned. If length indicates more characters than are remaining in the string from the start location to the end of the string, or if this argument is not used, the system returns the entire remainder of the string from the start location.</td>
<td></td>
</tr>
<tr>
<td>SV( var )</td>
<td>Retrieves the value of a designated session variable.</td>
<td>var – The name of the session variable</td>
<td>Typeless</td>
</tr>
<tr>
<td>TRUNC( numExpr, length )</td>
<td>Returns a numeric value, rounded to the specified length or precision and then truncated to the appropriate number of places.</td>
<td>- numExpr – The numeric expression containing the value you want to have rounded and then truncated</td>
<td>Numeric</td>
</tr>
<tr>
<td></td>
<td>NOTE: This function is equivalent to ROUND in T-SQL, with a third argument (function) of 1.</td>
<td>- length – The number of places or the precision to which the number should be rounded and then truncated</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When length is a positive number, numExpr is rounded to the number of decimal positions specified by length. When length is a negative number, numExpr is rounded on the left side of the decimal point, as specified by length.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UPPER( string )</td>
<td>Returns a string with lowercase characters converted to uppercase.</td>
<td>string – the string expression to be converted to uppercase</td>
<td>String</td>
</tr>
<tr>
<td>USERDESC( )</td>
<td>Returns the user description, as entered in the User Description field of the Users form, for the user whose action initiated the event handler.</td>
<td>—</td>
<td>String</td>
</tr>
<tr>
<td>Function / Syntax</td>
<td>What it does</td>
<td>Arguments</td>
<td>Return data type</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>USERNAME( )</td>
<td>Returns the user logon ID for the system user either logging on or logging off the system.</td>
<td>—</td>
<td>String</td>
</tr>
<tr>
<td>V( var )</td>
<td>Retrieves the value of a specified event variable.</td>
<td>var – The name of the variable for which you want to retrieve the value</td>
<td>Typeless</td>
</tr>
<tr>
<td>VARIABLENAME( )</td>
<td>Returns the name of the session variable for which a requested change triggered the SessionOnVarChanged event.</td>
<td>—</td>
<td>String</td>
</tr>
<tr>
<td>VARIABLEVALUE( )</td>
<td>Returns the new value of the session variable for which a requested change triggered the SessionOnVarChanged event.</td>
<td>—</td>
<td>String</td>
</tr>
</tbody>
</table>
| VOTINGDISPARITY( action )      | Indicates whether any two selected choices disagree for a specified Prompt action. A return of:  
  - FALSE indicates that there is no disagreement.  
  - TRUE indicates that at least two responders' selections disagree.  
  For more information, see Dealing with Indeterminate Voting Results on page 74. | action – The action sequence (step) number for the Prompt action | Boolean          |
| VOTINGRESULT( action )         | Returns the winning choice resulting from the voting on a specified Prompt action. | action – The action sequence (step) number for the Prompt action | String           |
| VOTINGTIE( action )            | Indicates whether a specified Prompt action has resulted in a tie vote. A return of:  
  - FALSE indicates that it was not a tie vote.  
  - TRUE indicates that it was a tie vote. | action – The action sequence (step) number for the Prompt action | Boolean          |
| WORKINGDIR( )                  | Returns the working directory for the event handler that is running. For more detailed information, see the online Help topic for this function.  
  NOTE: This function is equivalent to the WORKINGDIR( ) method in the WinStudio API. | —                                             | String           |
Pre-parser Functions

The following additional functions can be used to expand an event variable or event global constant textually, that is, with no assumptions about the structure or data-type of the contained value. These functions are useful in cases where the expanded value contains other grammar elements that must be further evaluated, for example, to share common expressions, expression elements, or groups of functions.

<table>
<thead>
<tr>
<th>Function / Syntax</th>
<th>What it does</th>
<th>Arguments</th>
<th>Return data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>TGC( conName )</td>
<td>Retrieves the value of an event global constant for which the value can contain other grammar elements.</td>
<td>conName – The name of the global constant</td>
<td>Varies</td>
</tr>
<tr>
<td>TV( varName )</td>
<td>Retrieves the value of an event variable in which that value can contain other grammar elements.</td>
<td>varName – The name of the variable containing other elements</td>
<td>Varies</td>
</tr>
</tbody>
</table>

When an event action begins, the system first evaluates all TV( ) references recursively, until no more references to known variables remain. Next, the system evaluates all TGC( ) references recursively, until no more references to known global constants remain. Finally, the resulting parameters string is passed to the parser to evaluate all other functions and operators contextually using the grammar found in Appendix C, Expression Grammar.

For example, consider the following parameters string:

```
CONDITION(TGC(Over1WeekOld)) TGC(OldHandlerPromptParms)
```

Assume that the system is using the following event global constant metadata:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Over1WeekOld</td>
<td>DATEDIFF(day, BEGINDATE( ), CURDATETIME( )) &gt; 7</td>
</tr>
<tr>
<td>OldHandlerPromptParms</td>
<td>TO(USERNAME( )) SUBJECT(&quot;Old Handler Alert&quot;) QUESTION(&quot;Do you really want to continue this old Handler?&quot;) CHOICES(&quot;1,sYes,0,sNo&quot;)</td>
</tr>
</tbody>
</table>

The effective parameters passed to the event system parser would be:

```
CONDITION(DATEDIFF(day, BEGINDATE( ), CURDATETIME( )) > 7) TO(USERNAME( )) SUBJECT("Old Handler Alert") QUESTION("Do you really want to continue this old Handler?") CHOICES("1,sYes,0,sNo")
```
Complex Expansions

More complex expansions are possible. For example:

```
GC(MyVarTV(VarSuffix))
```

In this example, first, the value of the current event handler's "VarSuffix" variable is retrieved. Then that value is appended to the name "MyVar" to construct an event global constant name, for which the value is then retrieved.

The value of the "VarSuffix" variable might be set dynamically by an event action on the handler, or it might be included in different event initial states that are linked from referring handlers, in which case the GC( ) reference itself could be moved to its own global constant and referred to from all actions of the handlers using TGC( ).

Note that no operator or other syntax is used around the TV( ) reference, because it is evaluated and substituted in-place textually, without regard to data type or context.

Nested Expansions

Nested expansions are also possible. For example, consider the following parameters string:

```
TGC(MarkItUp)
```

Assume that the system is using the following event global constant metadata:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>StdMarkup</td>
<td>0.1 * 5</td>
</tr>
<tr>
<td>MarkItUp</td>
<td>CONDITION(V(Price)TGC(Std Markup) &lt; 100) SETVARVALUES(Price=Price TGC(Std Markup))</td>
</tr>
</tbody>
</table>

This has the effect of increasing the value of the "Price" variable by 10% plus 5, but only if the resulting value is less than 100.
Expression Operators

Expression operators can be either unary or binary, meaning they can operate on either one or two expressions. Most operators are limited as to what kind of expressions they can operate with. The kinds of expressions possible include:

- **Scalar expressions** (scalarExpr) – These expressions can be either of a known type (numeric, string, or date) or an unknown type (typeless).

  In the following table, if the expression type is given as scalarExpr, it can be any of these four types.

- **Numeric expressions** (numericExpr) – These expressions evaluate using numeric values. These are the expressions used to perform mathematic operations. If a string or typeless expression is supplied where a numeric expression is expected, it is automatically converted into a numeric value. If that is not possible due to the presence of non-numeric characters, the current handler fails with an error.

- **String expressions** (stringExpr) – These expressions are text-based sets of characters. They may include numbers, but if so, the numbers are treated as text characters, not numerals. If another type of expression is supplied where a string expression is expected, it is automatically converted into a string representation.

- **Date expressions** (dateExpr) – These expressions involve dates or parts of dates, including times such as 10:00AM.

- **Typeless expressions** (typelessExpr) – These are expressions that could be one of at least two different types. The way the system treats these expressions depends on the context in which the expression is used.

- **Boolean expressions** (BooleanExpr) – These expressions consist of an OR conjunction of one or more AND conjunctions.

To construct the expressions used in Event Action Parameters on page 140, you can use the expression operators in the following table in conjunction with the Expression Functions on page 155.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Purpose</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalarExpr = scalarExpr</td>
<td>Is equal to</td>
<td>V(var) = 1</td>
</tr>
<tr>
<td>scalarExpr != scalarExpr</td>
<td>Is not equal to</td>
<td>V(var) != 1</td>
</tr>
<tr>
<td>scalarExpr &lt;= scalarExpr</td>
<td>Is greater than or equal to</td>
<td>V(var) &gt;= 1</td>
</tr>
<tr>
<td>scalarExpr &lt; scalarExpr</td>
<td>Is greater than</td>
<td>V(var) &gt; 1</td>
</tr>
<tr>
<td>scalarExpr &lt;&lt;= scalarExpr</td>
<td>Is less than or equal to</td>
<td>V(var) &lt;= 1</td>
</tr>
<tr>
<td>scalarExpr : list</td>
<td>Is in the list.</td>
<td>V(var) IN (1;2;3;4)</td>
</tr>
<tr>
<td>scalarExpr IN list</td>
<td>The list is enclosed in parentheses and the elements separated by semi-colons.</td>
<td>&quot;b&quot; : (&quot;a&quot;;&quot;b&quot;;&quot;c&quot;)</td>
</tr>
<tr>
<td>scalarExpr !: list</td>
<td>Is not in the list.</td>
<td>V(var) !: (1;2;3;4)</td>
</tr>
<tr>
<td>BooleanExpr AND BooleanExpr</td>
<td>Boolean AND</td>
<td>V(var) = 1 AND V(var2) &gt; 5</td>
</tr>
</tbody>
</table>
For the complete grammar available for constructing expressions, see Expression Grammar on page 173.
Expression Grammar

The following is the complete grammar available for constructing expressions for Event Action Parameters. Major sections include:

- Restrictions
- Start Symbol
- Character Sets
- Terminals
- Rules
  - Variable, Constant, and Event Parameter References
  - Expressions
  - Boolean Rules
  - Typeless Rules
  - String Rules
  - Numeric Rules
  - Date Rules
  - Restricted Arguments
- Keyword Paren Lists
- Enumerations
Restrictions

Expressions in this grammar are subject to the following restrictions:

- Superfluous parentheses are usually not allowed.
  
  For example, the following produces a parsing error:
  
  \[
  V(\text{this})=1 \text{ and } 2=V(\text{that})) \text{ and } l=(2)
  \]
  
  Instead, you can write it as follows to work around the grammar limitation:
  
  \[
  V(\text{this})=1 \text{ and } 2=V(\text{that}) \text{ and } l=2
  \]

- Operation chains containing a mix of typed and typeless arguments must begin with a typed value.
  
  For example, the following example produces an error because the first argument is not a typed value:
  
  \[
  '12'=V(\text{a})+'B'+V(c)
  \]
  
  Instead, and because string concatenation is not commutative, you can write it as follows:
  
  \[
  '12'=''+V(\text{a})+'B'+V(c)
  \]
  
  Alternatively, you can declare the type for the first argument as follows:
  
  \[
  '12'=\text{CAST}(V(\text{a}) \text{ AS STRING})+'B'+V(c)
  \]
  
  Again, the following example produces an error because the first argument is not a typed value:
  
  \[
  12>V(\text{b})+1+5
  \]
  
  Instead, you can write the expression in one of the following ways:
  
  \[
  12>0+V(\text{b})+1+5
  \]
  
  \[
  12>\text{CAST}(V(\text{b}) \text{ AS NUMBER})+1+5
  \]
  
  \[
  12>1+V(\text{b})+5
  \]
  
  The last solution works because numeric addition is commutative.

- Functions cannot be used for variable, parameter, or constant names.
  
  For example, because the letter y is an abbreviation for year in the \text{DATEPART}() and \text{DATEDIFF}() event functions, the following expression produces an error:
  
  \[
  y = 5
  \]
  
  Instead, you can declare y as \text{yVar}. The following expression does not produce an error.
  
  \[
  \text{yVar} = 5
  \]
Start Symbol

"Start Symbol" = <functionParenList>

Character Sets

The following table lists and describes the acceptable characters for elements of the code described. You can include any amount of white space between elements.

<table>
<thead>
<tr>
<th>The following</th>
<th>Designates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>The set of numerals: 0123456789</td>
</tr>
<tr>
<td>Letter</td>
<td>The set of all uppercase and lowercase letters: abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ</td>
</tr>
<tr>
<td>Alphanumeric</td>
<td>The set of all characters listed as part of the Number and Letter set</td>
</tr>
<tr>
<td>Printable</td>
<td>The set of all standard characters that can be printed onscreen. This includes the characters from #32 to #127 and #160 (nonbreaking space). The nonbreaking space character is included because it is often used in source code.</td>
</tr>
<tr>
<td>Whitespace</td>
<td>The set of all characters that are normally considered &quot;whitespace&quot; and ignored by the parser. The set consists of:  ■ A space (regular)  ■ Horizontal tab  ■ Line feed  ■ Vertical tab  ■ Form feed  ■ Carriage return  ■ Nonbreaking space</td>
</tr>
</tbody>
</table>

{ID Head} = {Letter} + []

{ID Tail} = {Alphanumeric} + [] + ['['] + [']'] + []

{String Ch 1} = {Printable} - ['"] + {LF} + {CR}

{String Ch 2} = {Printable} - ['"] + {LF} + {CR}
Terminals

String constants are constructed using one of the following rules. The string can be:

- Enclosed in double-quotes, containing no double-quote characters as illustrated by the following example:
  \[ \text{StringLiteral} = \"\{\text{String Ch 1}\}\" \]
- Enclosed in double-quotes, containing paired double-quotes that are each interpreted as a single embedded double-quote character as illustrated by the following example:
  \[ \text{StringLiteral} = \"\"\{\text{String Ch 1}\}\"\" \]
- Enclosed in single-quotes, containing no single-quote characters as illustrated by the following example:
  \[ \text{StringLiteral} = '\{\text{String Ch 2}\}' \]
- Enclosed in single-quotes, containing paired single-quotes that are each interpreted as a single single-quote character as illustrated by the following example:
  \[ \text{StringLiteral} = '\\'\{\text{String Ch 2}\}\'\' \]

Each string constant in an expression or parameter list can be constructed using any of these rules, independently of other string constants.

Integer constants contain no decimal point:

\[ \text{IntegerLiteral} = \{\text{Digit}\}+ \]

Real constants contain a decimal point:

\[ \text{RealLiteral} = \{\text{Digit}\}.\{\text{Digit}\}+ \]

Identifiers (variable, constant, and event parameter names) must begin with a letter or underscore, and continue with zero or more alphanumeric characters and/or underscores:

\[ \text{Id} = \{\text{ID Head}\}\{\text{ID Tail}\}+ \]
Rules

Variable, Constant, and Event Parameter References

<IdValue> consists of one of the following:

V(Id)
GC(Id)
SV(Id)
E(Id)

<FilterIdValue> consists of one of the following:

FV (Id)
FGC (Id)
FSV (Id)
FE (Id)

Expressions

A scalar expression is either of a known type (Number, Date, or String), or its type is unknown.

<Scalar Exp> consists of one of the following:

<Numeric-castable Exp>
<Date Exp>
<String Exp>
<Typeless Exp>

A numeric-castable expression is either Number, String, or an unknown type.

<Numeric-castable Exp> consists of one of the following:

<Numeric Exp>

Boolean Rules

A Boolean expression is an OR-conjunction of one or more AND-conjunctions.

<Boolean Exp> consists of one of the following:

<And Exp>
<And Exp> OR <Boolean Exp>

An AND expression is an AND-conjunction of one or more negatable predicates.

<AND Exp> consists of one of the following:

<Not Exp>
<Not Exp> AND <And Exp>
A **NOT expression** is a negatable predicate.

<Not Exp> consists of one of the following:

- NOT <Predicate>
- <Predicate>

A **predicate** consists of one of the following:

- A comparison of like-typed expressions
- A call to a Boolean function
- A Boolean sub-expression enclosed in parentheses

<Predicate> consists of one of the following:

- <String Exp> LIKE <String Exp>
- <String Exp> IN <Scalar Tuple>
- <String Exp> : <Scalar Tuple>
- <String Exp> !: <Scalar Tuple>
- <String Exp> = <String Exp>
- <String Exp> <> <String Exp>
- <String Exp> != <String Exp>
- <String Exp> > <String Exp>
- <String Exp> >= <String Exp>
- <String Exp> < <String Exp>
- <String Exp> <= <String Exp>
- <String Exp> LIKE <Typeless Exp>
- <String Exp> = <Typeless Exp>
- <String Exp> <> <Typeless Exp>
- <String Exp> != <Typeless Exp>
- <String Exp> > <Typeless Exp>
- <String Exp> >= <Typeless Exp>
- <String Exp> < <Typeless Exp>
- <String Exp> <= <Typeless Exp>
- <Date Exp> IN <Scalar Tuple>
- <Date Exp> : <Scalar Tuple>
- <Date Exp> !: <Scalar Tuple>
- <Date Exp> = <Date Exp>
- <Date Exp> <> <Date Exp>
- <Date Exp> != <Date Exp>
- <Date Exp> > <Date Exp>
- <Date Exp> >= <Date Exp>
- <Date Exp> < <Date Exp>
- <Date Exp> <= <Date Exp>
<Date Exp> >= <Date Exp>
<Date Exp> < <Date Exp>
<Date Exp> <= <Date Exp>
<Date Exp> = <Typeless Exp>
<Date Exp> <> <Typeless Exp>
<Date Exp> != <Typeless Exp>
<Date Exp> > <Typeless Exp>
<Date Exp> >= <Typeless Exp>
<Date Exp> < <Typeless Exp>
<Date Exp> <= <Typeless Exp>
<Typeless Exp> = <Date Exp>
<Typeless Exp> <> <Date Exp>
<Typeless Exp> != <Date Exp>
<Typeless Exp> > <Date Exp>
<Typeless Exp> >= <Date Exp>
<Typeless Exp> < <Date Exp>
<Typeless Exp> <= <Date Exp>
<Numeric Exp> IN <Scalar Tuple>
<Numeric Exp> : <Scalar Tuple>
<Numeric Exp> !: <Scalar Tuple>
<Numeric Exp> = <Numeric Exp>
<Numeric Exp> <> <Numeric Exp>
<Numeric Exp> != <Numeric Exp>
<Numeric Exp> > <Numeric Exp>
<Numeric Exp> >= <Numeric Exp>
<Numeric Exp> < <Numeric Exp>
<Numeric Exp> <= <Numeric Exp>
<Typeless Exp> IN <Scalar Tuple>
<Typeless Exp> : <Scalar Tuple>
<Typeless Exp> !: <Scalar Tuple>
<Typeless Exp> = <Typeless Exp>
<Typeless Exp> <> <Typeless Exp>
<Typeless Exp> != <Typeless Exp>
<Typeless Exp> > <Typeless Exp>
<Typeless Exp> >= <Typeless Exp>
<Typeless Exp> < <Typeless Exp>
<Typeless Exp> <= <Typeless Exp>
<Typeless Exp> >= <Typeless Exp>
<Typeless Exp> < <Typeless Exp>
<Typeless Exp> <= <Typeless Exp>
TRUE
FALSE
ANYHANDLERSFAILED ( )
HANDLERSYNCHRONOUS ( )
HANDLERSUSPENDS ( )
HANDLERTRANSACTIONAL ( )
HANDLERIGNORESF AILURE ( )
VOTINGDISPARITY (<EventActionRef>)
VOTINGTIE (<EventActionRef>)
HASBEGUN (<EventActionRef>)
HASFINISHED (<EventActionRef>)
INSIDEDATABASE ( )
PROPERTYMODIFIED (<String Exp>)
PROPERTYMODIFIED (<Typeless Exp>)

Sub-expression:

(<Boolean Exp>)

Typeless Rules

A typeless expression is a concatenation or sum of elements whose type (between String, Numeric, and Date) we cannot distinguish without context.

<Typeless Exp> consists of one of the following:

<Typeless Exp> + <Typeless Value>
<Typeless Value>

A typeless value can be any of the following:

- A variable reference
- A function call whose type cannot be determined without context
- A typeless sub-expression enclosed in parentheses

<Typeless Value> consists of one of the following:

<IdValue>
IF (<Boolean Exp>, <Typeless Exp>, <Typeless Exp>)
DBFUNCTION (<Parameter List>)
Framework Event Parameters:

PROPERTY (Id, <Numeric Exp>, <String Exp>)
PROPERTY (Id, <Numeric Exp>, <Typeless Exp>)
PROPERTY (Id, <Typeless Exp>, <String Exp>)
PROPERTY (Id, <Typeless Exp>, <Typeless Exp>)
P (Id, <Numeric Exp>, <String Exp>)
P (Id, <Numeric Exp>, <Typeless Exp>)
P (Id, <Typeless Exp>, <String Exp>)
P (Id, <Typeless Exp>, <Typeless Exp>)
PROPERTY (<Numeric Exp>, <String Exp>)
PROPERTY (<Numeric Exp>, <Typeless Exp>)
PROPERTY (<Typeless Exp>, <String Exp>)
PROPERTY (<Typeless Exp>, <Typeless Exp>)
P (<Numeric Exp>, <String Exp>)
P (<Numeric Exp>, <Typeless Exp>)
P (<Typeless Exp>, <String Exp>)
P (<Typeless Exp>, <Typeless Exp>)
PROPERTY (<String Exp>)
PROPERTY (<Typeless Exp>)
P (<String Exp>)
P (<Typeless Exp>)
METHODPARM (<Numeric Exp>)
METHODPARM (<Typeless Exp>)

Sub-expression:

(<Typeless Exp>)

Parameter List

A parameter list is a comma-separated list of scalar expressions of any type.

<Parameter List> consists of one of the following:

<Scalar Exp>
<Scalar Exp>, <Parameter List>

Scalar Tuples

A scalar tuple is a parenthesized set of one or more scalar expressions.

<Scalar Tuple> consists of the following: (<Scalar Expr Set>)
Scalar Expression Sets

A scalar expression set is a semi-colon-separated list of one or more scalar expressions, as in the following.

<Scalar Expr Set> consists of one of the following:

- <Scalar Exp>; <Scalar Expr Set>
- <Scalar Exp>

String Rules

A string expression is a concatenation of expressions of type String or of unknown type (see the second restriction under Restrictions on page 174).

<String Exp> consists of one of the following:

- <String Value>
- <String Exp> + <String Value>
- <String Exp> + <Typeless Value>

A string value can be any of the following:

- A string literal
- A quoted variable reference
- A function call returning a string value
- A string sub-expression enclosed in parentheses

<String Value> consists of one of the following:

- StringLiteral
- <FilterIdValue>
- IF (<Boolean Exp>, <String Exp>, <String Exp>)
- IF (<Boolean Exp>, <String Exp>, <Typeless Exp>)
- IF (<Boolean Exp>, <Typeless Exp>, <String Exp>)
String Built-in Functions:

CLIENTSUBSTITUTE (<String Exp>, <String Expr List>)
CLIENTSUBSTITUTE (<Typeless Exp>, <String Expr List>)
SUBSTITUTE (<String Exp>, <String Expr List>)
SUBSTITUTE (<Typeless Exp>, <String Expr List>)
SUBSTRING (<String Exp>, <Numeric Exp>)
SUBSTRING (<String Exp>, <Typeless Exp>)
SUBSTRING (<String Exp>, <Numeric Exp>, <Numeric Exp>)
SUBSTRING (<String Exp>, <Numeric Exp>, <Typeless Exp>)
SUBSTRING (<String Exp>, <Typeless Exp>, <Numeric Exp>)
SUBSTRING (<String Exp>, <Typeless Exp>, <Typeless Exp>)
SUBSTRING (<Typeless Exp>, <Numeric Exp>)
SUBSTRING (<Typeless Exp>, <Typeless Exp>)
SUBSTRING (<Typeless Exp>, <Numeric Exp>, <Numeric Exp>)
SUBSTRING (<Typeless Exp>, <Numeric Exp>, <Typeless Exp>)
SUBSTRING (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>)
SUBSTRING (<Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
UPPER (<String Exp>)
UPPER (<Typeless Exp>)
LOWER (<String Exp>)
LOWER (<Typeless Exp>)
REPLACE (<String Exp>, <String Exp>, <String Exp>)
REPLACE (<String Exp>, <String Exp>, <Typeless Exp>)
REPLACE (<String Exp>, <Typeless Exp>, <String Exp>)
REPLACE (<Typeless Exp>, <String Exp>, <String Exp>)
REPLACE (<Typeless Exp>, <String Exp>, <Typeless Exp>)
REPLACE (<Typeless Exp>, <String Exp>, <Typeless Exp>)
REPLACE (<Typeless Exp>, <Typeless Exp>, <Typeless Exp>)

Event Attributes:

EVENTNAME()
ORIGINATOR()
CONFIGNAME()
EVENTSTATE()
EVENTTITLE()
ACTIONTYPENAME()
VOTINGRESULT (<EventActionRef>)
RECIPIENTLIST (<EventActionRef>)
RESPONDERLIST (<EventActionRef>)
RESPONDERLIST (<EventActionRef>, <String Exp>)
RESPONDERLIST (<EventActionRef>, <Typeless Exp>)
Environment Attributes:

APPNAME()
COMPANYNAME()
USERNAME()
USERDESC()
WORKINGDIR()
FILECONTENTS (<String Exp>)
FILECONTENTS (<Typeless Exp>)
NEWGUID()

Framework Event Parameters:

IDO()
INITIATOR()
FILTERSTRING()
LOADFLAGS()
PROPERTYNAMES()
POSTQUERYACTIONS()
CUSTOMINSERT()
CUSTOMUPDATE()
CUSTOMDELETE()
VARIABLENAME()
VARIABLEVALUE()
FILTERPROPERTY (Id, <Numeric Exp>, <String Exp>)
FILTERPROPERTY (Id, <Numeric Exp>, <Typeless Exp>)
FILTERPROPERTY (Id, <Typeless Exp>, <String Exp>)
FILTERPROPERTY (Id, <Typeless Exp>, <Typeless Exp>)
FP (Id, <Numeric Exp>, <String Exp>)
FP (Id, <Numeric Exp>, <Typeless Exp>)
FP (Id, <Typeless Exp>, <String Exp>)
FP (Id, <Typeless Exp>, <Typeless Exp>)
FILTERPROPERTY (<Numeric Exp>, <String Exp>)
FILTERPROPERTY (<Numeric Exp>, <Typeless Exp>)
FILTERPROPERTY (<Typeless Exp>, <String Exp>)
FILTERPROPERTY (<Typeless Exp>, <Typeless Exp>)
FP (<Numeric Exp>, <String Exp>)
FP (<Numeric Exp>, <Typeless Exp>)
FP (<Typeless Exp>, <String Exp>)
FP (<Typeless Exp>, <Typeless Exp>)
FILTERPROPERTY (<String Exp>)
FILTERPROPERTY (<Typeless Exp>)
FP (<String Exp>)
FP (<Typeless Exp>)
METHOD ( )
FILTERMETHODPARAM (<Numeric Exp>)
FILTERMETHODPARAM (<Typeless Exp>)
FILTER (<String Exp>)
FILTER (<Typeless Exp>)
CAST (<Scalar Exp> AS STRING)

Sub-expression:

(<String Exp>)
<Message Conversation>

String Expression Lists

A string expression list is an ordered, comma-separated list of string and/or scalar expressions.

<String Expr List> consists of one of the following:

<Scalar Exp>
<String Expr List>, <Scalar Exp>

Message Conversations

A message conversation is one or more MESSAGE() calls separated by newline/concatenation operators.

<Message Conversation> consists of one of the following:

<Message>
<Message Conversation> | <Message>

Messages

A message is the MESSAGE function surrounding a parenthesized, ordered, comma-separated list of string and/or typeless expressions.

<Message> consists of one of the following:

MESSAGE (<String Exp>)
MESSAGE (<Typeless Exp>)
MESSAGE (<String Exp>, <String Expr List>)
MESSAGE (<Typeless Exp>, <String Expr List>)

Numeric Rules

<Numeric Exp> consists of the following: <Sum>
Sums

A sum is a chain of sums and/or differences:

<Sum> consists of one of the following:

<Addend>
<Sum> + <Addend>
<Sum> - <Addend>

Addends

An addend is a negatable expression, or a product.

These can be added to or subtracted from one another without parentheses surrounding each, due to the accepted arithmetic operator precedence.

<Addend> consists of one of the following:

<Negate Exp>
<Addend> * <Negate Exp>
<Addend> / <Negate Exp>

Negatable Expressions

A negatable expression is a numeric value with or without a unary negative.

<Negate Exp> consists of one of the following:

- <Numeric Value>
  <Numeric Value>
  - <IdValue>

Numeric Values

A numeric value can be any of the following:

- A numeric constant
- A function call returning a numeric value
- A numeric sub-expression enclosed in parentheses

<Numeric Value> consists of one of the following:

  IntegerLiteral
  RealLiteral
  IF (<Boolean Exp>, <Numeric Exp>, <Numeric Exp>)
  IF (<Boolean Exp>, <Numeric Exp>, <Typeless Exp>)
  IF (<Boolean Exp>, <Typeless Exp>, <Numeric Exp>)
Numeric Built-in Functions:

DATEDIFF (<TimeInterval>, <Date Exp>, <Date Exp>)
DATEDIFF (<TimeInterval>, <Date Exp>, <Typeless Exp>)
DATEDIFF (<TimeInterval>, <Typeless Exp>, <Date Exp>)
DATEDIFF (<TimeInterval>, <Typeless Exp>, <Typeless Exp>)
DATEPART (<TimeInterval>, <Date Exp>)
DATEPART (<TimeInterval>, <Typeless Exp>)
LEN (<String Exp>)
LEN (<Typeless Exp>)
INSTR (<String Exp>, <String Exp>)
INSTR (<String Exp>, <Typeless Exp>)
INSTR (<Typeless Exp>, <String Exp>)
INSTR (<Typeless Exp>, <Typeless Exp>)
CEILING (<Numeric Exp>)
CEILING (<Typeless Exp>)
FLOOR (<Numeric Exp>)
FLOOR (<Typeless Exp>)
POWER (<Numeric Exp>, <Numeric Exp>)
POWER (<Numeric Exp>, <Typeless Exp>)
POWER (<Typeless Exp>, <Numeric Exp>)
POWER (<Typeless Exp>, <Typeless Exp>)
ROUND (<Numeric Exp>, <Numeric Exp>)
ROUND (<Numeric Exp>, <Typeless Exp>)
ROUND (<Typeless Exp>, <Numeric Exp>)
ROUND (<Typeless Exp>, <Typeless Exp>)
TRUNC (<Numeric Exp>, <Numeric Exp>)
TRUNC (<Numeric Exp>, <Typeless Exp>)
TRUNC (<Typeless Exp>, <Numeric Exp>)
TRUNC (<Typeless Exp>, <Typeless Exp>)

Event State Attributes:

EVENTREVISION ( )
HANDLERSEQ ( )
ACTIONSEQ ( )

Event Attributes:

RECIPIENTS (<EventActionRef>)
RESPONDERS (<EventActionRef>)
RESPONDERS (<EventActionRef>, <String Exp>)
RESPONDERS (<EventActionRef>, <Typeless Exp>)
Framework Event Parameters:

RECORDCAP()
ROWS (Id) ROWS ( )
METHODPARMS ( )
CAST (<Scalar Exp> AS NUMBER)

Sub-expression:

(<Numeric Exp>)

Date Rules

A date expression is a date value (because there are no natural arithmetic date operators).

<Date Exp> consists of the following: <Date Value>

A date value is a function call returning a date value, or a date sub-expression enclosed in parentheses.

<Date Value> consists of one of the following:

DATE (<Numeric Exp>, <Numeric Exp>, <Numeric Exp>)
DATE (<Numeric Exp>, <Numeric Exp>, <Typeless Exp>)
DATE (<Numeric Exp>, <Typeless Exp>, <Numeric Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Numeric Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)}
DATE (<Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
<table>
<thead>
<tr>
<th>DATE</th>
<th>DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
</tbody>
</table>
DATE (<Typeless Exp>, <Numeric Exp>, <Numeric Exp>, <Numeric Exp>,
<Typeless Exp>, <Numeric Exp>, <Numeric Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Numeric Exp>, <Numeric Exp>,
<Typeless Exp>, <Numeric Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Numeric Exp>, <Numeric Exp>,
<Typeless Exp>, <Typeless Exp>, <Numeric Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Numeric Exp>, <Numeric Exp>,
<Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Numeric Exp>, <Typeless Exp>,
<Numeric Exp>, <Numeric Exp>, <Numeric Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Numeric Exp>, <Typeless Exp>,
<Numeric Exp>, <Numeric Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Numeric Exp>, <Typeless Exp>,
<Numeric Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Numeric Exp>,
<Numeric Exp>, <Numeric Exp>, <Numeric Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Numeric Exp>,
<Numeric Exp>, <Numeric Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Numeric Exp>,
<Numeric Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>,
<Numeric Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>,
<Typeless Exp>, <Numeric Exp>, <Numeric Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>,
<Typeless Exp>, <Numeric Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>,
<Typeless Exp>, <Typeless Exp>, <Numeric Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>,
<Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>,
<Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>,
<Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>,
<Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>,
<Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>,
<Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>,
<Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>,
<Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>,
<Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>, <Numeric Exp>, <Numeric Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Numeric Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Numeric Exp>, <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Typeless Exp> ',' <Numeric Exp>,
    <Typeless Exp>, <Typeless Exp>, <Numeric Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Typeless Exp> ',' <Numeric Exp>,
    <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Typeless Exp> ',' <Typeless Exp>,
    <Numeric Exp>, <Numeric Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Typeless Exp> ',' <Typeless Exp>,
    <Numeric Exp>, <Numeric Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Typeless Exp> ',' <Typeless Exp>,
    <Typeless Exp>, <Numeric Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Typeless Exp> ',' <Typeless Exp>,
    <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Typeless Exp> ',' <Typeless Exp>,
    <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Typeless Exp> ',' <Typeless Exp>,
    <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Typeless Exp> ',' <Typeless Exp>,
    <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Typeless Exp> ',' <Typeless Exp>,
    <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Typeless Exp> ',' <Typeless Exp>,
    <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
DATE (<Typeless Exp>, <Typeless Exp>, <Typeless Exp> ',' <Typeless Exp>,
    <Typeless Exp>, <Typeless Exp>, <Typeless Exp>)
CURDATETIME()
DATEADD (<TimeInterval>, <Numeric Exp>, <Date Exp>)
DATEADD (<TimeInterval>, <Numeric Exp>, <Typeless Exp>)
DATEADD (<TimeInterval>, <Typeless Exp>, <Date Exp>)
DATEADD (<TimeInterval>, <Typeless Exp>, <Typeless Exp>)
BEGINDATE()
CAST (<Scalar Exp> AS DATE)

**Sub-expression:**

(<Date Exp>)

**Date Enumerations**

<TimeInterval> consists of one of the following:

<DatePartYear>
<DatePartQuarter>
<DatePartMonth>
<DatePartDayOfYear>
<DatePartDay>
<DatePartWeek>
<DatePartWeekDay>
<DatePartHour>
<DatePartMinute>
<DatePartSecond>
<DatePartMillisecond>
<DatePartYear> consists of one of the following:
   year
   yy
   yyyy

<DatePartQuarter> consists of one of the following:
   quarter
   qq
   q

<DatePartMonth> consists of one of the following:
   month
   mm
   m

<DatePartDayOfYear> consists of one of the following:
   dayofyear
   dy
   y

<DatePartDay> consists of one of the following:
   day
   dd
   d

<DatePartWeek> consists of one of the following:
   week
   wk
   ww

<DatePartWeekDay> consists of one of the following:
   weekday
   dw

<DatePartHour> consists of one of the following:
   hour
   hh
<DatePartMinute> consists of one of the following:
  minute
  mi
  n

<DatePartSecond> consists of one of the following:
  second
  ss
  s

<DatePartMillisecond> consists of one of the following:
  millisecond
  ms

Restricted Arguments

<EventActionRef> consists of the following: IntegerLiteral

Keyword Paren Lists

A keyword paren list is one or more "KEYWORD(value)" statements separated by whitespace.

<KeywordParenList> consists of one of the following:

  <KEYWORD(value)>
  <KEYWORD(value)> ... <KEYWORD(value)>

For a complete list and description of KEYWORD(value) statements as they relate to action types, see Event Action Parameters on page 140
Enumerations

The following lists of keywords can be used with ACTION( ), VOTINGRULE( ), and TASKSTATUS( ) expressions.

**ACTION( )**

`<SaveAction>` consists of one of the following:

- `<SaveActionInsert>`
- `<SaveActionUpdate>`
- `<SaveActionDelete>`

`<SaveActionInsert>` consists of the following: INSERT

`<SaveActionUpdate>` consists of the following: UPDATE

`<SaveActionDelete>` consists of the following: DELETE

**VOTINGRULE( )**

`<VotingRule>` consists of one of the following:

- `<VotingRuleMajority>`
- `<VotingRulePlurality>`
- `<VotingRuleConditionalPlurality>`
- `<VotingRuleMinimumCount>`
- `<VotingRuleMinimumPercentage>`
- `<VotingRuleEarliestResponse>`
- `<VotingRulePreferredChoice>`
- `<VotingRuleMinimumCountPreferredChoice>`
- `<VotingRuleMinimumPercentagePreferredChoice>`

`<VotingRuleMajority>` consists of the following: Majority

`<VotingRulePlurality>` consists of the following: Plurality

`<VotingRuleConditionalPlurality>` consists of the following: ConditionalPlurality

`<VotingRuleMinimumCount>` consists of the following: MinimumCount

`<VotingRuleMinimumPercentage>` consists of the following: MinimumPercentage

`<VotingRuleEarliestResponse>` consists of the following: EarliestResponse

`<VotingRulePreferredChoice>` consists of the following: PreferredChoice

`<VotingRuleMinimumCountPreferredChoice>` consists of the following: MinimumCountPreferredChoice

`<VotingRuleMinimumPercentagePreferredChoice>` consists of the following: MinimumPercentagePreferredChoice
**TASKSTATUS()**

<InitialTaskStatus> consists of one of the following:

<TaskStatusReady>
<TaskStatusWaiting>

<TaskStatusReady> consists of the following: READY

<TaskStatusWaiting> consists of one of the following: WAITING
The following sample stored procedures provide sample code and function usage.

### Passing Parameters to a Synchronous Event

```
SET @MyEventParmId = NEWID()
EXEC InsertEventInputParameterSp @MyEventParmId, 'Parameter1', @Variable1
EXEC InsertEventInputParameterSp @MyEventParmId, 'Parameter2', @Variable2
EXEC InsertEventInputParameterSp @MyEventParmId, 'OutParameter3', NULL, 1
EXEC FireEventSp 'EventName', @MyEventParmId
PRINT dbo.EventOutputParameterValue(@MyEventParmId, 'OutParameter3')
```

### Calling a Synchronous Event within a Transaction and Handling Failure

First, determine the current site, after which you must name a configuration, by convention:

```sql
DECLARE @Site SiteType
SELECT @Site = site FROM parms
```

Then determine the current SessionId:

```sql
DECLARE @SessionId RowPointerType
SET @SessionId = dbo.SessionIdSp()
```

Finally, add the procedure code:

```sql
BEGIN TRANSACTION
UPDATE coitem
    SET due_date = dbo.CalcDueDate(@Parm1, @Parm2)
WHERE coitem.co_num = @CoNum
    AND coitem.co_line = @CoLine
    AND coitem.co_release = @CoRelease

SET @MyEventParmId = NEWID()
EXEC InsertEventInputParameterSp @MyEventParmId, 'CoNum', @CoNum
EXEC InsertEventInputParameterSp @MyEventParmId, 'CoLine', @CoLine
EXEC InsertEventInputParameterSp @MyEventParmId, 'CoRelease', @CoRelease
```
DECLARE
    @anyHandlersFailed [tinyint],
    @result [nvarchar](4000),
    @Infobar [nvarchar](4000)
EXEC @Severity = FireEventSp
    @eventName = 'SetCoitemDueDate',
    @configName = 'SyteLine',
    @sessionID = @SessionID,
    @eventTrxId = null,
    @eventParmId = @MyEventParmID OUTPUT,
    @transactional = 0,
    @anyHandlersFailed = @anyHandlersFailed output,
    @result = @result output,
    @Infobar = @infobar output
IF @Severity > 0
BEGIN
    EXEC RaiseError @Infobar, @Severity
    ROLLBACK TRANSACTION
END
...
COMMIT TRANSACTION
This appendix discusses the various concepts and strategies for synchronizing metadata belonging to different owners, and provides a number of detailed examples.

Overview and Rationale

Application event system metadata can come from three primary sources:

- The Infor system framework
- Applications produced by Infor and/or Infor’s business partners or other authorized vendor developers
- End-customer development

The ownership for the metadata from each of these sources is controlled by the Access As identifier. For more information on this identifier, see About the Access As Identifier on page 39.

Any of these sources can upgrade and reissue their metadata at times independent of the others. The upgrade process needs to do the following:

- Update any changed handlers that they own.
- Insert any new handlers they might have created since the last version.
- Maintain other owners’ handlers and the relationships between them.

Therefore, a synchronization mechanism is needed, to make sure that changes by one metadata owner do not adversely affect the functioning of another owner’s metadata.

This mechanism is provided in two components:

- The Access As identifier (see About the Access As Identifier on page 39)
- The App Metadata Sync and App Metadata Transport utilities, which both provide the capability to synchronize event metadata belonging to different owners.

Using these utilities, metadata developers can export their events and event handlers and make them available for import by other metadata owners. They also use this utility to import their own or others’ metadata into their system.

For more information about these utilities, see the online Help for each utility. This help is available in SyteLine by selecting Help > Customizing Forms and then performing a search for "App Metadata Sync" or "App Metadata Transport".
The Inherent Hierarchy of Metadata

In the application event system, there is no programmatic ranking of Access As identifiers. There is, however, an inherent hierarchy, based on the normal production flow and use of the system software. As illustrated in the following diagram, Infor’s framework developers are the first to develop event system objects. Other Infor application developers then can add their own event system objects, as can authorized business partners and other vendor developers. Finally, end-customers can make custom modifications and develop their own custom event system objects.

In reality, the system is designed so that no metadata owner can modify or delete the metadata of another owner. So, in that sense, they are all equal. However, this real-time production flow creates an inherent hierarchy that allows us to think of them as "higher-level" (that is, Infor) and "lower-level" (that is, end-customer) owners.

With that in mind, we can state the following general rules:

- Lower-level owners can insert their handlers between two higher-level handlers for the same event.
- In many cases, lower-level owners can override higher-level handlers.
- Higher-level handlers that are overridden remain in the metadata store but are marked as Inactive.

This means, among other things that, if the lower-level handler is later deleted, the higher-level handler is still available and can become active again.
Chronology rules allow downstream owners to integrate and control the sequence of their events and handlers with respect to those upstream. For more information, see Detailed Examples on page 204.

Maintaining Handler IDs Through Metadata Updates

Each event handler is identified with a unique (and hidden) ID, which is referenced by the Keep With field on the Event Handlers form. This ID, rather than the actual Handler Sequence number, becomes the "fixed" reference point for that handler. This means that an event handler owner does not need to worry about maintaining the Handler Sequence numbers across releases: The system takes care of it automatically, by preserving the hidden ID number and reassigning Handler Sequence numbers as required.

After each insertion, update, or deletion of a handler, and during a merge performed by the App Metadata Sync or App Metadata Transport utility, the system calculates new integers, if necessary, for display in the Handler Sequence field. The underlying ID, however, remains unchanged. When a handler is deactivated and another added in the same position, the new handler gets a new ID.

Protecting Running Events from Metadata Changes

Once an event handler begins executing, it is essential to prevent changes to its attributes and actions. Otherwise, unpredictable behavior could result, especially if actions are resequenced.

One way to do prevent these changes would be to make a copy of all active, non-obsolete handlers and their actions each time an event is triggered and control execution from this copy. However, that method would result in the persistence of a great number of identical copies, assuming that handlers are modified much less frequently than they are executed.

Since the system stores state data separately from metadata, it is sufficient to make a copy only when the metadata changes, and furthermore only when the corresponding event is triggered and a copy of the last metadata modifications has not yet been made.

In other words, handler metadata for an event can be created and edited as many times as necessary. The first time the event is triggered, a copy of the last saved metadata is made. This copy is called an event revision. The execution of the event's handlers is then controlled by the event revision and not by the original metadata (which happen to be identical at this point).

The event can be triggered as many times as necessary, all the time controlled by this event revision, as long as no intervening modifications have been made to the original metadata.

After one or more metadata edits have been saved, though, the next time the event is triggered, the system copies a new event revision from the last-saved metadata.

For more information about event revisions, see Event and Event Handler Revisions on page 36.
Detailed Examples

This section provides three detailed examples of how sequencing and synchronization work in the application event system.

Using Specific Chronology

The primary way for a lower-level handler creator to resequence existing handlers (from higher-level owners) is to use what is known as specific chronology. That is, the handler’s creator can attach the new handler to an existing handler and specify the order in which the two handlers are to execute with respect to each other.

The mechanism used to do this are the Keep With and Chronology fields on the Event Handlers form. These fields allow you to specify whether your handler should run before, after, or in place of the handler it is associated with, as in the following example.

For more information about the Keep With and Chronology fields, see the online Help for those fields.

Infor Creates a Framework Event with Three Handlers

The Infor framework team creates an event, FrameEvent, and creates three event handlers that execute in order when the event is generated.

![Diagram of FrameEvent event with three handlers: S1, S2, S3]

This event and these handlers are all included with the application software when it ships.

Note that, if a "lower-level" developer wants to add or modify handlers, the following rules apply:

- Lower-level handler creators cannot change the sequence of higher-level handlers.
- Any handlers lower-level creators want to use that will affect the sequence in specific ways must be associated with a particular, already-existing handler, using the **Keep With** and **Chronology** fields.

### A Business Partner Creates an Additional Handler

An Infor business partner decides to add a handler that will execute just after the first framework handler. The business partner creates the event handler and uses the **Keep With** and **Chronology** fields to keep their handler with ID (S1) and to execute **After** that handler.

The sequence is now as follows:

![Diagram of event sequence](image)

Note that, when the new event handler is saved, the system automatically assigns new **Handler Sequence** numbers, so that you can tell which handler executes in what order.

The business partner uses the **App Metadata Sync** or **App Metadata Transport** utility to export the new metadata, along with the existing metadata, to a file that can be imported by the end-customer.

The business partner then sells the add-on product to the end-customer. The add-on product includes this file, along with any product code the business partner has developed.
The End-customer Creates Three More Handlers

When the end-customer receives the add-on software from the third-party business partner, in addition to whatever other software installation procedures the customer must perform, the customer must also use the **App Metadata Sync** or **App Metadata Transport** utility to import the event metadata from the business partner.

The end-customer now decides to use a different handler in place of the second framework handler. The customer also wants two custom handlers to run just before the third framework handler. To accomplish this, the customer:

- Creates a handler and uses the **Keep With** field to assign this handler to ID 2 (S2). In the **Chronology** field, the customer selects **Instead**.
- Creates a second handler and uses the **Keep With** field to assign this handler to ID 3 (S3). In the **Chronology** field, the customer selects **Before**.
- Creates a third handler and uses the **Keep With** field to also assign this handler to ID 3 (S3). In the **Chronology** field, again the customer selects **Before**.
After saving their new handlers with the existing handlers, the sequence is now as follows:

Again, note that, when the customer saves the new handlers, the system automatically assigns and displays the correct **Handler Sequence** number on the **Event Handlers** form. The underlying Handler ID, however, remains unchanged.

Note that event handler S2 is now inactive and does not execute at all. It is still in the system, but the system ignores it in favor of the new customer handler.

Note also that the custom handlers C2 and C3 execute in that order unless you use the **Up / Down** buttons on the **Event Handler Sequence** form to alter the default order.

**Using Non-specific Chronology**

A second way for downstream developers to affect the sequence of handlers is with the use of non-specific chronology. Non-specific chronology allows developers to indicate that a particular handler should always execute either first or last.
This is done by selecting either the **First** or **Last** option from the **Chronology** drop-down field, as in the following example.

**Infor Creates an Event with No Handler**

The Infor development team adds code to generate a new application event (AppEvent) before a certain transaction posting is performed.

The transaction data is passed into the event and received from the event, so that any downstream subscribers can test and modify the values before the posting is performed if they want. An event failure is trapped and aborts the posting, displaying the error message returned by the event. Infor adds no event handlers for the event at this time.

The product is then released. The event is now published and available to downstream developers who install this product.
An End-customer Creates a Handler

An end-customer installs the software and decides to create a handler (EC1) to validate that the posting data is within a certain range. If it detects an out-of-range condition, the handler fails the event and aborts the posting.

At this point, the sequence is as follows:

Because no upstream handlers exist yet (and the end-customer is the furthest downstream developer), a specific chronology cannot be entered at this time.
A Business Partner Creates an Add-on Product

In the meantime, an Infor business partner creates an add-on product that stores custom parameters which can be used to adjust the transaction data before posting. They add their own handler (BP1) to input, adjust, and output the data.

In the Chronology field, they select First to indicate that handler BP1 should run before:

- Any existing, downstream handlers with no specified chronology that may already exist for this published event.

  **NOTE:** A specific chronology (Before, After, Instead) takes precedence over a non-specific one (First and Last).

- Any future upstream or peer handlers

With this add-on product, at the business partner level, the sequence is now as follows:
The End-customer Installs the Business Partner's Add-on Product

Now the end-customer purchases and installs the add-on from the Infor business partner and uses the **App Metadata Sync** or **App Metadata Transport** utility to synchronize the metadata from the business partner with their own.

Because the business partner specified that their handler should always run first, the sequence is now as follows:

![Diagram of synchronization sequence]

**Key**
- Infor
- Business Partner
- End Customer
- Inactive/Unmodified

Exit event
The End-customer Rearranges the Sequence

After installing the business partner’s add-on product, the end-customer decides that their handler really should run before the business partner’s. So the end-customer uses the **Keep With** and **Chronology** fields to associate handler EC1 with the business partner’s handler (BP1) and to execute **Before** it.

After saving the event handler EC1, the sequence is as follows:

```
AppEvent

Transaction Data

Handler ID = EC1

Transaction Data

Handler ID = BP1

Transaction Data

Exit event
```

This is possible because a specific chronology designation (see Using Specific Chronology on page 204) takes precedence over a non-specific chronology.
The End-customer Decides to Add Another Handler

After this, the end-customer adds a new handler (EC2) to save the final, adjusted transaction data into a data warehouse. They use the *Keep With* and *Chronology* fields to associate the new handler with the business partner’s handler, specifying that the new one execute *After* the business partner’s (BP1).

After saving the new event handler, the sequence is as follows:
Another Business Partner Adds a Handler

Independently of all this, a second Infor business partner installs the standard Infor software and decides to add another event handler for their add-on product. This event handler (AV1) is to post additional information to the journal based on the transaction data that was entered.

Because this handler should run after all transaction data has been processed, the business partner specifies in the Chronology field that event handler AV1 should execute Last.

At this point, the flow looks similar to the first business partner’s add-on flow:

However, the effect of designating this handler to run Last is quite different.
The End-customer Adds the Second Business Partner's Handler

Finally, the end-customer purchases and installs the second business partner’s add-on product. After using the **App Metadata Sync** or **App Metadata Transport** utility to synchronize the new metadata with the existing metadata, the flow now proceeds as follows:

![Diagram of the event flow]

Note that, even without the **Last** designation, in this case, handler AV1 would have executed last simply because the handler BP1 is designated to execute **First**, and the other two handlers are both "attached" to BP1.
Performing Upgrades

It is likely that at some point in the future, Infor will make changes to the existing events and handlers, and these changes will be included in an upgrade. This poses a few problems and questions. The upgrade process must:

- Update any changed Infor handlers and insert new ones while maintaining custom handlers belonging to business partners and end-customers.
- Keep custom handlers in the correct sequence with Infor’s handlers (especially for overrides) beyond the point where Infor inserts a new handler.

To illustrate how this works, we return to the previous examples (in Using Specific Chronology on page 204 and Using Non-specific Chronology on page 207).
Continuing the Specific Chronology Example...

After the end-customer has installed the Infor software and the business partner add-on product and added its own custom handlers, Infor introduces a new version in which one of the original base handlers (S2) has been modified and a new one, S4, has been added between S2 and S3. The new base flow in the upgrade is illustrated in the following diagram:

Note that, when this version is released, the numbers actually displayed in the **Handler Sequence** field of the **Event Handlers** form are as follows:

<table>
<thead>
<tr>
<th>Handler ID</th>
<th>Handler Sequence number</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>1</td>
</tr>
<tr>
<td>S2</td>
<td>2</td>
</tr>
<tr>
<td>S3</td>
<td>4</td>
</tr>
<tr>
<td>S4</td>
<td>3</td>
</tr>
</tbody>
</table>

Note that what was previously the third event handler in the flow (S3), now appears as **4** in the **Handler Sequence** field. This is because, when the Infor developer saves the new handler, the system automatically assigns the correct **Handler Sequence** number,
without changing the underlying Handler ID number.

The end-customer now installs the new Infor upgrade and uses the **App Metadata Sync** or **App Metadata Transport** utility to import and synchronize the updated metadata for this event. After the integration is complete, the new end-customer sequence is as follows:
Note that:

- Infor handler S2 has been updated. In this case, the update does not run, because of the end-customer handler C1, which replaces it. But the changes are there (stored as handler metadata) and available if the end-customer should ever decide to delete handler C1.

- The synchronization process inserts the new Infor handler between end-customer handlers C1 and C2. This is because end-customer handler C1 is "attached" to Infor handler S2; end-customer handlers C2 and C3 are "attached" to Infor handler S3; and the new Infor handler has been inserted between S2 and S3.

Continuing the Non-specific Chronology Example…

(Starting at the end of Using Non-specific Chronology:) After the end-customer has installed the Infor software and both business partner add-on products, Infor introduces a new version in which a new handler to validate data formats has been added and designated to run First. The new base flow in the upgrade is illustrated in the following diagram:

![Diagram showing the new upgrade process]

Note that this new upgrade will now cause the event to be generated and check data formats whenever the appropriate transaction is posted. This is different, because before the Infor base version of this event did nothing.
Now the end-customer installs the Infor upgrade and uses the **App Metadata Sync** or **App Metadata Transport** utility to import and synchronize the new handler metadata with the existing metadata for this event. Because the business partner handler BP1 is marked to execute **First**, that handler and all handlers "attached" to it (EC1 and EC2) are placed at the beginning. Because the business partner handler AV1 is marked to execute **Last**, that handler is placed at the end. Because the Infor handler SL1 is not marked to execute in any particular order, it is inserted between EC2 (which is attached to BP1) and AV1.

After the integration is complete, the new handler sequence is as follows:
Overriding Others’ Handlers

As demonstrated briefly in Using Specific Chronology on page 204, downstream event handler creators can create handlers that, in effect, replace an upstream handler. This is done primarily with the use of the Instead option in the Chronology field on the Event Handlers form. This option allows multiple overriding handlers to coexist. In other words, this option is considered a non-exclusive override.

There is also a second option for situations where a handler’s creator might want to override all other handlers, regardless of who owns them or when they were added to the sequence. This is done using the Exclusively (instead) option in the second Keep With field. This option is considered an exclusive override.

Non-exclusive Overrides

In this example, the Infor application development team creates an application event and adds an event handler, SL1, for it.

The application is released with this event metadata included as part of the standard product.

An Infor business partner installs the software with this event metadata. They decide, however, that they want to replace the standard Infor event handler with one of their own. So they create handler BP1 and select Instead in the Chronology field on the Event Handlers form. After saving their handler, the sequence is as follows:

Notice that this effectively bypasses and deactivates the original Infor event handler, SL1.
The business partner then releases the add-on product with this new event handler in place.

In the meantime, a second Infor business partner does the same thing. For the sake of the example, the second business partner’s handler is labeled AV1.

Now suppose that a customer purchases and installs the standard Infor product. At this point, the sequence is the same as what both business partners received originally:

When the event is generated, the Infor event handler executes.

The customer then purchases and installs the add-on product from the first business partner. At this point, the sequence is the same as the first business partner’s. The App Metadata Sync or App Metadata Transport utility recognizes the override of BP1 and automatically deactivates the Infor handler SL1.

But then, the customer also purchases the second business partner’s add-on product and installs it. The App Metadata Sync or App Metadata Transport utility recognizes the new
override of AV1 and allows both overrides to coexist. When the event is generated, both handlers BP1 and AV1 execute. The sequence now looks something like the following:

Note that, in this case, there is no reliable way to know the precise order in which event handlers BP1 and AV1 will execute, since they coexist at the same level in the sequence and execute independently of one another.

Exclusive Overrides

To continue the example, suppose that the Infor application development team now adds another event handler, SL2, for the same event. When Infor releases the upgrade, the sequence for this event now looks like this:

The first business partner installs the upgrade using the App Metadata Sync or App Metadata Transport utility. Once again, they decide they want to override the new Infor handler with one of their own. So they create a new handler, BP2, and assign it to run
Instead of SL2. When they release the new version of their add-on product, the sequence for this event now looks like this:

Note that, so far, this is typical non-exclusive override behavior.

The second business partner does the same thing as the first, with one exception: They specify that the new handler, AV2, is to execute Exclusively Instead, regardless of any other handlers that might exist at that point in the sequence. When they release their new version, the sequence for this event looks very similar to that of the first business partner:

However, the outcome is quite different.

Suppose the customer now installs the Infor software upgrade and tries to install the upgrades for both business partners as well. All goes well until the customer tries to install the new version of the second business partner’s add-on product. When the customer tries
to synchronize the event metadata from the second business partner, the **App Metadata Sync** or **App Metadata Transport** utility generates an error, because the handler AV2 is set to execute *exclusively*, which puts it in conflict with the first business partner’s handler BP2. The two cannot coexist.

To resolve the conflict, the customer would have to uninstall one of the business partner handlers and contact them for guidance.

In practice, this should be a very rare occurrence.

Disabling the Ability to Override

Owners of "higher-level" handlers can designate that their handlers cannot be overridden. Downstream developers cannot use a **Chronology** setting of **Instead** or **Exclusive Instead** with such a handler.

To designate an event handler as not subject to being overridden, on the **Event Handlers** form, use the **Overrideable** check box. When this check box is cleared, no lower-level handlers can override that handler. In this case, the **Instead** and **Exclusive Instead** options are disabled for any other handlers (downstream) that use the **Keep With** field to associate those handlers with this handler.
Dealing with Obsolete Handlers

If you are an Infor or business partner developer, you must not delete handler metadata that has been exported and made available to other (downstream) metadata developers. This is because those developers might have their own metadata associated with your metadata, by means of the Keep With feature, before your next release.

If your exported metadata removes a handler that is referenced by their handler’s Keep With field, a broken link occurs during import. This forces the App Metadata Sync or App Metadata Transport utility to relegate the downstream developer’s handler to “free floating” status, meaning its sequence is less controllable. This is especially true if you—or an upstream developer whose handlers are referenced by your handlers’ Keep With field—later resequence your handlers.

Therefore, when a handler is no longer needed, on the Event Handlers form you should mark it as obsolete, by selecting the check box labeled Obsolete. This effectively and permanently deactivates the handler but leaves it in the sequence, so that associations with it do not get destroyed.

For example, consider the following sequence of handlers:

- FrameEvent
  - Handler ID = S1
    - Handler ID = A1
      - Handler ID = S2
        - Handler ID = C1
          - Handler ID = C2
            - Handler ID = C3
            - Handler ID = S3
Suppose, after initially releasing this version of the metadata, Infor decides not to use the handler S1 anymore. Because a business partner has associated handler A1 with this handler, if Infor simply deletes handler S1, a broken link occurs the next time the business partner or a customer who uses the business partner’s add-on product tries to install the upgraded metadata. Handler A1 is relegated to a “free floating” status (as in the following diagram), and unexpected consequences can result after subsequent resequencing or synchronizing operations.
So, rather than delete handler S1, Infor would mark it as **Obsolete** before the next release. Then, when the update is installed, the reference to handler S1 is not broken, though the system ignores it when handling this event, as in the following:
Event Flow Options

The following flow diagram illustrates many of the possible ways an event can be generated and the types of flows that can result. This diagram highlights the differences between previously existing functionality and the new functionality offered by the application event system.
Glossary

Access As  
An identifier used to identify who created and owns a metadata object. This identifier is also used to control which metadata objects you can modify. You can modify only those metadata objects that are associated with the current Access As field value, as displayed on the Access As form.

For more information, see About the Access As Identifier on page 3-39.

adjourning event  
An event action that must wait for an external stimulus before it can continue. When the system encounters an adjourning event action, the event handler state is set to retest or to time out after a specified time. The event system then processes it at the next opportunity and resumes.

asynchronous event handler  
An event handler designed to execute independently of other event handlers, and whose triggering process does not block while waiting for it to finish. These event handlers are sent to an event queue, from which they execute in FIFO order.

See also asynchronous event handler.

database tier  
That part of the system framework that stores the actual data for:

- Rendering forms (the Forms database)
- Storing all customer business data (the Application database)

Programs can also run in this tier, where they can access data very quickly without having to traverse network paths.

For more information, see the “System Architecture” chapter in your System Administration Guide.

database tier  
That part of the system framework that stores the actual data for:

- Rendering forms (the Forms database)
- Storing all customer business data (the Application database)

Programs can also run in this tier, where they can access data very quickly without having to traverse network paths.

For more information, see the “System Architecture” chapter in your System Administration Guide.

end-customer  
The company that has bought and is using the Infor ERP SyteLine software. They might also have bought one or more add-on products by Infor business partners to customize and enhance the performance of their system.

Explorer  
A window in the application (similar to Windows Explorer) that displays folders containing form names, providing a means to find, organize, and open forms. Explorer is the default window when the application opens. To reopen Explorer if it is closed, select View > Explorer on the menu bar.
event

A uniquely named incident that can be triggered by:

- An action performed by somebody working in the system
- A particular condition that occurs while the system is running
- A certain value being exceeded in a database record
- Another event or one its handlers
- Other, similar occurrences

A particular event can possibly be triggered by multiple situations or conditions, and you can determine how the system responds to each situation.

To be useful, an event must have one or more event handlers defined to respond to the event.

For more information, see About Events on page 3-39.

event action

Metadata that specifies a unit of work to perform during the execution of an event handler. A single event handler can have multiple event actions. Depending on its action type, an event action can do such things as:

- Evaluate and compare expressions, using the results to select which event action of its event handler to perform next.
- Affect the event's visual state.
- Complete the event.
- Set event variables.
- Call methods.
- Perform other predefined tasks.

For more information, see About Event Actions on page 3-44.

event action state

A set of data that shows the current state of a running or finished event action. This data includes information about when the event action started running, its current status, the number of times it has run, and other information.

event action type

A designator that limits or directs what an event action can do. This designator essentially determines the unit of work that each event action performs.

event global constant

A named static value that event expressions can reference during processing of an event handler.

event handler

Metadata that defines a portion of work to be performed upon the firing of a particular event. Each event handler is uniquely identified in the system with the combination of an event name and a handler sequence number. Each event handler is comprised of one or more event actions and, optionally, an initial state.

For more information, see About Event Handlers on page 3-43.
**event handler state**  
A set of data that shows the current state of a running or finished event handler. This data includes information about when the event handler started running, its current status, timeout settings, and other information.

**event initial variable**  
See *initial variable*.

**event input parameter**  
A named static value that is passed to an event upon its triggering. This value can be set to be available as an output after the event finishes executing.

Any number of uniquely named event input parameters can be collected before firing an event. Upon firing the event, each is converted to an *event parameter*.

**event message**  
A message, initiated by the event system, sent from one system user to another, that in many respects simulates an e-mail message. Event messages are created by event actions of a Notify or Prompt type, or by *Inbox* activities such as *Forward*, *Reply*, and *Reply All*, or from the Send Message form.

Event messages can appear in the *Sent Items* folder of the sender, and the *Inbox* of each recipient.

For more information, see *Event Messages* on page 5-65.

**event output parameter**  
A named static value passed from an event upon its finish. Any number of uniquely named event output parameters can be associated with an event. Each output is created from an *event parameter* marked for output.

**event parameter**  
A named storage area retrievable by an event action, that is associated with an event that has been generated and is processing. The system creates event parameters from event input parameters when the event is generated.

Event parameters can be set to be available to whatever process generated the event, after the event finishes. In this case, they can also be set by event actions and can result in the creation of event output parameters.

**event queue**  
A *FIFO* list of events and event handlers to be processed asynchronously. Each entry has an associated user name, configuration name, and request date.

**event state**  
1. A collection of data related to the current status of a running or finished event. This status is viewed using the *Event Status* form.

2. An optional text string that displays on the *Event Status* form when the event reaches certain milestones or finishes executing successfully. This text string is defined by the event handler's creator and associated with the *Achieve Milestone* and *Finish* action types as a parameter.

**event trigger**  
A condition that causes an event to be generated with optional parameters.

For more information, see *About Event Triggers* on page 3-41.
event variable

A named storage area, the value of which can be set and retrieved by an event action associated with a running event handler. When associated with a synchronous event handler, an event variable can be designated as Persistent, in which case the value of the variable can be passed on to the next event handler.

FIFO

First In, First Out.

framework

The multi-tiered software structure that makes up the entire system. In this application, the framework consists of three basic tiers:

- The client tier, known as WinStudio.
- The middle tier
- The database tier

For more information, see the "System Architecture" chapter in your System Administration Guide.

framework event

An event which has been designed to be generated only in reference to certain framework occurrences.

initial variable

A named static value for an event variable associated with an event handler. This value provides the initial value of the event variable when the event handler begins executing.

Each event variable contains an authorization level that provides a default access within the scope of an event action that:

- Does not have a default access value defined on the Variable Access tab of the Event Actions form. In this case, default access value is determined on the Event Variable Groups form.
- Has a default access of Default on the Variable Access tab of the Event Actions form.

metadata

In the context of the application event system, refers to the practice of using uncompiled code and information about data formats that are interpreted during run time, rather than compiled code (also called "procedural code").

middle tier

The layer of software in the database system which provides the connections between the client tier (WinStudio) and the database tier. The middle tier has two primary functions:

- To provide access from the client tier to the database through IDOs (Intelligent Data Objects). The client tier never communicates directly with the database. In this respect, the middle tier can be thought of rather like a telephone: It allows two parties to talk to one another, but not face-to-face.
- To receive form-rendering requests from the client tier, retrieve the appropriate form-rendering data from the forms database, and return that data to the client so that the form displays correctly.
For more information about the middle tier, see the "System Architecture" chapter in your System Administration Guide.

**synchronous event handler**

An event handler designed to execute sequentially with other handlers, and whose triggering process blocks while waiting for it to finish (unless part of an event fired asynchronously). If any one event handler in the sequence fails, then the whole sequence fails.

*See also* asynchronous event handler.

**transactional events or event handlers**

One of a group of events or handlers that are included in a single transaction. Typically, all the event handlers and their actions must execute successfully before the transaction is considered finished and any results or data are committed.

If any of the events or handlers in a transaction do not finish successfully, the entire transaction fails and all data and variable values roll back (revert) to their original values.

**WinStudio**

The client software in the database system, sometimes referred to as the "presentation layer," through which users interact with information in the database. WinStudio displays forms and provides the interface through which users can find, add, edit, sort, and delete data.

For more information, see the "System Architecture" chapter in your System Administration Guide.
Event Handlers 43
Events 39
location 51
designing custom events and handlers 37–59
DEST function 143, 151
Destination parameter 143
disabling event handlers
   individually 21
   with Session Access As 20
Dispatch IDO Request action type 136
documentation, additional for Infor ERP SL 7

E
E function 158
EarliestResponse voting rule 73
elements of the application event system 39
ENTRYFORM function 143
enumerations 196
event action parameter forms 48
event actions 44
   controlling sequence 19
   defining 44
   parameters 140
   setting sequence order 56
Event Actions form 44
Event Global Constants form 49
Event Handler Revisions form 63
Event Handler Status form 62
event handlers 43
   controlling sequence 19
   defining 43
   designing custom 54
   disabling individually 21
   disabling with Session Access As 20
   restricting which run 20
   revisions 36
   setting sequence order 55
   setting up custom 54
   status of
      Failure 28
      Success 28
   success, failure, and retries 28
   suspending 24
   synchronicity (summary) 127
   synchronous and asynchronous 22
   transactional 32
Event Handlers form 43
event messages 65–74
   indeterminate voting results 74
   prompts and responses 71
   related forms 66
   voting rules 71
event queue
   Framework Event Service 34
   processing order 34
Event Queue form 62
Event Revisions form 63
Event Status form 62
event triggers 41
   defining 41
   retesting 42
   setting conditions for 42
Event Triggers form 41
Event Variable Groups form 49
event variables 49
EVENTNAME function 143, 158
EVENTREVISION function 158
events 39
   action types 135
   asynchronous 22
   defining 40
   firing from tiers 126
   flow diagram 18
   framework (core) 128
   global constants 49
   handlers 43
   handling 18
   message-related forms 66
      Inbox 66
      Saved Messages 68
      Send Message 69
   naming 40
   parameter forms 48
   parameters 45
   passing 47
   setting values and variables 47
   revisions 36
   setting up custom 54
   suspend-committing mode 25
   suspending 24
   suspend-validating mode 24
   synchronicity (summary) 127
   synchronous 22
   transactional 31
   triggers 41
      defining 41
      examples 18
      retesting 42
      setting conditions for 42
   types 40
   variables 49
   where they can be generated from 18
Events form 39
EVENTSTATE function 158
EVENTTITLE function 158
Exclude Blank Access As setting
   (Session Access As form) 20
Execute IDO Request action type 136
expansions for functions
   complex 170
   nested 170
expressions
  functions
    pre-parser 169
    standard 155
  grammar 173–197
    character sets 175
    enumerations 196
    rules 177
    start symbol 175
    terminals 176
  keyword paren lists 195
  numeric-castable 177
  operators 171
  rules
    Boolean 177
    dates 188
    numeric 185
    restricted arguments 195
    strings 182
  rules for 177
  scalar 177
  typeless, rules 180

F
  Fail action type 136
  Failure status, event handler 28
  failures
    causes 28
    handling with stored procedures 199
    ignoring 28
  FE function 159
  FGC function 159
  FILECONTENTS function 159
  FILTER function 143, 159
  FILTERFORM function 143
  FILTERMETHODPARM function 159
  FILTERPROPERTY (FP) function 160
  FILTERSTRING function 160
  Finish action type 136
  FireGenericNotifySp 122
  firing events (from tiers) 126
  FLOOR function 160
  forms 40
    Access As 39
    design
      authorization group for 51
      Event Actions 44
      Event Global Constants 49
      Event Handlers 43
      Event Triggers 41
      Event Variable Groups 49
      Events 39
      location 51
      Workflow Event Handler Activation 52
  event action parameters 48
  message-related 66
    Inbox 66
    Saved Messages 68
    Send Message 69
  status
    authorization group for 61
    Event Handler Revisions 63
    Event Handler Status 62
    Event Queue 62
    Event Revisions 63
    Event Status 62
    location 61
  FP function 160
  Framework Event Service 34
    administering 35
    processing order 34
    setting up 34
  framework events 40, 128
  FSV function 160
  functions 140
    ACTION 141
    ACTIONSEQ 155
    ACTIONTYPENAME 155
    ANYHANDLERSFAILED 155
    APPNAME 155
    BEGINDATE 155
    BODY 141
    CAST 156
    CATEGORY 141
    CC 141
    CEILING 156
    CHOICES 142
    CLIENTSUBSTITUTE 156
    COLLECTION 142, 150
    COMMIT 142
    COMPANYNAME 156
    complex expansions 170
    CONDITION 142
    CONFIGNAME 156
    CURRDATE TIME 157
    CUSTOMDELETE 157
    CUSTOMINSERT 157
    CUSTOMLOAD 143
    CUSTOMLOADPARMS 143
    CUSTOMUPDATE 157
    DATE(ymd) 157
    DATE(ymdhmsm) 157
    DATEADD 157
    DATEDIFF 158
    DATEPART 158
    DBFUNCTION 158
    DEST 143, 151
    E 158
    ENTRYFORM 143
    EVENTNAME 143, 158
    EVENTREVISION 158
    EVENTSTATE 158
<table>
<thead>
<tr>
<th>Event Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Title</td>
<td>158</td>
</tr>
<tr>
<td>FE</td>
<td>159</td>
</tr>
<tr>
<td>FGC</td>
<td>159</td>
</tr>
<tr>
<td>FILECONTENTS</td>
<td>159</td>
</tr>
<tr>
<td>FILTER</td>
<td>143, 159</td>
</tr>
<tr>
<td>FILTERFORM</td>
<td>143</td>
</tr>
<tr>
<td>FILTERMETHODPARM</td>
<td>159</td>
</tr>
<tr>
<td>FILTERPROPERTY (FP)</td>
<td>160</td>
</tr>
<tr>
<td>FLOOR</td>
<td>160</td>
</tr>
<tr>
<td>for expressions</td>
<td></td>
</tr>
<tr>
<td>pre-parser</td>
<td>169</td>
</tr>
<tr>
<td>standard</td>
<td>155</td>
</tr>
<tr>
<td>FP</td>
<td>160</td>
</tr>
<tr>
<td>FSV</td>
<td>160</td>
</tr>
<tr>
<td>FV</td>
<td>160</td>
</tr>
<tr>
<td>GC</td>
<td>160</td>
</tr>
<tr>
<td>HANDLERIGNORESFailure</td>
<td>161</td>
</tr>
<tr>
<td>HANDLERSUSPENDS</td>
<td>161</td>
</tr>
<tr>
<td>HANDLERSYNCHRONOUS</td>
<td>161</td>
</tr>
<tr>
<td>HANDLERTRANSACTIONAL</td>
<td>161</td>
</tr>
<tr>
<td>HASBEGUN</td>
<td>161</td>
</tr>
<tr>
<td>HASFINISHED</td>
<td>161</td>
</tr>
<tr>
<td>IDO</td>
<td>143, 161</td>
</tr>
<tr>
<td>IDOREQUEST</td>
<td>143</td>
</tr>
<tr>
<td>IF</td>
<td>161</td>
</tr>
<tr>
<td>IMAGE</td>
<td>144</td>
</tr>
<tr>
<td>INITIATOR</td>
<td>161</td>
</tr>
<tr>
<td>INSIDEDATABASE</td>
<td>161</td>
</tr>
<tr>
<td>INSTR</td>
<td>162</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>144</td>
</tr>
<tr>
<td>KEY</td>
<td>144</td>
</tr>
<tr>
<td>LEN</td>
<td>162</td>
</tr>
<tr>
<td>LOADFLAGS</td>
<td>162</td>
</tr>
<tr>
<td>LOWER</td>
<td>162</td>
</tr>
<tr>
<td>MESSAGE</td>
<td>162</td>
</tr>
<tr>
<td>METHOD</td>
<td>144, 145, 150, 162</td>
</tr>
<tr>
<td>METHODPARM</td>
<td>162</td>
</tr>
<tr>
<td>METHODPARMS</td>
<td>163</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>144</td>
</tr>
<tr>
<td>MODIFIEDPAYLOADACCESS</td>
<td>145</td>
</tr>
<tr>
<td>nested expansions</td>
<td>170</td>
</tr>
<tr>
<td>NEW</td>
<td>145</td>
</tr>
<tr>
<td>NEWGUID</td>
<td>163</td>
</tr>
<tr>
<td>NONRESPONDERLIST</td>
<td>163</td>
</tr>
<tr>
<td>OLD</td>
<td>145</td>
</tr>
<tr>
<td>OPTIMISTICLOCKING</td>
<td>145</td>
</tr>
<tr>
<td>ORDERBY</td>
<td>146</td>
</tr>
<tr>
<td>ORIGINATOR</td>
<td>163</td>
</tr>
<tr>
<td>P</td>
<td>163</td>
</tr>
<tr>
<td>PASSWORD</td>
<td>147</td>
</tr>
<tr>
<td>PAYLOADACCESS</td>
<td>148</td>
</tr>
<tr>
<td>POSTQUERYACTIONS</td>
<td>163</td>
</tr>
<tr>
<td>POWER</td>
<td>163</td>
</tr>
<tr>
<td>PREFCHOICE</td>
<td>148</td>
</tr>
<tr>
<td>PROPERTIES</td>
<td>148</td>
</tr>
<tr>
<td>PROPERTY (P)</td>
<td>164</td>
</tr>
<tr>
<td>PROPERTYMODIFIED</td>
<td>164</td>
</tr>
<tr>
<td>PROPERTYNAMES</td>
<td>164</td>
</tr>
<tr>
<td>QUESTION</td>
<td>149</td>
</tr>
<tr>
<td>QUORUM</td>
<td>149</td>
</tr>
<tr>
<td>RECIPIENTLIST</td>
<td>164</td>
</tr>
<tr>
<td>RECIPIENTS</td>
<td>164</td>
</tr>
<tr>
<td>RECORDCAP</td>
<td>149, 164</td>
</tr>
<tr>
<td>REPLACE</td>
<td>165</td>
</tr>
<tr>
<td>RESPONDERLIST</td>
<td>165</td>
</tr>
<tr>
<td>RESPONDERS</td>
<td>165</td>
</tr>
<tr>
<td>RESULT</td>
<td>149</td>
</tr>
<tr>
<td>RETEST</td>
<td>150</td>
</tr>
<tr>
<td>ROUND</td>
<td>166</td>
</tr>
<tr>
<td>ROW</td>
<td>150</td>
</tr>
<tr>
<td>ROWS</td>
<td>166</td>
</tr>
<tr>
<td>SAVEMESSAGE</td>
<td>150</td>
</tr>
<tr>
<td>SET</td>
<td>146, 149, 150, 151</td>
</tr>
<tr>
<td>SETGLOBALVALUES</td>
<td>150</td>
</tr>
<tr>
<td>SETMETHODPARMVALUES</td>
<td>144</td>
</tr>
<tr>
<td>SETPARMVALUES</td>
<td>147</td>
</tr>
<tr>
<td>SETPROPVALUES</td>
<td>148</td>
</tr>
<tr>
<td>SETVALUE</td>
<td>153</td>
</tr>
<tr>
<td>SETVARVALUES</td>
<td>153</td>
</tr>
<tr>
<td>STATE</td>
<td>150</td>
</tr>
<tr>
<td>SUBJECT</td>
<td>150</td>
</tr>
<tr>
<td>SUBSTITUTE</td>
<td>166</td>
</tr>
<tr>
<td>SUBSTRING</td>
<td>167</td>
</tr>
<tr>
<td>SV</td>
<td>167</td>
</tr>
<tr>
<td>SYNCHRONOUS</td>
<td>151</td>
</tr>
<tr>
<td>TASKNAME</td>
<td>151</td>
</tr>
<tr>
<td>TASKNUMBER</td>
<td>151</td>
</tr>
<tr>
<td>TASKPARMS</td>
<td>151</td>
</tr>
<tr>
<td>TASKSTATUS</td>
<td>151</td>
</tr>
<tr>
<td>TGC</td>
<td>169</td>
</tr>
<tr>
<td>TIMEOUT</td>
<td>151</td>
</tr>
<tr>
<td>TO</td>
<td>152</td>
</tr>
<tr>
<td>TRANSACTIONAL</td>
<td>152</td>
</tr>
<tr>
<td>TRUNC</td>
<td>167</td>
</tr>
<tr>
<td>TV</td>
<td>169</td>
</tr>
<tr>
<td>UNMODIFIEDPAYLOADACCESS</td>
<td>152</td>
</tr>
<tr>
<td>UPPER</td>
<td>167</td>
</tr>
<tr>
<td>URL</td>
<td>152</td>
</tr>
<tr>
<td>USELOCALPASSWORD</td>
<td>153</td>
</tr>
<tr>
<td>USERDESC</td>
<td>167</td>
</tr>
<tr>
<td>USERNAME</td>
<td>153, 168</td>
</tr>
<tr>
<td>V</td>
<td>168</td>
</tr>
<tr>
<td>VARIABLENAME</td>
<td>168</td>
</tr>
<tr>
<td>VARIABLEVALUE</td>
<td>168</td>
</tr>
<tr>
<td>VOTINGDISPARITY</td>
<td>168</td>
</tr>
<tr>
<td>VOTINGRESULT</td>
<td>168</td>
</tr>
<tr>
<td>VOTINGRULE</td>
<td>154</td>
</tr>
<tr>
<td>VOTINGTIE</td>
<td>168</td>
</tr>
<tr>
<td>WAITFORQUORUM</td>
<td>154</td>
</tr>
<tr>
<td>WORKINGDIR</td>
<td>168</td>
</tr>
<tr>
<td>XML</td>
<td>154</td>
</tr>
</tbody>
</table>
METHOD function 144, 145, 150, 162
METHODPARM function 162
METHODPARMS function 163
MINIMUM function 144
MinimumCount voting rule 72
MinimumCountPreferredChoice voting rule 73
MinimumPercentage voting rule 73
MinimumPercentagePreferredChoice voting rule 73
MODIFIEDPAYLOADACCESS function 145

N
naming events 40
nested expansions of functions 170
NEW function 145
NEWGUID function 163
NONRESPONDERLIST function 163
Notify action type 137
numeric expressions, rules 185
numeric-castable expressions 177

O
obsolete handlers 226
OLD function 145
operators for expressions 171
OPTIMISTICLOCKING function 145
ORDERBY function 146
ORIGINATOR function 163

P
P function 163
parameters
Action 141
Body 141
Category 141
CC (carbon copy) list 141
Choices 142
Collection 142, 150
Commit 142
Condition 142
Custom load method 143
Custom load method parameters 143
Description 143
Destination 143
Entry form 143
Event name 143
Filter 143
Filter form 143
for event actions 140
IDO 143
IDO request 143
Image 144
Interval 144
Key value 144
Method 144
Method Parameters 144
Minimum count or percentage 144
New value 145
Old value 145
Operation (METHOD) 145
Optimistic lock 145
Order By clause 146
Output 146
Output parameters 146
Parameter/Expression pairs 147
passing 47, 199
Password 147
Payload access 145, 148, 152
Preferred choice 148
Properties list 148
Property/Expression pairs (SETPROPVALUES) 148
Question 149
QUORUM 149
Record cap 149
Response (SET) 149
Restest interval 150
Result 149
Result set identifier 150
Result XML (SET) 150
Row 150
Save in Sent Items? 150
Session variable/Expression pairs 150
setting value and variables 47
State 150
Stored procedure name 150
Subject 150
Synchronous 151
syntax for 45
Task name 151
Task number 151
Task parameters 151
Task Status 151
Timeout 151
Timeout destination 151
Title (SET) 151
to recipient list 152
Transaction 152
URL 152
USELOCALPASSWORD 153
USERNAME 153
Value (SETVALUE) 153
Variable/Expression pairs 153
Voting rule 154
Wait for quorum 154
XML 154
XSLT 154
paren lists for keywords 195
PASSWORD function 147
PAYLOADACCESS function 148
performing upgrades, integrating metadata 216
plurality voting rule 72
POSTQUERYACTIONS function 163
POWER function 163
PREFCHOICE function 148
PreferredChoice voting rule 73
procedural code 13, 234
Prompt action type 138
prompts and responses 71
   indeterminate voting results 74
   voting rules 71
PROPERTIES function 148
PROPERTY (P) function 164
PROPERTYMODIFIED function 164
PROPERTYNAMES function 164

Q
QUESTION function 149
queued events
   Framework Event Service 34
      processing order 34
QUORUM function 149
Quorums 74

R
RECIPIENTLIST function 164
RECIPIENTS function 164
RECORDCAP function 149, 164
reference tables 125–172
REPLACE function 165
RESPONDERLIST function 165
RESPONDERS function 165
restricted arguments for expressions 195
restricting which handlers run 20
RESULT function 149
resumption of adjourned events 27
RETEST function 150
Retest Interval parameter 150
retesting event triggers 42
Retrying event handlers 29
retrying event handlers 28
revisions, events and event handlers 36
rolling back transactions 33
ROUND function 166
ROW function 150
ROWS function 166
rules
   expression grammar
      constant references 177
      dates 188
      event parameter references 177
      numeric 185
      restricted arguments 195
      strings 182
      typeless expressions 180
      variable references 177
   voting, prompts 71
Run Background Task action type 138
Saved Messages form 68
SAVEMESSAGE function 150
scalar 177
scalar expression expressions 177
Scalar tuples 181
scenarios, samples 75–123
Send Email action type 138
Send Message form 69
sequence
   event actions 19, 56
   event handlers 19, 55
Service Configuration Manager utility 34
Session Access As form 20
SessionOnLogin 128, 133
SessionOnLogout 128, 133
SessionOnVarChanged 128, 133
Set Attributes action type 138
SET function 146, 149, 150, 151
Set Values action type 138
SETGLOBVALUES function 150
SETMETHODPARMVALUES function 144
SETPARMVALUES function 147
SETPROPVALUES function 148
setting
   conditions for event triggers 42
   sequence order
      actions 56
      handlers 55
   variable and parameter values 47
SETVALUE function 153
SETVARVALUES function 153
Sleep action type 139
STATE function 150
status
   event handlers
      Failure 28
      Success 28
forms
   authorization group for 61
   Event Handler Revisions 63
   Event Handler Status 62
   Event Queue 62
   Event Revisions 63
   Event Status 62
   location 61
   tracking 61–63
stored procedures
   calling synchronous events and handling failure 199
   passing parameters to a synchronous event 199
   samples 199
strings, rules 182
SUBJECT function 150
SUBSTITUTE function 166
SUBSTRING function 167
Success event handler status 28
support
   contacting Infor 8
signing up for 8
suspend-committing mode 25
suspend-validating mode 24
suspension 24–26
event handlers 24
events 24
process for non-suspended events 26
process for suspended events 24
SV function 167
synchronicity 22–23
event handlers (summary) 127
events (summary) 127
synchronization of metadata 201–228
inherent hierarchy of metadata 202
overview and rationale 201
synchronous
calling events from stored procedures 199
event handlers 22
events 22
events in transactions 31
SYNCHRONOUS function 151
syntax for event parameters 45

T
TaskListCheck 134
TASKNAME function 151
TASKNUMBER function 151
TASKPARMS function 151
TASKSTATUS function 151
TGC function 169
ties in prompt votes 74
Timeout destination parameter 151
TIMEOUT function 151
Title parameter (SET) 151
TO function 152
tracking status of events 61–63
transactional
event handlers 32
events 31
TRANSACTIONAL function 152
transactions 31–33
rolling back 33
with asynchronous events 32
with synchronous events 31
Transform XML action type 139
translatable captions 67
triggers for events 41
defining 41
examples 18
retesting 42
setting conditions for 42
TRUNC function 167
TV function 169
types of events 40

U
UNMODIFIEDPAYLOADACCESS function 152
Update Collection action type 139
upgrades, integrating metadata through 216
UPPER function 167
URL function 152
USELOCALPASSWORD function 153
USERDESC function 167
USERNAME function 153, 168

V
V function 168
Value (SETVALUE) parameter 153
VARIABLENAME function 168
variables
for events 49
setting parameter values 47
VARIABLEVALUE function 168
voting results indeterminate for prompts 74
voting rules for prompts 71
ConditionalPlurality 72
EarliestResponse 73
Majority 72
MinimumCount 72
MinimumCountPreferredChoice 73
MinimumPercentage 73
MinimumPercentagePreferredChoice 73
Plurality 72
PreferredChoice 73
VOTINGDISPARITY function 168
VOTINGRESULT function 168
VOTINGRULE function 154
VOTINGTIE function 168

W
Wait action type 139
Wait for Quorum 74
WAITFORQUORUM function 154
Web-based Inbox form 68
where events can be named 40
Workflow Event Handler Activation form 52
Workflow event handlers 15
WORKINGDIR function 168

X
XML function 154
XSLT function 154