MAGIC SOFTWARE ENTERPRISES LTD.

Magic eDeveloper
Part of the Magic eBusiness Platform

Event Driven Architecture

The information in this document is subject to change without prior notice and does not represent a commitment on the part of MSE.

MSE makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose.

The software described in this document is furnished under a license agreement. The software may be used or copied only in accordance with the terms and conditions of the license agreement. It is against the law to copy the software on any medium except as specifically allowed in the license agreement.

No part of this document and/or databases may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or information recording and retrieval systems, for any purpose other than the purchaser's personal use, without the prior express written permission of MSE.

All references made to third-party trademarks are for informational purposes only regarding compatibility with the products of Magic Software Enterprises Ltd.

Unless otherwise noted, all names of companies, products, street addresses, and persons contained herein are part of a completely fictitious scenario or scenarios and are designed solely to document the use of Magic.

Magic® is a registered trademark of Magic Software Enterprises Ltd.
PC/TCP® Network Software is a registered trademark of FTP Software Inc.
Microsoft® is a registered trademark, and Windows™, WindowsNT™ and ActiveX™ are trademarks of Microsoft Corp.

Clip art images copyright by Presentation Task Force, a registered trademark of New Vision Technologies Inc.

All other product names are trademarks or registered trademarks of their respective holders.

04 03 02 6 5 4 3 2 1

Copyright © 2002 by Magic Software Enterprises Ltd. All rights reserved.

Contents

1 Overview

Event Handling Architecture 7
- Clearer Code 7
- Responsive Application 7
- Code Re-use 7

Event Handling vs. Record Main Logic 8
- Record Main – Dataview Definition Only 8
- Record Main Logic Supported 8

2 Events, Triggers, and Handlers

Terminology 9
- Event 9
- Trigger 9
- Handler 9

Event Types 10
- System Events 10
- Internal Events 10
- Timer Events 10
- Expression Events 10
- Error Events 10
- User Events 10

3 Handlers

Defining Event Handlers 11
- Event-Handler Logic 12

Executing a Handler 12
- Searching for the Handler 12
- Handler Hierarchy 13

Handler's Scope 13
Propagation 14
Enable/Disable Handler 14
Control-specific Handler 14

4 Raise Event Operation

General Usage 16
Invoking a Magic Internal Handler 16
Invoking a User-defined Handler 16
Clarifying Your Logic 17
The Raise Event Operation 17
Name 17
Wait 17
Arguments 17
Arguments Passed by Reference 18
Arguments Passed by Value 18
Arguments Passed to a Propagating Handler 18
Raising Events from Menus and Push Buttons 18
Push Buttons 18
Menu Repository 19

5 Event Polling

Immediate Handler Execution 20
Synchronous Raise Event Operation 20
Error Event 20
Queued Events 20
Asynchronous Raise Event Operations 20
System, Internal, Timer, and User Events 20
Expression Events 21
Polling the Events From the Queue 21
Idle Time 21
Batch Event Interval 21
Record Event Interval 22
Allow Events 22

6 Control Level Handlers

Control Level 23
Control Prefix 23
Control Suffix 23
Control Verification 24
Control Verification vs. Control Suffix 24
Using the Control Suffix Handler 24
Using the Control Verification Handler 25

7 User Events

Constructing a User Event 26
Description 26
Trigger 26
Force Exit 27
Using a Synchronous Raise Event Operation 28

8 Handler Resources and Information

Runtime Information and Resources 29
Variables 29
I/O Devices 30
Runtime Task Tree 30
Handler Scope 31
Referring to the Triggering Object 32
THIS() Function 32
The Triggering Variable 33
The Triggering Task 33
Handled Control 33

9 Handling Multi-marked Records

Handling Multi-marked Records 34
From Top to Bottom 34
Full Record Cycle 34
The Current Parked Record 35
Internal Handling of Multi-marked Records 35
Triggering Events During the Process 35

Supporting Functions 35
How Many Records are Marked? 35
The Currently Processed Record 36
Beginning and End of the Process 36
Maintaining Record Marking 36
Stopping the Process 36

10 Afterword

.... 37

Chapter

1

Overview

Magic eDeveloper incorporates a new event-based engine.

With Magic eDeveloper you can better define your business logic using the event-driven architecture in the Magic runtime engine. The event-driven programming paradigm lets you create full-featured applications with clearly defined logic.

Event Handling Architecture

An event-driven application can respond to any runtime occurrence generated by the end-user or by the developer internally as part of the business logic.

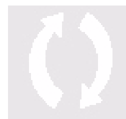
Clearer Code

Using event-driven architecture, your business logic is neatly defined in each task for the event the task should handle. This makes the task clearer, more understandable, and easier for other developers to maintain.

Responsive Application

The event-driven architecture lets you better handle runtime occurrences generated by the end-user or by third-party components, such as the database in use.

Code Re-use



The event-driven architecture lets you define your business logic as global logic that you define once and reuse throughout your application. Having such sharable logic segments saves you time in writing and maintaining repeated logic segments.

Event Handling vs. Record Main Logic

Until Magic Version 9, handling of the interactive stage of a task was defined entirely within the task's Record Main level. This type of logic definition produced long and tedious lists of operations in the Record Main level. In addition, the task dataview definition, defined by the Select and Link operations, was defined together with the task logic.

Record Main – Dataview Definition Only

When you use the Magic eDeveloper event-handling mechanism you only use the Record Main level to define the dataview. In this way the Record Main only includes the Select and Link operations that construct the task dataview. The task logic is defined in separated logic segments where each segment handles a specific interactive task stage.

Record Main Logic Supported

Magic eDeveloper still supports business logic execution defined as part of the Record Main level. However, you should avoid writing the application logic within the Record Main level, and use the event-handling mechanism instead. This produces a clearer, more responsive application.

Important:
Record Main logic is only supported in Online tasks and is not supported for Browser tasks. Browser-based application logic can only be defined using the event-handling mechanism.

Chapter

2

Events, Triggers, and Handlers

The event-driven architecture's building blocks are events, triggers, and handlers.

There are three basic elements that are used in the event-driven architecture: Events, Triggers, and Handlers. This chapter describes these elements.

Terminology

The basic definitions for Events, Triggers and Handlers are:

Event

An event is an abstract indication to the runtime engine that something has occurred within the running application. The runtime engine can either choose to ignore the event or respond to it. For example, a mouse click on a control issues an event in the runtime engine that a click has occurred. The runtime engine can then respond to this event accordingly.

Trigger

Every event is triggered when the application runs. Some events can be triggered as a response to an external activation, such as when a key is pressed or when the mouse is clicked. Other events can be triggered when a certain stage in the application is reached, such as an elapsed time period. This technical overview illustrates how events are defined in Magic, and also shows that some events can be defined as triggers of other events.

Handler

A handler is a unit of logic that is run when an event has occurred. A handler is defined to handle a specific event.

Event Types

Magic eDeveloper supports the following event types: System, Internal, Timer, Expression, Error, and User.

System Events

System events are triggered by keyboard actions. A System event is generally a key combination, such as pressing **F5** or **CTRL+R**.

Internal Events

Most Internal events are internal actions that the Magic engine handles. For example, whenever a **Next Line** event is triggered, Magic handles this event and performs the required logic to exit the current record and enter the next.

Some Internal events are triggered by interaction with the running application's interface. For example, the **Mouse Over** event is triggered when the end-user moves the mouse over a form control.

> Note:
>
> In previous Magic versions, the Keyboard Mapping file was used to assign key combinations to activate the Magic internal actions. This is actually setting System events as triggers for Internal events.

Timer Events

A Timer event is an event triggered at a defined time interval. For example, you can set a Timer event to be triggered every 10 minutes.

> Note:
>
> The Timer event is triggered exactly when the defined time interval elapses. This differs from the Timer event in previous Magic versions, which was only triggered when the time interval elapsed within the predefined Keyboard Idle Time.

Expression Events

An Expression event is an event defined by a Magic expression. The event is triggered whenever the defined expression is evaluated to a **True** value. Magic evaluates the value of this expression in time intervals defined in the **Keyboard idle seconds** Environment setting.

Error Events

An Error event is triggered when a database-related error occurs at runtime. For example, a **Duplicate index** error is triggered when Magic attempts to enter a record with duplicate index values.

User Events

You can create your own events, User events, as part of Magic tasks. You can use these to create additional events and clearly describe their functionality. All other event types are already defined in Magic.

Chapter

3

Handlers

You can define handlers in Toolkit mode to deal with events that occur during runtime.

A handler is the actual response to an occurring event. This chapter looks at the way a handler is defined, how the Magic engine executes a handler, and the various handler settings.

Defining Event Handlers

You can define handlers in every Magic Task Execution repository. If you want to create a new handler, you should create a new line in the Task Execution repository. Figure 3-1 below shows the handler line.

#	Level	Event	Details	Scope	Prop	Enable	Oper
4	Record	Main					0
5	Record	Suffix					0
6	Handler	Add Customer	Ctrl:	SubTree	No	Yes	0

Figure 3-1 The handler line in the Task Execution repository.

The **Event** property is the most significant handler property and it is mandatory. You should not define a handler without an event. This means that a handler cannot be a global handler for all event types. Zoom from the **Event** column to select the event.

When you select an event for a handler you determine that the handler should only respond when the selected event is triggered at runtime.

> **Note:**
>
> There are other handler properties that either have their own default values or are not mandatory. They are discussed further on in this document.

Event-Handler Logic

The operations listed in the Operations List, as shown in Figure 3-2 below, represent the handler logic executed when the selected event is triggered.

#	Level	Event	Details	Scope	Prop	Enable	Oper
4	Record	Main					0
5	Record	Suffix					0
6	Handler	Add Customer	Ctrl:	SubTree	No	Yes	2

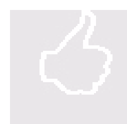
Operation : Handler On Add Customer

#	Operation	Name					Cnd
1	Verify : 0	Please Confirm Ent	Mod: Warnin	Disp: Box			Yes
2	Call Prog : 3	Update	Arg: 0	Frm: 0	Ret: ??		Yes

Figure 3-2 The Operations list for an **On Click** type handler.

> **Note:**
>
> Although an empty handler is still valid, normally a handler should include the operations you want to run.

Executing a Handler



A handler defined with an event is a valid handler. The Magic engine activates the logic defined within the handler when the corresponding event is triggered.

Searching for the Handler

When the corresponding event is triggered, Magic scans the runtime tree[1] from the bottom-most task up to the Main Program for a handler of the triggered event. Magic sequentially executes the operations defined for the handler once it has been found. After executing the last operation in this handler, the Magic engine continues up the runtime tree looking for another matching handler. After reaching the Main Program and executing any handlers found when handling internal events, the Magic engine looks for internal engine handlers for the triggered events.

1. The runtime tree is the collection of all the tasks that are currently running. The task order in the runtime tree is the order by which they were opened.

Handlers of the same event, which are defined in the same task, are executed according to their location in the Task Execution repository from the bottom upwards.

Handler Hierarchy

The previous section described how several handlers can handle the same event. These handlers can be defined in various tasks that are part of the runtime tree. They are executed according to their hierarchy in the runtime tree. The handler defined in the lowest level of a task is executed first. The handler defined in the Main Program, or in the Magic engine when dealing with Internal events, is executed last.

You can use the hierarchical handler structure and define high-level handlers for several different tasks. A handler defined in one program, Program A, is available for all other programs and subtasks called from Program A.

> **Tip:**
>
> The Main Program is the parent program of all other programs in the application, and it is always at the top of the runtime tree at runtime. This means that a handler defined in the Main Program is a global handler for the entire application.

Example: If you want to let the end-user check the current date by pressing **CTRL+D** anywhere in the application, you can define a handler on the CTRL+D System event and define the appropriate logic within this handler so it displays the current date.

Task:1 - Main Program

#	Level	Event	Details	Scope	Prop	Enable	Oper
2	Task	Suffix					0
3	Record	Main					0
4	Handler	Ctrl+D	Ctrl:	SubTree	No	Yes	1

Operation : Handler On Ctrl+D

#	Operation	Name					Cnd
1	Verify : 1	DSTR (DATE (),'##/##/	Mod: Warning	Disp: Box			Yes

Figure 3-3 A global handler defined in the Main Program.

The handler hierarchy lets you create high-level logic that can be used and re-used throughout the application. In addition, working with high-level handlers makes maintenance easier because modifications are carried out in one place.

Handler's Scope

You can define a handler to be available for the entire sub-tree, which is the part of the runtime tree from the current task downwards, or to only be available for the current task. You set this in the handler's **Scope** property.

When you set **Scope** to **Task**, Magic only executes the handler when the corresponding event is triggered within this task alone. If the event is triggered in a descendant task, Magic skips this handler as it scans the runtime tree.

When you set **Scope** to **SubTree**, the Magic engine executes this handler whether the corresponding event is triggered in this task or in one of its descendant tasks.

Global Handlers:

A handler defined in the Main Program has another value for the Scope property, **Global**. This Scope setting is relevant for Magic applications that are published as components.

Propagation

You might want to prevent the Magic engine from continuing to search for additional handlers in the runtime tree once it has run a particular handler. You can do this by setting the handler's Propagate property, labeled **Prop**, in the Propagation column.

If you set the property to **No**, the Magic engine stops scanning the runtime tree for additional handlers after executing the current handler's operations. If you set the property to **Yes**, the Magic engine continues to search the runtime tree for additional handlers once it has run the current handler's operations.

You can also set the Propagation property using an expression. The expression is evaluated when the execution handler is completed.

Tip:

You can create handlers that stop Internal event propagation to prevent the Magic engine from handling these events. For example, if you want to disable the **Delete** option and display a message when the user tries to delete, you can create a handler on the **Delete Records** Internal event with logic that displays the message, and then set the Propagation property to **No**.

Enable/Disable Handler

When you use a handler's Enable property you can switch the handler on or off during runtime. When you set the Enable property to **Yes**, the handler is active and is executed according to the settings you defined for the handler. If you set this property to **No**, the Magic engine ignores the handler.

When this property is set by an expression, the expression is evaluated when the Magic engine reaches the handler.

Control-specific Handler

You can also condition handler execution by specifying that the handler is only executed when the corresponding event is triggered from a specific form control.

You can limit the handler to a specific control by selecting the control from the Controls list. You access the Controls list by zooming from the Control property, labeled **Ctrl**, in the **Details** column of the handler line.

You can only define a handler as control-specific for a control of the same task.

A handler can only be specific to a control. It cannot be variable-specific. You should place the variable on the form to create a control and then select the control by its name at the handler line. If the control is removed, or the form re-generated, the handler's reference to the control is lost.

> **Note:**
>
> When there are several handlers for the same event in the same task, the Magic engine first searches the control-specific handlers and executes them if they match the control, and only after these handlers are executed does the Magic engine scan for non-control-specific handlers in the same task, again from the bottom of the Task Execution repository upwards.

Chapter

4

Raise Event Operation

You can utilize the Raise Event operation when you use the event-driven architecture in your application.

The Raise Event operation lets you trigger an event from various points in the application. This chapter explains the meaning and usage of the Raise Event operation properties.

General Usage

In general, the Raise Event operation is a request by the developer to execute handlers that are defined for an event at a given time. The handler can either be user-defined or a built-in Magic handler.

Invoking a Magic Internal Handler

A Raise Event operation for an Internal event type may invoke the corresponding internal handling for that event. For example, a Raise Event operation for the **Next Record** Internal event invokes the corresponding internal handler for this event, and the running task tries to move to the next record. Using this technique you can explicitly execute predefined engine handlers to bring about the required state of a task.

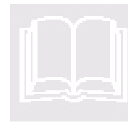
Invoking a User-defined Handler

If you want to execute a set of operations in different parts of the task or application, you do not need to write the same set of operations at every location. You can place this set of operations in a user-defined handler on a user-defined event and raise this event wherever you want to execute this set of operations. In this way you are using predefined logic.

Clarifying Your Logic

Even if the set of operations you want to execute should be executed at a single location or time, it is still preferable to write this set of operations in a handler invoked by a Raise Event operation. This lets you differentiate this set of operations, and your application logic will be more organized and understandable for other developers.

The Raise Event Operation



Like the other 13 Magic eDeveloper operations, the Raise Event operation is a simple operation. The following sections describe the Raise Event operation properties.

Name

The only mandatory Raise Event property is the event it should raise. You should define the event you want to raise by zooming from the **Name** column to open the **Event** dialog box.

This dialog box is similar to the handler dialog box except that it only offers three event types: System, Internal, and User. The other event types, Timer, Expression, and Error are not available since you do not need to raise these events explicitly.

Wait

The **Wait** property determines whether the Raise Event operation should be synchronous or asynchronous.

A synchronous Raise Event operation executes operations defined in the corresponding handlers immediately before the Magic engine continues to the operation preceding the Raise Event operation.

An asynchronous Raise Event operation does not execute the operations of corresponding handlers immediately. The raised event is entered into an event queue maintained by the Magic engine. The Magic engine polls the events in the event queue at defined timings. The corresponding handlers are only executed when the event is polled off the queue. The next chapter deals with the event Polling mechanism.

> **Important:**
>
> Internal event types cannot be raised synchronously. If you want to raise an Internal event, you should make sure that the **Wait** property is set to **No**. If you set this property to **Yes**, internal handling for these events is not executed.

Arguments

The Raise Event operation can pass arguments to handlers that would handle the raised event.

The list of arguments, accessible from the Raise Event **Arg**: field, lets you enter the arguments you want to pass to the handler.

The passed arguments can be received in the handler when you create Select Virtual operations inside the handler's set of operations. The passed arguments are entered into the handler's Select Virtual operations according to their order, so that the first argument is entered into the first virtual, etc.

If the corresponding handler has no Virtual Variables defined, the handler does not receive the passed argument values.

Arguments Passed by Reference

Variables passed as arguments from a synchronous Raise Event operation, where **Wait=Yes**, are updated by any change in the value of the corresponding virtual field of the handler. This is the virtual field that received the argument. If the argument passed is an expression, no new value can be returned.

Arguments Passed by Value

Variables passed as arguments from an asynchronous Raise Event operation, where **Wait=No**, will not be updated by changes in the value of the corresponding virtual field of the handler.

Arguments Passed to a Propagating Handler

Arguments passed to a handler that propagates the event are also passed to the next corresponding handler. If an argument is passed by reference, such as an argument passed as a variable from a synchronous raise event operation, and the first handler updates it, then the next handler receives the updated value. If an argument is passed by value, such as an argument passed as an expression or passed from an asynchronous raise event operation, the next handler always receives the original value of the argument passed by the Raise Event operation.

> Note:
>
> Even if the first handler does not have virtual variables defined to receive the arguments, the next handler can still receive the arguments. You do not have to define the virtual variables in a lower handler if you only need to receive the arguments in the next handler.

Raising Events from Menus and Push Buttons

In addition to the Raise Event operation there are other ways to raise events. You can define push buttons and menu entries to raise events when they are activated.

Push Buttons

The Magic push button control has a Raise Event property. Zoom from this property to select the event you want the push button to raise. The defined event is raised when the push button is activated in runtime. The available event types are: System, Internal, User, and None.

Menu Repository

In the Menu repository you can define a menu entry to raise an event. You can set a menu entry to raise an event by setting its entry type to **Event**. Zoom from the **Menu Params** column in the Event menu entry to select the event you want this menu entry to raise. The available event types are: System, Internal, and User.

> Note:
>
> Activating a push button control or activating an Event menu entry raises the event asynchronously, as though set with **Wait=No**.

Chapter 5

Event Polling

When does the Magic engine handle events?

A handler that corresponds to an event is not necessarily immediately invoked when the event is triggered. The point when a handler is invoked depends on the event type and the way it is triggered.

Immediate Handler Execution

In some cases handlers are executed immediately when the event for which they are defined is triggered.

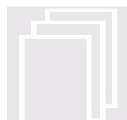
Synchronous Raise Event Operation

A handler whose event is triggered by a synchronous Raise Event operation is invoked immediately when the Raise Event operation is executed.

Error Event

A handler defined to handle an Error event is immediately invoked when the Error event is triggered by the database gateway in use.

Queued Events



Other events enter an event queue when they are triggered. Magic polls the events in this queue at pre-set times. The handlers are invoked when events are polled out of the event queue.

Asynchronous Raise Event Operations

An event triggered by an asynchronous Raise Event operation is entered into the event queue to be handled later on.

System, Internal, Timer, and User Events

All events, other than Error events, are entered into the event queue when they are triggered.

Expression Events

Expression events are also entered into the event queue when they are triggered. However, they are not immediately triggered when the expression evaluates to a **True** value. The expression evaluates each time period for when the keyboard was idle. The **Keyboard idle** time is an environment setting that you set in the Environment dialog box. The expression is evaluated when the **Keyboard idle** time interval passes, and if it evaluates to a **True** value, the event is triggered and enters the event queue.

Polling the Events From the Queue

The point at which the Magic engine polls events from the event queue is affected by the following settings:

- Idle Time
- Batch Event Interval
- Record Event Interval
- Allow Events

Idle Time

In an Online or Browser task, events in the event queue are polled whenever the task is in idle mode, parked on a control. Magic polls all events that have been queued whenever an idle state is reached.

> Important:
>
> Since an asynchronous Raise Event operation is executed at the next idle time, the event may actually be handled in a different location than expected. For example, an asynchronous 'Close' internal Raise Event executed before a Call operation to another task takes effect in the called task and not in the task that raised the event. This is because the next idle time after the event was queued occurs in the called task.

Batch Event Interval

Batch tasks have no idle time. When you run a Batch Task, you can instruct the Magic engine to poll events off the event queue at a given time interval. You define this interval in the **Batch Event Interval** setting in the System tab of the Environment dialog box. If you set the Batch Event Interval to 1000 (milliseconds), the Magic engine polls the events from the event queue every 1000 ms.

When this environment setting is set to zero, no event polling will occur by time interval. This environment setting is general and affects every batch task.

Record Event Interval

You can set your batch task to poll events off the event queue after processing a set amount of records. You define the setting in the **Record Event Interval** property of the Task Control dialog box's Behavior tab. When you set the property to 100, the Magic engine polls the events every time it finishes running through 100 Main Table records. If the setting is 1, the Magic engine polls events on every record.

The value of this property is set by an expression. The expression should evaluate to a number representing the record interval.

> Note:
>
> Magic polls the events off the event queue at both the Batch Event Interval and the Record Event Interval. Whenever one of these intervals is reached, the Magic engine polls the events.

Allow Events

You can set your Batch Task to work without interruption by completely disabling the event polling for that task. To do so you should set the **Allow Events Task** property to **No**.

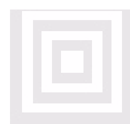
Chapter 6

Control Level Handlers

Magic eDeveloper introduces a new task level called the Control level.

Magic tasks provide handlers for predefined task activities. In previous Magic versions these task handlers were known as: Task Prefix and Task Suffix, Record Prefix and Record Suffix, and Group Prefix and Group Suffix. Magic eDeveloper offers an additional level of handling called the Control level.

Control Level



Unlike the Task and Record handlers, which are automatically defined in every task, you have to manually create Control level handlers. The Control level provides three kinds of handlers: Control Prefix, Control Suffix, and Control Verification.

Control Prefix

The Control Prefix is executed upon entering a control regardless of the tabbing direction. Any operation in this handler is executed just before the task is fully parked on the control.

Control Suffix

The Control Suffix is executed upon exiting a control regardless of the tabbing direction. Any operation in this handler is executed just after the task exits the control and before it parks on the next control.

> **Note:**
>
> Just before the Control Suffix is executed, the Magic engine updates the control variable with the newly entered data and performs the re-computation of values deriving from this variable.

Control Verification

The Control Verification handler is executed in the following circumstances:

- When the user skips from the verified control or preceding control to another record, the control verification is only activated if the record has been modified.
- When a user exits the control, the Control Verification handler is executed just before the Control Suffix handler.
- When the user skips over the verified control, a control that is set with a Control Verification handler, by switching from a control that precedes it to a control that follows it in the same record, and vice versa.
- When the user skips to a control that follows the verified control in another record, the control verification handler of the verified control in the new record is executed.

When exiting a control, the control verification handler is executed just before the Control Suffix, after the newly entered data is updated in the associated variable and once dependent expressions have been re-computed.

> Note:
>
> A Verify operation with an error mode that is executed in a Control Verification handler stops the expected sequence of operations and parks on the verified control.

Control Verification vs. Control Suffix

The main difference between the Control Suffix and the Control Verification handlers is that the Control Suffix is only executed when the user is parked on the handled control and exits that control. However, the Control Verification is executed whenever the user either exits the handled control or passes over it while navigating through other controls and records.

Using the Control Suffix Handler

Use the Control Suffix handler for operations that should only be run when the user is parked on the handled control and exits that control.

Using the Control Verification Handler

Use the Control Verification handler to perform operations whenever the task is requested to pass over the control.

Important:

The Control level handlers must always be control-specific. This means that you must select a control name for a Control level handler.

Tip:

The Control level handlers are executed regardless of the task flow direction. Use the FLOW function to set the handler operations to be activated according to task flow. For more information refer to the FLOW function description in the *Magic Reference Guide, Chapter 9 - Expression Rules.*

Chapter

7

User Events

Magic eDeveloper lets you create your own events.

Magic eDeveloper supports various event types, most of which are already defined. You can handle these events either by creating appropriate handlers or the Magic engine can handle them automatically. This chapter describes User events, how they are defined, and what their properties are.

Constructing a User Event

Like handlers, you can define events for any Magic task. The task events are created and defined in the Task event list, which you can access by pressing **CTRL+K**. Three properties define an event: its name, its trigger and its level of execution.

Description

This field is the event's name. It is not a mandatory field, and you can use and select a User event without a name. However, you should use a descriptive name to differentiate between the various User events.

Trigger

You can set a different event to automatically trigger your User event. You can use the following event types as triggers for a User event: System, Internal, Timer and Expression.

If you do not use a trigger for the User event, set the trigger type to **None**.

Note:
The only way to trigger a User event without assigning a trigger is through the Raise Event operation.

Force Exit

This property defines the task level from which the engine must exit before handling the event.

Force Exit = None

A User event set with **Force Exit=No** does not require the task to exit any level before it is handled, and it can be executed at any level of the task. This kind of event can be triggered synchronously, using the Raise Event operation set with **Wait=Yes**, at any level of the task, such as Task level, Record level, Control level, and a handler.

If such an event is triggered asynchronously, it is handled at the next idle time. Online task or Browser task idle time is the time during which the task is idle or parked on a control, or even within the edit stage of an Edit control.

Force Exit = Control

A User event set with **Force Exit=control** requires the task to exit the control on which it is parked and then handle the event. Once the event handling is complete, the task returns to the same control.

Using Force Exit = Control

Asynchronous events triggered while editing an Edit control are handled within the Edit mode. The new value entered for the Edit control is not yet updated in the associated variable. If a handler invoked in Edit mode refers to the parked variable, it will not reflect the new value. In addition, the entire task is not yet recomputed according to the newly entered value.

You can execute the handler after the newly entered value is updated in the associated variable and the task-dependent values are recomputed accordingly before the handler is executed. To do this you should set the Force Exit property to **Control**.

Force Exit = Record

A User event set with **Force Exit=Record** requires the task to exit the current record. The User event should only perform the Record Suffix if the record has been modified or if the Force Record Suffix task property is set to **Yes**. The User event completes the handling of the event and updates the record in the active transaction. Once this sequence is complete, the task returns to the same record and executes its Record Prefix level.

In other words the handler is executed after the Record Suffix, just before it is updated in the database. Once the handling is complete, the record is updated in the database.

Using Force Exit = Record

You can set the event to Force Exit the record if you want the record to be updated in the current transaction immediately after the event handling is completed.

> **Note:**
>
> When you set the event to Force Exit the record, any update operation performed through the executed handler is also updated when the engine exits the record and updates the record in the transaction.

Using a Synchronous Raise Event Operation

A User event set to Force Exit the Control or Record level, cannot be handled synchronously. This is because the Magic engine cannot perform the required sequence of exiting the requested levels within a running set of operations.

A synchronous Raise Event operation of an event set to Force Exit the Control or Record level ignores the event's Force Exit property and treats it as if it was set to **None**.

Chapter

8

Handler Resources and Information

There are various runtime resources and information available to handlers.

A handler can use and refer to the various runtime resources and information available within the running application. This chapter describes handler access to runtime resources and information.

Runtime Information and Resources

At runtime each task has a defined range of resources and runtime information that it can use. The term *Runtime resources* refers to variables and I/O devices. *Runtime information* refers to all information that can be queried about each active task in the runtime task tree, about the variables, or about the defined I/O devices. A task can refer to and use resources and task information that is not necessarily available during development but is available at runtime.

Variables

You can refer to the variables of all the active runtime tree tasks using their indexed location or name. You can use various functions that query or handle the application variables to refer to all the active task variables. The available variable-related functions are: VARATTR, VARCURRN, VARINDEX, VARMOD, VARNAME, VARPIC, VARPREV, and VARSET.

The Cascading Program Calls example shown in Figure 8-1 shows each program as an independent task in toolkit mode, with the task variables coded as A and B. At runtime the Program A and Program B variables are accessible from Program C. You can refer to the variables by their runtime order, or by the variables' names.

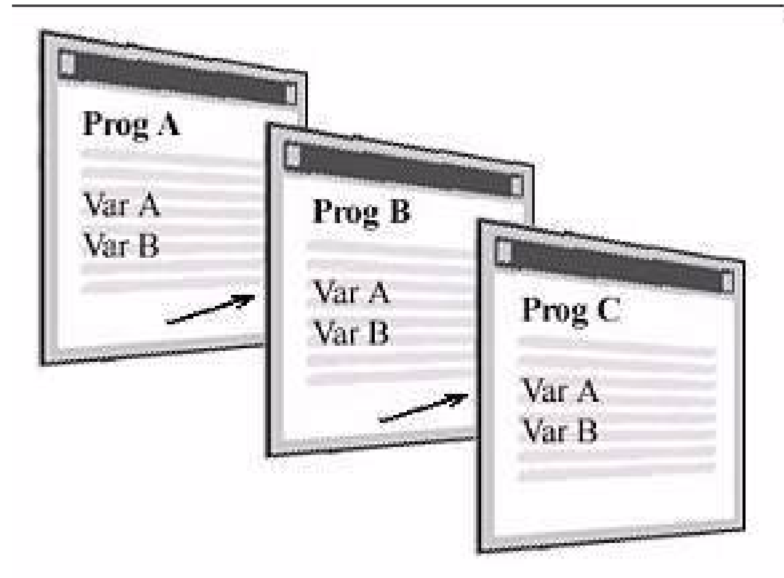


Figure 8-1 This example illustrates Cascading Program Calls. Program A calls Program B, which in turn calls Program C.

I/O Devices

You can use I/O devices defined in programs that are not visible to each other in toolkit mode, and you can output or input forms to a program I/O active at runtime, in other words, a program that is part of the runtime tree.

You can output or input forms to an I/O of an active ancestor task by setting the **I/O name to use** property of the I/O entry in the current task. This specifies the already opened ancestor task I/O device by its name. For more information, refer to the explanation of the **I/O name to use** property in the *Magic Reference Guide, Chapter 7 - Programs*.

Runtime Task Tree

When programs call other programs and subtasks, a task tree is formed in runtime. This is a single-branched tree that begins with the top program and goes through the series of called tasks to the last running task. The last active task can query its ancestor tasks in the runtime tree using a set of functions. The functions are directed to a specific task in the runtime tree by specifying the Task Generation. The Task Generation is a sequential number defined by the task location in the runtime tree, where the generation of the lowest task is 0, the generation of its direct ancestor task is 1, and so on up the tree.

In the Cascading Program Calls example shown above, the generation of Program A is 2, Program B is 1, and the generation of Program C is 0.

The following functions can query task information according to generation: CHEIGHT, CLEFT, CLEFTMDI, COUNTER, CTOP, CTOPMDI, CURROW, CURRPOSITION, CWIDTH, DBCACHE, EOF, EOP, LASTPARK, LEVEL, LINE, PAGE, ROLLBACK, STAT, VARINP, VIEWMOD, WINBOX, WINHWND.

Handler Scope

Often the handler that is activated can be defined in an ancestor task. However, as far as the runtime task tree is concerned, the logic executed in such a high-level handler is regarded as though it runs in the task that triggered the event. This means that even though the logic may be executed by an ancestor task, the entire runtime tree variables, the opened I/O devices in the runtime tree, and accessibility to the runtime tree tasks are available to the handler logic.

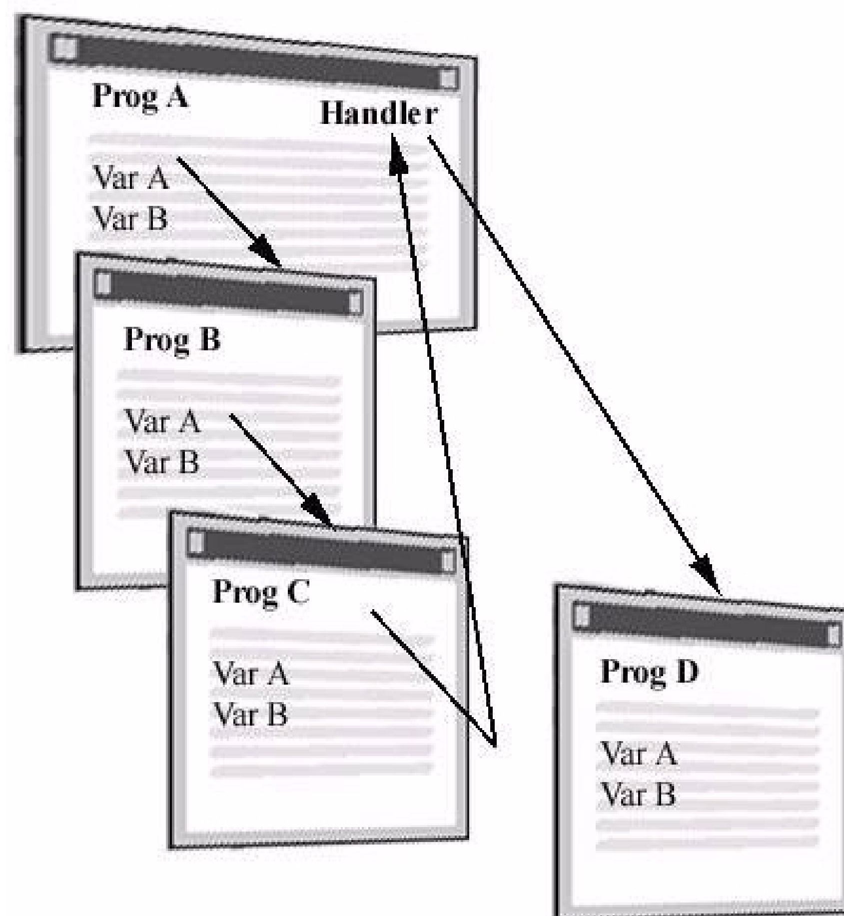


Figure 8-2 This example illustrates cascading program calls using simple call operations and event triggering.

In the example shown above, Program A calls Program B, which in turn calls program C. Program C raises an event that is handled by a handler defined in Program A. This handler then calls Program D.

The runtime tree visible to Program D is the same as if Program D had been called directly from Program C.

Variables

The entire runtime tree variables are available to Program D, starting with the first Program A variable and ending with the last variable of Program D.

Task Tree

The generation of each task is as follows: Program D = 0, Program C = 1, Program B = 2, and Program A = 3.

I/O Devices

Program D can output or input forms to an I/O device opened in any other program in the runtime tree using the **I/O name to use** property of its I/O entry.

> Note:
>
> When you refer to the runtime tree variables and tasks directly from the handler, and not from a task called from the handler, the generation and variable stack is the same as if the handler operations were executed from program C. For example, the generation available to the handler is: Program C = 0, Program B = 1, Program A = 2.

> Interactive Web Applications
>
> Calling a program with a modeless window in a Web application creates a new additional runtime task branch that starts at the top program and continues through the program from which the call operation was executed down to the called program.

Referring to the Triggering Object

Although handlers, and tasks called from the handlers can query the entire runtime tree, you might want to query the object that triggered the event. The object from which an event is triggered can be a variable, control, or task. The ability to refer to the triggering object is necessary for global handlers that need to handle the exact object from which the event was triggered.

THIS() Function

The THIS() function is a simple function used to refer to the triggering object. The triggering object can be the triggering task or the triggering variable. This function is used instead of the variable index in the variable-related functions and instead of the generation in generation-related functions.

The THIS() function requires no arguments and cannot be used in complex expressions, such as THIS()+1.

The Triggering Variable

The triggering variable is the variable on which the task is parked when an event is triggered. You can refer to the triggering variable using the various variable-related functions where the THIS() function is used as the variable index. For example, to get the attribute of the triggering variable you would use the expression: VARATTR(THIS()).

The Triggering Task

The triggering task is the task from which an event is triggered. You can refer to the triggering task using the various generation-related functions where the THIS() function is used as the task generation. For example, to get the level of the triggering task you would use the expression: LEVEL(THIS()).

Handled Control

You can also refer to the handled control from which an event was raised. This is usually used in mouse-related events, such as Click, DblClick, MouseOver, and MouseOut. You retrieve the handled control by its name using the HANDLEDCTRL() function. This function requires no arguments.

Chapter

9

Handling Multi-marked Records

The Magic engine handles events triggered on multi-marked records.

The Magic eDeveloper Table control allows the end-user to mark multiple records that are displayed in the table. You can design your application to perform a defined logic segment to handle the marked records automatically. For example, you can provide the end-user with an option to mark invoice records and activate a predefined event to summarize the total of each invoice and display the calculated sum.

This chapter describes multi-marked record handling. For information on Multi-marking and how to mark records, refer to the *Magic Reference Guide, Chapter 11 - Display Forms.*

Handling Multi-marked Records

When an event is triggered while records are marked in the Table control, the handler corresponding to the triggered event is executed for each record. All marked records are automatically handled one after the other.

From Top to Bottom

Marked records are always handled in the order in which they are listed in the task, from top to bottom. They are not handled in the order in which they were marked.

Full Record Cycle

Before every record is handled, the engine enters each record, performs the Task Prefix, and then the handler is executed. If the record has been updated and set to execute the Record Suffix, the Record Suffix is executed. The Magic engine moves to the next marked record and handles it in the same way. This continues until all marked records are handled.

The Current Parked Record

The task can be parked on a record that need not be either the first marked record, the last marked record, or a part of the marked records.

Before the Magic engine begins to handle the marked records, the engine exits the current parked record. This means that the Magic engine performs the Control Verification handlers and the Record Suffix of the record if the Record Suffix was updated.

When the handling of marked records is finished, the task remains parked on the last marked record.

> **Note:**
>
> The Magic engine parks on the last marked record once the full record cycle is complete. This means that when the last marked record is reached, the following process occurs: Record Prefix, Handler, Record Suffix, and again the Record Prefix to park on the record.

Internal Handling of Multi-marked Records

The only Internal event that handles multi-marked records is the Delete event. You can mark several records and invoke the **Delete Records** Internal event to delete all the marked records.

When triggering an Internal event the Magic engine first clears the record marking and then handles the Internal event.

Triggering Events During the Process

During the process of handling multi-marked records, the Magic engine can handle some other events that are triggered during the process, but it will not handle every event. The Magic engine can only handle events triggered synchronously. Any event triggered asynchronously is ignored during multi-marking.

Supporting Functions

Magic eDeveloper provides you with additional functions that allow you to control the process of handling multi-marked records better.

How Many Records are Marked?

You can use the MMCOUNT function to query the number of records that are currently marked. This function can be directed to a particular task generation, allowing you to query the number of marked records in ancestor tasks.

> **Tip:**
>
> This function returns zero if no record is marked. In this way you can check if records are marked or not.

The Currently Processed Record

You can query which record out of the marked record is handled. The MMCURR function returns the number of the currently processed record from the total of marked records. This function can be directed to a particular task generation, allowing you to query the currently handled marked record in ancestor tasks.

Beginning and End of the Process

The handler executed on multi-marked records is the same handler and the same logic for all marked records. Sometimes you only need to execute part of the logic when the handler is first executed; for example, to reset variables or open a dialog box to confirm the process. At other times you may need to execute only part of the logic in the last handler cycle at the end of the process; for example, to display result information.

You can use the MMCURR and MMCOUNT functions to identify these phases:

- When the MMCURR function returns a value of 1. This means that it is the first cycle and the beginning of the process.
- When the number of currently handled records equals the total number of marked records, in other words when MMCURR equals MMCOUNT. This means that it is the last cycle and the end of the process.

Maintaining Record Marking

When handling of the marked records is complete, the handled records remain marked. This lets the end-user perform several tasks on the same set of marked records. You can use the MMCLEAR function to clear record marking. When this function is evaluated, record marking is cleared after the handling process is completed.

Stopping the Process

You can use the MMSTOP function to stop the automatic process of handling marked records. This function lets you stop the handler from being executed on the remaining marked records.

When the MMSTOP function is evaluated during the process, the function completes the handler for the current running record. However, after this record is handled, the task parks on the record and the process stops. When you use the MMSTOP function to stop the process, the records remain marked.

> **Note:**
>
> Both the MMCLEAR and MMSTOP functions are only relevant for the current task. You cannot stop the process or clear the marking of ancestor tasks.

Chapter

10

Afterword

A few words before you start writing your event-driven applications.

We hope this document provided you with sufficient information about Magic eDeveloper's event-driven Architecture. In this technical overview we explained some of the event-driven Architecture terminology, how you can access the architecture in Magic toolkit mode, and how this architecture is executed at runtime.

We are confident that Magic eDeveloper's event-driven Architecture, together with eDeveloper's other advanced features, provides you with the best development platform for any large-scale application.