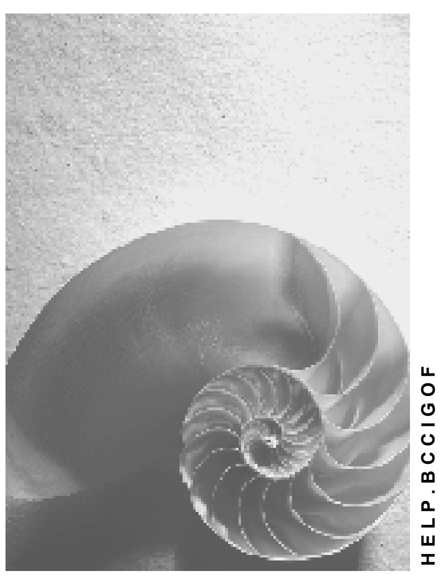
SAP Control Framework



Release 4.6C

Copyright

© Copyright 2001 SAP AG. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP AG. The information contained herein may be changed without prior notice.

Some software products marketed by SAP AG and its distributors contain proprietary software components of other software vendors.

Microsoft®, WINDOWS®, NT®, EXCEL®, Word®, PowerPoint® and SQL Server® are registered trademarks of
Microsoft Corporation.

IBM®, DB2®, OS/2®, DB2/6000®, Parallel Sysplex®, MVS/ESA®, RS/6000®, AIX®, S/390®, AS/400®, OS/390®, and OS/400® are registered trademarks of IBM Corporation.

ORACLE® is a registered trademark of ORACLE Corporation.

INFORMIX®-OnLine for SAP and Informix® Dynamic Server™ are registered trademarks of Informix Software Incorporated.

UNIX®, X/Open®, OSF/1®, and Motif® are registered trademarks of the Open Group.

HTML, DHTML, XML, XHTML are trademarks or registered trademarks of W3C®, World Wide Web Consortium,
Massachusetts Institute of Technology.

JAVA® is a registered trademark of Sun Microsystems, Inc.

JAVASCRIPT® is a registered trademark of Sun Microsystems, Inc., used under license for technology invented and implemented by Netscape.

SAP, SAP Logo, R/2, RIVA, R/3, ABAP, SAP ArchiveLink, SAP Business Workflow, WebFlow, SAP EarlyWatch, BAPI, SAPPHIRE, Management Cockpit, mySAP.com Logo and mySAP.com are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world. All other products mentioned are trademarks or registered trademarks of their respective companies.

Icons

Icon	Meaning
	Caution
	Example
	Note
	Recommendation
	Syntax

Contents

SAP Control Framework 6
Control Framework Architecture 8
Event Handling 10
Registering and Processing Events 13
Context Menu 15
Drag and Drop 17
Process Flow of a Drag and Drop Operation 18
Drag and Drop Events 20
Example of Drag and Drop Programming 22
Drag and Drop in WAN Environments 27
Lifetime Management 28
Automation Queue 30
Synchronizing the Automation Queue 32
Error Handling in Synchronization 34
Automation Queue Services 36
Using Controls in a WAN 42
Creating a Control: SAP Picture Example 44
Methods of Class CL_GUI_CFW 46
dispatch 47
flush 48
get_living_dynpro_controls 49
set_new_ok_code 50
update_view 51
Methods of Class CL_GUI_OBJECT 52
is_valid 53
free 54
Methods of Class CL_GUI_CONTROL 55
constructor 56
finalize 58
set_registered_events 59
get_registered_events 60
is_alive 61
set_alignment 62
set_position 63
set_visible 64
get_focus 65
set_focus 66
get_height 67
get_width 68
Methods of the Class CL_DRAGDROP 69
constructor 70
add 71

clear 73
destroy 74
get 75
get_handle 77
modify 78
remove 80
Methods of the Class CL_DRAGDROPOBJECT 81
set_flavor 82
abort 83

SAP Control Framework

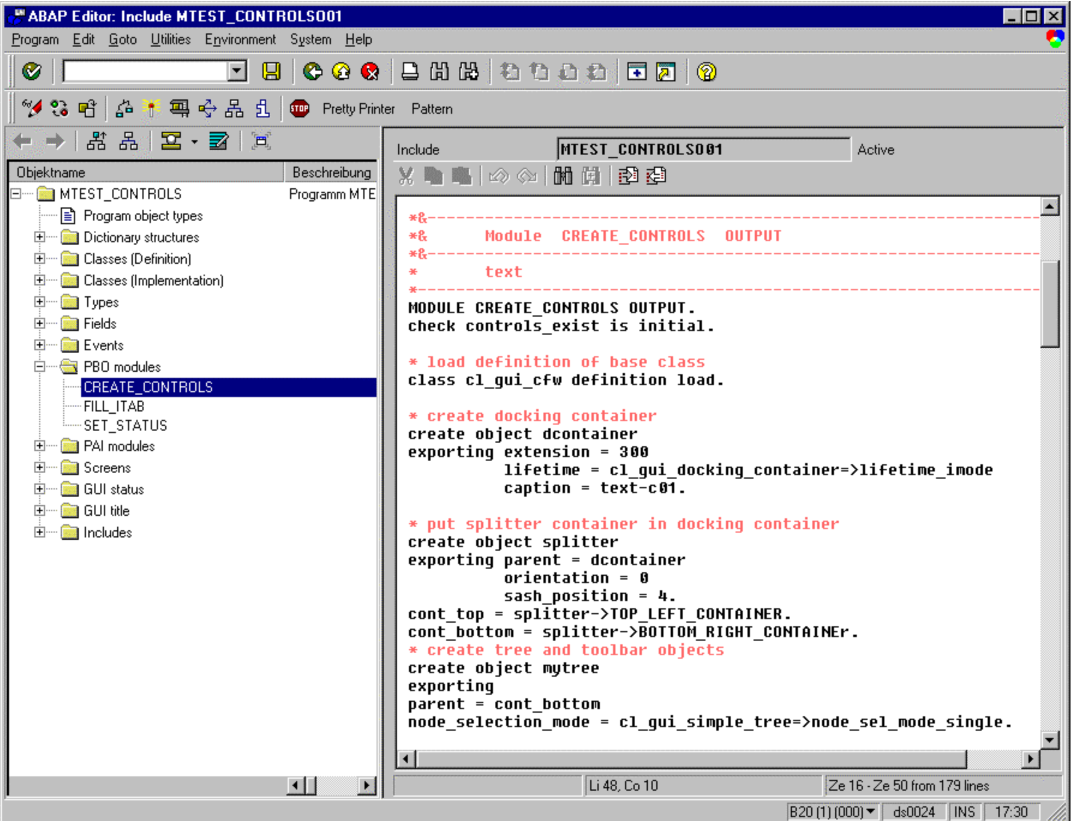
SAP Control Framework

Purpose

In the R/3 System, you can use ABAP to control desktop applications (custom controls). The application logic runs on the R/3 application server (automation client), which drives the custom control (automation server) at the frontend.

The SAPgui (**SAPGUI.APPLICATION**) functions as a container for custom controls at the frontend. A custom control can be either an ActiveX control or a JavaBean.

The following graphic illustrates an SAP Tree Control in use together with an SAP Textedit Control:



Features

The Control Framework supports controls (ActiveX and JavaBeans) that are implemented within the SAPgui.

The Automation Controller communicates with a single control (**SAPGUI.APPLICATION**), which itself is a container for further controls.

The Automation Controller is run from ABAP using the classes **CL_GUI_CFW**, **CL_GUI_OBJECT**, and **CL_GUI_CONTROL**. These allow you to create and destroy custom controls, set and get their attributes, and call their methods.

To assure adequate performance in the client/server environment, the system provides a buffer mechanism called the **automation queue**, which buffers a series of method calls to different instances of custom controls before sending them all to the frontend in a single step.

Events that are triggered in a custom control are processed in two steps:

- Irrelevant events are filtered out.
- Relevant events are forwarded to the application server. An ABAP Objects event is then triggered, returning control to the ABAP program, which can then react to the event.

The lifetime of a control is regulated by the **lifetime management**. Lifetime management automatically destroys controls at the frontend when they are no longer needed. However, you can, of course, destroy a control explicitly in your application program.

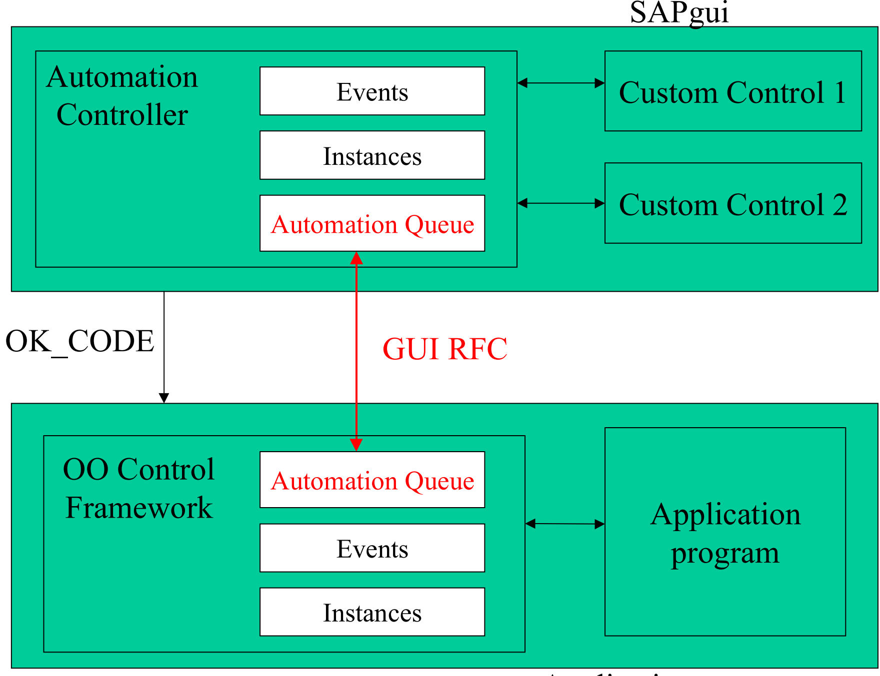
Constraints

Certain methods and events in some of the individual controls are not supported in the SAPGUI for HTML environment. Others are only available in a restricted form. For precise details, refer to the individual control documentation.

Control Framework Architecture

In the R/3 System, you can use ABAP to control desktop applications (custom controls). The application server is the automation client, which drives the custom controls (automation server) at the frontend. The SAPgui (**SAPGUI.APPLICATION**) serves as a container for the custom controls.

The following graphic illustrates the elements and the communication channels between them:



Automation Controller

The automation controller is the central instance at the frontend. It administers all instances of custom controls.

The Automation Controller also contains a list of the events that a custom control can trigger (**see** Event Handling [Page 10]).

All communication between the controls at the frontend and the application program at the backend runs through the Automation Controller.

ABAP Objects Control Framework

The ABAP Objects Control Framework has a similar function at the backend to that of the Automation Controller at the frontend. All method calls from an application program to a custom control run through the Control Framework. In order to avoid each method call establishing a separate connection to the frontend, the method calls are buffered in the automation queue. The automation queue is not sent to the frontend until you reach a synchronization point (**see** Automation Queue [Page 30]).

Like the Automation Controller, the Control Framework has a list of control events. This list also contains the corresponding handler methods that need to be called when the event occurs (**see** Event Handling [Page 10]).

The Control Framework also maintains a list of the control instances you have created. This list is the basis for the lifetime management of controls (**see** Lifetime Management [Page 28]).

Event Handling

Use

In an application program, the user can trigger events in a custom control. Control then returns to the application program, which can react to the events.

Typical events are, for example, double-clicking and dragging and dropping.

Integration

As already mentioned in the Control Framework Architecture [Page 8] section, both the Automation Controller and the ABAP Objects Control Framework administer tables of control events. These tables have to be constructed by the application program. The event table at the frontend contains control instances and events. The event table at the backend also contains the ABAP handler method registered for the events.

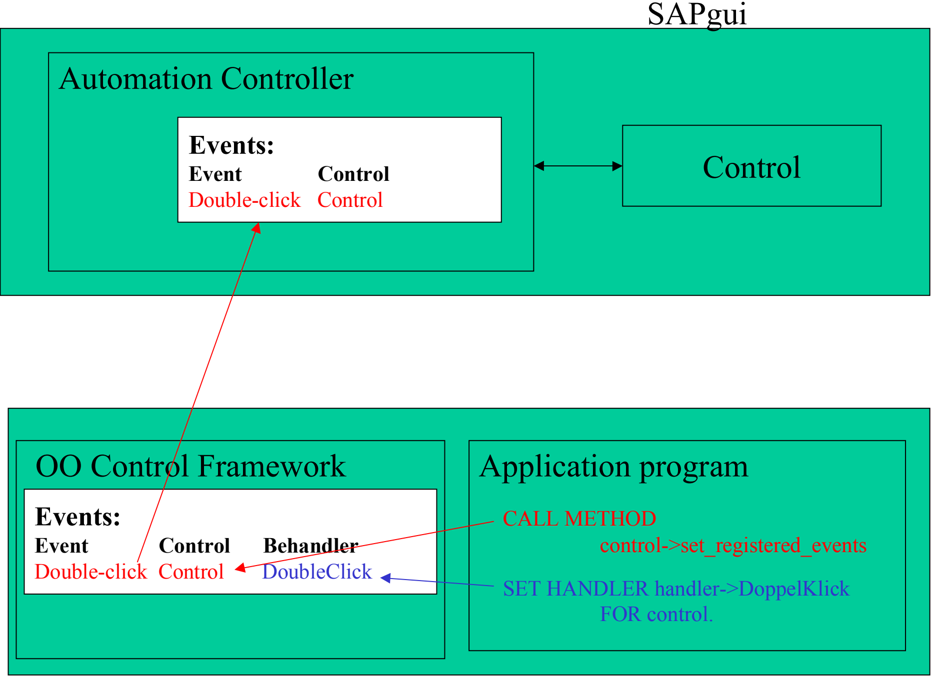
You construct the tables using a special ABAP Objects Control Framework method (**control->set_registered_events**). When you register the event, you must specify whether the event is to be processed as a **system event** or as an **application event**.

- **System event**: A system event is triggered before any automatic field checks (for example, required fields) have taken place on the screen, and before any field transport. The PAI and PBO events are not triggered. Consequently, you cannot access any values that the user has just changed on the screen. Furthermore, there is no field transport back to the screen after the event, so values that you have changed in the event handling are not updated on the screen.

  The handler method that you defined for the event is called automatically by the system. However, you can use the method set_new_ok_code [Page 49] to set a new value for the OK_CODE field. This then triggers the PAI and PBO modules, and you can evaluate the contents of the OK_CODE field as normal in a PAI module.

- **Application event**: This event is processed in the PAI event. Consequently, all field checks and field transport has taken place. If you want the event handler method to be called at a particular point in your application program, you must process the event using the static method **CL_GUI_CFW=>DISPATCH**.

You must set a handler method for the event in your application program using the ABAP statement **SET HANDLER**. You define the handler method in your program as a method of a (local) class. It is up to you whether you define the handler method as an instance method or a static method.



Features

When an event is triggered on a custom control, the Automation Controller checks whether the event is registered in its events table. If the event is not registered, the Automation Controller ignores it. If, on the other hand, it is registered, the Automation Controller generates an **OK_CODE** that it then passes to the ABAP Objects Control Framework.

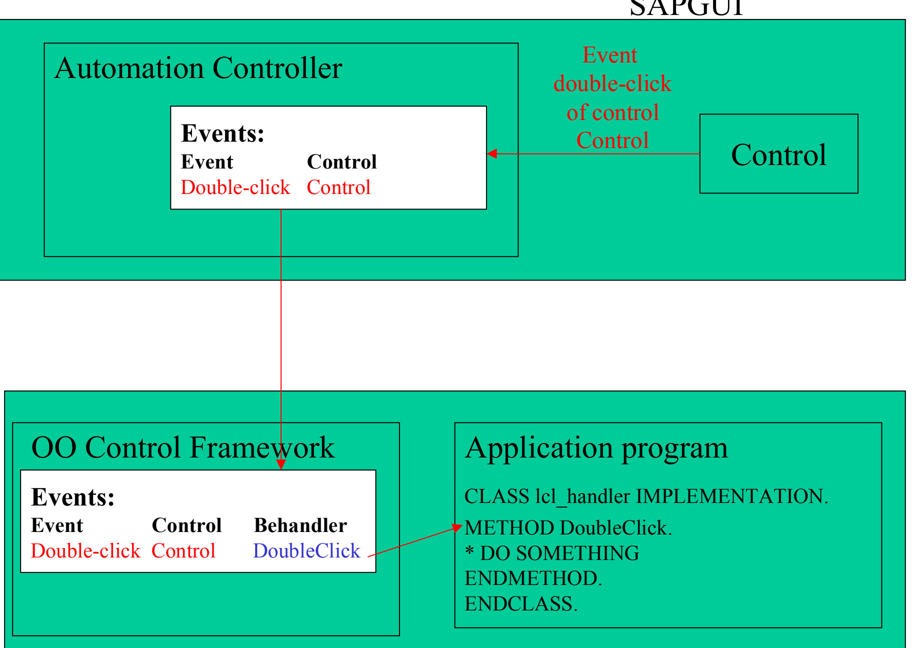
The registered handler method for the event is then called - either directly (for a system event) or when you call the static method **CL_GUI_CFW=>DISPATCH** (for an application event). The handler method receives the event parameter **sender**. This contains the object reference of the control that triggered the event.

To process an application event, you must call the static method **CL_GUI_CFW=>DISPATCH** within a PAI module.

The **OK_CODE** of an event is "spent" after the method **CL_GUI_CFW=>DISPATCH** has been called. Consequently, you cannot trigger the handler method for a second time by calling the dispatch method twice.

You can find out whether the event was successfully passed to a handler method by querying the parameter RC of the method CL_GUI_CFW=>DISPATCH [Page 46].

Event Handling



Registering and Processing Events

Purpose

The event mechanism of the Control Framework allows you to use handler methods in your programs to react to events triggered by the control (for example, a double-click).

Prerequisites

The following description has been generalized to apply to all custom controls. For more information specific to a particular control, refer to that control's documentation.

Process Flow

1. Assume you are working with a custom control that has the ABAP wrapper **cl_gui_xyz**.

```
DATA my_control TYPE REF TO cl_gui_xyz.
```

Registering Events with the Control Framework

2. Define an internal table (type **cntl_simple_events**) and a corresponding work area (type **cntl_simple_event**).

```
DATA events TYPE cntl_simple_events.
DATA wa_events TYPE  cntl_simple_event.
```

3. Now fill the event table with the relevant events. To do this, you need the event ID (**event_id** field). You can find this information in the Class Browser by looking at the attributes of the class **cl_gui_xyz**. You must also decide whether the event is to be a system event (**appl_event = ' '**) or an application event (**appl_event = 'X'**).

```
wa_events-eventid = event_id.
wa_events-appl_event = appl_event.
APPEND wa_events TO events.
```

4. You must now send the event table to the frontend so that it knows which events it has to direct to the backend.

```
CALL METHOD my_control->set_registered_events
        events = events.
```

To react to the events of you custom control, you must now specify a handler method for it. This can be either an instance method or a static method.

Processing an Event Using an Instance Method

5. Define the (local) class definition for the event handler. To do this, specify the name of the handler method (**Event_Handler**). You need to look at the class for the custom control cl_gui_xyz in the Class Browser to find out the name of the event (**event_name**) and its parameters (**event_parameter**). There is also a default event parameter **sender**, which is passed by all events. This contains the reference to the control that triggered the event.

```
CLASS lcl_event_receiver DEFINITION.
PUBLIC SECTION.
METHODS Event_Handler
  FOR EVENT event_name OF cl_gui_xyz
```

```
  IMPORTING event_parameter
          sender.
ENDCLASS.
```

6. Register the handler methods with the ABAP Objects Control Framework for the events.

```
DATA event_receiver TYPE REF TO lcl_event_receiver.
CREATE OBJECT event_receiver.
SET HANDLER event_receiver->Event_Handler
         FOR my_control.
```

Processing an Event Using a Static Method

7. Define the (local) class definition for the event handler. To do this, specify the name of the handler method (**`Event_Handler`**). You need to look at the class for the custom control cl_gui_xyz in the Class Browser to find out the name of the event (**`event_name`**) and its parameters (**`event_parameter`**).

```
CLASS lcl_event_receiver DEFINITION.
PUBLIC SECTION.
CLASS-METHODS Event_Handler
       FOR EVENT event_name OF cl_gui_xyz
       IMPORTING event_parameter
               sender.
ENDCLASS.
```

8. Register the handler methods with the ABAP Objects Control Framework for the events.

```
SET HANDLER lcl_event_receiver=>Event_Handler
         FOR my_control.
```

Processing Control Events

9. You define how you want the system to react to an event in the implementation of the handler method.

```
CLASS lcl_event_receiver IMPLEMENTATION.
METHOD Event_Handler.
* Event processing
ENDMETHOD
ENDCLASS.
```

10. If you registered your event as an application event, you need to process it using the method **`CL_GUI_CFW=>DISPATCH`**. For further information, refer to Event Handling [Page 10].

Context Menu

Context menus are not supported within SAPGUI for HTML.

Use

The context menu (right-hand mouse button or **SHIFT+F10**) allows you to display a context-sensitive menu. The context is defined by the position of the mouse pointer when the user requests the menu.

A context menu allows the user to choose functions that are relevant to the current context.

Features

If you want to provide a context menu in a custom control, you must register the event that is triggered when the user clicks the right mouse button (**context_menu**). The event triggered when the user chooses an entry from the context menu (**context_menu_selected**) is either registered directly by the control wrapper (for example, SAP Tree), or must be registered explicitly (SAP Picture).

If the user requests a context menu for an object, the application program is called using the normal event mechanism [Page 10]. In the event handler method for the **context_menu** event, the program receives a menu reference as an event parameter. The program uses the menu reference to construct the required menu. You can either predefine a context menu using the Menu Painter, or construct one dynamically in your program. The context menu is displayed automatically after the event handler method.

The above description does not apply to the SAP Picture. In the SAP Picture, the menu reference is not passed as an event parameter.

When the user chooses an entry from the context menu, a further event is triggered, which is passed to the application program. Accordingly, you must register a further handler method for this event. Use the handler method to analyze the function code. This is passed to the method as an event parameter.

Activities

- You register for the event **context_menu** and **context_menu_selected** using the set_registered_events [Page 58] method. The identification of the event depends on the control that you are using. This is described in the relevant control documentation.
- You must also define event handler methods for both events in your application.

The SAP Tree is an exception in terms of registering events. In the SAP Tree, you only have to register the event **context_menu**. The event **context_menu_selected** is automatically registered by the control wrapper.

Constructing a Context Menu

When you implement the handler method for the event **context_menu**, you must assign the menu to the control. You may need to check the particular context in which the user requested the context menu.

You construct the context menu using class **CL_CTMENU**. Almost all control wrappers pass a context menu object as an event parameter of the **context_menu** event. If this is not the case (for example, SAP Picture), you must create an object of the class **CL_CTMENU**.

You can use the following methods with the context menu object:

Method	Description
LOAD_GUI_STATUS	Loads a context menu that you have already defined in the Menu Painter (see below)
ADD_FUNCTION	Adds a function
ADD_MENU	Adds a menu that you defined in the Menu Painter
ADD_SUBMENU	Adds a menu that you defined in the Menu Painter as a submenu
ADD_SEPARATOR	Adds a separator
RESET	Reset to initial value
HIDE FUNCTIONS	Hides a function
SHOW_FUNCTIONS	Shows a function
DISABLE_FUNCTIONS	Inactivates a function
ENABLE_FUNCTIONS	Activates a function

The context menu is displayed automatically after the event handler method. The SAP Picture Control and SAP Toolbar Control are an exception to this. With these controls, you must display the context menu explicitly using the method display_context_menu [Ext.].

Evaluating the Function Code

You interpret the user's choice from the context menu in the handler method for the event **context_menu_selected**. You can identify and react to this choice using the function code.

Drag and Drop

Use

Drag and drop allows the user to select an object from one part of a custom control (source) and drop it on another part of a custom control (target). An action occurs in the second part that depends on the object type. Source and target may be either the same control or different controls.

Prerequisites

For a control to support drag and drop, the control wrapper must provide drag and drop events. You must then write handler methods for these events in your program. The events are registered automatically by the relevant control wrapper.

Features

A particular drag and drop behavior is set for each custom control. This behavior may be set globally for all elements of the control (for example, SAP Textedit), or you may be able to define a different behavior for each component (for example SAP Tree). Each behavior consists of one or more descriptions.

A description has the following attributes:

- **DragSrc**: Object is the source of a drag and drop procedure
- **DropTarget**: Object is the target of a drag and drop procedure
- **Flavor**: The flavor describes the type of a drag and drop description. In a drag and drop operation, you can only drop an object onto another if both have at least one common description.
- **Effect**: Specifies whether the drag and drop operations copies or moves the object.
- **Effect_In_Ctrl**: The drop effect used when you copy or move data within the same control.

As soon as a drag event is triggered, you must use the corresponding handler method to find out the affected object.

You must also define the action that is to be carried out on the drop event. The action usually depends on the object that you drop in the control.

If you assign more than one flavor to an object, you must define which flavor is to be used. You do this in the handler for another event.

Once the drop event is finished, you can use a further event to implement additional actions. This is particularly useful for deleting the dropped object from the source after a move operation.

Activities

Whenever you provide a drag and drop function to move objects, you should always provide an *Undo* function as well. You must implement this yourself in the application.

Process Flow of a Drag and Drop Operation

Prerequisites

The following section explains how a drag and drop operation works, examining into the roles of the application server and frontend, and going on to identify the individual steps required to program drag and drop in an application.

Process Flow

Application Server

1. You create the custom control [Page 43].
2. You register the drag and drop events [Page 20].
3. You define the drag and drop behavior for the individual custom controls or their components. To do this, you create an instance [Page 69] of the class CL_DRAGDROP [Page 68]. You then assign one or more flavors [Page 70] to this instance. These describe the drag and drop behavior of the relevant custom control. During the program, you can change [Page 77], delete [Page 79], and query [Page 74] the flavors in your program. You can also initialize [Page 72] or destroy [Page 73] the entire instance.
4. You assign flavors to the custom control using specific methods of the relevant control. For further information, refer to the corresponding control documentation.

Frontend

The following steps are performed by the system at the frontend. They are only listed here so that you can understand what happens during a drag and drop operation.

5. Once the use has selected an object with the left mouse button, the drag and drop service starts.
6. The drag and drop service checks whether a drag and drop behavior has been defined for the object, and whether the object can be dragged (DragSource attribute).
7. If, according to the DragSource attribute, the object can be dragged, the drag and drop operation starts. The mouse pointer then changes automatically.
8. As long as the left mouse button remains pressed, the system continually checks whether the mouse pointer is positioned over an object in a custom control that can receive a dropped object (DropTarget attribute), and whether the flavor of that object is the same as the flavor of the source. If this is the case, the mouse pointer changes again to inform the user.
9. If the user now drops the object, an event is triggered to inform the application server.

This concludes the drag and drop operation for the frontend. However, there has not yet been any change to the contents of the custom control.

Application Server

10. The drag and drop service of the application server creates an instance of the class CL_DRAGDROPOBJECT [Page 80]. You can use this instance (for example,

drag_drop_object) in all events of the drag and drop process as an event parameter. You can use it to find out the context between the events.

11. The drag and drop service checks whether the drag object and drop object have more than one flavor in common. If this is the case, the event **ONGETFLAVOR** is triggered. In the corresponding handler method, you must decide which flavor to use. You do this using the method set_flavor [Page 81].

12. Now, the drag and drop event **ONDRAG** is triggered. It has event parameters that tell you which object the user has dragged. Within the handler routine, you must pass the context (information about the source object) to the instance of the drag and drop data object created in step 9.
    drag_drop_object->object = mydragobject.

13. Next, the **ONDROP** event is triggered. The corresponding handler method serves to process the drag and drop data object. Here, you have to implement the changes that are to be made to the target object based on the drag and drop operation.

14. The last event of the drag and drop operation is **ONDROPCOMPLETE**. This is where you can make your last changes to the drag and drop object. In particular, you should use this event to delete the source object from the DragSource control and the corresponding data structures if you have used the drag and drop operation to move the object.

The Example of Drag and Drop Programming [Page 22] contains an example of a drag and drop operation between a SAP Tree and a SAP Textedit.

Drag and Drop Events

This section only describes those properties of drag and drop events that apply to all controls. The individual control wrappers may augment them. You should therefore consult the relevant control documentation to see if that control has any peculiarities.

Use

There are four standard events in a drag and drop operation at which control is returned to the application program. You use the event handler methods for these events to implement the actions that should be performed during the operation.

Some control wrappers offer additional drag and drop events. For further information, refer to the documentation of the individual controls.

Prerequisites

To be able to react to an event, you must first register it. Unlike normal event handling, you do not register drag and drop events with the Control Framework using the set_registered_events [Page 58] method Instead, they are registered automatically by the wrapper of the control that you are using.

However, you still have to specify handler methods for the events.

```
DATA tree TYPE REF TO cl_gui_simple_tree.
SET HANDLER dragdrop=>on_drag FOR tree.
```

The events are always registered as system events.

Features

In a drag and drop operation, the Control Framework does not pass any events to the application server until the object is dropped. At the application server, it is separated into up to four standard events that can occur within a drag and drop operation, as described in Process Flow of a Drag and Drop Operation [Page 18]. All events have a drag and drop data object as an event parameter. You use this parameter to manage the context of the drag and drop operation. The particular control wrapper that you are using also provides further information about the drag and drop context. For further information, refer to the documentation of the relevant control wrapper.

- **ONGETFLAVOR**: This event is only triggered if the source and target objects have more than one flavor in common. In the handler method, you must then specify which flavor should be used. To do this, use the set_flavor [Page 81] method on the drag and drop object.
  The event is triggered by the target object of the drag and drop operation.
- **ONDRAG**: This event is triggered when the drag and drop operation is complete at the frontend. When you handle this event, you must determine the context of the target object. You then pass this context to the instance of the class **CL_DRAGDROPOBJECT** that you received as an event parameter.
  The event is triggered by the source object of the drag and drop operation.
- **ONDROP**: When you handle this event, you define what should be done to the target object. To do this, use the event parameter for the context that you filled in the **ONDRAG** event. In this event, you must remember the following:

- Within the ONDROP event, you must make a dynamic typecast. You must catch all possible exceptions of the typecast. In the exception handling you must include handling for the case where you try to assign an invalid object. In this case, you must use the abort [Page 82] method to terminate the drag and drop processing.
- You should select the flavor you want to use so that it is possible to assign the drag and drop object to the right TypeCast.

  The event is triggered by the target object of the drag and drop operation.

- **ONDROPCOMPLETE**: Use this event to perform any further processing necessary after the end of the drag and drop operation. For example, this would be necessary following a move operation.
  The event is triggered by the source object of the drag and drop operation.

Example of Drag and Drop Programming

This example program uses a SAP Simple Tree Control and a SAP Textedit Control. The aim is to enable the user to move texts from the tree control into the textedit control.

The example has the program name **RSDEMO_DRAG_DROP_EDIT_TREE**.

```
*&---------------------------------------------------------------------*
*& Report  RSDEMO_DRAG_DROP_EDIT_TREE                                 *&
*---------------------------------------------------------------------*
REPORT  rsdemo_drag_drop_edit_tree    .
DATA ok_code TYPE sy-ucomm.
DATA node_itab LIKE node_str OCCURS 0.
DATA node LIKE node_str.
DATA container TYPE REF TO cl_gui_custom_container.
DATA splitter TYPE REF TO cl_gui_easy_splitter_container.
DATA right TYPE REF TO cl_gui_container.
DATA left  TYPE REF TO cl_gui_container.
DATA editor TYPE REF TO cl_gui_textedit.
DATA tree TYPE REF TO cl_gui_simple_tree.
DATA behaviour_left TYPE REF TO cl_dragdrop.
DATA behaviour_right TYPE REF TO cl_dragdrop.
DATA handle_tree TYPE i.
*---------------------------------------------------------------------*
*       CLASS lcl_treeobject DEFINITION
*  container class for drag object
*---------------------------------------------------------------------*
CLASS lcl_drag_object DEFINITION.
  PUBLIC SECTION.
    DATA text TYPE mtreesnode-text.
ENDCLASS.
*----------------------------------------------------------------------*
*       CLASS dragdrop_receiver DEFINITION
*  event handler class for drag&drop events
*----------------------------------------------------------------------*
CLASS lcl_dragdrop_receiver DEFINITION.
  PUBLIC SECTION.
    METHODS:
       flavor_select FOR EVENT on_get_flavor OF cl_gui_textedit
                     IMPORTING index line pos flavors dragdrop_object,
       left_drag FOR EVENT on_drag OF cl_gui_simple_tree
                     IMPORTING node_key drag_drop_object,
       right_drop FOR EVENT ON_DROP OF cl_gui_textedit
                     IMPORTING index line pos dragdrop_object,
       drop_complete FOR EVENT on_drop_complete OF cl_gui_simple_tree
                     IMPORTING node_key drag_drop_object.
ENDCLASS.
START-OF-SELECTION.
  CALL SCREEN 100.
*&---------------------------------------------------------------------*
*&      Module  START  OUTPUT
*&---------------------------------------------------------------------*
MODULE start OUTPUT.
```

```
  SET PF-STATUS 'BASE'.
  IF container is initial.
    CREATE OBJECT container
        EXPORTING container_name = 'CONTAINER'.
    CREATE OBJECT splitter
        EXPORTING parent = container
                  orientation = 1.
    left = splitter->top_left_container.
    right = splitter->bottom_right_container.
    CREATE OBJECT editor
        EXPORTING parent = right.
    CREATE OBJECT tree
        EXPORTING parent = left
                  node_selection_mode = tree->node_sel_mode_single.
* Definition of drag drop behaviour for tree
    CREATE OBJECT behaviour_left.
    CALL METHOD behaviour_left->add
        EXPORTING
              flavor = 'Tree_move_to_Edit'
              dragsrc = 'X'
              droptarget = ' '
              effect = cl_dragdrop=>copy.
   CALL METHOD behaviour_left->add
        EXPORTING
              flavor = 'Tree_copy_to_Edit'
              dragsrc = 'X'
              droptarget = ' '
              effect = cl_dragdrop=>copy.
    CALL METHOD behaviour_left->get_handle
         IMPORTING handle = handle_tree.
* Drag Drop behaviour of tree control nodes are defined in the node
* structure
    PERFORM fill_tree.
    CALL METHOD tree->add_nodes
         EXPORTING node_table = node_itab
                   table_structure_name = 'NODE_STR'.
* Definition of drag drop behaviour for tree
    CREATE OBJECT behaviour_right.
 CALL METHOD behaviour_right->add
        EXPORTING
              flavor = 'Tree_move_to_Edit'
              dragsrc = ' '
              droptarget = 'X'
              effect = cl_dragdrop=>copy.
 CALL METHOD behaviour_right->add
        EXPORTING
              flavor = 'Tree_copy_to_Edit'
              dragsrc = ' '
              droptarget = 'X'
              effect = cl_dragdrop=>copy.
    CALL METHOD editor->set_dragdrop
         EXPORTING dragdrop = behaviour_right.
```

```
* registration of drag and drop events

    SET HANDLER dragdrop=>flavor_select FOR editor.
    SET HANDLER dragdrop=>left_drag FOR tree.
    SET HANDLER dragdrop=>right_drop FOR editor.
    SET HANDLER dragdrop=>drop_complete for TREE.
  ENDIF.
ENDMODULE.                             " START  OUTPUT
*&---------------------------------------------------------------------*
*&      Module  EXIT  INPUT
*&---------------------------------------------------------------------*
MODULE exit INPUT.
  LEAVE PROGRAM.
ENDMODULE.                             " EXIT  INPUT
*&---------------------------------------------------------------------*
*&      Form  fill_tree
*&---------------------------------------------------------------------*
FORM fill_tree.
  DATA: node LIKE mtreesnode.
  CLEAR node.
  node-node_key = 'Root'.
  node-isfolder = 'X'.
  node-text = 'Text'.
  node-dragdropid = ' '.
  APPEND node TO node_itab.
  CLEAR node.
  node-node_key = 'Child1'.
  node-relatkey = 'Root'.
  node-relatship = cl_gui_simple_tree=>relat_last_child.
  node-text = 'DragDrop Text 1'.
  node-dragdropid = handle_tree.       " handle of behaviour
  APPEND node TO node_itab.
  CLEAR node.
  node-node_key = 'Child2'.
  node-relatkey = 'Root'.
  node-relatship = cl_gui_simple_tree=>relat_last_child.
  node-text = 'DragDrop Text 2'.
  node-dragdropid = handle_tree.       " handle of behaviour
  APPEND node TO node_itab.
ENDFORM.                               " fill_tree
*&---------------------------------------------------------------------*
*&      Module  USER_COMMAND_0100  INPUT
*&---------------------------------------------------------------------*
MODULE user_command_0100 INPUT.
  CALL METHOD cl_gui_cfw=>dispatch.
ENDMODULE.                             " USER_COMMAND_0100  INPUT
*---------------------------------------------------------------------*
*       CLASS DRAGDROP_RECEIVER IMPLEMENTATION
*---------------------------------------------------------------------*
CLASS lcl_dragdrop_receiver IMPLEMENTATION.
  METHOD flavor_select. " set the right flavor
    IF line > 5.
      SEARCH flavors FOR 'Tree_move_to_Edit'.
```

```
      IF sy-subrc = 0.
        CALL METHOD dragDROP_OBJECT->SET_FLAVOR

             EXPORTING newflavor = 'Tree_move_to_Edit'.
      ELSE.
        CALL METHOD dragdrop_object->abort.
      ENDIF.
    ELSE.
      SEARCH flavors FOR 'Tree_copy_to_Edit'.
      IF sy-subrc = 0.
        CALL METHOD dragdrop_object->set_flavor
             EXPORTING newflavor = 'Tree_copy_to_Edit'.
      ELSE.
        CALL METHOD dragdrop_object->abort.
      ENDIF.
    ENDIF.
  ENDMETHOD.
  METHOD left_drag. " define drag object
    DATA drag_object TYPE REF TO lcl_drag_object.
    READ TABLE node_itab WITH KEY node_key = node_key
                         INTO node.
    CREATE OBJECT drag_object.
    drag_object->text = node-text.
    drag_drop_object->object = drag_object.
ENDMETHOD.
  METHOD right_drop. " action in the drop object
    DATA textline(256).
    DATA text_table LIKE STANDARD TABLE OF textline.
    DATA drag_object TYPE REF TO lcl_drag_object.
    CATCH SYSTEM-EXCEPTIONS move_cast_error = 1.
      drag_object ?= dragdrop_object->object.
    ENDCATCH.
    IF sy-subrc = 1.
      " data object has unexpected class
                               " => cancel Drag & Drop operation
      CALL METHOD dragdrop_object->abort.
      EXIT.
    ENDIF.
    CALL METHOD editor->get_text_as_stream
         IMPORTING text       = text_table.
* Synchronize Automation Queue after Get Methods
    CALL METHOD cl_gui_cfw=>flush.
    textline = drag_object->text.
* Insert text in internal table
    INSERT textline INTO text_table INDEX 1.
* Send modified table to frontend
    CALL METHOD editor->set_text_as_stream
         EXPORTING  text = text_table
         EXCEPTIONS error_dp        = 1
                    error_dp_create = 2.
  ENDMETHOD.
  METHOD drop_complete. " do something after drop
    IF drag_drop_object->flavor = 'Tree_move_to_Edit'.
```

```
      CALL METHOD tree->delete_node
           EXPORTING node_key = node_key.
      delete node_itab where node_key = node_key.

    ENDIF.
    ENDMETHOD.
ENDCLASS.
```

Drag and Drop in WAN Environments

In a drag and drop operation, each flavor of each instance of the class **CL_DRAGDROP** brings a communication overhead of between 20 and 70 bytes. As long as you do not have too many instances of the class **CL_DRAGDROP** (<100), this is not a problem. Furthermore, this communication overhead occurs only once.

The Only Rule You Must Observe:

You must not create a separate instance of the class **CL_DRAGDROP** for each drag and drop-enabled object. Instead, all objects with the same behavior should share a single instance.

Lifetime Management

Use

The lifetime management controls the lifetime of a custom control at the frontend. When a control reaches the end of its lifetime, the R/3 System automatically destroys it at the frontend. The methods free [Page 53] and finalize [Page 57] are called by the system for the control. However, you can also destroy a control yourself by calling these methods in your program.

Features

You set the lifetime of a control when you create the control instance. There are two possible settings:

- **`my_control->lifetime_imode`**: The control remains alive for the lifetime of the internal session (that is, until a statement such as **`leave program`** or **`leave to transaction`**) The statements **`set screen 0. leave screen.`** only destroy the internal session if no more screen instances exist (for example, created using **`call screen`**). After this, the finalize [Page 57] method is called.
- **`my_control->lifetime_dynpro`**: The control remains alive for the lifetime of the screen instance, that is, for as long as the screen remains in the stack. After this, the free [Page 53] method is called.

  Using this mode automatically regulates the visibility of the control. Controls are only displayed when the screen on which they were created is active. When other screens are active, the controls are hidden.
- **`my_control->lifetime_default`**: If you create the control in a container, it inherits the lifetime of the container. If you do not create the control in a container (for example, because it is a container itself), the lifetime is set to **`my_control->lifetime_imode`**.

  When you specify the lifetime of a control, it may be shorter, but never longer, than that of its container.

  An instance of a screen is defined as follows:

  An instance is created when a screen is placed on the screen stack (for example, using **`call screen 100 (starting at…),`** or **`call transaction`**), or when it is the defined next screen in a screen sequence (whether set statically or dynamically).

  An instance is destroyed when the next screen is a screen other than that of the current instance (**`set screen 200`** or **`set screen 0`**).

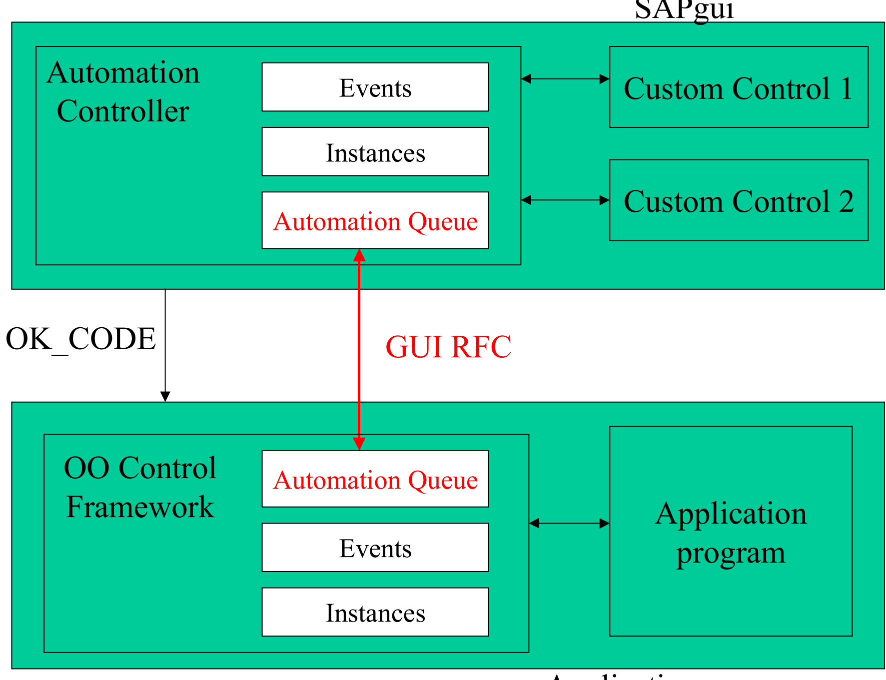
The lifetime of a control is specified in the attribute **`my_control->lifetime`**.

You can use the method is_alive [Page 60] to find out if an instantiated control still exists at the frontend. The method get_living_dynpro_controls [Page 48] returns a list of all controls that currently exist on the frontend.

Automation Queue

Use

Communication between the Automation Controller and the ABAP Objects Control Framework uses GUI RFC calls.



To minimize the network load between the backend and frontend, calls from the backend to the frontend are buffered and sent to the frontend in a single batch at defined synchronization points. A synchronization point occurs when you use a method call that is not buffered or explicitly call the generic synchronization method (`CALL METHOD cl_gui_cfw=>flush`). For further information, refer to Synchronizing the Automation Queue [Page 32].

Communication is based on Remote Function Call. It is synchronous, which means that there is a Remote Function Call at each synchronization point. Due to the architecture of the R/3 System, these RFC calls may not exceed a certain length, otherwise the connection between the presentation server and the application server will be automatically terminated.

Buffering operations improves performance considerably, since every non-buffered operation results opens a new RFC communication with the frontend. However, you should use buffered operations with care, particularly buffered read operations, since mistakes can lead to runtime errors. For further information, refer to Error Handling [Page 34].

Performance Notes

In analyzing performance, you should above all consider the number of synchronization points. In the screen flow logic, the automation queue is always synchronized after the PBO.

However, since you can only handle errors after the synchronization point, you need to strike a balance between optimal performance and ensuring that you handle errors adequately.

If you are working with large quantities of data, you must also be careful that the connection between the application server and presentation server is not terminated due to a timeout. If the connection does time out, you must build additional synchronization points into your application.

For information about tools to support your performance optimization, refer to Automation Queue Services [Page 36].

Prerequisites

There are three kinds of control wrapper methods:

- Methods that always synchronize the automation queue before they end.
- Methods that never synchronize the automation queue. In this case, the programmer is responsible for synchronizing the buffer.
- Methods in which you can specify whether the buffer should be synchronized by passing a parameter value.

Features

Buffered operations are collected in the automation queue. Each internal session has a **single** automation queue for **all** of its custom controls. When you synchronize the automation queue, its contents are passed to the frontend and executed there. The result is then returned to the backend.

Suppose you call a method of the SAP Tree control to set the selected node.

The method places two operations in the automation queue: Op_Tree_1 and Op_Tree_2.

You then call a method of a SAP Textedit Control to display the selected text (without flushing). The method places the operation Op_Textedit_1 in the queue.

The queue now looks like this:

Op_Tree_1

Op_Tree_2

Op_TextEdit_1

If you now synchronize the automation queue, it is transferred to the frontend, and the method calls are executed on the appropriate controls.

Synchronizing the Automation Queue

Purpose

The automation queue plays a central part in communication between the OO Control Framework and the Automation Server. It contains the buffered automation calls, and sends them from the backend to the frontend at special synchronization points. Once the automation queue has been processed at the frontend, the result is sent back to the backend.

The number of synchronization points is critical for the performance of your application. You can use the Automation Trace and the Performance Monitor to track this. Both tools are described in the Automation Queue Services [Page 36] section.

Process Flow

At certain points, the automation queue is automatically synchronized by the system:

- At the end of every PBO event:

The synchronization does not take place until after field transport to the screen. Consequently, the results of method calls that are processed by the automatic synchronization do not appear on the screen.

- After a handler event for a system event [Page 10] has been processed.

You can also synchronize the automation queue manually, using the method flush [Page 47] of class **`CL_GUI_CFW`**.

If you program carefully, you can allow the last explicit synchronization to be carried out implicitly by the system.

Using Buffered Operations to Improve Performance

In general, you can improve the performance of your controls by applying the following processing sequence. Its aim is to split the calls to all existing controls into two groups for each PBO/PAI step:

- Get control attributes
  - Buffered method calls to get all control attributes that you require
  - Synchronize the automation queue
- Processing / calculations
- Set control attributes
  - Buffered method calls to set the control attributes
  - Synchronize the automation queue

As a result of this, you only need two synchronizations for all of your controls. However, you may need to repeat the "Get control attributes" part. For example, if you need a piece of information

before you can decide what other information you require, you would need to get the information in two stages.

Buffered Method Calls to Get Control Attributes

When you use buffered method calls to get control attributes, you must note the following: The addresses of the ABAP variables into which the values are to be written are noted in the automation queue. The values are not passed to the variables until the synchronization. The addresses of the variables must remain valid up to this point. If a local variable no longer exists (for example, a local variable in a subroutine), the synchronization will cause a runtime error. The problem with this kind of error is that it does not show up in the Debugger, even when you select the setting *Automation Controller: Always process requests synchronously*.

Using global variables does not solve the problem either. Firstly, it is not good programming style. Secondly, if you synchronize too late, the application will not be working with up-to-date values.

The safe solution is to query the control attributes in a subroutine that synchronizes the automation queue at the end **and at every exit**.

When you process events, it is a good idea to get the attributes of the control that triggered the event and then to synchronize the automation queue.

(Pseudosyntax): Suppose you want to read the selected node of a tree control and the selected text of a textedit control.

```
FORM GET_CONTROL_PROPERTIES.
        DATA: tree_selected_node, combobox_selected_node.
        CALL METHOD tree->GET_SELECTED_NODE
                IMPORTING
                        NODE_KEY   = tree_selected_node
    <Error handling>
CALL METHOD textedit-> GET_SELECTION_POS
                IMPORTING
                        FROM_LINE = from_line
        FROM_POS  = from_pos
        TO_LINE   = to_line
        TO_POS    = to_pos.
<Error handling>
        CALL METHOD CL_GUI_CFW=>FLUSH
<Error handling>
ENDFORM.
```

Error Handling in Synchronization

Purpose

You cannot analyze an error in an automation call until after the synchronization point. The following example illustrates the problems that this can cause:

1. Suppose you call the methods **set_registered_events**, **add_column**, **add_nodes_and_items**, and **expand_nodes** one after the other.
   The method call for **add_nodes_and_items** contains an error.
2. Now you synchronize the automation queue using the method **cl_gui_cfw=>flush**.
   This sends the automation queue to the frontend and processes it.
3. The first two methods are processed with no problems.
4. However, in the third method, an error occurs. Once the error occurs, the automation queue processing is terminated, and an error is returned to the backend.
5. The exception **cntl_error** of method **cl_gui_cfw=>flush** is triggered.
   Consequently, you cannot immediately identify the method in which the error occurred.

In this case, you should use the Debugger [Page 36] and select the setting *Automation Controller: Always process requests synchronously*. You will then be able to see that the error is triggered in the method **add_nodes_and_items**.

This is particularly critical when the system synchronizes the automation queue. To allow you to handle the error in your program, the system triggers the system event **flush_error**. You should always register this event. If you do not, an error log is displayed.

Process Flow

To register the event **flush_error**, you must:

1. Create a (local) class for event handling.
2. Define an event handler method for the event **flush_error** of class **cl_gui_cfw**. The event returns information about the context in which the error occurred.
   Example (implemented as a static method):

```
*----------------------------------------*
*      CLASS lcl_event_handler DEFINITION
*----------------------------------------*
CLASS lcl_event_handler DEFINITION.
  PUBLIC SECTION.
    class-METHODS
      error FOR EVENT flush_error OF cl_gui_cfw
      IMPORTING DYNPRO_PROGRAM DYNPRO_NUMBER SITUATION.
```

ENDCLASS.

Event Parameters

Parameters	Description
DYNPRO_PROGRAM	Name of the program in which the error occurred
DYNPRO_NUMBER	Number of the screen on which the error occurred
SITUATION	Error in automation queue synchronization triggered by the system occurred in: **cl_gui_cfw=>flush_situation_pbo**: Synchronization after the PBO **cl_gui_cfw=>flush_situation_system_events**: Synchronization after a system event

3. Implement the event handler method. In it, you should analyze the context information:

```
*---------------------------------------*
*      CLASS lcl_event_handler IMPLEMENTATION
*- -------------------------------------*
CLASS lcl_event_handler IMPLEMENTATION.
  METHOD  error.
<do something>
  ENDMETHOD.
ENDCLASS.
```

4. Register the event at the OO Control Framework:

```
SET HANDLER lcl_event_handler=>error.
```

Once the event has been triggered, you should never try to continue working with the controls, since further errors could occur. You should therefore end your program with a controlled termination.

Automation Queue Services

Use

The following services are available to help you use controls in your applications:

Debugger: For identifying errors

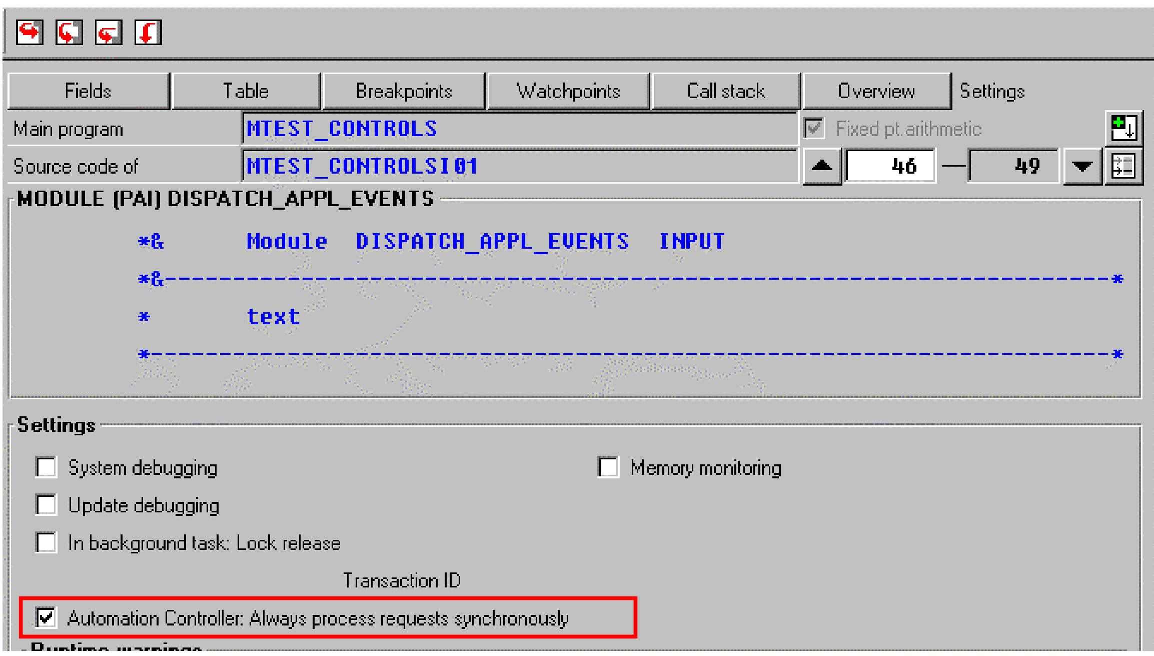
Performance display: Performance optimization

Automation Trace: Finding synchronization points

Features

Debugger

If you use buffered operations on your controls, an error in a method call will not become visible until you synchronize the automation queue. Therefore, when you are debugging a program, it makes sense to synchronize the automation queue after every method call. You can do this by selecting the option *Automation Controller: Always process requests synchronously* in the Debugger. This calls the generic method **`CALL METHOD cl_gui_cfw=>flush`** after every automation method.



If the error no longer occurs when the method is called directly, you have called the method **CL_GUI_CFW=>FLUSH** in the wrong place in your program.

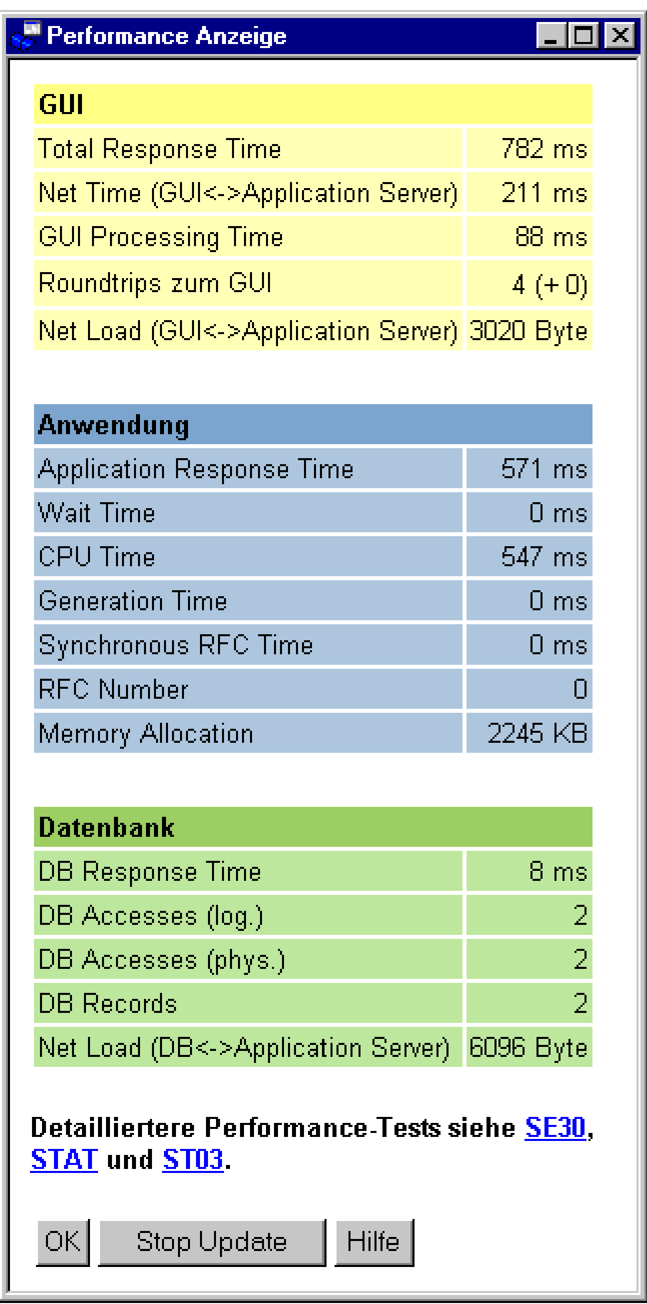
Include **error handling** after every method call (query the value of **SY-SUBRC**). Remember, however, that the error handling is normally only processed when you are debugging.

Performance Display

There are two ways in which you can check the number of round trips within your program:

- Switch on the round trip display in the status bar:

- Activate the performance display. (Choose *System* → *Utilities* → *Performance display*.)

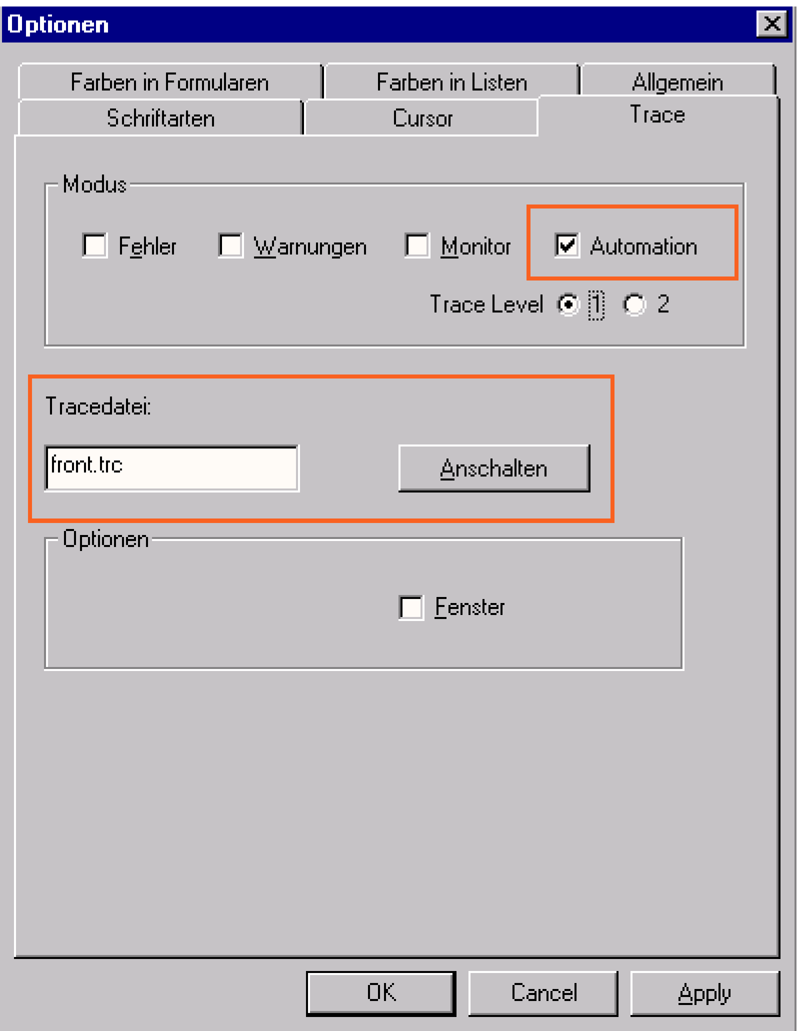


Automation Trace

You can create a trace for the automation queue. To do this, select *Automation* in the *Trace* group of the SAPgui settings. Now, all automation queue calls and their parameters (create object, call method, set/get property, free object) are logged in a trace file.

If an error occurs, it is logged in the trace file (**`HRESULT error_code`**).

You can also see how many flushes are required in each PBO/PAI round trip and eliminate those that are redundant.



When you analyze the method calls to the control using the trace, remember that the ABAP methods often do not correspond to the method names in the trace. This is because the method names in the trace are those of the automation calls to the control. Remember, too, that one method call in ABAP may lead to more than one automation method being called.

The method call to load an image into the SAP Picture has the following form in an ABAP program:

```
CALL METHOD picture->load_picture_from_url
     EXPORTING  url = 'http://www.sap-ag.de/germany/images/sapag.gif'
```

In the Automation Trace, the following would appear:

Trace
Automation: SAPAWOLE CALL METHOD OF 5 'loadImageFromURL'
Automation: Parameter #01 = "http://www.sap-ag.de/germany/images/sapag.gif", Type = STRING (VT_BSTR)
Automation: elapse time for call = 2.733 sec

Using Controls in a WAN

When you use controls in your programs, you place an extra load on the communication channel between the frontend and backend. In a LAN, and particularly in a WAN environment, this can be a critical factor.

The problem is alleviated somewhat by buffering mechanisms (see also Automation Queue [Page 30]). Use these points as a guideline to using controls in a WAN.

The documentation for the individual controls also contains more specific notes about using that control in a WAN.

Using CL_GUI_CFW=>FLUSH

The method CL_GUI_CFW=>FLUSH [Page 47] synchronizes the automation queue and the ABAP variables in it. Calling it often generates a synchronous RFC call from the application server to the frontend. To optimize the performance of your application, you should call this method as little as possible.

It is often a good idea to read all control attributes in a single automation queue (for example, at the beginning of the PAI) and retrieve them in a single synchronization. You should, in particular, do this when you read attributes that are not necessary in your event handlers or the PAI/PBO cycle.

You do not need to include a "safety flush" at the end of the PBO to ensure that all method calls are transported to the frontend. A flush at the end of the PBO is guaranteed. Consequently, you cannot construct an automation queue spread over several screens.

There is no guarantee that an automation queue will be sent when you call `CL_GUI_CFW=>FLUSH`. The queue recognizes whether it contains any return values. If this is not the case, it is not sent.

If you have a queue with no return values, and want to ensure that it is synchronized, you can use the Control Framework method CL_GUI_CFW=>UPDATE_VIEW [Page 50]. You should only use this method if you absolutely need to update the GUI. For example, you might have a long-running application in which you want to provide the user with regular updates on the status of an action.

After you have read the attributes of a control, the contents of the corresponding ABAP variables are not guaranteed until after the next flush. The contents of the ABAP variables remain undefined until this call occurs. In the future, there will be cases in which this flush is unnecessary. They will be recognized by the automation queue and the corresponding flush call will be ignored.

Creating Controls and Passing Data

Creating controls and passing data to them is normally a one-off procedure, which in comparison to using normal screen elements can be very runtime-intensive. You should therefore not use any unnecessary controls, or pass unnecessary data to the controls that you are using.

A typical example is a tabstrip control with several tab pages. If the pages contain controls, you should consider using application server scrolling instead of local scrolling, and not loading the controls until the corresponding page is activated by the user. The same applies to passing data to the controls on tab pages.

If you want to differentiate between LAN and WAN environments when you pass data to a control, you can use the function module **`SAPGUI_GET_WANFLAG`**. In some applications, you may

need to pass different amounts of data or use a complete fallback in a WAN application. The environment affects, for example, the number of same-level nodes that you can transfer to a tree control without having to introduce artificial intermediate levels.

Unlike screen elements, controls only have to be created and filled with data once. From a performance point of view, this means that they become more profitable the longer they exist. In applications that are called repeatedly, and therefore initialized repeatedly, controls can have a negative effect on performance. In applications that use the same screen for a long time, on the other hand, you may find that using controls results in improved performance.

You can always use the performance tools [Page 36] to check the advantages and disadvantages in terms of network load that using a control brings.

Storing Documents, Picture, and Other Data

Release 4.6A sees the introduction of a frontend cache for accessing documents from the Business Document Service (BDS). You are strongly recommended to store desktop documents, images, and other data in the BDS and not in the R/3 database. Documents from the BDS can be cached at the frontend, and therefore only have to be loaded over the network once.

Creating a Control: SAP Picture Example

Prerequisites

The following process applies to all SAP custom controls. The programming examples use the SAP Picture Control. However, to apply the example to other controls, you would only have to change the name of the control class.

The example also assumes that you are using the custom control in a Custom Container. The SAP Container documentation contains details of further scenarios.

Process Flow

Create the Instance

1. Define a reference variable for the Custom Container in which you want to place the custom control (**see** SAP Container [Ext.]).

DATA container TYPE REF TO cl_gui_custom_container.

2. Define a reference variable for the SAP Picture:

DATA picture TYPE REF TO cl_gui_picture.

3. Create the Custom Container. You must already have created the area **'CUSTOM'** for the Custom Container in the Screen Painter. When you create the container, you must also specify its lifetime [Page 28] (see constructor [Page 55]).

```
CREATE OBJECT container
   EXPORTING container_name = 'CUSTOM'
             lifetime       = lifetime.
```

4. Create the SAP Picture Control. You can also specify a lifetime for the SAP Picture, but it must not be longer than that of its container.

```
CREATE OBJECT picture
    EXPORTING parent  = container
              lifetime = lifetime.
```

Register the Events

5. There are three steps: Registering the events with the Control Framework, defining a handler method, and registering the hander method. These steps are explained under Registering and Processing Events [Page 13].

Use the Control

6. These steps are control-specific and therefore not described here.

Destroy the Control

The lifetime management [Page 28] is normally responsible for destroying any controls you use. However, the following two steps allow you to destroy the control yourself:

7. Use the method free [Page 53] to destroy the Custom Control at the frontend. If you no longer need the control container, release it as well:

```
CALL METHOD picture->free
    EXCEPTIONS cntl_error        = 1
               cntl_system_error = 2.
CALL METHOD container->free
    EXCEPTIONS cntl_error        = 1
               cntl_system_error = 2.
```

Pay careful attention to the sequence in which you destroy controls at the frontend. When you destroy a container, all controls in it are automatically destroyed as well. If you have already destroyed a control and try to destroy it again, an error occurs. You can check whether a control has already been destroyed using the method is_alive [Page 60].

8. Delete the reference variables to the custom control and the control container.

```
FREE PICTURE.
FREE CONTAINER.
```

Methods of Class CL_GUI_CFW

The class **CL_GUI_CFW** contains static methods that apply to all instantiated custom controls when you call them.

dispatch

Use this method to dispatch application events (**see** Event Handling [Page 10]) to the event handlers registered for the events. If you do not call the method within the PAI event of your application program, it is called automatically by the system after the PAI has been processed. The method returns a return code from which you can tell if the call was successful.

```
CALL METHOD cl_gui_cfw=>dispatch
     IMPORTING return_code = return_code.
```

Parameters	Description
return_code	**cl_gui_cfw=>rc_found**: The event was successfully directed to a handler method. **cl_gui_cfw=>rc_unknown**: The event was not registered in the event list. **cl_gui_cfw=>rc_noevent**: No event was triggered in a control. The function code was therefore a normal one (for example, from a menu entry). **cl_gui_cfw=>rc_nodispatch**: No handler method could be assigned to the event.

An event can only be dispatched once. After that, it is "spent". Consequently, attempting to dispatch the events a second time does not trigger the handler events again.

flush

Use this method to synchronize the automation queue [Page 30]. The buffered operations are sent to the frontend using GUI RFC. At the frontend, the automation queue is processed in the sequence in which you filled it.

If an error occurs, an exception is triggered. You must catch and handle this error. Since it is not possible to identify the cause of the error from the exception itself, there are tools available in the Debugger and the SAPgui to enable you to do so.

Debugger: Select the option *Automation Controller: Always process requests synchronously*. The system then automatically calls the method **`cl_gui_cfw=>flush`** after each method called by the Automation Controller.

SAPGUI: In the SAPgui settings, under *Trace*, select *Automation*. The communication between the application server and the Automation Controller is then logged in a trace file that you can analyze at a later date.

```
CALL METHOD cl_gui_cfw=>flush
          EXCEPTIONS CNTL_SYSTEM_ERROR = 1
                CNTL_ERROR = 2.
```

Do not use any more synchronizations in your program than are really necessary. Each synchronization opens a new RFC connection to the SAPgui.

get_living_dynpro_controls

This method returns a list of reference variables to all active custom controls.

```
CALL METHOD cl_gui_cfw=>get_living_dynpro_controls
                  IMPORTING control_list = control_list.
```

Parameters	Description
control_list	List of reference variables of active custom controls. The list has the type **CNTO_CONTROL_LIST** (defined in class **CL_GUI_CFW**).

set_new_ok_code

You may only use this method in the handler method of a system event. It sets an **`OK_CODE`** that triggers PAI processing. This means that data is transferred from the screen to the program, and you can take control of the program in your PAI modules.

```
CALL METHOD cl_gui_cfw=>set_new_ok_code
        EXPORTING new_code = new_code
        IMPORTING       rc = rc.
```

Parameters	Description
new_code	Function code that you want to place in the **`OK_CODE`** field (**`SY-UCOMM`**).
return_code	**`cl_gui_cfw=>rc_posted`**: The OK_CODE was set successfully and the automatic field checks and PAI will be triggered after the event handler method has finished. **`cl_gui_cfw=>rc_wrong_state`**: The method was not called from the handler method of a system event. **`cl_gui_cfw=>rc_invalid`**: The **`OK_CODE`** that you set is invalid.

update_view

Calling the flush [Page 47] method only updates the automation queue if the queue contains return values.

If you have a queue with no return values, and want to ensure that it is synchronized, you can use the Control Framework method **CL_GUI_CFW=>UPDATE_VIEW**. You should only use this method if you absolutely need to update the GUI. For example, you might have a long-running application in which you want to provide the user with regular updates on the status of an action.

```
CALL METHOD cl_gui_cfw=>update_view
         EXCEPTIONS CNTL_SYSTEM_ERROR = 1
               CNTL_ERROR        = 2.
```

Methods of Class CL_GUI_OBJECT

The class **CL_GUI_OBJECT** contains important methods for custom control wrappers. The only one relevant for application programs is the is_valid [Page 52] method.

is_valid

This method informs you whether a custom control for an object reference still exists at the frontend.

```
CALL METHOD my_control->is_valid
         IMPORTING result = result.
```

Parameters	Description
result	0: Custom control is no longer active at the frontend 1: Custom control is still active

free

Use this method to destroy a custom control at the frontend. Once you have called this method, you should also initialize the object reference (**`FREE my_control`**).

```
CALL METHOD my_control->free
    EXCEPTIONS cntl_error        = 1
               cntl_system_error = 2.
```

Methods of Class CL_GUI_CONTROL

The class **CL_GUI_CONTROL** contains methods that you need to set control attributes (for example, displaying the control), register events, and destroy controls.

constructor

This method is called by the control wrapper when you instantiate a control.

To instantiate a SAP control, always call the constructor of its class.

```
CREATE OBJECT my_control
  EXPORTING   clsid          = clsid
          lifetime        = lifetime
          shellstyle       = shellstyle
          parent          = parent
          autoalign        = autoalign
  EXCEPTIONS cntl_error        = 1
          cntl_system_error = 2
          create_error      = 3
          lifetime_error    = 4.
```

Parameters	Description
clsid	ID of the class
lifetime	Lifetime management parameter. The following values are permitted: **my_control->lifetime_imode**: The control remains alive for the duration of the internal session (that is, until the session is ended by one of the following statements: **leave program. leave to transaction. set screen 0, leave screen.**). After this, the finalize [Page 57] method is called. **my_control->lifetime_dynpro**: The control remains alive for the lifetime of the screen instance, that is, for as long as the screen remains in the stack. After this, the free [Page 53] method is called. Using this mode automatically regulates the visibility of the control. Controls are only displayed when the screen on which they were created is active. When other screens are active, the controls are hidden. **my_control->lifetime_default**: If you create the control in a container, it inherits the lifetime of the container. If you do not create the control in a container (for example, because it is a container itself), the lifetime is set to **my_control->lifetime_imode**.
Shellstyle	Controls the appearance and behavior of the control You can pass any constants from the ABAP include <**CTLDEF**> that begin with WS. You can combine styles by adding the constants together. The default value sets a suitable combination of style constants internally.
parent	Container in which the SAP Picture Control can be displayed (**see also** SAP Container [Ext.]).
autoalign	' ': Control is not automatically aligned 'X': Control is automatically aligned. This uses the maximum available space within a container.

finalize

This method is redefined by the relevant control wrapper. It contains specific functions for destroying the corresponding control. This method is called automatically by the free [Page 53] method, before the control is destroyed at the frontend.

```
CALL METHOD my_control->finalize.
```

set_registered_events

Use this method to register the events of the control. **See also:** Event Handling [Page 10]

```
CALL METHOD my_control->set_registered_events
     EXPORTING  events           = events
     EXCEPTIONS cntl_error       = 1
            cntl_system_error = 2
       illegal_event_combination = 3.
```

Parameters	Description
events	Table of events that you want to register for the custom control **`my_control`**.

The table **`events`** is a list of the events that you want to register. It is defined with reference to table type **`CNTL_SIMPLE_EVENTS`**. The table type is based on the structure **`CNTL_SIMPLE_EVENT`**, which consists of the following fields:

Field	Description
EVENTID	Event name
APPL_EVENT	Indicates whether the event is a system event (initial) or an application event (X).

The values that you assign to the field **`EVENTID`** are control-specific and therefore described in the documentation of the individual controls.

get_registered_events

This method returns a list of all events registered for custom control **my_control**.

```
CALL METHOD my_control->get_registered_events
     IMPORTING  events      = events
     EXCEPTIONS cntl_error = 1.
```

Parameters	Description
events	Table of events that you want to register for the custom control **my_control**.

The table **events** is a list of the events that you want to register. It is defined with reference to table type **CNTL_SIMPLE_EVENTS**. The table type is based on the structure **CNTL_SIMPLE_EVENT**, which consists of the following fields:

Field	Description
EVENTID	Event name
APPL_EVENT	Indicates whether the event is a system event (initial) or an application event (X).

The values that you assign to the field **EVENTID** are control-specific and therefore described in the documentation of the individual controls.

For general information about event handling, refer to the Event Handling [Page 10] section of the SAP Control Framework documentation.

is_alive

This method informs you whether a custom control for an object reference still exists at the frontend.

```
CALL METHOD my_control->is_alive
     RETURNING state = state.
```

Parameters	Description
state	**my_control->state_dead**: Custom control is no longer active at the frontend **my_control->state_alive**: Custom control is active on the current screen. **my_control->state_alive_on_other_dynpro**: Custom control is not active on the current screen, but is still active (but invisible) at the frontend.

set_alignment

Use this method to align the custom control within its container:

```
CALL METHOD my_control->set_alignment
     EXPORTING  alignment        = alignment
     EXCEPTIONS cntl_error       = 1
                cntl_system_error = 2.
```

Parameters	Description
alignment	Control alignment

The `alignment` parameter may consist of combinations of the following alignments:

Name	Description
my_control->align_at_left	Alignment with left-hand edge
my_control->align_at_right	Alignment with right-hand edge
my_control->align_at_top	Alignment with top edge
my_control->align_at_bottom	Alignment with bottom edge

You can combine these parameters by adding the components:

alignment = my_control->align_at_left + my_control->align_at_top.

set_position

Use this method to place the control at a particular position on the screen.

The position of the control is usually determined by its container.

```
CALL METHOD my_control->set_position
     EXPORTING  height        = height
            left          = left
            top           = top
            width         = width
     EXCEPTIONS cntl_error        = 1
            cntl_system_error = 2.
```

Parameters	Description
height	Height of the control
left	Left-hand edge of the control
top	Top edge of the control
width	Width of the control

set_visible

Use this method to change the visibility of a custom control.

```
CALL METHOD my_control->set_visible
     EXPORTING  visible          = visible
     EXCEPTIONS cntl_error       = 1
                cntl_system_error = 2.
```

Parameters	Description
visible	X: Custom control is visible ' ': Custom control is not visible

get_focus

This static method returns the object reference of the control that has the focus.

```
CALL METHOD cl_gui_control=>get_focus
    IMPORTING  control          = control
    EXCEPTIONS cntl_error       = 1
           cntl_system_error = 2.
```

Parameters	Description
control	Object reference (**TYPE REF TO cl_gui_control**) to the control that has the focus.

set_focus

Use this static method to set the focus to a custom control.

```
CALL METHOD cl_gui_control=>set_focus
    EXPORTING  control          = control
    EXCEPTIONS cntl_error       = 1
               cntl_system_error = 2.
```

Parameters	Description
control	Object reference (`TYPE REF TO cl_gui_control`) to the control on which you want to set the focus.

get_height

This method returns the height of the control.

```
CALL METHOD control->get_height
    IMPORTING  height          = height
    EXCEPTIONS cntl_error       = 1.
```

Parameters	Description
height	Current height of the control

get_width

This method returns the width of the control.

```
CALL METHOD control->get_width
    IMPORTING  width           = width
    EXCEPTIONS cntl_error       = 1.
```

Parameters	Description
width	Current width of the control

Methods of the Class CL_DRAGDROP

The class **CL_DRAGDROP** contains methods that describe the drag and drop [Page 17] behavior of a custom control.

constructor

The constructor creates an instance for the description of the drag and drop behavior of a control.

```
CREATE OBJECT dragdrop.
```

add

This method adds a new description to the drag and drop behavior. You can store any number of descriptions, but you may not add the same description more than once.

```
CALL METHOD dragdrop->add
    EXPORTING flavor       = flavor
              dragsrc      = dragsrc
              droptarget   = droptarget
              effect       = effect
              effect_in_ctrl = effect_in_ctrl
    EXCEPTIONS already_defined  = 1
               obj_invalid      = 2.
```

Parameters	Description
flavor	Description of the new flavor
dragsrc	'**X**': The description is a drag source
droptarget	'**X**': The description is a drop target
effect	Drop effect of the description between different custom controls. The following effects are supported: **dragdrop->copy**: Appearance of the mouse when using drag and drop to copy. **dragdrop->move**: Appearance of the mouse when using drag and drop to move. **dragdrop->none**: Drag and drop is not possible.
effect_in_ctrl	Drop effect of the description in the same custom control. The following effects are supported: **dragdrop->copy**: Appearance of the mouse when using drag and drop to copy. **dragdrop->move**: Appearance of the mouse when using drag and drop to move. **dragdrop->none**: Drag and drop is not possible. **dragdrop->use_default_effect**: Uses the same effect specified in the **effect** parameter.

Exceptions	Description
already_defined	The specified flavor has already been defined.
obj_invalid	The object has already been destroyed using the method destroy [Page 73].

If you use the `copy` and `move` effects when you define the flavor, the system uses the `move` effect when the user drags an object normally, and the `copy` effect when the user presses and holds the CTRL key while dragging.

clear

Deletes the contents of the instance. Once you have called this method, you cannot perform any more drag and drop operations on the corresponding custom control.

```
CALL METHOD dragdrop->clear
    EXCEPTIONS obj_invalid = 1.
```

Exceptions	Description
obj_invalid	The object has already been destroyed using the method destroy [Page 73].

destroy

Deletes the contents of the instance. The instance itself is also destroyed. Once you have called this method, you cannot perform any more drag and drop operations on the corresponding custom control.

```
CALL METHOD dragdrop->destroy.
```

get

Returns the complete description of a flavor.

```
CALL METHOD dragdrop->get
    EXPORTING  flavor        = flavor
    IMPORTING  isdragsrc     = isdragsrc
               isdroptarget  = isdroptarget
               effect        = effect
               effect_in_ctrl = effect_in_ctrl
    EXCEPTIONS not_found     = 1
               obj_invalid = 2.
```

Parameters	Description
flavor	Name of the flavor
dragsrc	'**X**': The description is a drag source
droptarget	'**X**': The description is a drop target
effect	Drop effect of the description between different custom controls. The following effects are supported: **dragdrop->copy**: Appearance of the mouse when using drag and drop to copy. **dragdrop->move**: Appearance of the mouse when using drag and drop to move. **dragdrop->none**: Drag and drop is not possible.
effect_in_ctrl	Drop effect of the description in the same custom control. The following effects are supported: **dragdrop->copy**: Appearance of the mouse when using drag and drop to copy. **dragdrop->move**: Appearance of the mouse when using drag and drop to move. **dragdrop->none**: Drag and drop is not possible. **dragdrop->use_default_effect**: Uses the same effect specified in the **effect** parameter.

Exceptions	Description
already_defined	The specified flavor has already been defined.

If you use the **copy** and **move** effects when you define the flavor, the system uses the **move** effect when the user drags an object normally, and the **copy** effect when the user presses and holds the CTRL key while dragging.

get_handle

This method returns the handle of the drag and drop position. In most cases, you will not need to use this method. However, for tabular mass data interfaces (such as the SAP Tree), you must copy this handle into the interface table.

```
CALL METHOD dragdrop->get_handle
    IMPORTING  handle = handle
    EXCEPTIONS obj_invalid = 1.
```

Parameters	Description
handle	Handle of the drag and drop description

Exceptions	Description
obj_invalid	The object has already been destroyed using the method destroy [Page 73].

modify

Use this method to change an existing flavor.

```
CALL METHOD dragdrop->modify
    EXPORTING  flavor        = flavor
               dragsrc       = dragsrc
               droptarget    = droptarget
               effect        = effect
               effect_in_ctrl = effect_in_ctrl
    EXCEPTIONS not_found   = 1
               obj_invalid = 2.
```

Parameters	Description
flavor	Name of the flavor
dragsrc	'`X`': The description is a drag source
droptarget	'`X`': The description is a drop target
effect	Drop effect of the description between different custom controls. The following effects are supported: **`dragdrop->copy`**: Appearance of the mouse when using drag and drop to copy. **`dragdrop->move`**: Appearance of the mouse when using drag and drop to move. **`dragdrop->none`**: Drag and drop is not possible.
effect_in_ctrl	Drop effect of the description in the same custom control. The following effects are supported: **`dragdrop->copy`**: Appearance of the mouse when using drag and drop to copy. **`dragdrop->move`**: Appearance of the mouse when using drag and drop to move. **`dragdrop->none`**: Drag and drop is not possible. **`dragdrop->use_default_effect`**: Uses the same effect specified in the **`effect`** parameter.

Exceptions	Description
not_found	The specified flavor does not exist
obj_invalid	The object has already been destroyed using the method destroy [Page 73].

If you use the **`copy`** and **`move`** effects when you define the flavor, the system uses the **`move`** effect when the user drags an object normally, and the **`copy`** effect when the user presses and holds the CTRL key while dragging.

remove

Use this method to delete a flavor.

```
CALL METHOD dragdrop->remove
    EXPORTING  flavor = flavor
    EXCEPTIONS not_found   = 1
            obj_invalid = 2.
```

Parameters	Description
flavor	Name of the flavor

Exceptions	Description
not_found	The specified flavor does not exist
obj_invalid	The object has already been destroyed using the method destroy [Page 73].

Methods of the Class CL_DRAGDROPOBJECT

The class CL_DRAGDROPOBJECT describes the context of a drag and drop operation [Page 17]. It contains information about the source object, the flavor of the drag and drop operation, and information about the source and target.

set_flavor

You can only use this method within event handling for the ONGETFLAVOR event. Use the **newflavor** parameter to determine the flavor that you want to use in the drag and drop operation. You receive a list of available flavors as an event parameter.

```
CALL METHOD dragdropobject->set_flavor
    EXPORTING newflavor = newflavor
    EXCEPTIONS illegal_state  = 1
           illegal_flavor = 2.
```

Parameters	Description
newflavor	Name of the flavor

Exceptions	Description
invalid_state	You did not call the method from within event handling for **ONGETFLAVOR**.
obj_invalid	You used a flavor that is not supported by the current drag and drop situation.

abort

Terminates the drag and drop operation immediately. No further events are triggered.

CALL METHOD dragdropobject->abort.