DUNDAS SOFTWARE

Dundas Software is a professional developer of class libraries and components for Microsoft DevStudio developers. Dundas has been developing software for major corporations, associations, and private organizations since 1987.

Dundas has provided software and services to developers and organizations in more than 50 countries throughout the world. Our core technology components have been included in every type of software imaginable, including retail and corporate internal projects. Our consulting services have also aided in customizing our components for the banking, financial, insurance, medical, and political markets.

This long term exposure has proven Dundas to be a reliable software provider, and Dundas is now enabling developers to utilize their robust software as ActiveX controls and COM components.

For more information on Dundas Software's products and services, visit us at www.dundas.com

Phone: (800) 463-1492

(416) 467-5100

Fax: (416) 422-4801

Email: sales@dundas.com

Ultimate Toolbox

Ultimate Toolbox includes more than 200 MFC classes, adding valuable features that include GUI classes, Framework classes, Utility classes, MAPI classes, OLE classes, Image classes, File classes and more.

Ultimate Toolbox has drop-in simplicity and integrates seamlessly with MFC, thereby becoming a part of the MFC framework.

Customers consistently report that the classes in Ultimate Toolbox are "well thought out, debugged, useful, and save time and coding".

New classes are released every two weeks, ensuring that the toolbox is always up to date.

Full source code is included for more than 200 classes, and an attractive yearly subscription plan is available. This subscription includes access to the new classes released every two weeks, source code for the new classes and full technical support.

For more information visit our web site at http://www.dundas.com/

COXMaskedEdit Overview

Copyright © Dundas Software Ltd. 1997 1999, All Rights Reserved

Class Members | Mask Characters

The COXMaskedEdit control extends the MFC CEdit control to provide restricted data input with visual cues, formatted data output, overtype capability, and a validation framework.

You can use COXMaskedEdit anywhere you would use a CEdit class. If no input mask is set, it will behave like a standard CEdit.

If you define an <u>input mask</u>, each character position in the Masked Edit control maps to either a placeholder of a specified type or a literal character. (*Literal characters, or 'literals', can give visual cues about the type of data being used. For example, the parentheses surrounding the area code of a telephone number are literals: (206) 777-2222.)*

The input mask prevents you from entering invalid characters into the control. If you attempt to enter a character that conflicts with the input mask, the control generates a ValidationError beep.

The insertion point automatically skips over literals as you enter data or move the insertion point.

When you insert or delete a character, all nonliteral characters to the right of the insertion point are shifted, as necessary. If shifting these characters leads to a validation error, the insertion or deletion is prevented, and a ValidationError beep is triggered.

For example, suppose the Mask property is defined as "?###", and the current value of the Text property is "A12." If you attempt to insert the letter "B" to the left of the letter "A," the "A" would shift to the right. Since the second value of the input mask requires a number, the letter "A" would cause the control to generate a ValidationError beep.

The Masked Edit control also validates the parameter value of the SetInputText function the user passes at run time. If you use the SetInputText function so that it conflicts with the input mask, the function will return an errorcode.

You may select text in the same way as for a standard text box control. When selected text is deleted, the control attempts to shift the remaining characters to the right of the selection. However, any remaining character that might cause a validation error during this shift is deleted, and no ValidationError beep is generated.

Normally, when a selection in the Masked Edit control is copied onto the Clipboard, the entire selection, including literals, is transferred onto the Clipboard. You can use the SetClipMode function to define the behavior for transferring only user-entered data onto the Clipboard or not - literal characters that are part of the input mask are not copied.

Using COXMaskedEdit

You can attach a COXMaskedEdit to an existing edit control by subclassing the latter.

This is remarkably simple to do in the DevStudio IDE when working with an MFC dialog.

Place a standard edit control on the dialog using the dialog editor. Invoke the Class Wizard and select the Member Variables page. Add a member variable for the ID of the edit control, selecting a CEdit control as the type.

Next, open the header file for the dialog. Include OXMaskedEdit.h.

In the AFX_DATA section for the dialog you will see the declaration for the edit control as a CEdit. Change this to COXMaskedEdit (or a class derived from COXMaskedEdit) and viola!

Typically you will call <u>SetMask</u> and <u>SetPromptSymbol</u> on the control in OnInitDialog to set up your particular mask etc.

Note that you won't need to call Create in this scenario.

Don't forget to include the OXMaskedEdit.cpp file in your project!

Depending on the order of compilation you may also find if helpful to include OXMaskedEdit.h in the dialog and/or main application .cpp file of your project.

COXMaskedEdit

Copyright © Dundas Software Ltd. 1997 1999, All Rights Reserved

Overview | Edit Mask Characters

COXMaskedEdit

The constructor. Can be called with a mask.

Create

Creates the control - much like CEdit create.

GetMask

Retrieves the current mask.

SetMask

Sets the mask.

ShowMask

Retrieves the data - both user input and literals.

GetInputData

Retrieves the data - user input only.

SetInputData

Allows programmatic entry of user data.

GetPromptSymbol

Retrieves the placeholder character.

SetPromptSymbol

Sets the placeholder character.

EmptyData

Clears input data. Optionally clears the mask.

IsInputEmpty

Determines if data has been entered by the user.

GetInsertMode

Retrieves the overtype state.

SetInsertMode

Sets the overtype state.

ValidationError

An overridable, called for errors.

OnValidate

An overridable method which allows for validation

when the control loses focus.

GetAutoTab

Retrieves the state of the auto tab option.

SetAutoTab

Sets the state of the auto tab option.

RPtoLP

Converts the real position to the logical position.

LPtoRP

Converts the logical position to the real position.

COXMaskedEdit Mask Characters

Class Members | Overview

These are the characters you can use to set the mask:

- . (period) Decimal placeholder. The actual character used is the one specified as the decimal placeholder in your international settings. This character is treated as a literal for masking purposes.
- , (comma) Thousands separator. The actual character used is the one specified as the thousands separator in your international settings. This character is treated as a literal for masking purposes.
- : (colon) Time separator. The actual character used is the one specified as the time separator in your international settings. This character is treated as a literal for masking purposes.
- / (slash) Date separator. The actual character used is the one specified as the date separator in your international settings. This character is treated as a literal for masking purposes.
- # Digit placeholder (0-9).

Α	Alphanumeric character placeholder (0-9 and a-Z).
?	Alphabetic placeholder (a-Z).
>	Alphabetic placeholder, but forces uppercase chars (A-Z).
<	Alphabetic placeholder, but forces them to lowercase (a-z).
&	Character placeholder. Valid values for this placeholder are ANSI characters in the following ranges: 32-126 and 128-255.
\	Literal escape. Use this to place your own literals in the mask - note that two backslashes must be used in string literals to accomodate for the fact that this is also treated as an escape character for ASNI/ISO string formatting.
	As an example, lets look at a string to mask an IP address:
	"IP \\Address: ###\\.###\\.###"
	This will appear as:
	IP Address:
	Assuming that the placeholder or 'prompt' symbol has

been set to the underscore.

Note that we needed to use the escape character to enable both the 'A' in Address and the periods to show as literals.

To display the string 'http:// ' we would have to use the escape char for the colon and slashes:

"http\\:\\/\\/ "

To display a backslash as a literal, we need to escape the escape, as in "c:\\\\AAAAAAAA\\.AAA"

See also: <u>COXMaskedEdit::SetMask</u> | <u>COXMaskedEdit::GetMask</u> | <u>COXMaskedEdit::COXMaskedEdit::SetPromptSymbol</u>

COXMaskedEdit::COXMaskedEdit

Class Members | Overview

COXMaskedEdit(LPCTSTR pszMask=_T(""))

Parameters

pszMask A mask can be specified at construction time.

Remarks

Constructs the object.

COXMaskedEdit::Create

Class Members | Overview

BOOL Create(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID)

Parameters

dwStyle Edit control styles.

rect Edit control size and position.

pParentWnd Edit control parent window.

nID Edit control ID.

Return Value

True on success.

Remarks

Same as CEdit::Create except that it sets the mask after creation and shows it as the text of the control (the mask may have been set by SetMask or the constructor).

See also: <u>SetMask</u> | <u>COXMaskedEdit</u>

COXMaskedEdit::EmptyData

Class Members | Overview

void EmptyData(BOOL bOnlyInput=FALSE)

Parameters

bOnlyInput Set to TRUE to only clear the data.

Remarks

Clears the contents of the masked edit. Depending on the value of *bOnlyInput*, this function clears all data (mask + input) or only input data.

COXMaskedEdit::GetAutoTab

Class Members | Overview

BOOL GetAutoTab() const

Return Value

TRUE if AutoTab mode is set, otherwise FALSE.

Remarks

If AutoTab mode is set then when the last allowed symbol is typed the focus goes to the next control with a WS_TABSTOP style. AutoTab mode is not set by default.

See also: SetAutoTab

COXMaskedEdit::GetInputData

Class Members | Overview

CString GetInputData() const

Returns

A CString which stores the string that was entered (excludes the literals).

Remarks

To retrieve the contents of the control including literals call <u>ShowMask</u>. You can also call the CWnd method GetWindowText to retrieve the actual contents of the control, including prompt symbols.

See also: GetMask

COXMaskedEdit::GetInsertMode

Class Members | Overview

BOOL GetInsertMode() const

Return Value

TRUE if insert mode enabled, otherwise FALSE.

Remarks

The standard CEdit control does not support over-typing. COXMaskedEdit supports an overtype mode, and GetInsertMode lets the coder check if this mode is set.

Overtype will only take effect if an input mask is set.

Note that internally the control will toggle the insert mode in response to the insert key, and will revert to overtype if there is no room to insert characters.

See Also: <u>SetInsertMode</u>

COXMaskedEdit::GetMask

Class Members | Overview

CString GetMask() const

Return Value

A string representing the current mask.

See also: SetMask | GetInputData | ShowMask | COXMaskedEdit | COXMaskedEdit Mask Characters

COXMaskedEdit::GetPromptSymbol

Class Members | Overview

TCHAR GetPromptSymbol()

Return Value

The current prompt symbol.

Remarks

The prompt symbol is shown occupying the space available for user
input. For example, a telephone number mask such as (###)\ ###\-#####
might typically use the underscore character as the prompt character,
resulting in a display of () in the control.

The default prompt character is the space.

COXMaskedEdit::IsInputEmpty

Class Members | Overview

BOOL IsInputEmpty()

Return Value

TRUE if the control only has mask and prompt symbols, FALSE otherwise.

Remarks

Retrieves the flag that specifies whether the control is populated with some text other than mask and prompt symbols.

See also: GetInputData | ShowMask

COXMaskedEdit::LPtoRP

Class Members | Overview

int LPtoRP(int nLogicalPos) const

Parameters

nLogicalPos Logical position in the edit control (only takes into account non-literal characters).

Return Value

The corresponding real position (taking into account all symbols including literals) or one (1) if *nLogicalPos* is not a valid logical position.

Remarks

Converts the logical position within the masked edit control to the corresponding real one.

All COXMaskedEdit functions that take a cursor position as an argument interpret it as a real position within the control (taking into account all symbols including literals). But sometimes we want to set the cursor at a position before or after a particular non-literal symbol. This is where this method comes in handy.

See also: RPtoLP

COXMaskedEdit::OnValidate

Class Members | Overview

virtual BOOL OnValidate()

Return Value

TRUE if the current text passes validation, otherwise FALSE.

Remarks

This virtual function gets called when the control loses focus, unless the OMMEN_VALIDATE notification handler of the parent decides otherwise (see below). Use in a COXMaskedEdit derived class to perform user validation. The default implementation returns TRUE.

If this method returns FALSE, the virtual method ValidationError will be called. The default behavior of COXMaskedEdit::ValidationError is to simply sound a message beep.

There is another way to provide data validation that can be used irregardless of the control being used as a base class.

When the control loses focus it will send a WM_NOTIFY notification to its parent passing the ID of the control in wParam and a pointer to a MENMHDR structure in IParam. The hdr.code member of the structure will contain OXMEN VALIDATE.

There are two BOOL members of the MENMHDR structure you can modify to return information on what action to take:

MENMHDR.bVaild Specifies whether data is valid.

MENMHDR.bDefaultValidation Specifies whether to call OnValidate.

See also: ValidationError

COXMaskedEdit::RPtoLP

Class Members | Overview

int RPtoLP(int nRealPos) const

Parameters

nRealPos The real position in the edit control (takes into account all

symbols including literals).

Return Value

Corresponding logical position (taking into account only nonliterals), or -1 if the character at the real position specified corresponds to a literal.

Remarks

Converts a real position (character location) within the masked edit control to a corresponding logical one.

See Also: <u>LPtoRP</u>

COXMaskedEdit::SetAutoTab

Class Members | Overview

void SetAutoTab(BOOL bAutoTab)

Parameters

bAutoTab TRUE will set the control in AutoTab mode, FALSE will set the control in Normal mode.

Remarks

Changes the AutoTab mode. If the AutoTab mode is set then when the last allowed symbol is typed the focus goes to the next control with a WS_TABSTOP style. AutoTab mode is turned off by default.

See Also: GetAutoTab

COXMaskedEdit::SetInputData

Class Members | Overview

BOOL SetInputData(LPCTSTR pszInputData, int nBeginPos=0, BOOL bAllowPrompt=TRUE)

Parameters

pszInputData Each character is entered into the control as if the user

typed it in.

nBeginPos Starting position for inserting or overwriting the Insert

symbols in the mask.

bAllowPrompt Determines whether or not the prompt symbol is a valid

input character.

Return Value

TRUE if some data was inserted, otherwise FALSE.

Remarks

Use this to programmatically insert *pszInputData* into the masked edit control. The mask is applied to the InputData.

COXMaskedEdit::SetInsertMode

Class Members | Overview

void SetInsertMode(BOOL bInsertMode)

Parameters

blnsertMode If TRUE characters are inserted; if FALSE characters will overwrite existing characters.

Remarks

The standard CEdit control does not support over-typing, but COXMaskedEdit does support an overtype mode. SetInsertMode lets the coder toggle this mode on or off, but the control can also change mode based on user input.

Internally (in OnKeyDown) the control will respond to the insert key to toggle insert mode using this function.

If the control is in insert mode and there is no room to insert characters the control will revert to over-type mode automatically.

Note that overtype mode is related to the input mask - if no mask is set the control will act like a normal CEdit and SetInsertMode will have no effect.

See Also: GetInsertMode

COXMaskedEdit::SetMask

Class Members | Overview

void SetMask(LPCTSTR pszMask=_T(""))

Parameters

pszMask A string containing the new mask.

Remarks

Changing the mask may involve loss of user data from the masked edit control.

See Also: GetMask | COXMaskedEdit Mask Characters

COXMaskedEdit::SetPromptSymbol

Class Members | Overview

void SetPromptSymbol(TCHAR chNewPromptSymbol)

Parameters

chNewPromptSymbol Sets a new prompt symbol. The new symbol cannot be NULL, a carriage return or a line feed.

Remarks

The prompt symbol is shown occupying the space available for user
input. For example, a telephone number mask such as (###)\ ###\-####
might typically use the underscore character as the prompt symbol,
resulting in a display of () in the control.

The default symbol is a space.

See Also: GetPromptSymbol

COXMaskedEdit::ShowMask

Class Members | Overview

CString ShowMask() const

Return Value

The fully formated mask with all input data. To retrieve only the characters entered by the user call the <u>GetInputData</u> method.

See also: COXMaskedEdit Mask Characters

COXMaskedEdit::ValidationError

Class Members | Overview

virtual void ValidationError()

Remarks

This is an overridable function that is called when data fails validation or an incorrect key is pressed.

See Also: OnValidate

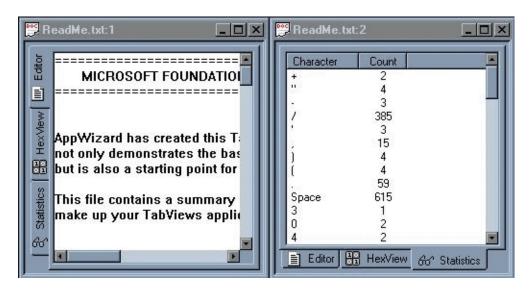
COX3DTabViewContainer Overview

Copyright © Dundas Software Ltd. 1997 - 1999, All Rights Reserved

Class Reference

Dependencies and Related Files

Example of a **3DTabViewContainer**:



For most applications it's not enough to use only one window to provide output. There are different solutions for this problem like splitters or docking windows but they usually have one common inconvenience: all of the windows are shown at the same time, taking up precious screen space while being unused.

A good example of how this problem can be resolved can be found in the **Microsoft Visual Studio IDE** with its "Output" window (with "Build", "Debug", "Find in Files..." panes) and "Result List" window (with "Search", "Lookup", "See Also" and "History" panes). We call these windows TabViews.

TabViews are a good alternative for splitter windows when you need to have more than one view per document. TabViews can be used within a docking window and can also serve as a container for associated windows (pages) that are implemented as dialog bars.

COX3DTabViewContainer introduces a new implementation of TabViews. The paradigm remains the same but we've changed the way associated windows (pages) are represented in the container. We use a standard **Tab control** and display a tab button for each page, and when a user clicks on a button the corresponding page is activated and displayed. Tab buttons can be positioned at any side of the container window by applying corresponding Tab control styles (refer to the Create() function for details).

COX3DTabViewContainer is derived from the standard **CTabCtrl** and implements all of the functionality needed to support tab views.

Here is a list of steps that should be taken in order to deploy **TabViews** in your application:

<u>First Case Scenario:</u> COX3DTabViewContainer will be used as a container for document view(s).

- 1) Embed a **COX3DTabViewContainer** member variable in the parent frame (main frame window for SDI application, MDIChild window for MDI application).
- 2) Override the parent frame's **CFrameWnd::OnCreateClient()** member function.
- 3) From within the overridden **OnCreateClient**, call the <u>Create()</u> member function of the **COX3DTabViewContainer**. You will have to specify the parent window and you can optionally specify the initial rectangle, window styles and the window ID. This is where you can specify the **Tab control styles** that define the way tab buttons are positioned and displayed.
- 4) To assign images to TabView pages you will have to create and load

an image list and associate it with a COX3DTabViewContainer object using the CTabCtrl::SetImageList() function.

5) After the COX3DTabViewContainer window is successfully created you can populate it with window objects using the AddPage() or InsertPage() functions. If you are inserting a view object you have to specify the runtime class and context information in order to keep the document/view architecture in place. If you are adding a window object that is not a document view then you will have to create it before adding it to the COX3DTabViewContainer window. In the AddPage() or InsertPage() functions you can specify the text that will be used as the page title in the corresponding tab button. You can also specify the index of the image in the tab control's image list that should be displayed in the tab button.

For example:

```
BOOL CChildFrame::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext* pContext)

{
    // TODO: Add your specialized code here and/or call the base class

    UNREFERENCED_PARAMETER(lpcs);
    if(!m_TabViewContainer.Create(this))

    return FALSE;

VERIFY(m_ilTabView.Create(IDB_IL_TABVIEWS,16,0,RGB(255,0,255)));

m_TabViewContainer.SetImageList(&m_ilTabView);

if(!m_TabViewContainer.AddPage(pContext->m_pNewViewClass, pContext,_T("Primary View"),0))

{
    return FALSE;

}

if(!m_TabViewContainer.AddPage(RUNTIME_CLASS(CMyView2), pContext,_T("View2"),1))
```

```
{
    return FALSE;
}

m_TabViewContainer.SetActivePageIndex(0);
return TRUE;
}
```

<u>Second Case Scenario</u>: The **COX3DTabViewContainer** will be used as a container for windows within a control bar.

- 1) Create your own CControlBar-derived class (you can use our **COXSizeControlBar** as a parent class if you need sizable docking windows). Let's call it CMyControlBar.
- 2) Embed a COX3DTabViewContainer member variable in this class.
- 3) Override the **CMyControlBar::OnCreate()** member function.
- 4) From within the overridden OnCreate(), call the <u>Create</u> member function of the **COX3DTabViewContainer** object. You have to specify the parent window and you can optionally specify the initial rectangle, window styles and window ID. This is where you can specify **Tab control styles** that defines the way tab buttons are positioned and displayed.
- 5) If you plan to assign images to TabView pages then you have to create and load an image list and associate it with the **COX3DTabViewContainer** using the **CTabCtrl::SetImageList()** function.
- 6) After the **COX3DTabViewContainer** window is successfully created you can populate it with window objects using the <u>AddPage()</u> or <u>InsertPage()</u> functions. Note that you have to create the window object before adding it to **COX3DTabViewContainer**. In the AddPage or InsertPage functions you can specify the text that will be used as the page title in a tab button. You can also specify the index of an image in

the tab control image list that should be displayed in the tab button.

7) Override the **CMyControlBar::OnSize()** member function and resize the **COX3DTabViewContainer** object.

For example:

```
int CMyControlBar::OnCreate(LPCREATESTRUCT lpCreateStruct)
 {
 if (COXSizeControlBar::OnCreate(lpCreateStruct) == -1)
   return -1;
 if(!m_TabViewContainer.Create(this))
   return -1;
 VERIFY(m_ilTabView.Create(IDB_IL_TABVIEWS,16,0,RGB(255,0,255)));
 m_TabViewContainer.SetImageList(&m_ilTabView);
 // edit control
 if(!edit.Create(WS_CHILD|ES_MULTILINE|ES_AUTOHSCROLL|
   ES_AUTOVSCROLL|WS_HSCROLL|WS_VSCROLL,CRect(0,0,0,0),
   &m_TabViewContainer,1))
 {
   return -1;
 }
 m_TabViewContainer.AddPage(&edit,_T("Edit"),0);
 // list box
 if(!listBox.Create(WS_CHILD|WS_HSCROLL|WS_VSCROLL,
   CRect(0,0,0,0),&m_TabViewContainer,2))
```

```
return -1;
}
m_TabViewContainer.AddPage(&listBox,_T("ListBox"),1);
// list control
if(!listCtrl.Create(WS_CHILD|LVS_REPORT,
  CRect(0,0,0,0),&m_TabViewContainer,3))
{
  return -1;
}
m_TabViewContainer.AddPage(&listCtrl,_T("List"),2);
// tree control
if(!treeCtrl.Create(WS_CHILD|TVS_HASLINES|TVS_LINESATROOT|TVS_HASBUTTONS,
  CRect(0,0,0,0),&m_TabViewContainer,4))
{
  return -1;
}
m_TabViewContainer.AddPage(&treeCtrl,_T("Tree"),3);
m\_TabViewContainer.SetActivePageIndex(0);
return 0;
}
```

Note that **any** child window can be used as a **COX3DTabViewContainer** page.

The steps to be taken in order to implement **COX3DTabViewContainer** in a CControlBar derived window should be used in general cases as well. A CControlBar derived window was used above since it is a likely choice for the parent window.

The following functions have been provided for those who need to dynamically change the contents of a **COX3DTabViewContainer** object:

In order to remove any page at run time call the DeletePage() function.

To set/retrieve the page title that is displayed in the corresponding tab button use <u>GetPageTitle()</u> and <u>SetPageTitle()</u>.

To set/retrieve the active page index call <u>GetActivePageIndex</u>() and <u>SetActivepageIndex</u>().

For more information examine the sample found in: <INSTALLDIR>\3DTabVws\Samples\Gui\3DTabView\TabViews.dsw

COX3DTabViewContainer Class Members

Copyright © Dundas Software Ltd. 1997 1999, All Rights Reserved

Overview

COX3DTabViewContainer

Constructs the object.

Create

Creates the 3DTabView container.

AddPage

Adds a new page to the 3DTabViewContainer.

InsertPage

Inserts a new page into the

3DTabViewContainer.

DeletePage

Deletes an existing page from the

3DTabView container.

GetPage

Retrieves a pointer to a page which is

specified by an index.

GetPageTitle

Retrieves the title of the specified page.

SetPageTitle

Sets the title of the specified page.

<u>GetPageImageIndex</u>

Retrieves the image index of the specified

page.

SetPageImageIndex

Sets the image index of the specified page.

GetPageCount

Retrieves the number of pages in the

3DTabView container.

FindPage

Retrieves a flag indicating whether or not the specified window is a page belonging to the 3DTabViewContainer. If it is then the

index of the located page is provided.

IsPage

Retrieves a flag indicating whether or not

the specified window is a page belonging

to the 3DTabViewContainer.

IsActivePage

Retrieves a flag indicating whether the

specified window is the currently active page for the 3DTabViewContainer.

GetActivePageIndex

Retrieves the index of the currently active

page.

GetActivePage

Retrieves a pointer to the currently active

page.

SetActivePage

Sets the specified page as the active page.

SetActivePageIndex

Sets the specified page as the active page

via the page index.

SetOffsetExternal

Sets the tab control offset from the parent

window borders.

<u>GetOffsetExternal</u>

Retrieves the tab control offset from the

parent window borders.

SetOffsetInternal

Sets the page window offset from the tab

control display rectangle.

GetOffsetInternal

Retrieves the page window offset from the tab control display rectangle.

AcceptDraggedObject

Toggles "drag object over" support for the tab control on or off.

<u>IsAcceptingDraggedObject</u>

Retrieves a flag that specifies whether the 3DTabView container activates the corresponding page window when an object is dragged over tab control items.

RecalcPageRect

Calculates the rectangle which specifies the coordinates of the active page window.

GetPageRect

Retrieves the rectangle which specifies the coordinates of the active page window.

COX3DTabViewContainer::COX3DTabViewContainer

Class Members | Overview

COX3DTabViewContainer()

Remarks

Constructor.

COX3DTabViewContainer::AcceptDraggedObject

Class Members | Overview

void AcceptDraggedObject(const BOOL bAccept=TRUE)

Parameters

bAccept Any object dragged over tab items will result in the corresponding page window being activated if set to TRUE.

Remarks

Toggles 'drag object over' support for the tab control on and off.

COX3DTabViewContainer::AddPage

Class Members | Overview

BOOL AddPage(CRuntimeClass* pClass, CCreateContext* pContext, LPCTSTR lpszTitle=NULL, const int nlmage=-1)

BOOL AddPage(CWnd* pWnd, LPCTSTR lpszTitle=NULL, const int nlmage=-1)

Parameters

pClass Pointer to the runtime class information of the new window to

be added as new page.

pContext Pointer to context information (refer to the description of the

CCreateContext class in the MFC documentation).

pWnd Pointer to a created window to be added as a new page.

IpszTitle Text that will be used as the page title for a tab button.

nlmage Index of the image in the image list associated with the

container that will be set to the corresponding button.

Return Value

TRUE if the new page was successfully added, otherwise FALSE.

Remarks

Adds a new page to a 3DTabView container. Use the first version of the function if you have to add a **CView** derived class that will be part of the document/view architecture of your application.

COX3DTabViewContainer::Create

Class Members | Overview

virtual BOOL Create(CWnd* pParentWnd, CRect rect=CRect(0,0,0,0), DWORD dwStyle=DEFAULT_ TABCTRLSTYLE, UINT nID=AFX_IDW_PANE_FIRST)

Parameters

pParentWnd Pointer to the window that is the 3DTabViewContainer's

parent.

rect Window rectangle.

dwStyle The 3DTabView container's style. By default we use

DEFAULT_TABCTRLSTYLE which expands as

TCS_MULTILINE | TCS_BOTTOM | TCS_HOTTRACK |

TCS_SCROLLOPPOSITE | TCS_RIGHTJUSTIFY | TCS_FOCUSNEVER | WS_VISIBLE | WS_CHILD.

nID The 3DTabViewContainer's ID.

Return Value

TRUE if the 3DTabViewContainer was successfully created, otherwise FALSE.

Remarks

Creates a 3DTabViewContainer.

COX3DTabViewContainer::DeletePage

Class Members | Overview

virtual BOOL DeletePage(const CWnd* pWnd, const BOOL bDestroy=TRUE)

virtual BOOL DeletePage(const int nIndex, const BOOL bDestroy=TRUE)

Parameters

pWnd Pointer to the page to be deleted.

nIndex Index of the page to be deleted.

bDestroy Flag which specifies if the window has to be destroyed.

Return Value

TRUE if the specified page was successfully deleted, otherwise FALSE.

Remarks

Deletes an existing page from a **3DTabViewContainer**.

COX3DTabViewContainer Dependencies and Related Files

Copyright © Dundas Software Ltd. 1997-1999, All Rights Reserved

Overview

SOURCE OX3DTabView.cpp

INCLUDE OX3DTabView.h

SAMPLE <INSTALLDIR>\3DTabVws\Samples\Gui\3DTabView\TabViews

COX3DTabViewContainer::FindPage

Class Members | Overview

BOOL FindPage(const CWnd* pTestWnd, int& nIndex) const

BOOL FindPage(const HWND hTestWnd, int& nIndex) const

Parameters

pTestWnd Pointer to the window to be tested as a

3DTabViewContainer's page.

hTestWnd Handle of the window to be tested as a

3DTabViewContainer's page.

nIndex Reference variable, stores the index of the page if found.

This index is zero (0) based.

Return Value

TRUE if the specified window is a 3DTabView container's page, otherwise FALSE.

Remarks

Retrieves a flag which specifies whether the specified window is a **3DTabViewContainer**'s page. If it is then the index of the located page is indicated by *nIndex*.

COX3DTabViewContainer::GetActivePage

Class Members | Overview

CWnd* GetActivePage() const

Return Value

Pointer to the currently active page of a **3DTabViewContainer**.

Remarks

Retrieves a pointer to the currently active page of a **3DTabViewContainer** object.

See also: GetActivePageIndex

COX3DTabViewContainer::GetActivePageIndex

Class Members | Overview

int GetActivePageIndex() const

Return Value

Index of a **3DTabViewContainer's** currently active page.

Remarks

Retrieves the index of the currently active page.

See also: GetActivePage

COX3DTabViewContainer::GetOffsetExternal

Class Members | Overview

DWORD GetOffsetExternal() const

Return Value

Offset in points from the parent window client area where the tab control will be displayed.

Remarks

Retrieves the tab control offset from the parent window's borders.

See also: SetOffsetExternal

COX3DTabViewContainer::GetOffsetInternal

Class Members | Overview

DWORD GetOffsetInternal() const

Return Value

Offset in points from the display area of the tab control and active page window.

Remarks

Retrieves the page window offset from the tab control display rectangle.

See also: SetOffsetInternal

COX3DTabViewContainer::GetPage

Class Members | Overview

CWnd* GetPage(const int nIndex) const

Parameters

nIndex Zero-based index of the page to be retrieved.

Return Value

Pointer to the corresponding page window or NULL if the specified index was out of range.

Remarks

Retrieves a pointer to the specified page.

COX3DTabViewContainer::GetPageCount

Class Members | Overview

int GetPageCount() const

Return Value

The number of pages in the 3DTabView container.

Remarks

Retrieves the number of pages in the COX3DTabViewContainer.

COX3DTabViewContainer::GetPageImageIndex

Class Members | Overview

int GetPageImageIndex(const CWnd* pWnd) const

int GetPageImageIndex(const int nIndex) const

Parameters

pWnd Pointer to the page whose image index is to be retrieved.

nIndex Zero-based index of the page whose image index is to be

retrieved.

Return Value

Image index of the specified page.

Remarks

Retrieves the image index of the specified page.

COX3DTabViewContainer::GetPageRect

Class Members | Overview

CRect GetPageRect() const

Return Value

The rectangle that specifies the coordinates of the active page window.

Remarks

Retrieves the rectangle which specifies the coordinates of the active page window.

COX3DTabViewContainer::GetPageTitle

Class Members | Overview

CString GetPageTitle(const CWnd* pWnd) const

CString GetPageTitle(const int nIndex) const

Parameters

pWnd Pointer to a page for which the title is to be retrieved.

nIndex Zero-based index of the page for which the image index is to

be retrieved.

Return Value

Title of the corresponding page.

Remarks

Retrieves the title of the specified page.

COX3DTabViewContainer::InsertPage

Class Members | Overview

virtual BOOL InsertPage(const int nIndex, CRuntimeClass* pClass, CCreateContext* pContext, LPCTSTR lpszTitle=NULL, const int nImage=1)

virtual BOOL InsertPage(const int nIndex, CWnd* pWnd, LPCTSTR lpszTitle=NULL, const int nImage=1)

Parameters

nIndex Zero-based index of the page to be added.

pClass Pointer to the runtime class information of the new window to

be added.

pContext Pointer to context information (refer to the description of the

CCreateContext class in the MFC documentation).

pWnd Pointer to the created window to be inserted as the new

page.

IpszTitle The text that will be used as the page title in a tab button.

nlmage Index of the image in the image list (which is associated with

the container) that will be used for the corresponding button.

Return Value

TRUE if the new page was successfully inserted, otherwise FALSE.

Remarks

Inserts a new page into a 3DTabViewContainer. Use the first version of

the function if you have to insert a CView derived class which is part of the document/view architecture of your application.

COX3DTabViewContainer::IsAcceptingDraggedObject

Class Members | Overview

BOOL IsAcceptingDraggedObject() const

Return Value

TRUE if the **3DTabViewContainer** activates the corresponding page window when an object is dragged over tab control items, otherwise FALSE.

Remarks

Retrieves the flag that specifies whether the 3DTabViewContainer activates the corresponding page window when an object is dragged over tab control items.

COX3DTabViewContainer::IsActivePage

Class Members | Overview

BOOL IsActivePage(const HWND hTestWnd) const

BOOL IsActivePage(const CWnd* pTestWnd) const

Parameters

hTestWnd Handle of the window to be tested as the currently active 3DTabViewContainer's page.

pTestWnd Pointer to the window to be tested as the currently active 3DTabViewContainer's page.

Return Value

TRUE if the specified window is the currently active 3DTabViewContainer's page, otherwise FALSE.

Remarks

Retrieves a flag which indicates whether or not the specified window is the currently active 3DTabViewContainer's page.

COX3DTabViewContainer::IsPage

Class Members | Overview

BOOL IsPage(const HWND hTestWnd) const

BOOL IsPage(const CWnd* pTestWnd) const

Parameters

hTestWnd Handle of the window to be tested as the 3DTabViewContainer's page.

pTestWnd Pointer to the window to be tested as the 3DTabViewContainer's page.

Return Value

TRUE if the specified window is a page of the **3DTabViewContainer**, otherwise FALSE.

Remarks

Retrieves a flag which indicates whether the specified window is a page of the 3DTabViewContainer.

COX3DTabView::RecalcPageRect

Class Members | Overview

void RecalcPageRect()

Remarks

Calculates the rectangle which specifies the coordinates of the active page window.

COX3DTabViewContainer::SetActivePage

Class Members | Overview

BOOL SetActivePage(const CWnd* pWnd)

Parameters

pWnd Pointer to the page to be set as active.

Return Value

TRUE if the specified page was successfully set as the active page.

Remarks

Sets the specified page as being active.

COX3DTabViewContainer::SetActivePageIndex

Class Members | Overview

virtual BOOL SetActivePageIndex(const int nIndex)

Parameters

nIndex Index of the page to be set as the active page.

Return Value

TRUE if a page with the specified index was successfully set as the active page.

Remarks

Sets a page with the specified index as the active page.

COX3DTabViewContainer::SetOffsetExternal

Class Members | Overview

void SetOffsetExternal(const DWORD dwOffset)

Parameters

dwOffset The offset in points from the parent window client area where the tab control will be displayed.

Remarks

Sets the tab control offset from the parent window borders.

See also: GetOffsetExternal

COX3DTabViewContainer::SetOffsetInternal

Class Members | Overview

void SetOffsetInternal(const DWORD dwOffset)

Parameters

dwOffset The offset in points from the display area of the tab control and active page window.

Remarks

Sets the page window offset from the tab control display rectangle.

See also: GetOffsetInternal

COX3DTabViewContainer::SetPageImageIndex

Class Members | Overview

BOOL SetPageImageIndex(const CWnd* pWnd, int nImage)

BOOL SetPageImageIndex(const int nIndex, int nImage)

Parameters

pWnd Pointer to the page for which the image is to be set.

nIndex Zero-based index of the page for which the image is to be set.

nlmage Index of an image in the image list that will be used as the

page image in the corresponding tab button.

Remarks

Sets the image index of the specified page.

COX3DTabViewContainer::SetPageTitle

Class Members | Overview

BOOL SetPageTitle(const CWnd* pWnd, LPCTSTR lpszTitle)

BOOL SetPageTitle(const int nIndex, LPCTSTR lpszTitle)

Parameters

pWnd Pointer to the page for which the title is to be set.

nIndex Zero-based index of the page for which the title is to be set.

IpszTitle The text that will be used as the page title in the tab button.

Remarks

Sets the title of the specified page.

COXToolTipCtrl Overview

Copyright © Dundas Software Ltd. 1997-1999, All Rights Reserved

COXToolTipCtrl Class Members

COXToolTipCtrl is an extended tooltip control that allows multiline tooltips, plus extended tooltip text. Extended tooltip text is extra text that is displayed if the user clicks on the tooltip window. If the tooltip contains extended text (as well as a standard tooltip string) then the info window will contain a small arrow that prompts the user to click on the window. Once the window is clicked the extended text is shown. If the window is clicked again then the window is reduced to displaying just the standard text.

The maximum width of the tooltips can be specified, and if the info text is too large to fit within these bounds then the text will be wrapped over multiple lines. The control also allows you to specify different **text and background colors** for the tooltips, and the **display font** can also be changed.

This class is a direct replacement for the **CToolTipCtrl** class. It incorporates the entire API of the standard CToolTipCtrl and introduces new features not found in the standard tooltip.

The control is used just like any other tooltip control. To use it simply call Create(...) and specify the parent window of the tool, then add tools to the control using the AddTool(...) member functions. For example, to add the tooltip to a formview or dialog:

tooltip.Create(this); tooltip.AddTool(GetDlgItem(IDC_CONTROL), T("Tooltip text\rThis is the extended\ntooltip text"));

where ID_CONTROL is the ID of a control.

To specify extended text for a tooltip simply append a '\r' after your tooltip

text, and then append the extended tooltip info.

As with the standard tooltip control you can specify the actual text for the tool at creation time (as shown above), or you can specify the LPSTR_TEXTCALLBACK value and provide a TTN_NEEDTEXT handler to return the text dynamically at runtime.

To handle the TTN_NEEDTEXT message you will need to add a message handler in the parent window as well as an entry in the message map. For example, in a view or form:

```
BEGIN_MESSAGE_MAP(CMyDlg, CDialog)
. . . . . . . . . . . . . . .
ON NOTIFY EX(TTN NEEDTEXT, 0, OnToolTipNotify)
END MESSAGE MAP()
BOOL CMyDlg::OnInitDialog()
{
CDialog::OnInitDialog();
  tooltip.Create(this);
  tooltip.AddTool(GetDlgItem(IDC_CONTROL), LPSTR_TEXTCAl
  . . . .
}
BOOL CMyDlg::OnToolTipNotify(UINT id, NMHDR* pNMHDR, L]
{
TOOLTIPTEXT *pTTT = (TOOLTIPTEXT *)pNMHDR;
```

```
UINT nID = pNMHDR->idFrom;
  if (nID == IDC CONTROL) // Fill in the text buffer
  {
_tcscpy(pTTT->szText, _T("Tooltip text\rExtended tooltip text"));
    return TRUE;
  }
return FALSE;
}
Alternatively you can supply text by either supplying a string resource:
    pTTT->lpszText = MAKEINTRESOURCE(nID);
    pTTT->hinst = AfxGetResourceHandle();
    return TRUE;
or by supplying a pointer to the text:
    pTTT->lpszText = T("Tooltip text\rExtended tooltip text");
    return TRUE:
```

Newline characters ('\n') can be embedded anywhere within the text or extended text to produce a multiline tooltip. If the width of the tooltip window is specified using SetMaxTipWidth() then the tooltip text will be wrapped to this length, and if necessary will be displayed on more than one line.

To change the font of the tooltips simply use the **SetFont**() member function.

The <u>GetToolInfo</u>/<u>SetToolInfo</u> functions and the HitTest functions are very similar to the CToolTipCtrl versions with the exception that they use a **OXTOOLINFO** structure instead of a TOOLINFO structure. This structure

is defined as

```
struct OXTOOLINFO : public TOOLINFO

{
#if (_WIN32_IE < 0x0300)
    LPARAM IParam; //Application defined value that is associated with the tool
#endif
    int nWidth; //Width of box, or 0 for default
    COLORREF clrTextColor; //text color
    COLORREF clrBackColor; //background color
}
```

and is very similar to the standard TOOLINFO. It is used in the same way except that the *uFlags* member is not used (yet).

To change the color of an individual tip use the <u>GetToolInfo/SetToolInfo</u> functions:

```
OXTOOLINFO ToolInfo;
if (m_toolTip.GetToolInfo(ToolInfo, GetDlgItem(IDC_CONTROL)))
{
    ToolInfo.clrBackColor = RGB(255, 255, 255);
    ToolInfo.clrTextColor = RGB( 0, 0, 255);
    m_toolTip.SetToolInfo(&ToolInfo);
}
```

The **ToolTipEx** sample that demonstrates the functionality of the **COXToolTipCtrl** class can be found in: <INSTALLDIR>\ToolTips\Samples\Gui\ToolTipEx\ToolTipEx.dsw.

COXToolTipCtrl Class Members

Copyright © Dundas Software Ltd. 1997-1999, All Rights Reserved

Overview

COXT	ToolT	ipCtrl
	<u> </u>	

Constructs and initializes the object.

GetToolInfo

Retrieves the information that a tooltip control

maintains about a tool.

SetToolInfo

Sets the information that a tooltip maintains for a

tool.

GetMargin

Retrieves the margins used for drawing the text

in the tooltip.

GetText

Retrieves the text that a tooltip control maintains

for a tool.

SetDelayTime

Sets the delay times for the tooltip in

milliseconds.

GetDelayTime

Retrieves the initial, pop-up, and reshow durations currently set for a tooltip control.

GetMaxTipWidth

Retrieves the maximum width of the tooltip window.

SetMaxTipWidth

Sets the maximum tooltip window width.

GetTipBkColor

Retrieves the background color.

SetTipBkColor

Sets the background colour for all tools maintained by this control.

<u>GetTipTextColor</u>

Retrieves the text color.

SetTipTextColor

Sets the text color for all tools maintained by this control.

Activate

Activates/deactivates the tooltip control.

GetToolCount

Retrieves a count of the tools registered with the tooltip control.

SetMargin

Sets the top, left, bottom, and right margins for a

tooltip window.

Create

Creates the tooltip window.

AddTool

Registers a tool with the tooltip control.

DelTool

Removes the specified tool.

RelayEvent

Passes a mouse message to a tooltip control for

processing.

HitTest

Retrieves a flag which specifies if a given point

is in the windows toolinfo bounding rectangle.

<u>Pop</u>

Hides the tooltip.

CalculateInfoBoxRect

Advanced overridable. Protected virtual function

that calculates the rectangle (in screen coords) that is best suited to displaying the tooltip info.

GetBoundsRect

Advanced overridable. Protected virtual function that calculates the smallest possible rectangle that will contain the text.

COXToolTipCtrl::Activate

Members | Overview

void Activate(BOOL bActivate)

Parameters

bActivate Specifies whether the tooltip control is to be activated or deactivated.

Remarks

Call this function to activate or deactivate a tooltip control. If *bActivate* is TRUE, the control is activated. If FALSE, it is deactivated. When a tooltip control is active, the tooltip information appears when the cursor is on a tool that is registered with the control; when it is inactive, the tooltip information does not appear, even when the cursor is over a registered tool.

COXToolTipCtrl::AddTool

Members | Overview

BOOL AddTool(CWnd* pWnd, UINT nIDText, LPCRECT lpRectTool = NULL, UINT nIDTool = 0)

BOOL AddTool(CWnd* pWnd, LPCTSTR szText, LPCRECT lpRectTool = NULL, UINT nIDTool = 0)

Parameters

pWnd Pointer to the window that contains the tool.

ID of the string resource that contains the text or the tool.

If the text contains a '\r' character, then all text before the \r is the standard tooltip text, and all text after the \r will be

displayed as extended text if the use clicks on the tooltip.

IpszText Pointer to the text for the tool. If the text contains a '\r'

character, then all text before the \r is the standard tooltip text, and all text after the \r will be displayed as extended

text if the user clicks on the tooltip. If the text is

LPSTR_TEXTCALLBACK then the control will send the TTN NEEDTEXT notification message to the parent

window to retrieve the text.

IpRectTool Pointer to a RECT structure containing coordinates of the

tool's bounding rectangle, using client coordinates relative

to the window identified by pWnd.

nIDTool ID of the tool.

Return Value

TRUE on success, FALSE otherwise.

Remarks

Registers a tool with the tooltip control, so that the information stored in the tooltip is displayed when the cursor is over the tool.

See also: GetToolCount | DelTool

COXToolTipCtrl::CalculateInfoBoxRect

Members | Overview

virtual CRect CalculateInfoBoxRect(CPoint& pt, COXToolTipInfo* pToolTip, CRect&rectTextBounds) const

Parameters

pt The top left corner of the region.

pToolTip Information on the tooltip.

rectTextBounds The minimum rectangle needed to contain the text.

Return Value

A rectangle containing the bounds of the tooltip.

Remarks

Given the bounding rectangle of some text, this function returns the rectangle (in screen coordinates) that is best suited to displaying the tooltip information (uses the current mouse position). You can override this method in a derived class to customize the calculation method.

COXToolTipCtrl::COXToolTipCtrl

Members | Overview

COXToolTipCtrl()

Remarks

Constructor. Creates an instance of this class.

COXToolTipCtrl::Create

Members | Overview

BOOL Create(CWnd* pParentWnd)

Parameters

pParentWnd A pointer to the tooltip control's parent.

Return Value

TRUE if successfulk, otherwise FALSE.

Remarks

Creates the tooltip window which initially is not visible.

COXToolTipCtrl::DelTool

Members | Overview

void DelTool(CWnd* pWnd, UINT nIDTool = 0)

Parameters

pWnd Pointer to the window that contains the tool.

nIDTool ID of the tool.

Remarks

Removes the tool specified by *pWnd* and *nIDTool* from the collection of tools supported by a tooltip control.

See also: AddTool

COXToolTipCtrl::GetBoundsRect

Members | Overview

virtual CRect GetBoundsRect(CString strText, int nWidth) const

Parameters

strText The text to be displayed (may be multiline).

nWidth The desired width. If this is zero (0) the width will be

calculated.

Return Value

The bounding RECT for the text (with the top left corner at 0, 0).

Remarks

Returns the smallest possible rectangle that will contain the text (including margins).

See also: CalculateInfoBoxRect

COXToolTipCtrl::GetDelayTime

Members | Overview

int GetDelayTime(DWORD dwDuration) const

Parameters

dwDuration Flag that specifies which duration value will be retrieved. It can be one of the following:

TTDT_AUTOPOP The length of time the tooltip

window remains visible if the

pointer is stationary within a tool's

bounding rectangle.

The length of time the pointer TTDT INITIAL

must remain stationary within a tool's bounding rectangle before

the tooltip window appears.

Return Value

The delay times for the tooltip are in milliseconds (mS).

Remarks

Retrieves the initial, pop-up, and reshow durations currently set for a tooltip control.

See also: <u>SetDelayTime</u>

COXToolTipCtrl::GetMargin

Members | Overview

void GetMargin(LPRECT lprc) const

Parameters

Iprc Address of a RECT structure that will store the margin information.

Remarks

Retrieves the margins used for drawing the text in the tooltip. The rectangle does not specify a bounding rect, but rather the top, bottom, left and right distances (in pixels) between the text and the edge of the tooltip window.

See also: GetBoundsRect

COXToolTipCtrl::GetMaxTipWidth

Members | Overview

int GetMaxTipWidth() const

Return Value

The maximum width for a tooltip window, or zero (0) if this width is calculated automatically.

Remarks

Retrieves the maximum width of the tooltip window.

See also: SetMaxTipWidth

COXToolTipCtrl::GetText

Members | Overview

void GetText(CString& str, CWnd* pWnd, UINT nIDTool = 0)

Parameters

str Reference to a CString object that stores the tools text.

pWnd Pointer to the window that contains the tool.

nIDTool ID of the tool.

Remarks

Retrieves the text that a tooltip control maintains for a tool. If *pWnd* and *nIDTool* specify a valid tool that has been previously registered, then *str* is filled with the tooltip's text.

COXToolTipCtrl::GetTipBkColor

Members | Overview

COLORREF GetTipBkColor() const

Return Value

A COLORREF value that represents the background color of the tooltip window.

Remarks

Retrieves the background color of the tooltip window.

See also: <u>SetTipBkColor</u> | <u>SetTipTextColor</u>

COXToolTipCtrl::GetTipTextColor

Members | Overview

COLORREF GetTipTextColor() const

Return Value

A COLORREF value that represents the text color.

Remarks

Retrieves the text color of the tooltip window.

See also: <u>SetTipTextColor</u> | <u>SetTipBkColor</u>

COXToolTipCtrl::GetToolCount

Members | Overview

int GetToolCount() const

Return Value

The number of tools registered with the tooltip control.

Remarks

Retrieves the number of tools registered with the tooltip control.

See also: AddTool | DelTool

COXToolTipCtrl::GetToolInfo

Members | Overview

BOOL GetToolInfo(OXTOOLINFO& ToolInfo, CWnd* pWnd, UINT nIDTool = 0)

Parameters

ToolInfo Reference to an OXTOOLINFO object that receives the tools

text.

pWnd Pointer to the window that contains the tool.

nIDTool ID of the tool.

Return Value

TRUE if successful, otherwise FALSE.

Remarks

Call this function to retrieve the information that a tooltip control maintains about a tool. If the control has information on the tool identified by *pWnd* and *nIDTool* then the COXToolTipInfo structure is populated with that information.

See the <u>overview</u> for a description of the OXTOOLINFO structure.

See also: SetToolInfo

COXToolTipCtrl::HitTest

Members | Overview

BOOL HitTest(CWnd* pWnd, POINT pt, OXTOOLINFO* pToolInfo) const

Parameters

pWnd Pointer to the window that contains the tool.

pt Pointer to a CPoint object containing the coordinates of the

point to be tested.

pToolInfo Pointer to a OXTOOLINFO structure that contains information

about the tool.

Return Value

TRUE if the point specified by the hit-test information is within the tools bounding rectangle; otherwise FALSE.

Remarks

Returns TRUE if the given *pt* is in the windows toolinfo bounding rectangle (*pt* is in client coordinates relative to the parent window). If this function returns TRUE the structure pointed to by *pToolInfo* is populated with information about the corresponding tool.

See the <u>overview</u> for information on the OXTOOLINFO structure.

COXToolTipCtrl::Pop

Members | Overview

void Pop()

Remarks

Hides the tooltip.

COXToolTipCtrl::RelayEvent

<u>Members</u> | <u>Overview</u>

void RelayEvent(MSG* pMsg)

Parameters

pMsg Pointer to a MSG structure that contains the message to relay.

Remarks

Call this function to pass a mouse message to a tooltip control for processing.

COXToolTipCtrl::SetDelayTime

Members | Overview

void SetDelayTime(DWORD dwDuration, int nTime)
void SetDelayTime(UINT nDelay)

Parameters

dwDuration Flag that specifies which duration value will be set. Can be one of the following values:

TTDT_AUTOPOP The length of time the tooltip

window remains visible if the

pointer is stationary within a tool's

bounding rectangle.

TTDT_INITIAL The length of time the pointer must

remain stationary within a tool's bounding rectangle before the

tooltip window appears.

nTime Specifies the new delay time, in milliseconds (mS).

nDelay Specifies the new delay time, in milliseconds (mS).

Remarks

Sets the delay times for the tooltip in milliseconds.

See also: GetDelayTime

COXToolTipCtrl::SetMargin

Members | Overview

void SetMargin(LPRECT lprc)

Parameters

lprc Address of a RECT structure that contains the margin information to be set. The members of the RECT structure do not define a bounding rectangle, but rather the top, bottom, left and right distances (in pixels) between the text and the the edge of the tooltip window.

Remarks

Sets the top, left, bottom, and right margins for a tooltip window. A margin is the distance, in pixels, between the tooltip window border and the text contained within the tooltip window.

See also: GetMargin

COXToolTipCtrl::SetMaxTipWidth

Members | Overview

int SetMaxTipWidth(int nWidth)

Parameters

nWidth The maximum width of a tooltip window, or zero (0) if this width is to be calculated automatically. The maximum tooltip width value does not indicate a tooltip window's actual width. Rather, if a tooltip string exceeds the maximum width, the control breaks the text into multiple lines, using spaces to determine line breaks. If the text cannot be segmented into multiple lines, it will be displayed on a single line. The length of this line may exceed the maximum tooltip width.

Return Value

The previous maximum tooltip window width.

Remarks

Sets the maximum tooltip window width.

See also: GetMaxTipWidth

COXToolTipCtrl::SetTipBkColor

Members | Overview

void SetTipBkColor(COLORREF clr)

Parameters

clr The new background color, as a COLORREF structure.

Remarks

Sets the background color for all tools maintained by this control. If the value is CLR_DEFAULT then the default system color is used.

See also: GetTipBkColor

COXToolTipCtrl::SetTipTextColor

<u>Members</u> | <u>Overview</u>

void SetTipTextColor(COLORREF clr)

Parameters

clr The new text color.

Remarks

Sets the text color for all tools maintained by this control. If the value is CLR_DEFAULT then the default system color is used.

COXToolTipCtrl::SetToolInfo

Members | Overview

void SetToolInfo(OXTOOLINFO* pToolInfo)

Parameters

pToolInfo A pointer to an OXTOOLINFO structure that specifies the information to be set.

Remarks

Applies the information that a tooltip maintains for a tool.

See the <u>overview</u> for a description of the OXTOOLINFO structure.



The ultimate set of Internet/Intranet products for C++, MFC and ATL developers. Dundas TCP/IP products allow for simple Email integration, fully secure real-time data streaming, and much, much more!

Need to Internet-enable an application, but don't have the time or necessary experience?

The Dundas TCP/IP client edition may be just what you need. This product lets you Internet-enable an application with incredible ease! Includes many client side protocols, such as Email (SMTP, POP3, IMAP4), FTP, and Web (HTTP). more

Need to create scalable Internet/Intranet applications fast?

The Dundas TCP/IP Enterprise Edition includes all of the components necessary for creating highly scalable client and server applications. more

Need to secure your Internet/Intranet data?

The Dundas TCP/IP Security Add-on provides industry standard SSL/TLS security for applications using the Dundas TCP/IP Client or Enterprise editions. more

CUT_RAS Overview

Copyright © Dundas Software Ltd. 1996-1999, All Rights Reserved

CUT RAS Class Members

Dependencies and Related Files

The RAS Dialup class (CUT_RAS) is part of the Ultimate TCP/IP client and enterprise editions and provides easy to use modem dialing capabilities. This class allows you to create and modify dial-up entries, dial and hang up connections and more. CUT_RAS works within MFC, ATL and straight SDK projects.

CUT_RAS works with the main system phonebook.

To utilize your own errorcodes you can use UTExtErr.h.

Sample Code:

The following code demonstrates how to set up a dialog box in order to dial the first entry in a system phone book, and a user defined WM (window message) is created to inform us of the dialup progress.

NOTE: for this demo to work you must have Dial Up Networking installed. Please refer to <u>CUT_RAS Dependencies and Related Files</u> for a listing of dependency files.

To run this sample code follow the steps below:

- 1. Open a new win32 project.
- 2. Insert a new dialog (named IDD_DIALOG1).
- 3. Rename the OK button of the dialog to IDC_DIAL.
- 4. Add an edit control named IDC_EDIT1.

5. Add a new header file "stdafx.h" that contains the following four lines of code:

```
#ifndef __UT_RAS12345_STDAFX
#define __UT_RAS12345_STDAFX
#include <windows.h>
#endif // UT_RAS12345_STDAFX
```

- 6. Locate and add ut_RAS.h & ut_RAS.cpp to the project.
- 7. Create a new c++ source file (lets call it main.cpp).
- 8. Copy and past the code shown below
- 9. Run the program.

For an MFC sample please see the project included with the install program for this class.

```
#include "stdafx.h" // the stdafx file

//The header file of CUT_RAS class

#include "ut_ras.h"

//resource header for a dialog (IDD_DIALOG1) which contains a canc

// a dial button and a read only edit box (IDC_EDIT1)

#include "resource.h"
```

// User defined window message that will be sent back to us by the

```
// CUT_RAS class to inform us of the dial up progress
#define WM_CUT_RAS_DIALSTATUS WM_USER+1
// Dialog proc Prototype
BOOL CALLBACK DialupDlgProc(HWND hwndDlg, UINT message
// Window Main function
int CALLBACK WinMain(HINSTANCE hInstance,HINSTANCE ,LP
{
//create modal dialog box
DialogBox(hInstance,MAKEINTRESOURCE(IDD_DIALOG1),NUL
return 0;
}
// Dialog proc
BOOL CALLBACK DialupDlgProc(HWND hwndDlg, UINT message
{
// a static instance of the CUT_RAS object
```

```
static CUT_RAS ras;
// Pool the window messages
switch (message) {
case WM_INITDIALOG:
{
// upon initializing this dialog lets instruct the CUT_RAS object to sen-
// our user defined message to report the dialup status.
ras.SetDialStatusCallback (hwndDlg,WM_CUT_RAS_DIALSTATUS
return 1;
}
// handle user inputs
case WM_COMMAND:
switch (LOWORD(wParam))
{
// user pressed the cancel button
case IDCANCEL:
{
```

```
// abort any bending dial attempt if any
ras.CancelDial();
// Hang up the current connection
ras.HangUp ();
// Exit closing this dialog
EndDialog(hwndDlg, IDOK);
break;
}
// user pressed the dial button
case IDC_DIAL:
{
// create a phone entry name place holder
LPRASENTRYNAME ren = new RASENTRYNAME [sizeof(RASEN
// clear any data in the new allocated memory
memset(ren,0,sizeof(RASENTRYNAME));
```

```
// initialize the RAS dll and enumerate the available entries
ras.EnumEntries();
// get the first entry
ras.GetEntry ( ren,0);
// dial the first entry in the phonebook
ras.Dial(ren->szEntryName);
// reclaim the allocated memory
delete [] ren;
break;
}
}
break;
// if this incoming message is the same as the one we specified as the d
// then get the string from the Lower word of the message parameter
// and display it in the status edit box.
```

```
case WM_CUT_RAS_DIALSTATUS:
{
SetDlgItemText(hwndDlg, IDC_EDIT1,(LPCSTR)(LPARAM)lParam)
return 1;
}
// the window is closing
case WM_CLOSE:{
// abort any pending dial attempts (if any exist)
ras.CancelDial();
// Hang up the current connection
ras.HangUp ();
EndDialog(hwndDlg, IDOK);
break;
}
return 0;
}
```

CUT_RAS Class Members

Copyright © Dundas Software Ltd. 1996-2000. All Rights Reserved

Overview

CUT RAS

Class constructor.

~CUT_RAS

Destructor of the class.

InitRAS

Loads in the RAS DLLs dynamically.

OnError

Callback function which is raised when an error occurs. You can use this function to provide extended error handling and debugging capabilities.

Dial

Dials a given phonebook entry.

HangUp

Call this function to hang up a currently active

RAS connection.

SetDialStatusCallback

Sets the window which will receive messages

during dialing.

GetDialState

Call this function to retrieve the current state of the dialup process. If this is not available then the last dialup state is retrieved.

CancelDial

Cancels the current dial-up process.

IsConnected

Call this function to check the state of the current connection.

EnumDevices

Enumerates all of the available RAS dialup devices.

GetDeviceCount

Retrieves the number of available RAS dialup devices.

GetDevice

Call this function to retrieve device information.

Populates the supplied RASDEVINFO structure with the device information.

EnumConnections

Call this function to enumerate all of the running RAS connections and store them in an internal array of RASCONN structures.

GetConnectionCount

Retrieves the current number of RAS connections.

GetConnection

Fills in the supplied RASCONN structure with the connection information for the specified running connection.

EnumEntries

Call this function to enumerate all of the phonebook entry names for the main phonebook (the CUT_RAS class only supports the main phonebook).

GetEntryCount

Returns the number of available phonebook entries for the main phonebook (the CUT_RAS class only supports the main phonebook).

GetEntry

Call this function to fill in the given RASENTRYNAME structure with a phonebook entry's name.

GetEntryProperties

Retrieves the properties for a given phonebook

entry.

SetEntryProperties

Call this function to create or modify a

phonebook entry.

<u>SetDialEntryParams</u>

Sets the dialing parameters for a given

phonebook entry.

GetDialEntryParams

Retrieves the dialing parameters for a given

phonebook entry.

DoesEntryExist

Call this function to see if the specified

phonebook entry exists in the main RAS

phonebook.

<u>ValidateEntryName</u>

Checks to see if the specified entry name is a

valid name for a new phonebook entry.

ClearEntryPassword

Call this function to clear the password from a

given phonebook entry.

DeleteEntry

Deletes an entry from a phone book.

RenameEntry

Call this function to rename the given phonebook entry.

GetEntryPhoneNumber

Retrieves the phone number and area code for the given phonebook entry from the main phonebook.

<u>GetEntryUserName</u>

Retrieves the main phonebook's username for the given phonebook entry.

GetEntryPassword

Retrieves the main phonebook's password for the specified phonebook entry.

GetLastRASError

Retrieves the errorcode from the last RAS function called.

GetRASErrorString

Returns the appropriate error string corresponding to the passed RAS error code.

CUT_RAS Dependencies and Related Files

Copyright © Dundas Software Ltd. 1996-2000, All Rights Reserved

Overview

SOURCE Ut_ras.cpp

INCLUDE Ut_err.h

Ut_ras.h

LIBRARIES RasApi32.dll OR Rnaph.dll (comes with Dial Up Networking).

SAMPLE <INSTALLDIR>\RasDial\Sample\Ras\TestApp.dsw

CUT_RAS::~CUT_RAS

Members | Overview

virtual ~CUT_RAS()

Remarks

Destructor function. Frees the loaded RAS DLL and reclaims any locally allocated memory.

CUT_RAS::CUT_RAS

Members | Overview

CUT_RAS()

Remarks

Constructor.

CUT_RAS::CancelDial

Members | Overview

void CancelDial()

Remarks

Cancels the current dial-up process that was initiated with <u>Dial</u>. **CancelDial** is usually used inside of the Dial callback message handler.

See <u>SetDialStatusCallback</u> for more details.

Once **CancelDial** is called the <u>Dial</u> function will clean up after itself and return without completing the dial operation.

CUT_RAS::ClearEntryPassword

Members | Overview

int ClearEntryPassword(LPCSTR szEntryName)

Parameters

szEntryName Phonebook entry to delete the password from.

Return Value

UTE_SUCCESS Operation completed successfully.

UTE ERROR Function call failed (see GetLastRASError

for more details).

UTE_RAS_LOAD_ERROR Unable to load the RAS DLLs.

Remarks

Call this function to clear the password from a given phonebook entry. This can also be performed by using the <u>SetDialEntryParams</u> function, but **ClearEntryPassword** encapsulates the operation.

See also: <u>SetDialEntryParams</u>

CUT_RAS::DeleteEntry

Members | Overview

int DeleteEntry(LPCSTR szEntryName)

Parameters

szEntryName Name of the phonebook entry to delete.

Return Value

UTE_SUCCESS Operation completed successfully.

UTE ERROR Operation failed (see <u>GetLastRASError</u> for

more details).

UTE_RAS_LOAD_ERROR Unable to load the RAS DLLs.

Remarks

Call this function to delete an entry from the main phone book.

CUT_RAS::Dial

Members | Overview

int Dial(LPCSTR szEntry,LPCSTR szUserName = NULL, LPCSTR szPassword = NULL,LPCSTR szNumber = NULL)

Parameters

szEntry The name of the phone-book entry to dial.

szUserName A string containing the user's username. This string is

used to authenticate the user's access to the remote

access server.

szPassword A string containing the user's password. The password is

used to authenticate the user's access to the remote

access server.

szNumber A string containing an overriding phone number. An

empty string ("") indicates that the phone-book entry's phone number should be used. NOTE: If *szEntry* is ""

then szNumber cannot be NULL.

Return Value

UTE_SUCCESS Operation completed successfully.

UTE_ERROR Operation failed.

UTE RAS DIALINIT ERROR Error occurred when the dialing process

was initiated (see **GetLastRASError** for

more details).

UTE_RAS_DIAL_ERROR Error occurred during the dialing process.

Use the **GetDialState** function to retrieve

more detailed information.

UTE RAS LOAD ERROR Unable to load the RAS DLLs.

Remarks

Dials a given phonebook entry. The *szUserName*, *szPassword* and *szNumber* parameters are optional and can be NULL. To override information located in the dialup entry you would then specify these parameters (the phonebook entry itself will not be modified).

Under Win98 and NT this function can dial without a phonebook entry by using the given information and the first available dial-up device. Although a dialup connection can be made without specifying a phonebook entry it is recommended that you do so for reliability purposes.

This function does not return until the dialing has either been successfully completed or an error occurs.

To monitor the connection use the <u>SetDialStatusCallback</u> function to send dial status information (as a message) to a given window. To retrieve more information when this function returns use the <u>GetDialState</u> function.

See also: <u>DoesEntryExist</u>

Example

In the following example we will enumerate the available phonebook ϵ

```
#include "stdafx.h" // this header includes nothing but the #include <w
#include "ut ras.h" // the header files for the CUT RAS class
// resource header for a dialog (IDD_DIALOG1) that contains a read of
#include "resource.h"
// our user defined window message for dialup status
#define WM_CUT_RAS_DIALSTATUS WM_USER+1
// prototype of the only dialog procedure
BOOL CALLBACK DialupDlgProc(HWND hwndDlg, UINT message
// windows main entry point
int CALLBACK WinMain(HINSTANCE hInstance,HINSTANCE ,LP
{
// create modal dialog box
DialogBox (hInstance, MAKEINTRESOURCE (IDD\_DIALOG1), NULI \\
return 0;
```

```
// enumerate the available phonebook entries and attempt to connect to
BOOL CALLBACK DialupDlgProc(HWND hwndDlg, UINT message
{
// static instance of the CUT_RAS class
static CUT_RAS ras;
// dispatch the window messages
switch (message) {
// initializing the dialog
case WM INITDIALOG:
{
// we want the CUT_RAS class to send us the WM_CUT_RAS_DIAL:
// informing us of the progress of the dialup attempt
ras.SetDialStatusCallback (hwndDlg,WM_CUT_RAS_DIALSTATUS
return 1;
}
// if the message is the one we have asked the CUT_RAS to use
```

```
// then inform us of the dialup attempt progress
case WM_CUT_RAS_DIALSTATUS:
{
// Update the edit box with the string passed by the CUT_RAS class.
// The lower word of the message parameter is the string describing the
SetDlgItemText(hwndDlg, IDC_EDIT1,(LPCSTR)(LPARAM)lParam)
return 1;
}
// if the message is a user command
case WM_COMMAND:
switch (LOWORD(wParam))
{
// the user clicked on the Cancel button
case IDCANCEL:
{
// cancel any pending new dialup attempt on this instance of the CUT_
ras.CancelDial ();
```

```
// hang-up the current connection
ras.HangUp ();
// cLose the dialog
EndDialog(hwndDlg, IDOK);
break;
}
// the user pressed the Dial button
case IDC_DIAL:
{
LPRASENTRYNAME ren = new RASENTRYNAME [sizeof(RASEN
// get the first entry
ras.EnumEntries();
memset(ren,0,sizeof(RASENTRYNAME));
ras.GetEntry (ren,0);
// dial the first entry
ras.Dial(ren->szEntryName);
```

```
break;
break;
case WM_CLOSE:
{
ras.CancelDial ();
ras.HangUp ();
EndDialog(hwndDlg, IDOK);
break;
}
return 0;
}
```

CUT_RAS::DoesEntryExist

Members | Overview

int DoesEntryExist(LPCSTR szEntryName)

Parameters

szEntryName The name of the main phonebook entry to check for.

Return Value

UTE_SUCCESS Operation completed successfully.

UTE_ERROR Operation failed.

UTE_RAS_LOAD_ERROR Unable to load the RAS DLLs.

Remarks

Call this function to see if the specified phonebook entry exists in the main RAS phonebook.

CUT_RAS::EnumConnections

Members | Overview

int EnumConnections()

Return Value

UTE_SUCCESS Operation completed successfully.

UTE_RAS_ENUM_ERROR Enumeration error occurred (see GetLastRASError for more details).

UTE_RAS_LOAD_ERROR Unable to load the RAS DLLs.

Remarks

Call this function to enumerate all of the running RAS connections and store them in an internal array of RASCONN structures.

Once enumerated the data can be retrieved by using the <u>GetConnectionCount</u> and <u>GetConnection</u> functions. **EnumConnections** MUST be called for these functions to succeed.

See also: GetConnection | GetConnectionCount

Example

//retrieves all available RAS connections

CComboBox* cb3 = (CComboBox*)GetDlgItem(IDC_COMBO3);
m ras.EnumConnections();

```
int cnt = m_ras.GetConnectionCount();
RASCONN rc;
for(x = 0; x < cnt; x++){
    m_ras.GetConnection(&rc,x);
    cb3->AddString(rc.szEntryName);
}
cb3->SetCurSel(0);
```

CUT_RAS::EnumDevices

Members | Overview

in EnumDevices()

Return Value

UTE_SUCCESS Operation completed successfully.

UTE_RAS_ENUM_ERROR Enumeration error occurred (see <u>GetLastRASError</u> for more details).

UTE_RAS_LOAD_ERROR Unable to load the RAS DLLs.

Remarks

Call this function to enumerate all of the available RAS dialup devices. Stores the devices in an internal array of RASDEVINFO structures. Once it is enumerated the data can be retrieved by using the GetDeviceCount and GetDeviceCount and GetDevice functions. EnumDevices MUST be called for these functions to succeed.

Refer to the MSDN library for more information on the RASDEVINFO structure.

See also: GetDeviceCount | GetDevice

Example

//retrieve all available dialup devices

CComboBox* cb2 = (CComboBox*)GetDlgItem(IDC_COMBO2);

```
m_ras.EnumDevices();
cnt = m_ras.GetDeviceCount();
RASDEVINFO rdi;
for(x = 0; x < cnt; x++){
    m_ras.GetDevice(&rdi,x);
cb2->AddString(rdi.szDeviceName);
}
cb2->SetCurSel(0);
```

CUT_RAS::EnumEntries

Members | Overview

int EnumEntries()

Return Value

UTE_SUCCESS Operation completed successfully.

UTE_RAS_ENUM_ERROR Enumeration error occurred (see GetLastRASError for more details).

UTE_RAS_LOAD_ERROR Unable to load the RAS DLLs.

Remarks

Call this function to enumerate all of the phonebook entry names for the main phonebook (this free version of the CUT_RAS class only supports the main phonebook) and stores them in an internal array of RASENTRYNAME structures.

Once enumerated the data can be retrieved by using the <u>GetEntryCount</u> and <u>GetEntry</u> functions. **EnumEntries** MUST be called for these functions to succeed.

Refer to the MSDN library for more information on the RASENTRYNAME structure.

See also: GetEntryCount | GetEntry

Example

//enumerates all main phonebook entries

```
CComboBox* cb = (CComboBox*)GetDlgItem(IDC_COMBO1);
m_ras.EnumEntries();
int cnt = m_ras.GetEntryCount();
RASENTRYNAME ren;
for(int x = 0; x < cnt; x++){
    m_ras.GetEntry(&ren,x);
    cb->AddString(ren.szEntryName);
}
cb->SetCurSel(0);
```

CUT_RAS::GetConnection

Members | Overview

int GetConnection(LPRASCONN rc,DWORD index)

Parameters

rc A pointer to an existing RASCONN structure.

index A zero-based index (0 to (GetConnectionCount()-1)), specifying the RAS connection from which information will be retrieved.

Return Value

UTE SUCCESS Operation completed successfully.

UTE ERROR An error occurred and there is no data to

return.

UTE_INDEX_OUTOFRANGE The given index does not point to a valid

RASCONN entry.

UTE_RAS_LOAD_ERROR Unable to load the RAS DLLs.

Remarks

Call this function to fill in the given RASCONN structure with the connection information for the specified running connection. A RASCONN structure contains the phonebook entry's name as well as the device's type and name.

Refer to the MSDN library for more information on RASCONN structures.

See also: GetConnectionCount

Example

```
//retrieves and shows connection information for all running connection
CComboBox* cb3 = (CComboBox*)GetDlgItem(IDC_COMBO3);
m_ras.EnumConnections();
int cnt = m_ras.GetConnectionCount();
RASCONN rc;
for(x = 0; x < cnt; x++){
    m_ras.GetConnection(&rc,x);
    cb3->AddString(rc.szEntryName);
}
cb3->SetCurSel(0)
```

CUT RAS::GetConnectionCount

Members | Overview

DWORD GetConnectionCount()

Return Value

The number of available connections. Returns -1 if an error occurs.

Remarks

Call this function to retrieve the number of connections found during the enumeration. <u>EnumConnections</u> needs to be called before this function will return any valuable information.

See also: EnumConnections | GetConnection

Example

//retrieves and displays the number of available connections

```
CComboBox* cb3 = (CComboBox*)GetDlgItem(IDC_COMBO3);
m_ras.EnumConnections();
int cnt = m_ras.GetConnectionCount();
RASCONN rc;
for(int x = 0; x < cnt; x++){
    m_ras.GetConnection(&rc,x);
    cb3->AddString(rc.szEntryName);
```

```
}
cb3->SetCurSel(0);
```

CUT_RAS::GetDevice

Members | Overview

int GetDevice(LPRASDEVINFO rdi,DWORD index)

Parameters

rdi A pointer to an existing RASDEVINFO structure.

index A zero-based index (0 to (GetDeviceCount()-1)). Specifies the

device from which information will be retrieved.

Return Value

UTE_SUCCESS Operation completed successfully.

UTE_ERROR Operation failed and there is no data to be

returned.

UTE_INDEX_OUTOFRANGE The given index does not point to a valid

device entry.

UTE_RAS_LOAD_ERROR Unable to load the RAS DLLs.

Remarks

Call this function to fill in the given RASDEVINFO structure with the specified device information. RASDEVINFO contains the device's type and name.

Refer to the MSDN library for more information on RASDEVINFO structures.

See also: EnumDevices | GetDeviceCount

```
//retrieves and lists all available RAS devices
```

```
CComboBox* cb2 = (CComboBox*)GetDlgItem(IDC_COMBO2);
m_ras.EnumDevices();
int cnt = m_ras.GetDeviceCount();
RASDEVINFO rdi;
for( int x = 0; x < cnt; x++){
    m_ras.GetDevice(&rdi,x);
    cb2->AddString(rdi.szDeviceName);
}
cb2->SetCurSel(0);
```

CUT_RAS::GetDeviceCount

Members | Overview

DWORD GetDeviceCount()

Return Value

DWORD The number of devices found, or -1 if an error occurs.

Remarks

Call this function to retrieve the number of available devices. EnumDevices MUST be called before this function will return any valuable information.

See also: GetDevice | EnumDevices

Example: see <u>GetDevice</u>

CUT_RAS::GetDialEntryParams

Members | Overview

int GetDialEntryParams(LPCSTR szEntryName,LPRASDIALPARAMS pRasDialParams, BOOL* bClearPassword= NULL)

Parameters

szEntryName A phonebook entry name.

pRasDialParams A pointer to a RASDIALPARAMS structure.

bClearPassword If TRUE then the password has been cleared, if FALSE then the password is valid.

Return Value

UTE_SUCCESS Operation completed successfully.

UTE_ERROR Operation failed.

UTE_NULL_PARAM *pRasDialParams* is NULL.

UTE_RAS_LOAD_ERROR Unable to load the RAS DLLs.

Remarks

Call this function to retrieve the dialing parameters for the given phonebook entry.

You MUST use this function in order to retrieve the username and password for a phonebook entry.

The RASDIALPARAMS also contains other information as well.

See also: GetEntryPassword | GetEntryUserName | GetEntryPhoneNumber

CUT_RAS::GetDialState

Members | Overview

DWORD GetDialState()

Return Value

DWORD The LOWORD contains the the current (or last) state of the dialup process (see RASCONNSTATE in the MSDN library for a complete listing of states). The HIWORD contains the errorcode which applies to the current state (for a complete listing of errorcodes see the RASERROR.H header file of the Windows SDK or the "RAS Error Values" topic in the MSDN library).

Remarks

Call this function to retrieve the current state of the dialup process. If this is not available then the last dialup state is retrieved. This is useful for determining the state of the dialup after the <u>Dial</u> function has completed.

See also: GetRASErrorString

CUT_RAS::GetEntry

Members | Overview

int GetEntry(LPRASENTRYNAME ren,DWORD index)

Parameters

ren A pointer to an existing RASENTRYNAME structure.

index A zero-based index (0 to (GetEntryCount()-1)). Determines the entry from which to retrieve information.

Return Value

UTE_SUCCESS Operation completed successfully.

UTE_ERROR Operation failed and there is no data to be

returned.

UTE_INDEX_OUTOFRANGE The supplied index does not point to a

valid phonebook entry.

UTE_RAS_LOAD_ERROR Unable to load the RAS DLLs.

Remarks

Call this function to fill in the given RASENTRYNAME structure with the phonebook entry's name.

See also: EnumEntries | GetEntryCount

```
// stores the first phonebook entry's name to a RASENTRYNAME stru
LPRASENTRYNAME ren = new RASENTRYNAME [sizeof(RASEN

// get the first phonebook entry
ras.EnumEntries();
memset(ren,0,sizeof(RASENTRYNAME));
ras.GetEntry (ren,0);
delete [] ren;
```

CUT_RAS::GetEntryCount

Members | Overview

DWORD GetEntryCount()

Return Value

DWORD Number of available phonebook entries, or -1 if an error occurs.

Remarks

Returns the number of phonebook entries. <u>EnumEntries</u> MUST be called before this function will return any valuable information.

Refer to the MSDN library for more information on RASDEVINFO structures.

See also: GetEntry | EnumEntries

m_ras.GetEntry(&ren,x);

Example

//retrieves phonebook entries and displays them

```
CComboBox* cb = (CComboBox*)GetDlgItem(IDC_COMBO1);

m_ras.EnumEntries();

int cnt = m_ras.GetEntryCount();

RASENTRYNAME ren;

for(int x = 0; x < cnt; x++){
```

```
cb->AddString(ren.szEntryName);
}
cb->SetCurSel(0);
```

CUT_RAS::GetEntryPassword

Members | Overview

int GetEntryPassword(LPCSTR szEntryName,LPSTR szPassword, long nMaxLen)

Parameters

szEntryName A phonebook entry.

szPassword Pointer to a string buffer that will store the password.

nMaxLen Length of the szPassword buffer.

Return Value

UTE_SUCCESS Operation completed successfully.

UTE_ERROR The function failed (see <u>GetLastRASError</u>

for more details).

UTE BUFFER TOO SHORT The password string is larger than the

specified nMaxLen.

UTE_NULL_PARAM The passed buffer is NULL.

UTE RAS LOAD ERROR Unable to load the RAS DLLs.

Remarks

Returns the main phonebook's password for the specified phonebook entry. This function is more efficient than the <u>GetEntryProperties</u> function when it comes to retrieving a password.

See also: <u>GetEntryProperties</u> | <u>GetEntryUserName</u> | <u>GetEntryPhoneNumber</u>

CUT_RAS::GetEntryPhoneNumber

Members | Overview

int GetEntryPhoneNumber(LPCSTR szEntryName, LPSTR szPhoneNumber, long nPhoneNumberLen, LPSTR szAreaCode, long nAreaCodeLen)

Parameters

szEntryName The desired phonebook entry.

szPhoneNumber Pointer to a string buffer that will store the retrieved

phone number.

nPhoneNumberLen Length of the szPhoneNumber buffer.

szAreaCode Pointer to a string buffer that will store the retrieved

area code.

nAreaCodeLen Length of the szAreaCode buffer.

Return Value

UTE SUCCESS Operation completed successfully.

UTE ERROR Function failed (see GetLastRASError for

more details).

UTE_RAS_LOAD_ERROR Unable to load the RAS DLLs.

Remarks

Retrieves the main phonebook number and area code for the given phonebook entry. This function reduces the amount of work required to retrieve a phone number compared to using the <u>GetEntryProperties</u> function directly.

See also: GetEntryProperties

```
// retrieves an entry phone number and displays it if the number was su
if (m_pRas->GetEntryPhoneNumber( m_szEntryName,szTemp,sizeof(
{
    SetDlgItemText(IDC_PHONE_NUMBER,szTemp);
    SetDlgItemText(IDC_AREA_CODE,szAreaCode);
}
```

CUT_RAS::GetEntryProperties

Members | Overview

int GetEntryProperties(LPCSTR szEntryName, LPRASENTRY pRasEntry,DWORD* pnRasEntryLen)

Parameters

szEntryName A phonebook entry.

pRasEntry A pointer to an allocated RASENTRY structure. The

dwSize member of the structure must be set to

"sizeof(RASENTRY)".

pnRasEntryLen The size of the *pRasEntry* buffer.

Return Value

UTE_SUCCESS Operation completed successfully.

UTE_ERROR Function failed (see GetLastRASError for

more details).

UTE_BUFFER_TOO_SHORT The supplied buffer was too small.

UTE_RAS_LOAD_ERROR Unable to load the RAS DLLs.

Remarks

Retrieves the properties for a given phonebook entry. This function takes a pointer to a RASENTRY structure which must already exist.

NOTE: the RASENTRY structure can be followed by an array of null-

terminated alternate phone number strings. The last string is terminated by two consecutive null characters. The dwAlternateOffset member of the RASENTRY structure contains the offset to the first string.

Set the *nRasEntryLen* parameter to the total length of the given data buffer. If the buffer is too small then the function will fail.

To help ensure success call the function with *pRasEntry* set to NULL and *nRasEntryLen* set to 0. If the function returns with a value of **UTE_BUFFER_TOO_SHORT** then *nRasEntryLen* will contain the required size of the buffer. Allocate a buffer of the required size and call the function again.

```
// if the entry name is passed then a modification call is made
// to this dialog instead of an add call.
if (m_pRas->GetEntryProperties(m_szEntryName, NULL,&size) == U
{
    m_rasEntry = new RASENTRY[size];
    m_rasEntry->dwSize = size;

// retrieve all properties for the selected book entry
if (m_pRas->GetEntryProperties(m_szEntryName, m_rasEntry,&size)
{
```

```
// get the username
if (m_pRas->GetEntryUserName(m_szEntryName, szTemp,sizeof(szT
{
SetDlgItemText(IDC_USER,szTemp);
}
// get the password
if (m_pRas->GetEntryPassword(m_szEntryName, szTemp,sizeof(szTe
{
SetDlgItemText(IDC_PASSWORD,szTemp);
}
// get phone number and area code
if (m_pRas->GetEntryPhoneNumber(m_szEntryName,szTemp,sizeof(s))
{
SetDlgItemText(IDC_PHONE_NUMBER,szTemp);
SetDlgItemText(IDC_AREA_CODE,szAreaCode);
}
```

CUT_RAS::GetEntryUserName

Members | Overview

int GetEntryUserName(LPCSTR szEntryName,LPSTR szUserName, long nMaxLen)

Parameters

szEntryName Phonebook entry to retrieve the username from.

szUserName Pointer to a string buffer that will store the retrieved

username value.

nMaxLen Length of the szUserName buffer.

Return Value

UTE_SUCCESS Operation completed successfully.

UTE ERROR Function failed (see GetLastRASError for

more details).

UTE_BUFFER_TOO_SHORT The username string is larger than the

specified *nMaxLen* parameter.

UTE_RAS_LOAD_ERROR Unable to load the RAS DLLs.

Remarks

Returns the main phonebook's username for the given phonebook entry. This function reduces the amount of work required to retrieve a username compared to using the GetEntryProperties function directly.

See also: GetEntryProperties | GetEntryPassword

```
// retrieve username for the specified phonebook entry
char szTemp[ ]
if (m_pRas->GetEntryUserName(m_szEntryName, szTemp,sizeof(szT
{
    SetDlgItemText(IDC_USER,szTemp);
}
```

CUT_RAS::GetLastRASError

Members | Overview

DWORD GetLastRASError()

Return Value

DWORD Errorcode of the last RAS operation.

Remarks

This function returns the errorcode from the last RAS function called. For a complete listing of errorcodes see the RASERROR.H header file of the Windows SDK or the "RAS Error Values" topic in the MSDN library.

Calling this function is useful if extended error information is required.

See also: GetErrorString | GetRASErrorString

```
// if error is a RAS error display the appropriate error string,
// otherwise use CUT_ERR::GetErrorString.
int CUT_TestRas::OnError(int nError)
{
   if (nError != UTE_SUCCESS && nError != UTE_BUFFER_TOO_SF)
{
   // if the error is a RAS error dial then display the error string
```

```
if (nError != UTE_RAS_DIAL_ERROR )
{
MessageBox(NULL,CUT_ERR::GetErrorString (nError),"ON ERROI
}else
{
// lets make sure that the error is within the limit
if (GetLastRASError() > RASBASE && GetLastRASError() <= ERR()</pre>
{
MessageBox(NULL,GetRASErrorString (GetLastRASError()),"ON El
}
return nError;
}
```

CUT_RAS::GetRASErrorString

Members | Overview

LPCSTR GetRASErrorString(DWORD error)

Parameters

error DWORD specifying the RAS error code.

Return Value

LPCSTR A string describing the error code.

Remarks

This function returns the appropriate error string corresponding to the passed RAS error code. For a complete listing of errorcodes see the RASERROR.H header file of the Windows SDK or the "RAS Error Values" topic in the MSDN library.

Calling this function is useful if extended error information is required.

See also: GetErrorString

```
//display an errorstring based on an errorcode
int CUT_TestRas::OnError(int nError)
{
if (nError != UTE_SUCCESS && nError != UTE_BUFFER_TOO_SF
```

```
{
// if the error is a RAS dialing error display the error string
if (nError != UTE_RAS_DIAL_ERROR )
{
MessageBox(NULL,CUT_ERR::GetErrorString (nError),"ON ERROI
}
else
{
// lets make sure that the error is within the limits of our errorcodes
if (GetLastRASError() > RASBASE && GetLastRASError() <= ERR</pre>
MessageBox(NULL, GetRASErrorString (GetLastRASError()), "ON El
}
return nError;
```

CUT_RAS::HangUp

Members | Overview

int HangUp()

int HangUp(HRASCONN rasConn)

Parameters

rasConn A RAS connection handle.

Return Value

UTE_SUCCESS Successfully hung up the connection or it

was not connected to begin with.

UTE_RAS_HANDLE_ERROR NULL RAS handle supplied.

UTE_RAS_LOAD_ERROR Unable to load the RAS DLLs.

Remarks

Call this function to hang up a currently active RAS connection. If *rasConn* is not specified then the function attempts to hang up the current connection that was initiated with the <u>Dial</u> function. Otherwise it will hang up the specified RAS connection by using the supplied RAS connection handle.

RAS connection handles can be retrieved using the **EnumConnections** function.

Once the hang-up process starts this function will wait until the hang-up is complete. Hang-up times vary and depend on the modem. Usually 1-3

seconds is the norm.

If the hang-up status cannot be monitored (which is the case on some older versions of Win95) then a 4 second waiting period is applied to ensure a proper hang-up.

```
//hangs up a specified connection
void CTestAppDlg::OnHangUp()
{
RASCONN rConection;
int index= 0;
//if we are currently trying to connect then exit the connection attempt
m_ras.CancelDial();
//Get the Selected connection out of the ComboBox
index = m_ctlConnections.GetCurSel();
if (index != LB_ERR && index < m_ctlConnections.GetCount())</pre>
{
m_ras.GetConnection(&rConection,index);
m_ras.HangUp(rConection.hrasconn);
}
```

```
else
m_ras.HangUp();
//display the connections
CComboBox* cb3 = (CComboBox*)GetDlgItem(IDC_COMBO3);
cb3->ResetContent();
cb3->SetWindowText("");
m_ras.EnumConnections();
int cnt = m_ras.GetConnectionCount();
RASCONN rc;
for(int x = 0; x < cnt; x++){
m_ras.GetConnection(&rc,x);
cb3->AddString(rc.szEntryName);
}
cb3->SetCurSel(0);
}
```

CUT_RAS::InitRAS

Members | Overview

BOOL InitRAS()

Return Value

UTE_SUCCESS Operation completed successfully.

UTE RAS LOAD ERROR Could not load the RAS DLLs.

Remarks

This function loads in the RAS DLLs dynamically. This allows the class to be used in applications where RAS may or may not be installed. For Win95 OSR2 and later the RASAPI32.DLL is used. For earlier Win95 versions RNAPH.DLL is used for some functions. This DLL may be redistributed since it is not a part of RAS but a patch for early Win95 versions.

This function SHOULD BE CALLED prior to calling other functions of CUT_RAS. Note that InitRAS is called internally by some functions of the CUT_RAS class such as EnumEntries and EnumConnections.

CUT RAS::IsConnected

Members | Overview

BOOL IsConnected()

Return Value

Returns TRUE if connected, otherwise it returns FALSE.

Remarks

Call this function to check the state of the current connection. If the connection is still good then TRUE is returned but if the connection has been terminated FALSE is returned.

This function is useful for monitoring a connection throughout its lifetime, since disconnections are quite common for a variety of reasons (ISP timeouts, voice priority over data on busy phone lines, noisy phone lines, etc.).

```
// update the connection status of this application's instance of the RAS
void CTestAppDlg::OnTimer(UINT nIDEvent)
{
    CString strStatus;
// check if we are connected
if (m_ras.IsConnected())
strStatus = "Connected";
```

```
else
```

```
strStatus = "Disconnected";
SetDlgItemText(IDC_STATUS,strStatus);
CDialog::OnTimer(nIDEvent);
}
```

CUT_RAS::OnError

Members | Overview

virtual int OnError(int nError)

Parameters

nError The errorcode of the error that occurred.

Return Value

UTE_SUCCESS Operation completed successfully.

UTE_ERROR Operation failed.

Remarks

All functions which have return error codes call this function. Override **OnError** to provide extended error handling and debugging capabilities.

See also: GetLastRASError | GetRASErrorString

Example

//display an errorstring via the OnError callback

```
int CUT_TestRas::OnError(int nError)
{
if (nError != UTE_SUCCESS && nError != UTE_BUFFER_TOO_SF
{
```

```
if (nError != UTE_RAS_DIAL_ERROR )
{
    MessageBox(NULL,CUT_ERR::GetErrorString (nError),"ON ERROF
}else
{
    if (GetLastRASError() > RASBASE && GetLastRASError() <= ERROR
MessageBox(NULL,GetRASErrorString (GetLastRASError()),"ON El}
}
return nError;
}</pre>
```

CUT_RAS::RenameEntry

Members | Overview

int RenameEntry(LPCSTR szEntryName,LPCSTR szNewEntryName)

Parameters

szEntryName Phonebook entry to rename.

szNewEntryName New name of the phonebook entry.

Return Value

UTE_SUCCESS Operation completed successfully.

UTE ERROR Function failed (see GetLastRASError for

more details)

UTE_RAS_LOAD_ERROR Unable to load the RAS DLLs.

Remarks

Call this function to rename the given phonebook entry.

CUT_RAS::SetDialEntryParams

Members | Overview

int SetDialEntryParams(LPCSTR szEntryName,LPRASDIALPARAMS pRasDialParams, BOOL bClearPassword = FALSE)

Parameters

szEntryName A phonebook entry name.

pRasDialParams A pointer to a RASDIALPARAMS structure.

bClearPassword Set this to TRUE to remove the password from the given entry. Set this to FALSE to set the password.

Return Value

UTE_SUCCESS Operation completed successfully.

UTE ERROR Function failed (see GetLastRASError for

more details).

UTE NULL PARAM The pRasDialParams parameter is NULL.

UTE_RAS_LOAD_ERROR Unable to load the RAS DLLs.

Remarks

Call this function to set the dialing parameters for the given phonebook entry. The phonebook entry MUST exist for this function to succeed.

Use **SetDialEntryParams** to store the username and password for a phonebook entry. The RASDIALPARAMS structure also contains other

optional information.

Refer to the MSDN library for more information on the RASDIALPARAMS structure.

```
// set the dialing parameters for a dialup entry
if (m_pRas->SetEntryProperties(m_szEntryName,m_rasEntry)== UTF
{
// now set the Dialup properties
LPRASDIALPARAMS pDialParam = new RASDIALPARAMS[sizeo
memset(pDialParam,0,sizeof(RASDIALPARAMS));
pDialParam->dwSize= sizeof(RASDIALPARAMS);
// Set the Entry name
strcpy(pDialParam->szEntryName,m_szEntryName);
// set the phone number
GetDlgItemText(IDC_PHONE_NUMBER,pDialParam->szPhoneNum
// set the user name
```

```
GetDlgItemText(IDC_USER,pDialParam->szUserName,UNLEN);

// set the password string

GetDlgItemText(IDC_PASSWORD,pDialParam->szPassword,PW

// now update the entry's Dialup parameters

m_pRas->SetDialEntryParams (m_szEntryName,pDialParam,FAL delete [] pDialParam;
}
```

CUT RAS::SetDialStatusCallback

Members | Overview

int SetDialStatusCallback(HWND hWnd, int nMessageID)

Parameters

hWnd Handle of the window which will receive the window

messages.

nMessageID ID of the message to send.

Return Value

UTE_SUCCESS This function always return as successful.

Remarks

Sets the window which will retrieve messages during dialing (see the <u>Dial</u> function). Messages are sent each time the dialing state changes, and this function is useful for showing the user the current state within a window or dialog box.

The format of the message sent back by the class to the window specified by the *hWnd* handle is:

wParam The LOWORD contains a string indicating the current state of the dialup process. The HIWORD contains the errorcode which applies to the current state. See the RASERROR.H header file for a complete listing of errorcodes.

IParam

Points to a string buffer which describes the current state.

Examples

<u>Dial</u> is invoked as an asynchronous operation and the function returns immediately. The user must therefore specify a notification handler that the RAS object uses to inform the client application when the connection operation state changes or an error occurs. This handler is implemented below. For more information see the sample code in the <u>Overview</u> page.

MFC Sample

```
Header File
   // Generated message map functions
   //{{AFX_MSG(CTestAppDlg)
   afx_msg LRESULT OnDialStatus(WPARAM wParam, LPARAM
   . . . . . . .
   //}}AFX MSG
   DECLARE_MESSAGE_MAP()
Source File
   BEGIN_MESSAGE_MAP(CTestAppDlg, CDialog)
   //{{AFX_MSG_MAP(CTestAppDlg)
   ON_MESSAGE(WM_USER+1,OnDialStatus)
```

```
//}}AFX_MSG_MAP
   END_MESSAGE_MAP()
(in OnInitDialog)
// specify a notification handler that RAS uses
// to inform the client application when the connection
// operation state changes or an error occurs
m_ras.SetDialStatusCallback(m_hWnd,WM_USER);
LRESULT CTestAppDlg::OnDialStatus(WPARAM wParam, LPARAN
DWORD dwError = HIWORD(wParam);
UpdateData();
CString szTemp = (LPCSTR)m_strStatus;
szTemp += "\r\n";
szTemp += (LPCSTR)lParam;
SetDlgItemText(IDC_EDIT1,szTemp);
if(dwError){
```

```
MessageBox(m_ras.GetRASErrorString(dwError));
}
m_ctlStatus.LineScroll(100000);
return 0;
}
SDK Sample
    ras.SetDialStatusCallback (hwndDlg,WM_USER+1);
    case WM_USER+1 :
    {
        SetDlgItemText(hwndDlg, IDC_EDIT1,(LPCSTR)(LPARAM)lPa return 1;
     }
}
```

CUT_RAS::SetEntryProperties

Members | Overview

int SetEntryProperties(LPCSTR szEntryName, LPRASENTRY pRasEntry,DWORD nRasEntryLen = sizeof(RASENTRY)

Parameters

szEntryName Name of the phonebook entry to add or update.

pRasEntry A pointer to a RASENTRY structure which contains the

data for the entry to add or update. The dwSize member

of the structure must be set to sizeof(RASENTRY). NOTE: This can optionally be followed by a list

of alternative phone numbers.

nRasEntryLen Size of the RAS entry. Defaults to sizeof(RASENTRY). It

can be set if a list of alternative phone numbers are also

available.

Return Value

UTE SUCCESS Operation completed successfully.

UTE ERROR Function failed (see GetLastRASError for

more details).

UTE_RAS_LOAD_ERROR Unable to load the RAS DLLs.

Remarks

Call this function to create or modify a phonebook entry. If the given entry does not exist then a new one is created, but if the entry already exists

then it is overwritten. This function takes a pointer to a RASENTRY structure which contains all of the fields required to make a complete phonebook entry except for the user name and password (see SetDialEntryParams). NOTE: the RASENTRY structure can be followed by an array of null-terminated alternate phone number strings. The last string is terminated by two consecutive null characters. The dwAlternateOffset member of the RASENTRY structure contains the offset to the first string.

Set the *nRasEntryLen* parameter to the total length of the given data. *nRasEntryLen* has a default value of **sizeof**(RASENTRY).

Use <u>ValidateEntryName</u> to make sure that a new entry name is valid.

See also: GetEntryProperties | GetEntry

Example

{

// read the values from the dialog's controls and update the

```
// Ras Entry structure then Set the Properties
if (m_bModify){
// Update the Dialogs controls based on the entry's Data
GetDlgItemText(IDC\_ENTRYNAME, m\_szEntryName, RAS\_MaxEntulation (IDC\_ENTRYNAME, m\_szEntulation (IDC\_ENTRYNAE, m\_szEntulation (IDC\_ENTRYNAME, m\_szEntulation (IDC\_ENTRYNAME,
if (m_pRas->DoesEntryExist (m_szEntryName) == UTE_SUCCESS)
{
// ask if the user wants to modify the existing entry
MessageBox("An entry with this name already exists");
}else
{
if (m_pRas->ValidateEntryName (m_szEntryName) == UTE_SUCCE;
{
// get the new phone number
GetDlgItemText(IDC_PHONE_NUMBER,m_rasEntry->szLocalPhone
// get the new are code number
GetDlgItemText(IDC AREA CODE,m rasEntry->szAreaCode,RAS
if (m_pRas->SetEntryProperties(m_szEntryName,m_rasEntry)== UTE
```

```
{
// now set the Dialup properties
LPRASDIALPARAMS pDialParam = new RASDIALPARAMS[sizeo
memset(pDialParam,0,sizeof(RASDIALPARAMS));
pDialParam->dwSize= sizeof(RASDIALPARAMS);
// Set the Entry name
strcpy(pDialParam->szEntryName ,m_szEntryName);
// set the phone number
GetDlgItemText(IDC_PHONE_NUMBER,pDialParam->szPhoneNum
// set the user name
GetDlgItemText(IDC_USER,pDialParam->szUserName,UNLEN);
// set the password string
GetDlgItemText(IDC_PASSWORD,pDialParam->szPassword,PWLEI
// now update the entry's Dialup parameters
m_pRas->SetDialEntryParams (m_szEntryName,pDialParam,FALSE)
delete [] pDialParam ;
```

```
CDialog::OnOK ();
}
}else // the selected name is not valid name
MessageBox("Name not valid");
}
}else // if we are not adding a new entry then just exit
CDialog::OnOK ();
}
```

CUT_RAS::ValidateEntryName

Members | Overview

int ValidateEntryName(LPCSTR szEntryName)

Parameters

szEntryName A string specifying the phonebook entry to check.

Return Value

UTE SUCCESS Operation completed successfully.

UTE ERROR Function failed (see GetLastRASError for

more details).

UTE_RAS_LOAD_ERROR Unable to load the RAS DLLs.

Remarks

Checks to see if the specified phonebook entry is a valid name for a new entry. This function will also fail if the phonebook entry already exists.

CUT_ERR Overview

Copyright © Dundas Software Ltd. 1996-1999, All Rights Reserved

CUT ERR Class Members

Dependencies and Related Files

CUT_ERR can be thought of as a sort of 'namespace' class which organizes the declaration of error codes and their associated error text. The motivation for the class was to move error code definitions out of the realm of the preprocessor and into the code itself, so that values would be visible during debug sessions, etc.

The class uses an enumeration to define error codes that will be used for other classes in the library. These could have been declared as const members, but not all compilers support this feature.

The constructor, destructor, copy constructor and assignment operator are declared as private members. This class cannot be instantiated or used as a base class.

Along with the enumeration of errorcodes, two static public methods are declared. GetErrorString() allows the user to retrieve a text description of the error. GetSocketError() can be used during development to retrieve the error associated with a winsock API call that has returned SOCKET ERROR.

GetErrorString calls the static private method InitErrorStrings() the first time it is called. InitErrorStrings() then initializes an array of string constants.

Adding New Codes

The list of errors available (in uterr.h) is easily extended by adding the new constant (e.g. UTE_THE_ERROR_DUNDAS_FORGOT) to the

CUT_ERROR_CONSTANTS enumeration and then adding a line of code to the initialization method InitErrorStrings. You might like to leave the core code alone, so we've provided a mechanism for extending the errors without modifying uterr.h.

An additional header 'UTExtErr.h' can be found in the Include directory. You can use this file to insert new errorcodes and their corresponding text into the CUT_ERR enumeration. You can edit this file in place if you want to extend your new errors to all projects, or you can use it on a perproject basis by copying it to your project directory. (Make sure your project pre-processor additional includes contains ".\" so that this version is included from within uterr.h).

Inside UTExtErr.h you will find the following (commented) sample:

```
#ifndef _EXT_ERR_STRINGS // Add your error constants here. See e
// UTE_DATABASE_CONNECT_ERROR,
// UTE_NOT_SUPPORTED,
#else
// Add your error string associations here. See example below:
// pStrings[UTE_DATABASE_CONNECT_ERROR] =_T("Database of the content of the content
```

Add the mnemonic for your new code followed by a comma, and specify a string to be returned from GetErrorString().

Usage

While the enumerated 'constants' are available without the need for scoping, the static <u>GetErrorString</u> needs to be prefaced by the class name. (This is the 'namespace' aspect of the class.)

For example, to display all of the available errorcodes and their numeric value one could write:

```
for (int i = 0; i < CUT_MAX_ERROR; i++)
```

cout << i << ". " << CUT_ERR::GetErrorString(i) << endl;</pre>

This may be something you want to do in order to have a numeric reference for debugging. We don't explicitly assign values to the codes inside the enumeration, so you can not just refer to the header to see what 'error 43' means. Being able to call <u>GetErrorString</u> provides some compensation.

Many of the client and server classes that use these enumerations as return codes do so via a virtual <u>OnError</u> method which can be overridden or modified to trace error messages through the use of <u>GetErrorString</u>. In most cases we leave it up to you to implement this so that you can then select the appropriate output device.

CUT_ERR Class Members

Copyright © Dundas Software Ltd. 1996-1999, All Rights Reserved

Overview

GetErrorString

Retrieves the error text for a given error.

GetSocketError

Retrieves the error and string description of a winsock error for a socket.

CUT_ERR Dependencies and Related Files

Copyright © Dundas Software Ltd. 1996-1999, All Rights Reserved

Overview

INCLUDE uterr.h

CUT_ERR::GetErrorString

Members | Overview

static const TCHAR* GetErrorString(int err)

Parameters

err The enumerated error code.

Return Value

Returns a text description of the error that has occurred. If the call is made with an *err* value that is out of range the string then "CUT_ERR::GetErrorString: ErrorString array improperly initialized" will be returned.

Remarks

You will usually call this method to obtain a description of an error returned from one of the client or server Ultimate TCPIP v3.0 classes. Note that you must use the CUT_ERR:: scoping preface - this is a static method of a class that is never instantiated.

Example

```
// test for the success/failure of a SendFile operation.
if(UTE_SUCCESS != (res = MyDerivedWSClient.SendFile("myfile.tx
#ifdef _DEBUG
cout << CUT_ERR::GetErrorString(res) << endl;</pre>
```

```
#endif
switch (res)
{
case UTE_ABORTED:
// send aborted by user
break;
case UTE_FILE_OPEN_ERROR:
// unable to open specified file
break;
case UTE_CONNECT_TERMINATED:
// remote connection terminated
break;
default:
assert(0); // should never happen...
break;
}
}
```

CUT_ERR::GetSocketError

Members | Overview

static LPCSTR GetSocketError(SOCKET * s, int * result=NULL)

Parameters

s A pointer to the socket to be queried.

result An optional integer that will receive the actual winsock error code.

Return Value

A string containing a readable description of the last error to occur on the socket (if available).

Remarks

In most cases, the predefined return codes are sufficient to indicate what went wrong with a call to a method of one of the Ultimate TCPIP classes.

Occasionally you will find that you need to deal with a return value of SOCKET_ERROR from a call to a winsock API function, particularly when extending a class.

This ususally involves a call to WSAGetLastError, and a switch statement to determine which error constant was returned. Often you will need the switch to determine a course of action to take, but if all you are concerned with is knowing what the error was, **GetSocketError** can be a time and space saver.

WSAGetLastError will return the last error that occurred on any socket in the current thread. The code in **GetSocketError** takes a pointer to the

socket in order to query the socket. This is done by using the GetSockOpt call, rather than WSAGetLastError.

If the socket is invalid the return string will be "GetSockOpt error: WSAENOTSOCK", indicating that the call to GetSockOpt failed. Other errors may be returned as GetSockOpt errors.

If you want the actual error code pass a pointer to the *result* parameter.

For some TCPIP methods you should call WSAGetLastError to determine the socket error if a particular call returns UTE_ERROR or -1. This is indicated where appropriate.

If the socket is still valid (i.e. the method was not involved with creating or destroying the socket) you may find it more convenient to call **GetSocketError** instead of WSAGetLastError.

GetSocketError is intended more as a development aid than a library export.

See also: GetErrorString

Example

```
// if an error occurs connecting then in the OnError callback
// we are just examining the error and then returning an errorcode.
if(connect(m_socket, (LPSOCKADDR)&m_sockAddr, sizeof(m_sock
{
    // you can place a break point here to examine the string
    LPCSTR str = CUT_ERR::GetSocketError(&m_socket);
    return OnError(UTE_SOCK_CONNECT_FAILED);
}
```

Dundas Mailer Control 1.0

Copyright © Dundas Software Ltd. 2000, All Rights Reserved.

Overview | Properties | Methods

The Dundas Mailer Control 1.0 is a free commercial email control which utilizes the SMTP, MIME and NNTP protocols to send email messages and post news articles. It has been designed for use within Active Server Pages (ASP) and can be used by any person or organization that is hosting web sites using Microsoft's Internet Information Server. It is also used in conjunction with the Dundas <u>Upload</u> Control when dealing with uploaded files to be sent either as attachments or embedded html objects.

The Dundas Mailer control is extremely fast and reliable. The following is a listing of the control's features:

- Specify multiple recipients by using the <u>TOs</u>, <u>CCs</u>, and <u>BCCs</u> collections.
- Ability to post news articles to news groups.
- Supports both text-based and html-based email.
- Support for embedded html objects within an html-based email (audio files, images, etc.).
- Ability to send multiple file attachments.
- ReplyTo property (implemented as a collection).
- Message headers can be set to non-ASCII values with the <u>EncodeHeader</u> method.
- You can specify the character set to be used for the message body (for non-English emails).

- Priority property used to indicate low, high or normal priorities.
- A <u>QuickSend</u> method which allows you to send an email with a minimal amount of code.
- Use multiple SMTP relays for improved speed and reliability.
- Validate email addresses with the ValidateAddress method.
- Use one or more specified DNS servers for Direct Sending of email.
- ConfirmRead property.
- <u>ReturnReceipt</u> property.
- Optionally load the message body from file.
- Supports the addition of custom headers to an email.
- MSDN integrated help.
- Full control over various time out properties.

The Dundas Mailer control exposes two methods which will send an email message. The <u>QuickSend</u> method lets you send an email with a minimal amount of code, and its advantage is its simplicity and ease of use. <u>SendMail</u> also sends an message but it does require that certain properties and collections to be set first. The advantage of **SendMail** is that it incorporates **ALL** features of the control.

To post an article to a news group you can call the **PostArticle** method.

The ProgID of the control is "Dundas.Mailer".

Most methods of this control will throw an exception if an error occurs. Trap for the success/failure of mail operations by using an **On Error Resume Next** statement and examine the Err object after a function call.

See Also: <u>Product Overview</u> | <u>How to Use the Dundas Mailer Control</u> | <u>Dundas Upload Control</u>

Product Overview (Dundas Mailer Control 1.0)

Overview | Properties | Methods

The Dundas Mailer Control 1.0 is a free commercial email control which utilizes the SMTP, MIME and NNTP protocols to send email messages and post news articles. It has been designed for use within Active Server Pages (ASP) and can be used by web developers or ASP developers to quickly and painlessly add web-based email capabilities to their ASP applications.

The Dundas <u>Upload</u> control, which also incorporates an Upload Progress component and a State Server executable, can be used in conjunction with the <u>Mailer</u> control when working with uploaded files in order to provide developers with web-based email solutions which fully support a web-farm environment.

<u>Features List Installation/Redistribution</u>
How to use the Documentation

Features List

The following is just some of the features of the Dundas Mailer control:

- Choose between text and/or html-based email.
- 2. Fully supports embedded html objects within an html-based email (audio files, graphics files, etc.).
- 3. Ability to post news articles to news groups. You have the option of posting to the root directory or as a response to a particular article.
- 4. Specify multiple recipients by using the TOs, CCs, and BCCs collections.
- 5. Message headers and the message body can be set to non-ASCII values.

- 6. Validate your email addresses to save your bandwidth.
- 7. Specify multiple SMTP and DNS servers to make your email operations lightning quick.
- 8. Fully supports the use of custom headers.
- 9. Optionally prioritizes your messages. You can also request confirmation when the user retrieves and/or reads an email message.
- 10. Send multiple attachments with your messages.
- 11. MSDN integrated html help.

Installation/Redistribution

Controls and Samples Install

The name of the install file is: "AspMail.exe".

The windows account under which the install runs needs to have administrative priviledges for a successful installation.

The following files will be copied to the selected installation directory:

- 1) <u>ASP pages</u>: AutoResponse.asp, PostArticle.asp, PostArticle_Process.asp, SendMail.asp, SendMail_Process.asp, SendHtmlEmail.asp, SendHtmlEmail_Process.asp and Error.asp.
- 2) Various graphics files required by the ASP pages (InstallDirectory/Images/).
- 3) ReadMe.htm.

Start menu entries are as follows:

Dundas Software | Free Products | Mailer and Upload Controls | ReadMe

Documentation Install

To uninstall the documentation run the install file again (FPDocInst.exe) and choose either "Remove From MSDN" if you integrated into the MSDN or "Remove All" if you did not integrate into the MSDN.

How to use the Documentation

Documentation for the Mailer control consists of the following:

- 1. Compiled html help (optionally integrated into the MSDN). The html help provides detailed explanations for all members of the control as well as a control overview. Other topics which should be especially useful are:
 - How to Use the Dundas Mailer Control
 - Overview (Dundas Mailer Control 1.0)
 - QuickStart (Dundas Mailer Control 1.0)
 - Tutorial 1: Sending an Email with an Attachment
 - Tutorial 2: Sending an Html Email with an Embedded Image
 - Tutorial 3: Posting to a Newsgroup
- ReadMe.htm. This file is distributed with the Mailer control, and consists of installation information, release notes and a troubleshooting section.

How to Use the Dundas Mailer Control

Overview | Properties | Methods

The Dundas Mailer Control is a free fully commercial email control which allows you to send both html and text-based emails as well as post articles to news groups. It is used in conjunction with the <u>Dundas Upload Control</u> when dealing with uploaded files, html embedded objects and file attachments.

Two methods are used for sending an email: <u>QuickSend</u> and <u>SendMail</u>. Refer to the sections below for a detailed explanation on how to use the Mailer control with either of these two methods. To post articles to news groups you can utilize the <u>PostArticle</u> method. See <u>Tutorial 3</u>: <u>Posting to a NewsGroup</u> for further details.

Refer to the section <u>below</u> for details on how to use the Mailer and Upload controls together. You can also examine <u>Tutorial 1: Sending an Email with an Attachment</u> or <u>Tutorial 2: Sending an Html Email with an Embedded Image</u> for example source of this.

NOTE: All sending methods utilize the <u>SMTPRelayServers</u> and <u>DNSServers</u> collections. To maximize the speed of the mail operation populate these collections before sending the email message.

Using QuickSend

- 1. Create an instance of the Mailer control. The ProgID of the control is "Dundas.Mailer".
- Optionally set properties (including collections) used by the QuickSend method. Unlike SendMail (which incorporates all of the control's functionality) QuickSend only utilizes some of the control's properties and collections (all of which are optional). See <u>QuickSend</u> for a listing of these properties.
- 3. Call QuickSend to send an email. Notice that this method's

- arguments are used to set any required message headers.
- 4. Error trap for the success/failure of the operation by using an On Error Resume Next statement and examine VBScript's Err object (Err.Number will be a non-zero value if an error occurred) immediately following the function call.
- 5. Destroy the instance of the Mailer control.

Advantage - Lets you send an email with a minimal amount of coding.

Using SendMail

- Create an instance of the Mailer control. The ProgID of the control is "Dundas.Mailer".
- 2. Unlike the QuickSend method SendMail does not use function arguments at all. It relies on various properties and collections to set message headers and other message elements (e.g. <u>Body</u> property, <u>SMTPRelayServers</u> collection, <u>TOs</u> collection, <u>TimeOutConnect</u> property, etc.). For the function call to succeed you MUST specify a destination address by adding an <u>Address</u> object to one of the following collections: <u>TOs</u>, <u>CCs</u>, or <u>BCCs</u>. Use the <u>Add</u> method of the collections to do this.
- 3. Once you have specified a destination address set any other properties (collections incuded) that you would like to be used for the mailing operation.
- 4. Error trap for the success/failure of the operation by using an On Error Resume Next statement and examine VBScript's Err object (Err.Number will be a non-zero value if an error occurred) immediately following the function call.
- 5. Destroy the instance of the Mailer control.

<u>Advantage</u> - Although more coding is required to send an email compared to QuickSend the SendMail method offers you a tremendous amount of flexibilty and options for the mailing operation. It utilizes ALL

properties/collections of the control.

Using the Upload and Mailer Controls

Use the Dundas Upload control to save uploaded files and access form elements. Once you have saved the uploaded files to disk you can then access form elements via the <u>Form</u> collection. You can also iterate through all uploaded files via the Upload control's <u>Files</u> collection, which stores <u>UploadedFile</u> objects.

If you are just sending an attachment with the email then add the uploaded file to the Mailer control's <u>Attachments</u> collection. Use the **UploadedFile** object's **Path** property as the *FileName* argument of the <u>Attachments.Add</u> method call. Then call the **SendMail** method to send the email with the attachment.

To embed uploaded files into an html email loop through the **Files** collection of the Upload control and add the **UploadedFile** objects to the **HtmlEmbeddedObjs** collection, and then embed the object by wrapping the appropriate html tags around the object in the Mailer control's **HtmlBody** property. You can use the **UploadedFile** object's **TagName** property to determine which file input box the object originated from, and you can also use VBScripts's InStr method in conjunction with the **UploadedFile** objects **ContentType** property to determine the type of the object. This lets you make sure that an object is not embedded into the html body of the email with the wrong tags around it.

There are four tutorials which illustrate the various send message operations:

- QuickStart (Illustrates how to send an email with a minimum of code).
- Tutorial 1: Sending an Email with an Attachment
- Tutorial 2: Sending an Html Email with an Embedded Image
- <u>Tutorial 3: Posting to a Newsgroup</u>

QuickStart (Dundas Mailer Control 1.0)

Overview | Properties | Methods

The following source code demonstrates how to send an email with both the <u>QuickSend</u> and the <u>SendMail</u> methods. The difference between the two methods is that **SendMail** incorporates all functionality of the control, while **QuickSend** only utilizes some of the control's properties. Please note that this sample sends the email messages with a minimal amount of code so not all of the Mailer control's functionality is demonstrated.

Sending an Email with the QuickSend Method <%
'most control methods throw an exception if an error occurs so we will
On Error Resume Next

Dim objMailer 'Mailer control

'create instance of Mailer control
Set objMailer = Server.CreateObject("Dundas.Mailer")

'send email

objMailer.QuickSend "FromMe@SomeServer.com", "ToSomeUser@SomeServer.com", "ToSomeUser.com", "ToSomeUser.c

Set objMailer = Nothing %>

Sending an Email with the SendMail Method <%

'most control methods throw an exception if an error occurs so we will On Error Resume Next

Dim objMailer 'Mailer control

```
'create instance of Mailer control
Set objMailer = Server.CreateObject("Dundas.Mailer")

'set Mailer control properties and collection items
objMailer.TOs.Add "martine@dundas.com"
objMailer.FromAddress = "FromMe@SomeServer.com"
objMailer.Subject = "Subject"
objMailer.Body = "This is the body."

'send email
objMailer.SendMail
'you can test for the success/failure of the operation by examining VBS
Set objMailer = Nothing
%>
```

Tutorial 1: Send an Email with an Attachment

Overview | Properties | Methods

The following source code demonstrates how to send an email along with a file attachment via an ASP page.

Files are uploaded by the user via file input elements in a form (with an EncType of "multipart/form-data"). Then the Dundas <u>Upload</u> Control is used to save uploaded files to disk. Once this is done we then iterate through all uploaded files via the Upload control's <u>Files</u> collection and add each file to the Mailer control's <u>Attachments</u> collection. Once all files have been added to this collection the email is sent with the <u>SendMail</u> method of the Mailer control.

Assumptions

- A form with an ENCTYPE of "multipart/form-data" is POSTING data to this ASP page.
- The form contains one or more file input elements (e.g. <input type="file" name="txtFile">

<% Dim objUpload 'stores upload control instance Dim objEmail 'stores mailer control instance Dim strPath 'stores path of the asp page Dim Index 'counter variable

'functions will throw an exception if not successful so On Error Resum On Error Resume Next

Set objUpload = Server.CreateObject("Dundas.Upload") 'Upload objec Set objEmail = Server.CreateObject("Dundas.Mailer") 'Mailer object

'create temporary directory to store uploaded files (if it doesn't already

' at the same directory level as this asp page strPath = Server.MapPath(".") & "\temp\" objUpload.DirectoryCreate strPath

'save the uploaded files to the temp directory.

'doing this populates the Upload control's collections

'note that we could also save to memory with the SaveToMemory methologiUpload.Save strPath

'add an Address object to the TOs collection (this specifies the destinat objEmail.TOs.Add "someone@somewhere.com"

'specify the message subject objEmail.Subject = "Some Subject"

'specify an SMTP server. Doing this increases the speed and reliability objEmail.SMTPRelayServers.Add "somesmtpserver.com"

'set the message body objEmail.Body = "This is the message body"

'now loop through all uploaded files (uploaded via file input boxes), an 'each uploaded file to the Mail control's Attachments collection 'NOTE: you can use either a For Each loop or a standard For loop here For Each Item in objUpload.Files

'Note: we are using the OriginalPath property of the UploadedFile obje' (which composes the Files collection) for the ContentName argument object. This lets the recipient of the email see the original filename of (e.g. SomePicture.jpg) instead of the name by which the attachment w which are saved to disk have a GUID preceding the original filename objEmail.Attachments.Add Item.Path,Item.OriginalPath

'now send the email objEmail.SendMail

'test for success/failure of the SendMail operation using VBScript's Err If Err.Number <> 0 Then

'an error occurred so output the relevant error string Response.Write "The following error occurred: " & Err.Description Else

'successful, so output a success message to user Response.Write "The email was successfully forwarded to the specified End If

Set objEmail = Nothing 'release resources
Set objUpload = Nothing
%>

Tutorial 2: Sending an Html Email with an Embedded Image

Overview | Properties | Methods

Files to be embedded are uploaded by the user via file input elements in a form (with an EncType of "multipart/form-data"). Then the Dundas Upload Control is used to save uploaded files to disk with its Save method (we could also save directly to memory as well). Once this is done we can then iterate through all uploaded files via the Upload control's Files collection and check to see if valid file types were uploaded for each file input element (lets assume that there are two file input boxes, one for an image file and the other for an audio file). If a valid file type was uploaded we add the file to the Mailer control's HtmlEmbeddedObjs collection, and then embed the object into the Mailer control's HtmlBody property. When we embed the object into the HtmlBody property we "wrap" the appropriate tags around the object to be embedded and identify it by using the contentID (cid). We then continue looping through the **Files** collection. Once the objects have been added to the HtmlEmbeddedObjs collection the email is sent via the SendMail method of the Mailer control.

Please note that you CAN NOT use ASP's Request object to retrieve form element values when the ENCTYPE of the form is "multipart/form-data". To retrieve form values you MUST call the **Save** method of the Upload control. Not only does this save the uploaded files to disk but it also populates the Upload controls collections, thereby letting you access form elements with the Upload control's <u>Form</u> collection (e.g. strToAddress = objUpload.Form("txtTO")).

Assumptions

- A form with an ENCTYPE of "multipart/form-data" is POSTING data to this ASP page.
- The form contains two (2) file input elements named "txtAudio" and "txtImage" (e.g. <input type="file" name="txtAudio">).

<% Dim objUpload 'stores an Upload control object
Dim objEmail 'stores a Mailer control object
Dim strPath 'stores path to a directory we create at same level as this A

'The SendMail function will throw an exception if the operation is 'unsuccessful, so we enable inline error trapping
On Error Resume Next

Set objUpload = Server.CreateObject("Dundas.Upload") 'Upload objec Set objEmail = Server.CreateObject("Dundas.Mailer") 'Mailer object

'create a directory at same level as this ASP page

' this directory is used to store the uploaded files (which are renamed w ' the original filename)

'NOTE: to delete these files you would have to use the ImpersonateUse
' the Upload control since the IUSR_ account SHOULD NOT have per strPath = Server.MapPath(".") & "\temp"
objUpload.DirectoryCreate strPath

'save the uploaded files. THIS POPULATES THE UPLOAD CONTR(objUpload.Save strPath

'specify the recipient of this message objEmail.TOs.Add "SomeUsername@SomeDomain.com"

'specify the subject of the email objEmail.Subject = "This is the subject."

'specify the sender of the message objEmail.FromAddress = "MyUsername@MyDomain.com"

'specify an SMTP Relay server. This increases the speed and reliability objEmail.SMTPRelayServers.Add "SomeSmtpServer.com"

'initialize the HtmlBody property, we'll throw a header into it objEmail.HTMLBody = "<Html><Head></Head><Body><H2>This i

```
'now lets loop through the uploaded files to be embedded.
```

For Each Item in objUpload.Files

'we will check to see which file input element is responsible for the upl If (Item.TagName = "txtAudio") Then

'we now know the file came from the txtAudio box, but lets make sure 'the user actually uploaded some sort of audio file by using InStr and t If InStr(1,Item.ContentType,"audio") Then

'the file is actually an audio type, so add it to the HtmlEmbeddedObjs ('and embed it into the HtmlBody property. Note that we set the Conter 'of the HtmlEmbeddedObj to the OriginalPath property of the uploade 'if the client email software displays the name of the file the filename 'Also note that we MAKE SURE that the ID for the embedded object is objEmail.HtmlEmbeddedObjs.Add Item.Path, 1, Item.OriginalPath objEmail.HtmlBody = objEmail.HtmlBody & "<BGSound src=cid:1>< End If

End If

If (Item.TagName = "txtImage") Then

'lets make sure user uploaded a valid image file

If InStr(1,Item.ContentType,"image") Then

objEmail.HtmlEmbeddedObjs.Add Item.Path, 2, Item.OriginalPath objEmail.HtmlBody = objEmail.HtmlBody & ""

End If

End If

Next

'finish html body by adding closing html tags objEmail.HTMLBody = objEmail.HTMLBody & "</body></html>"

'send the email objEmail.SendMail

'test for success/failure

If Err.Number <> 0 Then

'an error occurred so output error message

Response.Write "Sorry, the following error occurred: " & Err.Descripti Else

'success!

Response.Write "The html email was successfully sent." End If

'release resources
Set objEmail = Nothing
Set objUpload = Nothing
%>

Tutorial 3: Posting to a Newsgroup

Overview | Properties | Methods

The following source code demonstrates how to post an article to a specified news group at a news server.

We create an instance of the <u>Dundas Mailer Control</u>, set the <u>FromAddress</u> property (used as the ReplyTo address if there are no Address objects in the <u>ReplyTOs</u> collection) and then set the <u>Body</u> property which constitutes the article body. For this demonstration we will post an article in reply to an existing article which we read. To post our article underneath an existing article (as opposed to posting to the root directory) we will add a custom header named "References" to the Mailer control's <u>CustomHeaders</u> collection, and set its value to the ID of the article we are responding to.

<%

Dim objEmail 'stores instance of Mailer object

'PostArticle throws an exception if it is not successful so we use ' inline error trapping with an On Error Resume Next statement On Error Resume Next

'create instance of Mailer control
Set objEmail = Server.CreateObject("Dundas.Mailer")

'specify an SMTP server to increase the speed and reliability of the ope objEmail.SMTPRelayServers.Add "SomeSmtpServer.com"

'set the subject of the article objEmail.Subject = "This is the article subject."

'add a custom header named "References" to the CustomHeaders colled 'the value of the header to the ID of the article to post beneath (responsible beneath). CustomHeaders. Add "References", "TheArticleID"

'set the body of the article objEmail.Body = "This is the body of the article."

'set the FromAddress (This is used as the ReplyTo address if are no Ad 'objects in the ReplyTOs collection)
objEmail.FromAddress = "UserName@DomainName.com"

'now attempt to post the article (we will use a public Microsoft news goobjEmail. PostArticle "msnews.microsoft.com", "microsoft.a.test"

'now trap for success/failure of the article posting

If Err.Number <> 0 Then

'PostArticle failed.

Response.Write "The following error occurred: " & Err.Description Else

'PostArticle succeeded

Response.Write "The article was successfully posted."

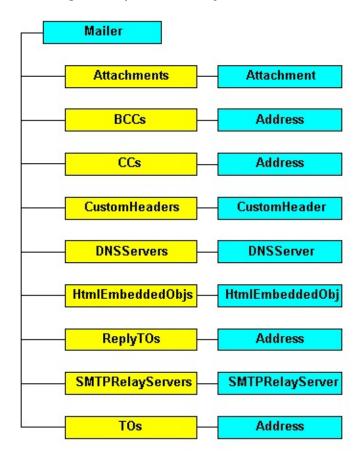
End If

Set objEmail = Nothing %>

Object Hierarchy (Dundas Mailer Control 1.0)

Overview | Properties | Methods

NOTE: The yellow rectangles represent collections while the blue rectangles represent objects.



See Also: Overview (Dundas Mailer Control 1.0) | How to Use the Dundas Mailer Control

Troubleshooting the Mailer Control Demos

The uninstall does not remove the UploadControl.dll and DSMailer.dll

The inetinfo.exe process may be referencing the dlls during the uninstall process. Reboot your computer and remove the dlls manually.

Object variable not set

This will occur as a result of referencing a collection item which does not exist (e.g. using the Form collection to reference a checkbox which is not checked).

To rectify this error check to make sure that the item you are referencing in the collection has the correct spelling. To test for an empty checkbox use the following line of code: "If objUpload.Form("chkMyCheckbox") Is Nothing Then ... Else...End If".

The relevant Err. Number will be "91".

Server.CreateObject failed. Invalid class string specified

The specified ProgID of either the Mailer or Upload control is incorrect. The ProgID of the Mailer control is "Dundas.Mailer", and the ProgID of the Upload control is "Dundas.Upload".

This can also be caused by a control which is not registered. Register the control in question by using regsvr32.exe.

HTTP Error 403 Forbidden: Execute Access Forbidden

You have not set up the permissions for your ASP files correctly. Use Windows Explorer to set up proper NTFS directory permissions if you are running Win NT or Win 2000 (Read access is required), and also make sure that the Web permissions set in MMC (Microsoft Management Console) are at least Read with Script Execute rights. Also make sure that the Samples/Images folder has Read permissions as well.

HTTP Error 404 File Not Found

The URL you are specifying in your browser is incorrect. Determine where your ASP pages are and enter the appropriate URL (e.g. http://127.0.0.1/RelativePathFromRootWebDirectory/PostArticle.ASP

Access is Denied

The appropriate permissions have not been set up for the deletion of files or folders. You need to either give the default web account (usually IUSR_MACHINENAME) Full Control NTFS permission for the folder in question or you can use the ImpersonateUser method of the Upload control to utilize a user account which does have the right to delete files/folders.

VBSCript RunTime Error "800a01ad" ActiveX component can not create object

Either the UploadControl.dll or DSMailer.dll does not have the appropriate permissions. Make sure that the default web account has Full Control rights for both dll's.

Error "800a001e" A required priviledge is not held by the client

This is probably caused by calling the ImpersonateUser method of the Upload control in conjunction with the "Run in Separate Memory Space" box being checked in your Home Directories property page (found in the Microsoft Management Console, or MMC). To get around this you can either grant the default web account "Act as Part of the Operating System" priviledges or alternatively you can just turn off the "Run in Separate Memory Space" option. Refer to either the MSDN or the Free Programs Documentation (examine the ImpersonateUser topic) for details on how to grant operating system priviledges.

Dundas Mailer Control 1.0 Properties

Copyright © Dundas Software Ltd. 2000, All Rights Reserved.

Overview | Properties | Methods

Body The text-based body of the email. Not used

with the **QuickSend** method.

BodyCharSet The character set to be used for the email's

text-based body.

ConfirmRead If this is set to TRUE and the recipient's server

supports this feature then a confirmation email will be sent to the supplied From address when

the email is read by the recipient.

<u>FromAddress</u> The email address of the person sending the

email.

FromName The name of the email sender. This is the

name which sometimes appears before the email address of the sender in the From field.

<u>HtmlBody</u> Use this property to specify the body of the

email in html format. Not used with the

QuickSend method.

HtmlBodyCharSet The character set to be used for the email's

html-based body.

<u>Organization</u> The organization sending the email.

<u>Priority</u> The priority rating of the email.

ReturnReceipt If this is set to TRUE and the recipient's server

supports this feature then a confirmation email will be sent to the supplied From address when

the email is retrieved by the recipient.

Subject The subject field of the email. Not used with the

QuickSend method.

TimeOutConnect The connection time out value.

<u>TimeOutReceive</u> The receive time out value.

<u>TimeOutSend</u> The send time out value.

Collection Properties

<u>Attachments</u> Collection of <u>Attachment</u> objects, specifying the

attachments to be sent with the email (not used

by the QuickSend method).

<u>CustomHeaders</u> Collection of <u>CustomHeader</u> objects, specifying

the custom headers to be added to the email

(not used by the QuickSend method).

<u>DNSServers</u> Collection of <u>DNSServer</u> objects, specifying the

DNS server(s) to be used. Improves the speed

and efficiency of the mail operation.

HtmlEmbeddedObjs Collection of HtmlEmbeddedObj objects,

specifying the objects which are embedded into

the **HtmlBody** of the email. Not used by the

QuickSend method.

SMTPRelayServers Collection of SMTPRelayServer objects,

specifying the SMTP server(s) to be used. Improves the speed and efficiency of the mail operation. If no SMTP servers are used then

the email is sent using DirectSend.

<u>TOs</u> Collection of <u>Address</u> objects, specifying the

recipients of the email (not used by the

QuickSend method).

CCs Collection of Address objects. This specifies

the CC recipients of the email. Not used by the

QuickSend method.

BCCs Collection of Address objects. This specifies

the BCC recipients of the email. Not used by

the QuickSend method.

ReplyTOs Collection of Address objects. This specifies

the addresses to be used for message replies.

Not used by the QuickSend method.

Body (Dundas Mailer Control 1.0)

Overview | Properties | Methods

Determines the body of the email (unless you are calling the **QuickSend** method).

Syntax

MailerObject.Body = [string]

The **Body** property syntax has the following parts:

Part	Description
string	A string expression which determines the body of the email.

Remarks

Set this property to specify the body of an email message.

The QuickSend method does not use this property.

Please note that the maximum allowable size of the email body is 32Kb.

NOTE: if you set both the **Body** property and the <u>HtmlBody</u> property before sending the message then the type of body displayed is dependent on the default settings of the recipient's email software. If the recipient defaults to html email then the body displayed will be determined by the **HtmlBody** property. However, if the recipient's email program defaults to text-based email then the **Body** property determines the body of the message.

See Also: SendMail

BodyCharSet (Dundas Mailer Control 1.0)

Overview | Properties | Methods

This property determines the character set to be used for the text-based body of the email.

Syntax

MailerObject.BodyCharSet = [string]

The **BodyCharSet** property syntax has the following parts:

Part	Description
string	Any valid character set. Defaults to "US-ASCII".

Remarks

Use this property to select a character set for the message body. The value of **BodyCharSet** is used in conjunction with the <u>Body</u> property which determines the text-based body of the email. Alternatively you can set the **BodyCharSet** property and load the message body from file via the <u>LoadBodyFromFile</u> method.

To select a character set for an html-based email use the <u>HtmlBodyCharSet</u> property.

See Also: **Body**

ConfirmRead (Dundas Mailer Control 1.0)

Overview | Properties | Methods

If this property is set to TRUE and the recipient's user agent (email software) supports this feature then a confirmation email will be sent when the email is read by the recipient.

Syntax

MailerObject.ConfirmRead = [boolean]

The **ConfirmRead** property syntax has the following parts:

Part	Description
	Set this to TRUE to request confirmation that the email has been read. Defaults to FALSE.

Remarks

If **ConfirmRead** is set to TRUE and this feature is also supported by the client's user agent then a notice will be sent back to the address in the email's From header confirming that the email has been read.

NOTE: **Make sure** you supply a valid email address for the From field when sending the message, since it is the address in the From header which determines where the confirmation reply is sent. If you are using **SendMail** then ensure that the **FromAddress** property has been set before sending the email. If you are using the **QuickSend** method make sure that you supply the correct email address for the *From* argument.

This property can be used in conjunction with all methods which send an email.

The **PostArticle** method does not utilize the **ConfirmRead** property.

Microsoft's Outlook supports this feature, while OutLook Express does not.

See Also: ReturnReceipt

FromAddress (Dundas Mailer Control 1.0)

Overview | Properties | Methods

The email address of the message sender.

Syntax

MailerObject.FromAddress = [string]

The **FromAddress** property syntax has the following parts:

Part	Description
	The email address the message is being sent from. Defaults to a zero-length string.

Remarks

If the **FromName** property is a zero-length string then this **FromAddress** property determines the contents of the From header, and when the client reads the message this email address will be what appears in the client software's From field (e.g. johnd@dundas.com).

If the <u>FromName</u> property has been set then both the **FromName** and **FromAddress** properties make up the From header of the email. When the client reads the message the From field will then contain either both properties (e.g. John Doe <johnd@dundas.com>) or just the **FromName** value, depending on the client's user agent.

Although this property does not have to be set when using <u>SendMail</u> (the function calls will not fail) we highly recommend that you do specify a valid email address for **FromAddress**.

NOTE: You MUST set this property if you want to use either the ConfirmRead property or the ReturnReceipt property.

If the <u>ReplyTo</u> collection has no items in it then the address used for replies is retrieved from this **FromAddress** property. You MUST SET the **FromAddress** property when posting an article with the <u>PostArticle</u> method.

See Also: FromName

FromName (Dundas Mailer Control 1.0)

Overview | Properties | Methods

The name of the email sender.

Syntax

MailerObject.FromName = [string]

The **FromName** property syntax has the following parts:

Part	Description
_	Name of the person sending the message. Defaults to a zero-length string (optional).

Remarks

The **FromAddress** property MUST be set if you want to use **FromName**.

If the **FromName** property is a zero-length string then the <u>FromAddress</u> property determines the contents of the From header of the email, and when the client reads the message this email address will be what appears in the client software's From field (e.g. johnd@dundas.com).

If the **FromName** property has been set then both the **FromName** and **FromAddress** properties make up the From header of the email. When the client reads the message the From field will then contain either both properties (e.g. John Doe <johnd@dundas.com>), or the value of **FromName**, depending on the client's user agent.

See Also: FromAddress

HtmlBody (Dundas Mailer Control 1.0)

Overview | Properties | Methods

The html body of the message.

Syntax

MailerObject.HtmlBody = [string]

The **HtmlBody** property syntax has the following parts:

Part	Description
	A string expression containing html code which constitutes the body of the email.

Remarks

Use this property to send the body of the email in an html format.

Using html for the body of your email allows you to embed html objects into your message body via the httml/embeddedObjs collection.

It is also possible to set the email body by using the <u>LoadBodyFromFile</u> method. Just make sure that the appropriate html tags are included in the ASCII file.

The **HtmlBody** property MUST CONTAIN all standard html headers that any html page would contain (e.g. <html><head></head><body></body></html>). It is the responsibility of the developer to supply these tags when setting the **HtmlBody** property.

The <u>QuickSend</u> method can only send text-based email. You must use <u>SendMail</u> in order to send an email with an html body.

If you set both the <u>Body</u> property and the **HtmlBody** property before sending the message then the type of body displayed is dependent on the default settings of the client's email software. If the client defaults to html email then the body displayed will be determined by the **HtmlBody** property. However, if the client's email program defaults to text-based email then the **Body** property determines the body of the message.

Note that you will need to replace any carriage return/linefeed characters with
 or tags from text gathered with textboxes if you want to display the email body on more than one line.

The <u>PostArticle</u> method can also use html but we strongly discourage this practice. Many servers WILL NOT accept articles in an html format.

The maximum allowable size of the email body is 32Kb.

Refer to <u>Tutorial 2: Sending an Html Email with an Embedded Image</u> for example source code on sending an html email.

See Also: Body | Dundas Upload Control | HtmlEmbeddedObjs Collection

HtmlBodyCharSet (Dundas Mailer Control 1.0)

Overview | Properties | Methods

This property determines the character set to be used for the body of an html-based email.

Syntax

MailerObject.HtmlBodyCharSet = [string]

The **HtmlBodyCharSet** property syntax has the following parts:

Part	Description
string	Any valid character set. Defaults to US-ASCII.

Remarks

Use this property to select a character set for the message body. The value of **HtmlBodyCharSet** is used in conjunction with the <u>HtmlBody</u> property which determines the html-based body of the email. Alternatively you can set the **HtmlBodyCharSet** property and load the message body from file via the <u>LoadBodyFromFile</u> method (make sure that the text file has the appropriate html tags in it).

To select a character set for text-based email use the <u>BodyCharSet</u> property.

See Also:

HtmlBody">httmlBody

TimeOutConnect (Dundas Mailer Control 1.0)

Overview | Properties | Methods

Retrieves or sets the connection time out value in seconds.

Syntax

MailerObject.TimeOutConnect = [long]

The **TimeOutConnect** property syntax has the following parts:

Part	Description
long	Any positive <i>long</i> expression. The time to wait for a connection to be established before cancelling the operation. Defaults to 30 seconds.

Remarks

You can increase this value if your connection attempts to the server continually time out.

Note that this timeout value is multiplied if you specify more than one SMTP or DNS servers. In otherwords, the timeout value is applied on a per connection attempt basis.

See Also: <u>TimeOutReceive</u> | <u>TimeOutSend</u>

Organization (Dundas Mailer Control 1.0)

Overview | Properties | Methods

This property determines the contents of the Organization header of the email, indicating the organization the message was sent from.

Syntax

MailerObject.Organization = [string]

The **Organization** property syntax has the following parts:

Part	Description
_	The organization the message was sent from. Defaults to a zero-length string.

Remarks

This method adds an Organization header which describes the organization to which the sender belongs, or to which the machine belongs. The intent of this header is to help identify the person posting the message, since host names are often so cryptic that it is difficult to determine the organization which sent the message.

See Also: FromAddress | FromName

Priority (Dundas Mailer Control 1.0)

Overview | Properties | Methods

This property determines the priority of the message.

Syntax

MailerObject.Priority = [integer]

The **Priority** property syntax has the following parts:

Part	Description
	Priority level of the message, ranging from one (1) to five (5). One is the highest priority while five is the lowest priority. Defaults to -1 (no priority rating).

Remarks

Priority determines the contents of the X-Priority header of the email. If the value of this property is -1 then the email message will not contain the X-Priority header.

The use of this header depends on the client email software being used by the recipient of the message. For example, Outlook Express assigns special icons to messages of high priority and low priority.

Usually a priority of 1 or 2 will be indicated as high priority while 4 or 5 will be indicated as a low priority.

ReturnReceipt (Dundas Mailer Control 1.0)

Overview | Properties | Methods

If this property is set to TRUE and the client's user agent (email software) supports this feature then confirmation will be sent to the address in the email's From header when the email is retrieved by the recipient.

Syntax

MailerObject.ReturnReceipt = [boolean]

The **ReturnReceipt** property syntax has the following parts:

Part	Description
	Set this to TRUE to request confirmation that the email has been retrieved by the recipient. Defaults to FALSE.

Remarks

If **ReturnReceipt** is set to TRUE and this feature is also supported by the client's user agent then a notice will be sent back to the address in the email's From header confirming that the email has been retrieved by the recipient.

NOTE: **Make sure** you supply a valid email address for the From field when sending the message, since it is the address in the From header which determines where the confirmation reply is sent. If you are using **SendMail** ensure that the **FromAddress** property has been set before sending the email. If you are using the **QuickSend** method make sure that you supply the correct email address for the *From* argument.

This property can be used in conjunction with ALL methods which send an email. <u>PostArticle</u>, however, does not utilize the **ReturnReceipt** property.

Microsoft's Outlook supports this feature, while OutLook Express does not.

See Also: ConfirmRead

Subject (Dundas Mailer Control 1.0)

Overview | Properties | Methods

Use this property to set the Subject field of your message if you are not using the <u>QuickSend</u> method to send the email.

Syntax

MailerObject.Subject = [string]

The **Subject** property syntax has the following parts:

Part	Description
string	A string expression describing the subject matter of the email.

Remarks

Use this property to set the Subject field of the email if you are using the <u>SendMail</u> method to send the email message. The **QuickSend** method sets the Subject field via its *Subject* argument.

We strongly recommend that you set this property before sending an email.

See Also: Body | HtmlBody

TimeOutReceive (Dundas Mailer Control 1.0)

Overview | Properties | Methods

Use this property to retrieve or set the receive time out value.

Syntax

MailerObject.TimeOutReceive = [long]

The **TimeOutReceive** property syntax has the following parts:

Part	Description
long	The amount of time (in seconds) to wait for a reply from the SMTP server. Defaults to 30 seconds.

Remarks

You can increase this property if a requested mail operation continually times out on you.

See Also: <u>TimeOutConnect</u> | <u>TimeOutSend</u>

TimeOutSend (Dundas Mailer Control 1.0)

Overview | Properties | Methods

Use this property to retrieve or set the send time out value.

Syntax

MailerObject.TimeOutSend = [long]

The **TimeOutSend** property syntax has the following parts:

Part	Description
long	The amount of time (in seconds) that a message will wait to be sent before the operation times out.

Remarks

This property setting determines how long the message will wait in the queue before the send mail operation times out.

See Also: <u>TimeOutConnect</u> | <u>TimeOutReceive</u>

Dundas Mailer Control 1.0 Methods

Copyright © Dundas Software Ltd. 2000, All Rights Reserved.

Overview | Properties | Methods

EncodeHeader Use this method to encode an html header. Useful

for non-ASCII header values.

LoadBodyFromFile Loads the message body from a file.

QuickSend Sends an email without having to set any properties

or collections.

<u>PostArticle</u> Call this method to post an article to a news group.

SendMail Sends an email. Unlike **QuickSend** this does not use

any arguments, so you must set properties and

collections before calling this method.

<u>ValidateAddress</u> Call this function if you want to validate an email

address.

EncodeHeader (Dundas Mailer Control 1.0)

Overview | Properties | Methods

Use this method to encode an html header. Useful for non-US-ASCII header values.

Syntax

[string] = MailerObject.EncodeHeader(Text As String, CharSet As String, [Encoding As Long = 0])

The **EncodeHeader** method syntax has the following parts:

Part	Description
Text	The string value of the header which is to be encoded.
CharSet	The character set to be used for the header.
Encoding	Optional. The type of encoding to be used, either BASE64 (1), or QUOTEDPRINTABLE (0). Defaults to 0.
string	The encoded version of the header.

Return Values

A string expression which is the encoded value of the *Text* argument.

Remarks

This method lets you specify character sets other than the standard US-ASCII character set for the message headers. This is particularly useful for setting the Subject header and the FromName component of the From header to non-English values.

To use "Quoted Printable" encoding set the *Encoding* argument to zero (0) or use the default value. To use BASE64 encoding specify a value of one (1) for the *Encoding* argument.

Use either <u>BodyCharSet</u> or <u>HtmlBodyCharSet</u> to specify a non-"US-ASCII" character set for the message body.

See Also: <u>CustomHeader Object</u> | <u>CustomHeaders Collection</u> | <u>Add Method (CustomHeaders collection)</u>

LoadBodyFromFile (Dundas Mailer Control 1.0)

Overview | Properties | Methods

Call this method to load the message body from any text file. **LoadBodyFromFile** can be used for both text-based and html-based emails.

Syntax

MailerObject.LoadBodyFromFile(*FileName* as string, [*HtmlText* as boolean = False])

The **LoadBodyFromFile** method syntax has the following parts:

Part	Description
FileName	The full pathname of the file to load into the message body.
HtmlText	If TRUE then the file contents are loaded into the <u>HtmlBody</u> property, otherwise the file contents are loaded into the <u>Body</u> property (text-based). Defaults to FALSE.

Remarks

Use this method to load the message body from file. The file can be any valid ASCII file (e.g. .htm and .txt files).

An exception is thrown if an error occurs. Trap for the success/failure of the operation by examining VBScript's Err object immediately after calling this method (the Number property of the Err object will be a non-zero value if an error occurred). MAKE SURE that you have enabled inline error trapping by using an **On Error Resume Next** statement at the beginning of the ASP page.

If you use this method to set the **HtmlBody** property then it is up to you

to make sure that the ASCII file contains all necessary html tags (e.g. html, <body>, etc.).

See Also: Body | HtmlBody

PostArticle (Dundas Mailer Control 1.0)

Overview | Properties | Methods

Call this method to post an article to a newsgroup. You can post to either the root of the article listing or beneath an existing article.

Syntax

MailerObject.PostArticle(Server as string, [Group as string = "", UserName as string = "", Password as string = "", Port as long = 119])

The **PostArticle** method syntax has the following parts:

Part	Description
Server	The name of the server to post the article to (e.g. msnews.microsoft.com), or its corresponding IP address.
Group	The name of the news group to post the article to. Defaults to a zero-length string.
UserName	Username of the news group account (usually not required by newsgroups). Defaults to a zero-length string.
Password	Password of the news group account (usually not required by newsgroups). Defaults to a zero-length string.
Port	The port to be used when posting the article. Defaults to port 119.

Remarks

An exception is thrown if an error occurs. Trap for the success/failure of the operation by examining VBScript's Err object immediately after calling

this method (the Number property of the Err object will be a non-zero value if an error occurred). MAKE SURE that you have enabled inline error trapping by using an On Error Resume Next statement at the beginning of the ASP page.

The material to be posted is determined by the <u>Body</u> property (you can alternatively use the <u>HtmlBody</u> property but not all news servers support html-based articles). **PostArticle** uses all properties of the Mailer control except for the ones listed below:

- Priority
- ReturnReceipt
- ConfirmRead
- TOs collection
- CCs collection
- BCCs collection
- DNSServers collection, and
- SMTPRelayServers collection

You MUST specify the name of the server the article is to be posted to. You can then set the *Group* argument to the name of the news group to post to or alternatively you can leave this as a zero-length string and then add the "Group" custom header to the <u>CustomHeaders</u> collection. Set the value of this custom header to the name of the news group to post to.

If there are no items in the <u>ReplyTOs</u> collection then the <u>FromAddress</u> property determines the address to be used for replies. You will also HAVE TO set the **Body** property before sending the article. If you try to post an "empty" article a trappable error will occur.

You can post to more than one group by specifying multiple news groups for the *Group* argument and then separating the multiple entries with commas. If you are using the **CustomHeaders** collection to specify the

news groups then just add the names of the desired news groups to this collection.

You also have the option of posting either to the root of the news article listing (the default) or you can post your article so that it appears below an existing article. To have your article appear as a node of an existing article you have to know the MessageID of the desired "root" article. If you know the MessageID you can add a custom header named "References" to the CustomHeaders collection. Set this custom header to the MessageID of the desired "root" article. When the PostArticle method executes it looks in the CustomHeaders collection for the "References:" header and if this header exists the article is posted beneath the article specified by the MessageID.

The *UserName* and *Password* arguments are only required if the newsgroup you belong to requires authentication (which is usually not the case).

To use an optional parameter without specifying any values for previous optional parameters just enter consecutive commas for the missing arguments. For example, "PostArticle("someserver.com",,,,323)" is a valid function call.

For example source code illustrating how to post a news atricle see <u>Tutorial 3: Posting to a Newsgroup</u>.

See Also: <u>CustomHeaders Collection</u> | <u>Add Method (CustomHeaders collection)</u>

QuickSend (Dundas Mailer Control 1.0)

Overview | Properties | Methods

Use this method to send an email. Most headers are set explicitly via the method's arguments.

Syntax

[long] = MailerObject.QuickSend(From as string, To as string, Subject as string, Body as string, [CC as string, Attachments as string])

The **QuickSend** method syntax has the following parts:

Part	Description
From	The sender of the email. Usually this is the email address of the person sending the message. You can optionally precede the sender's address with the name of the sender.
То	Destination address of the email. You can optionally precede the address with the recipient's name.
Subject	Subject of the message.
Body	Body of the message.
СС	Optional. Email address(es) to have carbon copies of the message sent to.
Attachments	Optional. A comma-delimited list of attachments to be sent with the email.

Remarks

An exception is thrown if an error occurs. Trap for the success/failure of the operation by examining VBScript's Err object immediately after calling this method (the Number property of the Err object will be a non-zero value if an error occurred). MAKE SURE that you have enabled inline error trapping by using an On Error Resume Next statement at the beginning of the ASP page. If an SMTP relay server is being used to send the message then the indicated success/failure only pertains to getting the message to the relay server. The mail operation is then in the hands of the relay server.

You can use this method to send an email without having to set any properties or collections.

You must provide values for the *From*, *To*, *Subject* and *Body* arguments when calling this method.

The following is a listing of optional properties/collections which are utilized by **QuickSend**:

- Priority property.
- All time out properties (<u>TimeOutConnect</u>, <u>TimeOutReceive</u>, <u>TimeOutSend</u>).
- ReturnReceipt property.
- ConfirmRead property.
- <u>SMTPRelayServers</u> collection.
- **DNSServers** collection.

Note that **QuickSend** does not support html email. To send a message with an html body call the <u>SendMail</u> method.

To specify multiple values for the *To* or *Cc* arguments separate the multiple entries with commas.

For example source code on how to use this method see QuickStart.

See Also: SendMail

SendMail (Dundas Mailer Control 1.0)

Overview | Properties | Methods

Use this method to send an email.

Syntax

MailerObject.SendMail()

Remarks

An exception is thrown if an error occurs. Trap for the success/failure of the operation by examining VBScript's Err object immediately after calling this method (the Number property of the Err object will be a non-zero value if an error occurred). MAKE SURE that you have enabled inline error trapping by using an On Error Resume Next statement at the beginning of the ASP page. If an SMTP relay server is being used to send the message then the indicated success/failure only pertains to getting the message to the relay server. The mail operation is then in the hands of the relay server.

To send an email with this method you **MUST** specify the message recipient(s) by using either the <u>TOs</u>, <u>CCs</u> or <u>BCCs</u> collections. To set the body of the email use the <u>Body</u> property and/or the <u>HtmlBody</u> property. To specify the Subject use the <u>Subject</u> property.

SendMail uses ALL properties and collections of the Dundas Mailer control.

The advantage of using **SendMail** as opposed to <u>QuickSend</u> is that **SendMail** incorporates all features of the Dundas Mailer control.

Unlike the **QuickSend** method **SendMail** is capable of sending a message with an html body.

For example source code illustrating the use of SendMail see one of the

following tutorials:

- QuickStart
- Tutorial 1: Sending an Email with an Attachment
- Tutorial 2: Sending an Html Email with an Embedded Image

See Also: QuickSend

ValidateAddress (Dundas Mailer Control 1.0)

Overview | Properties | Methods

Call this method to validate an email address.

Syntax

[long] = MailerObject.ValidateAddress(Address as string)

The **ValidateAddress** method syntax has the following parts:

Part	Description
Address	A string expression specifying the address to be validated.
long	The return value of the method call signifying the validity of the supplied address.

Return Values

This function will return zero (0) if the supplied email address is valid. A non-zero return value indicates an invalid address. The following is a listing of return values and their meanings:

- One (1): The specified address was empty.
- Two (2): The specified address was too large.
- Three (3): The SMTP send operation failed.
- Four (4): The specified user was not found.
- Five (5): The MX record was not found.
- Six (6): The address validation operation failed.

NOTE: unlike the other Mailer control methods ValidateAddress WILL

NOT raise an exception if an error occurs.

Remarks

IMPORTANT: If you do not specify a <u>DNS server</u> before calling this method then it will take a substantial amount of time to perform the validation. Make sure you use a DNS server!

ValidateAddress uses the DNSServers collections to perform mail exchange lookups, so it is **highly recommended** that you add multiple DNS servers to this collection. If no DNS servers have been added to the DNSServers collection then the following default servers will be tried in their respective order: "A.ROOT-SERVERS.NET", "B.ROOT-SERVERS.NET" and "C.ROOT-SERVERS.NET". If these servers are not available then the method call will fail and return an error code.

To validate an address this method first determines if the specified domain name exists (remember that an email address consists of username@domainname) by performing a DNS lookup of mail exchange records. If the domain name exists we then initiate sending a message to the specified username for the domain whose existence was just verified. If the server for the specified domain name then refuses to accept any data for the username ValidateAddress returns FALSE, indicating that the address does not exist. However, if the server requests data we assume that the specified username exists and the function call returns TRUE.

NOTE: In certain instances the **ValidateAddress** method may return erroneous information concerning the existence of an email address. This may occur as a result of:

- The mail server being down temporarily.
- The SMTP protocol implementation of the server.

You can increase the efficiency and speed of **ValidateAddress** by adding DNS servers to the <u>DNSServers</u> collection.

See Also: <u>DNSServers Collection</u>

Address Object (Dundas Mailer Control 1.0)

An **Address** object stores details about a particular email address.

Remarks

This object is used in the <u>TOs</u>, <u>CCs</u>, <u>BCCs</u> and <u>ReplyTOs</u> collections. To add **Address** objects to these collections use their Add methods.

The **Address** object consists of the following two (2) properties:

- Address as *string*. Any valid email address (e.g. jimmys@dundas.com).
- Name as string. An optional name which will precede the Address property (e.g. "Jimmy Smith")

If we specify an address and a name then the name will precede the address property or be the only thing displayed, depending on the type of email software being used by the client. For example, if the specified *Address* property is "jimmys@dundas.com" and the *Name* property has been set to "Jimmy Smith" then the address displayed by the client's email software will be either "Jimmy Smith" <jimmys@dundas.com> or just "Jimmy Smith".

The Name property defaults to a zero-length string.

See Also: BCCs collection | CCs collection | ReplyTos collection | TOs collection

Attachment Object (Dundas Mailer Control 1.0)

This object is used to store details about a file attachment to be sent with the email. Not used by the <u>QuickSend</u> method.

Remarks

This object comprises the <u>Attachments</u> collection. To add <u>Attachment</u> objects to the <u>Attachment</u> collection use the collection's <u>Add</u> method. To remove a single attachment call the <u>Remove</u> method of the collection. To clear all <u>Attachment</u> objects from the collection use the <u>Clear</u> method.

The **Attachment** object consists of the following six (6) properties:

- *FileName* as string. The full pathname of the file to be attached to the message.
- ContentName as string. Name to be displayed in the client email software. Defaults to the specified FileName argument.
- *ContentType* as string. Specifies the content type of the attachment. Defaults to "Application/Octet-stream".
- EncodingType as string. The type of MIME encoding used. This
 defaults to "BASE64".
- *CharSet* as string. Specifies the character set to be used for the attachment. Defaults to "US-ASCII".
- *CustomHeaders* as string. Used to add custom headers to the file attachment. Defaults to a zero-length string.

If you do not specify a value for the *ContentName* property then the client email software will reference the attachment by using the *FileName* parameter.

The Content-Type header field (represented by the *ContentType* property) is used to specify the nature of the data in the attachment. It uses type and subtype identifiers in the format of "type/sub-type" (e.g.

text/html, text/plain, etc.) and also provides auxiliary information that may be required for certain types. After the type and subtype names the remainder of the header field is simply a set of parameters, specified in an attribute/value notation. These parameters differ for different types, and the ordering of the parameters is not important. Among the defined arguments is a "charset" parameter which determines the character set to be used for the attachment. Comments are allowed in accordance with RFC 822, and the rules for structured header fields may be found in RFC 2046.

The *CharSet* property determines the character set to be used for the attachment.

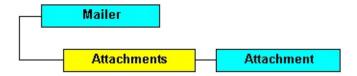
The two supported encoding types (as specified by the *EncodingType* argument) are "BASE64" and "QUOTEDPRINTABLE". If you want to explicitly set the *EncodingType* parameter then keep in mind that it must be uppercase, and the value of this argument is used for the value of the "Content-Transfer-Encoding" header. If an invalid type is specified then "BASE64" is used. For more information about the "Content-Transfer-Encoding" header refer to RFC 2045.

Note that the <u>QuickSend</u> method DOES NOT utilize the **Attachments** collection.

If you are using the Dundas <u>Upload</u> control you can loop through all items in the the Upload control's <u>Files</u> collection and add each <u>UploadedFile</u> object to the **Attachments** collection. The uploaded files will then be sent along with the email as attachments. Note that the *ContentType* of the **UploadedFile** will be used for the *ContentType* of the <u>Attachment</u> object, unless you set this argument explicitly. See <u>Tutorial 1: Sending an Email with an Attachment</u> for sample code which demonstrates using the Upload control in conjunction with the Mailer control.

See Also: Attachments Collection | Add Method | Remove Method | Item Method | Count Property

Attachments Collection (Dundas Mailer Control 1.0)



The **Attachments** collection is 0-based and stores <u>Attachment</u> objects. This collection determines how many attachments will be sent with the email if you use the <u>SendMail</u> method.

Remarks

To specify the attachments to be sent with the email add **Attachment** objects to this collection (unless you are using the <u>QuickSend</u> method to send the mail).

The **Attachments** collection supports the following methods and properties:

- Add(FileName as string, [ContentName as string = FileName, ContentType as string = "Application/Octet-stream", EncodingType as string = "BASE64", CharSet as string = "US-ASCII", CustomHeaders as string = ""]). Adds an attachment to the collection.
- Remove(Index). Removes an attachment from the collection.
- <u>Count</u> (read-only property). The number of **Attachment** objects stored in the collection.
- <u>Item(Index)</u>. Used to retrieve an <u>Attachment</u> object from the collection.
- <u>Clear()</u>. Removes all objects from the collection.

The *Index* argument can be either a zero-based numerical index or a string key. The string key is defined by the *FileName* property of the **Attachment** object.

If you are using the Dundas <u>Upload</u> control you can loop through all items in the the Upload control's <u>Files</u> collection and add each <u>UploadedFile</u> object to the **Attachments** collection. The uploaded files will then be sent along with the email as attachments. Note that the *ContentType* of the **UploadedFile** will be used for the *ContentType* of the <u>Attachment</u> object, unless you set this argument explicitly. See <u>Tutorial 1: Sending an Email with an Attachment</u> for sample code which demonstrates using the Upload control in conjunction with the Mailer control.

The **QuickSend** function **does not use** this collection.

See Also: Attachment Object | Add Method | Remove Method | Clear Method | Item Method | Count Property

CustomHeader Object (Dundas Mailer Control 1.0)

This object is used with the **CustomHeaders** collection.

Remarks

To add **CustomHeader** objects to the <u>CustomHeaders</u> collection use the collection's <u>Add</u> method. To remove one item call the <u>Remove</u> method of the collection. To clear all objects from the collection use the <u>Clear</u> method.

The **CustomHeader** object consists of the following two (2) properties:

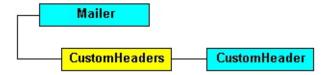
- Name as string. The name of the custom header to be added to the message.
- Value as string. The value of the specified custom header.

If the header is specific for your application then by convention the header name should start with the characters "X-".

The **QuickSend** method does not utilize the **CustomHeaders** collection.

See Also: <u>CustomHeaders Collection</u> | <u>Add Method</u> | <u>Remove Method</u> | <u>Item Method</u> | <u>Count Property</u>

CustomHeaders Collection (Dundas Mailer Control 1.0)



Stores <u>CustomHeader</u> objects. This collection is 0-based and determines the custom headers to be added to the email message (unless you are using the <u>QuickSend</u> method).

Remarks

This collection specifies the custom headers to be added to the email.

The **CustomHeaders** collection supports the following methods and properties:

- Add(Name as string, Value as string). Adds a **CustomHeader** object to the collection.
- Remove(Index). Removes a particular object from the collection.
- <u>Count</u> (read-only property). The number of **CustomHeader** objects stored in the collection.
- <u>Item(Index)</u>. Used to retrieve a particular object from the collection.
- Clear(). Removes all CustomHeader objects from the collection.

If the header is specific for your application then by convention the header name should start with the characters "X-".

Custom headers are very helpful if you want to automate your Internet messaging application, since headers are at the top of messages and require less reading as compared to the whole message.

The QuickSend function does not use this collection.

See Also: CustomHeader Object | Add Method | Remove Method | Clear

Method | Item Method | Count Property

DNSServer Object (Dundas Mailer Control 1.0)

This object is used with the **DNSServers** collection.

Remarks

To add **DNSServer** objects to the <u>DNSServers</u> collection use the collection's <u>Add</u> method. To remove one item call the <u>Remove</u> method of the collection. To clear all objects from the collection use the <u>Clear</u> method.

The **DNSServer** object consists of the following property:

- Name as string. The name of the DNS server to be used, or its corresponding IP address.
- *TCPIPRetryCount* as long. The maximum number of times to attempt to connect to the server using the TCP/IP protocol. Defaults to 1.
- *UDPRetryCount* as long. The maximum number of times to attempt to connect to the server using the UDP protocol. Defaults to 3.

If the <u>SMTPRelayServers</u> collection is empty then the email message is sent directly to the destination server, so a DNS server is required to determine where the message is to be sent. However, if the <u>SMTPRelayServers</u> collection is not empty then the message is sent using the specified relay server, and the <u>DNSServers</u> collection is ignored since the relay server is responsible for determining where the message is being sent to. If both the <u>SMTPRelayServers</u> and <u>DNSServers</u> collections are empty the DNS server to be used defaults to "A.ROOT-SERVERS.NET". If this fails "B.ROOT-SERVERS.NET" will be tried. If this too fails then "C.ROOT-SERVERS.NET" will be tried. If all of these fail then the mail operation will fail as well.

There are two (2) methods of connecting to a DNS server: using the TCP/IP protocol and using the UDP protocol. Different DNS servers may

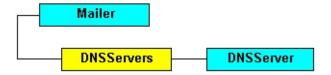
support either protocol or both protocols. By default the TCP/IP method is tried first. If this fails we then attempt to contact the server using UDP. If you know what protocols your specified DNS servers use you can optimize the DNS lookup operations by specifying which protocol to use when working with the DNS server(s).

Specifying multiple DNS servers (when not using SMTP relays) increases the reliability and performance of your mail operation.

This collection can be used by all mail sending functions (as well as PostArticle), depending on whether SMTP relay servers are being used or not.

See Also: <u>CustomHeaders Collection</u> | <u>Add Method</u> | <u>Remove Method</u> | <u>Item Method</u> | <u>Count Property</u>

DNSServers Collection (Dundas Mailer Control 1.0)



Stores <u>DNSServer</u> objects. This collection is 0-based and determines the DNS server(s) which will perform the DNS lookups necessary to send the email message if SMTP relay servers are not being used.

Remarks

This collection specifies the DNS server(s) to be used if the email is being sent directly to the destination server.

The **DNSServers** collection supports the following methods and properties:

- Add(Name as string, TCPIP as long, UDP as long). Adds a
 DNSServer object to the collection.
- Remove(Index). Removes a particular object from the collection.
- <u>Count</u> (read-only property). The number of **DNSServer** objects stored in the collection.
- <u>Item(Index)</u>. Used to retrieve a particular object from the collection.
- <u>Clear()</u>. Removes all objects from the collection.

The *Index* argument can be either a zero-based numerical index or a string key. The string key is defined by the *Name* argument.

If the <u>SMTPRelayServers</u> collection is empty then the email message is sent directly to the destination server, so a DNS server is required to determine where the message is to be sent. However, if the **SMTPRelayServers** collection is not empty then the message is sent using the specified relay server, and the **DNSServers** collection is

ignored since the relay server is responsible for determining where the message is being sent to. If both the **SMTPRelayServers** and **DNSServers** collections are empty the DNS server to be used defaults to "A.ROOT-SERVERS.NET". If this fails "B.ROOT-SERVERS.NET" will be tried. If this too fails then "C.ROOT-SERVERS.NET" will be tried. If all of these fail then the mail operation will fail as well.

Note that the first server in the collection will be tried first. If it is not available the next <u>DNSServer</u> object will be tried, and so on and so forth.

There are two (2) methods of connecting to a DNS server: using the TCP/IP protocol and using the UDP protocol. Different DNS servers may support either protocol or both protocols. By default the TCP/IP method is tried first. If this fails we then attempt to contact the server using UDP. However, if you know what protocols your specified DNS servers use you can optimize the DNS lookup operations by specifying which protocol to use when working with the DNS server(s).

This collection may be used by all send mail functions (as well as PostArticle), depending on whether SMTP relay servers are being used or not.

See Also: DNSServer Object | Add Method | Remove Method | Clear Method | Item Method | Count Property | SMTPRelayServers Collection

HtmlEmbeddedObj Object (Dundas Mailer Control 1.0)

This object is stored in the **HtmlEmbeddedObjs** collection.

Remarks

To add **HtmlEmbeddedObj** objects to the <u>HtmlEmbeddedObjs</u> collection use the collection's <u>Add</u> method. To remove a single object call the <u>Remove</u> method of the collection. To clear all objects from the collection use the <u>Clear</u> method.

The **HtmlEmbeddedObj** object consists of the following four (4) properties:

- *FileName* as string. The name of the file to be embedded.
- ContentID as string. A unique ID which is used to identify each embedded object.
- ContentName as string. Optional. The new name of the file attachment. If you do not use this argument then the FileName argument is used instead.
- *CustomHeaders* as string.

Add **HtmlEmbeddedObj** objects to the <u>HtmlEmbeddedObjs</u> collection to embed audio files, graphics files etc. to the html body of the email message.

To embed objects into an html email you will need to "wrap" the appropriate tags around the **HtmlEmbeddedObj** objects being embedded into the <u>HtmlBody</u> property. Audio elements can be inserted as the BGSOUND attribute of the BODY tag, while graphics files are commonly inserted using tags. Refer to <u>Tutorial 2: Sending an Html Email with an Embedded Image</u> for example source code of this.

When you add **HtmlEmbeddedObj** objects to the **HtmlEmbeddedObjs** collection MAKE SURE that the values you supply for the *ContentID*

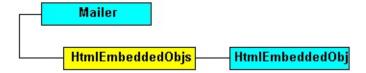
argument are unique. This *ContentID* is referenced within html tags with "cid:theuniquenumber". It is also important to use lower-case when specifying "cid". Microsoft's Outlook will not embed objects if you use "CID".

If the client email software supports html email it might not support embedded html objects. In this case the *ContentName* argument will be displayed in the email body instead of the embedded object's content.

NOTE: The **QuickSend** method can not send an html-based email.

See Also: <u>HtmlEmbeddedObjs Collection</u> | <u>Add Method</u> | <u>Remove Method</u> | <u>Item Method</u> | <u>Count Property</u> | <u>Tutorial 2: Sending an Html Email with an Embedded Image</u>

HtmlEmbeddedObjs Collection (Dundas Mailer Control 1.0)



Stores <u>HtmlEmbeddedObj</u> objects. This collection is 0-based and determines the objects which can be embedded into the html body of the email.

Remarks

This collection stores any objects to be embedded into the html body of the email. Objects which can be embedded into the html body of the message are audio files, graphics files, etc.

The **HtmlEmbeddedObjs** collection supports the following methods and properties:

- Add (FileName as string, ContentID as string, [ContentName as string, CustomHeaders as string]). Adds an HtmlEmbeddedObj object to the collection.
- Remove(Index). Removes a particular object from the collection.
- <u>Count</u> (read-only property). The number of objects stored in the collection.
- <u>Item(Index)</u>. Used to retrieve a particular object from the collection.
- <u>Clear()</u>. Removes all objects from the collection.

The *Index* argument can be either a zero-based numerical index or a string key. The string key is defined by the *FileName* argument.

To embed objects into an html email you will need to "wrap" the appropriate tags around the **HtmlEmbeddedObj** object when adding the objects to the **HtmlBody** property. Audio elements can be inserted as the

BGSound attribute of the Body tag, while graphics files are commonly inserted using tags. Refer to <u>Tutorial 2: Sending an Html Email</u> with an <u>Embedded Image</u> for example source code of this.

To embed objects into an email you will need to iterate through the <u>Files</u> collection of the Dundas <u>Upload</u> control.

When you add objects to this collection MAKE SURE that the values you supply for the *ContentID* argument are unique. This *ContentID* is referenced within html tags with "cid:theuniquenumber". It is also important to use lower-case when specifying "cid". Microsoft's Outlook will not embed objects if you use "CID".

The **QuickSend** method does not utilize this collection.

Note that if the <u>HtmlBody</u> property is a zero-length string (the default) then the items in this collection will be ignored. Items will also be ignored if both the <u>HtmlBody</u> and <u>Body</u> properties are set and the client's email software displays text-based email by default.

Refer to <u>Tutorial 2</u>: <u>Sending an Html Email with an Embedded Image</u> for example source code on how to send an html email with embedded objects.

See Also: <u>HtmlEmbeddedObj Object</u> | <u>Add Method</u> | <u>Remove Method</u> | <u>Clear Method</u> | <u>Item Method</u> | <u>Count Property</u>

SMTPRelayServer Object (Dundas Mailer Control 1.0)

This object is used by the **SMTPRelayServers** collection.

Remarks

To add **SMTPRelayServer** objects to the **SMTPRelayServers** collection use the collection's <u>Add</u> method. To remove one item call the <u>Remove</u> method of the collection. To clear all objects from the collection use the <u>Clear</u> method.

The **SMTPRelayServer** object consists of the following properties:

- Name as string. The name of the SMTP relay server.
- *Port* as long. The port number to be used with the server. Defaults to 25.
- Localhost as string. The local computer's name or it's IP address.
 Used with the HELO command, this defaults to "local.com".
- *User* as string. A valid username (if the SMTP server requires authentication). Defaults to a zero-length string.
- Password as string. A valid password (if the SMTP server requires authentication). Defaults to a zero-length string.

SMTP relay servers are used to relay a message to a destination server.

If the <u>SMTPRelayServers</u> collection is empty then the email message is sent directly to the destination server, so a DNS server is required to determine where the message is to be sent. However, if the <u>SMTPRelayServers</u> collection is not empty then the message is sent using the specified relay server, and the <u>DNSServers</u> collection is ignored since the relay server is responsible for determining where the message is being sent to. If both the <u>SMTPRelayServers</u> and <u>DNSServers</u> collections are empty the DNS server to be used defaults to "A.ROOT-SERVERS.NET". If this fails "B.ROOT-SERVERS.NET" will be

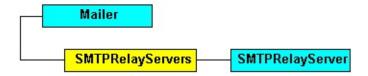
tried. If this too fails then "C.ROOT-SERVERS.NET" will be tried. If all of these fail then the mail operation will fail as well and return an error code.

Specifying multiple SMTP relay servers increases the reliablity and performance of your mail operation.

All mail send operations (including <u>PostArticle</u>) utilize the **SMTPRelayServer** collection.

See Also: <u>SMTPRelayServers Collection</u> | <u>Add Method</u> | <u>Remove Method</u> | <u>Item Method</u> | <u>Count Property</u>

SMTPRelayServers Collection (Dundas Mailer Control 1.0)



This collection stores <u>SMTPRelayServer</u> objects, and determines the SMTP relay servers to be used when sending the email (if any). Note that it is 0-based.

Remarks

The **SMTPRelayServers** collection supports the following methods and properties:

- Add(Name as string, [Port as long = 25, LocalHost as string = "local.com", User as string = "", Password as string = ""). Adds an SMTPRelayServer object to the collection.
- Remove(Index). Removes a particular object from the collection.
- <u>Count</u> (read-only property). The number of objects stored in the collection.
- <u>Item(Index)</u>. Used to retrieve a particular object from the collection.
- <u>Clear()</u>. Removes all objects from the collection.

If the **SMTPRelayServers** collection is empty then the email message is sent directly to the destination server, so a DNS server is required to determine where the message is to be sent. However, if the **SMTPRelayServers** collection is not empty then the message is sent using the specified relay server, and the <u>DNSServers</u> collection is ignored since the relay server is responsible for determining where the message is being sent to. If both of these collections are empty then the DNS server to be used defaults to "A.ROOT-SERVERS.NET". If this fails "B.ROOT-SERVERS.NET" will be tried. If this too fails then "C.ROOT-SERVERS.NET" will be tried. If all of these fail then the mail operation

will fail as well.

It is highly recommended that you specify at least one relay server. Using a relay increases the speed of the operation, and a relay will also attempt to send a message multiple times if it can not be sent with the first attempt. A Direct Send mail operation (performed when no SMTP servers have been specified) will only attempt to send the email one (1) time.

All mail send operations (including PostArticle) utilize this collection.

See Also: SMTPRelayServer Object | Add Method | Clear Method | Count Property | Item Method | Remove Method | DNSServers Collection

TOs, CCs, BCCs and ReplyTOs collections (Dundas Mailer Control 1.0)



These collections store <u>Address</u> objects, and determine the recipient(s) of the email. They are all 0-based.

Remarks

These collections support the following methods and properties:

- Add(Address as string, [Name as string]). Adds an Address object to the collection.
- Remove(Index). Removes a particular object from the collection.
- <u>Count</u> (read-only property). The number of **Address** objects stored in the collection.
- <u>Item(Index)</u>. Used to retrieve a particular object from the collection.
- Clear(). Removes all Address objects from the collection.

The <u>QuickSend</u> method **does not use** these collections. All other mail send methods (as well as <u>PostArticle</u>) use these collections.

You can send an email to multiple recipients by adding more than one **Address** object to these collections. You can also specify multiple addresses with just one (1) **Address** object by separating the multiple entries with commas. However, if you use just one (1) **Address** object to specify multiple addresses then you CAN NOT precede the address with the name of the message sender (e.g. "Firstname Lastname" fromme@someserver.com).

The **ReplyTOs** collection is used if the recipient of the email decides to

reply to the email message. If there are no items in this collection then the ReplyTo address is taken from the <u>FromAddress</u> property. Note that all **Address** objects will be used for the reply message.

To have multiple addresses appear in the To field of the resulting email just add multiple **Address** objects to the **ReplyTOs** collection.

See Also: Address Object | Add Method | Remove Method | Clear Method | Item Method | Count Property

Count Property (All Dundas Mailer collections)

Use this read-only property to determine the number of elements currently stored in any Dundas Mailer collection.

Syntax

AnyMailerCollection.Count

Remarks

Use this property to find out how many elements are currently being stored in any Dundas Mailer collection. This is useful for determining the upper loop delimiter when iterating through all of a collection's items. Note that you can also iterate through any Dundas Mailer collection with **For...Each** loops.

See Also: Remove Method | Item Method | Clear Method

Add Method (Attachments collection)

Adds an Attachment object to the Attachments collection.

Syntax

AttachmentsCollection.Add (*FileName* as string, [*ContentName* as string = *FileName*, *ContentType* as string = "Application/Octet-stream", *EncodingType* as string = "BASE64", *CharSet* as string = "US-ASCII", *CustomHeaders* as string = ""])

The **Add** method syntax has the following parts:

Part	Description
FileName	The full pathname of the file to be added.
ContentName	The name by which the attachment will be referred to in the client's email software.
ContentType	Specifies the media type and subtype of data in the body of the message.
EncodingType	The type of encoding to be used for the attachment. Must be uppercase.
CharSet	The character set to be used.
CustomHeaders	Any custom header to be added to the attachment.

Remarks

Use this method to add an **Attachment** object to the <u>Attachments</u> collection. To remove an **Attachment** use the <u>Remove</u> method. To remove all **Attachments** from the collection call the <u>Clear</u> method.

To send multiple attachments just add the desired file attachments to this collection and call the <u>SendMail</u> method.

If you do not specify a value for the *ContentName* property then the client email software will reference the attachment by using the *FileName* parameter.

To add a custom header to the attachment use the *CustomHeaders* argument.

The Content-Type header field (represented by the *ContentType* argument) is used to specify the nature of the data in the attachment. It uses type and subtype identifiers in the format of "type/sub-type" (e.g. text/html, text/plain, etc.) and also provides auxiliary information that may be required for certain types. After the type and subtype names the remainder of the header field is simply a set of parameters, specified in an attribute/value notation. These parameters differ for different types, and the ordering of the parameters is not important. Among the defined arguments is a "charset" parameter which determines the character set to be used for the attachment. Comments are allowed in accordance with RFC 822, and the rules for structured header fields may be found in RFC 2046.

The *CharSet* argument determines the character set to be used for the attachment. This parameter is not case sensitive.

The two supported encoding types (as specified by the *EncodingType* argument) are "BASE64" and "QUOTEDPRINTABLE". If you want to explicitly set the *EncodingType* parameter then keep in mind that it must be uppercase, and the value of this argument is used for the value of the "Content-Transfer-Encoding" header. If an invalid type is specified then "BASE64" is used. You may want to use "QUOTEDPRINTABLE" if the attachment content can be represented with ASCII characters. For more information about the "Content-Transfer-Encoding" header refer to RFC 2045.

If you are using the Dundas <u>Upload</u> control you can loop through all items in the Upload control's <u>Files</u> collection and add each <u>UploadedFile</u>

object to the Mailer control's **Attachments** collection. The uploaded files will then be sent along with the email as attachments. The *ContentType* of the **UploadedFile** will be used for the *ContentType* of the <u>Attachment</u> object unless you set this argument explicitly. See <u>Tutorial 1: Sending an Email with an Attachment</u> for sample code which demonstrates using the Upload control in conjunction with the Mailer control. You can also refer to the Upload control's <u>tutorials</u> for further code samples and instructions.

See Also: Attachments Collection | Clear Method | Count Property | Item Method | Remove Method

Add Method (CustomHeaders collection)

Adds a **CustomHeader** object to the **CustomHeaders** collection.

Syntax

CustomHeadersCollection.Add (Name as string, Value as string)

The **Add** method syntax has the following parts:

Part	Description
Name	The name of the custom header to be added.
Value	The value of the custom header to be added.

Remarks

Use this method to add a **CustomHeader** object to the <u>CustomHeaders</u> collection. To remove a <u>CustomHeader</u> object use the <u>Remove</u> method. To remove all **CustomHeader** objects from the collection call the <u>Clear</u> method.

Please note that the **QuickSend** method does not utilize this collection.

See Also: CustomHeaders Collection | Clear Method | Count Property | Item Method | Remove Method

Add Method (DNSServers collection)

Adds a **DNSServer** object to the **DNSServers** collection.

Syntax

DNSServersCollection.Add (*Name* as string, [*TCPIP* as long = 1, *UDP* as long = 3])

The **Add** method syntax has the following parts:

Part	Description
Name	The name of the DNS server, or its corresponding IP address.
TCPIP	The maximum number of connection attempts using the TCP/IP protocol. Defaults to one (1).
UDP	The maximum number of connection attempts using the UDP protocol. Defaults to three (3).

Remarks

Use this method to add a **DNSServer** object to the <u>DNSServers</u> collection. To remove a **DNSServer** object use the <u>Remove</u> method. To remove all objects from the collection call the <u>Clear</u> method.

To make sure that a DNS server is available to process emails it is recommended that you specify more than one server.

Note that the first server in the collection will be tried first. If it is not available the next <u>DNSServer</u> object will contacted, and so on.

There are two (2) methods of connecting to a DNS server: using the TCP/IP protocol and using the UDP protocol. Different DNS servers may support either protocol or both protocols. By default the TCP/IP method is

tried first. If this fails we then attempt to contact the server using UDP. However, if you know what protocols your specified DNS servers use you can optimize the DNS lookup operations by specifying which protocol to use when working with the DNS servers (i.e. set the protocol argument which isn't used by your DNS server to zero).

See Also: <u>DNSServers Collection</u> | <u>Clear Method</u> | <u>Count Property</u> | <u>Item Method</u> | <u>Remove Method</u>

Add Method (HtmlEmbeddedObjs collection)

Adds an <u>HtmlEmbeddedObj</u> object to the **HtmlEmbeddedObjs** collection.

Syntax

HtmlEmbeddedObjsCollection.Add (*FileName* as string, *ContentID* as string, [*ContentName* as string, *CustomHeaders* as string])

The **Add** method syntax has the following parts:

Part	Description
FileName	The filename of the object to be embedded.
ContentID	A unique ID which is used to identify each embedded object.
ContentName	The name which will be displayed in the email. If this is not set then the <i>FileName</i> will be displayed.
CustomHeaders	Any custom header.

Remarks

Use this method to add an **HtmlEmbeddedObj** object to the <u>HtmlEmbeddedObjs</u> collection. To remove an object use the Remove method. To remove all objects from the collection call the Clear method.

Objects which can be embedded are graphics files, audio files, etc.

MAKE SURE that the values you supply for the *ContentID* argument are unique. This *ContentID* is referenced within html tags with "cid:uniquenumber" (see the sample source code below for an example

of this). It is also important to use lower-case when specifying "cid". Microsoft's Outlook will not embed objects if you use "CID".

Use the Dundas Upload control to allow the user to specify objects to be embedded in the email. The Upload control exposes a Files collection which consists of **UploadedFile** objects which can then be used as the embedded objects. To embed the uploaded files loop through the **Files** collection and add them to the HtmlEmbeddedObjs collection, and then embed the object by wrapping the appropriate html tags around the object in the Mailer control's HtmlBody property. You can use the **UploadedFile** object's **TagName** property to determine which file input box the object originated from, and you can also use VBScripts's InStr method in conjunction with the **UploadedFile** objects **ContentType** property to determine the type of the object. This lets you make sure that an object is not embedded into the html body of the email with the wrong tags around it (see the code sample below). You can also refer to Tutorial 2: Sending an Html Email with an Embedded Image for sample source code which demonstrates using the Upload control in conjunction with the Mailer control to embed objects into an email.

Please note that it is up to you to wrap the appropriate tags around the embedded objects. This lets you control where in the email the objects will appear. The cid (content ID) is used within the html tags to identify which object in the **HtmlEmbeddedObjs** collection is to be embedded (see the example source code below).

See Also: <u>HtmlEmbeddedObjs Collection</u> | <u>Clear Method</u> | <u>Count Property</u> | <u>Item Method</u> | <u>Remove Method</u>

Example

'This small code snippet assumes that there is an input element of the F 'txtBGSound which lets the user upload a file to the server (to be used 'sound of the html-based email). We check to see if a file was uploaded 'particular file input box and then we check the ContentType property 'file to make sure that the user uploaded a valid audio file.

' NOTE: it is assumed here that objUpload. Save has already been called

```
'initialize the HtmlBody property
objEmail.HtmlBody = "<html><body>"
'loop through all files uploaded by user
for i = 0 to objUpload.Files.Count - 1
    'check to see if a file was uploaded using a file input box named "t
    if (objUpload.Files(i).TagName = "txtBGSound") then
        'now make sure that the user actually uploaded a valid audio fi
        if (InStr(1,objUpload.Files(i).ContentType,"audio")) then
            'now add the object to the HtmlEmbeddedObjs collection,
            ' the object into the HtmlBody property, wrapping the app
            ' around it. Note that we are setting the ContentName argu
            'HtmlEmbeddedObj object to the OriginalPath property o
            ' object (e.g. c:\MyPic.jpg). Then the resulting name of the
            'email will not be preceded with a guid (all uploaded files
            ' to disk with a guid as the first part of their filename).
            objEmail.HTMLEmbeddedObjs.Add objUpload.Files(i).I
            objEmail.HTMLBody = objEmail.HTMLBody & "<BGS
        end if
    end if
next
'now finish setting the HtmlBody property
objEmail.HtmlBody = objEmail.HtmlBody & "</body></html>"
```

Add Method (SMTPRelayServers collection)

Adds an **SMTPRelayServer** object to the **SMTPRelayServers** collection.

Syntax

SMTPRelayServersCollection.Add (*Name* as string, [*Port* as long = 25, *LocalHost* as string = "local.com", *User* as string = "", *Password* as string = ""])

The **Add** method syntax has the following parts:

Part	Description
Name	The name of the SMTP server, or its corresponding IP address.
Port	The port to use when connecting to the server. Defaults to 25.
LocalHost	The local computer's name or its IP address. Used with the HELO command, this defaults to "local.com".
User	A valid username if the NNTP server requires authentication. ONLY applies to NNTP servers!
	The password corresponding to the supplied username (if the NNTP server requires authentication). ONLY applies to NNTP servers!

Remarks

Use this method to add an **SMTPRelayServer** object to the <u>SMTPRelayServers</u> collection. To remove an object use the <u>Remove</u> method. To remove all objects from the collection call the <u>Clear</u> method.

If the specified server requires authentication then you must supply a valid username and password. However, most SMTP servers do not require authentication.

If you specify more than one (1) server then the first server in this collection will be used to send the email. However, if this server is down then the next server in the collection will be used to send the email, and so on. Specifying more than one server increases the chances of successfully sending your email, and using a relay server is also faster than sending the message directly.

You can set an optional value without having to provide values for previous optional arguments by separating the arguments with commas. For example, to provide a username and password you could call the Add method with the following syntax: SMTPRelayServersCollection.Add "someserver.com",,,"MyUsername","MyPassword".

NOTE: The username and password parameters are only implemented for NNTP servers. If you need to implement a username/password for an SMTP server you will have to purchase the source code to the Mailer control.

See Also: <u>SMTPRelayServers Collection</u> | <u>Clear Method</u> | <u>Count Property</u> | <u>Item Method</u> | <u>Remove Method</u>

Add Method (TOs, CCs, BCCs and ReplyTOs collections)

Adds an Address object to the relevant collection.

Syntax

Collection.Add (Address as string, [Name as string])

The **Add** method syntax has the following parts:

Part	Description
Address	An email address to send the message to.
Name	Optional. The "name" of the email recipient.

Remarks

Use this method to add an **Address** object to the specified collection. The email will be sent to every <u>Address</u> object in the collection. To remove an **Address** use the <u>Remove</u> method. To remove all addresses from the collection call the <u>Clear</u> method.

The optional *Name* argument lets you specify a user-friendly name which will appear in the appropriate field of the email (i.e. To, CC, BCC or ReplyTo fields). For example, let's assume that the specified address for John Doe is "johnd@dundas.com". If we also set the *Name* argument to "John Doe" then the recipient of the email will see either "John Doe" <johnd@dundas.com> or "John Doe", depending on the email program the client is using.

NOTE: one of these collections needs to have an **Address** object in order for the **SendMail** operation to proceed. If none of these collections have been populated then the <u>SendMail</u> operation will immediately fail since no destination address has been specified.

See Also: TOs, CCs, BCCs and ReplyTOs collections | Clear Method | Count Property | Item Method | Remove Method

Clear (All Dundas Mailer collections)

Removes all elements from a Dundas Mailer collection.

Syntax

AnyMailerCollection.Clear

Remarks

Use this method to empty a collection of all elements. To remove just one element in a collection use the **Remove** method.

See Also: Count Property | Item Method | Remove Method

Remove Method (All Dundas Mailer collections)

Call this method to remove one (1) element from any Dundas Mailer collection.

Syntax

AnyMailerCollection.Remove (Index)

The **Remove** method syntax has the following parts:

Part	Description
	This argument can be either a number (integer or long data type) or a string key. All of the Dundas Mailer collections are zero (0) based.

Remarks

To identify the collection item to be removed you can specify either a number (integer or long data types) or a string key for the *Index* argument. If you specify a number then the index of the element will be used (the index being zero-based). For example, the statement "TOs.Remove 0" will remove the first element in the TOs collection. If a string key is specified then the element to be removed will be identified by its key. Refer to the following for a listing of the various collection keys:

- TOs Collection string key = Address property
- CCs Collection string key = Address property
- BCCs Collection string key = Address property
- Attachments Collection string key = FileName property
- HTMLEmbeddedObjs Collection string key = FileName property

- CustomHeaders Collection string key = Name property
- DNSServers Collection string key = Name property
- SMTPServers Collection string key = Name property

For example, to remove an email address named "johnd@dundas.com" from the <u>TOs</u> collection you would enter: TOs.Remove "johnd@dundas.com".

See Also: Count Property | Item Method | Clear Method

Item Method (All Dundas Mailer collections)

Call this method to retrieve an element from any Dundas Mailer collection. This is the default member of all collections.

Syntax

AnyMailerCollection.Item (*Index*)

The **Item** method syntax has the following parts:

Part	Description
	This argument can be either a number (integer or long data type) or a string key. All of the Dundas Mailer collections are zero (0) based.

Remarks

If the *Index* argument is specified as a number then the item is retrieved via the index of the collection (zero-based). For example, to retrieve the first element in the TOs collection you could use the following statement: Set objAddress = TOsCollection.Item(0). If a string is specified then the element is retrieved via its key. For a listing of collection keys refer to the following:

- TOs Collection string key = Address property
- CCs Collection string key = Address property
- BCCs Collection string key = Address property
- Attachments Collection string key = FileName property
- HTMLEmbeddedObjs Collection string key = FileName property
- CustomHeaders Collection string key = Name property

- DNSServers Collection string key = Name property
- SMTPServers Collection string key = Name property

This method is the default collection method so you do not have to explicitly declare it. For example, to retrieve the first **Address** object from the **TOs** collection you could use this statement: Set objAddress = TOs(0).

See Also: Count Property | Clear Method | Remove Method

DundasChart

Designed from the ground up for ASP developers, Dundas Chart is easy to use, flexible, and backed by Dundas Software's award winning team of software engineers.

Dundas Chart delivers presentation-quality graphics, **and supports both 2D and 3D charts**.

With blazing performance, stunning presentation quality graphics, power, speed and unmatched flexibility, Dundas Chart has all the features you need.

The 3D charting control will also automatically leverage 3D accelerated video hardware (if present) in your system or web server.

Dundas Chart 3D Features

- Stunning presentation quality
- Hardware acceleration utilizes 3D accelerated hardware from ATI,
 3Dfx, nVidia, and an array of other manufacturers
- OpenGL State-of-the-art OpenGL implementation
- Rotation you can rotate your charts programmatically.
- Selection (drilldown) determine what slice a user clicked on and implement a drilldown scenario with ease!
- Wide variety of chart types
- Texture flexibility use a bitmap image as a customtexture
- **Texture/Color blending -** Not only can you use textures, but you can also change the texture's color
- **Grayscale texturing** If colors are not in the plans, grayscaling is also supported, giving you added image flexibility
- Template Creator to pre-load texture/color/size types save time

in creating chart types by using the template creator to store/load pre-made colour schemes/textures and sizes

- **Edge Beveling -** edges can be beveled to give added customization to charts; make your edges as rounded or as square as is needed.
- Annotation add notes anywhere on the chart add single or multiple notes to the chart in any position.
- Vertical annotation text and notes can be displayed vertically for added customization.
- User-configurable element positioning position legends, titles, annotations and other elements either programmatically or by using simple drag and drop actions

Dundas Chart Editions:

ASP/Windows DNA 2000 Development

Specially designed for use in server-side projects, Dundas Chart (**2D and 3D**) offers full support for your ASP and Windows DNA development efforts with a full set of server side features.

Enterprise Edition

Dundas Chart Enterprise provides the complete C++/ATL sourcecode, documentation, project and workspace files needed to build the product. Dundas Chart Enterprise offers unlimited customization and flexibility. If you have very specialized requirements or porting needs then look to Dundas Chart Enterprise for the solution!

For more information visit our web site at **Dundas Chart**

Dundas Pie Chart Server Control 2.0

Copyright © Dundas Software Ltd. 2000. All Rights Reserved

Overview | Properties | Methods

The Dundas Pie Chart Server Control 2.0 allows you to quickly and painlessly add dynamic pie charts to your ASP applications.

Features:

- Add or remove pie chart slices (elements).
- Change the title of the pie chart.
- Rotate the chart progammatically.
- Implement selection and drilldown with ease.
- Change the pie slice labels as well as the labels for the pie slices in the Legend.
- Change the sizes of the pie slices.
- Change the minimum "collected" value (see below for more details).
- Change the textures and/or colors of pie elements.
- Explode/collapse pie elements.

To use this control in web pages you **MUST USE** either an tag (see the source code for the two supplied ASP pages) or an <A> tag. In this demo there is one ASP page (let's call it the secondary page) which creates an instance of the ASP Pie Chart control, loads a template (created with the Template Creator), adds the specified data and then calls SendJPEG to send the data to the client. The first page (lets call it the main page) sets up the user interface, retrieves pie chart variables

(like the number of slices, whether the slices are exploded, etc.) and then retrieves the pie chart image by embedding the secondary ASP page in an tag with the SRC attribute set to the secondary page. The first page can be either an html or ASP file.

Small data elements (pie slices) are "collected" and then displayed together as one pie slice called the "collected" element. You can change the minimum data value at which slices are collected with the SetCollectedLimit method. Call SetCollectedProp to manipulate the properties of the collected slice (texture, color, etc.).

To make sure that the browser never uses a cached copy of the jpeg set the **Expires** property of the **Response** object to either zero or a negative value. See a <u>tutorial</u> for an example of this.

To output a pie chart jpeg perform the following actions in their specified order:

- 1. Create an instance of an ASPPieChart object by using CreateObject (e.g. CreateObject("Dundas.PieChartServer.2")).
- 2. Set the directory properties of the control which specify working directories for different aspects of chart activity. These properties are DirTemplate and DirTexture, and deal with the directories used to store templates and textures, respectively.
- 3. Load a template via the <u>LoadTemplate</u> method.
- 4. Set the properties of the pie chart (e.g. exploded pieces, collected element properties, etc.)
- 5. Add pie chart data with the AddData method.
- 6. Send the resulting jpeg directly to a client with the <u>SendJPEG</u> method or alternatively you can save the jpeg to disk with <u>CreateJPEGFile</u> and then present the image to a user with a standard tag.
- 7. If you want to generate another Jpeg using the same template then repeat Steps 4 to 6. If you want to create another jpeg with a

different template repeat steps 1 to 6.

- 8. Destroy the Pie Chart object by setting it to Nothing (e.g. Set objPieChart = Nothing).
- 9. MAKE SURE that there is no html code in this ASP page which outputs the jpeg.

The ProgID of the control is: Dundas.PieChartServer.2

IMPORTANT: to successfully output a pie chart jpeg the Intel compression library (ijl15.dll) must exist either in your system directory or at the same directory level as the server control (AspPieChart.dll).

To minimize the amount of whitespace in the jpeg make the pie chart image as large as possible in the Template Creator.

For information concerning debugging, as well as FAQs and troubleshooting go to our <u>developer site</u>.

See Also: <u>Dundas Pie Chart Overview</u> | <u>Template Creator Overview</u>

AddData

Overview | Properties | Methods

Call this method to add data elements (slices) to a pie chart.

Syntax

long = PieChartObject.AddData(Value As Double, [DataLabel As String],
[Color As Long =&HFFFFFF], [Texture As String])

The **AddData** method syntax has the following parts:

Part	Description
Value	The size of the pie chart slice (element).
DataLabel	The label of the pie slice. If this is not set then your resulting jpeg will not have pie slice labels.
LegendLabel	The legend title of the pie slice. If this is not set then your resulting jpeg will not have labels (text) for the legend.
Color	The color of the pie slice (defaults to white). Use either a hexadecimal value or VBScript's RGB function.
Texture	The texture of the pie slice.
long	Returns a <i>long</i> indicating the success or failure of the operation. Zero (0) is returned if successful, otherwise a

non-zero value is returned.

Returns

A long which indicates the success/failure of the operation. Zero (0) is returned if successful, otherwise a non-zero value is returned. If an error occurs you can use this return value as the argument for GetErrorText in order to retrieve a relevant text error message, or alternatively you can rem out the "On Error Resume Next" statement and observe the thrown exception for a description of the error.

Remarks

The size of the pie slice is dictated by the *Data* argument. The size is not in any specific units (e.g. degrees or radians), rather it is represented as a percentage of the total number of specified sizes. For example, if there are four slices and they all have a *Data* value of 10 then their individual sizes will be: (10/(10 + 10 + 10 + 10)) * 100%.

If you do not specify a color or texture for an element then the color or texture for that particular element will be obtained from the loaded template (if it exists). If the same element does not exist in the template then the default color is white and the texture is set to "none". For example, assume we have loaded a template which contains four (4) pie slices. If we add pie slice #1 with **AddData** and we do not specify a color or texture then the color and texture for pie element #1 in the template will be used. If, however, we add pie element #5 without specifying a color or texture then there will be no texture used and the color of the pie slice will be white since the template we loaded does not have a fifth pie slice.

To set the *Color* argument you can use either a hexadecimal value or VBScript's **RGB** function.

Please note that a template only provides textures and colors for pie slices. You MUST SPECIFY the pie slice labels and legend labels if you

want to display them in your jpeg.

IMPORTANT: You MUST load the desired template BEFORE calling **AddData**, otherwise values set in the **AddData** call will be overwritten by the pie chart attributes stored in the template.

See Also: LoadTemplate | Overview (Dundas Pie Chart Server Control)

AddLabel

Overview | Properties | Methods

Call this method to set legend elements.

Syntax

long = ChartObject.AddLabel(Text As String)

The **AddLabel** method syntax has the following parts:

Part	Description
Text	The text for the legend element.
	Returns a <i>long</i> indicating the success or failure of the operation. Zero (0) is returned if successful, otherwise a non-zero value is returned.

Returns

A long which indicates the success/failure of the operation. Zero (0) is returned if successful, otherwise a non-zero value is returned. If an error occurs use this return value as the argument for GetErrorText in order to retrieve a relevant text error message.

Remarks

AddLabel lets you set legend elements programmatically.

AngleX

Overview | Properties | Methods

Set this property to the amount of rotation around the X-axis for the previously displayed chart object before calling the Rotate method.

Syntax

PieChartObject.AngleX = [double]

The **AngleX** property syntax has the following parts:

Part	Description
double	The degree of rotation of the chart object around the X-axis the last time it was displayed.

Remarks

In order to rotate a chart you must let the chart instance know what the rotational angles were after the last rotation operation (defaults are used when the first rotation occurs) before calling the **Rotate** method. This means that you must preserve state information (e.g. use session-level variables, hidden input boxes, etc.) between Chart object creation/deletion.

Once you have set the **AngleX**, <u>AngleY</u> and <u>AngleZ</u> properties the chart object then knows its location as set by a previous rotation operation, and you can then call the **Rotate** method to rotate the chart. Once this is done MAKE SURE that you record the new X, Y and Z angles for the chart (to be used if the chart is rotated again). To save the chart's new position record (e.g. session variables) the **AngleX**, **AngleY** and **AngleZ** properties.

Refer to the Rotation Tutorial for sample source code demonstrating

rotation.

See Also: Rotate | Tutorial: Rotation

AngleY

Overview | Properties | Methods

Set this property to the amount of rotation around the Y-axis for the previously displayed chart object before calling the Rotate method.

Syntax

PieChartObject.AngleY = [double]

The **AngleY** property syntax has the following parts:

Part	Description
double	The degree of rotation of the chart object around the Y-axis the last time it was displayed.

Remarks

In order to rotate a chart you must let the chart instance know what the rotational angles were after the last rotation operation (defaults are used when the first rotation occurs) before calling the **Rotate** method. This means that you must preserve state information (session-level variables, hidden inputboxes, etc.) between Chart object creation/deletion.

Once you have set the <u>AngleX</u>, **AngleY** and <u>AngleZ</u> properties the chart object then knows its location as set by a previous rotation operation, and you can then call the **Rotate** method to rotate the chart. Once this is done MAKE SURE that you record the new X, Y and Z angles for the chart (to be used if the chart is rotated again). To save the chart's new position record (e.g. session variables) the **AngleX**, **AngleY** and **AngleZ** properties.

Refer to the <u>Rotation Tutorial</u> for sample source code demonstrating rotation.

See Also: Rotate | Tutorial: Rotation

AngleZ

Overview | Properties | Methods

Set this property to the amount of rotation around the Z-axis for the previously displayed chart object before calling the Rotate method.

Syntax

PieChartObject.AngleZ = [double]

The **AngleZ** property syntax has the following parts:

Part	Description
double	The degree of rotation of the chart object around the Z-axis the last time it was displayed.

Remarks

In order to rotate a chart you must let the chart instance know what the rotational angles were after the last rotation operation (defaults are used when the first rotation occurs) before calling the **Rotate** method. This means that you must preserve state information (session-level variables, hidden inputboxes, etc.) between Chart object creation/deletion.

Once you have set the <u>AngleX</u>, <u>AngleY</u> and **AngleZ** properties the chart object then knows its location as set by a previous rotation operation, and you can then call the **Rotate** method to rotate the chart. Once this is done MAKE SURE that you record the new X, Y and Z angles for the chart (to be used if the chart is rotated again). To save the chart's new position record (e.g. session variables) the **AngleX**, **AngleY** and **AngleZ** properties.

Refer to the <u>Rotation Tutorial</u> for sample source code demonstrating rotation.

See Also: Rotate | Tutorial: Rotation

CodePage

Overview | Properties | Methods

Set this property to utilize a language other than english.

Syntax

PieChartObject.CodePage = [long]

The **CodePage** property syntax has the following parts:

Part	Description
_	Represents a valid code page for the system which is running the ASP engine.

Remarks

A code page is a character set which is used to interpret and display data. Differing languages and locales can be used by setting this property to the desired codepage.

Set this property at the beginning of your ASP code, before you call SendJPEG.

Please note that the Dundas Pie Chart Server component fully supports Unicode.

Refer to the table below for a listing of code pages:

Code Page Description

1252

ISO Character Set (default)

850	Multilingual
437	U.S. English
874	Thai
932	Japanese
936	Chinese (simplified)
949	Korean
950	Chinese (traditional)
1250	Central European

Cyrillic

1253 Greek

1251

1254 Turkish

1255 Hebrew

1256 Arabic

1257 Baltic

For more information about code pages refer to the MSDN library.

CreateJpegFile

Overview | Properties | Methods

Saves the resulting jpeg to disk.

Syntax

long = PieChartObject.CreateJpegFile(FileName As String, Width As
Long, Height As Long, [Compression As Long])

The **CreateJpegFile** method syntax has the following parts:

Part	Description
FileName	The full pathname of the file to be saved to disk.
Width	Width of the saved jpeg in pixels.
Height	Height of the saved jpeg in pixels.
Compression	Amount of compression (0 to 100).
long	Returns a <i>long</i> indicating the success or failure of the operation. Zero (0) is returned if successful, otherwise a non-zero value is returned.

Returns

A long which indicates the success/failure of the operation. Zero (0) is returned if successful, otherwise a non-zero value is returned. If an error occurs you can use this return value as the argument for GetErrorText in order to retrieve a relevant text error message, or alternatively you can rem out the "On Error Resume Next" statement and observe the thrown exception for a description of the error.

Remarks

You can use this as a test method to see if the control is actually producing a jpeg.

You can also use this method in a VB (Visual Basic) application and then call VB's *LoadPicture* method to assign the jpeg to a Picture Box or Image control.

See Also: SendJPEG

DirTemplate

Overview | Properties | Methods

Sets or retrieves the directory where templates are stored.

Syntax

PieChartObject.DirTemplate = [string]

The **DirTemplate** property syntax has the following parts:

Part	Description
string	The full pathname of the folder which stores the templates.

Remarks

The template folder is where all of your textures must be located in order to be available for loading into a pie chart.

You MUST set this property before loading a template.

See Also: DirTexture

DirTexture

Overview | Properties | Methods

Set or retieves the directory where the textures must be stored.

Syntax

PieChartObject.DirTexture = [string]

The **DirTexture** property syntax has the following parts:

Part	Description
string	The full pathname of the folder which stores the textures.

Remarks

Your textures must be located in this folder.

NOTE: You MUST set this property BEFORE loading a template.

See Also: <u>DirTemplate</u>

GetErrorText

Overview | Properties | Methods

Outputs a relevant error string based on a returned error code.

Syntax

string = PieChartObject.GetErrorText(ErrorCode As Long)

The **GetErrorText** method syntax has the following parts:

Part	Description
ErrorCode	An error code which is returned from a function call.
string	An error string which describes the error that occurred.

Returns

An error string.

Remarks

All functions with the exception of **GetErrorText** return an error code (a long) which signifies the success/failure of the operation. Zero (0) is returned if the operation is successful, otherwise a non-zero value is returned. If the return is non-zero you can use **GetErrorText** to retrieve an error string.

NOTE: The Pie Chart Server control will also throw an exception if an error occurs, so instead of using the GetErrorText method you

can examine the thrown exception for a meaningful error message.

When attempting to obtain a meaningful error message you MUST load the page producing the jpeg (i.e. the page with the SendJPEG call) directly into your browser (i.e. do not load this page with an tag from another page).

GetSelectedPosition

Overview | Properties | Methods

Use this method to determine which data element the user selected (zero-based).

Syntax

long = PieChartObject.GetSelectedPosition()

Remarks

Call this method **ONLY WHEN** you have successfully called the <u>Select</u> method inside the page which is handling the item selection, specified by the SRC attribute of the <A> tag immediately preceding the server-side chart image tag (e.g. <A HREF SRC="PageToHandleSelectedItem.asp">). The return value will be the data element that the user clicked on.

This is useful in drilldown scenarios where you can take appropriate action (e.g show another chart image) based on the pie slice the user selected.

GetSelectedPosition is zero-based, so a return of zero (0) indicates that the user selected the first element added to the chart with the **AddData** method.

You should first call <u>GetSelectedSeries</u> to make sure that the user clicked on a pie slice or a legend item.

If the user clicks on a legend item then the corresponding pie slice's position is returned.

IMPORTANT: The return number represents the selected element as per the order that the data elements were added via the <u>AddData</u> method. This has certain ramifications since a pie chart types can be set up to use

a "collected" element. If an element is collected it will still be counted as a position. For example, if the first element is collected (and therefore only visible as the collected element) then **GetSelectedPosition** will return one (as opposed to zero) if the user clicks on the **first visible** pie slice.

NOTE: Even though the page which handles the selection does not output a jpeg we **MUST** reproduce the exact same chart which is created by the page within the tag (the page which outputs the jpeg via **SendJPEG**). In order to guarantee that the exact same chart is reproduced we *highly recommend* using a server-side include for the code which creates the chart (starting from the CreateObject call all the way to setting the Chart object to Nothing).

For sample source on this refer to the <u>Tutorial</u>: <u>Drilldown and Selection of Data Elements</u>. You can also examine the Selection, Grand Prix or Gross Domestic Product (Drilldown) samples which are distributed with the Chart Server Control for sample source code about the **Select**, **GetSelectedPosition** and **GetSelectedSeries** methods.

See Also: GetSelectedSeries | Select

GetSelectedSeries

Overview | Properties | Methods

Use this method to determine which data series the user selected (zero-based). Note that pie charts only use one data series.

Syntax

long = PieChartObject.GetSelectedSeries()

Remarks

GetSelectedSeries is zero-based, and will:

- return -999 if the user clicked on the collected data element.
- return a negative number if the user clicked anywhere except a data element or the legend item that corresponds to a pie slice.
- always return zero (0) if the user selected an actual data element (i.e. pie slice).

Even though pie charts have only one data series you should still call this method to make sure an actual slice or legend item was selected!

Call this method **ONLY WHEN** you have successfully called the <u>Select</u> method inside the page which is handling the item selection (as specified by the SRC attribute of the <A> tag immediately preceding the serverside chart image tag). The return value will be the data element that the user clicked on inside of the chart server-side imagemap (<A HREF SRC="PageToHandleSelectedItem.asp">).

Important: Even though the page which handles the selection (i.e the page which has the **GetSelectedSeries** call) does not output a jpeg we **MUST** reproduce the exact same chart which is created by the page

within the tag (the page which outputs the jpeg via **SendJPEG**). In order to guarantee that the exact same chart is reproduced we *highly recommend* using a server-side include for the code which creates the chart (starting from the CreateObject call all the way to setting the Chart object to Nothing).

For sample source on this refer to the <u>Tutorial</u>: <u>Drilldown and Selection of</u> <u>Data Elements</u>.

See Also: <u>GetSelectedPosition</u> | <u>Select</u> | <u>SetCollectedLimit</u> | <u>SetCollectedProp</u>

LoadTemplate

Overview | Properties | Methods

Loads a template into a pie chart.

Syntax

long = PieChartObject.LoadTemplate(FileName As String)

The **LoadTemplate** method syntax has the following parts:

Part	Description
FileName	The name of the template to load.
long	Returns a <i>long</i> indicating the success or failure of the operation. Zero (0) is returned if successful, otherwise a non-zero value is returned.

Returns

A long which indicates the success/failure of the operation. Zero (0) is returned if successful, otherwise a non-zero value is returned. If an error occurs you can use this return value as the argument for GetErrorText in order to retrieve a relevant text error message, or alternatively you can rem out the "On Error Resume Next" statement and observe the thrown exception for a description of the error.

Remarks

Templates define chart characteristics such as the title, default textures and colors for data elements (pie slices), labels, lighting properties, etc. Use the Template Creator to create or change templates for use with the Dundas Pie Chart Server control.

Call **LoadTemplate** BEFORE you set the <u>Title</u> property or use any of the object's methods, otherwise the default template values will overwrite your properties/data.

For a description of the steps which should be followed to create a pie chart and the order in which they should be performed see the <u>Dundas</u> <u>Pie Chart Server Control Overview</u>.

See Also: <u>Template Creator Overview</u> | <u>DirTemplate</u>

Dundas Pie Chart Server Control Methods

Copyright © Dundas Software Ltd. 2000. All Rights Reserved

Overview | Properties | Methods

AddData Call this method to add data elements (slices) to a

pie chart.

<u>AddLabel</u> Sets the legend text for a pie slice.

<u>CreateJPEGFile</u> Saves a pie chart to disk as a jpeg file.

GetErrorText Retrieves an error string based on a returned error

code.

GetSelectedPosition Returns the pie slice the user selected (clicked on).

GetSelectedSeries Returns the data series the user selected (clicked

on).

<u>LoadTemplate</u> Loads a template into a pie chart object.

Select MUST be called before using GetSelectedPosition

and GetSelectedSeries.

<u>SendJPEG</u> Outputs the pie chart object in a jpeg format.

SetCollectedLimit Sets the limit at which data elements will be

incorporated into the collected data element.

<u>SetCollectedProp</u> Sets the properties of the collected data element.

<u>SetExploded</u> Explodes a pie slice.

Dundas Pie Chart Server Control Properties

Copyright © Dundas Software Ltd. 2000. All Rights Reserved

Overview | Properties | Methods

AngleX The amount of rotation around the X-axis for the

previously displayed chart object (must be set before

calling the **Rotate** method).

AngleY The amount of rotation around the Y-axis for the previously

displayed chart object (must be set before calling the

Rotate method).

AngleZ The amount of rotation around the Z-axis for the

previously displayed chart object (must be set before

calling the **Rotate** method).

<u>CodePage</u> The codepage to be used.

<u>DirTemplate</u> Sets or retrieves the folder which stores the templates.

<u>DirTexture</u> Sets or retrieves the folder which stores the textures.

<u>Title</u> Sets or retrieves the title of the pie chart.

Rotate

Overview | Properties | Methods

Call this method to rotate a chart.

Syntax

long = PieChartObject.Rotate(Angle As Double, Axis As Long)

The **Rotate** method syntax has the following parts:

Part	Description
Angle	This is the angle at which the rotation should be set for the axis passed in. The angle is specified in degrees, and is relative to the current position (e.g. the value specified in a previous call to Rotate or one found in the template).
Axis	The axis to have the chart rotated around. This value should be zero (0) for the X-axis, one (1) for the Y-axis, and two (2) for the Z-axis.
long	Returns a <i>long</i> indicating the success or failure of the operation. Zero (0) is returned if successful, otherwise a non-zero value is returned.

Returns

A long which indicates the success/failure of the operation. Zero (0) is returned if successful, otherwise a non-zero value is returned. If an error occurs use this return value as the argument for GetErrorText in order to retrieve a relevant text error message.

Remarks

Make sure you only call this method after adding all data to the chart object.

In order to rotate a chart you must let the chart instance know what the rotational angles were after the last rotation operation (defaults are used when the first rotation occurs) before calling the **Rotate** method. This means that you must preserve state information (session-level variables, hidden inputboxes, etc.) between Chart object creation/deletion.

To set the initial rotation angles use your preserved state information and set the <u>AngleX</u>, <u>AngleY</u> and <u>AngleZ</u> properties. Once you have set the previous rotation angles you can call the **Rotate** method to rotate the chart. Once this is done MAKE SURE that you record the new X, Y and Z angles for the chart (to be used if the chart is rotated again). To save the chart's new position record (e.g. session variables) the **AngleX**, **AngleY** and **AngleZ** properties.

NOTE: if you repeatedly call this method (to rotate around all three axis) and you then call <u>SendJPEG</u> you must remember the order of rotation for the different axis. In otherwords, if you rotate around the X, Y, and Z-axis in that order then the next time you rotate the chart you must rotate around the X, Y, and Z-axis using the same order.

Note that rotation occurs in screen coordinates, not chart coordinates (e.g. the z-axis is always at right angles to the screen).

Refer to the <u>Rotation Tutorial</u> for sample source code demonstrating rotation.

See Also: AngleX | AngleY | AngleZ | Tutorial: Rotation

Select

Overview | Properties | Methods

Call this method so that you can utilize the <u>GetSelectedPosition</u> and <u>GetSelectedSeries</u> methods when working with a server-side chart imagemap (i.e. implementing drilldown and/or selection).

Syntax

long = PieChartObject. Select(X As Long, Y As Long, Width As Long,
Height As Long)

The **Select** method syntax has the following parts:

Part	Description
Х	The X-coordinate of the mouse position where the user clicked. Use the querystring's first number (from the imagemap) for this argument.
Y	The Y-coordinate of the mouse position where the user clicked. Use the resulting querystring's second number for this argument.
Width	The height of the jpeg image (as set in the SendJPEG() call).
Height	The width of the jpeg image (as set in the SendJPEG() call).
long	Returns a <i>long</i> indicating the success or failure of the operation. Zero (0) is returned if successful, otherwise a non-zero value is returned.

Returns

A long which indicates the success/failure of the operation. Zero (0) is returned if successful, otherwise a non-zero value is returned. If an error occurs use this return value as the argument for GetErrorText in order to retrieve a relevant text error message.

Remarks

Call this method in a page which is loaded as a result of a server-side chart imagemap being clicked on in order to find out which data element was selected. This method is used when working with a drill-down type of scenario.

To set the *X* and *Y* arguments you will need to parse the passed querystring item (the mouse position with a syntax of "201,75") which results from the *IsMap* attribute within the tag. To parse this data () use the *InStr* VBScript function. The first number in the querystring will be the mouse' X-coordinate, while the second querystring value is the corresponding mouse' Y-coordinate.

IMPORTANT: Even though the page which handles the selection does not output a jpeg we **MUST** reproduce the exact same chart which is created by the page within the tag (the page which outputs the jpeg via **SendJPEG**). In order to guarantee that the exact same chart is reproduced we *highly recommend* using a server-side include for the code which creates the chart (starting from the CreateObject call all the way to setting the Chart object to Nothing).

The height and width arguments should match those specified in the corresponding **SendJPEG** call as well as those which may have been specified as Height and Width attributes in the relevant tag.

For sample source on this refer to the <u>Tutorial</u>: <u>Drilldown and Selection of Data Elements</u>. You can also examine the Selection, Grand Prix or Gross Domestic Product (Drilldown) samples which are distributed with the Chart Server Control for sample source code about the **Select**, **GetSelectedPosition** and **GetSelectedSeries** methods.

See Also: GetSelectedPosition | GetSelectedSeries | Tutorial: Drilldown

and Selection of Data Elements

SendJPEG

Overview | Properties | Methods

Outputs a binary stream in JPEG format by using the Response.BinaryWrite Asp method.

Syntax

long = PieChartObject.SendJPEG(Width As Long, Height As Long,
[Compression As Long = 20], [Smoothing As Long = 20])

The **SendJPEG** method syntax has the following parts:

Part	Description
Width	Width of the jpeg (pixels).
Height	Height of the jpeg (pixels).
Compression	The amount by which the original pie chart image is compressed. Ranges from 0 to 100 (no compression to maximum compression).
Smoothing	The amount of smoothing that occurs for the jpeg image. Ranges from 0 to 100 (no smoothing to maximum smoothing).
long	Returns a <i>long</i> indicating the success or failure of the operation. Zero (0) is returned if successful, otherwise a non-zero value is returned.

Returns

A long which indicates the success/failure of the operation. Zero (0) is

returned if successful, otherwise a non-zero value is returned. If an error occurs you can use this return value as the argument for <u>GetErrorText</u> in order to retrieve a relevant text error message, or alternatively you can rem out the "On Error Resume Next" statement and observe the thrown exception for a description of the error.

Remarks

Outputs a jpeg file by using the Response.BinaryWrite Asp method.

There are two ways to embed this jpeg file into an Asp page. The first is to use an tag and set the SRC attribute to the secondary Asp page which is responsible for creating the Asp Pie Chart object and creating the pie chart (see Overview for more details). The second method is the same as the first except we use an <A> tag instead of an tag. For an example of this see the Asp demo pages. You can also refer to the Overview for more information.

IMPORTANT: The Response.BinaryWrite method utilizes HTTP headers, and as a result you CAN NOT insert any html code into the Asp page before calling this method!

See Also: <u>CreateJPEGFile</u> | <u>Overview</u>

SetCollectedLimit

Overview | Properties | Methods

Sets the value at which a data element (pie slice) will be collected.

Syntax

long = PieChartObject.SetCollectedLimit(Limit As Double, [LimitType As Long])

The **SetCollectedLimit** method syntax has the following parts:

Part	Description
Limit	The value at which a data element will become a "collected" element.
LimitType	The limit type. If this is zero (0) then the size of the element as a percentage will be used. If this is set to one (1) then the size of the pie element will be used.
long	Returns a <i>long</i> indicating the success or failure of the operation. Zero (0) is returned if successful, otherwise a nonzero value is returned.

Returns

A long which indicates the success/failure of the operation. Zero (0) is returned if successful, otherwise a non-zero value is returned. If an error occurs you can use this return value as the argument for GetErrorText in

order to retrieve a relevant text error message, or alternatively you can rem out the "On Error Resume Next" statement and observe the thrown exception for a description of the error.

Remarks

Small data elements are "collected" and then displayed together as one pie slice called the "collected" element. **SetCollectedLimit** determines at which point an element will be a part of the collected element.

See Also: Overview | SetCollectedProp

SetCollectedProp

Overview | Properties | Methods

Sets the properties for the collected data element.

Syntax

long = PieChartObject.SetCollectedProp([Label As String], [Color As
Long = 16711680], [Texture As String])

The **SetCollectedProp** method syntax has the following parts:

Part	Description
Label	The label which will be displayed next to the collected data element.
Color	The underlying color to be used for the collected pie slice.
Texture	The texture to be used for the collected data element.
long	Returns a <i>long</i> indicating the success or failure of the operation. Zero (0) is returned if successful, otherwise a nonzero value is returned.

Returns

A long which indicates the success/failure of the operation. Zero (0) is returned if successful, otherwise a non-zero value is returned. If an error occurs you can use this return value as the argument for GetErrorText in order to retrieve a relevant text error message, or alternatively you can rem out the "On Error Resume Next" statement and observe the thrown exception for a description of the error.

Remarks

Small data elements are "collected" and then displayed together as one pie slice called the "collected" element.

SetCollectedProp sets the properties of the collected pie slice. If the template (which MUST be loaded before calling this method) defines default values for the collected data element then these default values will be used if you do not specify a collected property. For example, if you call **SetCollectedProp** and specify a label and underlying color to be used for the collected pie slice then the default values (if they are defined in the template) will not be used, but the default texture will be used since we did not specify a texture in the **SetCollectedProp** call.

See Also: <u>SetCollectedLimit</u> | <u>LoadTemplate</u> | <u>Template Creator</u> Overview

SetExploded

Overview | Properties | Methods

Call this method to explode a particular pie slice.

Syntax

long = PieChartObject.SetExploded([Index As Long = -1])

The **SetExploded** method syntax has the following parts:

Part	Description
	Zero (0) based index of the data elements (pie slices) in the pie chart. Set this to -1 (or do not specify the index at all) to explode the collected pie slice.
_	Returns a <i>long</i> indicating the success or failure of the operation. Zero (0) is returned if successful, otherwise a non-zero value is returned.

Returns

A long which indicates the success/failure of the operation. Zero (0) is returned if successful, otherwise a non-zero value is returned. If an error occurs you can use this return value as the argument for GetErrorText in order to retrieve a relevant text error message, or alternatively you can rem out the "On Error Resume Next" statement and observe the thrown exception for a description of the error.

Remarks

Call this function to explode a pie element or the collected element.

If you specify a data element to be exploded and the element ends up being collected it will not be exploded (unless the collected data element is set up to be exploded). However, if you were to decrease the limit at which elements are collected by calling SetCollectedLimit and the same pie slice is no longer a collected element it will then be exploded. In otherwords, it remembers that it is supposed to be exploded.

You MUST call AddData before exploding pie elements.

See Also: SetCollectedLimit | SetCollectedProp

Title

Overview | Properties | Methods

Sets or retrieves the title of the pie chart.

Syntax

PieChartObject.Title = [string]

The **Title** property syntax has the following parts:

Part	Description
string	The title of the pie chart.

Remarks

Sets or retrieves the title of the pie chart. Use this property AFTER the template is loaded, otherwise the default title from the template will be used instead.

See Also: LoadTemplate

Template Creator Overview

Copyright © Dundas Software Ltd. 2000, All Rights Reserved

The Template Creator enables editing and manipulation of a pie chart template. A template is a binary file that stores reusable chart information, and all templates have a file extension of ".cuc". You can also create a new template with the Template Creator.

Templates define the characteristics of a chart, and are loaded by the Dundas Pie Chart Server control via the <u>LoadTemplate</u> method.

Pie chart properties are edited by property pages. To display a property sheet either double-click on a pie chart element or right-click over an element and select 'Properties'.

IMPORTANT: the Template Creator will only function correctly if your screen resolution is set up to display at least 32k colors.

The Dundas Pie Chart Server control can load a template and create a jpeg image from it. To minimize the amount of whitespace in the resulting jpeg make the pie chart image as large as possible in the Template Creator.

By default a new template will not display the last pie slice because it is collected. To view this last slice (which is collected and therefore not visible) open the Pie Element property page for ANY visible slice and set the Collected Data Limit to zero percent. This makes sure that no elements are collected (all pie slices will be visible). If you want to determine which slice is the collected slice open the same property page and set the Collected Data Limit to 100%, thereby causing all slices to be collected. The resulting pie chart will then display only the collected slice. To change the texture/color of the collected slice right-click over it, select the Properties menu item and then choose the Textures property page.

The values (sizes) of the slices are indicated by Tooltips which appear

when you rest the mouse cursor over a pie slice.

The creator allows you to use gradient colors for the template background, legends, static notes (labels) and the title.

Some chart properties can only be set by the Dundas Pie Chart Server control (like data values or labels), while others can only be set by the Template Creator (e.g. background colors and textures, the pie orientation or the light direction). Some properties can be set by both the control and by the Template Creator (e.g. textures and colors of pie slices). Refer to the table below for more detailed information:

Property Name	Template Creator	Dundas Pie Chart Server Control
Background		
all properties	Yes	No
Title		
position	Yes	No
font properties	Yes	No
text	Yes	Yes
Pie chart orientation	Yes	No

Light parameters and direction

Yes

No

Legend

position	Yes	No
border style	Yes	No
font properties	Yes	No
labels	No	Yes

Pie chart

size	Yes	No
rounding radius	Yes	No
number of rounding points	Yes	No
parameters for collected data	Yes	Yes
label for collected data	Yes	Yes

number of pie slices	Yes (slices will have default values)	Yes
side texture	Yes	No
disable textures	Yes	No
disable lighting	Yes	No

Pie slice (element)

value	No	No
color/texture	Yes (saved as the default values)	No
exploded	Yes (not saved)	No
label text	Yes (not saved)	No
label font	Yes	Yes

Static notes

all properties Yes N	10
----------------------	----

Please note that a template does not save data for a chart object.

The filename of the Template Creator is PieTEditor.exe.

To be able to resize a legend the AutoSize option MUST BE selected (in the <u>Lables</u> property page for a legend element).

IMPORTANT: for the Template Creator to work correctly the Intel compression library (ijl15.dll) must be located either in your system directory or at the same directory level as the Template Creator. Also, in order to use the distributed textures the "Textures" folder (which stores these textures) must also exist at the same directory level as the Creator).

See Also: <u>Dundas Pie Chart Server - Demo Overview</u> | <u>Overview</u> (<u>Dundas Pie Chart Server Control</u>) | <u>Collected Data</u>

How to use the Template Creator

The Template Creator lets you create and/or manipulate templates to be used with the <u>Dundas Pie Chart Server Control</u>. The Template Creator can be accessed via the Start Menu Entry named "Pie Chart Template Creator".

The <u>application menu</u> lets you create, open, and save templates to disk. The <u>popup menu</u> allows you to open a property sheet for property editing, switch to either Size or Move mode, and work with static text labels.

The properties of elements are available for editing via property sheets. The possible property pages are: <u>Pie Element; Data Label; Texture; Size - Move; Light; Labels; Line Patterns; Title;</u> and <u>Scene</u>.

Note that the property pages displayed depends on the type of element you click over, so only some of the pages listed above will be shown.

To rotate a chart use the keyboard's navigational keys. For more details see Rotating a Chart.

For a listing of pie chart properties that can be edited via the Template Creator see the <u>overview</u>.

By default a new template will not display the last pie slice because it is collected. To view this last slice (which is collected and therefore not visible) open the Pie Element property page for ANY visible slice and set the Collected Data Limit to zero percent. This makes sure that no elements are collected (all pie slices will be visible). If you want to determine which slice is the collected slice open the same property page and set the Collected Data Limit to 100%, thereby causing all slices to be collected. The resulting pie chart will then display only the collected 'slice'. You can then change the texture/color of the collected slice by right-clicking over it, selecting the Properties menu item and then choosing the Textures property page.

A new template randomly sets the colors of the pie slices.

IMPORTANT: for the Template Creator to work correctly the Intel compression library (ijl15.dll) must be located either in your system directory or at the same directory level as the Template Creator. Also, in order to use the distributed textures the "Textures" folder (which stores these textures) must also exist at the same directory level as the Template Creator).

See Also: Template Creator Overview | Collected Data Elements

Application Menu - Template Creator

The application menu lets you perform the following actions:

File

New Create a new template. By default the first slice is

collected.

Open Open a template for editing.

Save the template.

Save As Save the template with a new filename.

Pie

Add Pie Slice Add another pie slice to the pie chart.

Add 3 Pie

Adds three pie slices.

Slices

Delete Pie Slice Delete the last pie slice in the pie chart (each slice

is identified by an index).

Delete 3 Pie

Slices

Delete the last three pie slices in the template.

See Also: Popup Menu | Collected Data Elements

Collected Data Elements

Collected data consists of all pie slices which fall below a minmum value, and is indicated to the user with a special "collected" pie slice which has its own color/texture. This collected pie slice eliminates situations where there are numerous slices which can not be drawn properly (i.e. if there are twenty slices which are very small then the resulting pie chart would be very messy if there was no collected pie slice).

By default a NEW template will not display the last pie slice because it is collected. To view this last slice (which is collected and therefore not visible) open the Pie Element property page for ANY visible slice and set the Collected Data Limit to zero percent. This makes sure that no elements are collected (all pie slices will be visible). If you want to determine which slice is the collected slice open the same property page and set the Collected Data Limit to 100%, thereby causing all slices to be collected. The resulting pie chart will then display only the collected 'slice'. To change the texture/color of the collected slice right-click over it, select the Properties menu item and then choose the Textures property page.

A new template randomly sets the colors of the pie slices. The last slice is collected, so the collected pie slice is shown instead of the last slice.

The Collected Data Limit determines how small a slice must be before it is added to the collected slice. This can be either a percentage of the entire pie or an absolute value.

See Also: <u>How to use the Template Creator</u> | <u>Template Creator Overview</u> | <u>Pie Element Property Page</u>

Data Label Property Page - Template Creator

The data label property page is only available for pie slices.

To display this property page either double-click on a pie slice or right-click over a pie slice and select 'Properties'.

The following is a description of what you can do via this property page:

Edit Data Label Change the label text for the pie slice you right-

clicked over.

Orientation Set the orientation of the pie element label to either

vertical or horizontal.

Font (all items) Set the font type, font size, and font color.

Effects Set the color of the label.

See Also: <u>How to use the Template Creator</u> | <u>Popup Menu - Template</u> Creator

Gradient Property Page - Template Creator

The gradient property page is available for all template items except pie slices.

To display this property page either double-click on an element or rightclick over an element and select 'Properties'.

The following is a description of what you can do via this property page:

Solid Displays the selected item with a uniformly colored

background.

Gradient Displays the selected item with a colored gradient

background.

Transparent Displays the selected item with a transparent

background. Not available for the background of the

template itself.

Shading Styles:

Horizontal Shading occurs horizontally.

Vertical Shading occurs vertically.

Diagonal Up Shading occurs from the bottom left-hand corner to

the top right-hand corner.

Diagonal Shading occurs from the bottom right-hand corner to

Down the top left-hand corner.

From Corner Shading occurs from one corner only.

From Center Shading occurs from the center on outwards.

Color Determines which two (2) colors are to be used for

the shading.

Fill Styles Determines how the shading style will be

implemented. You can choose any one of the four (4)

available fill styles.

See Also: How to use the Template Creator

Labels Property Page - Template Creator

The labels property page is only available for legend elements.

To display this property page either double-click on a legend element or right-click over a legend element and select 'Properties'.

To be able to resize the legend the AutoSize option MUST be selected.

The following is a description of what you can do via this property page:

Change the text of a particular legend item. **Edit Labels**

Specify the font to be used for the legend item. **Font**

Font Style Specify the font style to be used for the legend item.

Size Specify the font size for the legend item.

Effects:

Color The text color of ALL the legend items.

Automatically sizes the legend items. This MUST be Auto Size

checked if you want to resize the legend.

See Also: How to use the Template Creator

Light Property Page - Template Creator

The light property page is available for the area element (background) as well as pie slices. Please note that setting light parameters for one chart element will change the light settings for the entire pie chart.

To display this property page either double-click on the background or pie slice or alternatively right-click over the background or a pie element and select 'Properties'.

The following is a description of what you can do via this property page:

Directional Sphere

Use this item to set the direction that the light will strike your pie chart. You can either use the red directional arrows or alternatively you can left-click, hold the (large red ball) button down and move the directional sphere in the

desired direction.

Light Disabled

Disable all lighting effects.

Rotation Step

Sets the increment by which the light direction will change (e.g. how quickly the red ball will rotate).

Light

Parameters:

Ambient Change the RGB (Red, Green, Blue) values of the

ambient (non-directional) light.

Diffuse Change the RGB values of the diffuse light. Diffuse light is reflected, and its intensity varies depending on the angle between the pixel's normal vector and the direction of the light source.

Specular

Change the RGB values of the specular light. Specular light is also reflected, but its intensity varies with the angle between the viewer and the direction of the reflected light.

See Also: How to use the Template Creator

Line Patterns Property Page - Template Creator

The line patterns property page is only available for legend elements.

To display this property page either double-click on a legend element or right-click over a legend element and select 'Properties'.

The following is a description of what you can do via this property page:

Default You may select one of several pre-defined line patterns

Patterns for the legend's outline.

Pattern Use the radio buttons to define your own custom

legend outline.

Line

Parameters:

Line Width Sets the width of the legend's outline.

Color Sets the color of the legend's outline.

Factor Magnifies the specified line pattern. For example, if the

outline consists of dot - space - dot and you magnify by a fctor of two then the resulting outline will then be: dot-

dot - space - space - dot - dot.

See Also: <u>How to use the Template Creator</u>

Pie Element Property Page - Template Creator

The pie element property page is only available when you click over a pie slice.

The properties exposed by this page apply to ALL pie elements except for the Exploded option (only applied to the slice which was clicked over).

To display this property page either double-click on a pie slice or rightclick over a pie slice and select 'Properties'.

The following is a description of what you can do via this page:

Pie Parameters:

Number of The number of points used to draw the

Points circumference of the pie chart.

Vertical Radius The radius of the pie edge. Imagine the pie chart

when it is rotated to the point where you can only see its edge. The rounding of the pie at both ends

is determined by this value.

Vertical Number The number of points to be used to define the

of circumference of the pie chart's edge (e.g.

Points thickness). This property determines the number of

points used to draw that part of the pie whose radius is determined by the Vertical Radius.

Exploded Explodes/collapses the relevant pie slice.

Collected Data: There is only one collected pie element. You may

set the collected properties listed below by clicking

over ANY pie element. To set other collected

properties (e.g. texture) click over the collected pie

element itself.

Absolute The value used to determine if a slice is collected

will be an absolute value.

Percentage The value used to determine if a slice is collected

will be a percentage of the entire pie chart.

Limit The value (either absolute or a percentage) at

which a pie slice will be collected.

See Also: <u>How to use the Template Creator</u> | <u>Template Creator Overview</u>

Popup Menu - Template Creator

The popup menu lets you perform the following actions:

Properties Displays the property sheet appropriate for the

element which you right-clicked over.

Get Another

Element

You can select an area element (background), legend element or title element. Another popup menu will appear, allowing you to edit the element.

Add Notes Lets you insert a static notes element (a generic text

element) when you click over the background.

Switch to Size/ Switch to Move Toggles between sizing mode and moving mode.

Delete Delete the static notes element which you clicked

over. Only for static notes.

To display the popup menu right-click over any pie chart element.

NOTE: the items displayed by the popup menu are dependent on where you right-click in the pie chart, so not all of the preceding items may be displayed.

See Also: Application Menu - Template Creator

Rotating a Chart - Template Creator

To rotate a chart object in the Template Creator use the keyboard's navigational keys. Refer to the following table for a detailed description on how the different keys rotate a pie chart:

Key Description

Up Arrow Rotates the top of the chart away from the user.

Down Arrow Rotates the top of the chart towards the user.

Left Arrow Rotates the chart clockwise around its vertical axis.

Right Arrow Rotates the chart counter-clockwise around its vertical

axis.

Home Rotates the chart counter-clockwise around its horizontal

axis.

End Rotates the chart clockwise around its horizontal axis.

Page Up Same as the End key.

Page Down Same as the Home key.

See Also: <u>How to use the Template Creator</u>

Scene Property Page - Template Creator

The scene property page is only available when you right-click over a pie slice.

To display this property page either double-click on a pie slice or right-click over a pie slice and select 'Properties'.

The following is a description of what you can do via this property page:

Rotation To rotate the pie chart object use either the red

directional arrows or left-click on the object and rotate it

while holding the left mouse button down.

Rotation Step Determines the increment by which the scene will be rotated.

Reset Resets the pie chart to its default position.

See Also: How to use the Template Creator | Template Creator Overview

Size - Move Property Page - Template Creator

The size-move property page is available for every pie chart element except the background (which is an area element).

To display this property page either double-click on an element or right-click over an element and select 'Properties'.

Every element has an associated bounding rectangle (you can see this if you click on the element in question). When you view the size-move page notice that there are two sets of arrows: an inner set and an outer set separated by the outline of a square. This square represents the bounding rectangle of the element that is being sized/moved. Arrows inside and outside of the same edge for the square outline will move the associated bounding rectangle side in the direction indicated by the arrow.

The difference between moving and resizing is this: if sizing mode is on then changing one side of the bounding rectangle WILL NOT result in any of the other sides of the bounding rectangle being moved as well. If moving mode is selected then adjusting one bounding edge WILL result in the rest of the bounding edges being moved as well, since we are moving the pie chart element and not resizing it.

To size/move an object click on the object with the left mouse button and size/move it by dragging the mouse.

The following is a description of what you can do via this property page:

Object:

Element Selects one element to move/resize.

Group Used to move/size a group of pie chart

elements. This is actually used with other

Dundas Chart chart types and is not applicable

for this sample.

Chart Move/resize the chart object.

Operation:

Size Size the selected pie chart object.

Move the selected object.

Steps:

Horizontal step Sets the horizontal increment by which the pie

chart or pie chart object will be moved/resized.

Vertical step Sets the vertical increment by which the pie

chart or pie chart object will be moved/resized.

See Also: <u>How to use the Template Creator</u>

Texture Property Page - Template Creator

The texture property page is available for pie slices, legends, and the background. It lets you specify textures and colors and also lets you import textures (jpeg or bitmap formats).

Note that the elements for this property page may differ, depending on the chart element the Textures page applies to.

To display this property page either double-click on an appropriate element or right-click over the element and select 'Properties'.

The following is a description of what you can do via this property page:

Type

No Texture	No texture will be used.
Texture	Use the texture selected from the texture window (the window to the immediate right of the Texture button). When this option is selected you can not specify a color.
Colored Texture	Enabled for pie slices. This lets you specify a texture and a color.

Tile

Enabled for legends and the background. If this is checked then the selected texture will be tiled. If it is not selected the texture will be enlarged to fill the relevant window area.

Side Texture Only available for pie slices. Sets the texture to be

used for the sides of a pie slice. This is useful for

exploded pie segments.

Group Used for other Dundas Chart chart types. Not

applicable to the pie chart.

Back to Series Used for other Dundas Chart chart types. Not

applicable to the pie chart.

Textures Disabled Disables the use of textures. Note that the elements

still own the textures.

Color Select a color for the element. You can specify a

color AND a texture for a pie element.

Import Texture Import a texture to be used for the element. The

texture should be a jpeg or bitmap, and the

maximum size of the image is 1024×1024 pixels. The width and height of the image must also be to the base 2 (e.g. $2^m = Width$ and $2^n = Height$ in

pixels).

Textures or colors for a legend element will be applied to the background of the entire legend.

IMPORTANT: The maximum size of an imported texture (jpeg or bitmap) is 1024 X 1024 pixels. Also, the width and height of the image must be to

the base 2 (e.g. 2^m = Width and 2^n = Height in pixels, where n and m are whole numbers).

See Also: How to use the Template Creator

Title Property Page - Template Creator

The title property page is available for title elements only.

To display this property page either double-click on a title element or right-click over the element and select 'Properties'.

The following is a description of what you can do via this page:

Edit Title Determines the text of the title element.

Orientation

Horizontal The horizontal orientation can be changed for

static notes elements.

Vertical The vertical orientation can be changed for static

notes elements.

Effects

Auto Size Automatically sizes the title in its bounding

rectangle.

Vertical Center Centers the title vertically within its bounding

rectangle. Only available if Auto Size is not

selected.

Horizontal Center Centers the title horizontally within its bounding

rectangle. Only available if Auto Size is not

selected.

Color Sets the color of the title text.

Font Sets the font to be used for the title.

Font Style Sets the font style to be used for the title.

Size The size of the font to be used.

See Also: How to use the Template Creator

Dundas Pie Chart Overview

Copyright © Dundas Software Ltd. 2000. All Rights Reserved

This free ASP component consists of two parts - the <u>Template Creator</u> and the <u>Dundas Pie Chart Server control</u> (implemented via ASP pages).

A template is a binary file that stores reusable chart information, and the Template Creator lets you create new templates and also modify existing ones. The Dundas Pie Chart Server control loads a template, adds data and then creates a pie chart image in a jpeg format. This image can then be saved to file or transferred directly to a web page.

Note that the Dundas Pie Chart Server control and the Template Creator both use OpenGL libraries (OpenGL32.dll and glu32.dll). These libraries are installed by default with Windows 98 and Windows NT. In addition they also depend on the presence of the ijl15.dll Intel Compression library, which MUST exist at the same directory level as both the server control and the Template Creator (unless it is in your system directory).

See Also: Pie Chart Control Overview | Template Creator Overview

Tutorial: Creating a Basic Pie Chart

To use this control in a web site you will need to use two (2) web pages. One ASP page (let's call it the secondary page) creates an instance of the ASP Pie Chart control, loads a template, adds the specified data and then calls <u>SendJPEG</u> to send the data to the client. The other page (lets call it the main page) retrieves the pie chart image by embedding the secondary ASP page in an tag or <A> tag with the SRC attribute set to the secondary page. The main page can be either an html or ASP file.

If you do not know how to create a template using the Template Creator then click here for instructions on how to do so. This tutorial DOES NOT cover how to make your pie chart templates, it assumes you already have a template ready to be loaded.

IMPORTANT: the Intel compression library (ijl15.dll) MUST be located in either your system directory or at the same directory level as both the chart server control and the Template Creator, otherwise both the control and the Creator will not function correctly.

Click <u>here</u> for instructions on how to debug the Pie Chart Server. *For further information concerning debugging, as well as FAQs, and troubleshooting go to our <u>developer site</u>.*

To output a pie chart jpeg perform the following actions in their specified order:

- 1. Create the main page (ASP or html) and embed the secondary page (ASP) into it with either an tag or <A> tag.
- 2. Create the secondary page (ASP) and perform the following actions in their indicated order:
- The Pie Chart Server will throw an exception if an error occurs, so enable error handling with an "On Error Resume Next" statement.

- Create an ASPPieChart object by using CreateObject (e.g. CreateObject("Dundas.PieChartServer.1")).
- Set the directory properties of the control which specify working directories for different aspects of chart activity. These properties are <u>DirTemplate</u> and <u>DirTexture</u>, and deal with the directories used to store templates and textures, respectively.
- Load a template via the <u>LoadTemplate</u> method.
- Set the properties of the pie chart (e.g. exploded pieces, collected element properties, etc.)
- Add pie chart data with the <u>AddData</u> method.
- Send the resulting jpeg directly to a client with the <u>SendJPEG</u> method or alternatively you can save the jpeg to disk with <u>CreateJPEGFile</u> and then present the image to a user with a standard tag.
- If you want to generate another Jpeg using the same template then repeat Steps 4 to 6. If you want to create another jpeg with a different template repeat steps 1 to 6.
- Destroy the Pie Chart object by setting it to Nothing (e.g. Set objPieChart = Nothing).
- MAKE SURE that there is no html code in this ASP page which outputs the jpeg.

The ProgID of the control is: Dundas.PieChartServer.2

Example

'Displays a pie chart in an html page

```
'lets make the main page, and call it Main.htm <Html> <Body> <IMG SRC="MakeJpeg.asp"> </Body></Html>
```

'now make the secondary page which outputs the jpeg 'let's call it MakeJpeg.asp <%

'make sure browser never uses cached copy of jpeg

Response.Buffer = True

Response.CacheControl = "Private"

Response. Expires = -100

'enable error handling, since an exception will be thrown if an error occ 'disable the Resume Next statement and load this page directly into yo 'the raised exception.

On Error Resume Next

'retrieve the physical directory where ASP pages are located strPath = Server.MapPath(".")

'create an instance of the control

Set objPieChart = Server.CreateObject("Dundas.PieChartServer.1")

'set the Template directory of the control
objPieChart.DirTemplate = strPath & "\Templates\"

'set the Textures directory of the control
objPieChart.DirTexture = strPath & "\Textures\"

'load any template (made with the Template Editor)
ret = objPieChart.LoadTemplate("Textures.cuc")

'check for an error. If the method call is not successful then the

'returned error code will not be zero (0) and in addition an exception w
'See the "Debugging" section at the bottom of this page for inform

If ret <> 0 Then

Response.Write objPieChart.GetErrorText(ret)

End If

'add 3 slices to the pie chart and specify values (sizes), slice labels and

```
ret = objPieChart.AddData(25, "Label1", "Slice1")
ret = objPieChart.AddData(25, "Label2", "Slice2")
ret = objPieChart.AddData(25, "Label3", "Slice3")
'check for an error. If method not successful then output the
' corresponding error string. See the "Debugging" section below for
' more details concerning debugging the pie chart server
If ret <> 0 Then
Response.Write objPieChart.GetErrorText(ret)
End If

'output graphics, specifying the width and height of the image in pixels
ret = objPieChart.SendJPEG(450,350)
If ret <> 0 Then
Response.Write objPieChart.GetErrorText(ret)
End If
```

Notice that there is NO HTML CODE in this page which outputs the jpeg!

Debugging the Pie Chart Server

%>

'destroy AspPieChart object
Set objPieChart = Nothing

If an error occurs then an exception will be raised and the returned error code will not be zero (0). Since an exception will be raised MAKE SURE that the page which outputs the jpeg has an "On Error Resume Next" statement.

To debug the Pie Chart Server control you should load the page which contains the "SendJPEG" call directly into your browser, and you can then use either the control's GetErrorText method (explained in detail below) or you can merely examine the exception thrown by the control (make sure the "On Error Resume Next"

statement is disabled).

If an error occurs and a pie chart image is not produced you can also retrieve a relevant error string by loading the page which produces the JPEG directly into your browser and observing the error string generated by the control's GetErrorText method. For example, in the preceding sample code you could load the MakeJpeg.asp page directly into your browser and then see what **GetErrorText** returns.

Note that this method of retrieving relevant error strings will only work if the page that produces the JPEG image contains NO html code (e.g. the page only contains server-side ASP code).

See Also: <u>Tutorial:</u> <u>Drilldown and Selection of Data Elements | <u>Tutorial:</u> Rotation</u>

Tutorial: Drilldown and Selection of Data Elements

Drilldown and the selection of chart items is accomplished by using a chart image as a hyperlinked server-side imagemap. To use a chart image as a server-side imagemap specify the *IsMap* attribute inside the relevant tag.

Use the following syntax in the ASP page which displays a chart being used as the imagemap: . Note that the specified Asp page in the <A> tag is the page which must determine which series/data element the user selected.

To determine what data series/data element the user selected (clicked on) call the <u>Select</u> method and then examine the returns from the <u>GetSelectedSeries</u> and <u>GetSelectedPosition</u> methods.

Once you have determined what the user selected you can perform the appropriate action, giving you "drilldown" capabilities.

Note that clicking on a server-side imagemap will allow the page handling the selected item to retrieve the mouse coordinates of the clicked-on position since the mouse coordinates are available via the resulting querystring. Use VBSCript's *InStr* method to parse the querystring and retrieve the mouse X, Y-coordinates. Then use these X and Y values when calling the **Select** method.

IMPORTANT: the EXACT same chart must be reproduced in the page which handles the selection as compared to the Chart generated by the page specified within the tag. Therefore it is *highly recommended* that you use a server-side include for the source code which actually produces the chart image (starting from the CreateObject call all the way to setting the Chart object to Nothing). This will ensure that the chart produced in the page which handles the selection (using the **Select, GetSelectedPosition**, and **GetSelectedSeries** methods) will

be exactly the same as the chart produced by the page which outputs the chart jpeg (using the SendJPEG method).

Refer to the sample source code below for further clarification.

If you do not know how to create a template using the Template Creator then click here for instructions on how to do so. This tutorial DOES NOT cover how to make your pie chart templates, it assumes you already have a template ready to be loaded.

Example

Scenario - We will use a pie chart as an imagemap (with 4 pie slices) in the main page (Main.asp).

- When the user selects an element we will explode the selected elemen
- Note that we are using Session-level variables.
- A secondary page named MakeJpeg.asp will create and output the resulting jpeg to the main page.
- To see if a function call was successful you can examine the returned error code (zero equals success). To get a relevant error string use the GetErrorText method.

Main.asp

```
<%@ Language=VBScript %>
<%Response.Buffer = True %>
<HTML>
<HEAD></HEAD>
<BODY>
<%
```

Dim QString 'the querystring resulting from the user clicking 'on the Chart server-side imagemap

Dim Position 'the data element the user selected

Dim XPosition 'the mouse X-coordinate where the user clicked

Dim YPosition 'the mouse Y-coordinate wherre the user clicked

Dim ret 'returned error code from function call

Dim objChart 'Dundas Chart Server object Dim strErrorCode 'string error message

'retrieve the querystring with the mouse coordinates QString = Request.QueryString()

'initialize the selected position to -1 so that the SetExploded call does ${\bf r}$ ' initially explode a data element.

Session("Position") = -1

'the following code only executes if the user clicked on a pie slice If QString <> "" Then

Position = InStr(1,QString,",",1)

XPosition = CInt(Mid(QString,1,Position - 1))

YPosition = CInt(Mid(QString,Position+1))

'create instance of the Chart control
Set objChart = CreateObject("Dundas.ChartServer")

'NOTE: Even though this page does not output a jpeg we MUST ' reproduce the exact same chart which is produced by the page within ' the tag. In order to guarentee that the exact same chart is ' reproduced we highly recommend using a server-side include for the ' code which creates the chart (starting from the CreateObject ' call all the way to setting the Chart object to Nothing).

'now set the control's directory properties
objChart.DirTexture = "c:\dschart\dschart_local\Textures\"
objChart.DirTemplate = "c:\dschart\dschart_local\Templates\"

'load a pie chart template made with the Template Creator 'note that a return value of zero indicates success ret = objChart.LoadTemplate("selection.cuc")

'now add some data to the pie chart

```
ret = objChart.AddData(3.2)
ret = objChart.AddData(2.8)
ret = objChart.AddData(9.1)
ret = objChart.AddData(4.5)
'now call the Select method so we can determine the selected element.
'note that the width and height specified in the Select method call
' should be the same as that specifed in the <IMG> tag found
' at the bottom of this page.
ret = objChart.Select(XPosition, YPosition, 300, 300)
'lets check return value (useful for debugging)
If ret <> 0 Then
strErrorCode = objChart.GetErrorText(ret)
End If
'NOTE: if you are using a chart other than a sphere or pie chart
' for the imagemap you would then need to call GetSelectedSeries
'as well as calling GetSelectedPosition. If you are using a sphere or pic
' chart and are using the "collected" data element you will still need
'to call GetSelectedSeries in order to determine if the user clicked on
' the collected data slice (return value will be -999).
Position = objChart.GetSelectedPosition()
'this session-level variable lets the MakeJpeg.asp page know what was
Session("Position") = Position
'release resources
Set objChart = Nothing
End If
%>
<!--Make the Dundas Chart server-side imagemap -->
<center><a href="Main.asp">
<center><IMG height=300 width=300 src="MakeJpeg.asp" ID=Pictur</pre>
</BODY>
```

```
</HTML>
MakeJpeg.asp
<%@ Language=VBScript %>
<% Option Explicit %>
<%
'disable caching so that image is always retrieved from server
Response.Buffer = True
Response.CacheControl = "Private"
Response. Expires = -1000
Dim Position 'item selected by the user
Dim ret 'returned error code from function calls
Dim objChart 'Dundas Chart Server instance
Dim strError 'error string
'retrieve the selected element as determined by the main page
Position = Session("Position")
'create an instance of the control
Set objChart = CreateObject("Dundas.ChartServer")
'set up the control's directory properties
objChart.DirTexture = "c:\dschart\dschart_local\Textures\"
objChart.DirTemplate = "c:\dschart\dschart_local\Templates\"
'load the pie chart template, created in the Template Creator
ret = objChart.LoadTemplate("selection.cuc")
'now add data to the pie chart
ret = objChart.AddData(3.2)
ret = objChart.AddData(2.8)
ret = objChart.AddData(9.1)
ret = objChart.AddData(4.5)
```

'now we will explode the data element selected by the user objChart.SetExploded(Position)

'now output the chart jpeg. Note that you can step through your code
' and check the strError variable if SendJPEG was unsuccessful
ret = objChart.SendJpeg(300,300)
If ret <> 0 Then
strError=objChart.GetErrorText(ret)
End If

'release resources
Set objChart = Nothing

See Also: Tutorial: Creating a Basic Pie Chart | Tutorial: Rotation

%>

Tutorial: Rotation Example

In order to rotate a chart you must let the chart instance know what the rotational angles were after the last rotation operation (defaults are used when the first rotation occurs) before calling the **Rotate** method. This means that you must preserve state information (session-level variables, hidden inputboxes, etc.) between Chart object creation/deletion.

To set the initial rotation angles use your preserved state information and set the <u>AngleX</u>, <u>AngleY</u> and <u>AngleZ</u> properties. Once you have set the previous rotation angles you can then call the **Rotate** method to rotate the chart. Before the chart object is destroyed MAKE SURE that you store the new rotation angles (to be used for the next rotation operation). this can be done using session variables, hidden input boxes, etc.

Refer to the sample source code below for further clarification.

Example

If you do not know how to create a template using the Template Creator then click here for instructions on how to do so. This tutorial DOES NOT cover how to make your pie chart templates, it assumes you already have a template ready to be loaded.

Scenario - We will allow the user to rotate a pie chart clockwise.

- The amount of rotation will be hard-coded to 20 degrees.
- State information concerning the last displayed position of the chart will be preserved by using session-level variables.
- Main.asp will display the chart, while MakeJpeg.asp will actually create the chart jpeg.

Main.asp

- <Html>
- <Head>
- </Head>

```
<Body><Center>
<IMG height="300" src="MakeJpeg.asp" width="300"></Center>
<form action="Main.asp" method="post" id="form1" name="form1">
<input type="submit" name="cmdRotate" id="cmdRotate" value="Rot
</form>
</Body>
</Html>
MakeJpeg.asp
<%
'disable caching so that image is always retrieved from server
Response.Buffer = True
Response.CacheControl = "Private"
Response. Expires = -1000
'create an instance of the control
Set objChart = Server.CreateObject("Dundas.ChartServer")
'set required directory properties
objChart.DirTexture = "c:\dschart\dschart local\Textures"
objChart.DirTemplate = "c:\dschart\dschart_local\Templates"
'load a pie chart template, made with the Template Creator
ret = objChart.LoadTemplate("Rotation.cuc")
'now add some data to the pie chart
ret = objChart.AddData(900)
ret = objChart.AddData(1900)
ret = objChart.AddData(200)
ret = objChart.AddData(400)
'if one rotation operation has already occurred then we need to
' let the chart know its previous location
If Session("AngleX") <> "" Then objChart.AngleX = Session("AngleX
If Session("AngleY") <> "" Then objChart.AngleY = Session("AngleY
```

If Session("AngleZ") <> "" Then objChart.AngleZ = Session("AngleZ")

'now rotate the chart 20 degrees clockwise (around the window's z-axis ret = objChart.Rotate(-20,2)

'now set session-level variables so we can set the chart object's 'previous location the next time we rotate the chart

Session("AngleX") = objChart.AngleX

Session("AngleY") = objChart.AngleY

Session("AngleZ") = objChart.AngleZ

'now output the pie chart jpeg
ret = objChart.SendJpeg(300,300)
'lets check the return value (useful for debugging)
If ret <> 0 Then
strErrorCode = objChart.GetErrorText(ret)
End If

'release resources
Set objChart = Nothing
%>

See Also: <u>Tutorial:</u> <u>Creating a Basic Pie Chart | <u>Tutorial:</u> <u>Drilldown and Selection of Data Elements</u></u>

Distributed Samples

The distributed ASP samples are implemented with two (2) ASP pages each. The first page (PieChart1.asp and PieChart2.asp for samples 1 and 2, respectively) sets up the user interface and displays the chart, while the second page (MakeJPEG1.asp and MakeJPEG2.asp for samples 1 and 2, respectively) loads the selected template, adds the specified data to the pie chart and then outputs the image as a jpeg. The second page is embedded into the first using an tag (you can also use an <A> tag).

To run the ASP demos you must be set up to host web pages with either Microsoft Internet Information Server (IIS) or Microsoft Personal Web Server.

The first demo uses four templates which are displayed to a user via a listbox in the main ASP page. Note that you can modify these templates by using the Template Creator any time you wish. The second ASP demo randomly selects one of five templates and also explodes pie slices at random.

To successfully run these demos make sure that the "Templates" and "Textures" folders exist at the same level as the ASP pages that make up the samples. The Template Creator should also be located here.

Dundas Upload Control 2.0

Copyright © Dundas Software Ltd. 2000, All Rights Reserved.

Overview | Properties | Methods

The Dundas Upload Control 2.0 is a free, fully tested commercial MTS component which allows you to accept, save and manipulate files uploaded via a web browser which is compliant with RFC 1867 (e.g. Netscape 3.0+ and Microsoft's IE 4.0+). It is designed to be used in Active Server Pages (ASP) which are hosted with Microsoft's IIS (Windows NT or Windows 2000).

The Upload control also ships with the UploadProgress component and the Dundas State Server. The UploadProgress component works in conjunction with the State Server and Upload control and enables web developers to quickly and painlessly utilize progress bars to display the progress of an upload operation. The State Server component can run anywhere on your network, so it is "web-farm" ready, and may be used to store any state information.

You can also use the Upload control along with our free <u>Dundas Mailer</u> <u>Control</u> in order to give your applications web-based email capabilities which can handle multiple-file upload operations. The Mailer control, like the Upload control, is also a fully developed and tested commercial product which is free.

Features List Installation/Redistribution
How to use the Documentation

Features List

The following are just some of the features of the Dundas Upload Control 2.0:

- It ships web-farm ready.
- MTS support (object pooling).
- ADO support via safe-array-of-bytes variants, allowing you to persist uploaded files as BLOBs.
- Handles uploading of multiple files.
- Upload form data either all at once or incrementally. Incremental retrieval of data allows the developer to access some of the form data prior to the upload as well as skip files to be uploaded.
- Lets you specify a maximum allowable limit for uploaded data, a maximum number of files to be uploaded as well as a maximum allowable file size.
- Implement a progress bar via the ProgressBar and StateServer components.
- Save uploaded files either to disk or to memory.
- Capable of performing numerous file and directory operations.
- Impersonate user accounts other than the IUSR (or default) account, thereby allowing you to perform operations for which the default account does not have permissions.
- COM registration.
- MacBinary support (for Macintosh client machines).
- Uploaded files ACL, owner and attribute manipulation.

Installation and Redistribution

A previous version of the Upload control (version 1.0) was shipped with the free Dundas Mailer control. This first control allowed users to upload multiple files and also supported other basic upload operations. It is recommended that you uninstall this previous version if you installed the Dundas Mailer control.

Controls and Samples Install

The name of the install file is: "AspUpload.exe".

The windows account under which the install runs needs to have **administrative priviledges** for a successful installation.

The Dundas State Server (StateServer.exe) when installed is set up to run as a normal executable. This has the advantage of giving you a GUI which displays the operations which have occurred or are occurring. Please note, however, that you can run the State Server as a service. Running the executable as a service can increase its performance up to five times (five hundred percent).

Start menu entries are as follows:

Dundas Software | Free Products | Asp Upload Control | ReadMe

Documentation Install

To uninstall this documentation run the install file again (FPDocInst.exe) and choose either "Remove From MSDN" if you integrated into the MSDN or "Remove All" if you did not integrate into the MSDN.

How to use the Documentation

Documentation for the Upload control consists of the following:

1. Compiled html help (optionally integrated into the MSDN). The html help provides detailed explanations for all members of the control as well as a control overview and tutorials. Topics which should be especially useful are:

- How to Use the Dundas Upload Control
- Overview (Dundas Upload Control 2.0)
- QuickStart (Dundas Upload Control 2.0)
- <u>Tutorial 1: Uploading Multiple Files and Using the Form and Files Collections</u>
- <u>Tutorial 2: Retrieving Form Data Incrementally Using the</u> GetNextFile Method
- Tutorial 3: ADO Support and Saving a File as a BLOB
- Tutorial 4: Implementing a Progress Bar
- 2. ReadMe.htm. This file is distributed with the Upload control, and consists of installation information, release notes and a troubleshooting section for your convenience.

Overview (Dundas Upload Control 2.0)

Copyright © Dundas Software Ltd. 2000, All Rights Reserved.

Overview | Properties | Methods

The Dundas Upload Control is an MTS component which allows you to accept, save and manipulate files uploaded via a web browser. It is used in Active Server Pages (ASP) which are hosted with either Microsoft's IIS (Windows NT or Windows 2000) or Microsoft's PWS (Windows 9X).

Dundas Upload Control 2.0 Features:

- Web farm ready.
- MTS support (object pooling).
- ADO support via safe-array-of-bytes variants. Allows you to persist uploaded files as BLOBs.
- Handles uploading of multiple files and form data.
- Upload form data either all at once or incrementally. Incremental retrieval of data allows the developer to access some of the form data prior to the upload as well as skip files to be uploaded.
- Lets you specify a maximum allowable limit for uploaded data, a maximum number of files to be uploaded as well as a maximum allowable file size.
- Implement a progress bar via the ProgressBar component and the Dundas StateServer.
- Save uploaded files either to disk or to memory.
- Performs numerous file and directory operations.
- Impersonate user accounts other than the IUSR (or default) account,

thereby allowing you to perform operations for which the default account does not have permissions.

- COM registration.
- MacBinary support (for Macintosh client machines).
- Uploaded files ACL, owner and attribute manipulation.
- Disable or set the default values of certain features via the registry.
- Handles file downloading.
- Optionally save uploaded files with unique filenames.

To create an instance of the Upload control use the CreateObject method with a ProgID of either "Dundas.Upload" or "Dundas.Upload.2".

To utilize named constants in your code use a server-side include for the DSUpload.inc file, which defines the constants for you.

Most methods of the control will throw an exception if an error occurs. Trap for the success/failure of the various operations by using an **On Error Resume Next** statement and then examine the Err object after a method call.

You should find the following topics especially useful:

- How to Use the Dundas Upload Control
- Tutorial 1: Uploading Multiple Files and Using the Form and Files Collections
- <u>Tutorial 2: Retrieving Form Data Incrementally Using the GetNextFile</u>
 <u>Method</u>
- Tutorial 3: ADO Support and Saving a File as a BLOB
- Tutorial 4: Implementing a Progress Bar

See Also: <u>Progress Bar Component Overview</u> | <u>State Server Component Overview</u>

How to Use the Dundas Upload Control

Overview | Properties | Methods

To use the Dundas Upload control the client must be using a browser which is capable of html form-based uploading (as per RFC 1867). Netscape 3.0+ and Microsoft's Internet Explorer 4.0+ both support this.

The POST html form MUST use an "EncType" of: "Multipart/Form-Data" (i.e. <form method="POST" name="SomeName" EncType="Multipart/Form-Data">.

Click <u>here</u> for list of steps to be followed when implementing the Upload control.

Files to be uploaded are determined through the use of file input boxes in the POST form.

The **Form** and **Files** collections of the control are especially important. The <u>Form</u> collection is used to access form data such as input elements, select elements, etc... . This collection also allows you to retrieve the values of multiple-entry form items (e.g. multi-select listbox). The <u>Files</u> collection (consisting of <u>UploadedFile</u> objects) is used to access and manipulate uploaded files.

To determine the file input box which corresponds to a particular uploaded file use the TagName property of the UploadedFile object (the TagName property is the same as the Name attribute of the corresponding file input box). You can iterate through all uploaded files with either a standard For loop or a For Each loop. To determine the file type that was uploaded use VBScript's InStr method with the ContentType property of an UploadedFile object. Note that the NextFile object also exposes the ContentType and TagName properties. Remember that calling the Save or SaveToMemory methods of the NextFile object will also update the Files collection.

To retrieve the values of form elements other than file input boxes use the control's **Form** collection (you can not retrieve form elements with the ASP Request.Form object when the encoding type of the form is "Multipart/Form-Data"). To retrieve particular form values use the name of the form element (e.g. strVariable = objUpload.Form("txtSomeTextbox")). You can also retrieve items by their numerical index. For example, to retrieve the value of the first form element you can use: strVariable = objUpload.Form(0).

Form data is either retrieved all at once or incrementally. To retrieve all of the form data with one method call you must use either the control's <u>Save</u> or <u>SaveToMemory</u> methods. The **Save** method saves all uploaded files to disk locally, while **SaveToMemory** saves uploaded files to memory. Irregardless of which method is used it is **VERY IMPORTANT** to know that either method populates the **Files** collection with ALL uploaded files as well as the control's **Form** collection with ALL form data.

You can retrieve form data incrementally by repeatedly calling the GetNextFile method (as opposed to using either Save or SaveToMemory). GetNextFile returns NextFile objects, with each **NextFile** object containing the header information for one uploaded file. The first **NextFile** object will correspond to the first file uploaded by the user, the next **NextFile** object (obtained by calling **GetNextFile** again) will contain the header information for the next populated file input box in the POST form, etc.... It is **IMPORTANT** to realize that each call to the **GetNextFile** method will populate the control's <u>Form</u> collection with all form data which appears **BEFORE** the form's populated file input box which corresponds to the current **NextFile** object. The **NextFile** object (like the control itself) exposes both **Save** and **SaveToMemory** methods. Use either of these methods while looping through all **NextFile** objects to optionally save uploaded files in their entirety. You can use whatever deciding criteria you want, but the <u>TagName</u> and <u>ContentType</u> properties in particular can be very useful when deciding whether or not to save files uploaded by the user (see the paragraph below). To skip uploading a file just call the GetNextFile method again without first calling the object's **Save** or **SaveToMemory** methods. Note that if you choose to retrieve the file in its entirety by calling either Save or SaveToMemory then the Files

collection of the control will be populated with the uploaded file in question.

If a form element was not populated by the user then there will be no corresponding **FormItem** object in the control's **Form** collection (e.g. if you loop through the Form collection with a For ... Each loop there will be no **FormItem** object for the form elements left empty by the user). If you retrieve the value of a particular form element which was left empty by the user the return will be "Empty". You can test to see if the user left a form element blank, or empty, by using VBScript's *IsEmpty* method.

To find out how to implement a progress bar (via the <u>ProgressBar</u> and <u>StateServer</u> Components) click here.

You can disable or preset certain features of the Upload control through the use of registry keys. Click here for more information.

NOTE: to utilize named constants in your ASP code use a server-side include and include the UploadControl.inc file (distributed with the application).

Most methods will throw an exception if an error occurs so make sure that you have enabled inline error trapping by using an **On Error Resume Next** statement. Examine VBScript's Err object after a function call to check for the success/failure of the operation.

The Dundas Upload Control 2.0 provides many methods for performing file operations, such as <u>FileCopy</u>, <u>FileMove</u>, etc... .

You can also temorarily assume a valid Windows account other than the default web account by calling the ImpersonateUser method. You might want to use this method if the default web account does not have the rights to perform a particular operation. Once you have finished using the impersonated account you can revert back to the default account by calling the ImpersonationTerminate method. Other methods of the control which are related to rights/permissions are: AllowAccess, DenyAccess, RevokeAccess and SetOwner.

Perform the following steps when utilizing the Upload control:

- 1. Make sure that the form which is POSTING the data is using an EncType of "Multipart/Form-Data". You will also need to use file input elements (<input type="file">) in this form so the user can browse for files to be uploaded.
- 2. Enable inline error trapping in the page being POSTED to with an "On Error Resume Next" statement. Most methods of the control will throw an exception if unsuccessful, so you can perform error trapping using VBScript's Err object.
- 3. Create an instance of the Upload control in the page which the form is POSTING to. To do this use the Server.CreateObject method with a ProgID of either "Dundas.Upload" or "Dundas.Upload.2".
- 4. You can either retrieve ALL form data (uploaded files as well as form input elements) at once by using the <u>Save</u> or <u>SaveToMemory</u> methods of the control. Alternatively you can retrieve form data incrementally by using the <u>GetNextFile</u> method. Please note that retrieving form data incrementally will result in the <u>Form</u> and **Files** collections also being populated incrementally.
- 5. Retrieve any form element values via the control's <u>Form</u> collection. Note that empty form elements WILL NOT be inserted into the **Form** collection, and attempting to retrieve the value of a form element which was left empty by the user will result in an exception being thrown (which you can trap for).
- 6. Manipulate files saved to either disk or memory via the <u>Files</u> collection. This collection is populated with either one **Save** or **SaveToMermory** call of the control, unlike the **Save** and **SaveToMemory** methods of the <u>NextFile</u> object (**NextFile** objects are returned by **GetNextFile** calls) which populates the **Files** collection one file at a time.
- 7. Set the Upload object to nothing.

See Also: Overview (Dundas Upload Control 2.0) | Tutorial 1: Uploading Multiple Files and Using the Form and Files Collections | Tutorial 2:

Retrieving Form Data Incrementally Using the GetNextFile Method | Tutorial 3: ADO Support and Saving a File as a BLOB | Tutorial 4: Implementing a Progress Bar

Disabling and Setting Features via the Registry

Overview | Properties | Methods

Some of the Upload Control's features (i.e. methods) can be disabled/enabled through the use of registry values in the "HKEY_LOCAL_MACHINE\SOFTWARE\Dundas Software\DundasUpload" key.

An exception will be thrown if a disabled method is called.

If a registry value does not exist or it is set to zero (0) then the method is enabled. However, setting a registry entry to any non-zero value will cause the corresponding method to be disabled.

You can also preset the value of *any* Upload control property to be readonly as well as having a specific value. To accomplish this just create a registry value (under the same key) with the exact same name as the relevant property and set it to the desired default. The type of the value depends on the data type of the property. If the property in question is a boolean or a numeric data type then the corresponding registry type should be a DWORD. A string property equates to a string registry data type. If the user attempts to change the value of a property which has been set to read-only via the registry then an exception will be raised.

Refer to the table below for a listing of possible registry entries and their corresponding methods:

Value Corresponding Method(s)

DisableImpersonation <u>ImpersonateUser</u>,

<u>ImpersonationTerminate</u>

DisableSecurity AllowAccess, DenyAccess,

RevokeAccess

DisableRegisterServer <u>RegisterServer</u>

DisableSend <u>SendBinary</u>

DisableDirectoryDelete <u>DirectoryDelete</u>

DisableDirectoryCreate <u>DirectoryCreate</u>

DisableFileCopy <u>FileCopy</u>

DisableFileMove <u>FileMove</u>

DisableFileDelete <u>FileDelete</u>

DisableMemoryUpload <u>SaveToMemory</u>

DisableProgressBar UploadProgress Component creation

NameOfRelevantProperty Default value. You can set the default

value of ANY Upload control property

by creating a key with the same name as the relevant property and then setting its value to the desired

default value.

QuickStart (Dundas Upload Control 2.0)

Overview | Properties | Methods

The following code demonstrates how to populate the Upload control's collections, save uploaded files to disk and retrieve form data. Please note that for brevity we do not perform any error checking (done with VBScript's Err object and raised exceptions).

We will assume that a form with an encoding type of "Multipart/Form-Data" is POSTING to the ASP page which contains the following code:

```
<%@ Language=VBScript %> <%
```

'most control methods throw an exception if an error occurs so we will

'On Error Resume Next statement

On Error Resume Next

'create instance of Upload control
Set objUpload = Server.CreateObject("Dundas.Upload.2")

'save uploaded files to to disk. Note that we could also save the 'uploaded files to memory. This populates the Form and Files collectic objUpload.Save "c:\temp\"

'retrieve the value of a form element called txtName strName = objUpload.Form("txtName")

'retrieve the size of the first uploaded file strSize = objUpload.Files(0).Size

'release resources
Set objUpload = Nothing
%>

Tutorial 1: Uploading Multiple Files and Using the Form and Files Collections

The objectives of this tutorial are to:

- Demonstrate how to save uploaded files and populate the <u>Files</u> and <u>Form collections</u>.
- Show how to access the items in the Files and Form collections.
- Demonstrate how to use <u>ContentType</u> property to check for the type of files which were uploaded.
- Show how to retrieve the name of the file input boxes responsible for the uploads by using the TagName property.

We are assuming that a form with an EncType of "Multipart/Form-Data" is POSTING data to the ASP page which contains the following code.

Example

<% 'most control methods throw an exception if an error occurs so we 'On Error Resume Next statement for error trapping purposes
On Error Resume Next

'create an instance of the Upload control
Set objUpload = Server.CreateObject ("Dundas.Upload.2")

'we will save all uploaded data (form data as well as any uploaded files 'note that we could also save the uploaded files to memory with the Sa' 'also note that both the Save and SaveToMemory methods populate the 'Files collections with one method call.

'its also important to realize that by default files saved to disk will have 'this can be changed with the UseUniqueNames property objUpload.Save "c:\temp\"

'check to see if method call was successful using VBScript's Err object 'an error occurred we will redirect user to a fictitious error page

If Err.Number <> 0 Then

Response.Redirect "Error.asp"

Else

'use a For Each loop and check to see if the uploaded file is an

'executable (utilizing VBScript's InStr method), if it is delete it from d 'but first we will output the name of the file input box(es) responsible f For Each objUploadedFile in objUpload.Files

Response.Write "The "" & objUploadedFile.TagName & "" If InStr(1,objUploadedFile.ContentType,"octet-stream") Then

'if the default web account does not have the right to delete files

' for the folder you save uploaded files to (in this case c:\temp) then yo' the required rights manually or you could use the ImpersonateUser mobjUploadedFile.Delete

End If

Next

'we will just output the name of the populated form elements and their For Each objFormItem In objUpload.Form

Response.Write "
br>The name of the form item is: " & objFormItem Response.Write "
br>The value of the form item is: " & objFormItem. Next

End If

'Release resources Set objUpload = Nothing %>

Tutorial 2: Retrieving Form Data Incrementally Using the GetNextFile Method

The objectives of this tutorial are to:

- Demonstrate how to optionally upload files using the <u>GetNextFile</u> method (retrieves form data incrementally).
- Show how the <u>Form</u> collection is populated by <u>GetNextFile</u> method calls.
- Demonstrate how to use the <u>TagName</u> property to check which uploaded files originated from which file input boxes.
- Demonstrate how to use the <u>ContentType</u> property to determine the type of file the user wishes to upload.

We are assuming that a form with an EncType of "Multipart/Form-Data" is POSTING data to an ASP page which contains the following code.

Example

<% 'most control methods throw an exception if an error occurs so we 'Resume Next statement for error trapping purposes

On Error Resume Next

'create an instance of the Upload control
Set objUpload = Server.CreateObject("Dundas.Upload.2")

'retrieve the first NextFile object, which contains header data only for 'the first file uploaded by user. Note that the Form collection will be 'populated with all form data which occurs up to the first populated 'file input box (which results in this first NextFile object)

Set objNextFile = objUpload.GetNextFile 'check to see if method call was successful using VBScript's Err object 'an error occurred we will redirect user to a fictitious error page

If Err.Number <> 0 Then Response.Redirect "Error.asp"

'we will retrieve NextFile objects until there are no more uploaded files 'to process (each NextFile object corresponds to a populated file input 'if the file is not an executable we will save the uploaded file to memor 'it is an executable then we will not save it at all (by just calling GetNe Do Until objNextFile Is Nothing

'NOTE: you can retrieve any form data here as long as it occurs in the BEFORE the populated file input box which corresponds to the current strSomestring = objUpload.Form("SomeFormElement")

'now save file if not an *.exe, and for demonstration purposes we will (
' the name of the file input box from which the uploaded file originated If InStr(1,objNextFile.ContentType,"octet-stream",1) = 0 Then objNextFile.SaveToMemory Response.Write objNextFile.TagName & "
br>" End If

'call NextFile again, to retrieve the header data for the next uploaded fi 'once again it should be noted that this will populate the Form collection 'with all data up to the corresponding populated file input box Set objNextFile = objUpload.GetNextFile

Loop

'release resources
Set objUpload = Nothing
%>

Tutorial 3: ADO Support and Saving a File as a BLOB

The objectives of this tutorial are to:

- Demonstrate how to work with an ADO recordset with a DSNless connection.
- Show how to save an uploaded file as a BLOB.

We are assuming that a form with an EncType of "Multipart/Form-Data" is POSTING data to the ASP page which contains the following code.

Assumptions

- 1. A SQL Server 7.0 database named "Temp" exists with a SQL username and password of "sa" and "", respectively.
- 2. This database has a table called "Test".
- 3. The table has an "image" column named "Picture".
- 4. The name of the server is "SOMESERVER".
- 5. The POST form uses one or more file input boxes.

Example

<% 'use inline error trapping so that any exceptions raised by the Uplo On Error Resume Next

'create an instance of the Uplaod control
Set objUpload = Server.CreateObject("Dundas.Upload.2")

'we will temporarily save uploaded files to disk objUpload.Save "c:\temp"

'set the connection string strConnect = "Driver={SQL Server};Server=SOMESERVER;Database

```
'create an ADO recordset
Set rs = Server.CreateObject("ADODB.Recordset")
'open a recordset using the connection string
rs.Open "Test", strConnect, 2, 3
'loop through all uploaded files and save to the database as BLOBs if
' an image was actually uploaded (we use the ContentType property of
' the UploadedFile object to accomplish this)
For Each objFile In objUpload.Files
    If InStr(1,objFile.ContentType,"image") <> 0 Then
        'add a new record and insert the image file
        rs.AddNew
        rs("Picture"). Value = objFile. Binary
    End If
Next
'save changes and close recordset
rs.Update
rs.Close
'release resources
Set rs = Nothing
Set objUpload = Nothing
%>
```

Tutorial 4: Implementing a Progress Bar

The objective of this tutorial is to:

• Demonstrate how to utilize the <u>Upload</u> control, <u>UploadProgress</u> component and the <u>State Server</u> in order to implement a progress bar.

To implement a progress bar you will need to have three elements: an html Input Form; a Progress Bar ASP page; and a Submit Action ASP page. These elements utilize the Dundas Upload control to gather the upload data and POST the upload operation details to the State Server. The State Server (StateServer.exe) is responsible for storing upload progress information which can then be updated and retrieved using an instance of either the UploadProgress component or the Upload control. The purpose of the Upload Progress component is to create unique Progress IDs, retrieve the progress information from the State Server and also delete information stored at the State Server when an upload is finished.

To use a progress bar you must perform the following actions for these three required items:

1. Input Form

- 1. Before submitting the form data the developer must create an instance of the Upload Progress component (the ProgID is "Dundas.UploadProgress) and request a new Progress ID.
- 2. Open the Progress Bar window and pass this Progress ID as a parameter.
- 3. Submit the form data, once again passing this Progress ID as a parameter to the Submit Action ASP page.

2. Progress Bar ASP

1. This page should be refreshed (e.g. every two seconds).

- The progress information is accessed by using an instance of the <u>Upload Progress</u> component along with the Progress ID passed to this page by the Input Form. To access the upload data set the <u>ProgressID</u> property of the Upload Progress instance to the passed Progress ID, call the <u>GetProgress</u> method and then utilize the <u>TotalSize</u>, <u>UploadedSize</u> and <u>PercentCompleted</u> properties.
- The state information must be deleted when the upload operation is either completed or an error occurs (an exception will be thrown). To delete the upload information at the State Server just call the <u>DeleteProgress</u> method of the Upload Progress instance.
- 4. The window may be closed when the upload reaches either 100% or an error occurs.

3. Submit Action ASP

 While uploading the data the Upload control (the ProgID is "Dundas.Upload.2") needs to post the progress status of the upload operation to the State Server. To accomplish this just set the <u>ProgressID</u> property of the control instance to the passed Progress ID.

How you implement these steps is totally up to you. If you require more detailed instructions on utilizing a progress bar then refer to the following or see the fully commented <u>sample code</u>.

NOTE: You **MUST USE** an <%@ENABLESESSIONSTATE = FALSE%> statement at the top of these ASP pages so that more than one ASP page will be processed at the same time.

Detailed Instructions

1. Create a main ASP page (we will call this page "main.asp"). This

page must:

- 1. Have a form with an EncType of "multipart/form-data". This form needs to have one or more file input boxes for the uploading of data.
- 2. POST the data to a secondary page (we will call this page "process.asp").
- 3. Create an instance of the **UploadProgress** component before POSTING the data (the ProgID is "Dundas.UploadProgress)." Once this is done you must then obtain a new Progress ID (which uniquely identifies each upload operation) by calling the <u>GetNewProgressID</u> method of the **UploadProgress** component.
- 4. Pass this Progress ID to process.asp when POSTING the data (how this is done is up to you). In the following example we accomplish this by using a standard form button (instead of a "submit" button) and we specify a client-side function for the **OnClick** event. In this function we use "document.form.action" and "document.form.submit" statements to submit the form data to process.asp (after opening the progressbar.asp window --> see the next step). Note that the "action statement" uses a querystring to pass the Process ID to process.asp.
- 5. Create the progress bar window (before POSTING the data). This window is an ASP page (let's call it "progressbar.asp") and once again you HAVE to let this page know what the Progress ID is (which you obtained from the UploadProgress component). We accomplish this by using a "window.open" statement in our form button's OnClick event. Note that this "window.open" statement uses a querystring to pass this Progress ID.
- 2. Create the page which will process the data (process.asp). This page must:
 - 1. Utilize the passed Progress ID and set the <u>ProgressID</u> property of the **Upload** control to this Progress ID BEFORE calling either the <u>Save</u>, <u>SaveToMemory</u> or <u>GetNextFile</u> method. Once this is

done you can proceed with the uploading of data. By setting this property the Upload control instance will continuously update the State Server with upload operation details, which can then be retrieved via the Progress Upload component in the progressbar.asp page.

- 3. Create the progress bar window via an ASP page ("progressbar.asp"). This page must:
 - 1. Be able to display the amount of data which has currently been uploaded. How you accomplish this is up to you. Note that the following sample code uses embedded tables with differing colors to accomplish this.
 - 2. Be refreshed, for example, every two (2) seconds. You can use a meta refresh tag for this.
 - 3. Retrieve details concerning the upload operation. This is accomplished by creating an instance of the **UploadProgress** component, setting its <u>ProgressID</u> property to the Progress ID passed by main.asp, calling the <u>GetProgress</u> method and then utilizing the <u>TotalSize</u>, <u>UploadedSize</u> and <u>PercentCompleted</u> properties. Every time the page is refreshed you can use these properties to update your "progess bar".
 - 4. If the State Server (StateServer.exe) is located on another machine compared to this ASP page then you must set the <u>StateServer</u> property of the <u>UploadProgress</u> and <u>Upload</u> objects to the IP address of the machine where the <u>StateServer</u> is located. If you have set the "Port" property of the <u>StateServer</u> to something other than the default then you must set the <u>StateServerPort</u> property of the <u>UploadProgress</u> and <u>Upload</u> objects to the same port.
 - 5. Delete the progress information once the upload operation is cancelled or completed. This is done by calling the DeleteProgress method of the **UploadProgress** component.
 - 6. You can close the window once the upload is completed or cancelled. How you do this is up to you. Note that the following

sample code accomplishes this through client-side script.

NOTE: You **MUST USE** an <%@ENABLESESSIONSTATE = FALSE%> statement at the top of these ASP pages so that more than one ASP page will be processed at the same time.

Assumptions for the following Sample Code

- 1. There are three pages to this sample: main.asp, progressbar.asp and process.asp (as described above).
- 2. The form in main.asp is POSTING data to process.asp with an EncType of "mulitpart/form-data". It is assumed that this form has one file input box named "file1".

main.asp

```
<%@ENABLESESSIONSTATE=FALSE%> 'turn off session state sup
' one ASP page can be processed at the same time.
<html><head></head>
<body color="black" bgcolor="white">
<%
dim objUploadProgress 'UploadProgress component instance
dim ProgressID 'new progress (state) id - uniquely identifies this uploa</pre>
```

'create an instance of the UploadProgress component set objUploadProgress = server.CreateObject("Dundas.UploadProgress

```
function Upload()
//this javascript function runs when the user clicks on the form's button
//it opens the progressbar.asp window and then submits the form data to
if (ProgressID !=-1){
//only open progressbar.asp window if there is a valid id. We will cente
Param = "SCROLLBARS=no, RESIZABLE=no, TOOLBAR=no, STA]
Param += ",TOP=" + String(window.screen.Height/2 - 50);
Param += ",LEFT=" + String(window.screen.Width/2 - 200);
//note that we pass the Progress ID to progressbar.asp
window.open("ProgressBar.asp?ProgressID=<%=ProgressID%>", null
}
//now that progress bar window is open submit the form data to Proces
// the ProgressID as a querystring parameter
document.frmMain.action = "Process.asp?ProgressID=<%=ProgressIE
document.frmMain.submit();
//-->
</Script>
<form name="frmMain" action="process.asp" enctype="multipart/forr</pre>
Please choose a file to be uploaded: <input type="file" name="File1">
<hr><hr><
<!--Note that we use a standard button and not a submit button. We do
we can open the progress bar window (Progress.asp) as well as pass the
data to the page which processes the submitted form data -->
<input type="button" name="submit1" value="Upload File" OnClick=
</form>
</body>
</html>
process.asp
<%@ENABLESESSIONSTATE=FALSE%>
```

<%'we just disabled session state so that more than one asp page can b

On Error Resume Next

'create an instance of the Upload component
Dim objUpload
Set objUpload = Server.CreateObject("Dundas.Upload.2")

'set the ProgressID property of the control to the state (progress) ID when 'was obtained in main.asp and passed as a querystring. By setting this the Upload control instance will continuously update the State Server 'information concerning this particular upload operation.

objUpload.ProgressID = Request.QueryString("ProgressID")

'set maximum file size to 2 MByte objUpload.MaxFileSize = 200000000

'save uploaded data to memory, and then check for errors objUpload.SaveToMemory

'if an error occurred we will close the progressbar.asp window by 'deleting the state information at the State Server

If Err.Number <> 0 Then dim objUploadProgress

set objUploadProgress = server.CreateObject("Dundas.UploadProgress objUploadProgress.ProgressID = Request.QueryString("ProgressID") objUploadProgress.DeleteProgress

End If

'release resources

Set objUpload = Nothing %>

progressbar.asp

<%@ENABLESESSIONSTATE=FALSE%> <html><head>

```
<meta http-equiv=refresh content="1,ProgressBar.asp?ProgressID=<% </head>
<%
```

On Error Resume Next

'force this page to expire immediately Response.Expires = -10000

'create an instance of the UploadProgress component Set objUploadProgress = Server.CreateObject("Dundas.UploadProgres

'if the State Server is running on a different machine compared to these 'then this is where you would have to set the State Server's IP address 'objUploadProgress.StateServer = "127.0.0.1"

'retrieve progress data from the StateServer objUploadProgress.ProgressID = Request.QueryString("ProgressID") objUploadProgress.GetProgress Percentage = objUploadProgress.PercentCompleted TotalSize = objUploadProgress.TotalSize UploadedSize = objUploadProgress.UploadedSize

'if there is no data yet for this particular upload operation then these pro' be set to negative one (-1)

If TotalSize = -1 OR UploadedSize = -1 Then

'if no data yet set to zero, since we do not want to display negative one TotalSize = 0

UploadedSize = 0

End If

'If an error occurs set the percentage value to -1. The client-side script will close the progress bar window when this occurs. Note that a negate be encountered since we delete the state information for this particula operation when 100% is reached, so the next call to the GetProgress necessary.

```
' result in an exception being raised, thereby resulting in this window b
If Err.number <> 0 Then
Percentage = -1
End If
'delete progress data when we hit 100%
If objUploadProgress.PercentCompleted = 100 Then
objUploadProgress.DeleteProgress
End If
'release resources
Set objUploadProgress = Nothing
%>
<script>
<!-- set a variable to the percent complete -->
var val = <%=Percentage%>;
// If there are any errors or upload is complete then close window
if(val == -1){
top.close();
}
</script>
<body>
<!-- use embedded tables with different color to simulate a progress ba
%" bgcolor="
>
```

```
</body>
<P align="center"><FONT size="2">
<%=Percentage%>% (Uploaded <%=UploadedSize%> of <%=TotalSi</FONT></P>
</html>
```

Properties (Dundas Upload Control 2.0)

Overview | Properties | Methods

CheckMacBinary You can set this property to TRUE before uploading

so that each uploaded file is checked for a MacBinary

format.

MaxFileCount The maximum number of files which can be uploaded.

MaxFileSize The maximum size of any file uploaded by the user (in

bytes).

MaxUploadSize The maximum amount of data that can be uploaded

to the server (in bytes). This includes form data as

well as uploaded files.

<u>ProgressID</u> Set this property to a unique ID (obtained by the

<u>UploadProgress</u> component from the <u>Dundas State</u> <u>Server</u>) when processing an upload which utilizes a

progress bar window.

StateServer Set this property to the IP address of the <u>Dundas</u>

State Server when processing an upload which

utilizes a progress bar window.

<u>StateServerPort</u> Set this property to the port that the <u>State Server</u> is

listening on when processing an upload which utilizes

a progress bar window.

<u>UseUniqueNames</u> Stores uploaded files with unique filenames. This is

done by preceding the original filename with a GUID

and "_".

<u>UseVirtualDir</u> Set this property to TRUE so that virtual directories

are used. If TRUE then any method which utilizes a

Path argument should use a virtual pathname and not

a physical pathname.

Collection Properties

Files Collection of UploadedFile objects, each of which

represents a file uploaded from an html form.

<u>Form</u> Collection of <u>FormItem</u> objects, with each object

corresponding to a form element. Note that input elements of the file type (<input type="file">) are not

represented in the Form collection.

CheckMacBinary (Dundas Upload Control 2.0)

Overview | Properties | Methods

Set this property to TRUE before uploading so that each uploaded file is checked for a MacBinary format.

Syntax

UploadObject.CheckMacBinary = [boolean]

The **CheckMacBinary** property syntax has the following parts:

Part	Description
	If set to TRUE (before uploading) then each uploaded file will be checked for a MacBinary format.

Remarks

Client machines which are Macintosh may utilize a protocol called "MacBinary".

MacBinary formatted files consist of a 128-byte header, a data fork and a resource fork. If you set this property to TRUE and a MacBinary file is encountered then only the data fork portion of the uploaded file will be saved.

MaxFileCount (Dundas Upload Control 2.0)

Overview | Properties | Methods

The maximum number of files which can be uploaded to the server.

Syntax

UploadObject.MaxFileCount = [long]

The **MaxFileCount** property syntax has the following parts:

Part	Description
	The maximum number of files which can be uploaded to the server. Defaults to negative one (no limit).

Remarks

Set this property before calling <u>Save</u> or <u>SaveToMemory</u> in order to limit the number of files a user is allowed to upload. If this limit is exceeded and you call either the control's or the <u>NextFile</u> object's **Save** or **SaveToMemory** methods an exception will be raised and the save operation will be cancelled.

Note that if the maximum number of allowable uploads is exceeded as a result of a **NextFile** object's save operation then any previous uploaded data will still be available from the control's Form collection.

Make sure that you use an **On Error Resume Next** statement so that any raised exceptions are handled. To retrieve a relevant error string use VBScript's Err.Description property.

Set this property to negative one (-1) for no limit.

See Also: MaxFileSize | MaxUploadSize

MaxFileSize (Dundas Upload Control 2.0)

Overview | Properties | Methods

The maximum allowable file size in bytes.

Syntax

UploadObject.MaxFileSize = [long]

The **MaxFileSize** property syntax has the following parts:

Part	Description
	The maximum allowable file size in bytes. Defaults to negative one (no limit).

Remarks

Set this property before calling <u>Save</u> or <u>SaveToMemory</u> in order to limit the size of files uploaded by a user. If you call either the control's or the <u>NextFile</u> object's **Save** or **SaveToMemory** methods and this limit is exceeded then an exception will be raised.

Note that if a file is encountered which exceeds this limit then no further form data will be processed. However, you can still access any form data processed previously through the control's <u>Form</u> collection.

Set this property to negative one (-1) for no limit.

Make sure that you use an **On Error Resume Next** statement so that any raised exceptions are handled in your ASP page. To retrieve a relevant error string use VBScript's Err.Description property.

See Also: MaxFileCount | MaxUploadSize

MaxUploadSize (Dundas Upload Control 2.0)

Overview | Properties | Methods

The maximum amount of data which can be uploaded to the server.

Syntax

UploadObject.MaxUploadSize = [long]

The **MaxUploadSize** property syntax has the following parts:

Part	Description
_	The maximum amount of data which can be uploaded to the server (in bytes). Defaults to negative one (no limit).

Remarks

Uploaded data consists of all uploaded files as well as any form data (as a result of a POST operation).

If you do not set this property then the default of negative one (-1) is used and no limit is set.

Calling the <u>Save</u> or <u>SaveToMemory</u> methods of either the control or the <u>NextFile</u> object will result in an exception being thrown if the uploaded data exceeds the maximum allowable limit. As a result no data will be saved and the control's collections will not be populated.

Unlike the **MaxFileCount** or the **MaxFileSize** properties if this allowable data limit is exceeded then there will be no available data via the control's <u>Form</u> collection irregardless of when the exception is thrown.

Set this property to negative one (-1) for no limit to the uploaded data.

See Also: MaxFileCount | MaxFileSize

ProgressID (Dundas Upload Control 2.0)

Overview | Properties | Methods

Set this property to a unique ID (obtained by the <u>UploadProgress</u> component from the <u>Dundas State Server</u>) when processing an upload which utilizes a progress bar window.

Syntax

long = UploadObject.ProgressID

The **ProgressID** property syntax has the following parts:

Part	Description
long	An ID which uniquely identifies one particular upload operation. Defaults to negative one (disabled).

Remarks

You must set this property before calling either the <u>Save</u> or <u>SaveToMemory</u> methods of the control if you want to utilize the **StateServer** and the **UploadProgress** component in order to display a progress bar.

Setting this property allows the **State Server** component's data for the upload operation in question to be continuously updated by the **Upload** Control. Inotherwords, as the uploading of form data occurs the **State Server** is continuously being updated as to how much data has been uploaded, thereby allowing the **UploadProgress** component to retrieve this information in order to update a progress bar window.

A unique ID is generated by calling the <u>GetNewProgressID</u> method of the <u>UploadProgress</u> component.

See Also: Tutorial 4: Implementing a Progress Bar | StateServer |

<u>StateServerPort</u>

StateServer (Dundas Upload Control 2.0)

Overview | Properties | Methods

Set this property to the IP address of the <u>Dundas State Server</u> when processing an upload which utilizes a progress bar window.

Syntax

UploadObject.StateServer = *string*

The **StateServer** property syntax has the following parts:

Part	Description
string	The IP address of the machine which is running the Dundas State Server. Defaults to the local loopback address (127.0.0.1).

Remarks

You must set this property before calling either the <u>Save</u> or <u>SaveToMemory</u> methods of the **Upload** control if: you want to utilize the **StateServer** and the **UploadProgress** component in order to display a progress bar; and the State Server is running at a different machine compared to where the ASP page(s) in question are located.

You do not have to set this property if the State Server is running on the same machine which the ASP page in question is running on since the local loopback address is the default.

See Also: <u>Tutorial 4: Implementing a Progress Bar</u> | <u>StateServerPort</u> | <u>ProgressID</u>

StateServerPort (Dundas Upload Control 2.0)

Overview | Properties | Methods

Set this property to the port that the <u>State Server</u> is listening on when processing an upload which utilizes a progress bar window.

Syntax

UploadObject.StateServerPort = *long*

The **StateServerPort** property syntax has the following parts:

Part	Description
	The port which the Dundas State Server is listening on. Defaults to 6723, which is also the default port that the State Server is set to.

Remarks

You must set this property before calling either the <u>Save</u> or <u>SaveToMemory</u> methods of the **Upload** control if: you want to utilize the **StateServer** and the <u>UploadProgress</u> component in order to display a progress bar and; the port that the **State Server** is listening on was changed from the default (which is also port 6723).

See Also: <u>Tutorial 4: Implementing a Progress Bar</u> | <u>StateServer</u> | <u>ProgressID</u>

UseUniqueNames (Dundas Upload Control 2.0)

Overview | Properties | Methods

A boolean property which determines if uploaded files are saved to disk with unique filenames.

Syntax

UploadObject.UseUniqueNames = [boolean]

The **UseUniqueNames** property syntax has the following parts:

Part	Description
	If this property is set to TRUE (the default) then files will be saved to disk with unique filenames. If it is set to FALSE then files will be saved to disk using their original filenames.

Remarks

If this is TRUE (the default) then the resulting filename will have the following format: GUID OriginalFilename.

We highly recommend that you save files with unique filenames if you are saving to disk. This ensures that two users who upload a file with the same name at the same time does not cause problems.

See Also: Save

UseVirtualDir (Dundas Upload Control 2.0)

Overview | Properties | Methods

A boolean property which determines if file save, copy or move operations should use physical or virtual directories/paths.

Syntax

UploadObject.UseVirtualDir = [boolean]

The **UseVirtualDir** property syntax has the following parts:

Part	Description
boolean	If this property is set to TRUE then you must specify a virtual directory if you are copying, moving or saving files to disk. If it is set to FALSE (the default) then you must specify a physical directory/path.

Remarks

If this property is TRUE then make sure that the virtual path specified in a <u>Save</u> method call **begins with a forward slash**. The first directory specified after the forward slash must be preceded with a forward slash, and any subsequent sub-directories (physical folders located beneath the virtual directory) are separated with forward slashes.

You will also have to specify virtual paths when calling any other methods which take a path argument (e.g. <u>Move</u>, <u>Copy</u>, etc.).

Use the following format for a relative path argument in a function call: *objUpload.Save* "/MyVirtualDir/ASubDirectory".

See Also: Save

Methods (Dundas Upload Control 2.0)

Overview | Properties | Methods

<u>DirectoryCreate</u> Creates a directory.

<u>DirectoryDelete</u> Deletes a directory.

<u>DirectoryExists</u> Checks to see if the specified directory exists.

<u>FileCopy</u> Copy a file from one location to another.

<u>FileDelete</u> Deletes a specified file.

<u>FileExists</u> Checks to see if the specified file actaully exists.

FileMove Moves a file from one location to another.

GetFileDirName Call this method to extract the directory name from a

fully specified pathname.

GetFileExt Call this method to extract the file extension from a

fully specified pathname.

GetFileName Call this method to extract the filename from a

fully specified pathname.

GetNextFile Used to retrieve form data incrementally, this

method returns **NextFile** objects.

GetUniqueName This method returns a GUID (useful for saving

files with unique names).

<u>ImpersonateUser</u> Call this method to temporarily utilize a

Windows account other than the default (usually the IUSR account).

ImpersonationTerminate Call this method to revert back to the default

user account after having called the

ImpersonateUser method.

RegisterServer Call this method to register a COM component.

Save Saves uploaded files to disk and populates the

Upload control's Form and Files collections.

SaveToMemory Saves all uploaded form data (uploaded files as

well as form element values) to memory. This method also populates the <u>Files</u> and <u>Form</u>

collections.

SendBinary Call this method to download files from the

server.

DirectoryCreate (Dundas Upload Control 2.0)

Overview | Properties | Methods

Call this method to create a folder.

Syntax

UploadObject.DirectoryCreate(*Path* As String, [*MustNotExist* As Boolean = FALSE])

The **DirectoryCreate** method syntax has the following parts:

Part	Description
Path	The name of the folder to be created. Either a virtual or physical pathname must be used, depending on the UseVirtualDir property setting.
	A boolean which specifies whether or not the directory must not exist.

Remarks

An exception is thrown if the operation fails. Trap for success/failure by examining VBScript's Err object immediately after calling this method (the Number property of the Err object will be a non-zero value if it failed). MAKE SURE that you have enabled inline error trapping by using an **On Error Resume Next** statement at the beginning of the ASP page.

Note that an exception is also thrown if you set the *MustNotExist* argument to TRUE and the folder already exists. The resulting error string will be "Can not create a file when that file already exists."

If **UseVirtualDir** is set to TRUE then the specified path needs to be relative. It should start with a forward slash and be followed by a valid

alias. Any following folders need to be separated with forward slashes.

If the default user account does not have permission to create a folder then an "Access Denied" error will occur. You can use the ImpersonateUser method to temporarily assume another user account which has the permission to create a folder and then call ImpersonationTerminate to resume using the default account.

See Also: <u>DirectoryDelete</u> | <u>DirectoryExists</u>

DirectoryDelete (Dundas Upload Control 2.0)

Overview | Properties | Methods

Call this method to delete a folder.

Syntax

UploadObject.DirectoryDelete(Path As String[, DeleteContent As Bool = FALSE])

The **DirectoryDelete** method syntax has the following parts:

Part	Description
Path	The name of the folder to be deleted. Either a virtual or physical pathname must be used, depending on the UseVirtualDir property setting.
	If set to TRUE then the folder and all of its contents will be deleted.

Remarks

An exception is thrown if the operation fails (the Number property of the Err object will be a non-zero value upon failure). MAKE SURE that you have enabled inline error trapping by using an **On Error Resume Next** statement at the beginning of the ASP page.

If the default user account does not have permission to delete a folder then an "Access Denied" error will occur. You can use the ImpersonateUser method to temporarily assume another user account which has the permission to delete a folder and then call ImpersonationTerminate to resume using the default account. For more details on deleting uploaded files at the server refer to the UploadedFileObject topic.

An exception is raised if the specified folder does not exist, and the resulting error string is "The system can not find the file specified". Trying to delete a folder which is not empty and *DeleteContent* = FALSE also results in an exception, and the resulting error string from this situation is "The directory is not empty".

If **UseVirtualDir** is set to TRUE then the specified path needs to be relative. It should start with a forward slash and be followed by a valid alias, and any sub-folders need to be separated with forward slashes. If you specify a physical pathname when **UseVirtualDir** is TRUE an exception will be thrown.

See Also: <u>DirectoryCreate</u> | <u>DirectoryExists</u>

DirectoryExists (Dundas Upload Control 2.0)

Overview | Properties | Methods

Call this method to check if the specified folder already exists on the server.

Syntax

boolean = UploadObject.DirectoryExists (Path as String)

Part	Description
Path	The pathname of the folder to be checked.

Remarks

Returns TRUE if the folder is found on the server, otherwise FALSE is returned.

Note that the type of path specified depends on the <u>UseVirtualDir</u> property setting. If this property is FALSE then a physical pathname is required. If **UseVirtualDir** is set to TRUE then the specified path needs to be relative. It should start with a forward slash and be followed by a valid alias, and any sub-folders need to be separated with forward slashes.

See Also: <u>DirectoryCreate</u> | <u>DirectoryDelete</u>

FileCopy (Dundas Upload Control 2.0)

Overview | Properties | Methods

Call this method to copy a file from a specified source to a specified destination at the server.

Syntax

UploadObject.FileCopy (*Source* As String, *Destination* As String[, *FailIfExists* As Boolean = TRUE]

The **FileCopy** method syntax has the following parts:

Part	Description
Source	The name of the file to be copied. Use either a relative or absolute pathname, depending on the setting of the UseVirtualDir property.
Destination	The name of the folder to copy the file to. Either a relative or absolute pathname, depending on the setting of the UseVirtualDir property.
FailIfExits	If TRUE then an exception will be thrown if the file already exists at the specified destination. If this is FALSE an existing file will be overwritten. Defaults to TRUE.

Remarks

An exception is thrown if the operation fails. Trap for success/failure by examining VBScript's Err object immediately after calling this method (the Number property of the Err object will be a non-zero value if it failed). MAKE SURE that you have enabled inline error trapping by using an **On Error**

Resume Next statement at the beginning of the ASP page.

If **UseVirtualDir** is set to TRUE then the specified path needs to be relative. It should start with a forward slash and be followed by a valid alias, and any sub-folders need to be separated with forward slashes. If you specify a physical pathname when **UseVirtualDir** is TRUE an exception will be thrown.

If the default user account does not have the appropriate permission to perform this action then an "Access Denied" error will occur. You can use the ImpersonateUser method to temporarily assume another user account which has the appropriate permission, perform the operation and then call ImpersonationTerminate to resume using the default account.

See Also: FileDelete | FileExists | FileMove

FileDelete (Dundas Upload Control 2.0)

Overview | Properties | Methods

Call this method to delete a file.

Syntax

UploadObject.FileDelete (Path As String)

The **FileDelete** method syntax has the following parts:

Part	Description	
	Location of the file to be deleted. Use either a relative or absolute pathname, depending on the setting of the UseVirtualDir property.	

Remarks

An exception is thrown if the operation fails. Trap for success/failure by examining VBScript's Err object immediately after calling this method (the Number property of the Err object will be a non-zero value if it failed). MAKE SURE that you have enabled inline error trapping by using an **On Error Resume Next** statement at the beginning of the ASP page.

If **UseVirtualDir** is set to TRUE then the specified path needs to be relative. It should start with a forward slash and be followed by a valid alias, and any sub-folders need to be separated with forward slashes. If you specify a physical pathname when **UseVirtualDir** is TRUE an exception will be thrown.

If the default user account does not have the appropriate permission to perform this action then an "Access Denied" error will occur. You can use the ImpersonateUser method to temporarily assume another user account which has the appropriate permission, perform the operation and

then call ImpersonationTerminate to resume using the default account.

See Also: FileCopy | FileExists | FileMove

FileExists (Dundas Upload Control 2.0)

Overview | Properties | Methods

Call this method to see if the specified file exists at the specified location.

Syntax

UploadObject.FileExists(Path As String) As Boolean

The **FileExists** method syntax has the following parts:

Part	Description
	The name of the file to be checked. Use either a relative or absolute pathname, depending on the setting of the UseVirtualDir property.

Remarks

This method returns TRUE if the file is found, otherwise FALSE is returned.

If **UseVirtualDir** is set to TRUE then the specified path needs to be relative. It should start with a forward slash and be followed by a valid alias, and any sub-folders need to be separated with forward slashes.

See Also: FileCopy | FileDelete | FileMove

FileMove (Dundas Upload Control 2.0)

Overview | Properties | Methods

Call this method to move a file from one location to another.

Syntax

UploadObject.FileMove(*Source* As String, *Destination* As String[, FailIfExsists *As Boolean = TRUE*])

The **FileMove** method syntax has the following parts:

Part	Description
Source	Name of the file to be moved. Use either a relative or absolute pathname, depending on the setting of the UseVirtualDir property.
Destination	Destination location. Use either a relative or absolute pathname, depending on the setting of the <u>UseVirtualDir</u> property.
FaillfExists	If TRUE an exception will be thrown if the file already exists at the specified destination. If this is FALSE then an existing file with the same name will be overwritten. Defaults to TRUE.

Remarks

An exception is thrown if the operation fails. Trap for success/failure by examining VBScript's Err object immediately after calling this method (the Number property of the Err object will be a non-zero value if it failed). MAKE SURE that you have enabled inline error trapping by using an **On Error Resume Next** statement at the beginning of the ASP page.

This method can be used to rename a file by specifying the same directory path with differing filenames for the *Source* and *Destination* arguments.

If **UseVirtualDir** is set to TRUE then the specified path needs to be relative. It should start with a forward slash and be followed by a valid alias, and any sub-folders need to be separated with forward slashes. If you specify a physical pathname when **UseVirtualDir** is TRUE an exception will be thrown.

If the default user account does not have the appropriate permission to perform this action then an "Access Denied" error will occur. You can use the ImpersonateUser method to temporarily assume another user account which has the appropriate permission, perform the operation and then call ImpersonationTerminate to resume using the default account.

See Also: FileCopy | FileExists | FileDelete

GetFileDirName (Dundas Upload Control 2.0)

Overview | Properties | Methods

Call this method to extract the directory name from a fully specified path.

Syntax

UploadObject.GetFileDirName(Path As String) As String

The **GetFileDirName** method syntax has the following part(s):

Part	Description
Path	Full path and filename.

Remarks

This method truncates the filename from a fully specified pathname. Note that this method does not check for the existence of the specified directory or file.

An exception IS NOT RAISED if the specified file in the pathname does not exist.

See Also: GetFileExt | GetFileName

GetFileExt (Dundas Upload Control 2.0)

Overview | Properties | Methods

Call this method to extract the file extension from a fully specified path.

Syntax

UploadObject.GetFileExt(Path As String) As String

The **GetFileExt** method syntax has the following part(s):

Part	Description
Path	Full path and filename with a file extension.

Remarks

This method simply truncates and returns the extension from a fully specified path. Note that this method does not check for the existence of the directory nor the file.

An exception IS NOT RAISED if the specified file in the pathname does not exist.

See Also: GetFileDirName | GetFileName

GetFileName (Dundas Upload Control 2.0)

Overview | Properties | Methods

Call this method to extract the filename from a fully specified path.

Syntax

UploadObject.GetFileName(Path As String) As String

The **GetFileName** method syntax has the following part(s):

Part	Description
Path	Full path and filename with extension

Remarks

This method simply truncates the filename (with the extension) from a full pathname. Note that this method does not check for the existence of the directory nor the file.

This method is very useful if you are looping through the <u>Files</u> collection and want to copy or move files saved to disk to another location, possibly depending on some sort of criteria concerning the uploaded file. Use **GetFileName** with the <u>UploadedFile</u> object's <u>Path</u> property as the *Path* argument so that you can copy the file(s) to another location using the same filename. See the <u>Copy</u> method for sample source code illustrating this.

An exception IS NOT RAISED if the specified file in the pathname does not exist.

See Also: GetFileDirName | GetFileExt

GetNextFile (Dundas Upload Control 2.0)

Overview | Properties | Methods

Call this method to retrieve the headers for uploaded files one at a time.

Syntax

Set NextFileObject = UploadObject.GetNextFile()

Remarks

This method reads an HTML form in a top-down manner and returns a NextFile object when data from a populated file input box is encountered. Any form data (e.g. textbox data) prior to the populated file input box is uploaded to the Form collection. "Nothing" is returned if no populated file input box is found.

Please note that if the user did not specify any files to be uploaded then calling this method for the first time will result in the Upload control's **Form** collection to be populated with ALL values entered by the user into the form's elements.

To loop through all uploaded files use a "Do Until" loop, and test to see if the returned **NextFile** object is "Nothing". See the sample code below for an example of this.

Only the header for the file is uploaded, and you can programmatically decide if you want to retrieve the file in its entirety, depending on some sort of criteria (e.g. if the file is an image). If you want to retrieve the entire file then call either the Save method or the SaveToMemory method of the NextFile again. If you call GetNextFile again WITHOUT calling SaveToMemory then the file in question will NOT be uploaded.

If you call **Save** or **SaveToMemory** then the uploaded file will also have a corresponding <u>UploadedFile</u> object appended to the control's <u>Files</u> collection.

The advantage of this method is that you are not forced to retrieve all form data at once. The ability to accept or reject uploads is also advantageous to the developer.

See Also: NextFile Object | Tutorial 2: Retrieving Form Data Incrementally Using the GetNextFile Method

Example

'Note: it is assumed that a form (which contains one or more file input

'create an instance of the Upload control Set objUpload = Server.Create

'retrieve the header for the first uploaded file Set objNextFile = objUpload.GetNextFile()

Do Until objNextFile Is Nothing

'loop through all uploaded files and retrieve the uploaded file if it is an

If InStr(1,objNextFile.contenttype,"Image",1) Then objNextFile.SaveToMemory()

End If

'retrieve header for next uploaded file
Set objNextFile = objUpload.GetNextFile()

Loop

'destroy object Set Upload = Nothing

GetUniqueName (Dundas Upload Control 2.0)

Overview | Properties | Methods

This method returns a unique name (a GUID).

Syntax

string = UploadObject.GetUniqueName()

Remarks

Call this method to retrieve a GUID (a Globally Unique Identifier). This can be useful when saving files with unique filenames.

Note that you can also utilize unique filenames by setting the <u>UseUniqueNames</u> property to TRUE. If the **UseUniqueNames** property is set to TRUE (the default) then all files will be saved with a GUID and an "_" preceding the original filename.

See Also: <u>UseUniqueNames</u>

ImpersonateUser (Dundas Upload Control 2.0)

Overview | Properties | Methods

Call this method to switch from the default user account to another account with different permissions.

Syntax

UploadObject.ImpersonateUser(*UserName* As String, *Password* As String, [*Domain* As String], [*Type* As Long = 2])

The **ImpersonateUser** method syntax has the following parts:

Part	Description
UserName	The username of the Windows account to use.
Password	The password of the Windows account to use.
	The domain of the Windows account to use. If not specified the domain of the current default account is used.
Туре	Specifies the type of logon operation to perform.

Remarks

An exception is thrown if this operation fails. You can trap for success/failure by examining VBScript's Err object immediately after calling this method (the Number property of the Err object will be a non-zero value if it failed). MAKE SURE that you have enabled inline error trapping by using an **On Error Resume Next** statement at the beginning of the ASP page.

This method is useful if you need to perform an operation for which the

default user account (usually the IUSR account) does not have the appropriate permissions. For example, you can call **ImpersonateUser** to use an account which has the rights to delete, move or copy files, and when the file operation is complete you can then call ImpersonationTerminate to revert back to the default account.

IMPORTANT: If your virtual web site directory has "Run in separate memory space" checked then in order to use **ImpersonateUser** you **MUST** give the default account "Part of the operating system" priviledges. See below for instructions on how to do this.

The *Type* argument determines how the logon operation is to be performed (you can use the distributed include file named UploadControl.inc to incorporate named constants in your ASP source code). The possible values for this are:

- LOGON_INTERACTIVE (2).
- LOGON_NETWORK (3).
- LOGON_BATCH (4).
- LOGON_SERVICE (5).

For more details concerning the logon type refer to your MSDN under the "LogonUser" topic.

Granting Operating System Priviledges in Windows NT:

- 1. Open "Start Menu | Administrative Tools | User Manager".
- 2. Then open "Policies | User Rights" from the menu, MAKING SURE that the "Show Advanced User Rights" box is checked.
- 3. From the dropdown listbox select "Act as Part of the Operating System".
- 4. Add the default user account to the list of accounts which have this priviledge by clicking on the ADD button. If the default account is not

listed then add it by clicking on "Show Users" and selecting the default account and then clicking the "Add" and "OK" buttons.

5. Reboot.

Granting Operating System Priviledges in Windows 2000:

- 1. Open "Administrative Tools | Local Security Policy" and then doubleclick on "Local Policies".
- 2. Open "User Rights Assignment" and then double-click on "Act as Part of the Operating System".
- 3. Add the default (i.e. IUSR) account to the current list. If the default account is not listed follow Step 4 from above.

See Also: ImpersonationTerminate

ImpersonationTerminate (Dundas Upload Control 2.0)

Overview | Properties | Methods

Call this method to revert back to using the default user account.

Syntax

UploadObject.ImpersonationTerminate()

Remarks

An exception is thrown if the operation fails. You can trap for success/failure by examining VBScript's Err object immediately after calling this method (the Number property of the Err object will be a non-zero value if it failed). MAKE SURE that you have enabled inline error trapping by using an **On Error Resume Next** statement at the beginning of the ASP page.

If you need to perform an operation for which the default user account does not have permission then call the ImpersonateUser method to utilize an account which has wider permissions than the default account (which is usually the IUSR account). Once you no longer need these wider permissions call the ImpersonationTerminate method to revert back to using the default account.

See Also: ImpersonateUser

RegisterServer (Dundas Upload Control 2.0)

Overview | Properties | Methods

Use this method to register/unregister a (COM) component at the server.

Syntax

UploadObject.RegisterServer(*Path* As String[, *Unregister* As Boolean = False])

The **RegisterServer** method syntax has the following part(s):

Part	Description
	Pathname of the component to be registered. Use either a relative or absolute pathname, depending on the setting of the <u>UseVirtualDir</u> property.
Unregister	Set this boolean to FALSE to register the component (the default). Set it to TRUE to unregister the component.

Remarks

This method will raise an exception if the specified file does not exist. To make sure that the file in question exists you can use the <u>FileExists</u> method before calling **RegisterServer**.

If **UseVirtualDir** is set to TRUE then the specified path needs to be relative. It should start with a forward slash and be followed by a valid alias, and any sub-folders need to be separated with forward slashes. If you specify a physical pathname when **UseVirtualDir** is TRUE an

exception will be thrown.

See Also: FileExists

Save (Upload Control and NextFile Object)

Overview | Properties | Methods

Call this method to save uploaded files to disk and populate the control's collections.

Syntax

Object.Save(Path As string)

The **Save** method syntax has the following parts:

Part	Description
Path	A string expression specifying the folder path to save the uploaded files in. Use either a relative or absolute pathname, depending on the setting of the <u>UseVirtualDir</u> property.

Remarks

Call this method to save uploaded files to disk. If the <u>UseUniqueNames</u> property is set to TRUE (the default) then all files will be saved with a GUID and an "_" preceding the original filename. For example, if the name of an uploaded file is "index.txt" then the corresponding file saved to disk will be "SomeGUID_index.txt". We highly recommend that you utilize unique names for saved files.

You **MUST EITHER CALL THIS METHOD**, the <u>SaveToMemory</u> method or the <u>GetNextFile</u> method before attempting to use either the <u>Form</u> or <u>Files</u> collections. Not only do these methods retrieve uploaded files but they also populate the Upload control's collections. A trappable error will occur if you try to utilize these collections before calling either of these methods.

Calling a save method will throw an exception if the uploaded data

exceeds the maximum allowable limit, as set by the <u>MaxUploadSize</u> property. As a result no files will be saved and the control's collections will not be populated.

An exception will also be thrown if the specified path does not exist.

If **UseVirtualDir** is set to TRUE then the specified path needs to be relative. It should start with a forward slash and be followed by a valid alias, and any sub-folders need to be separated with forward slashes. If you specify a physical pathname when **UseVirtualDir** is TRUE an exception will be thrown.

NOTE: If you do not want to automatically retrieve all downloaded files and form data then alternatively you can use the **GetNextFile** method to retrieve the form data.

See Also: <u>UseUniqueNames</u> | <u>MaxUploadSize</u> | <u>SaveToMemory</u> | <u>GetNextFile</u> | <u>Tutorial 1: Uploading Multiple Files and Using the Form and Files Collections</u> | <u>Tutorial 2: Retrieving Form Data Incrementally Using the GetNextFile Method</u>

SaveToMemory (Upload Control and NextFile Object)

Call this method to save uploaded files to memory and populate both the Files and Form collections.

Syntax

Object.SaveToMemory()

Remarks

You **MUST EITHER CALL THIS METHOD** or the **Save** method before attempting to use either the <u>Form</u> or <u>Files</u> collections. Not only do these methods retrieve uploaded files but they also populate the Upload control's collections. A trappable error will occur if you try to utilize these collections before calling one of these methods.

If files are saved to memory then calling any of the following methods will result in an exception being thrown since there will be no files saved to disk: FileCopy, FileDelete, FileMove, SetAttributes, GetAttributes and the SetOwner methods of the UploadedFile object.

Calling the **SaveToMemory** method will also throw an exception if the uploaded data exceeds the maximum allowable limit, as set by the <u>MaxUploadSize</u> property. As a result no form data will be retrieved.

Note that calling the control's **SaveToMemory** method populates the <u>Files</u> and <u>Form</u> collections with one method call, while calling the **SaveToMemory** method of a <u>NextFile</u> object populates the **Form** and **Files** collections incrementally.

See Also: Save | Tutorial 1: Uploading Multiple Files and Using the Form and Files Collections | Tutorial 2: Retrieving Form Data Incrementally Using the GetNextFile Method

SendBinary (Dundas Upload Control 2.0)

Overview | Properties | Methods

Call this method to download files from the server.

Syntax

UploadObject.SendBinary(Path As String[, ContentType As String = "default"])

The **SendBinary** method syntax has the following part(s):

Part	Description
Path	The pathname of the file to be downloaded. Use either a relative or absolute pathname, depending on the setting of the UseVirtualDir property.
ContentType	The type of file to be downloaded.

Remarks

An exception is thrown if the operation fails. Trap for success/failure by examining VBScript's Err object immediately after calling this method (the Number property of the Err object will be a non-zero value if it failed). MAKE SURE that you have enabled inline error trapping by using an **On Error Resume Next** statement at the beginning of the ASP page.

If **UseVirtualDir** is set to TRUE then the specified path needs to be relative. It should start with a forward slash and be followed by a valid alias, and any sub-folders need to be separated with forward slashes. If you specify a physical pathname when **UseVirtualDir** is TRUE an exception will be thrown.

If you specify an empty content type the resulting Response.ContentType header is not set. If it is set to the default (which is "default") then the resulting content type header will be set to a value which corresponds to the associated file extension.

Please note that it is much more efficient to use the <u>SaveToMemory</u> method when calling **SendBinary**.

The data sent back to the client will constitute the contents of the resulting web page displayed in the client's browser.

Files Collection (Dundas Upload Control 2.0)

Overview | Properties | Methods

The **Files** collection stores <u>UploadedFile</u> objects which are a result of files uploaded to the server.

Remarks

The **Files** collection stores one (1) <u>UploadedFile</u> object for every file uploaded to the server as a result of file input inboxes (e.g. <input type="file" name="File1") in a Form. The Form must POST to the ASP page which uses the **Files** collection, and the ENCTYPE of the form must be "Multipart/Form-Data".

This collection is zero-based.

You **MUST CALL** one of the following methods to populate the **Files** collection: the <u>Save</u> method of either the Upload object or a <u>NextFile</u> object; or the <u>SaveToMemory</u> method of either an Upload object or a **NextFile** object. Please note that using the **NextFile** object's **Save** or **SaveToMemory** methods will populate the **Files** collection one file at a time, unlike the Upload control's save methods which populates the **Files** collection with one method call.

Loop through this collection when you want to utilize the uploaded files (e.g. as Attachment objects for the Mailer control). You can use the **Count** property of the collection as the upper loop delimiter with a standard For loop, or alternatively you can iterate through the collection by using a For Each loop.

See Also: <u>UploadedFile Object | NextFile Object | Tutorial 1: Uploading Multiple Files and Using the Form and Files Collections | Tutorial 3: ADO Support and Saving a File as a BLOB</u>

Example

Assumptions

- Instances of the Upload and Mailer controls have been created
- Another html or ASP page has POSTED files to the page whice
- The uploaded files will be used as email attachments.

For Each Item in objUpload.Files

'note that we are using the OriginalPath property of the UploadedF' ContentName property of the Attachment objects (ContentName' displayed in the client's email software). If we do not do this then that the uploaded file was saved under will be used instead (all up are automatically saved with a GUID preceding the original filent objMailer.Attachments.Add Item.Path, Item.OriginalPath Next

^{&#}x27; we will loop through all UploadedFile objects and use them ' as e

Form Collection (Dundas Upload Control 2.0)

Overview | Properties | Methods

Consists of FormItem objects which store uploaded form data.

Remarks

To use the Upload control the html form which POSTS the data must have an ENCTYPE of "Multipart/Form-Data".

The **Form** collection is used to retrieve form data. You MUST use this collection when examining form elements and their values since a form with an ENCTYPE of "Multipart/Form-Data" POSTS the data in such a way that you can no longer use ASP's Request.Form object to retrieve the data.

This collection is zero-based.

If a form element was not populated by the user then there will be no corresponding **FormItem** object in the control's **Form** collection (e.g. if you loop through the Form collection with a For ... Each loop there will be no **FormItem** object for the form elements left empty by the user). If you retrieve the value of a particular form element which was left empty by the user the return will be "Empty". You can test to see if the user left a form element blank, or empty, by using VBScript's *IsEmpty* method.

To use the Form collection you **MUST CALL** one of the following methods: the <u>Save</u> method of either the Upload control or a <u>NextFile</u> object; or the <u>SaveToMemory</u> method of either the Upload control or a **NextFile** object. Please note that using the <u>GetNextFile</u> method (which returns a **NextFile** object) will populate the **Form** collection with all form data up to the file input box in question. The Upload object's save methods, however, populates the Form collection with ALL POSTED form data in one method call.

To retrieve form items you can specify the name of the form element via

its string key (e.g. strVariable = objUpload.Form("txtSomeTextBox"). Alternatively you can retrieve an element's value by specifying the appropriate numerical index (e.g. strVariable = objupload.Form(0)).

You can retrieve multiple selections for one form element (e.g. a listbox) by using the **Count** property of the **FormItem** object (this will have the total number of items selected for the form element) along with the **Value** method. Use a For loop with the **Count** property of the **FormItem** object as the upper loop delimiter and call the **Value** method for each item. For example, to retrieve all selected elements for a multi-item listbox (named "ListBox") you could use the following syntax:

For i = 0 To objUpload.Form("ListBox").Count - 1

Response.Write objUpload.Form("ListBox").Value(i)

Next

Note: alternatively you can also use a For ... Each loop to access multiple-item form element values.

To iterate through all form elements you can use a standard For loop with the upper delimiter set to one less than the Form collection's **Count** property. You can also loop through all form elements by using a For Each loop (e.g. For Each Item in objUpload.Form). Then access each form element using the following syntax: strSomeString = objUpload.Form(Item).

The Form collection supports the following properties and methods:

Properties Description

Count Read-only property which stores the number of **FormItem** objects in the collection.

Methods Description

Use this method to retrieve items from the collection.The default member of the collection.

The *Item* method is the default member of the collection so it does not have to be used explicitly. For example, "Set objFormItem = objUpload.Form(0).Item" accomplishes the same thing as "Set objFormItem = objUpload.Form(0)".

See Also: FormItem Object | How to Use the Dundas Upload Control |
Tutorial 1: Uploading Multiple Files and Using the Form and Files
Collections

Count Property (All Dundas Upload Collections)

Use this read-only property to determine the number of elements currently stored in any Dundas Upload collection.

Syntax

AnyUploadCollection.Count

Remarks

Use this property to find out how many elements are currently being stored in any Dundas Upload collection. This is useful for determining the upper loop delimiter when iterating through all of a collection's items. Note that you can also iterate through any Dundas Upload collection with **For...Each** loops.

See Also: <u>Item</u>

Item (All Dundas Upload Collections)

Call this method to retrieve an element from any Dundas Upload collection. This is the default member of all collections.

Syntax

AnyMailerCollection.Item (*Index*)

The **Item** method syntax has the following parts:

Part	Description
	This argument can be either a number (integer or long data type) or a string key. All of the Dundas Upload collections are zero (0) based.

Remarks

If the *Index* argument is specified as a number then the item is retrieved via the index of the collection (zero-based). For example, to retrieve the first element in the Files collection you could use the following statement: Set objUploadedFile = Files.Item(0). If a string is specified then the element is retrieved via its key. For a listing of collection keys refer to the following:

- <u>Files</u> Collection string key = *TagName* property of the <u>UploadedFile</u> object.
- <u>Form</u> Collection string key = *TagName* property of the <u>FormItem</u> object.

This method is the default collection method so you do not have to explicitly declare it. For example, to retrieve the first UploadedFile object in the Files collection you could use the following: Set objUploadedFile = objUpload.Files(0).

Note: if you specify an index which is out of range then an exception will be thrown.

See Also: Count Property

UploadedFile Object (Dundas Upload Control 2.0)

Comprises the <u>Files</u> collection, and stores data for one uploaded file.

Remarks

The **UploadedFile** object is used to store information and data about an uploaded file, and each object corresponds to a file uploaded via an html form.

You must first call the control's <u>Save</u> or <u>SaveToMemory</u> method before attempting to use the **Files** collection. See the sample code below for an illustration of this.

If you call the **SaveToMemory** method then the **Path** property will be a zero-length string.

Example

'this sample code illustrates how to populate the Files collection ' with

'method failures usually throw an exception On Error Resume Next

'create an instance of the Upload Control

Set objUpload = Server.CreateObject("Dundas.Upload.2")

'we will examine the Err object to see if an error occurred, if so we will redirect user to an error page

If Err.Number <> 0 Then

Response.Redirect "Error.asp?Error=" & Err.Description End If

'call the SaveToMemory method of the control to populate the Files co objUpload.SaveToMemory

'loop through all uploaded files, and output their size in bytes

For Each objUploadedFile in objUpload.Files
Response.Write("Size: " & objUploadedFile.Size & "

Next

'destroy Upload object Set objUpload = Nothing

UploadedFile Object Properties

Path Read-only property which stores the uploaded

file's local path at the server.

OriginalPath This read-only property stores the original

pathname (at the client) of the uploaded file.

Size Read-only property which stores the uploaded

file's size in bytes.

<u>TagName</u> Read-only property which stores the name of the

form's file input box from which the uploaded file

originated.

ContentType Read-only property which stores the uploaded

file's content type.

Binary Read-only property which stores the uploaded

file's data as a safe-array-of-bytes. Useful for saving uploaded image files into a database

BLOB.

Binary (UploadedFile Object)

Overview

Read-only property which stores the uploaded file's data as a safe-array-of-bytes.

Syntax

UploadObject.Binary = [variant]

The **Binary** property syntax has the following parts:

Part	Description
variant	The file's data as a safe-array-of-bytes.

Remarks

To use the <u>Files</u> collection you must first call either the <u>Save</u> or <u>SaveToMemory</u> methods of either the <u>Upload</u> control or a <u>NextFile</u> object.

You can save uploaded binary files to a BLOB (e.g. a SQL Server "Image" column) by using an ADO recordset with a DSNless connection. Note that you can not utilize a system DSN to open the database.

See Also: <u>Tutorial 3</u>: ADO Support and Saving a File as a BLOB

ContentType (UploadedFile and NextFile Objects)

Overview

Read-only property which stores the uploaded file's content type.

Syntax

[string] = Object.ContentType

The **ContentType** property syntax has the following parts:

Part	Description
string	The value of the contenttype header.

Remarks

To use the **Files** collection (which stores <u>UploadedFile</u> objects) you must first call either the <u>Save</u> or <u>SaveToMemory</u> methods.

To work with a <u>NextFile</u> object you must first call the <u>GetNextFile</u> method.

The format of the content type is as follows: "Type/Sub-Type" (e.g. "image/bmp").

To determine what type of file has been uploaded you can use VBScript's *InStr* method with this property as the string to be searched. For example, to see if the uploaded file is an image you could use this line of code: InStr(1,objUploadedFile.ContentType,"image",1).

OriginalPath (UploadedFile and NextFile Objects)

Overview

This read-only property stores the original pathname (at the client) of the uploaded file.

Syntax

[string] = Object.OriginalPath

The **OriginalPath** property syntax has the following parts:

Part	Description
string	The full pathname (at the client) of the uploaded file.

Remarks

To use the **Files** collection (which stores <u>UploadedFile</u> objects) you must first call either the <u>Save</u> or <u>SaveToMemory</u> methods.

To work with a NextFile object you must first call the GetNextFile method.

This property is useful if you want to save files which have been uploaded to memory to disk with their original filenames (i.e. filename at the client). For sample source code demonstrating this see the SaveAs topic.

See Also: Path

Path (UploadedFile Object)

Overview

Read-only property which stores the uploaded file's local path at the server.

Syntax

[string] = Object.Path

The **Path** property syntax has the following parts:

Part	Description
string	The physical location of the uploaded file at the server.

Remarks

To use the **Files** collection (which stores **UploadedFile** objects) you must first call either the <u>Save</u> or <u>SaveToMemory</u> methods. If you use the **SaveToMemory** method then this property will be a zero-length string.

The **Path** property returns the physical path of the uploaded file at the server, even if the <u>UseVirtualDir</u> property has been set to TRUE.

This property is useful if you want to copy/move uploaded files using the filenames they were originally saved to disk with. For sample source code demonstrating this see the <u>Copy</u> topic.

See Also: OriginalPath

Size (UploadedFile Object)

Overview

Read-only property which stores the uploaded file's size in bytes.

Syntax

[long] = Object.Size

The **Size** property syntax has the following parts:

Part	Description
long	The size of the uploaded file in bytes.

Remarks

To use the **Files** collection (which stores **UploadedFile** objects) you must first call either the <u>Save</u> or <u>SaveToMemory</u> methods.

If you want to set the allowable size of a file to be uploaded use the <u>MaxFileSize</u> property.

TagName (UploadedFile, NextFile and FormItem Objects)

Overview | Properties | Methods

Read-only property which stores the name of the form's file input box from which the uploaded file originated.

Syntax

[string] = Object.TagName

The **TagName** property syntax has the following parts:

Part	Description
	The name of the form's file input box from which the uploaded file originated.

Remarks

To use the <u>Files</u> collection (which stores <u>UploadedFile</u> objects) you must first call either the <u>Save</u> or <u>SaveToMemory</u> methods. To use a <u>NextFile</u> object you must first call the <u>GetNextFile</u> method.

This property is extremely useful if you want users to upload a particular file type for a specific file input box. Examine the uploaded file's content type, and if the content type for a particular file input box is not correct you can take action accordingly. This is especially useful if you are retrieving uploaded files one at a time via the **GetNextFile** method (using **NextFile** objects) since this method gives you the option of not allowing the upload of a file to occur (only headers are retrieved with the **GetNextFile** method).

Refer to the source code below for further illustration on how to ensure that a file is a certain type for a given file input box.

See Also: ContentType

Example

'this sample code will assume there a file input box inside a POST forn 'box is "Audio", and we will perform type checking to make sure that 'the user actually uploaded a valid audio file to the server from this input box inside a POST forn 'box is "Audio", and we will perform type checking to make sure that

'create an instance of the control objUpload = Server.CreateObject("Dundas.Upload.2")

'now lets loop through the uploaded files

For Each Item in objUpload.Files

'we will check to see which file input element is responsible for the upl If (Item.TagName = "Audio") Then

'we now know the file came from the Audio file input box, but lets mal 'that the user actually uploaded some sort of audio file

If InStr(1,Item.ContentType,"audio") Then

'the file is actually an audio type

Response.Write "The uploaded file is a valid audio file."

End If

End If

Next

'release resources

Set objUpload = Nothing

UploadedFile Object Methods

AllowAccess Adds an allowance access control entity for a file which

has been saved to disk.

Copy Call this method to copy files which have been saved to

disk to a specified destination on the server.

<u>Delete</u> Call this method to delete uploaded files which have

been saved to disk.

DenyAccess Call this method to add a denial control entity for a file

saved to disk.

GetAttributes Call this method to retrieve the attributes of an

uploaded file which has been saved to disk.

Move Call this method to move files which have been saved to

disk to a specified destination on the server.

RevokeAccess Call this method to remove an allowance access control

entity for a file which has been saved to disk.

SaveAs Call this method to save an uploaded file in memory to disk.

<u>SetAttributes</u> Call this method to change the attributes of a file which has

been saved to disk.

SetOwner Call this method to set an owner to a file which has been saved

to disk.

AllowAccess (UploadedFile Object)

Adds an allowance access control entity for a file which has been saved to disk.

Syntax

UploadedFileObject.AllowAccess(Account As String, Flags As Long)

The **AllowAccess** method syntax has the following parts:

Part	Description
Account	The name of a valid Windows account. You can optionally precede the account name with the domain name and a backslash.
Flags	A long which determines the type of access to be granted.

Remarks

Call this method to set the access rights for an uploaded file which has been saved to disk. This method is not applicable if the <u>SaveToMemory</u> method has been used instead of the <u>Save</u> method, since there will not be a copy of the uploaded file on disk.

To use named constants use a server-side include for the DSUpload.inc file which is distributed with the Upload Control installation.

The Account argument uses the following syntax: "[Domain\]Username".

The *Flags* argument can have any of the following values or any logical combination of the following values:

Description

Value

Generic Access Rights

GENERIC_ALL	&H10000000
GENERIC_EXECUTE	&H20000000
GENERIC_WRITE	&H40000000
GENERIC_READ	&H80000000

Standard Access Rights

DELETE	&H00010000
READ_CONTROL	&H00020000
WRITE_DAC	&H00040000
WRITE_OWNER	&H00008000
WRITE_SYNCHRONIZE	&H00010000

Specific Access Rights for files

FILE_GENERIC_READ &H120089

FILE	GENERIC	WRITE	&H120116
		V V I \ I I I I	Q1 1±20±±0

FILE GENERIC EXECUTE &H1200A0

FILE	READ	DATA	&H0001

FILE_WRITE_DATA &H0002

FILE_APPEND_DATA &H0004

FILE_READ_EA &H0008

FILE_WRITE_EA &H0010

FILE_EXECUTE &H0020

FILE_READ_ATTRIBUTES &H0080

FILE_WRITE_ATTRIBUTES &H0100

FILE_ALL_ACCESS &H001f03ff

See Also: DenyAccess | RevokeAccess

Copy (UploadedFile Object)

Call this method to copy files which have been saved to disk to a specified destination on the server.

Syntax

UploadedFileObject.Copy (*Destination* As String [, *Overwrite* As Bool = FALSE])

The **Copy** method syntax has the following parts:

Part	Description
	The full pathname of the destination (including the filename).
	If this is TRUE then an existing file with the same filename will be overwritten. Defaults to FALSE

Remarks

An exception is thrown if the operation fails. Trap for success/failure by examining VBScript's Err object immediately after calling this method (the Number property of the Err object will be a non-zero value if it failed). MAKE SURE that you have enabled inline error trapping by using an **On Error Resume Next** statement at the beginning of the ASP page.

An exception will be thrown if you attempt to copy a file if the <u>Files</u> collection was populated using the <u>SaveToMemory</u> method. If files are saved using a virtual path (i.e. the <u>UseVirtualDir</u> property has been set to TRUE) then a virtual path should also be used for the <u>Destination</u> argument.

When calling this method you must specify the directory and filename for the desired destination (e.g. "e:\temp\myfile.txt"). To copy the file to a directory using the same name that the file was saved to disk with you can use the Upload control's GetFileName method, using the UploadedFile object's Path property as the argument. See the sample source code below for an illustration of this.

See Also: Move | Delete | Save

Example

'we will assume that the Files collection has been populated using the I control's Save method (e.g. uploaded files have been saved to disk).

'this sample code copies the files saved to disk to a directory named "te using the filenames that the files were saved to disk with.

'create instance of control
Set objUpload = Server.CreateObject("Dundas.Upload.2")

'populate collections and retrieve all uploaded form data (including 'uploaded files) objUpload.Save "c:\SomeDir"

'loop through all uploaded files and copy each file to the "temp" directory, using the same filenames that the files were saved 'to disk with.

For Each objUploadedFile in objUpload.Files objUploadedFile.Copy "c:\temp\" & objUpload.GetFileName(objUpload.Next

'release resources Set objUpload = Nothing

Delete (UploadedFile Object)

Call this method to delete uploaded files which have been saved to disk.

Syntax

UploadedFileObject.Delete

Remarks

An exception is thrown if the operation fails. Trap for success/failure by examining VBScript's Err object immediately after calling this method (the Number property of the Err object will be a non-zero value if it failed). MAKE SURE that you have enabled inline error trapping by using an **On Error Resume Next** statement at the beginning of the ASP page.

An exception will be thrown if you attempt to delete a file if the <u>Files</u> collection has been populated by calling <u>SaveToMemory</u> instead of the **Save** method.

See Also: Move | Copy | Save

DenyAccess (UploadedFile Object)

Call this method to add a denial control entity for a file saved to disk.

Syntax

UploadedFileObject.DenyAccess(Account As String, Flags As Long)

The **DenyAccess** method syntax has the following parts:

Part	Description
Account	A valid Windows account name. You can optionally precede the account name with the domain name and a backslash.
Flags	A long which determines the type of access to be denied.

Remarks

This method is not applicable if the <u>SaveToMemory</u> method has been used instead of the <u>Save</u> method, since there will not be a copy of the uploaded file on disk.

To use named constants use a server-side include for the DSUpload.inc file which is distributed with the Upload Control installation.

The Account argument uses the following syntax: "[Domain\]Username".

The *Flags* argument can have any of the following values or any logical combination of the following values:

Description Value

Generic Access Rights

GENERIC EXECUTE &H20000000

GENERIC_WRITE &H40000000

GENERIC_READ &H80000000

Standard Access Rights

Г	ELE	=TF	&H00010000
ᆫ	\prime $lue{}$ $lue{}$ $lue{}$ $lue{}$	_ L	

READ CONTROL &H00020000

WRITE DAC &H00040000

WRITE_OWNER &H00008000

WRITE_SYNCHRONIZE &H00010000

Specific Access Rights for files

FILE_GENERIC_READ &H120089

FILE_GENERIC_WRITE &H120116

FILE GENERIC EXECUTE &H1200A0

FILE_READ_DATA	&H0001
FILE_WRITE_DATA	&H0002
FILE_APPEND_DATA	&H0004
FILE_READ_EA	&H0008
FILE_WRITE_EA	&H0010
FILE_EXECUTE	&H0020
FILE_READ_ATTRIBUTES	&H0080
FILE_WRITE_ATTRIBUTES	&H0100
FILE_ALL_ACCESS	&H001f03ff

See Also: <u>AllowAccess</u> | <u>RevokeAccess</u>

GetAttributes (UploadedFile Object)

Call this method to retrieve the attributes of an uploaded file which has been saved to disk.

Syntax

long = UploadedFileObject.GetAttributes()

Remarks

This method returns the file's attributes as a long.

Please note that calling this method will result in an exception being thrown if the file has been saved to memory using the <u>SaveToMemory</u> method.

The attributes argument can have any of the following values or any logical combination of the following values:

Value	Description.
0	No attributes are set.
1	Read-only file.
2	Hidden file.
4	System file.
8	Disk drive volume label.
16	Folder or directory. Attribute is read-only.
32	File has changed since last backup (archive). Attribute is read/write.

- 64 Link or shortcut. Attribute is read-only.
- 128 Compressed file. Attribute is read-only.

Note: Calling this method will result in an exception being thrown if the file was saved to memory.

See Also: <u>SetAttributes</u>

Move (UploadedFile Object)

Call this method to move files which have been saved to disk to a specified destination on the server.

Syntax

UploadedFileObject.Move(Path As String[, FailIfExsists As Boolean =
True])

The **Move** method syntax has the following part(s):

Part	Description
Path	The full pathname of the destination (including the filename). If you saved uploaded files to disk using a virtual directory then this argument must also use a virtual directory.
FaillfExsists	If TRUE then an exception will be thrown if the file already exists at the destination folder. Defaults to TRUE.

Remarks

An exception is thrown if the operation fails. Trap for success/failure by examining VBScript's Err object immediately after calling this method (the Number property of the Err object will be a non-zero value if it failed). MAKE SURE that you have enabled inline error trapping by using an **On Error Resume Next** statement at the beginning of the ASP page.

An exception will also be thrown if the <u>Files</u> collection was populated by using the <u>SaveToMemory</u> method.

You must use a virtual directory for the *Path* argument if a virtual directory was used in the <u>Save</u> method call (i.e. the <u>UseVirtualDir</u>

property was set to TRUE before calling the **Save** method). The syntax for the *Path* argument would then be: "/VirtualDir/AnyOtherDirs/FileName".

See Also: Copy | Delete | Save

RevokeAccess (UploadedFile Object)

Call this method to remove an allowance access control entity for a file which has been saved to disk.

Syntax

UploadedFileObject.RevokeAccess(Account As String)

The **RevokeAccess** method syntax has the following part(s):

Part	Description
	The name of a valid Windows account. You can optionally precede the account name with the domain name and a backslash.

Remarks

This method will throw an exception if an invalid account name has been specified.

The Account argument uses the following syntax: "[Domain\]Username".

See Also: DenyAccess | AllowAccess

SaveAs (UploadedFile Object)

Call this method to save an uploaded file in memory to disk.

Syntax

UploadedFileObject.SaveAs (*Path* As String)

The **SaveAs** method syntax has the following part(s):

Part	Description
	The full pathname (i.e. path and filename) of the file's destination. If <u>UseVirtualDir</u> has been set to TRUE then this argument must also use a virtual directory.

Remarks

To utilize this method the file must have been saved to memory using the <u>SaveToMemory</u> method. An exception will be thrown if the file in question was not saved to memory. An exception will also be raised if you call this method twice for the same file.

If you have set the **UseVirtualDir** property to TRUE then the *Path* argument must also use a virtual directory. The syntax for the *Path* argument would then be: "/VirtualDir/AnyOtherDirs/FileName".

When calling this method you must specify the directory and filename for the desired destination (e.g. "e:\temp\myfile.txt"). To save the file to a directory using the original filename (i.e. the filename at the client) you can use the Upload control's GetFileName method, using the UploadedFile object's OriginalPath property as the argument. See the sample source code below for an illustration of this.

See Also: Save | SaveToMemory

Example

'we will assume that the Files collection has been populated using the I control's SaveToMemory method.

'this sample code saves the files in memory to a directory named "temp" using the file's original names (at the client)

'create instance of control
Set objUpload = Server.CreateObject("Dundas.Upload.2")

'populate collections and retrieve all uploaded form data (including 'uploaded files) objUpload.SaveToMemory

'loop through all uploaded files and save each file to the "temp" directory, using their original filenames
For Each objUploadedFile in objUpload.Files
objUploadedFile.SaveAs "c:\temp\" & objUpload.GetFileName(objUp)
Next

'release resources Set objUpload = Nothing

SetAttributes (UploadedFile Object)

Call this method to change the attributes of a file which has been saved to disk.

Syntax

UploadedFileObject.SetAttributes (*Attributes* As Long)

The **SetAttributes** method syntax has the following part(s):

Part	Description
Attributes	A long which determines the file attributes to be set.

Remarks

Calling this method will result in an exception being thrown if the file was saved to memory.

The attributes argument can have any of the following values or any logical combination of the following values:

Value	Description.
0	No attributes are set.
1	Read-only file.
2	Hidden file.
4	System file.
8	Disk drive volume label.
16	Folder or directory. Attribute is read-only.

32	File has changed since last backup (archive) Attribute is read/write.
64	Link or shortcut. Attribute is read-only.
128	Compressed file. Attribute is read-only.

See Also: GetAttributes

SetOwner (UploadedFile Object)

Call this method to set an owner to a file which has been saved to disk.

Syntax

UploadedFileObject.SetOwner(Account As String)

The **SetOwner** method syntax has the following part(s):

Part	Description
	The name of a valid Windows account. You can optionally precede the account name with the domain name and a backslash.

Remarks

An exception is thrown if the operation fails. Trap for success/failure by examining VBScript's Err object immediately after calling this method (the Number property of the Err object will be a non-zero value if it failed). MAKE SURE that you have enabled inline error trapping by using an **On Error Resume Next** statement at the beginning of the ASP page.

Note: Permission issues should be considered when using this method since an exception will be thrown if the user does not have the right to set the owner. Use the <u>ImpersonateUser</u> method to temporarily assume another Windows account which has the permission to set the file's owner. Once this is done call the <u>ImpersonationTerminate</u> method to resume using the default account.

See Also: ImpersonateUser | ImpersonationTerminate

FormItem Object (Dundas Upload Control 2.0)

Comprises the <u>Form</u> collection, and stores input data for a form element.

Remarks

The **FormItem** object is used to retrieve data from the POST form.

To use the Upload control the html form which POSTS the data must have an ENCTYPE of "Multipart/Form-Data".

To retrieve data you **MUST FIRST CALL** either the control's <u>Save</u> method, the <u>SaveToMemory</u> method or the <u>GetNextFile</u> method first so that the **Form** collection is populated. Once this is done you can retrieve data via the **Form** collection's **FormItem** objects.

Note: the **Save** and **SaveToMemory** methods will retrieve ALL form data with one method call, unlike the **GetNextFile** method which retrieves all form data up to the first populated file input box encountered. To retrieve all form data with **GetNextFile** call **GetNextFile** until the function returns "Nothing." for sample code demonstrating how to do this see the **GetNextFile** topic.

To retrieve form items you can specify the name of the form element via its string key (e.g. strVariable = objUpload.Form("txtSomeTextBox")). Note that the string key is actually the *TagName* property of the **FormItem** object. Alternatively you can retrieve an element's value by specifying the appropriate numerical index (e.g. strVariable = objupload.Form(0)). To retireve the value you do not have to specify *Value*.

NOTE: If a form element is left empty by the user then attempting to access the element's data via the Upload control's **Form** collection will result in a value of "Empty" being returned. To test for an empty form element use the following line of code: "If IsEmpty(objUpload.Form("MyFormElement")) Then ... Else...End If".

To retrieve multiple selections for one form element (e.g. a listbox) use the **Count** property of the **FormItem** object (this will have the total number of items selected for the form element). Use a For loop with the **Count** property of the **FormItem** object as the upper loop delimiter and call the **Value** method for each item. For example, to retrieve all selected elements for a multi-item listbox (named "ListBox") you could use the following syntax:

```
For i = 0 To objUpload.Form("ListBox").Count - 1

Response.Write objUpload.Form("ListBox").Value(i)

Next
```

Note: alternatively you could use a For ... Each loop to access these multiple selections. To use a For ... Each loop utilize the following syntax:

For Each Item in objUpload.Form

strSomeVariable = objUpload.Form(Item)

Next

See Also: Form Collection | How to Use the Dundas Upload Control

Count Property (FormItem Object)

Overview

This property indicates the number of data elements selected for a form element with multiple items.

Syntax

FormItemObject.Count

Remarks

This count property is useful if multiple data has been entered/selected in a form item (e.g. a multiple item listbox). Use this property to determine the total number of selected data entries.

You can retrieve the value of each data item using the <u>Value</u> property of the <u>FormItem</u> object in question.

See Also: Value

Value (FormItem Object)

Overview

Use this property to retrieve the value of a form element.

Syntax

string = FormItemObject.Value

Remarks

This property is essential for retrieving data from a multiple-entry form element (e.g. a multiple item listbox). It is also the default member of the **FormItem** object, so it does not have to be declared explicitly.

To retrieve multiple selected values for one single form element use the following syntax: string = objFormItem.[Value](Index), where Index is a zero-based long.

See Also: Count Property (FormItem Object)

NextFile Object (Dundas Upload Control 2.0)

Contains header information for a file originating from a file input box.

Remarks

A **NextFile** object is returned from a <u>GetNextFile</u> call.

The **NextFile** object stores header information for a file resulting from a form's file input box. You can examine the exposed properties of the **NextFile** object and decide whether or not you want to allow the user to upload the file. To allow the upload to occur call the <u>Save</u> or <u>SaveToMemory</u> methods of the **NextFile** object. To cancel the upload just call the **GetNextFile** method again without calling either the **Save** or **SaveToMemory** method first.

To see sample source code illustrating how to loop through all files the user wants to upload to the server see the **GetNextFile** topic.

See Also: GetNextFile | Tutorial 2: Retrieving Form Data Incrementally Using the GetNextFile Method

Example

'this sample code will determine if an image file has been specified by 'upload to occur. Note that we are assuming that a form has POSTED 'to this page, using an encoding type of "Multipart/Form-Data".

'create an instance of the Upload control
Set objUpload = Server.CreateObject("Dundas.Upload.2")

'call GetNextFile to retrieve the header for the file which the user want 'to upload. Note that GetNextFile will return a value of "Nothing" if 'the user did not specify any files at all to be uploaded. Also note that 'GetNextFile populates the control's Form collection with the values o 'ALL form elements which occur in the html form ONLY up until the

'encountered, populated file input box. If no populated file input box 'is found then all form element values will be inserted into the Form co Set objNextFile = objUpload.GetNextFile()

If InStr(1,objNextFile.ContentType,"image") Then objNextFile.SaveToMemory End If

'release resources Set objUpload = Nothing

NextFile Object Properties

Attributes Read-write property which allows you to retrieve or set the

attributes of a file stored in a NextFile object.

ContentType Read-only property which stores the file's content type.

FileName Read-write property which determines the file's full

pathname (at the server).

OriginalPath This read-only property stores the original pathname (at

the client) of the file in question.

<u>TagName</u> Read-only property which stores the name of the form's

file input box from which the file originated.

Attributes (NextFile Object)

Overview

This read-write property lets you retrieve or set the attributes of a file stored in a NextFile object.

Syntax

NextFileObject.Attributes = [long]

Part	Description
_	A long which determines the attributes of the file in question.

Remarks

To set this property make sure that you call either <u>Save</u> or <u>SaveToMemory</u> AFTER the attributes have been set. To set or retrieve a file's attributes AFTER the file has been saved to disk or memory use the Upload control's <u>GetAttributes</u> and <u>SetAttributes</u> methods.

A file's attributes can be any of the following values or any logical combination of the following values:

Value	Description.
0	No attributes are set.
1	Read-only file.
2	Hidden file.

4	System file.
8	Disk drive volume label.
16	Folder or directory. Attribute is read-only.
32	File has changed since last backup (archive). Attribute is read/write.
64	Link or shortcut. Attribute is read-only.
128	Compressed file. Attribute is read-only.

See Also: GetAttributes | SetAttributes

FileName (NextFile Object)

Overview

Read-write property which determines the uploaded file's full pathname (at the server).

Syntax

NextFileObject.FileName = [string]

The **Path** property syntax has the following parts:

Part	Description
string	The full pathname of the uploaded file (at the server).

Remarks

If files are to be saved with uniques filenames (the default) then the value of this property will be: GUID_OriginalFileName. If uniques filenames are not being used then this property will just consist of the original filename (i.e. the filename at the client).

If you would like to allow the upload of the file to occur and you want to specify the name the file is to be saved with then set this property BEFORE calling the <u>Save</u> method (<u>SaveToMemory</u> is not relevant here).

Important: if you specify the filename which the file is to be saved under then it is your responsibility to make sure that the name is unique (if unique filenames are desired). Use the <u>GetUniqueName</u> method of the Upload control to help you accomplish this.

See Also: GetUniqueName

NextFile Object Methods

Save Uploads and saves the file to disk.

<u>SaveToMemory</u> Uploads and saves the file to memory.

UploadProgress Component Overview

<u>Component Properties</u> | <u>Component Methods</u>

Use the **UploadProgress** component when you want to display a progress bar for an upload operation.

The UploadProgress Component is used to access the State Server and retrieve upload information which can then be used to update a progress bar window. It can retrieve new progress ID's, retrieve the progress information stored in the State Server and also delete state information by ID. It is also responsible for retrieving a new Progress ID when an upload operation is initiated (accomplished via the GetNewProgressID method).

To retrieve upload information set the <u>ProgressID</u> property to the unique Progress ID, call the <u>GetProgress</u> method and then utilize the <u>PercentCompleted</u>, <u>TotalSize</u>, and <u>UploadedSize</u> properties. Please note that these properties will be negative one (-1) if no data has been uploaded yet.

For detailed instructions on how to implement a progress bar see <u>Tutorial 4: Implementing a Progress Bar</u>.

Any errors which occur will result in an exception being thrown.

See Also: Overview (Dundas State Server)

UploadProgress Component Methods

The following is a list of all methods of the UploadProgress Component:

Method Description

GetNewProgressID Returns a new ProgressID (obtained from

the State Server Component) which is used to uniquely identify the upload operation.

GetProgress Connects to the State Server component

and causes the <u>TotalSize</u>, <u>UploadedSize</u> and <u>PercentCompleted</u> properties of the

Progress Bar component to be recalculated. You must obtain an ID (via **GetNewProgressID**) before calling this

method.

DeleteProgress Connects to the State Server component

and deletes the progress state information for the given ProgressID. You must obtain an ID (via **GetNewProgressID**) before

calling this method.

Please note that any errors will result in an exception being thrown.

See Also: <u>Progress Bar Component Methods</u> | <u>Overview (Progress Bar Component)</u>

UploadProgress Component Properties

The following is a listing of all UploadProgress Component properties:

Property	Data Type	Description
ProgressID	long	A unique Progress ID (set by the GetNewProgressID method).
PercentCompleted	l long	Read-only, the upload completion percentage (set by the GetProgress method). This will be negative one (-1) if there has been no data uploaded yet.
StateServer	string	The IP address of the machine where the State Server executable is running.
StateServerPort	long	The port number to use for the State Server Component.
TotalSize	long	Read-only, the total amount of data being uploaded (set by the GetProgress method). This will be negative one (-1) if there has been no data uploaded yet.

UploadedSize

long

Read-only, the amount of data which has been uploaded (set by the <u>GetProgress</u> method). This will be negative one (-1) if there has been no data uploaded yet.

See Also: <u>UploadProgress Component Methods</u> | <u>Overview</u> (<u>UploadProgress Component</u>)

Dundas State Server Overview

The Dundas State Server (StateServer.exe) is used to store upload progress information which can be stored and retrieved using the UploadProgress and Upload Components. This data is accessible from anywhere on a network, thereby allowing the State Server to function in a web-farm scenario.

The State Server can also be set up to run as a service (see below for more details).

Perform the following steps to set up the Server:

- Move the StateServer.exe to the machine which is to be used to store state information.
- 2. You can either run the server as a normal executable or alternatively you can run it as a service. To run it as a service refer to the section immediately following this listing of steps.
- 3. Set the following properties or alternatively use their defaults:
 - To accept a request from any machine on the network do not insert any entries into the "Valid Client's IP Addresses" listbox. However, if you want to restrict which machines on the network can access the server then enter their IP addresses into this listbox. Note that by specifying one or more addresses here only those IP addresses will be allowed to access the State Server.
 - 2. Set the port number which the server should listen on (this is the port number which you must specify when connecting to the State Server) or use the default port (6723).
 - 3. The timeout value determines when a request will be timed-out. You can accept the default of 15 seconds or set this yourself.
 - 4. The "State Valid For" value determines how long state information for a particular upload operation will be stored. You

can increase or decrease this value as you see fit.

5. The "Max. Connections" value determines how many components can access the State Server at one time. The default value is "20", but you may change this if you so desire.

If you have set up the State Server to run as a service then there will be no GUI to set these properties with. You will need to run the State Server manually (see below), set up your properties and then set the State Server back to running as a service.

To set the State Server up to run as a service you will need to run the executable from the command line with the following switches:

- i Run the Server as a service the next time the machine is booted.
- u Causes the State Server to be uninstalled as a service.
- m Runs the State Server manually from the command line. Use this option if the server is running as a service and you want to set or change one or more properties.

For example, to set up the State Server to run as a service use the following sytax from the command line (using a fictitious path): "e:\StateServer.exe /i".

For detailed help on how to utilize the State Server in conjunction with the UploadProgress component see <u>Tutorial 4: Implementing a Progress Bar</u>.

See Also: Overview (Progress Bar Component)

Copyright

Dundas Chart, Ultimate Toolbox, Ultimate Grid, Ultimate ChartPro, Ultimate Diagram, Hyperview, HyperHost, Ultimate FastMaps!, Ultimate TCP/IP, Ultimate VB/NTService, Ultimate 16 Bit CoolTools, Ultimate Wizard Factory and their respective documentation are copyright (c) Dundas Software Ltd.

Microsoft, MS, MS-DOS, Internet Explorer, Microsoft Developer Studio, Windows, Windows NT, Win32, and Win32s are registered trademarks of Microsoft Corporation.

Some files distributed with Ultimate Toolbox and/or incorporated with Dundas Chart are copyright (c) 1991-1995, Thomas G. Lane and form part of the Independent JPEG Group's software.

Some files distributed with Ultimate Toolbox are copyright (c) 1995, Intel Corporation.

OpenGL is a registered trademark of Silicon Graphics, Inc.

All other products and company names cited herein may be the trademarks of their respective owners.

DUNDAS SOFTWARE

Technical Support

We understand that when you use our development products you are making an investment in Dundas Software, and are putting your faith in us. That is why we take technical support very seriously.

All of these products, although free, are not shareware/freeware. They are commercial-quality components that come with extensive documentation, tutorials, examples, and fully-commented sample applications. You are entitled to free <u>email support</u>, we will attempt to help you with any of your inquiries as quickly as possible.

To further help our users, Dundas has a dedicated <u>Online Developer's</u> <u>Site</u>, where you will find even more examples, helpful articles, hints and tips, and notices concerning point releases and code updates.

Dundas Software's Priority support is also available, and comes with direct telephone and email assistance. Our professional support team can give you dozens of time saving pointers and development tips that can dramatically compress your development cycle. They can also offer you suggestions on how to enhance your programs, helping to make a great result even better. Call our sales department to find out more about Dundas Priority support.

Phone: 416 467-5100

Fax: 416 422-4801

E-mail: sales@dundas.com