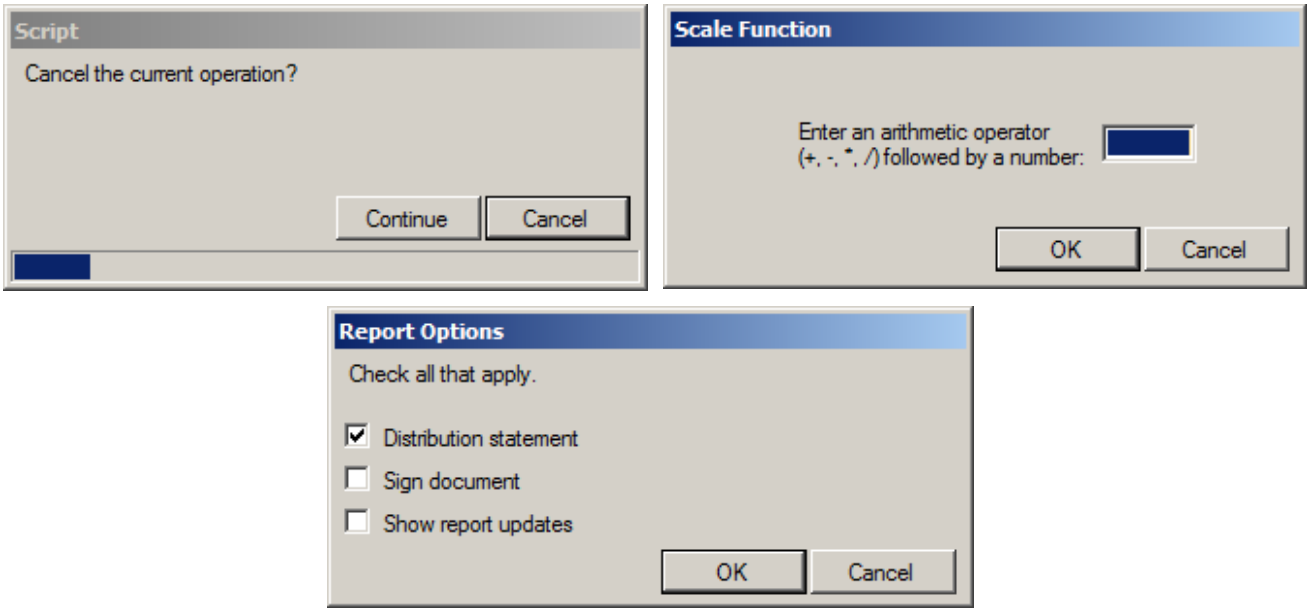
A Multipurpose Message Dialog

Darryl Bryk
U.S. Army RDECOM-TARDEC
Warren, MI 48397



Introduction

This article will describe a C# class for a more versatile Windows message dialog with the ability for custom buttons, textboxes, checkboxes, and a progress bar indicator. The .NET provided MessageBox is quite versatile for general use, but there was a need to have buttons like “Continue” or “Skip”, and rather than make up a custom dialog for each type needed, this class was developed. Over time, additions to the class were added for a non-modal progress bar indicator, with the option to cancel or continue, and the ability to define and get input from checkboxes and textboxes. Controls are automatically positioned, with the window expanding if more room is needed, and defaults for buttons, checkboxes and textboxes can be defined.

Using the Code

The class CMsgDlg is derived from the Windows Form. A few member variables may require some explanation. Recall that the Windows Form MessageBox is made to use the usual “OK”, “Cancel”, “Yes”, “No”, buttons and the clicked button result is returned in a MessageBoxButtons object. Since this new class has buttons that can be customized, an alternative method of returning which button was clicked must be used.

Buttons added to the form are defined by a string, and the button that was clicked is returned as this string. The dialog may be defined with multiple buttons and the default button can be set by passing the first character of the button text as an escape character ('\xd'). Dialogs that are called using checkboxes or textboxes are returned as a List<string>. To set checkboxes to the checked state (default is not checked) the escape character '\xc' can be used when defining the checkbox. There's nothing special about the escape values used here, but they need to be something that would not be likely to be used in the text string, so they can be identified as such, and stripped off from the string before being used in the form.

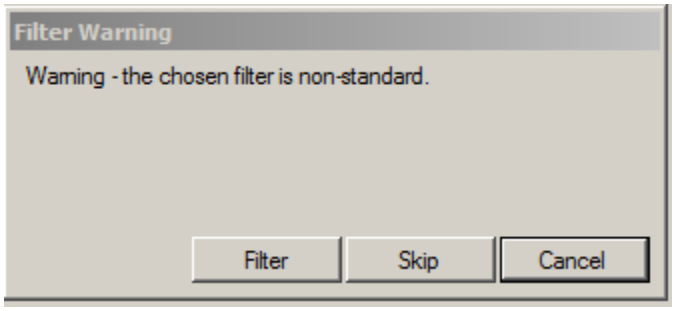
```
    private List<string> _lResults = new List<string>(); // Return list
    private string _btnResult; // Button clicked

    public const char escDef = '\xd'; // Default button
    public const char escChk = '\xc'; // Default check
```

To display a dialog with just buttons this class member may be called:

```
    public string ShowDialog(string text, string caption, string[] btnText) {…
```

where text is the message displayed in the dialog, caption becomes the form caption, and the array of strings btnText defines the button labels.

As an example, a dialog with three buttons "Filter", "Skip", and "Cancel" is called like this below. Each button will be placed in the form side by side, left to right, in the order as called. Note that the button that was clicked is just returned as a string. So for example, if the Cancel button was clicked, then the return string is "Cancel".

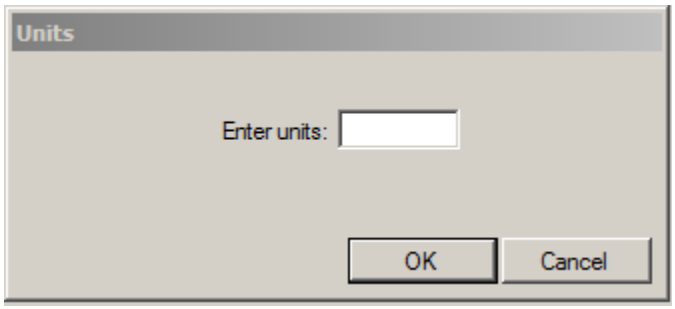



```
    string res = new CMsgDlg().ShowDialog("Warning - the chosen filter is non-"
       +standard.", "Filter Warning", new string[] { "Filter", "Skip", "Cancel" });

    if (res == "Cancel") break; // Exit loop
    else if (res == "Skip") continue; // Don't filter and continue to next iteration
    // …else just Filter it
```

To display a textbox dialog this method may be called:

```
    public List<string> ShowDialogTextBox(string caption, string[] lbl_txtboxText,
            string[] btnText = null) {…
```

lbl_txtboxText is an array of strings, where the prompt string is the first element of lbl_txtboxText ("Enter units:" in example below), and the second element is the default for the textbox (in this case a blank string). This second element also is used to define how wide the textbox should be.



An example textbox dialog may be called like this:

```
List<string> astr = new CMsgDlg().ShowDialogTextBox("Units",
  new string[] { "Enter units:", "                    " });

if (astr[0] == "Cancel") return;
string unit = astr[1].Trim(); // Entered string
```

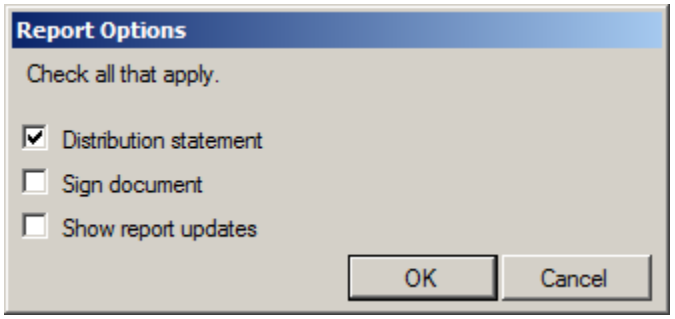
The return List<string> astr has as its first element (astr[0]), the button that was clicked, and the second element (astr[1]), is the string that the user typed into the textbox. Note that lbl_txtboxText is an array so that multiple textboxes could have been used on the form, alternating between the textbox label, and the default string. The btnText is optional, and if not supplied, the buttons are defaulted to “OK” and “Cancel”, as is done in the example above.

A multi-line textbox will be produced by including a newline (‘\n’) in the textbox default (second element) for each extra line. This example code shown below will provide a two line textbox.

```
List<string> lstr = new CMsgDlg().ShowDialogTextBox("Chart Edit", "Title:",
    otitle.Text + "        \n ");
```

The definition for the checkbox dialog is:

```
public List<string> ShowDialogCheckBox(string lblText, string caption,
        string[] chkBoxText, string[] btnText = null) {…
```



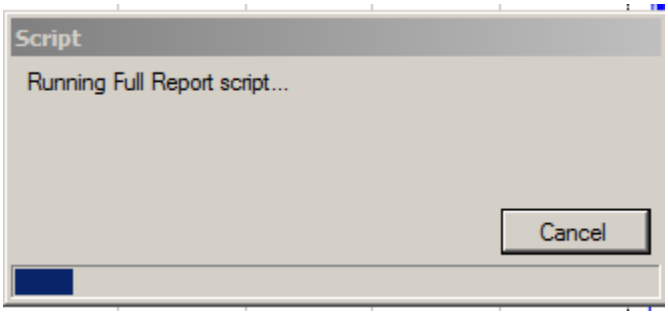
A dialog with checkboxes may be called like this:

```
    string chk_distr = (_bDistr ? CMsgDlg.escChk.ToString() : "") + "Distribution
       statement";
    string chk_Sign = (_bSign ? CMsgDlg.escChk.ToString() : "") + "Sign document";
    string chk_Show = (_bShow ? CMsgDlg.escChk.ToString() : "") + "Show report updates";

    List<string> astr = new CMsgDlg().ShowDialogCheckBox("Check all that apply.",
        "Report Options", new string[] { chk_distr, chk_Sign, chk_Show },
        new string[] { "\xdOK", "Cancel" });

    if (astr[0] == "Cancel") return;

    _bDistr = (astr[1][0] == CMsgDlg.escChk);  // Distr. statement
    _bSign = (astr[2][0] == CMsgDlg.escChk);   // Sign
    _bShow = (astr[3][0] == CMsgDlg.escChk);   // Show updates
```

The first three statements above define the checkbox labels and whether they should be defaulted to the "checked" state. For example, the first statement says if _bDistr is true, then the escChk (escape character defined above in the class) is inserted as the first character in the text label "Distribution statement". The member function checks for this to know if it should set the checkbox to "checked". If the escape character is not present then the checkbox defaults to not checked. The 1st element of the returned List<string> astr is the button clicked. If not the "Cancel" button, then the checkbox states are retrieved (escape characters) from the first character of these elements astr[1][0], astr[2][0], etc. So in this case _bDistr is true if the first character of astr[1] is the escChk character, _bSign is true if the first character of astr[2] is the escChk character, etc. The strings are returned in the same order that they were passed to the method.

The progress bar dialog method definition is:

```
    public CMsgDlg ShowProgress(Form parent, string text, string caption, string btnText
           = "Cancel") {…
```



This will initialize the display of a non-modal dialog for use in a loop, with a progress bar that gets updated with each iteration of the loop. Note that the parent Form is passed to this function. This is done strictly for positioning the dialog to default to the center of the parent form, because non-modal dialogs are not centered automatically. The btnText is optional and defaults to just the "Cancel" button.

An example call to this method:

```
CMsgDlg dlg = new CMsgDlg().ShowProgress(f1, "Cancel the current operation?",
    "Script");

int i = 0;
foreach (TreeNode node in chkNodes) { // For each checked node
    if (dlg.Result((double)++i / chkNodes.Count) == "Cancel") goto Exit;

    …

}

If (dlg != null) dlg.Close();
```

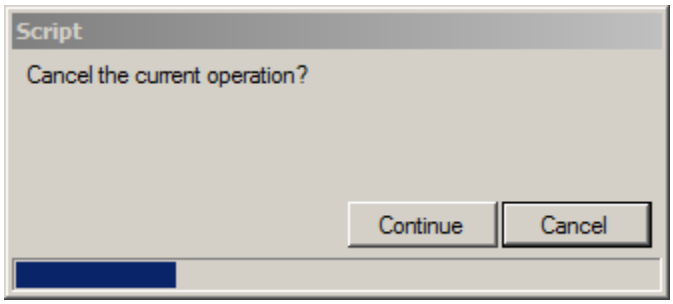
After the instantiating call to CMsgDlg().ShowProgress, the dlg object is retained to be used in the foreach loop to have access to the class's Result method to check if the user clicked the "Cancel" button, and to update the progress bar at each iteration of the loop.

The Result method is shown below:

```
//-------------------------------------------------------------------------
// Returns button result (for non-modal use) and updates progress by perc %
//-------------------------------------------------------------------------
public string Result(double perc, string text = "") {
    try {
        if (_prgBar != null) {
            if (perc <= 1.0) perc *= 100; // Convert to percentage
            _prgBar.Value = (int)(perc + 0.5);
        }
        if (text != "") _lbl.Text = text;
    }
    catch (Exception) { // Keep from crashing if passed invalid value
    }

    return _btnResult;
}
```

Note that an optional text string can be passed to change the text from its initial text (as called from ShowProgress) to, for example, "% completed". The perc value is converted to a percentage if less than 1 and used to show the progress bar's percent complete. In the foreach loop, this perc value is passed as a counter "i" divided by the total number of iterations chkNodes.Count. The counter "i" is incremented with each pass of the loop. The try block with the "null" catch ensures that even some erroneous call to the dialog that throws an exception doesn't halt the caller's loop. Note that it's the responsibility of the calling function to close the dialog, e.g. after the calling loop is exited.

Even though this is a non-modal dialog, a button click can be responded to by the dialog. If the user clicks the Cancel button, the private member variable _btnResult is set in the btn_Click method. The btn_Click method will then invoke the Cancel method, passing "true" to invoke a modal dialog which prompts the user to cancel or continue.



The Cancel method (shown below) makes the non-modal dialog non-visible, which effectively pauses the application. It then calls the modal ShowDialog, adds a Continue button and changes the text message, asking the user if they want to cancel the operation. If Continue is chosen, then the Continue button is removed from the form, the original message text is restored, and the non-modal dialog is made visible again, thereby permitting the application to continue and the loop to resume. If Cancel is chosen, then "Cancel" is returned in _btnResult, which is returned by the Result method to the caller in the loop, and the loop is exited.

Note that after the form's visible attribute is set to false and ShowDialog is called, the modal dialog has the same appearance as the progress bar dialog that was made non-visible (it has the Cancel button and progress bar), so just the Continue button needs to be added. Since the class hasn't been re-instantiated, the "this" operator is still pointing to the current form, the same form which was made non-visible. So it effectively acts as if the non-modal dialog was just changed to a modal dialog. When the form is made visible again, this causes it to continue, as it was, as a non-modal dialog.

```
    //--------------------------------------------------------------------------
    // Cancel handler for non-modal
    //--------------------------------------------------------------------------
    public void Cancel(bool bAsk = false) {
        if (bAsk) { // Ask 1st
            this.Visible = false;   // Disables non-modal dlg
            string lbl = _lbl.Text; // Save in case continue

            if (ShowDialog("Cancel the current operation?", this.Text,
                    "Continue") == "Continue") { // Already has Cancel btn
                _lbl.Text = lbl; // Restore
                _buttons.Last().Dispose(); // Continue btn
                _buttons.Remove(_buttons.Last());
                this.Visible = true; // Enable non-modal again
                Form1.TheForm().MsgDlg = this; // Restore for form - Close() nulls
                return;
            }
        }

        _lbl.Text = "Closing...";
        _btnResult = "Cancel"; // Set to close on next Result() check
    }
```

When the form is non-modal it doesn't always have the focus, so depending on how much computation is done in the loop, it may not respond right away. The class also has a KeyDown handler looking for the Esc key, which is often responded to more promptly than the Cancel button click. The main application can also have a KeyDown event looking for the Esc key, and so that's why the Cancel method is declared public, so that the main application key handler can call Cancel as well. The main application (Form1) needs to have access to the CMsgDlg object (to call Cancel), so when ShowProgress is first called, a public global variable in Form1 is

assigned the "this" object so it will have access to the Cancel method. This Form1 global is also restored here in the Cancel method because closing the modal dialog (after the button click) sets it to null.

The class also contains a ShowDialog method (shown declared below) that can utilize checkboxes and textboxes on the same form, although it is currently designed to put the checkboxes above the textboxes only.

```
public List<string> ShowDialog(string text, string caption, string[] btnText,
                    string[] chkBoxText = null, string[] txtboxText = null) {
```

Although this class was developed to handle most kinds of simple user dialogs on its own, there often is still a requirement for designing a more complicated input form, perhaps with dynamic requirements (i.e. not known at design time). For example, there was a need to design a dialog with checkboxes and textboxes which dynamically added textboxes based on how many bookmarks were found in a Word template document. So although the form is created through the designer, not by this class, the AddTextBox method was made public so it could be called multiple times to add a textbox for each bookmark. In this case, since the class's default form was not used, AddTextBox needs to know what form to add the textbox onto, and so this is why there's an alternate constructor to the class declared as: public CMsgDlg(Control form). So for more general usage, the methods AddTextBox and AddCheckBox can be defined as public so that these methods can be called with the alternate constructor, with the form passed, so the controls can be added from an external pre-designed window's form as needed, and the positioning of the controls is still handled by the class.

Conclusion

A general purpose message dialog class was discussed which can have multiple custom labeled buttons, checkboxes, textboxes, or a non-modal progress bar dialog. The class is by no means exhaustive of all message dialog capabilities, but will hopefully offer the developer a good starting point for a more powerful general purpose message dialog.

Appendix

The CMsgDlg class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace MsgDlgSpace {
//-----------------------------------------------------------------------
// CMsgDlg class
// A Windows message dialog with the ability for custom named buttons,
// textboxes, checkboxes, and a progress bar indicator.
//-----------------------------------------------------------------------
public class CMsgDlg : Form {
    private System.ComponentModel.IContainer components = null;
    private System.Drawing.Size _btnSize = new System.Drawing.Size(75, 23);
    private List<Button> _buttons = new List<Button>(); // Buttons list
    private string _btnResult; // Button clicked
    private List<string> _lResults = new List<string>(); // Return list
    private ProgressBar _prgBar;
    private Label _lbl;   // Default text label
    private Control _ctl; // For external form calls to members

    private const int _margin = 5;

    private static System.Drawing.Point _location; // Saved form position

    public const char escDef = '\xd'; // Default button
    public const char escChk = '\xc'; // Default check

    //-----------------------------------------------------------------------
    // Constructors
    //-----------------------------------------------------------------------
    public CMsgDlg(Control form) { // Alt. constructor for member access from ext. class
        _ctl = form;
    }
    public CMsgDlg() {
        _ctl = this; // For members that can use ext. forms

        this.SuspendLayout();
        this.Name = "CMsgDlg";
        this.ClientSize = new System.Drawing.Size(325, 120); // Form starting size
        this.ControlBox = false;
        this.SizeGripStyle = System.Windows.Forms.SizeGripStyle.Hide;
        this.StartPosition = FormStartPosition.CenterParent; // Only works for modal
        this.TopMost = true;
        this.ShowInTaskbar = false;
        this.Cursor = Cursors.Arrow;
        //this.AutoScaleDimensions = new System.Drawing.SizeF(325, 120);
        //this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
        this.AutoSize = true;
```

```
    // Label - generic label for messages on the form
    _lbl = new Label();
    _lbl.AutoSize = true;
    //_lbl.SetBounds(_margin, _margin, this.ClientSize.Width - _margin * 2,
    //               this.ClientSize.Height - _btnSize.Height - _margin * 3);
    _lbl.Location = new System.Drawing.Point(_margin, _margin);
    //_lbl.AutoEllipsis = true;
    this.Controls.Add(_lbl);
    this.ResumeLayout(false);
}

//---------------------------------------------------------------------------
// Modifies label text
//---------------------------------------------------------------------------
public void SetLableText(string text) {
    _lbl.Text = text;
}

//---------------------------------------------------------------------------
// Add buttons - will appear in order left to right - btn 0... btn.Count.
//---------------------------------------------------------------------------
private void AddButtons(string[] btnText) {
    if (btnText == null) return;

    for (int i = btnText.Count() - 1; i >= 0; i--) {
        if (btnText[i] != "") AddButton(btnText[i]); // New btn to left of previous
    }
}

//---------------------------------------------------------------------------
// Add a button (positioned left of previous button - 1st button is bottom
// right). btnText preceded with '\xd' makes it the (default) accept button.
//---------------------------------------------------------------------------
private void AddButton(string btnText) {
    if (btnText == "") btnText = "OK"; // Default

    int left, top; // Button location
    Button btn;

    if (_buttons.Count != 0) { // At least 1 prev. - add to left of previous
        btn = _buttons.Last();
        left = btn.Left - _btnSize.Width - 2; // Separation = 2
        top = btn.Top;
    }
    else { // Add 1st button (bottom right)
        left = this.ClientSize.Width - _margin - _btnSize.Width;
        top = this.ClientSize.Height - _btnSize.Height - _margin; // At bottom
        if (_prgBar != null) top -= _prgBar.Height;

        // Move _btn and resize form height if needed
        int t = GetNextTopPos();
        if (top < t) {
            top = t;
```

```
            if (top + _btnSize.Height + _margin > this.ClientSize.Height) {
                this.Height = top + _btnSize.Height + _margin;
            }
        }
    }

    btn = new Button();
    _buttons.Add(btn);
    btn.Size = _btnSize; // Default size
    btn.AutoSize = true;

    int strwidth = TextRenderer.MeasureText(btnText, btn.Font).Width; // In pixels
    if (strwidth + 8 > _btnSize.Width) { // Need 8 extra for buttons
        btn.Width = strwidth + 8; // Expand button for text
        left = this.ClientSize.Width - _margin - btn.Width;
    }
    btn.Location = new System.Drawing.Point(left, top);

    if (btnText[0] == escDef) { // Default button
        btnText = btnText.Remove(0, 1); // Must be removed or changes button size
        this.AcceptButton = btn;
    }
    btn.Text = btnText;
    if (btn.Text == "Cancel") this.CancelButton = btn;

    this.Controls.Add(btn);
    btn.Click += new System.EventHandler(btn_Click);
}

//---------------------------------------------------------------------------
// Show non-modal progress dialog
//---------------------------------------------------------------------------
public CMsgDlg ShowProgress(Form parent, string text, string caption, string btnText
       = "Cancel") {
    Form1.TheForm().MsgDlg = this; // For Esc key to cancel (set KeyPreview = true)

    // Add status bar to put progress bar on
    StatusBar sBar = new StatusBar();
    sBar.ClientSize = new System.Drawing.Size(this.ClientSize.Width, 17);
    this.Controls.Add(sBar);

    // Progress bar
    _prgBar = new System.Windows.Forms.ProgressBar();
    _prgBar.Dock = DockStyle.Fill;
    _prgBar.Style = ProgressBarStyle.Continuous;
    _prgBar.Maximum = 100; // Percent
    sBar.Controls.Add(_prgBar);

    if (_prgBar.Bottom > this.ClientSize.Height) this.Height = _prgBar.Bottom;

    return Show(parent, text, caption, btnText);
}

//---------------------------------------------------------------------------
```

```
// KeyDown handler (active for non-modal)
//---------------------------------------------------------------------------
private void KeyDownHandler(object sender, KeyEventArgs e) {
    if (e.KeyCode == Keys.Escape) { // Handle Esc key exiting
        if (!this.Modal)
            Cancel(true);
        else Close();
    }
}

//---------------------------------------------------------------------------
// Show non-modal dialog (for ShowProgress())
//---------------------------------------------------------------------------
private CMsgDlg Show(Form parent, string text, string caption,
        string btnText = "OK", string btn2Text = "", string btn3Text = "") {
    this.Text = caption;
    _lbl.Text = text;

    AddButtons(new string[] { btnText, btn2Text, btn3Text });

    // Non-modal must be centered manually
    this.StartPosition = FormStartPosition.Manual;
    if (_location.X != 0) // Saved before
        this.Location = _location;
    else if (parent != null) { // Center of parent
        this.Location = new System.Drawing.Point(
            parent.Location.X + (parent.Width - this.Width) / 2,
            parent.Location.Y + (parent.Height - this.Height) / 2);
    }
    this.Show(); // Call base class

    // Setup key handler (Needs to be done here)
    this.CancelButton = null;
    this.KeyPreview = true;
    this.KeyDown += KeyDownHandler; // For Esc key to cancel (set KeyPreview = true)

    return this;
}

//---------------------------------------------------------------------------
// Show modal dialog; returns button clicked string
//---------------------------------------------------------------------------
public string ShowDialog(string text, string caption, string[] btnText) {
    this.Text = caption;
    if (text != null) _lbl.Text = text; // May be null for AddTextBox()

    AddButtons(btnText);

    if (_location.X != 0) { // Use prev. location
        this.StartPosition = FormStartPosition.Manual;
        this.Location = _location;
    }

    this.ShowDialog(); // Call base class
```

```
    return _btnResult;
}

//---------------------------------------------------------------------------
// Show modal dialog (simplified for 1-2 buttons)
//---------------------------------------------------------------------------
public string ShowDialog(string text, string caption, string btnText = "OK",
                         string btn2Text = "") {
    return ShowDialog(text, caption, new string[] { btnText, btn2Text });
}

//---------------------------------------------------------------------------
// Show modal dialog with chkbox and txtbox options
//---------------------------------------------------------------------------
public List<string> ShowDialog(string text, string caption, string[] btnText,
                    string[] chkBoxText = null, string[] txtboxText = null) {
    this.Text = caption;
    _lbl.Text = text;

    if (chkBoxText != null) {
        foreach (string str in chkBoxText) AddCheckBox(str);
    }

    if (txtboxText != null) {
        for (int i = 0; i < txtboxText.Length; i += 2) {
            AddTextBox(txtboxText[i], txtboxText.Count() > i + 1 ? txtboxText[i + 1]
                : "");
        }
    }

    if (btnText == null) btnText = new string[] { "\xdOK", "Cancel" }; // Use
         defaults
    AddButtons(btnText);

    if (_location.X != 0) { // Use prev. location
        this.StartPosition = FormStartPosition.Manual;
        this.Location = _location;
    }

    this.ShowDialog(); // Call base class

    return _lResults;
}

//---------------------------------------------------------------------------
// Text box dialog
//---------------------------------------------------------------------------
public List<string> ShowDialogTextBox(string caption, string lblText,
        string txtboxText, string[] btnText = null) {
    return ShowDialogTextBox(caption, new string[] { lblText, txtboxText },
        btnText);
}

//---------------------------------------------------------------------------
```

```
// Multi-textbox dialog. Pass lbl_txtboxText as string[], where 1st string
// is textbox label, and 2nd string is default textbox text.
//-----------------------------------------------------------------------
public List<string> ShowDialogTextBox(string caption, string[] lbl_txtboxText,
                                      string[] btnText = null) {
    for (int i = 0; i < lbl_txtboxText.Length; i += 2) {
        AddTextBox(lbl_txtboxText[i], lbl_txtboxText.Count() > i + 1 ?
             lbl_txtboxText[i + 1] : "");
    }

    if (btnText == null) // Use defaults
        ShowDialog(null, caption, new string[] { "\xdOK", "Cancel" });
    else ShowDialog(null, caption, btnText);

    return _lResults;
}

//-----------------------------------------------------------------------
// Add a labelled text box; width and no. of lines defined by passed tbxText
//-----------------------------------------------------------------------
public TextBox AddTextBox(string lblText, string tbxText) {
    Label lbl = new Label();
    lbl.Top = GetNextTopPos();
    lbl.Text = lblText;
    lbl.AutoSize = true;
    _ctl.Controls.Add(lbl);

    string[] astr = tbxText.Split(new char[] { '\r', '\n' },
                   StringSplitOptions.RemoveEmptyEntries);
    int nlines = astr.Count();

    TextBox tbx = new TextBox();

    if (nlines > 1) {
        tbx.Multiline = true;
        tbx.AcceptsReturn = true;
        tbx.WordWrap = true;

        // Get longest string
        string longstr = astr[0];
        for (int i = 1; i < astr.Length; i++) {
            if (astr[i].Length > longstr.Length) longstr = astr[i];
        }
        tbx.Width = TextRenderer.MeasureText(longstr, tbx.Font).Width;
        if (tbx.Width + lbl.Width + _margin * 2 < _ctl.Width) // Expand to form
                  width
            tbx.Width = _ctl.Width - lbl.Width - _margin * 2;

        tbx.Height = TextRenderer.MeasureText("M", tbx.Font).Height * nlines + 6;
        tbx.Top = lbl.Top;
    }
    else {
        tbx.Width = TextRenderer.MeasureText(tbxText, tbx.Font).Width;
        tbx.Height = TextRenderer.MeasureText("M", tbx.Font).Height + 6;
```

```
        //tbx.Top = lbl.Top + lbl.Height / 2 - tbx.Height / 2; // Centered on label
        tbx.Top = lbl.Top;
        lbl.Top = tbx.Top + tbx.Height / 2 - lbl.Height / 2; // Center on tbx
    }
    tbx.Lines = astr; // Set to passed default (maintains \n's)

    // Expand form if needed
    int w = lbl.Width + tbx.Width + _margin * 2;
    if (w > _ctl.Width) _ctl.Width = w;
    int h = tbx.Top + tbx.Height + _margin * 2;
    if (h > _ctl.Height) _ctl.Height = h;

    lbl.Left = (_margin + _ctl.ClientSize.Width) / 2 - (lbl.Width + tbx.Width +
                _margin) / 2;
    tbx.Left = lbl.Left + lbl.Width + 1; // Pos. textbox right of label

    // Assoc. tbx with its label in case ID needed
    tbx.Name = lbl.Text;
    tbx.Tag = lbl;

    _ctl.Controls.Add(tbx);
    //_txtBox.Select(); //.Focus();

    return tbx;
}

//---------------------------------------------------------------------------
// Check box dialog
//---------------------------------------------------------------------------
public List<string> ShowDialogCheckBox(string lblText, string caption,
        string[] chkBoxText, string[] btnText = null) {
    _lbl.Text = lblText; // So AddCheckBox() can determine placement

    foreach (string str in chkBoxText) AddCheckBox(str);

    if (btnText == null) // Use defaults
        ShowDialog(lblText, caption, new string[] { "\xdOK", "Cancel" });
    else ShowDialog(lblText, caption, btnText);

    return _lResults;
}

//---------------------------------------------------------------------------
// Add a check box (pass 1st char. of text as escChk for checked default)
//---------------------------------------------------------------------------
private void AddCheckBox(string text) {
    CheckBox chkBox = new CheckBox();

    if (text[0] == escChk) { // Esc seq.
        text = text.Remove(0, 1);
        chkBox.Checked = true; // Def. to checked
    }
    else chkBox.Checked = false;
```

```
    chkBox.Text = text;
    chkBox.AutoSize = true;

    // Expand form if needed
    chkBox.Left = _margin;
    int size = chkBox.Width + _margin * 2;
    if (size > this.Width) this.Width = size;
    chkBox.Top = GetNextTopPos();
    size = chkBox.Bottom + _btnSize.Height + _margin * 2;
    if (size > this.Height) this.Height = size;

    _ctl.Controls.Add(chkBox);
}

//---------------------------------------------------------------------------
// Button click handler, stores _btnResult for return. Extra controls get
// stored in _lResults string list.
//---------------------------------------------------------------------------
private void btn_Click(object sender, EventArgs e) {
    _btnResult = (string)((Button)sender).Text;
    _lResults.Add(_btnResult);

    // Get any extra controls
    foreach (Control ctl in this.Controls) {
        if (ctl is TextBox) {
            _lResults.Add(ctl.Text.Trim(new char[] { ' ', '\n', '\r' })); // Trim
                    off extra \n's...
        }
        else if (ctl is CheckBox) {
            string str = ctl.Text;
            CheckBox chkbox = (CheckBox)ctl;
            if (chkbox.Checked)
                str = str.Insert(0, escChk.ToString()); // Insert code for checked
            _lResults.Add(str);
        }
    }
    //Application.DoEvents();
    if (!this.Modal)
        Cancel(true);
    else Close();
}

//---------------------------------------------------------------------------
// Returns button result (for non-modal use) and updates progress by perc %
//---------------------------------------------------------------------------
public string Result(double perc, string text = "") {
    try {
        if (_prgBar != null) {
            if (perc <= 1.0) perc *= 100; // Convert to percentage
            _prgBar.Value = (int)(perc + 0.5);
        }
        if (text != "") _lbl.Text = text;
        //Application.DoEvents(); // Force events update
    }
```

```
    catch (Exception) { // Keep from crashing if passed invalid value
    }

    return _btnResult;
}

//---------------------------------------------------------------------------
// Close handler
//---------------------------------------------------------------------------
public new void Close() {
    Form1.TheForm().MsgDlg = null; // Reset for form
    _location = this.Location; // Save user preference
    base.Close();
}

//---------------------------------------------------------------------------
// Clean up any resources being used
//---------------------------------------------------------------------------
protected override void Dispose(bool disposing) {
    if (disposing && (components != null)) {
        components.Dispose();
    }
    base.Dispose(disposing);
}

//---------------------------------------------------------------------------
// Cancel handler for non-modal
//---------------------------------------------------------------------------
public void Cancel(bool bAsk = false) {
    if (bAsk) { // Ask 1st
        this.Visible = false;   // Disables non-modal dlg
        string lbl = _lbl.Text; // Save in case continue
        _location = this.Location; // Save user preference for ShowDialog()

        if (ShowDialog("Cancel the current operation?", this.Text,
                "Continue") == "Continue") { // Already has Cancel btn
            _lbl.Text = lbl; // Restore
            _buttons.Last().Dispose(); // Continue btn
            _buttons.Remove(_buttons.Last());
            this.Visible = true; // Enable non-modal again
            Form1.TheForm().MsgDlg = this; // Restore for form - Close() nulls
            return;
        }
    }

    _lbl.Text = "Closing...";
    _btnResult = "Cancel"; // Set to close on next Result() check
}

//---------------------------------------------------------------------------
// Returns next available top pos. for an added control
//---------------------------------------------------------------------------
private int GetNextTopPos() {
    int top = _ctl.Height - _ctl.ClientSize.Height;
```

```
        if (top == 0) top = _margin;

        foreach (Control ctl in _ctl.Controls) {
            if (ctl is TextBox || ctl is CheckBox) {
                if (top < ctl.Bottom + _margin) top = ctl.Bottom + _margin;
                if (ctl is TextBox) top += 2; // Allow more space
            }
        }

        return top;
    }
}
}
```