The Clickable KornShell–Adding an Interface to your Scripts

In a previous article, *Creating Really Simple RSS From A Database*, I described how we can use the utilities in MKS Toolkit to build an RSS feed from the contents of a database describing MKS press releases. In this article, we'll look at how MKS Toolkit (and MKS KornShell, in particular) can help us build a graphical interface for entering the information into that database. Now, of course, we could just use Microsoft Access itself to enter this data, but we don't learn much about MKS Toolkit that way, do we? And the techniques that we cover here are very useful whenever you want to create a Windows interface to any kind of MKS KornShell script. In fact, several of the graphical utilities included in the MKS Toolkit are actually scripts with a graphical interface created using techniques like these.

REMEMBER THE DATABASE

Before we begin building our dialog, let's briefly review what the database looks like. The `press_releases.mdb` database is a Microsoft Access database which has had a data source name of `press_releases.dsn` already established for it. It contains a single table named `Releases` which contains a record for each press release. Each record has five fields: `Date`, `Title`, `Description`, `Link`, and `Live`. These fields contain, respectively, the date when the release was issued, the headline for the release, a description of the release (often the first paragraph), the link to the full release on the MKS Web site, and a Yes/No indicator of whether the release has gone live (that is, it has been released to media outlets).

The earlier article also covered how we can use the MKS Toolkit **db** utility to issue SQL commands to our database. Thus, it should come as no surprise that we are again going to eventually use this utility to add the new records that we enter through our graphical interface.

DESIGNING THE INTERFACE

Okay, before we start writing the actual MKS KornShell script that builds our interface, we should decide what that interface is going to look like. Windows interfaces have a variety of standard items called controls to let you enter information. You see these controls almost every time you see a Windows dialog. The basic ones are edit boxes which let you enter strings of text, list boxes which display lists that you can select items from, combo boxes which display a dropdown list for you to choose an item from that is displayed in the box at the top of the list, radio buttons which let you choose one of several options (like buttons on a car radio, choosing one deselects the others), check boxes which you let either check or uncheck them, and buttons which let you click on them to trigger an action.

So for our interface, we know we need to enter the values for each of the five fields and then trigger the adding of a new record to the database. Now we could just use five edit boxes, one for each field in a database record, but we wouldn't learn much about creating different types of controls and on top of that, some of the database fields have specific value requirements. So, let's look at each field we need to enter and see what the appropriate way to enter it is.

The title is simply a string of text, so a simple edit box will do for it. The same goes for the link to the full release. The description, on the other hand, while still a string of text, is usually a paragraph or more in length and needs much more room, so we'll need some kind of expanded edit box for it. The Date field really has three separate components: month, day, and year. Each of these components has a limited range of acceptable values. The month can be only be one of the 12 months; the day is between 1 and 31; and the year, for our purposes, can only be between 1999 and 2009. This sounds like three combo boxes: one with a list of month names, one with a list of days in the month, and one with a list of years. Finally, for indicating whether a release has gone live or not, two options present themselves. We could use a check box which when checked would indicate the release was live. Or we could use two radio buttons: one for yes, it's live and the other for no, it's not. For our purposes, we'll go with the radio button option.

Now that we've decided what we need to enter the information for a new database record, let's look at what else we need. As mentioned earlier, we need to be able to trigger adding our new record to the database. This looks like an

obvious job for a button labeled **Add**. It's also a good idea to provide a way to get out of our interface, so let's also add a **Done** button that will let us exit.

Okay, so now that we know what controls we need to include in our interface, how should we lay them out? To help keep things easily understandable, we'll keep the layout simple. Let's say, one line for the title, several lines for description, a line for the link, a line with the three combo boxes for the date, a line with the two radio buttons specifying whether or not the release is live, and finally a line with the Add and Done buttons centered. To help you understand what I'm talking about, I'm going to cheat a little bit and direct you to *Figure 1: The Press Release Entry Interface* which shows the completed interface.

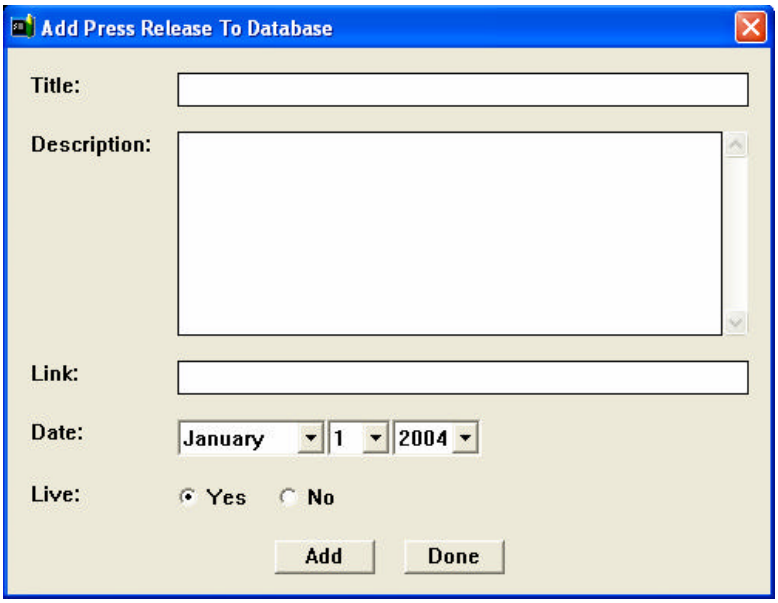

Figure 1: The Press Release Entry Interface

BUILDING THE INTERFACE

Now that we know what we want our interface to look like, how do we actually build it? Well, we could use a resource creator like the one provided in Microsoft Visual Studio and graphically build the interface, placing each of our desired controls and setting the appropriate properties for each. With this approach, we would then use the MKS Toolkit **`bindres`** utility to physically attach the compiled resource file for the interface to our MKS KornShell script. However, not everyone has access to such a resource creator. Fortunately, MKS KornShell includes the built-in **`dlg`** utility. This utility, through its various sub-commands, lets you load and close dialogs, add and delete controls, check for dialog-related events, and perform many other dialog manipulations. We will be using this utility extensively as we create our interface.

The MKS KornShell script described in the following sections is named `database_interface.ksh` and can be found on the MKS Web site at http://www.mkssoftware.com/docs/wp/dlggui.zip. This script sets up and operate the interface describe here.

The Basic Dialog

Before we can worry about placing controls or checking for events, we first need a basic Windows dialog. Since we don't want to use a resource creator to create such a dialog, we need to use a dialog from an existing program. The command:

```
dlg list program
```

displays the names or numbers associated with the Windows dialog in the specified *program*. For example:

```
dlg list sh.exe
```

shows the dialogs available in the MKS KornShell. On this list is the number 102. This is an empty dialog with no controls, just what we need for our database interface.

If you look at the `database_interface.ksh` script, you'll find the following commands in lines 57-59:

```
57  dlg load sh.exe 102
58  dlg setpos -x -1 -y -1 -w $DLGW -h $DLGH
59  dlg settext "Add Press Release To Database"
```

The first line loads the desired dialog (number 102 from `sh.exe`); the second positions the dialog on the screen (setting x,y values of `-1,-1` centers the dialog) and sets its height and width to the values in the variables `DLGH` and `DLGW`, defined earlier in the script (lines 42-43); and the third sets the title of the dialog.

You may have noticed that these lines appear in a function named `build_dlg`. By placing all the commands for drawing a dialog in a function, we can easily redraw the dialog at any time.

Adding Controls

Now that we have a basic dialog, it's time to add the individual controls. First, let's add the edit box for the title of the press release. First though, lets add a static control to label the actual edit box.

```
65  dlg addcontrol -C $IDS_TITLE -t Static -n "Title:"             \
66                 -w $STATICW -h $STATICH
```

This command sets the control's handle to the value of `IDS_TITLE` (we predefined a number of unique handle values on lines 10-15 of the script), specifies that it is a static control named "Title:", and sets the height and width of the control. To add the actual edit box, we use the command:

```
68  dlg addcontrol -C $IDE_TITLE -t Edit -h $STATICH               \
69                 -s $((ES_AUTOHSCROLL+WS_TABSTOP+WS_BORDER))
```

Like the previous command, this assigns a handle to the control, specifies its type (an edit box), and sets the height. It also uses the **`-s`** option to specify a number of Windows properties that are to be applied to the control. The variables for these Windows properties are defined earlier in the script and use the standard Microsoft names. You can find a complete list of these properties at http://doc.ddart.net/msdn/header/include/winuser.rh.html. Note that no width is specified for this control. We'll deal with this in *Laying Out the Dialog* below.

You may be starting to see a pattern in how we are naming the variables for our control handles. All such variables begin with `ID` followed by a single letter indicating the type of control. As you have seen, we use `S` for static and `E` for edit box. As we get to other controls, you will see that we use `B` for buttons, `R` for radio buttons, `C` for combo boxes, and so on. This three letter prefix is then followed by an underscore and the name (or an approximation) of the control. For example, the handle for the **Add** button is stored in the `IDB_ADD` variable. Though not entirely necessary, this technique lets us refer to the various control handles by their easy-to-remember variable names rather then having to keep track of individual id numbers. Having a coding convention like this is also fairly standard practice in any software development projects.

The commands for adding the label and edit box for the press release description are similar to those used for the title. The major difference is that we want this edit box to have multiple lines. To accomplish this, we set the height to hold six lines and specify a slightly different set of properties:

```
77  dlg addcontrol -C $IDS_DESC -t Static -n "Description:"        \
78                 -w $STATICW -h $STATICH

80  dlg addcontrol -C $IDE_DESC -t Edit -h $((EDITH * 6))          \
81                 -s $((WS_VSCROLL+ES_MULTILINE+WS_TABSTOP+WS_BORDER))
```

By specifying the `WS_VSCROLL` and `ES_MULTILINE` properties, the edit box will have a vertical scroll bar and can take up multiple lines. By omitting the `ES_AUTOHSCROLL` property used on the **Title** edit box, this edit box will not scroll horizontally when you hit the end of the line when entering data, but instead the text will wrap to the next line.

The commands for adding the label and edit box for entering the link to the full press release is pretty much the same as those for the **Title** edit box, so we'll just assume they can be added with little trouble.

We're now ready to add the three combo boxes that provide the dropdown lists for entering the date. Here are the commands to do so:

```
102 dlg addcontrol -C $IDS_DATE -t Static -n "Date:"               \
103                -w $STATICW -h $STATICH

105 dlg addcontrol -C $IDC_MONTH -t ComboBox -n ""                 \
106                -w 97 -h $COMBOH                                \
107                -s $((WS_TABSTOP+CBS_DROPDOWNLIST+WS_BORDER+    \
108                      CBS_SORT+WS_VSCROLL+CBS_AUTOHSCROLL))
```


```
110 dlg addcontrol -C $IDC_DAY -t ComboBox -n ""                    \
111                -w 42 -h $COMBOH                                 \
112                -s $((WS_TABSTOP+CBS_DROPDOWNLIST+WS_BORDER+     \
113                      CBS_SORT+WS_VSCROLL+CBS_AUTOHSCROLL))

115 dlg addcontrol -C $IDC_YEAR -t ComboBox -n ""                   \
116                -w 58 -h $COMBOH                                 \
117                -s $((WS_TABSTOP+CBS_DROPDOWNLIST+WS_BORDER+     \
118                      CBS_SORT+WS_VSCROLL+CBS_AUTOHSCROLL))
```

As with the edit boxes, you can see that we first added a static control to label the line with "Date:". We then added a combo box for each of month, day, and year that was set to the appropriate width and with the desired properties.

But wait a minute! How does our interface know what it is to display in each of these dropdown lists? Good question, and an easy one to answer. The following commands define the contents of the lists to be displayed:

```
120 dlg addtext -c $IDC_MONTH January February March April May June \
121                           July August September October November December

122 dlg addtext -c $IDC_DAY 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16  \
123                         17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

124 dlg addtext -c $IDC_YEAR 1999 2000 2001 2002 2003 2004 2005     \
125                           2006 2007 2008 2009
```

Normally, if we just specified these lists, the date would default to January 1, 1999, that is, to the first item in each list. But that means we are likely to constantly be changing the date. So, we're going to set the year to default to the current year (which is 2004 as of this writing). We can do this with the **`setcursel`** sub-command of **`dlg`**:

```
129 dlg setcursel -c $IDC_YEAR -i 5
```

This specifies which item in the dropdown list is set as the default. But isn't 2004 the sixth item in the list of years that we assigned to the `IDC_YEAR` control? Yes, it is, but the **`setcursel`** sub-command actually starts counting at zero and not one, so the sixth item has an index of five.

For simplicity, we did this statically. We could easily have used the **`date`** command to get the current date, and dynamically set the year, month and day combo boxes to those values. I'll leave that enhancement as an exercise, try using `date +"%e/%B/%Y"` and shell variable substitution rather then separate calls to **`date`**.

Next up is the radio buttons for specifying whether or not the press release is live. The following commands define the two radio buttons labeled "Yes" and "No" and the static control that labels them:

```
135 dlg addcontrol -C $IDS_LIVE -t Static -n "Live:"               \
136                -w $STATICW -h $STATICH

138 dlg addcontrol -C $IDR_LIVE -t Button -n "Yes"                 \
139                -w $BUTTONW -h $BUTTONH                         \
140                -s $((WS_GROUP + WS_TABSTOP + BS_AUTORADIOBUTTON))

142 dlg addcontrol -C $IDR_NONLIVE -t Button -n "No"               \
143                -w $BUTTONW -h $BUTTONH                         \
144                -s $((WS_TABSTOP + BS_AUTORADIOBUTTON))
```

Notice that to create a radio button, you use the button type just like you would for a normal button. It is the `BS_AUTORADIOBUTTON` property set by the **`-s`** option that turns it into a radio button.

We now just have two more controls to add to our interface: the Add and Done buttons. Here are the commands to do just that:

```
150 dlg addcontrol -C $IDB_ADD -t Button -n "Add"                  \
```

```
151                 -w $BUTTONW -h $BUTTONH                          \
152                 -s $WS_TABSTOP

154 dlg addcontrol -C $IDB_DONE -t Button -n "Done"                  \
155                 -w $BUTTONW -h $BUTTONH                          \
156                 -s $WS_TABSTOP
```

And that's all there is to it. We have now added all the controls that we need to implement our graphical interface.

Laying Out the Dialog

Creating the controls is only the first step in creating the dialog you see in *Figure 1: The Press Release Entry Interface*. The next step in building our database interface is to actually lay out the controls on our dialog. The **`layout`** sub-command of **`dlg`** allows us to specify the location of each control on the dialog. These locations are often given relative to the dialog itself or to other controls. Here is the first part of the **`dlg layout`** command that positions items on our interface:

```
161 dlg layout                                                       \
162    "move left($IDS_TITLE)       left(parent) +$MARGIN"           \
163    "move top($IDS_TITLE)        top(parent) +$MARGIN"            \
164    "move left($IDE_TITLE)       right($IDS_TITLE)"               \
165    "move top($IDE_TITLE)        top(parent) +$MARGIN"            \
166    "stretch right($IDE_TITLE)   right(parent) -$MARGIN"          \
167                                                                  \
168    "move left($IDS_DESC)        left(parent) +$MARGIN"           \
169    "move top($IDS_DESC)         bottom($IDS_TITLE) +$MARGIN"     \
170    "move left($IDE_DESC)        right($IDS_TITLE)"               \
171    "move top($IDE_DESC)         bottom($IDE_TITLE) +$MARGIN"     \
172    "stretch right($IDE_DESC)    right(parent) -$MARGIN"          \
```

Essentially, all controls and the dialog itself (called `parent`) have `left`, `right`, `top`, and `bottom` values which define how far from the left (for `left` and `right`) and top (for `top` and `bottom`) edges of the screen, the item is located. For example, the left edge of the **Title** edit box is referred to as `left($IDE_TITLE)`. By using the `move` and `stretch` instructions you can specify where to place the edges of each control.

For example, in the above code, the left edge of the static control labeling the **Title** edit box is placed `MARGIN` units from the left edge of the parent dialog, while its top is placed `MARGIN` units from the top of that dialog. `MARGIN` is defined on line 44 of the script within the `build_dlg` function. Similarly, the left side of the **Title** edit box itself is positioned at the right side of the static control labeling it. As you may remember, we never actually specified a width for this control. That is because we are simply going to ignore any width specified and use the `stretch` instruction to extend the edit box across the dialog until its right edge is `MARGIN` units from the right edge of the parent dialog box.

This same technique was used in the code above to position the static and edit box controls for the press release description; however instead of locating the top of these controls relative to the top edge of the dialog, we located them `MARGIN` units below the bottom of the static and edit box controls for the release title.

As mentioned above, this code is just the first section of the long **`dlg layout`** command found starting on line 161 in the `database_interface.ksh` script that lays out the entire dialog. Using the same methods described here, it proceeds to lay out each row of controls one under another.

Like the basic definition of the dialog and the adding of the controls, these layout commands appear within the `bld_dlg` function of our script.

Taking Action

With all the dialog controls defined and positioned, it's time to look at the main body of the `database_interface.ksh` script. The first thing this section of code does is display the dialog by calling the `bld_dlg` function. It then makes sure that the **Yes** radio button for specifying if the release is live or not is checked with:

```
214 dlg checkbutton -c $IDR_LIVE 1
```

and positions the cursor in the **Title** edit box with:

```
215 dlg setfocus -c $IDE_TITLE
```

The script has now reached the point where it is ready to start checking for dialog-related events. This will be done with the **dlg event** sub-command of **dlg**. But first we should clear out any old events before looking for new ones. The command:

```
221 dlg event flush
```

removes any dialog-related events that had not been processed.

To look for new events of interest, we simply need to execute **dlg event** over and over again until something happens to stop the process. The easiest way to do this is with an endless loop such as:

```
222 while :
223 do
224     dlg event msg ctrl
        ...
255 done
```

The **dlg event** command in this loop returns the message generated by the event in the variable `msg` and the identifier of the control that generated it in the variable `ctrl`. We can then use a MKS KornShell **case** statement to act on the different messages and control the identifier combinations generated. In truth, though, there are really only three events we are interested in: closing the dialog via the **X** in the upper right corner or via the system menu, clicking the **Done** button, or clicking the **Add** button. Closing the dialog generates the `close` message, while clicking the **Done** and **Add** buttons generate the `'command $IDB_ADD'` and `'command $IDB_DONE'` message/control ID combinations, respectively. So, our case statement must check for these three situations and perform the necessary actions.

Let's start with the **Add** button. When the user clicks this button we want to add the values of each of the fields as a new record in our database. First, we need to get those values. **dlg** provides the **gettext** command to do just that. So, our first step is to use a series of **dlg gettext** commands to the retrieve the contents of the **Title**, **Description**, and **Link** edit boxes and store them in the variables `title`, `desc`, and `link`. One interesting thing about assigning variables using the **gettext** command is the format of that command. Traditionally we would assign a variable like this:

```
Variable=value
```

Using **dlg gettext** on the other hand, we need to use syntax like this:

```
dlg gettext –c control_handle value
```

Lines 228-234 of the `database_interface.ksh` script contain the exact syntax needed for our database interface.

We can also use **dlg gettext** to retrieve the values chosen from the combo boxes for the date of the release. By concatenating the values for month, day, and year in the variable `db_date`, we have a complete date that can be added to the database. String concatenation within a shell script is done like this:

```
236 db_date="$month $day, $year"
```

Now that we have all the textual information and have created a date string from the combo boxes, we need to find out whether the release is live or not from the radio button controls. To do this we use the **dlg isbuttonchecked** command to see whether the **Yes** radio button was selected. The **Yes** button has the `IDR_LIVE` handle and we would query it like this:

```
234 dlg isbuttonchecked -c $IDR_LIVE live
```

If **Yes** was selected, this command returns 1 in the variable `live`; otherwise, if **No** was selected then **Yes** is unselected and the command returns 0. Fortunately, 1 and 0 are exactly the values to be passed to the database for the `Live` Yes/No field of the new record to be added.

Now that we retrieved all the values entered through our interface, it is time to add the new record to the database. To do so, we simply use the following **db** command:

```
237 db -d press_releases.dsn "INSERT INTO Releases VALUES   \
238                           ('$db_date', '$title', '$desc', '$link', '$live')"
```

Once we have added the new record, we can use **`dlg settext`** to clear out the edit boxes, **`dlg setcursel`** to reset the date combo boxes, and **`dlg checkbutton`** to reset the radio buttons. We then use **`dlg setfocus`** to move the cursor back to the edit box for the title and instruct the endless loop to continue. These are the same commands we have discussed above so we wont go into the details here. You can refer to the `database_interface.ksh` script on the MKS Web site for the details on how exactly we use these commands.

As for the other two events we were concerned about, a close examination reveals that closing the dialog and clicking **Done** should do exactly the same thing: close the dialog and exit the loop.

```
250 "command $IDB_DONE" |                    # Done button
251 "close ")
252     dlg close
253     break;;
```

As shown, we can simply combine these situations in our **`case`** statement. When either of these events occur, we use **`dlg close`** to close the dialog and the MKS KornShell **`break`** statement to break out of the endless loop (consequently, reaching the end of the script).

IS THAT ALL?

The graphical interface that we have created does the job, but that doesn't mean it can't be improved. For example, once we choose a month, we could change the list of available days that appears in the **Day** combo box to only include the days in that month (for example, selecting June would only show days 1-30 while July would show days 1-31). Alternatively, we could add code to check the date entered and when invalid, issue an error message using the **`msgbox`** utility instead of blindly adding the record as the script currently does. As mentioned earlier, we could also use the MKS Toolkit **`date`** command to get the current date and set the **Month**, **Day**, and **Year** combo boxes to default to the current date.

We could also make sure that there are values entered for all fields before adding a new record to the database and disable (grey out) the **Add** button until values had been entered for all three of the **Title**, **Description**, and **Link** edit boxes. **Date** and **Live** have default values so they don't need to be considered. Another possible enhancement here would be to check to see if a record containing the data entered already exists in the database, and if so, not add a duplicate record.

Finally, we could add use the **`filebox`** utility to add a file selector that would let users chose the database (or at least the data source name) to work with.

CONCLUSIONS

Well, we're done. We've used the tools of the MKS Toolkit (particularly, MKS KornShell's built-in **`dlg`** utility) to create and operate a graphical interface for entering press release information into a database. With any luck, as we've looked at how these tools were used for this particular application, you've been inspired as to how you yourself can use them to meet your own particular needs. After all, your users may not be as command line savvy as you.

For more information about the MKS Toolkit products please visit http://www.mkssoftware.com and to view the full reference pages for the commands mentioned in this document visit http://www.mkssoftware.com/docs/cmd_index.asp.