A Sample Application for

Programmers New to PureBasic

Table of Contents

Introduction

You don't see this anymore, but back in the early days of PCs and BASIC, computer magazines came out every month with one or two applications with code listings and detailed discussions about the code that anyone who knew the basics of coding could follow along. These applications were developed and submitted by the readers themselves and ran the gamut from calorie trackers to music organizers and everything else in between. Even if you had no use for an application, reviewing the code let you learn coding techniques.

In the spirit of those long-gone days, I present this sample application and code discussion. It's not a tutorial. It's assumed you know about variables, procedures, loops, and stuff like that. But just the basics is fine. You don't need a high level of knowledge. For this project, I've used PureBasic in a representative, real world manner, using some of the same practices and coding techniques you would find in larger, more complex projects, but I've also stayed away from the more complex features, like pointers, modules, and assembly code to make the project accessible to the most people. I hope you find it useful!

Getting Started

The zip file contains the complete PureBasic project in the Contacts folder. Drag this folder to wherever you keep your PureBasic projects.

The zip file also contains a blank database called contacts.mdb. Drag this blank database to the computer location of your choice, and then use the ODBC control panel to create a DSN (Data Source Name). Call it dbContacts. As far as I know, the only difference is that a User DSN works only for the logged in user, while a System DSN works for anyone who logs in, so create whichever works best for you. It doesn't affect the application.

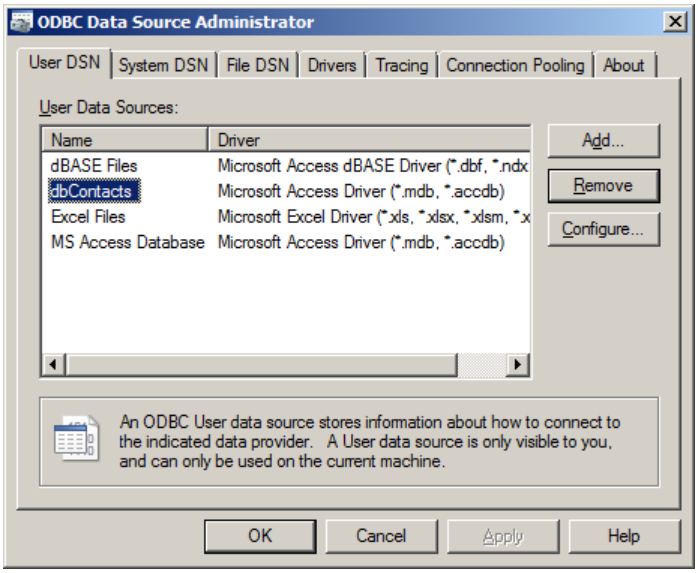

Figure 1- ODBC Control Panel

You can now fire up PureBasic and open the Contacts project. The project should compile fine in the free demo version of PureBasic.

If setting up the ODBC DSN gives you trouble, this is very common. Most problems are caused by using the 64-bit ODBC control instead of the 32-bit one. You must use the 32-bit version of ODBC. However, 64-bit versions of Windows open the 64-bit version when you go through Control Panel. If you just go into Control Panel and select the ODBC applet, you get the 64-bit version.

To get around this, look in your Windows folder for a folder called SysWOW64. This folder contains 32-bit versions of various things. Look for the file called odbcad32.exe and double-click on it. This is what you should use to create your DSN.

The Application

The application is a contact manager. It consists of a form that displays a list of records with a selector bar. The form has a toolbar with buttons to add and delete records as well as to refresh the list of records. The arrow keys move the selector bar up and down, but the grid itself is not editable.

To edit a record, double-click on it or select it and press Enter. The record is displayed in its own form. While editing or adding a record, the add, delete, and navigation buttons are disabled, and only the save and cancel buttons are enabled. The reverse is true when navigating. When you delete a record, you will be asked for confirmation.

The application has a Help menu with an About entry that opens a modal form for the purpose of illustrating how this is done. The form in which records are added or edited is modeless and

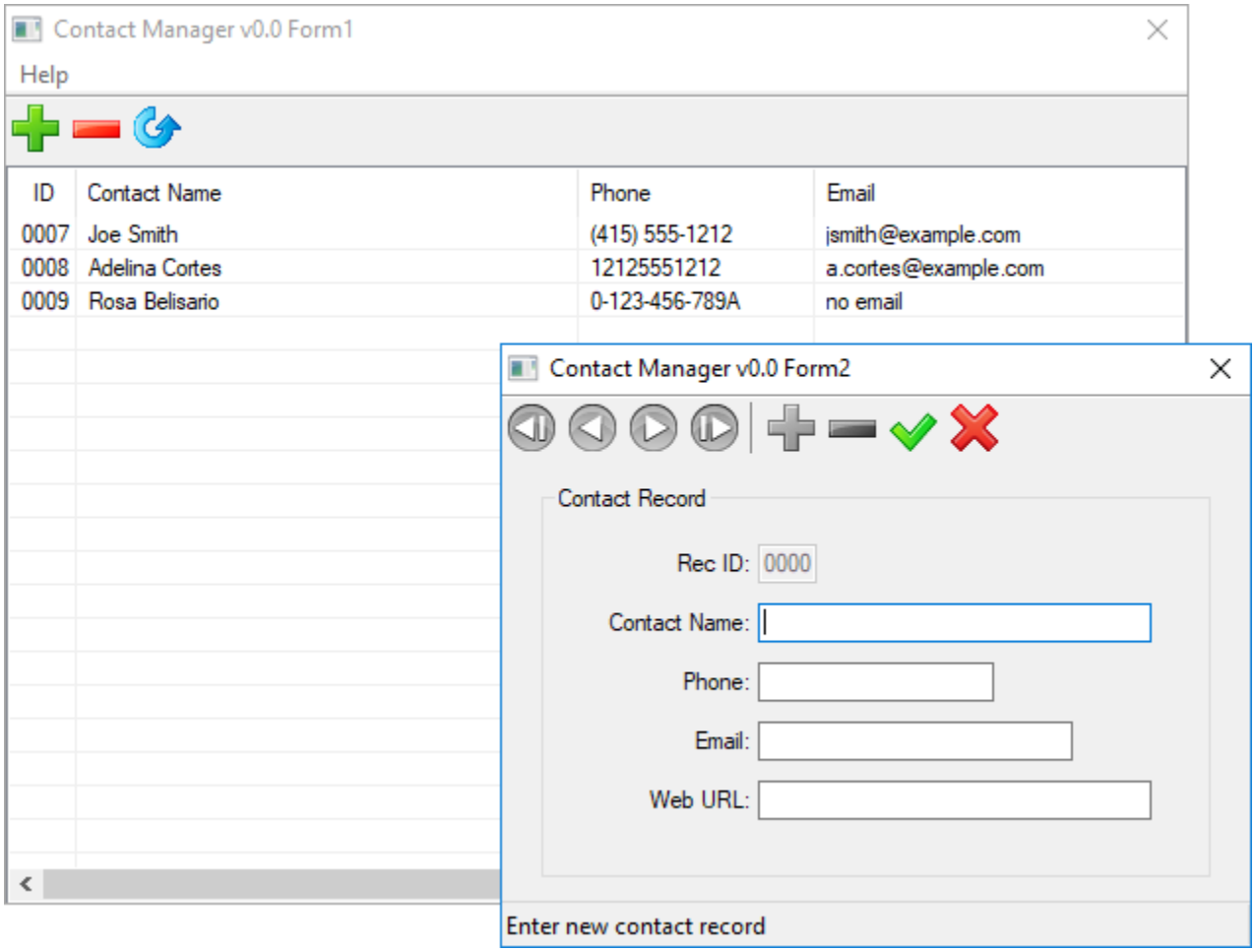

Figure 2 - Contacts Application Screenshot (Windows 10)

multiple instances of it can be opened. Modal and modeless will be discussed in more detail later on.

Data Management

All the procedures to implement the interaction shown in Figure 3 are in the file appDatabase.pb. A call to QueryContacts, if the query succeeds, copies the records returned by the query from the database to ContactsList, so you can think of ContactsList as an in-memory database. The structure ContactsRecord holds a single record, which you can think of as the current record. A call to InsertRecord, for example, attempts to insert the record contained in ContactsRecord. To edit a record, load the record's ID in ContactsRecord and call GetContacts. If the contact exists, the rest of the data will be placed in the rest of the ContactsRecord structure. Likewise, to update a record, load the updated values in ContactsRecord and call UpdateRecord. And to delete a record, place its record ID in ContactsRecord and call DeleteRecord.

At some point, data has to be transferred between the gadgets on a form and ContactsRecord. The procedures that handle this have to go in each form's code file.

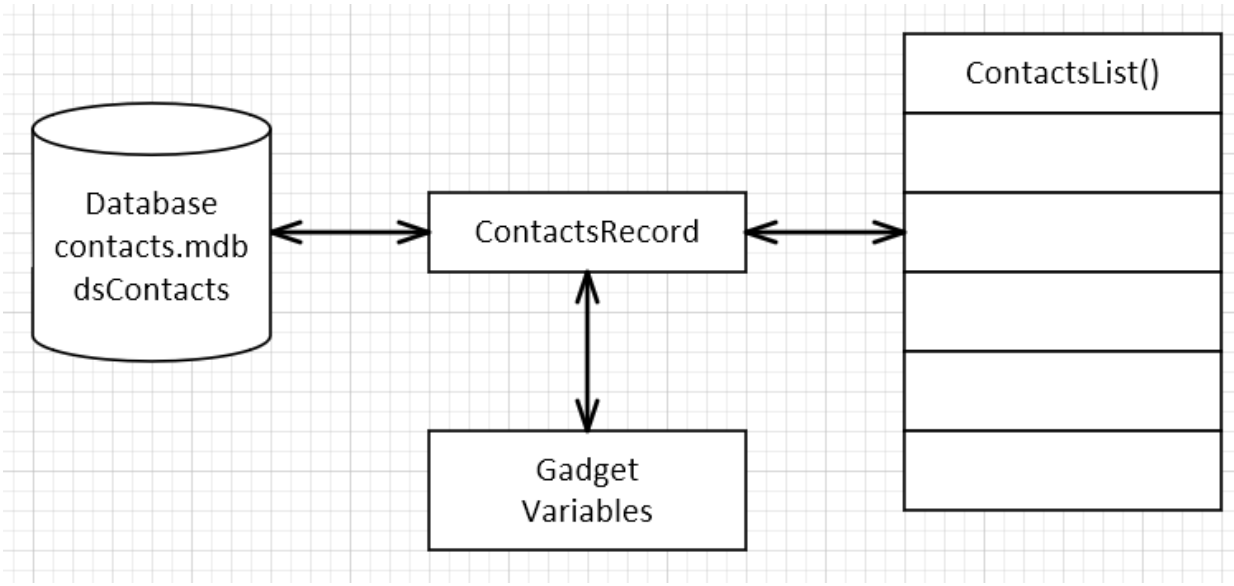

Figure 3 – Contacts Data Flow

The Project Manager

I'll emphasize a couple of things that may not be obvious at first. First, the fact that a file appears in the project manager window has no bearing on whether that file will be compiled or not. Files must be explicitly included in code using IncludeFile statements.

Second, all names must be defined before they can be referenced. This applies to both code and data. You can define a variable as global, but unless the compiler comes across the variable's definition in the source code *before* it comes across a reference to the variable, it won't work. This means that you have to include your code files in the right order—and by "include," I mean with IncludeFile statements, not merely included in the project's files.

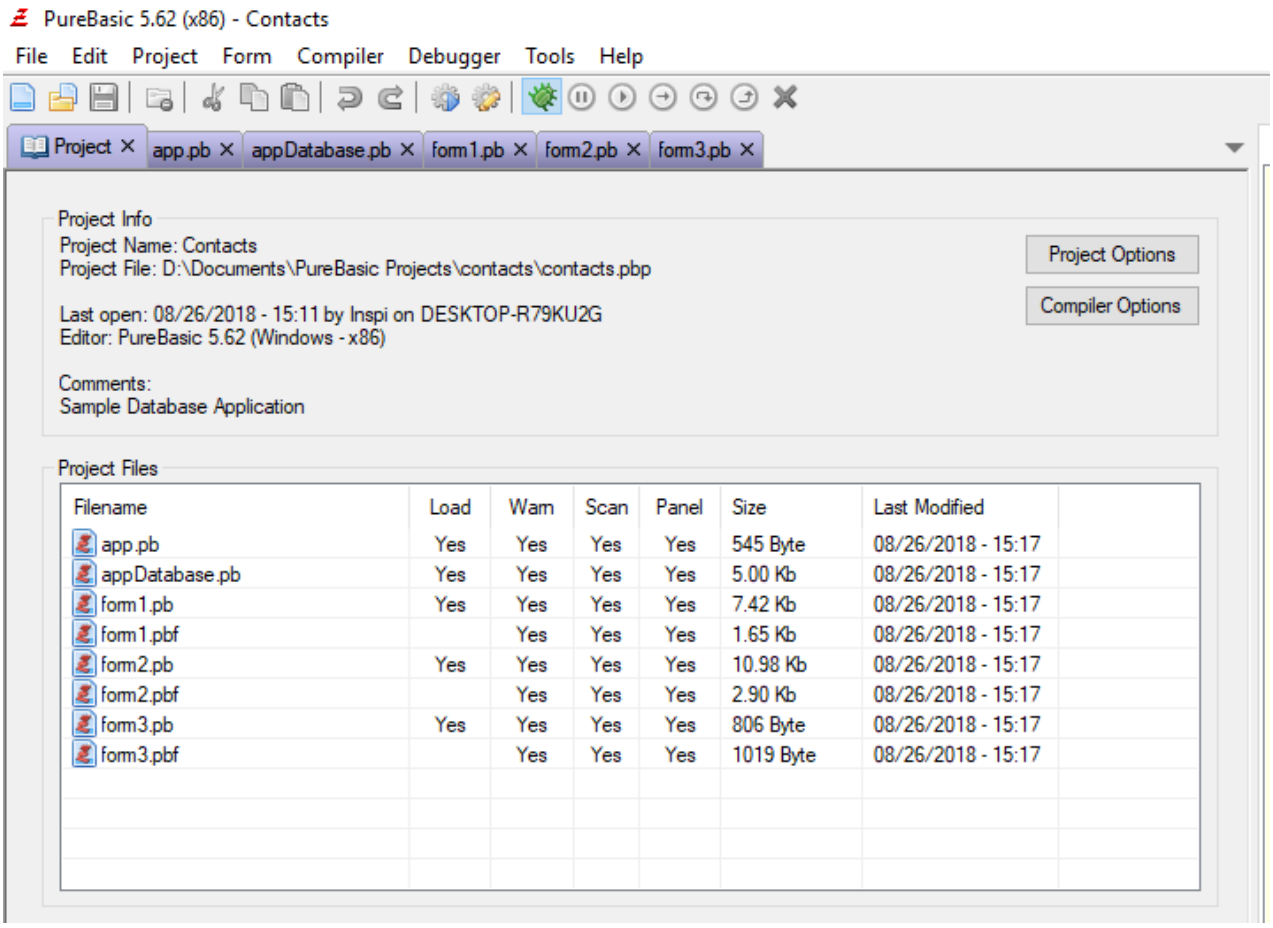

Figure 4 - Contacts in Project Manager

General Code Organization

When developing an application, the question of how to organize your code files comes up. Here's the method I use. It's simple and scalable. It starts with the observation that a GUI application consists of a collection of forms. Everything you do in a GUI application is carried out in one form or another. Therefore, under this method, every form has one and only one code file.

Aside from the collection of form files, every application needs a bootstrapping code stub that kicks things off, initializing the application and opening the main form's. I call this code stub app.pb. I call it this (instead of main.pb, for example), only so that it sorts at the top in the project manager.

Finally, I organize procedures that can be called from various forms into their own files and call them appSomething. For example, the file appDatabase.bp contains all the database procedures. I use the prefix "app" so that these files sort next to the main file app.pb.

Using the Form Designer

I mentioned the principle of one file per form, but there are clearly two files per form in the project manager. There's a form1.pb and a form1.pbf, for example. What's going on?

Form Designer is a very convenient tool to design and build forms. You lay out the form visually, and Form Designer generates a PureBasic file with the pbf extension. However, in the case of at least three gadgets (menus, tool bars, and status bars), the form designer allows you to create them but does not allow you to set any of their properties. This means that you can't finish designing your forms in Form Designer. You must finish them by tweaking the code produced by Form Designer. However, if you edit pbf files, Form Designer will revert your edits the next time it opens the files.

My workaround is to use Form Designer as usual to create my form. Then I take the generated code from the pbf file and paste it into the pb file, which is the actual code file. Then I tweak the code in the pb file to do whatever the form designer couldn't do. The pbf file just rides along in the project until the next time I need to edit the form, so as far as working code goes, it really is one file per form. Note that the pbf files are set not to load automatically in the project manager.

Naming Convention

Starting with file names, here is the naming convention I use. The main file, the file that launches the main form, is always called app.pb. Files that hold related thematic procedures called from various places are given descriptive names (e.g. Database, String, Encryption, etc.) and the prefix "app." This will make them sort following the app.pb file.

Form files are called "form" and numbered. The numbering can be sequential or hierarchical. For example, if your application's second form can itself open three other forms, you could call the second form "form20.pb," and the child forms "form21.pb," "form22.pb," and "form23.pb."

Within form files, all procedures and global variables are prefixed with the letter F and the form's number. Thus, the form handler for Form22 would be F22HandleForm. The form's variable would be F22Form, and a string gadget on the form that held a phone number would be called F22Phone.

PureBasic has a feature that lets you put your code in modules to avoid naming conflicts. If we did that, we wouldn't have to prefix our variables. But using modules is more complex, and I wanted to keep this simple. Perhaps a second sample application will use modules, or perhaps we can convert this application to use modules later on.

.

app.pb

This is the file that initializes the application and launches the main form. When the main form closes and control returns here, the End statement formally ends the application (the application would end without it anyway).

The first executable statement is EnableExplicit. I use it in all non-trivial projects.

Then there's a structure called FormsStructure, which we will use to track our open forms. We'll see how this works in the Managing Forms section.

Then there's a structure called AppDataStructure, where we keep application data. This structure might include more information in more complex applications, but we minimally store the database name and number, plus a map of FormStructures, which again, will be explained in the Managing Forms section.

After creating the AppData variable, we use IncludeFile to pull in the rest of the code, and here's an example of how the order of code matters. If the two included files are included *before* the global declaration of AppData, compilation will fail because the two included files refer to the global AppData variable.

Now, you know there are three forms, but only the file for one of them is being included. The explanation is this. In any application, there is a hierarchy of forms. Forms are opened by parent forms, and they themselves can open child forms. The practice, then, is to have each parent form include only its child forms. In the case of the main file, it includes only the main form, form1 in our case.

After setting the type of database we will use and setting the database name, we call the handler for the main form. Note that you can create constants in the project manager (Project Options, Constants tab), so if you put the database name in a constant, you can use app.pb without change from project to project.

form1.pb

Overview

This is a typical, fully developed form that has two child forms, a modal one and a modeless one, so let's look at its overall structure. It begins with procedure declarations, followed by file includes. Then come constant and variable definitions, and finally, all the procedures.

Recall the discussion on Form Designer and how we copied code from the pbf files it created. Here we see that code, bracketed between a couple of comments. For illustration purposes, I've documented the changes made to the code generated by Form Designer.

Note that there are two enumerations with the same name. The documentation mentions it, but it bears repeating that enumerations don't use names to distinguish one from another but rather to combine them. When compiling, the system will create a single enumeration from them all, in the order they were encountered.

The HandleForm Procedure

This is the procedure you call to use a form until the user closes it. The procedure has two sections. The first initializes the form, and the second handles the Event Loop.

Modal and Modeless Forms

You have encountered these types of forms even if you're not familiar with the terminology. A modal form is a form that seizes the application's focus, freezing out the rest of the application. Once you open a modal form, you can't do anything outside it until you close the form. An example of this is the Print dialog box in Word. As long as it's open, you can't do anything else in Word.

A modeless form, on the other hand, does not seize the application's focus. While a modeless form is open, you can click on other parts of the application and even carry out other tasks. An example of this is the Replace function in Word (Control-h). While the form is open, you can click on your document and make changes to it. The Replace form loses focus but does not go away.

Here's how we implement modal forms in this application. We disable the current form and call the next form's handler. The form being opened must have an event loop in its handler. Without an event loop, control returns to the calling function immediately, which is modeless behavior.

```
Procedure F1About()
    DisableWindow(F1Form, #True)
    F3HandleForm()
    DisableWindow(F1Form, #False)
    SetActiveWindow(F1Form)
EndProcedure
```

Figure 5 – Launching a form modally.

We see how to implement a modeless form in Figure 6. Because the handler for a modeless form has no event loop, calling the handler only initializes and displays the form. Because control returns immediately to the caller, code placed after the call to the handler executes immediately after the call. This is not usually the behavior you want. After the call to the handler, control must be passed back to this form's event loop, as this form's event loop handles events for both forms.

```
Procedure F1EditRecord()
  ContactsRecord\ID = Val(GetGadgetText(F1lstContacts))
  If ContactsRecord\ID > 0
    SelectElement(ContactsList(), GetGadgetState(F1lstContacts))
    GetContacts()
    F2HandleForm()
    ; control returns while Form 2 is still open
  EndIf
EndProcedure
```

Figure 6 – Launching a modeless form.

The Event Loop

The event loop is your form's control center. Its purpose is to monitor user actions and to respond to them appropriately. The PureBasic forums have some great tutorials on event loops in general to get you started.

In our implementation, we wait for a Windows event. Then we determine in what form the Windows event occurred. Having determined the form, we then determine whether the form was closed, or a choice was made in the form's menu or toolbar, or a gadget on the form was acted on in some way (e.g. click, double-click, change, etc.). Accordingly, we then execute the appropriate procedure.

The event loop can be confusing because of the various functions needed. Here's a summary of each function's purpose.

- WindowEvent() (with Wait) returns the Windows event that just happened.
- EventWindow() returns the form in which the event happened.
- EventMenu() returns the menu in which a choice was selected.
- EventGadget() returns the gadget that was acted on.
- EventType() returns the action carried out on the gadget (e.g. click, double-click, etc.)

In your event loop, you always execute WindowEvent(), but you only execute the other functions as needed. For example, you only execute EventMenu() if WindowEvent() indicates that a menu choice was made. If you execute EventMenu() when no menu choice was made, the results is zero.

Form Management

If our application allows multiple non-modal forms of the same type, we will need to keep track of the open forms and their variables. Here's why. Suppose Form1 opens Form2. When Form2 is created, its number is stored in a global variable (F2Form under our naming convention). Now suppose Form1 opens a *second* Form2 with data from a different record. The code will again store Form2's number in the variable F2Form, overwriting the *other* Form2's number. The form will still be visible on the screen, and the operating system won't lose track of it, but we won't be able to access it.

```
Structure FormsStructure
  Name.s
  List Gadgets.i()
EndStructure

later...

Map Forms.FormsStructure()
```

Figure 7 – Forms Structure (app.pb)

To keep track of our open forms, we need a structure to hold their information. See Figure 7 for how to implement this in code. FormsStructure holds the information for one form. There's a string variable to hold the form's name (e.g. form1, form2, etc.), and there's a list of integers to hold all the form's gadget numbers. A list is the most appropriate type for this because the number of gadgets is variable, and because we don't need random access to them.

```
; code from event loop in F1HandleForm
  OpenF1Form()
  AddMapElement(AppData\Forms(), Str(F1Form))
  F1StoreForm()

Procedure F1StoreForm()
  AppData\Forms()\Name = "form1"
  AddElement(AppData\Forms()\Gadgets())
  AppData\Forms()\Gadgets() = F1Menu
  AddElement(AppData\Forms()\Gadgets())
  AppData\Forms()\Gadgets() = F1ToolBar
  AddElement(AppData\Forms()\Gadgets())
  AppData\Forms()\Gadgets() = F1StatusBar
  AddElement(AppData\Forms()\Gadgets())
  AppData\Forms()\Gadgets() = F1lstContacts
EndProcedure
```

Figure 8 – Saving an open form (Form1.pb)

Having defined the structure, we then create a map of elements of that structure. Since any open non-modal form can receive focus, we do need random access, and thus a map is the most appropriate type of structure.

Figure 8 shows how we track forms after opening them (See the form handler's event loop). The OpenF1Form procedure stores the form's number in the global variable F1form. We then add an element to the Forms map, using the string conversion of the form's number stored in F1Form. This only creates an empty map element, of course, so we then load the element in the F1StoreForm procedure.

We've stored the information for every form we've opened. Now what? Now suppose one of those forms receives focus because the user clicked on it. Figure 9 shows how to proceed. The EventWindow procedure returns the active form's number, which we then use to find that form's entry in the Forms structure. All this happens in the form's event loop.

Having found the form's entry in the Forms map, we then determine which form it is and restore the form's gadget variables.

```
EventWin = EventWindow()
FindMapElement(AppData\Forms(), Str(EventWin))
Select AppData\Forms()\Name
  Case "form1"
    F1form = EventWin
    F1RestoreForm()
    Select WinEvent

Procedure F1RestoreForm()
  ResetList(AppData\Forms()\Gadgets())
  NextElement (AppData\Forms()\Gadgets())
  F1Menu = AppData\Forms()\Gadgets()
  NextElement (AppData\Forms()\Gadgets())
  F1ToolBar = AppData\Forms()\Gadgets()
  NextElement (AppData\Forms()\Gadgets())
  F1StatusBar = AppData\Forms()\Gadgets()
  NextElement (AppData\Forms()\Gadgets())
  F1lstContacts = AppData\Forms()\Gadgets()
EndProcedure
```

Figure 9 – Restoring a saved form (Form1.pb)

Note that, until now, we have only added forms to the Forms map. When a form receives focus, we restore its values from the Forms map, but we don't delete the map entry. This is only logical, since we can switch between forms an unlimited number of times. We only delete a form from

our Forms map when we close the form. You can see this in Figure 10, deep in the event loop. With one line, we delete the map entry, and that concludes the form management cycle.

```
Case "form2"
 Shared F2CurrentMode
 F2form = EventWin
 F2RestoreForm()
 Select WinEvent
   Case #PB_Event_CloseWindow
     DeleteMapElement(AppData\Forms(), Str(F2form))
     CloseWindow(F2form)
   Case #PB_Event_Menu
```

Figure 10 – Deleting a saved form (Form1.pb)

form2.pb

Overview

The file has a layout similar to form1.pb. The differences are that it includes navigation procedures that the first form did not need because it displays a list of records, and that the HandleForm procedure lacks an event loop because this form's events are handled by its parent's event loop.

Navigation

PureBasic's database navigation functions pair up nicely with standard navigation movement. There's a function to move to the first record, the last record, the next record, etc. Building a single procedure that handles all navigation is pretty easy, as you can see in the code listing. We implement wraparound by testing attempts to move to the next or previous record. If they return zero, it means there's no next or previous record, so we must be at the end or the start of the database.

After moving to the correct record, we copy it to the current record structure ContactsRecord and display it on the screen.

Form Operation Modes

Depending on what the user is doing, toolbar buttons may need to be disabled and keys may have to have different functions. For example, you shouldn't be able to move away from a record that you are editing, so while you're editing or entering a new record, the navigation buttons must be disabled. Another example is the Delete key. While navigating, the Delete key deletes the current record. But if you're editing the current record, the Delete key should have its usual function of deleting characters. To handle this, we define a Data Entry Mode for the case where the user is editing (new or existing record), and a Navigation Mode for the case where the user is browsing.

To implement these two modes, we have the F2SetMode procedure and the shared variable F2CurrentMode. For more readable code, an enumeration initializes several constants starting

with #F2Mode. Whenever the form is loaded or reloaded with information, it's set to Navigation Mode. The form can then be set to Data Entry mode through three actions. One is to enter the form by pressing the Tab key. This will give focus to the first field, and you'll be in Data Entry Mode. The second time occurs in F2AddRecord. Although the call to F2FillForm sets the form to Navigation Mode, Data Entry Mode is set specifically immediately after. Finally, the third time is in F2SaveRecord.

Inserting and Updating

When we save a record, we need to know whether we're saving a new record or updating an existing one because the SQL for each is different. To keep track of this, the shared variable F2isAdding is set to #True or #False. It's set to #True in the F2AddRecord procedure and set to #False in the F2SaveRecord and F2CancelEdit procedures.

Cancelling Form Editing

When the user enters a new record, the software starts with a blank form. If the user then cancels entering the new record, it would be desirable to return to form to the state it was before. This is true as well when the user edits an existing record. To achieve this, we save the current record ContactsRecord in ContactsBackup whenever we enter Data Entry Mode, and if the user cancels editing, we restore the record in F2CancelEdit.

appDatabase.pb

Introduction

Here we gather the global data structures and the procedures we use with the database. The procedure DB2Record copies the current record from the database to ContactsRecord structure. The procedure List2Record does the same from ContactsList, which acts as an in-memory database. You can examine the other procedures. It's all very basic because the application is so simple. In more complex projects, you would have many more procedures.

You can also use this technique to organize groups of procedures along a theme. If you have a set of string processing procedures that could be called from various places, place them all in a file, call it appStrings.pb, and just make sure to include them in app.pb.

Macros

Here's an opportunity to use the macro feature. When binding variables to database fields, empty strings must be bound with SetDatabaseNull, not SetDatabaseString. This means every string variable has to be tested before binding. That leads to a lot of code, but we can use a macro to avoid that.

See Figure 11. A macro is defined and called very much like a procedure. This kind of macro doesn't record keystrokes. Instead, it substitutes text within the source code itself as the compiler parses the source code. The compiler takes each BindStringVar it encounters and replaces it with the code in the macro, replacing as well the argument placeholders with the supplied arguments.

It's important to understand that macros replace *text*, not code. That's why we can pass the field names to the macro. Without qualifying them, the field names make no sense. The name WebURL, for example, doesn't exist anywhere. It only exists in the context of ContactsRecord. That is to say, the compiler recognizes ContactsRecord\WebURL, but not WebURL alone. And yet the macro works. Why? Because we're not passing WebURL as a variable. We are passing it simply as a string of text. The macro is expanded *before* compiling takes place.

```
; without macros, we'd have to use this code for each field
; If ContactsRecord\FullName = ""
;   SetDatabaseNull(AppDatabase, 0)
; Else
;   SetDatabaseString(Appdata\Database, 0, ContactsRecord\FullName)
; EndIf

Macro BindStringVar(Ndx, Field)
  If ContactsRecord\Field = ""
    SetDatabaseNull(AppData\Database, Ndx)
  Else
    SetDatabaseString(AppData\Database, Ndx, ContactsRecord\Field)
  EndIf
EndMacro

Procedure SetBoundVars()
  BindStringVar(0, FullName)
  BindStringVar(1, Phone)
  BindStringVar(2, Email)
  BindStringVar(3, WebURL)
EndProcedure
```

Figure 11 – Macros (appDatabase.pb)

Conclusion

This ends the code discussion and walkthrough. There are many ways of arranging your code and building projects. I've presented one that I've developed and that makes sense and works for me. I hope you find some of it useful and can use some of the techniques presented here.

I hope you agree that good code is readable and understandable as well as being fast, efficient, and robust. Good code is all of those things. If you spot an improvement to this document or to the software it discusses, please let me know. Thank you!