

POL TUTORIAL

Contents

Object Library for Palm OS ® platforms tutorial.....	2
Step 1: Creating starter application	3
Step 2: Adding a grid control.....	13
Step 3: Adding modal form	23
Step 4: Modifying data.....	35
Step 5: Adding search functionality.....	39
Step 6: Integration with global find service	43

Object Library for Palm OS ® platforms tutorial

This tutorial will guide you through creation of simple Palm OS ® application using Object Library for Palm OS ® platforms (briefly POL). Let it be a small address book application. Though small size of source code this application will store all information in Palm database and provide user with simple interface for editing address entries. At the end of this tutorial you'll know enough to use POL in your own applications.

Note:

The tutorial assumes that:

- a) You have Metrowerks CodeWarrior 9.0 properly installed on your computer.
- b) You have POL 4.0 or newer properly installed on your computer.
- c) You know basic elements of Palm OS API.
- d) You know C++ programming language.

Step 1: Creating starter application

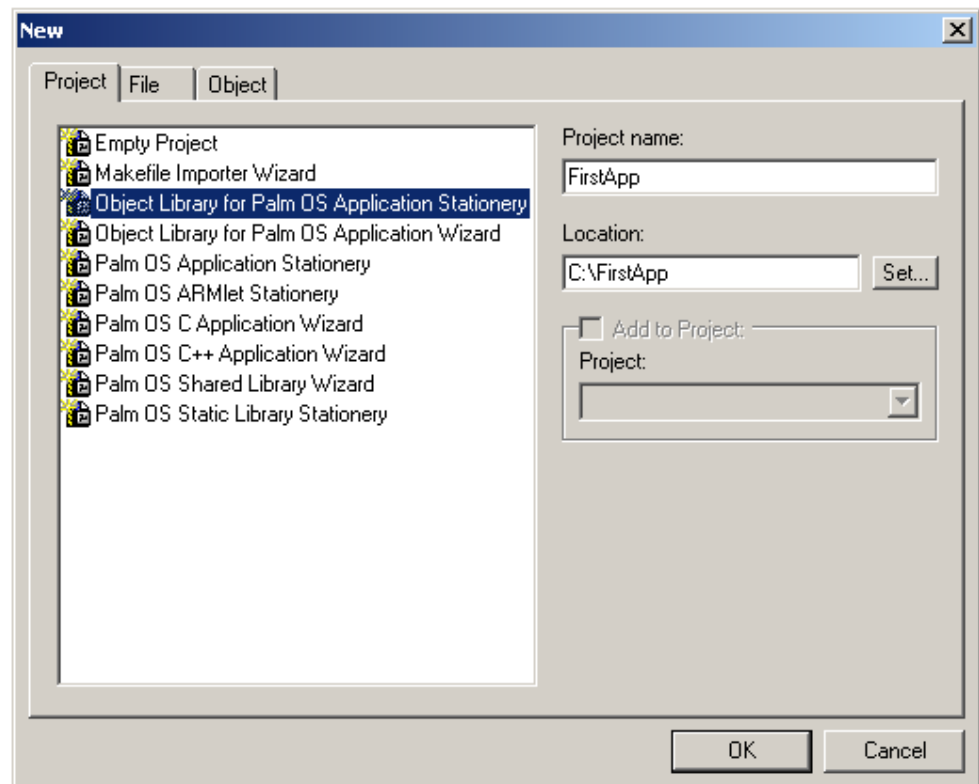
You can find necessary project files for this step in **Step 1** folder.

Before starting coding you should create a starter application. It's a typical application with minimal functionality. It consists of predefined project structure, the most common lines of code and resources. To create our first application use steps listed below:

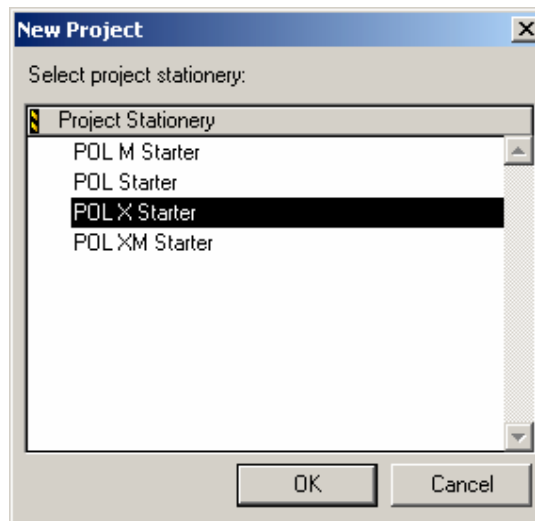
1. Launch CodeWarrior IDE from Window Start menu.

Note: You must configure Metrowerks CodeWarrior IDE to start Palm OS Emulator with IDE simultaneously to complete instruction steps normally. For more details please see *"Targeting the Palm OS ® Platform"* documentation.

2. Select **New** command from **File** menu in Metrowerks CodeWarrior.
3. Choose Object Library for Palm Stationery in **New** dialog in **Project** tab.
4. Type **FirstApp** in **Project name** field
5. Specify project location in the **Location** field (for example *C:\FirstApp*).
6. Press OK button.



7. The next dialog shows all available POL stationeries:

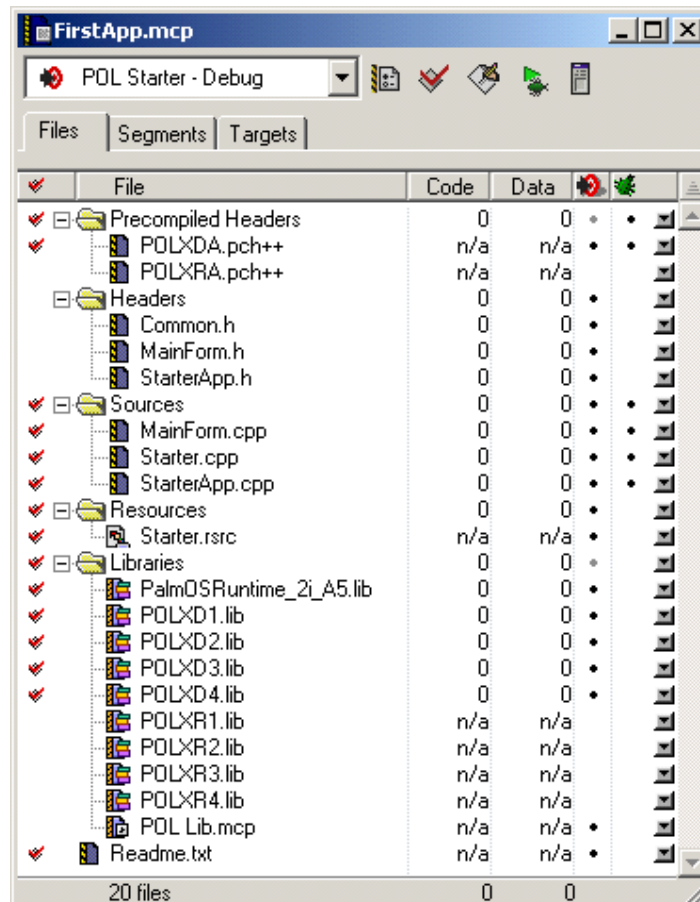


The following POL stationeries are available:

Project stationery	C++ exceptions support	CW expanded mode support
POL Starter	Disabled	Disabled
POL X Starter	Enabled	Disabled
POL M Starter	Disabled	Enabled
POL XM Starter	Enabled	Enabled

Select **POL X Starter** stationery and press OK button. CodeWarrior will create new project for you.

- Now let's look into our project. Open all folders in project window as shown on the picture below:



As you can see, start project contains a lot of files. It's at least more than single C-file with `PilotMain()` function. Such approach is very useful for large and complex C++ projects because large and complex code is divided by number of small and independent files. The short description of all project files is listed below:

"Precompiled Headers" folder contains header files with required directives for C++ compiler and POL. The content of this folder depends on current POL target. You should not change these files. Just ignore them.

"Headers" and *"Sources"* folders provide us with the most interesting number of files. These folders contain all source code of our project. Class declarations are stored in *"Headers"* group and class implementations are stored in *"Sources"* group.

"Resources" folder contains application resources. In this simple example it's single file *"Starter.rsrc"*.

"Libraries" folder contains all libraries used by project. Additionally this folder contains link to POL library project.

At the end of the *"Files"* list you can see *"Readme.txt"* file. This file is a typical *"Read me"* document for this project. You may want to maintain this file if you like to document and keep project notes.

Now you can take a closer look on all these files.

9. Application resources are the best place to see our application before executing. Please open application resources by double-clicking on *"Starter.rsrc"* file in project window. *"Constructor for Palm OS"* window will be opened. Please review *"Starter.rsrc"* window. This window contains all resources used in our application.

Initially our application consists of 3 forms:

- a) *Main* form will be shown on the screen after application start. *Main* form has its own *Main* menu bar. This menu bar is used for opening *About* and *Options* forms.
- b) *Options* form can be used later if you want to provide some additional customization for your application.
- c) *About* form will display your information and copyright notes.

This project has additional resources, such as bitmaps and pull-down menus. Select **Exit** command from **File** menu in *"Constructor for Palm OS"* window. Let's take a look on the other project components.

10. Please open *"Common.h"* file in *"Headers"* group by double-clicking it in project window. This file is included into every CPP-file of our project. Modify this file if you want to use other libraries or want to declare common macros. Close *"Common.h"* window.
11. Please open *"Starter.cpp"* file in *"Sources"* group. This file is an entry point of application (our program starts from this file). But hey! Where is `PilotMain()` function? It's hidden from you. POL hides all non-object paradigms (such as global functions or global variables) from you. Instead, this file contains `DEFAULT_STARTER` macro. This macro provides POL with information about our application class. In our case it's `CStarterApp`

class. An application class takes control when user starts our application. Additionally, it takes control in some another situations (for example it's responsible for processing searching action). Close "*Starter.cpp*" window.

12. Open "*StarterApp.h*" file in "*Headers*" group. This file declares our application class (CStarterApp class). This class is derived from CPalmApp class. CPalmApp class is a base application class supplied by POL. It performs some background work and calls your functions in response to some events.

In this example, CStarterApp simply overrides NormalLaunch() virtual function. This function is inherited from CPalmApp class and it is called when user starts our application from Palm OS Application Launcher. So, this function is our entry point for sysAppLaunchCmdNormalLaunch launch code.

13. Let's look into implementation of NormalLaunch() function. For this reason switch to project window and open "*StarterApp.cpp*" file (please don't close "*StarterApp.h*" window). This file contains only implementation of CStarterApp::NormalLaunch() method. NormalLaunch() simply opens *Main* form on the screen and returns zero error code to indicate successfully completed operation. Please note, that CForm::GotoForm() function is used instead of FrmGotoForm() API equivalent. CForm is yet another POL class used for working with forms. You'll know more about this class later.

Now it's more important to look on the single parameter of GotoForm() function. It's an ID of form resource to be loaded. MainForm is passed to this parameter because *Main* form is a start form of our application.

There exists one important question: how does POL know what form procedure (or possible C++ class) should handle user input and various events for this form? GotoForm() method doesn't provide POL with such information. The next paragraph will explain it. But now simply close "*StarterApp.cpp*" window and switch to previously opened "*StarterApp.h*" window.

14. You already know about role of application class. In this paragraph you will learn a new functionality of this class. Please take a notice of lines between BEGIN_FORM_MAP() and END_FORM_MAP() macros.

This group of macros has its own name: *form map*. It's an answer to the question from the previous paragraph. Form map binds form resources to C++ form classes. In our case CMainForm class will receive and handle all events while form with resource ID MainForm is active. POL will automatically create instance of CMainForm class in response to frmLoadEvent event for *Main* form.

Please close "*StarterApp.h*" window. It's good time to learn more about form classes in POL.

15. CForm is central class of POL framework. It's responsible for solving two tasks in every form-based Palm OS application:

- a) It provides high-level interface to Palm OS API. So, you can use CForm methods rather than form-API functions.

- b) It contains event-routing facilities. This major functionality of `CForm` class allows you to handle Palm OS events in an extremely simple way.

The last statement should be explained in more details.

As you know, every form-based Palm OS application must provide form-procedure almost for every form resource. Form procedure determines application behavior while corresponding form is active.

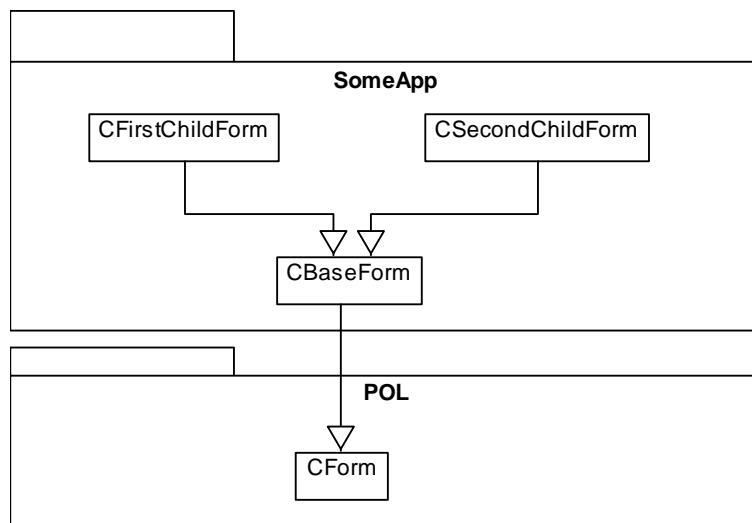
Form is active if it is shown on the screen and it has a focus (i.e. handles all key and pen input). Palm OS sends all key and pen events to the active form procedure. Form controls also send command events to the active form procedure (for example, button control will send you `ctlSelectEvent` if user taps button using stylus).

Traditional form procedure looks like giant `switch`-operator of C programming language. The `switch`-operator represents every Palm OS event handled by form procedure through one of `case`-statements. This practice is very difficult and tedious. Just imagine such procedure for typical form with few tables and several buttons!

Event-routing facilities of `CForm` class are intended to solve this problem in POL. You don't need to write complex form procedures. POL provides you with simple solution of the problem:

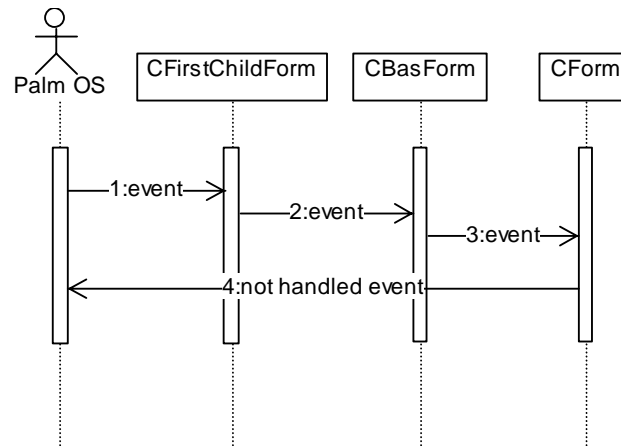
- a) Instead of writing form procedures you will inherit your form classes from `CForm` class supplied by POL.
- b) Instead of writing large `switch`-operators you will declare *event maps* in your form classes. Event maps will route Palm OS events among methods in hierarchy of form classes.

An event routing is specific for object-oriented frameworks. Please look at the picture below:



This picture represents sample hierarchy of form classes in some application. The sample application has two forms: “*First Child Form*” and “*Second Child Form*”. Every form is represented by corresponding form class. All common code of these forms is located in `CBaseForm` class. `CBaseForm` class also inherits all functionality of `CForm` class supplied by POL.

Every event for “*First Child Form*” is processed in the following way:



Superior form class of hierarchy (CFirstChildForm class in this example) receives first chance to handle an event. The event will be sent to base form class (CBaseForm class in this example) if superior class did not handle it. An event passes through the base classes until it will be handled or a root of form hierarchy (always CForm class) will be reached. The framework will return not handled events back to Palm OS.

As you suspect, “*Second Child Form*” handles events by the same way.

A conclusion:

- a) Events are sent in direction from the child to parent form classes.
- b) Child class controls events processing. It can bypass events to base class.

These rules allow you to create mature hierarchies of form classes. However in practice you will inherit your form classes directly from CForm or CModalForm as shown in the example below.

```

// Main form class
class CMainForm : public CForm
{
public:
    // Event handlers
    Boolean OnOpen(EventPtr pEvent, Boolean& bHandled);
    Boolean OnHello(EventPtr pEvent, Boolean& bHandled);

    // Event map
    BEGIN_EVENT_MAP(CForm)
        EVENT_MAP_ENTRY(frmOpenEvent, OnOpen)
        EVENT_MAP_COMMAND_ENTRY(MainHelloButton, OnHello)
    END_EVENT_MAP()
};
  
```

The lines between BEGIN_EVENT_MAP and END_EVENT_MAP macros are an example of *event map* mentioned above. An event map provides POL with the following information:

- a) It specifies the list of events handled by the form class (frmOpenEvent and ctlSelectEvent events in this example). Every event appears to be automatically bound to corresponding handler (OnOpen() and OnHello() methods in this example).
- b) It provides POL with information about form hierarchy. Due to this information events can be sent from child to parent class. The

information is passed via name of base form class (CForm class in this example) as a parameter to BEGIN_EVENT_MAP macro.

- c) There exist several types of event map entries (i.e. EVENT_MAP_ENTRY and EVENT_MAP_COMMAND_ENTRY). The EVENT_MAP_ENTRY is the most common type of entry: it takes the identifier of handled event through the first parameter. Macros like EVENT_MAP_COMMAND_ENTRY are more specific. They serve to simplify handling specific events. For example, EVENT_MAP_COMMAND_ENTRY macro handles ctlSelectEvent event only. It takes control ID through the first parameter. It's useful if you have form with several buttons and you want to provide separate event handler for every button:

```
BEGIN_EVENT_MAP(CForm)
    EVENT_MAP_COMMAND_ENTRY(MainFirstButton, OnFirst)
    EVENT_MAP_COMMAND_ENTRY(MainSecondButton, OnSecond)
    EVENT_MAP_COMMAND_ENTRY(MainThirdButton, OnThird)
END_EVENT_MAP()
```

All event handlers must have the same prototype:

```
Boolean <Event Handler Name>(EventPtr pEvent, Boolean& bHandled)
```

For example, the following event handler displays Hello alert in response to click on Hello button in our form:

```
Boolean CMainForm::OnHello(EventPtr pEvent, Boolean& bHandled)
{
    FrmAlert(HelloAlert);
    return true;
}
```

The value returned from an event handler will be passed to Palm OS. The meaning of this result value depends on event type. Please note that the base form class can handle the same event too. The event handler of the child form overrides the same event handler of the base form class.

The first parameter of event handler contains additional information about processed event. The framework receives this information directly from Palm OS. This additional information can be accessed in a standard way (i.e. pEvent->data.frmOpen in case of frmOpenEvent) because type of the event is known inside of the event handler function.

The second parameter is specific for POL framework. Child classes control event processing in parent classes through this parameter. The framework assumes that the event was handled by the function if bHandled variable contains true value upon function return. Unhandled events are sent to base form classes. Properly handled event bypasses form hierarchy. Because this functionality is more appropriate for almost all the events, bHandled variable contains true value by default. You can override this default behavior by assigning false value to bHandled variable as shown in the example below:

```
Boolean CMainForm::OnOpen(EventPtr pEvent, Boolean& bHandled)
{
    // Some code...
    bHandled = false;
    return false;
}
```

Please note that the handler of frmOpenEvent event is the best example for bHandled action because this event is handled by base CForm class

too. The handler of this event in `CForm` class calls `FrmDrawForm()` API function to display the form on the screen. If you'll try to comment assignment `false` value to `bHandled` variable then `CForm` handler will not be called and the form will not shown on the screen. So, you must explicitly set `bHandled` to `false` value in this example. Alternatively you can manually call `FrmDrawForm()` API function.

Well, now you know many details about event maps. Please open "*MainForm.h*" file in project window to see it again in our application.

This file contains declaration of `CMainForm` class. This class represents *Main* form of our application. The declaration consists of event map and prototypes of event handlers. Currently *Main* form has no buttons, but it has a menu. Hence control events will be generated when user selects menu items in this form. Three menu items are accessible from *Main* menu bar:

- Tools ⇒ Options
- Tools ⇒ Exit
- Help ⇒ About

`CMainForm` class provides three corresponding menu handlers:

```
Boolean OnOptions(EventPtr pEvent, Boolean& bHandled);
Boolean OnExit(EventPtr pEvent, Boolean& bHandled);
Boolean OnAbout(EventPtr pEvent, Boolean& bHandled);
```

These handlers are mapped to menu identifiers through yet another type of event entry: `EVENT_MAP_MENU_ENTRY`. This macro handles `menuEvent` event.

Please close "*MainForm.h*" window.

16. Let's look on implementations of these event handles. Open "*MainForm.cpp*" file in project window. There you can see short description of every event handler:

OnOptions() handler

This handler simply displays *Options* dialog. You can later customize this dialog or remove it.

This code is interesting because it shows a new feature of `CForm` class. It can display modal dialogs without providing corresponding form class. It's very useful if you want to display simple dialog with several controls. This dialog will allow user to input some data and it will be automatically closed after pressing any button. This approach is ideal for *Options* dialog.

If you want to know more about this functionality please look for `FrmDoDialog()` API function in "*Palm OS ® Reference*".

OnExit() handler

This handler calls `Stop()` method of `CPalmApp` class. This method ends execution of our application.

OnAbout() handler

This handler displays *About* dialog. The dialog is displayed in the same way, as *Options* dialog was shown on the screen. The dialog will be automatically closed after user presses OK button.

As you can see, `CForm::DoDialog()` is very useful function for simple dialogs. Unfortunately it doesn't provide a way for displaying fully controlled modal forms on the screen. You will see later how to achieve it using POL.

That's all about event maps. Please close "*MainForm.cpp*" window.

17. You've almost finished first step. You have already seen all C++ code of this starter application. There exists only one unknown folder "*Libraries*" in the project window.

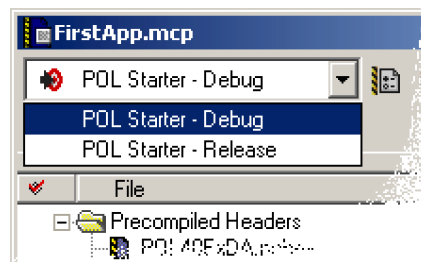
This folder contains libraries used by starter application:

"*PalmOSRuntime_2i_A5.lib*" is a standard runtime library file.

The following 6 files are POL-specific library files. Their names depend on chosen settings and target of the project. It is not necessary to know what exactly every name means for this tutorial, but general information about project structure can be interesting and useful.

POL stationery projects are already preconfigured and consist of two targets: *Debug* and *Release*.

You can switch project targets directly in project window:



Debug and *Release* targets use different compiler and linker options. Additionally these targets include different POL library files. *POLXD(1,2,3,4).lib* libraries are included into *Debug* target. *Release* target includes *POLXR(1,2,3,4).lib* files. As you can see, *D* letter in library name indicates *Debug* target and *R* button indicates *Release* target.

The project includes three POL library files for every target (*POLXD(1,2,3,4).lib* and *POLXR(1,2,3,4).lib*). It follows from the limitation of code segment size in Palm OS. Library files indexed with 1 are included into first segment. 2 in the file indicates second code segment and so on. It means that you've created a multi-segment application, so it can grow almost without any limitations.

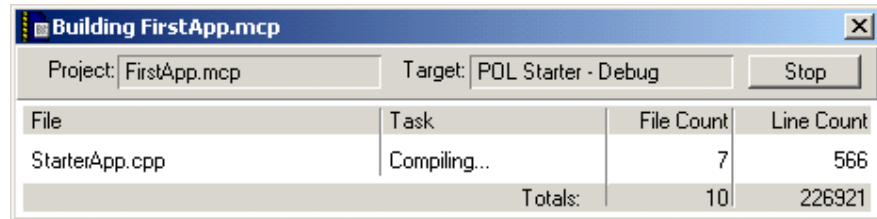
"*POL Lib.mcp*" is a reference to "*POL Library*" project you can use to view POL source code. It can be very helpful during debugging your applications. If you double-click on this item, "*POL Library*" project will be opened in CodeWarrior IDE.

18. Yep! We can run our application.

Note: You must properly configure Metrowerks CodeWarrior IDE to start Palm OS Emulator automatically to complete this step. For more details please see "*Targeting the Palm OS ® Platform*" documentation.

Select **Debug** command from **Metrowerks CodeWarrior / Project** menu or simply press **F5** key.

You'll see build progress on the screen:



Building FirstApp.mcp			
Project: FirstApp.mcp		Target: POL Starter - Debug	Stop
File	Task	File Count	Line Count
StarterApp.cpp	Compiling...	7	566
Totals:		10	226921

After compiling and linking our application will be executed on Palm OS Emulator. The final picture may differ depend on ROM-file you are using:



19. Try to select various menu commands to see how it works.
20. To stop the debugging select **Kill** command from **Metrowerks CodeWarrior / Debug** menu or simply press **Shift+F5** keys.

Congratulations! You've completed the first step.

Note: if you've finished your current work, please:

- a) Select **File / Exit** menu command in Metrowerks CodeWarrior application.
- b) Right click on Palm OS Emulator and select **Exit** menu item.

Step 2: Adding a grid control

You can find necessary project files for this step in **Step 2** folder.

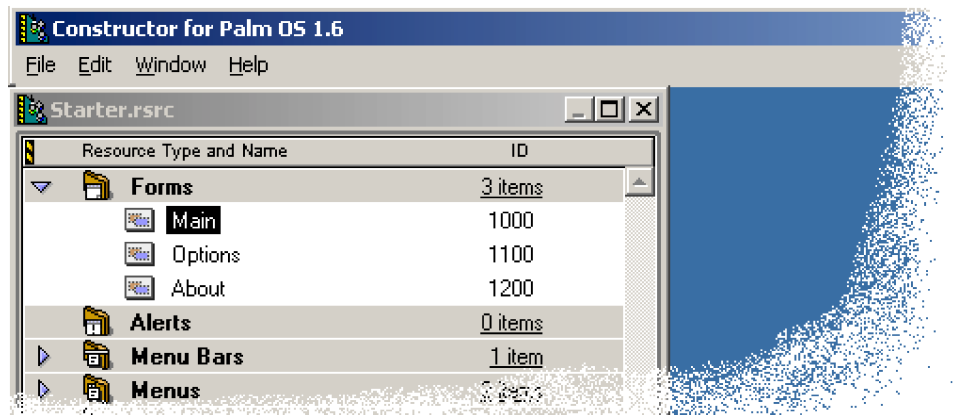
In this chapter we will add a lot of functionality to our application: it will read address entries from database and display data in the grid.

Note: if you start this step as separate step you need preliminary:

- a) Launch CodeWarrior IDE from Window Start menu.
- b) Select **Open** command from **File** menu in Metrowerks CodeWarrior IDE and select **FirstApp.mcp** project (this tutorial assumes that it's located in *C:\FirstApp* folder).

If you continue executing previous step, please follow the instruction below.

1. We'll start this chapter with modifying application resources. Please open application resources (it's "*Starter.rsrc*" file in project window). "*Constructor for Palm OS*" window must be shown.
2. Double-click *Forms/Main* form resource in "*Starter.rsrc*" window: Form editor window must be shown.



3. Fill in "*Form Title*" attribute with "*First App*" value.
4. Select label with "*Place your controls here*" caption in the *Main* form.
5. Press **Delete** key. The label must be removed from the *Main* form.
6. Press **Ctrl+Y** key combination or select **Catalog** menu item from the **Window** menu. The *Catalog* window must be shown.
7. Drag **Label** UI object from *Catalog* window to *Main* form.
8. Assign the following attributes to the new label:

Attribute Name	Attribute Value
Object Identifier	Name
Text	Name
Left Origin	2
Top Origin	18

9. Drag **Label** UI object from *Catalog* window to *Main* form.
10. Assign the following attributes to the new label:

Attribute Name	Attribute Value
Object Identifier	Phone

Attribute Name	Attribute Value
Text	Phone
Left Origin	82
Top Origin	18

11. Drag **Table** UI object from *Catalog* window to *Main* form.

12. Assign the following attributes to new table:

Attribute Name	Attribute Value
Object Identifier	Address
Left Origin	2
Top Origin	30
Width	148
Height	110
Rows	10

13. Select “*Column Widths*” attribute for the table.

Press **Ctrl+K** keys or select **New Column Width** command from **Edit** menu. The second “*Column Width*” item must be appeared in *Column Widths* list.

14. Select “*Column Width 1*” item and set it to 80.

15. Select “*Column Width 2*” item and set it to 68.

16. Drag **Scrollbar** UI object from *Catalog* window to *Main* form.

17. Assign the following attributes to the new scrollbar:

Attribute Name	Attribute Value
Object Identifier	Address
Left Origin	151
Top Origin	30
Height	110

18. Press **Ctrl+S** key or select **Save** menu item from **File** menu.

19. Select **Exit** command from **File** menu. “*Constructor for Palm OS*” window must be closed.

20. Palm OS table manager provides flexible API for displaying tabular data and controls. We will display a list of addresses in *Address* table, which was early added to *Main* form resource. Palm OS table manager doesn't provide automatic table scrolling. Application developer is responsible for scrolling tables and synchronizing scrollbar control with table content.

POL applications can use high-level `CGrid` class (or one of its descendants) for working with table manager. `CGrid` class can automatically maintain grid data and synchronize associated table scrollbar.

Additionally you can bind grid object to application database using `CDBGrid` class. Bound to database grid automatically maintains record indexes.

21. Open “*MainForm.h*” file from project window.

22. Add the following included before `CMainForm` class declaration:

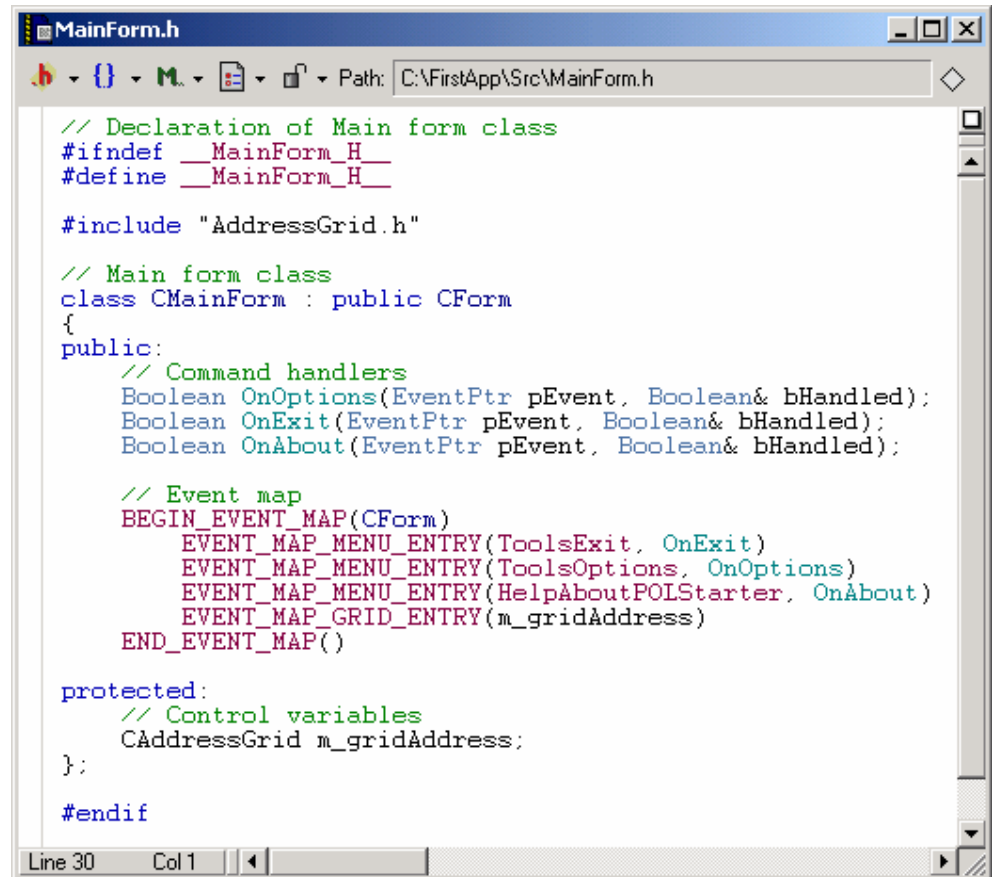
```
#include "AddressGrid.h"
```

“AddressGrid.h” file will be created later in this chapter. This file will include declaration of CAddressGrid class. CAddressGrid class will display scrollable table (grid) of addresses on the screen. This class will be inherited from CDBGrid class. CDBGrid class is base class for database bound grid controls provided by POL.

23. Add an instance of CAddressGrid class to main form class by putting the following lines to the end of CMainForm class declaration:

```
protected:
    // Control variables
    CAddressGrid m_gridAddress;
```

CMainForm class should have a look like a picture below:



Though we have already a grid variable in our form class, this variable is not attached neither to table resource, nor to scrollbar resource and nor to database with address entries on Palm. All these tasks are required for correct work. So we will add database variable to our application class.

24. Save your changes by pressing **Ctrl+S** key combination and close “MainForm.h” window.
25. POL contains a lot of low-level and high-level classes for accessing data entries of Palm OS data and resource manager. For example, POL has a CDatabase class; this class represents standard Palm OS database. In our example, we will add CDatabase variable to our application class. This variable will represent real database with address entries.
26. Open “StarterApp.h” file in project window. Add the following lines to the end of CStarterApp class declaration:


```
protected:
    CDatabase m_Database;
```

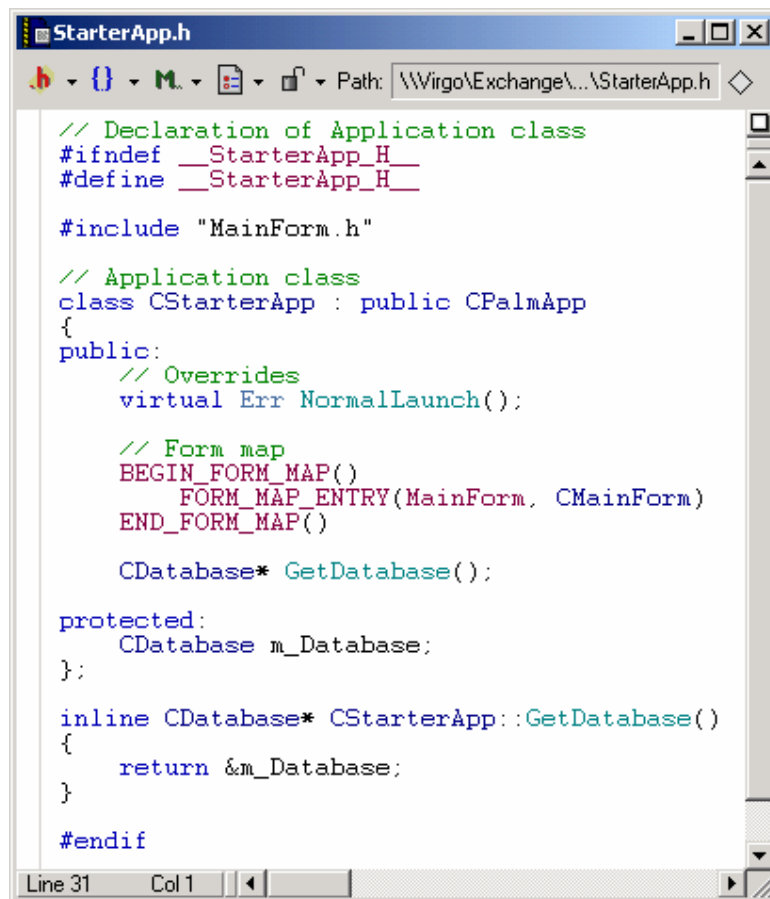
27. Add declaration of corresponding get-method to public section of class declaration for providing access to database variable from external classes:

```
CDatabase* GetDatabase();
```

28. Add implementation of GetDatabase() method below CStarterApp class declaration:

```
inline CDatabase* CStarterApp::GetDatabase()
{
    return &m_Database;
}
```

Finally, your application class should have a look like the picture below:



29. Save your changes by pressing **Ctrl+S** key combination or selecting **Save** menu item from **File** menu.

30. Close "StarterApp.h" window.

31. Though application class has already CDatabase variable, this variable should be attached to real database on Palm device. We will add database initialization code into NormalLaunch() method. It's a convenient place for database initialization because this method will be called every time user starts our application from Palm OS Application Launcher.

32. Open "StarterApp.cpp" file. Add the following lines to the beginning of NormalLaunch() method:


```
try
{
    m_Database.OpenOrCreate(0, "FirstAppData", 'POL ', 'data');
}
catch (CDBException& e)
{
    e.Display();
    return e.GetCode();
}
```

Lines above use `CDatabase::OpenOrCreate()` for initializing application database. This method performs the following steps:

- a) It looks for database with specified name (*"FirstAppData"* is a database name for this application);
- b) if database already exists, this method opens it;
- c) if database doesn't exist, then it will be created and opened.

Of course, every operation can fail due to some reasons (for example, out of memory space). Response to different errors depends on options used in your project. If you create application where C++ exceptions support is on, POL methods will throw C++ exceptions in response to every trapped error. It's very useful because you can handle all errors using extremely elegant programming style: POL has a rich collection of exception classes where all exception classes are derived from single `CException` class.

The code above shows how to handle an error. It uses `CDBException` class. The `CDBException` can be thrown if application database is not opened and/or created successfully.

C++ exceptions have a big disadvantage because they are very expensive for application size. Palm OS devices have their own specific – usually there low memory space, slow CPUs and severe segment size limitations. If you'll consider this reality, you may want not to use C++ exceptions in your application. In this case POL methods will inform you about errors through return codes (most of functions return standard `Err` code) and `ErrThrow` macro if return operator cannot be performed.

Application stationeries provide the simplest way to specify desired error handling during project creation. Our survey application uses C++ exceptions (we have chosen **POL X Starter** stationery at the start of this tutorial) to provide you with more clear code.

33. Please save your changes and close *"StarterApp.cpp"* window.

34. Open *"MainForm.h"* file. We will add `frmOpenEvent` event handler to `CMainForm` class.

35. Add declaration of event handler for this event:

```
Boolean OnOpen(EventPtr pEvent, Boolean& bHandled);
```

36. Register event map entry for this event:

```
EVENT_MAP_ENTRY(frmOpenEvent, OnOpen)
```

37. Save changes and close *"MainForm.h"* file.

38. Open *"MainForm.cpp"* file.

39. Append include directive just after another include directives at the top of *"MainForm.cpp"* file:

```
#include "StarterApp.h"
```

40. Add implementation of `frmOpenEvent` event handler:

```
// Open event handler
Boolean CMainForm::OnOpen(EventPtr pEvent, Boolean& bHandled)
{
    CStarterApp* pApp =
    (CStarterApp*)CStarterApp::GetInstance();
    CDatabase* pDatabase = pApp->GetDatabase();

    // Initialize grid object
    m_gridAddress.Attach(MainAddressTable, MainAddressScrollBar,
    pDatabase);

    bHandled = false;
    return false;
}
```

This function initializes `m_gridAddress` variable by attaching database variable and table/scrollbar resources to grid variable. `CDBGrid::Attach()` method is used for this purpose.


Please note that database variable is stored in central application class to share it among multiple forms. The reference to `CStarterApp` class is required to get pointer to database variable. `CPalmApp` class can be easily accessed from any part of your application using `CStarterApp::GetInstance()` method. This method returns reference to single instance of application class. It can be used from launch codes where global variable area is accessible (for example, from `sysAppLaunchCmdNormalLaunch` code). The pointer must be explicitly converted to `CStarterApp*` because `CPalmApp::GetInstance()` method returns pointer to `CPalmApp` class.

`bHandled` flag was explicitly changed to `false` value at the end of this function to prevent bypassing `frmOpenEvent` event handler default in base `CForm` class. You can read more about event routing facilities in this tutorial above.

41. Save latest changes and close "*MainForm.cpp*" file.

42. We have attached database variable and table/scrollbar resources to `m_gridAddress` variable. But `CDBGrid` class still doesn't know how to display data retrieved from database, i.e. we have not specified yet whatever fields should be displayed in table columns. `CGrid` (parent of `CDBGrid` class) class has `GetItemText()` virtual method. This method is called from `CGrid` class. It must return text for table cell in specified row and column. This text will be displayed on the screen. We will retrieve this text from application database.

Please note that `GetItemText()` method can be useful for text cells only. If you want to display graphical images in the cells, then you must override `CGrid::DrawItem()` virtual function.

43. Press **Ctrl+N** key sequence or click on  (New Text File) button on main CodeWarrior toolbar. New text window has appeared.

44. Type the following code in this window:

```
// Declaration of Address grid class
#ifdef __AddressGrid_H__
#define __AddressGrid_H__

// Address grid class
class CAddressGrid : public CDBGrid
{

```

```

public:
    // Returns pointer to item's text or NULL value if no text
    // should be shown
    virtual const Char* GetItemText(Int16 wAbsoluteRow, Int16
wColumn);

protected:
    // Text buffer used for filling grid
    CString m_strName;
    CString m_strPhone;
};

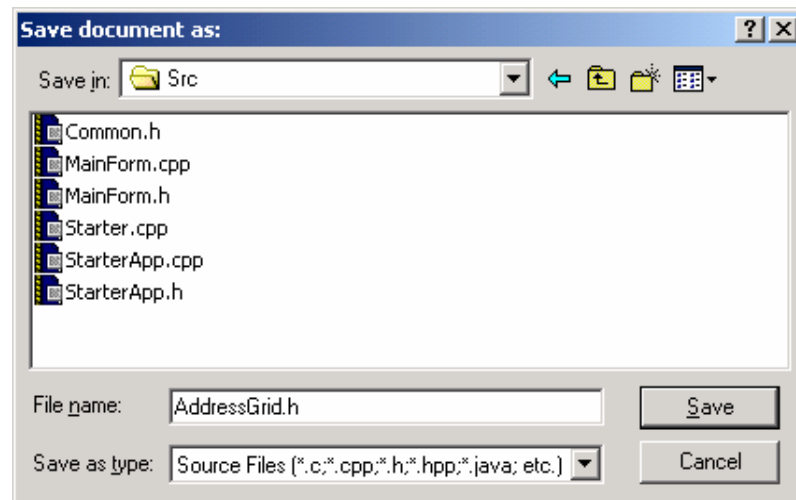
#endif


```

This code declares CAddressGrid class as CDBGrid descendant. GetItemText() is a single method overridden in CAddressGrid class. m_strName and m_strPhone string variables will be used in GetItemText() function.

Please meet a new CString class. A CString object consists of a variable-length sequence of characters. CString provides functions and operators using syntax similar to that of Basic.

45. Press **Ctrl+S** key sequence or select **Save** item from **File** menu. **Save document as:** dialog must be shown.
46. Locate **Src** subfolder in your project folder and type *“AddressGrid.h”* in *“File name”* field.



47. Press **Save** button.
48. Close *“AddressGrid.h”* window.
49. Again press **Ctrl+N** key sequence or click on  (New Text File) button on main CodeWarrior toolbar. New text window must be shown.
50. Type the following code in this window:

```

// Implementation of Address grid class
#include "Common.h"
#include "AddressGrid.h"

// Returns pointer to item's text
const Char* CAddressGrid::GetItemText(Int16 wAbsoluteRow, Int16
wColumn)
{
    UInt16 wIndex = GetRowID(wAbsoluteRow);
    CDBStream dbs(m_pDatabase->QueryRecord(wIndex));
    dbs >> m_strName >> m_strPhone;
}

```

```
        return (wColumn == 0 ? m_strName : m_strPhone);  
    }
```

These several lines of code are very interesting: as you remember, `GetItemText()` function has return text displayed in table cells. This text will be taken from application database.

`CDBGrid` class simplifies this operation because it maintains a list of database record indexes during grid scrolling. Record indexes are stored in row ID information of table data structure. `CDBGrid` keeps only indexes of displayed records to minimize memory usage.

You can retrieve row ID using `CTable::GetRowID()` function. This function takes relative row number (i.e. relative row number of the first displayed row is 0 even if grid is scrolled up). `CDGBrid::GetItemText()` function takes relative row number as parameter, so you can simply pass it to `GetRowID()` function.

If you know record index, you can start reading data from database record. There exists only one problem: Palm OS databases keep arrays of unformatted memory chunks. So, if you need to keep two or more fields (addressee's name and phone number) in a single record, you need to separate these fields in memory chunk from each other.

This problem can be easily solved using POL stream classes. The "stream" is the central concept of the I/O classes in POL. You can think of a stream object as a "smart file" that acts as a source and destination for bytes. A stream's characteristics are determined by its class and by customized insertion and extraction operators. Stream classes have four big advantages you can use:

- a) streams are universal – you can use steam classes with any data source (i.e. database record, file, string in memory, network socket, serial port, exchange manager socket and so on);
- b) one stream can restore exactly the same information as if it was stored using another stream, including fields format and fields count (stream is responsible for separating addressee's name from phone number field value in memory chunk);
- c) streams are easy in use (you can use well known "<<" and ">>" C++ operations with general C types and some specific POL types, including `CString` class and so on; you can also override these operations for any custom type).
- d) stream classes have optional built-in buffering capabilities – you can turn on buffering capabilities for certain operations to speedup data exchange.

`CDBStream` is a database-specific stream type intended for reading database records. It can be used with `DmQueryRecord()` API function for reading memory of record chunks. This approach doesn't require record locking as it is required for record modification. So this approach is optimized for fast read-only access to database memory. `CRecordStream` contrary to `CDBStream` class is yet another database stream type optimized for record modification.

Now you can easily interpret the following code:

```
CDBStream dbs(m_pDatabase->QueryRecord(wIndex));  
dbs >> m_strName >> m_strPhone;
```

First line of code creates `CDBStream` object and initializes it with record using index taken from `CDBGrid` object. Second line of code reads addressee's name and phone number from database record.

Our table has two columns: first column will display addressees' names and second column will display phone numbers. Terminating `return` operator returns text of appropriate field according to column number (columns are numbered starting from zero).

Please note that `m_strName` and `m_strPhone` variables are not allocated in a stack as automatic variables but they are declared as `CAddressGrid` member variables. As you see, `GetItemText()` function must return a pointer to the text to be displayed outside of the function. Automatic variables will be destroyed after finishing `GetItemText()` function, so the text cannot be used outside the function. Member variables contrary to automatic variables have the same lifetime as containing object, so we can safely use `m_strName` and `m_strPhone` memory.

51. Press **Ctrl+S** key sequence or select **Save** item from **File** menu. **Save document as:** dialog must be shown.
52. Locate **Src** subfolder in your project folder and type *"AddressGrid.cpp"* in *"File name"* field
53. Press **Save** button.
54. Close *"AddressGrid.cpp"* window.
55. Add *"AddressGrid.h"* and *"AddressGrid.cpp"* files to your project:
 - a) Select **Add Files** menu item from **Project** menu. **Select files to add...** dialog must be shown.
 - b) Locate **Src** subfolder in your project folder select.
 - c) Select both *"AddressGrid.h"* and *"AddressGrid.cpp"* files.
 - d) Press **Open** button. **Add Files** dialog must be shown. Both **POL Starter – Debug** and **POL Starter – Release** targets must be checked.
 - e) Press OK button.
 - f) Drag *"AddressGrid.h"* file to **Headers** folder of project window.
 - g) Drag *"AddressGrid.cpp"* file to **Sources** folder of project window.
56. Please rebuild your application by pressing **F7** key sequence or selecting **Make** command from **Project** menu. Everything should be compiled without any warning or error.

Congratulations! You've completed the second step.

In the next step we will add a code to allow user creating, modifying and deleting records.

Note: if you've finished your current work, please:

- a) Select **File / Exit** menu command in Metrowerks CodeWarrior application.

- b) Right click on Palm OS Emulator and select **Exit** menu item.

Step 3: Adding modal form

You can find necessary project files for this step in **Step 3** folder.

User will create and modify address records in a separate “*Edit Address*” form. This form will be called from *Main* form by selecting address entry to be edited or by pressing *New* button to create a whole new record. This form will contain *OK* and *Cancel* buttons to accept or decline changes. There will be *Delete* button to delete edited record to have interface similar to standard “*Address Book*” application.

Form development consists of two steps:

- a) creating form resource;
- b) adding C++ form class.

Note: if you start this step as separate step you need preliminary:

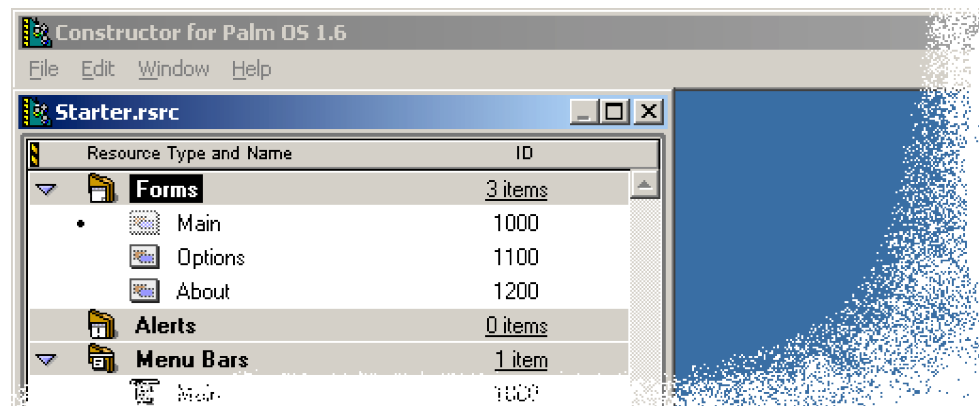
- a) Launch CodeWarrior IDE from Window Start menu.
- b) Select **Open** command from **File** menu in Metrowerks CodeWarrior IDE and select **FirstApp.mcp** project (this tutorial assumes that it's located in *C:\FirstApp* folder).

If you continue executing previous step, please follow the instruction below.

1. We will start form development with creating corresponding form resource.
Open “*Starter.rsrc*” file from project window.
2. Double-click *Forms/Main* form resource in “*Starter.rsrc*” window.
3. Press **Ctrl+Y** key combination or select **Catalog** menu item from **Window** menu. The *Catalog* window must be shown.
4. Drag **Button** UI object from *Catalog* window to *Main* form.
5. Assign the following attributes to new button:

Attribute Name	Attribute Value
Object Identifier	New
Label	New
Left Origin	108
Top Origin	145

6. Select *Forms* folder in “*Starter.rsrc*” window.



7. Press **Ctrl+K** key sequence or select **New Form Resource** item from **Edit** menu. New form with “*untitled*” identifier must be created.

8. Type *“Edit Address”* in place of default *“untitled”* identifier.
9. Double-click on new form. Form editor window must be shown.
10. Assign the following attributes to the new form:

Attribute Name	Attribute Value
Form Title	Edit Address
Modal	<input checked="" type="checkbox"/>

11. Drag **Label** UI object from *Catalog* window to *“Edit Address”* form.
12. Assign the following attributes to the new label:

Attribute Name	Attribute Value
Object Identifier	Name
Text	Name:
Left Origin	4
Top Origin	20

13. Drag **Field** UI object from *Catalog* window to *“Edit Address”* form.
14. Assign the following attributes to the new field:

Attribute Name	Attribute Value
Object Identifier	Name
Left Origin	40
Top Origin	20
Width	116
Single Line	<input checked="" type="checkbox"/>
Max Characters	50

15. Drag **Label** UI object from *Catalog* window to *“Edit Address”* form.
16. Assign the following attributes to the new label:

Attribute Name	Attribute Value
Object Identifier	Phone
Text	Phone:
Left Origin	4
Top Origin	36

17. Drag **Field** UI object from *Catalog* window to *“Edit Address”* form.
18. Assign the following attributes to the new field:

Attribute Name	Attribute Value
Object Identifier	Phone
Left Origin	40
Top Origin	36
Width	116
Single Line	<input checked="" type="checkbox"/>
Max Characters	50

19. Drag **Label** UI object from *Catalog* window to *“Edit Address”* form.
20. Assign the following attributes to the new label:

Attribute Name	Attribute Value
Object Identifier	Address

Attribute Name	Attribute Value
Text	Address:
Left Origin	4
Top Origin	52

21. Drag **Field** UI object from *Catalog* window to “*Edit Address*” form.

22. Assign the following attributes to the new field:

Attribute Name	Attribute Value
Object Identifier	Address
Left Origin	40
Top Origin	52
Width	116
Single Line	<input checked="" type="checkbox"/>
Max Characters	100

23. Drag **Label** UI object from *Catalog* window to “*Edit Address*” form.

24. Assign the following attributes to the new label:

Attribute Name	Attribute Value
Object Identifier	Notes
Text	Notes:
Left Origin	4
Top Origin	68

25. Drag **Field** UI object from *Catalog* window to “*Edit Address*” form.

26. Assign the following attributes to the new field:

Attribute Name	Attribute Value
Object Identifier	Notes
Left Origin	4
Top Origin	80
Width	145
Height	60
Max Characters	500

27. Drag **Scrollbar** UI object from *Catalog* window to “*Edit Address*” form.

28. Assign the following attributes to the new scrollbar:

Attribute Name	Attribute Value
Object Identifier	Notes
Left Origin	149
Height	60
Top Origin	80

29. Drag **Button** UI object from *Catalog* window to “*Edit Address*” form.

30. Assign the following attributes to the new button:

Attribute Name	Attribute Value
Object Identifier	OK
Text	OK
Left Origin	4
Top Origin	145

31. Drag **Button** UI object from *Catalog* window to “*Edit Address*” form.

32. Assign the following attributes to the new button:

Attribute Name	Attribute Value
Object Identifier	Cancel
Text	Cancel
Left Origin	46
Top Origin	145

33. Drag **Button** UI object from *Catalog* window to “*Edit Address*” form.

34. Assign the following attributes to the new button:

Attribute Name	Attribute Value
Object Identifier	Delete
Text	Delete
Left Origin	120
Top Origin	145

35. Select **Alerts** folder in “*Starter.rsrc*” window.

36. Press **Ctrl+K** key sequence or select **New Alert Resource** item from **Edit** menu. New alert with “*untitled*” identifier must be created.

37. Type “*Delete Address*” in place of default “*untitled*” identifier.

38. Double-click on new alert. Alert editor window must be shown.

39. Assign the following attributes to the new alert:

Attribute Name	Attribute Value
Alert Title	Warning
Title	Delete Address
Message	Delete this address?

40. Select “*Button Titles*” attribute for the alert.

41. Press **Ctrl+K** keys or select **New Button Title** command from **Edit** menu.

42. Second “*Button Title*” item must appear in *Column Widths* list.

43. Select “*Item Text 0*” item and set it to Yes.

44. Select “*Item Text 1*” item and set it to No.

45. Press **Ctrl+S** key or select **Save** menu item from **File** menu.

46. Select **Exit** command from **File** menu. “*Constructor for Palm OS*” must be closed.

47. Form resource is ready, so we can start creating corresponding C++ form class. But before coding let's think how to switch easily to “*Edit Address*” form from parent *Main* form. Palm OS provides us with number of ways for this operation:

- a) `FrmGotoForm()` API function (and `CForm::GotoForm()` equivalent) closes currently active form before activating a new form. After exiting from “*Edit Address*” form we will be forced to re-open *Main* form. In this case *Address* table will be reinitialized and it will display the latest changes.

Unfortunately current grid selection and scrollbar position will be lost because grid is reinitialized in `frmOpenEvent` event handler of *Main* form. To restore previous grid selection and scrollbar position special code is required.

There exist simpler way for solving this problem.

- b) `FrmPopupForm()` API function (and `CForm::PopupForm()` equivalent) doesn't close parent form before activating new form. So, previous grid selection and scrollbar position will not be changed after exiting from "*Edit Address*" form.

However there exists another obstacle. You need to redraw grid in *Main* form to display the latest changes. Because the grid is located in *Main* form, refreshing should occur in *Main* form code. To call this code you may want to know when "*Edit Address*" form was closed. But `FrmPopupForm()` is not modal function – it adds necessary events to event queue to display popup form and immediately returns to calling code. So, popup form will be opened asynchronously. Your *Main* form will not know when "*Edit Address*" form is closed and you will not know when *Address* grid must be refreshed.

Of course, you can add some code to resolve this problem (for example, send custom event to *Main* form after exiting from "*Edit Address*" form), but there exist yet more simple solution.

- c) `FrmDoDialog()` API function (and `CForm::DoDialog()` equivalent) runs internal event loop and returns to parent form after closing modal form. Grid selection and scrollbar position in *Main* form will not be changed. Also, you will exactly know whatever data was changed and whenever grid must refresh its data due to `FrmDoDialog()` function which returns button ID the user tapped to close the form (i.e. we will save changes and refresh the grid if ID of OK button is returned).

Unfortunately this solution is not perfect due to "*Edit Address*" has *Delete* button. As you remember, we want to confirm record deletion if user presses *Delete* button. User will be able to cancel delete operation and continue edit/view record. But `FrmDoDialog()` always closes modal form if user taps on any button. So, we can show confirmation alert after "*Edit Address*" form is closed and we cannot allow user to continue record edit/view if *No* button is pressed!

- d) Standard Palm OS form routines don't allow easily to implement this simple user interface. Any method either requires some additional coding or cannot provide required features.


POL solves this problem by creating a whole new type of user forms not available in standard Palm OS API. You can simply derive your form class from `CModalForm` and you will be able to create customizable modal forms. `CModalForm` descendants can be customized using event maps as any normal POL form.

`CModalForm` class provides you with yet another advantage. It's possible to use Dynamic Data Exchange (DDX) maps in any POL form. If you use

the DDX mechanism, you set the initial values of the form object member variables, binds them with corresponding fields and controls in a form and the framework's DDX mechanism automatically transfers the values of the member variables to the fields and controls of the form, where they appear when the form itself appears in response to `CModalForm::DoModal()` or `CForm::GotoForm()`.

The same mechanism transfers values back from the fields and controls to the member variables when the user closes the form.

We will derive `CEditAddress` form class from `CModalForm` class in our example. Event map allows to process click on *Delete* button by a normal way. DDX mechanism allows you automatically transfer values between member variables and fields' data.

57. Press **Ctrl+N** key sequence or click on  (New Text File) button on main CodeWarrior toolbar. New text window must be shown.

58. Type the following code in this window:

```
// Declaration of Edit Address form class
#ifndef __EditAddressForm_H__
#define __EditAddressForm_H__

// Edit Address form class
class CEditAddressForm : public CModalForm
{
public:
    // Requested operation
    enum ModalResult
    {
        mrOK,
        mrCancel,
        mrDelete
    };

    // Constructor
    CEditAddressForm() : CModalForm(EditAddressForm) { }

    // Command handlers
    Boolean OnOpen(EventPtr pEvent, Boolean& bHandled);
    Boolean OnOK(EventPtr pEvent, Boolean& bHandled);
    Boolean OnCancel(EventPtr pEvent, Boolean& bHandled);
    Boolean OnDelete(EventPtr pEvent, Boolean& bHandled);

    // Event map
    BEGIN_EVENT_MAP(CModalForm)
        EVENT_MAP_ENTRY(frmOpenEvent, OnOpen)
        EVENT_MAP_COMMAND_ENTRY(EditAddressOKButton, OnOK)
        EVENT_MAP_COMMAND_ENTRY(EditAddressCancelButton,
        OnCancel)
        EVENT_MAP_COMMAND_ENTRY(EditAddressDeleteButton,
        OnDelete)
    END_EVENT_MAP()

    // DDX map
    BEGIN_DDX_MAP()
        DDX_TEXT(EditAddressNameField, m_strName)
        DDX_TEXT(EditAddressPhoneField, m_strPhone)
        DDX_TEXT(EditAddressAddressField, m_strAddress)
        DDX_MEMO(EditAddressNotesField,
        EditAddressNotesScrollBar, m_strNotes)
    END_DDX_MAP()

    // Form data
    CString m_strName;
    CString m_strPhone;
    CString m_strAddress;
```

```
        CString m_strNotes;  
  
protected:  
    // Control variables  
    CMemo m_memoNotes;  
};  
  
#endif
```

CModalForm class has two constructors: the first constructor takes form ID to be loaded and the second form constructor doesn't take form ID and doesn't load form resource into memory.

- If we will pass form ID to form constructor, then CModalForm::DoModal() method must be invoked without any parameters because form resources have been already loaded into memory.
- We can omit this parameter and don't override form constructor, but pass form ID to CModalForm::DoModal() method to load form resource inside this method.

CEditAddress class constructor passes form ID to base CModalForm class because we can hide *Delete* button before showing “*Edit Address*” form on the screen if this form will be shown for new record. It requires us to load form into memory before calling DoModal() method. You'll see such code below in this chapter.

Please note that CEditAddressForm class is derived from CModalForm class, so CModalForm (not CForm) class is passed to BEGIN_EVENT_MAP macro as a parameter.

CString member variables declared in this class will be used for storing fields' data of the form. DDX map macros bind these variables to the field identifiers used for editing these variables. DDX_TEXT macro is used for binding a string or numeric variables with single-line fields. DDX_MEMO macro is used for binding string variables with multi-line scrollable fields (this macro additionally takes scrollbar ID to adjust scrollbar according to text length).

Palm OS has standard multi-line field control. However this field control cannot automatically synchronize its content with scrollbar control. Palm OS developers must manually write such synchronization code. POL has already provided you with predefined CMemo class. This class can perform all “dirty” work for you, just initialize CMemo variable with corresponding field ID and scrollbar.

ModalResult enumeration defines constants to indicate requested user operation:


- mrOK – record must be saved;
- mrCancel – no change must be made;
- mrDelete – record must be deleted.

59. Press **Ctrl+S** key sequence or select **Save** item from **File** menu. **Save document as:** dialog must be shown.

60. Locate **Src** subfolder in your project folder and type “*EditAddressForm.h*” in “*File name*” field

61. Press **Save** button.

62. Close “*EditAddressForm.h*” window.

63. Press **Ctrl+N** key sequence or click on  (New Text File) button on main CodeWarrior toolbar. New text window must be shown.

64. Type the following code:

```
// Implementation of Main form class
#include "Common.h"
#include "EditAddressForm.h"

// Open event handler
Boolean CEditAddressForm::OnOpen(EventPtr pEvent, Boolean&
bHandled)
{
    m_memoNotes.Attach(EditAddressNotesField,
EditAddressNotesScrollBar);
    bHandled = false;
    return false;
}

// OK button handler
Boolean CEditAddressForm::OnOK(EventPtr pEvent, Boolean&
bHandled)
{
    CloseForm(mrOK);
    return true;
}

// Cancel button handler
Boolean CEditAddressForm::OnCancel(EventPtr pEvent, Boolean&
bHandled)
{
    CloseForm(mrCancel);
    return true;
}

// Delete button handler
Boolean CEditAddressForm::OnDelete(EventPtr pEvent, Boolean&
bHandled)
{
    if (FrmAlert>DeleteAddressAlert) == 0)
        CloseForm(mrDelete);
    return true;
}
```

This code implements all the functionality of “*Edit Address*” form.

OnOK(), OnCancel() and OnDelete() methods close modal form using CloseForm() function. This function takes modal result parameter. This parameter will be returned from CModalForm::DoModal() method and used later in our main form to determine an operation requested by user.

OnDelete() handler confirms delete operation without closing the form using FrmAlert() API function.

Again, please note that OnOpen() event handler explicitly defines bHandled flag with false value as always.

65. Press **Ctrl+S** key sequence or select **Save** item from **File** menu. **Save document as:** dialog must be shown.

66. Locate **Src** subfolder in your project folder and type “*EditAddressForm.cpp*” in “*File name*” field

67. Press **Save** button.

68. Close “*EditAddressForm.cpp*” window.

69. Add *"EditAddressForm.h"* and *"EditAddressForm.cpp"* files to your project:

- a) Select **Add Files** menu item from **Project** menu. **Select files to add...** dialog must be shown.
- b) Locate **Src** subfolder in your project folder select
- c) Select both *"EditAddressForm.h"* and *"EditAddressForm.cpp"* files.
- d) Press **Open** button. **Add Files** dialog must be shown. Both **POL Starter – Debug** and **POL Starter – Release** targets must be checked.
- e) Press OK button.
- f) Drag *"EditAddressForm.h"* file to **Headers** folder of project window.
- g) Drag *"EditAddressForm.cpp"* file to **Sources** folder of project window.

70. Now we can add some code to call *"Edit Address"* form from *Main* form. As mentioned previously, this form will be shown if user selects existing records in a grid (tblSelectEvent event) or if user presses **New** button for creating new records.

Please open *"MainForm.h"* file.

71. Add the following new event handlers:

```
Boolean OnNew(EventPtr pEvent, Boolean& bHandled);
Boolean OnSelect(EventPtr pEvent, Boolean& bHandled);
```

72. Add the corresponding entries to the event map:

```
EVENT_MAP_COMMAND_ENTRY(MainNewButton, OnNew)
EVENT_MAP_ENTRY(tblSelectEvent, OnSelect)
```

73. Save your changes and close *"MainForm.h"* window.

74. Open *"MainForm.cpp"* file.

Append include directive just after another include directives at the top of *"MainForm.cpp"* file:

```
#include "EditAddressForm.h"
```

75. Add OnNew() and OnSelect() methods implementation to the file:

```
// New command handler
Boolean CMainForm::OnNew(EventPtr pEvent, Boolean& bHandled)
{
    CEditAddressForm frmEditAddress;
    frmEditAddress.HideObject(EditAddressDeleteButton);
    if (frmEditAddress.DoModal() == CEditAddressForm::mrOK)
    {
        // Create record
    }
    return true;
}

// Select event handler
Boolean CMainForm::OnSelect(EventPtr pEvent, Boolean& bHandled)
{
    CEditAddressForm frmEditAddress;
    switch (frmEditAddress.DoModal())
    {
    case CEditAddressForm::mrOK:
        // Modify record
        break;
    case CEditAddressForm::mrDelete:
        // Delete record
        break;
    }
```

```
    }  
    return true;  
}
```

`CMemo::Attach()` is used to initialize `m_memoNotes` variable with proper field ID and scrollbar ID. After initialization `m_memoNotes` variable will automatically handle all user's input and synchronize the scrollbar with the field data.

Early discussed `CModalForm::DoModal()` function displays modal form on the screen and returns control to calling function after closing the modal form. This function returns modal result code. This code was passed to `CModalForm::CloseForm()` function in `OnOK()`, `OnCancel()` and `OnDelete()` handlers of `CEditAddress` form class. So, we can easily determine button used to close the "Edit Address" form.

76. Save your changes and close "*MainForm.cpp*" window.

77. Every Palm OS application consists of one or more code segments. Every code segment is limited up to 64Kb size. If code exceeds segment boundaries it must be placed into second segment and so forth. The compiler handles cross-segment function calls using quite another approach.

You must manually create new code segments as required and specify whatever code should be located in appropriate segment. It's especially important because of only one (first) code segment can be loaded for some application launch codes. By default every POL project already consist of four code segments, but a number of segments can be expanded later.

There exists yet one complexity: your project can be compiled using different memory models (*small*, *smart* and *large*). However, in practice it's recommended to use *small* memory model for Palm OS applications. This model is not so convenient as *smart* memory model (sometimes you receive "16-bit code reference is out of range" linker error) but *small* memory model reproduces much smaller and faster code than *smart* model does. Moreover, the *smart* memory model cannot automatically solve segment size limitation, so it is not recommended to use this model in your Palm OS projects.

Our project use small memory model. We will perform some steps in this tutorial to show you how to resolve the problems described above:

Please switch to **Segments** tab in project window. This tab consists of three default code segments. Parts of POL library are located in these segments.

Our project grows bit by bit so very soon we will receive described above linker errors. These errors can be solved by rearranging code (you must place depending code parts closer to each other).

Switch to **Segments** tab in project window.

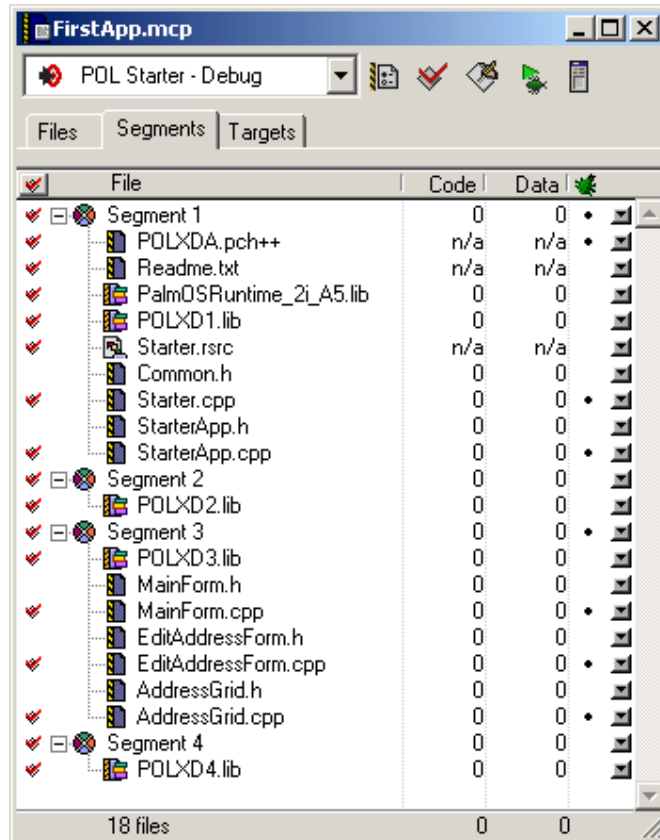
78. Drag the following files from the first and second segments into the third code segment:

- MainForm.h
- MainForm.cpp
- EditAddressForm.h

- EditAddressForm.cpp
- AddressGrid.h
- AddressGrid.cpp

79. Rearrange these files by dragging corresponding items in project window in the order listed above.

Your project window must have exactly the same look as shown on the picture below:



80. Select "*POL Starter – Release*" target in the targets combo-box located in the top left corner of the project window.
81. Perform the 78 - 79 steps to rearrange files for release configuration of our project.
82. Again, select "*POL Starter – Debug*" target in the targets combo-box located in the top left corner of the project window.
83. Switch back to **Files** tab in project window. You'll see normal view of our project.
84. Rebuild our application by pressing **F7** key sequence or selecting **Make** command from **Project** menu. Everything must be compiled without any warning or error.
85. Select **Debug** command from **Project** menu or simply press **F5** key. After that press *New* button on Palm OS Emulator. The application must have a look similar to image below:



Congratulations! You've completed the third step.

In the next step we will finally add the code to allow user creating, modifying and deleting records.

Note: if you've finished your current work, please:

- a) Select **File / Exit** menu command in Metrowerks CodeWarrior application.
- b) Right click on Palm OS Emulator and select **Exit** menu item.

Step 4: Modifying data

You can find necessary project files for this step in **Step 4** folder.

In this chapter we will add a code for creating, modifying and deleting records. This code will be located in `OnNew()` and `OnSelect()` methods of `CMainForm` class. These methods already include short comments in corresponding places.

As you know, `CDBStream` class is intended for fast and free of locking way of reading data from database record.

POL provides you with yet another record stream class. `CRecordStream` class is intended for creating and modifying records. This class locks a record during reading and writing to prevent data corruption by other parts of your program that can have access to the same record at the same time. Locked record cannot be rewritten until it is unlocked. We will use `CRecordStream` class in our example for creating and modifying data.

`CDBStream` class can unambiguously read data written by `CRecordStream` due to every POL stream class uses the same data formatting.

Note: if you start this step as separate step you need preliminary:

- a) Launch CodeWarrior IDE from Window Start menu.
- b) Select **Open** command from **File** menu in Metrowerks CodeWarrior IDE and select **FirstApp.mcp** project (this tutorial assumes that it's located in `C:\FirstApp` folder).

If you continue executing previous step, please follow the instruction below.

1. Open `"MainForm.cpp"` file.
2. Add the following code just after `"Create record"` comment to `OnNew()` function:

```
CStarterApp* pApp = (CStarterApp*)CStarterApp::GetInstance();
CDatabase* pDatabase = pApp->GetDatabase();
CRecordStream rs(pDatabase);
rs << frmEditAddress.m_strName << frmEditAddress.m_strPhone <<
frmEditAddress.m_strAddress << frmEditAddress.m_strNotes;
m_gridAddress.AddRow();
```

These lines create and fill new database record with information entered in `"Edit Address"` form.

`CRecordStream` class has a constructor that takes pointer to database and optional record index. It will try to open an existing database record if you pass record index to the constructor. If you want to create a new record simply omit this parameter. Lines above pass the database pointer only to `CRecordStream` constructor, so new record will be created.

Then field values of `"Edit Address"` form are passed to record stream using overloaded `<<` operator. Please note that `CEditAddress` member variables will contain values of associated fields if user taps on `OK` button thanks to DDX mechanism (see above in this tutorial).

`CDBGrid::AddRow()` call tells `Address` grid about new record in a database. `CDBGrid` will look for a new record at the end of database if this method is called without parameter.

3. Insert the following code between `CEditAddressForm` form constructor and the following switch-operator into `OnSelect()` function:

```
CStarterApp* pApp = (CStarterApp*)CStarterApp::GetInstance();
CDatabase* pDatabase = pApp->GetDatabase();
Int16 nAbsRow = m_gridAddress.RelativeToAbsolute(pEvent->data.tblSelect.row);
UInt16 wIndex = m_gridAddress.GetCurrentIndex();
CRecordStream rs(pDatabase, wIndex);
rs >> frmEditAddress.m_strName >> frmEditAddress.m_strPhone >>
frmEditAddress.m_strAddress >> frmEditAddress.m_strNotes;
```

tblSelectEvent event is sent to form class if user selects row in a grid. This event passes relative position of selected row. CGrid::RelativeToAbsolute() method is used for converting relative row number to absolute row number by the following formula.

Absolute Row Number = Relative Row Number + Scrollbar Position

Absolute row number will be used later for refreshing row image (if record will be modified) or row deleting (if record will be deleted).

Current selected row can be determined by yet another way:

```
Int16 nAbsRow = m_gridAddress.GetSelection();
```

Use any method you like.

CDBGrid::GetCurrentIndex() returns record index (not the row position) for selected row. This index is used in the next line of code to read record information. Please note that CRecordStream class (not CDBStream class) is used to read record data. As you already know, CRecordStream locks record data to prevent simultaneous modifications. This feature of CRecordStream class is very useful in our example because record can be locked until user will close “Edit Address” form.

4. The same stream variable will be used to store new data in a record if user presses OK button. Add record modification code just after “Modify record” to OnSelect() function:

```
rs.Seek(0, fileOriginBeginning);
rs << frmEditAddress.m_strName << frmEditAddress.m_strPhone <<
frmEditAddress.m_strAddress << frmEditAddress.m_strNotes;
m_gridAddress.Refresh(nAbsRow);
```

CRecordStream class supports seeking position. A seeking position is a place where a stream reads or writes operation start from. Number of transferred bytes is added to seeking position after every read or write operation. So, end of stream will be reached after entire record data was read from the stream before “Edit Address” form is shown on the screen. To write newly modified data back to the stream, seeking position must be moved to the beginning of the stream. CRecordStream::Seek(0, fileOriginBeginning) method moves seeking position to beginning of the stream.

Further write operations pass “Edit Address” member variables back to the record. These member variables represent the latest user input in “Edit Address” form. The values were taken using DDX map declared in CEditAddress form class.

CGrid::Refresh() method redraws modified row image on the screen as soon as record was stored in a database.

5. Add the following code to OnSelect() method below “Delete record” comment.

```
rs.Close();
pDatabase->RemoveRecord(wIndex);
```

```
m_gridAddress.DeleteRow(nAbsRow);
```

CRecordStream::Close() method detaches record stream resources from database early locked record before deleting it.

CDatabase::RemoveRecord() removes selected record from database.

CDBGGrid::DeleteRow() method informs grid about deleted record to remove corresponding row from the screen.

6. Save your changes and close "MainForm.cpp" window.
7. Rebuild your application by pressing **F7** key sequence or selecting **Make** command from **Project** menu. Everything must be compiled without any warning or error.
8. Select **Debug** command from **Project** menu or simply press **F5** key. Input several new records in our application and verify all actions.
9. It seems our application to be almost ready – it already can be used in everyday life. Let's add some glitter to its user interface.

You can see that particular cells (instead of full rows) are selected. It's standard look of Palm OS tables. Though selecting of the full rows instead of particular cells is more appropriate. You can easily customize look of POL grids using CGrid::SetGridMode() method.



Standard grid mode



Full row selection grid mode

Terminate the debug session and add the following line of code to OnOpen() method, just after m_gridAddress.Attach() method call:

```
m_gridAddress.SetGridMode(CGrid::GM_FULLROWSELECT);
```

10. By default, CGrid scrolls content in response to **PgUp** and **PgDown** system keys if input focus is set to grid. However, if your form contains only one grid to be scrolled, you may want always pass **PgUp** and **PgDown** key combinations to grid object. CGrid::SetIgnoreFocus()

allows you to control its behavior. Add yet one line to `OnOpen()` method below `SetGridMode()`, write:

```
m_gridAddress.SetIgnoreFocus(true);
```

11. POL grids extend standard Palm OS tables and provide you with different draw styles: row dividers, column dividers, border around grid and style of divider lines.

We will simply add a border around the grid. Add the following line of code to `OnOpen()` method below `SetIgnoreFocus()` call:

```
m_gridAddress.SetDrawStyle(CGrid::DS_BORDER);
```

12. Save your changes and close *"MainForm.cpp"* window.
13. Select **Debug** command from **Project** menu or simply press **F5** key. The application must have a look similar to image below:



Congratulations! You've completed the fourth step.

In the next step we will add code to allow user searching data in the *Address* grid.

Note: if you've finished your current work, please:

- a) Select **File / Exit** menu command in Metrowerks CodeWarrior application.
- b) Right click on Palm OS Emulator and select **Exit** menu item.

Step 5: Adding search functionality

You can find necessary project files for this step in **Step 5** folder.

Simple search functionality makes our application easier for user. We will add lookup field below the table. We will override `keyDownEvent` event and look for appropriate name in our database.

Note: if you start this step as separate step you need preliminary:

- a) Launch CodeWarrior IDE from Window Start menu.
- b) Select **Open** command from **File** menu in Metrowerks CodeWarrior IDE and select **FirstApp.mcp** project (this tutorial assumes that it's located in *C:\FirstApp* folder).

If you continue executing previous step, please follow the instruction below.

1. Open "*Starter.rsrc*" file from project window.
2. Double-click *Forms/Main* form resource in "*Starter.rsrc*" window.
3. Press **Ctrl+Y** key combination or select **Catalog** menu item from **Window** menu. The *Catalog* window must be shown.
4. Drag **Label** UI object from *Catalog* window to *Main* form.
5. Assign the following attributes to the new button:

Attribute Name	Attribute Value
Object Identifier	Lookup
Text	Lookup:
Left Origin	2
Top Origin	145

6. Drag **Field** UI object from *Catalog* window to *Main* form.
7. Assign the following attributes to the new button:

Attribute Name	Attribute Value
Object Identifier	Lookup
Left Origin	40
Top Origin	145
Width	50
Single Line	<input checked="" type="checkbox"/>
Max Characters	5

8. Press **Ctrl+S** key or select **Save** menu item from **File** menu.
9. Select **Exit** command from **File** menu. "*Constructor for Palm OS*" must be closed.
10. Open "*MainForm.h*" file in project window.
11. Add new member variable to control variables section:

```
CField m_fldLookup;
```

`CField` class represents standard Palm OS field objects. `m_fldLookup` variable will represent newly added *Lookup* field.

12. Add new handlers for `keyDownEvent` and `fldChangedEvent` events:

```
Boolean OnKeyDown(EventPtr pEvent, Boolean& bHandled);  
Boolean OnChanged(EventPtr pEvent, Boolean& bHandled);
```

13. Add corresponding event entries to event map:

```
EVENT_MAP_ENTRY(keyDownEvent, OnKeyDown)
EVENT_MAP_ENTRY(fldChangedEvent, OnChanged)
```

14. Add lookup method to protected section of CMianForm class:

```
// Internal methods
void Lookup();
```

15. Save these changes and close “MainForm.h” window.

16. Open “MainForm.cpp” file.

17. Add m_fldLookup initialization to OnOpen() method after grid initialization:

```
m_fldLookup.Attach(MainLookupField);
SetFocus(MainLookupField);
```

Please note that you pass field ID instead of field index (as FrmSetFocus() API function requires) to CForm::SetFocus() method.

18. Add the similar focus restoring to OnSelect() method before return operator:

```
SetFocus(MainLookupField);
```

19. Add implementation of OnKeyDown() method:

```
// Key down handler
Boolean CMainForm::OnKeyDown(EventPtr pEvent, Boolean& bHandled)
{
    // If special key is not pressed
    if (pEvent->data.keyDown.modifiers == 0)
    {
        SetFocus(MainLookupField);
        // Give field object a chance to handle this event
        if (FldHandleEvent(m_fldLookup, pEvent))
            Lookup();
        return true;
    }
    return false;
}
```

This method will be called in response to user input to *Lookup* field.

20. Add implementation of OnChanged() method:

```
// Field changed handled
Boolean CMainForm::OnChanged(EventPtr pEvent, Boolean& bHandled)
{
    Lookup();
    return true;
}
```

This method will be called in response to clipboard operations on *Lookup* field.

21. Add implementation of Lookup() method:

```
// Looks for database record by name
void CMainForm::Lookup()
{
    const Char* pszCriteria = m_fldLookup.GetTextPtr();
    Intl6 nCriteriaLength = StrLen(pszCriteria);
    if (nCriteriaLength > 0)
    {
        CStarterApp* pApp =
            (CStarterApp*)CStarterApp::GetInstance();
        CDatabase* pDatabase = pApp->GetDatabase();
    }
}
```



```
// Loop through database to find appropriate record
CMemHandle memHandle((Boolean)false);
UInt16 wIndex, wCount = pDatabase->NumRecords();
for (wIndex = 0; wIndex < wCount; wIndex++)
{
    memHandle = pDatabase->QueryRecord(wIndex);
    const Char* pszName = (const
Char*)memHandle.Lock();
    if (StrNCaselessCompare(pszName, pszCriteria,
nCriteriaLength) == 0)
    {
        // Matched record is found
        m_gridAddress.SetSelection(wIndex);
        break;
    }
}

// End of database was reached, so no record found
if (wIndex == wCount)
    m_fldLookup.Delete(nCriteriaLength - 1,
nCriteriaLength);
}
else
    m_gridAddress.ClearSelection();
}
```

General implementation of this method most likely is known to every Palm OS developer, so we will describe POL-specific code only.

CMemHandle is a smart pointer to class. This class differs from CAutoPointer and CSmartPointer classes because it uses Palm OS native memory management instead of C++ new operator.

We can use this class directly with database record chunks thanks to this feature. Just pass false to auto-free parameter of CMemHandle to prevent freeing of records' memory. CMemHandle still is very useful because it will automatically unlock memory chunks for unused records.

The line:

```
const Char* pszName = (const Char*)memHandle.Lock();
```

locks record memory chunk and converts it directly into the string. This string is *Name* field of address record, because *Name* data is written to the record first (i.e. record begins with *Name* field):

```
rs << frmEditAddress.m_strName << frmEditAddress.m_strPhone <<
frmEditAddress.m_strAddress << frmEditAddress.m_strNotes;
```

CDBStream class can be used to achieve the similar result:

```
CString strName;
CDBStream dbs(m_pDatabase->QueryRecord(wIndex));
dbs >> strName;
```

But this method is slightly slower because it uses dynamic strings, streams and performs additional memory copying.

Please note that the record index is passed to row number parameter in SetSelection() method:

```
m_gridAddress.SetSelection(wIndex);
```

This line of code can be valid only if row number is equal to record index. Our simple example stores all records in one category and doesn't filter grid content, so this code is valid. However, in more complicated application you will increment row number in search loop and use special *MoveNext*-method methods instead of simple index incrementing.

`CDBGrid::SetSelection()` method will select the found row on the screen and automatically scroll grid to this row if selected row is not visible.

`CDBGrid::ClearSelection()` method will clear selection from the grid if search criteria is empty.

22. Save changes and close *"MinaForm.cpp"* window.

23. Select **Debug** command from **Project** menu or simply press **F5** key. The application must have a look similar to image below:



Congratulations! You've completed the fifth step.

In the next step we will integrate our application into system global search.

Note: if you've finished your current work, please:

- a) Select **File / Exit** menu command in Metrowerks CodeWarrior application.
- b) Right click on Palm OS Emulator and select **Exit** menu item.

Step 6: Integration with global find service

You can find necessary project files for this step in **Step 6** folder.

Palm OS software is most often used on different organizers. Global find service provides user with convenient way to find data among multiple applications.

POL supports `sysAppLaunchCmdFind`, `sysAppLaunchCmdGoTo` launch codes as well as you can handle any non-standard launch code.

`sysAppLaunchCmdFind` is used for implementing global find.

The big complexity of this launch code is denied access to global variables, static local variables, virtual functions, exceptions support, RTTI, and code segments other than first segment. Though CodeWarrior allows accessing virtual functions, RTTI and exception tables in Expanded mode, POL doesn't support this mode.

It means you cannot use exceptions support, stream classes and `CString` class for this launch code. Instead, POL provides you with lightweight classes like `CMemHandle` and `CResource`. You can still use collections, Net lib, exchange manager, serial manager, file manager, `CDatabase` and `CRecord` classes.

`CPalmApp` class also cannot be used with this launch code because this class uses virtual functions. POL provides you with special `CPalmStApp` class for non-global launch codes. This class is implemented as C++ template. It takes information about your application class via template parameter. This technique provides compiler with required information about inheritance while compilation time and allows use static binding.

`sysAppLaunchCmdGoTo` is used for bringing up found data. This code provides access to the global information, but this launch code has another specific – your application can be recurrently called if it is already active, so you should be careful.

Note: if you start this step as separate step you need preliminary:

- a) Launch CodeWarrior IDE from Window Start menu.
- b) Select **Open** command from **File** menu in Metrowerks CodeWarrior IDE and select **FirstApp.mcp** project (this tutorial assumes that it's located in *C:\FirstApp* folder).

If you continue executing previous step, please follow the instruction below.

1. Open *"Starter.rsrc"* file from project window.
2. Double-click *Forms/Main* form resource in *"Starter.rsrc"* window.
3. Select *Strings* folder in *"Starter.rsrc"* window.
4. Press **Ctrl+K** key combination or select **New String Resource** from **Edit** menu. New string with *"untitled"* identifier must be created.
5. Type *"First App"* in place of default *"untitled"* identifier.
6. Double-click on this string resource. String editor window must be shown.
7. Type *"First App"* in multi-line field.
This string will be displayed as application title in system fin dialog.
8. Press **Ctrl+S** key or select **Save** menu item from **File** menu.
9. Select **Exit** command from **File** menu. *"Constructor for Palm OS"* window must be closed.

10. Migration from CPalmApp to CPalmStApp is quite simple. Open “StarterApp.h” file from project window.

11. Change CPalmApp class name to CPalmStApp<CStarterApp>:

You need simply pass derived class name to CPalmStApp class:

```
class CStarterApp : public CPalmStApp<CStarterApp>
```

That's all! Now our application supports special launch codes.

12. Add handlers for sysAppLaunchCmdFind and sysAppLaunchCmdGoToLaunch codes to overrides section:

```
virtual Err GotoLaunch(GoToParamsPtr pGoToParams);  
Err FindLaunch(FindParamsPtr pFindParams);
```

Note, that FindLaunch() handler is not virtual. It should be called using static binding rules.

13. We want to select found record in Address table of Main form. It will be done using the following scheme:

The index of record to be selected (equal to row number in this example) will be stored as static member of CStarterApp class. We will initialize this variable with proper record index in GotoLaunch() method. NormalLaunch() handler will also initialize this member with -1 value to indicate no record to be selected. OnOpen() handler of CMainForm will read this variable and select proper row.

Add record index to protected section of CStarterApp class:

```
static UInt16 m_wGotoIndex;
```

14. Add corresponding get-accessor for this member to public section of CStarterApp class:

```
static UInt16 GetGotoIndex();
```

15. Add implementation of GetGotoIndex() method below the class declaration:

```
inline UInt16 CStarterApp::GetGotoIndex()  
{  
    return m_wGotoIndex;  
}
```

16. Save this change and close “StarterApp.h” window.

17. Open “StarterApp.cpp” file from project window.

18. Add pair m_wGotoIndex declaration to beginning of CPP-file.

```
// Global goto index  
UInt16 CStarterApp::m_wGotoIndex = -1;
```

19. Add m_wGotoIndex initialization to NormalLaunch() method before CForm::GotoForm() call:

```
m_wGotoIndex = -1;
```

20. Add FindLaunch() method implementation:

```
// Find launch handler  
Err CStarterApp::FindLaunch(FindParamsPtr pFindParams)  
{  
    DmSearchStateType stateInfo;  
    UInt16 wCardNo;  
    LocalID liDbID;  
    if (CDatabase::GetNextDatabaseByTypeCreator(true, stateInfo,  
        wCardNo, liDbID, 'POL ', 'data'))
```

```
{
    pFindParams->more = false;
    return errNone;
}

m_Database.Open(wCardNo, liDbID, pFindParams->dbAccessMode);
CResource res(strRsc, FirstAppString);
if (FindDrawHeader(pFindParams, (const Char*)res.Lock()))
    return errNone;

UInt16 wIndex = pFindParams->recordNum;
for (;;)
{
    if ((wIndex & 0x0F) && EvtSysEventAvail(true))
    {
        pFindParams->more = true;
        return errNone;
    }

    CMemHandle memHandle(m_Database.QueryRecord(wIndex),
false);
    if (! memHandle)
    {
        pFindParams->more = false;
        break;
    }

    const Char* pszString = (const Char*)memHandle.Lock();
    UInt16 wPos;
    if (FindStrInStr(pszString, pFindParams->strToFind,
&wPos))
    {
        if (! FindSaveMatch(pFindParams, wIndex, wPos,
0, 0, wCardNo, liDbID))
        {
            RectangleType rc;
            FindGetLineBounds(pFindParams, &rc);
            WinDrawTruncChars(pszString,
StrLen(pszString), rc.topLeft.x + 1,
rc.topLeft.y, rc.extent.x - 2);
            pFindParams->lineNumber++;
        }
        else
            break;
    }
    wIndex++;
}
return errNone;
}
```

You can read more about typical sysAppLaunchCmdFind handler in *"Palm OS® Programmer's Companion"*.

This implementation simply looks for matched *Name* attribute in application database. As previously mentioned, we cannot use stream classes without global launch codes. So, CMemHandle is used instead of CDBStream class.

CResource can simplify load any resources (i.e. strings, bitmaps, etc.) into memory. It automatically unlocks and frees used memory.

21. Add GotoLaunch() method implementation:

```
// Goto launch handler
Err CStarterApp::GotoLaunch(GoToParamsPtr pGoToParams)
{
    m_wGotoIndex = pGoToParams->recordNum;
    if (m_wLaunchFlags & sysAppLaunchFlagNewGlobals)
    {
        // The application must be executed by a normal way
    }
}
```

```
        m_Database.Open(pGoToParams->dbCardNo, pGoToParams->dbID);
        CForm::GotoForm(MainForm);
    }
    else
    {
        // The application already launched and
        // we can select record in the table
        CMainForm* pMainForm = CMainForm::GetInstance();
        CGrid& rMainGrid = pMainForm->GetGrid();
        rMainGrid.SetSelection(m_wGotoIndex);
    }
    return errNone;
}
```

As previously mentioned, `sysAppLaunchCmdGoTo` handler can be called if application is already active. You can check it by analyzing `CPalmStApp::m_wLaunchFlags` member. If `sysAppLaunchFlagNewGlobals` bit is set, the application was not active during find operation. In this case we must open database and show *Main* form in a normal way. We will select found record in the handler of `frmOpenEvent` according to `m_wGotoIndex` variable initialized in this function.

If `sysAppLaunchFlagNewGlobals` bit is cleared, the application is already active, and the database is already opened, so we must not reopen it. We can simply select found record in *Address* grid. `CGrid::SetSelection()` method will scroll the content of the grid if selected row is not visible on the screen.

`CMainForm::GetInstance()` static method will return pointer to single instance of `CMainForm` class (we assume that only one instance of *Main* form can be created). `CMainForm::GetGrid()` method will return reference to `m_gridAddress` member variable of `CMainForm` class. We will add `GetInstance()` and `GetGrid()` methods to `CMainForm` class in this tutorial step later.

22. Save your changes and close “*StarterApp.cpp*” window.

23. Open “*MainForm.h*” file from project window.

24. Add declaration of `GetInstance()` and `GetGrid()` methods to `CMainForm` class:

```
// Public methods
static CMainForm* GetInstance();
CGrid& GetGrid();
```

25. Add implementation of `GetInstance()` and `GetGrid()` methods below the class declaration:

```
inline CMainForm* CMainForm::GetInstance()
{
    return m_pInstance;
}

inline CGrid& CMainForm::GetGrid()
{
    return m_gridAddress;
}
```

26. Add `m_pInstance` member declaration to protected section of `CMainForm` class into control variables section:

```
static CMainForm* m_pInstance;
```

27. Add declaration of `CMainForm` constructor without parameters to public section:

```
// Constructor
CMainForm();
```

28. Add implementation of `CMainForm` constructor to initialize `m_pInstance` member variable:

```
inline CMainForm::CMainForm()
{
    m_pInstance = this;
}
```

29. Save changes and close “*MainForm.h*” window.

30. Open “*MainForm.cpp*” file.

31. Add pair `m_pInstance` declaration to beginning of CPP-file.

```
// Global form instance
CMainForm* CMainForm::m_pInstance = NULL;
```

32. Add code to select found record to `OnOpen()` method below `SetFocus()` function call:

```
// Select found record
UInt16 wIndex = CStarterApp::GetGotoIndex();
if (wIndex != -1)
    m_gridAddress.SetSelection(wIndex);
```

This code will select found record if the application was not active when find operation was performed and the application was started using `sysAppLaunchCmdGoTo` launch code.

33. Save changes and close “*MainForm.cpp*” window.

34. Select **Debug** command from **Project** menu or simply press **F5** key. The application must properly handle global find command. See an example below:



Congratulations! You've completed the latest step in this tutorial.

We hope that our POL framework will provide you with an actual project development technology for Palm OS platforms.

Good luck!

POL Team