The Essential

Basic4ppc

version 6.9

Ariel Zamir

Anywhere Software 2010 downloads and updates www.Basic4ppc.com

Table of contents

Part I - The Essential Basic4ppc

Introduction

Basic4ppc

Basic4ppc is a modern, easy to use, strong and simple programming language and IDE (Integrated Development Environment). The language targets both desktop Windows computers and mobile devices that run a Windows Mobile operating system: Mobile Phones and PocketPCs. Since desktop application can be easily developed using it, it is many times used when a program needs to run on both machines. Uniquely, code and project are 100% identical for both device (phone) and desktop – the **same code** that runs on your mobile phone can run on the desktop and vice versa (given the desktop application does not use the entire screen, of course…). Basic4ppc has gained a growing popularity during the last years. Developed early 2004, the language has since undergone major improvement and is used to develop business-standard applications, games, applications sold on electronic marketplaces and games. Basic4ppc supports modern database and graphics management, uses developer transparent memory management and offers libraries for GUI sometimes outgo those native ones supplies by Microsoft.

Basic4ppc is developed by Anywhere Software, which utilizes a model in which the IDE and the compiler itself are sold over the internet, while all additional libraries are supplied free of charge, most of which are open source. Many additional libraries are developed by users and are published on the Basic4ppc forum on Basic4ppc.com.

Basic4ppc is based on Microsoft's .NET technology – the latest standard from Microsoft. Applications written with Basic4ppc are then compiled to .NET executables (modern Windows .exe files) and can be distributed freely.

The History of Basic4ppc

Basic4ppc 1.0 was introduced in 2005. This early version had already had concepts such as the Device IDE and the Direct Naming References ("Runtime Controls Manipulation"). During the years, many features were added, such as the first standalone compilation (v.3, 2006) that initially was only a combination of a runtime and source at the same file, libraries support (v.4, 2006), smartphones support (v.5), real compilation, modules and AutoScale.

What can Basic4ppc do

Basic4ppc is a general purpose client application language. It is used for database applications, games, business applications and so on. You will be able to interact with servers, connect to peripheral devices, use networks and more.

Devices

Basic4ppc supports all devices running Windows Mobile. This includes a large variety of devices, from PocketPCs, mobile phones and special purpose devices. You can take advantage of feature inherently supported by windows (most of the features in most devices). However, some devices supply specific dlls for special features they have. These can be special sensors (G sensors in mobile phones, for example) and more. In some cases there are parallel Basic4ppc dlls that wrap the libraries supplied by the manufacturer. Otherwise, you might write your own dll and use it with Basic4ppc.

Desktop

Basic4ppc application can address Windows Desktops. This lets you write both stand alone applications to use on desktops, and applications that interact with one another, running both on device and on desktop.

Prerequisites

Supported operating system

Basic4ppc supports the following operating systems:

- Desktop: Windows Server 2003, Windows XP (all service packs, both 32 and 64 bit), Windows Vista, Windows 7, Itanium-based versions of these.
- Device
  Pocket PC 2002, Smartphone 2003, Windows Mobile 2003, Windows Mobile 2003 SE, Smartphone 2005, Windows Mobile 5.0, Windows Mobile 6.0 Standard (Smartphone), Windows Mobile 6.0 Professional (Touch Screen), Windows Mobile 6.1, Windows Mobile 6.5.

.NET

Basic4ppc is based on the Microsoft's .NET technology. In order to run both Basic4ppc itself and the applications you create, a Microsoft component called ".NET framework" must be installed on the target computer. The .NET framework is usually pre-installed on most modern desktops, PocketPCs and phones. If it is not, a message will pop when you run the program, indicating you should install it. The following paragraphs explain how to install the .NET framework on both desktop and device. The .NET framework component is downloaded from Microsoft website for free. It is called ".NET framework redistributable". Note: The links below direct you to version 2.0 of the .NET framework. Currently, this is the most common version and the only one that is required for Basic4ppc. However, more advanced versions exist, and you may be interested in them as well. If you already have a more advanced version, you do not need to install version 2.0.

- **Desktop**

  The following links allow you to download the .NET framework:

  - For x86 operating systems
    http://www.microsoft.com/downloads/details.aspx?FamilyID=0856EACB-4362-4B0D-8EDD-AAB15C5E04F5&displaylang=en
  - For 64 bit systems:
    http://www.microsoft.com/downloads/details.aspx?familyid=B44A0000-ACF8-4FA1-AFFB-40E78D788B00&displaylang=en
  - For 64 bit Itanium based systems:
    http://www.microsoft.com/downloads/details.aspx?familyid=53C2548B-BEC7-4AB4-8CBE-33E07CFC83A7&displaylang=en

- **Device**

  - The .NET framework for the device is different than the one for desktops. Differences mainly include reduced capabilities in some areas, less functions for programmers and reduced memory consumption. The .NET framework for the device is called .NET compact framework, or .NET CF. It can be downloaded here:
    http://www.microsoft.com/downloads/details.aspx?familyid=AEA55F2F-07B5-4A8C-8A44-B4E1B196D5C0&displaylang=en
  - The .NET SDK (Software Development Kit) is also required in order to compile programs for the Device. It can be downloaded here:
    http://www.microsoft.com/downloads/details.aspx?familyid=fe6f2099-b7b4-4f47-a244-c96d69c35dec&displaylang=en
  - In order to display proper error messages on the device, and additional file is required. If you see this message when debugging on the device:
    "An error message cannot be displayed because an optional resource assembly containing it cannot be found", then you know you should follow the

instructions in this forum thread: http://www.basic4ppc.com/forum/4461-post10.html

Downloading Basic4ppc

Download Basic4ppc at http://www.basic4ppc.com/Downloads.html. Links on the website will direct you to the .NET framework and some additional software as well.

Why Basic4ppc

Basic4ppc gains a growing popularity during the last couple of years. Amongst the most important reasons to use it are:

- Basic4ppc is simple. It is deriving from the BASIC language syntax which has inspired hundred of languages worldwide (actually, one of the two Microsoft's .NET leading languages, Visual Basic .NET, is based on the BASIC syntax as well). This lets you both learn the language easily, and maintain code written previously (this becomes a significant advantage when you get back to a project you created just 6 months ago…).
- Baisc4ppc is straightforward. It supports modern programming paradigms such as Event driven programming, but spare you the hassle of many rarely used features that make life complex.
- Basic4ppc is strong. Its rich variety of extension libraries allows you to create almost anything.
- Basic4ppc is modular. It lets you reuse your code using modules and libraries, thus enhances productivity.
- Basic4ppc is mobility oriented. Device IDE, 100% code identity (desktop/device), and AutoScale are just a few of the features that ease your life when you create mobile applications.
- Basic4ppc is based on Microsoft's .NET technology. This means it supports modern memory management and that programs written run on many operating systems.

- Basic4ppc has tens of extension libraries, and more are constantly written. Official libraries are always free (usually published along with their source code) and will always be. Once you purchase Basic4ppc, you get all future libraries for free, even after your free updates period (of the IDE) is over.
- Forum. Basic4ppc has a very strong, supportive forum. If you are not a forum-guy naturally, you may consider it again – the forum is where things happen. This is the place where new announcements are made, libraries are published and questions are being answered. More than one industry standard project owes its progress to the advices its developer got on the forum!

How a Basic4ppc program runs

After a program is written, there are two ways by which it can be run.

Interpreter

When you run a program via the IDE using the "Run" option, Basic4ppc first runs a rough verification to ensure the program can be run. Then, it runs an interpreter that performs your program a line at a time.

Compiler

When choosing the "Compile" option in the File menu, the program is compiled to a .NET executable. This executable can be run on every machine with the .NET framework installed (as stated previously, this includes all new devices and computers running Windows). A compiled program is much faster than a program run interpreted.

Who this book is for

This book is for any programmer using Basic4ppc. The purpose of this book is to be a fundamental guide to Basic4ppc, covering concepts, basic language structures, commonly used techniques, and important additional libraries. Beginners will find a way to learn Basic4ppc, and more advanced users may use it as a reference.

The Basic4ppc Forum

The Basic4ppc forum is the place where a community of friendly developers assists one another. This is the place where people post their source code, questions, suggestions and comments. People come and go all the time. Some of the users are very experienced and are responsible for a great contribution to the community. This is also the place where new libraries – an essential part of any program – are published. This is a calm place, worth visiting anyhow, even just to get the feel. The language is English. Sub forums exist in German, Spanish, Italian, Portuguese, French, Russian and Chinese. Even if you are not a forum kind of guy, this one is worth checking out, as it is an essential source of help for most users at some point. You will find many references to this forum along the guide.

How to use this book

Terminology

- **Device** refers to either mobile phone, PocketPC, or any other device that runs a Windows Mobile operation system of any kind (in short, not your desktop/laptop).
- **Desktop** refers to both laptop and desktop – anything that runs Windows.

Forum posts in this guide (I tried to include as many as possible, because they reflect user opinions), are preceded by the word "Forum".

Credits

Major parts of Basic4ppc libraries and an irreplaceable part of the accumulated knowledge are due to the prospering community. The community is mainly alive on the Basic4ppc

forum. As aforesaid, this is a place you want to check with any problem you have.
Community members whose libraries are referenced along this guide are mainly

1. First of all – known by the username Agraham, Andrew Graham of UK, probably the best known, most fertile contributor to the forum and an excellent programmer for himself
2. Filippo of Schwäb. Gmünd, Germany. Author or many important user controls.
3. Klaus of Switzerland – a most central forum member, happy to assist with an endless patience and a real professional. A reference to his excellent Graphics tutorial is given in the Graphics chapter.

We owe great appreciation and a huge Thank You to many-many others – I cannot mention everyone but the community is waiting on the Forum!

Some code examples and many topics included in this guide are taken from the forum, with credits where appropriate.
Many of the libraries developed for Basic4ppc are written by users. Some of which are referenced to in this guide. The credit is always given to the author and copyright is always as defined by author. Anyhow, absolute majority of these are open source, for free use. They are all available for download at the Basic4ppc forum under the Additional Libraries category, or under the Official Updates, if they had been published directly by Anywhere software.

Had any mistake been made crediting authors and the wonderful developers of this forum, please accept our apology. Rest assured it is in good faith. Please let us know about it and we will try to fix it as soon as possible.

Beginning Basic4ppc

The purpose of this chapter is to make you familiar with the essentials of working with Basic4ppc. There are 5 topics – just what you need to start.

It assumes no previous knowledge. Experienced programmers might find it to packed: here's a list of what you can get of each part and what to skip:
- **Downloading**: technical details about where to find what.
- **First program**: very basic, for your first interaction with Basic4ppc.
- **First program revised**: Hello World with GUI: you can skip if you have experienced visual programming languages (VB, C# and so on).
- **Compiling and testing on device**: important to know.
- **Using the device IDE**: unless you've done it, worth a look.
- **Getting help**: very important in Basic4ppc.

Downloading:

- Trial version

  Free version (trial) is available for 30 days (evaluation period). It has all functionality the full version has, except it does not compile into EXEs. The full version includes this functionality. Download them here: http://www.basic4ppc.com/Downloads.html.
- Purchase Basic4ppc here: http://www.basic4ppc.com/Purchase.html.
- Installation under Windows Vista:

  When running under Windows Vista it is recommended to install Basic4ppc under My Documents. For security reasons, Windows Vista does not allow programs to access folders under the Program Files folder. Installing under the My Programs folder (**and please note this is the default)** will cause some conflicts, some of which may be solved by running Basic4ppc as an administrator. Forum:

http://www.basic4ppc.com/forum/questions-help-needed/5021-problem-compiling.html#post29213.

- Please also refer to the Prerequisites section in the first chapter for information about how to get the .NET framework and the .NET CF, if you need it.
- Associating files extension to Basic4ppc

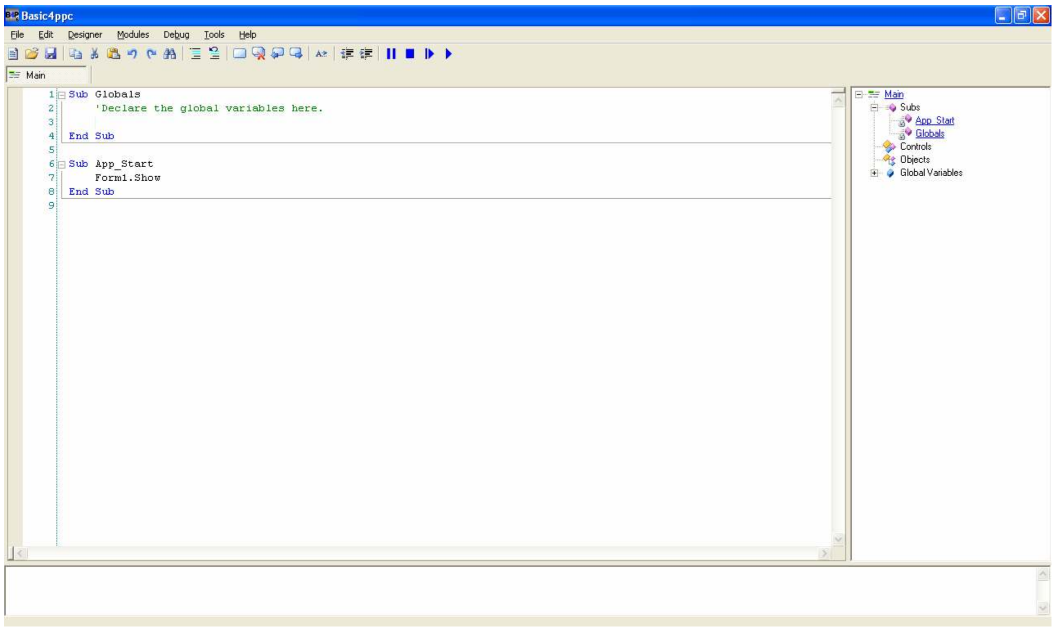
  Many program associate files with given extensions with themselves, so that the right icon is shown next to the file and when double clicking the files opens with the right application (for example, Microsoft word associates .doc files to itself). This forum thread explains how to do this on your device with Basic4ppc source code files (.SBP): http://www.basic4ppc.com/forum/code-samples-tips/1150-add-file-association-basic4ppc-source-code-files-device.html. It is not a must, but it helps.

First program - A simple "Hello world"

Purpose: understanding the basic concepts of a Basic4ppc program.

Writing code:

- Open Basic4ppc:
- A screen appears:



```
1 Sub Globals
2     'Declare the global variables here.
3
4 End Sub
5
6 Sub App_Start
7     Form1.Show
8 End Sub
9
```

The code in the window shows:

```
1   Sub Globals
2       'Declare the global variables here.
3
4   End Sub
5
6   Sub App_Start
7       Form1.Show
8   End Sub
```

Replace line 7 with the following code (write it in “Sub App_Start”, before the End Sub statement):

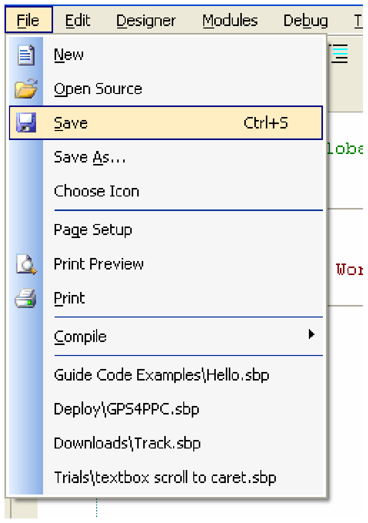
```
7       MsgBox("Hello World")
```

So that your code now should appear:

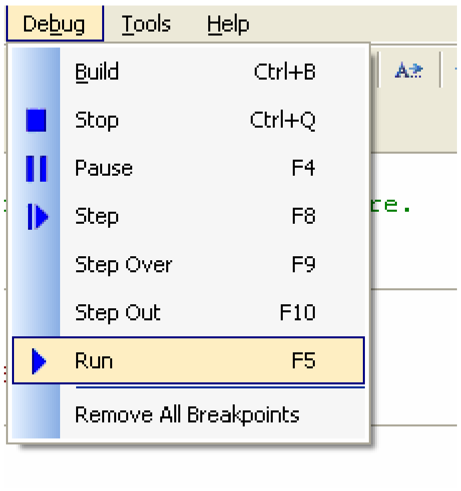
```
1   Sub Globals
2       'Declare the global variables here.
3
```

```
End Sub

Sub App_Start
    Msgbox("Hello World")
End Sub
```

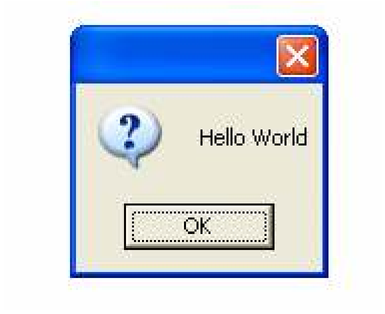
Now go to File – Save. Choose a folder, give you program the name “Hello” and click Save.



Now your program is ready to run. Go to Debug menu, and click the Run option (as you see in the menu, you could have pressed F5 instead, or hit this icon at to top toolbar:



If everything went ok, you should now see a message stating "Hello World" (having problems when running on Windows Vista and up? See the previous section in this chapter):


Hello World

Some other changes have appeared on your screen, but this is the most important ones. The others are about to be covered later.

Click the OK button. The program stops.

Analysis

Now, we will analyze the program a line at a time, to explain what each statement does.

```
1    Sub Globals
```

The first line starts with the reserved word **Sub**. A reserved word is a word with a meaning to Basic4ppc. Such a word cannot be used as a variable name or for any other use rather than the predefined use. A list of Basic4ppc reserved words is available on the Help file (main help). Generally, you should get used to working with the help files (a more complete explanation about using the help is at the end of this chapter).

A Sub is a composition of some statements. A statement is the most basic unit of code you can write. All Basic4ppc code is written inside Subs. The first thing that happens when your program runs is the first statement in a special Sub, called Sub App_Start (App stands for Application). All of this will be explained thoroughly later, and is mentioned here just to give you a brief idea of what's going on.

The first Sub, however, can be another special Sub (special in the means that its name is meaningful for Basic4ppc, whereas regular Subs you can name whatever you want). This is the Sub called “Globals”. Its role will be explained later (it stores variables that any other Subs can use, in short) but it does nothing here (it’s empty), so there are only few things to say.

```
2       'Declare the global variables here.
```

The second line starts with the character ‘ (apostrophe). A line that starts with an apostrophe is a comment – the compiler ignores it when creating the program from your code. In this case, Basic4ppc tells you what should be placed inside Sub Globals.

```
3
4     End Sub
```

Third line is empty – empty lines are ignored.
Forth line says “End Sub”. This indicates the place where Sub Global ends.

```
5
6     Sub App_Start
```

Fifth line is empty and ignored.
Sixth line declares the beginning of the main Sub in any program: Sub App_Start. This is the programs “entry point” – the place from which the program starts.

```
7       Msgbox("Hello World")
```

The Seventh line is the actual program. Notice this is the only line we have written ourselves. Msgbox stands for Message Box, which is a standard way Windows platforms let us display information to the user. This line actually tells your computer, "Use the default operating system's messaging system to display the note "Hello World" to the user". On Widows systems, you then get a message with the text you entered and an OK button. The program reaches this line, displays a message box and waits for you to click OK. Only then it proceeds to the next line. This is the way Msgbox operates and it is not standard, because usually lines are executed in the order they are written without the program waiting for user input.

Note, that the message text is contained inside brackets. This is very important, because this indicates to Basic4ppc that the words "Hello World" are **a string:** text that should be treated as one peace.

```
8    End Sub
```

The Eighth line ends the main Sub. When reaching this line, the program checks if it has anything else left open. It has no other UI, so it closes - but this is rare. Many programs just start here. If it has anything else, it waits for any other active user interface peaces to close, so it does not necessarily stop when it reaches the Sub App_Start's "End Sub" statement.

Save your work by selecting File – Save or pressing Ctrl+S (your program is automatically saved when you run it, but it is a good habit).

First program revised: Hello World with Graphical User Interface (GUI)

Concept

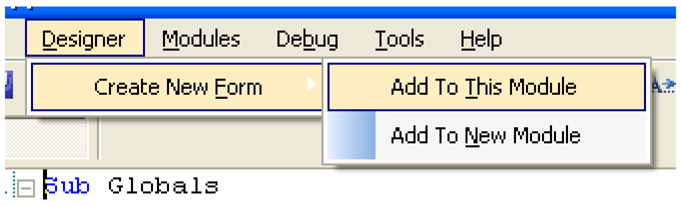
The program you built on the previous section is simple and demonstrates some fundamentals of Basic4ppc. However, one basic thing is not shown – how to create GUI. Creating GUI is a basic task in modern programming. You should be familiar with the basic terminology there is on this task and the basic process involved in the building of GUI: the graphical designer part, creating a form, placing controls and using events and properties. I will explain the meaning of these terms on the go.

Planning the program

Lets say we just want a very simple hello world example, just with a more complex user interface than the single line, message box based user interface we created the previous section. When the program runs a button will appear, and when you press it a text will show saying “Hello world”.

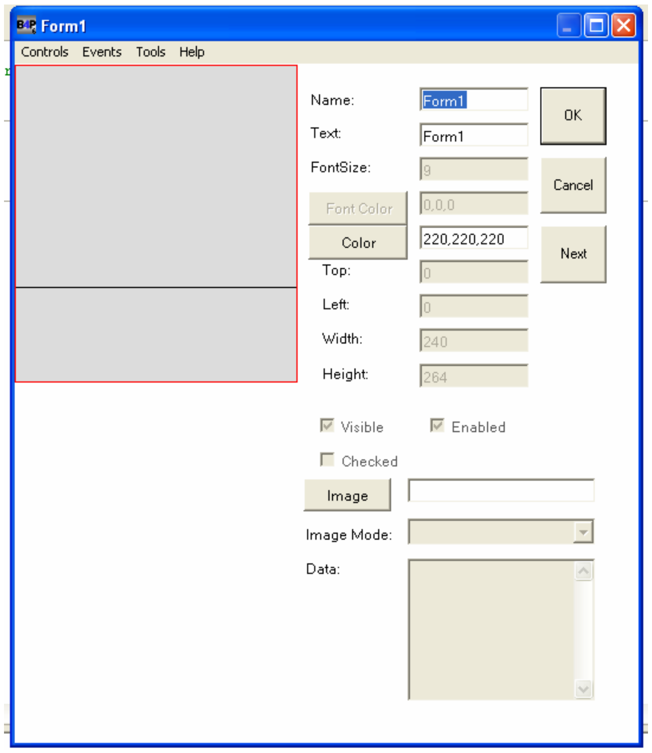
Designer

Open Basic4ppc. If it is already opened, save your work, and go to File, New. A new program appears with the regular template. Go to Designer – Create New Form – Add To This Module:



```
Sub Globals
```

The Designer Window appears:



Name: Form1
Text: Form1
FontSize: 9
Font Color: 0,0,0
Color: 220,220,220
Top: 0
Left: 0
Width: 240
Height: 264
Visible ✓ Enabled ✓
Checked
Image Mode:
Data:

We will discuss it in details later, but for now what you need to know is this:

- On Windows, every window you see is called "Form". Even the small message box you created on the previous example is displayed on top of a form (the small window where it resides).
- The visual designer shown here lets you graphically edit the form you are about to display.
- You draw things on the form and then you display it. This way you can make changes to the form when it is hidden and display it all at once.
- You place your drawings on the form in two main ways: directly drawing on the form, which is rarely needed (but lets you control every bit, and will be discussed in a later chapter) or, most commonly, by placing <u>controls</u> on the form.

<u>Controls in a nutshell</u>

Controls are basic concept you should understand. In short, a control is a component of a program, previously created (by whom? could be anyone). You take this small “program component” and add it to your program, thus taking advantage of what it can do. And what is it doing? This can be anything. Usually controls are used as UI (User Interface) elements, so many controls display something on the screen and allow some sort of user input. For example, a control called Button displays a button on the screen and lets the user press it. When the user actually presses it, a special Sub, called “**Event**”, is “called” (that is, run). You define what’s going on inside this Sub, so you can actually write code that runs whenever the user clicks the button.

Every control has its own unique name. This way you can have more than one button – one is called button1, another one is called btnPressToStop and so on.

In order to set what is written on the button, or the color of it, and so on, you need to be able to interact yourself, as a programmer, with the code someone has written for you. This is done using Properties, which are a special way to set values to the control, or get the values the control already has, for certain characteristics of it. For example, the text of a Button control is set by changing the Text property this button has. This is shown later.

An **Event** is the Sub that is called when something in the control happens. Every control has its event(s). In this example we will use an event called “Click” that belongs to the Button control.

A **Property** is the name of the variable that belongs to a control, and has a special meaning. Changing a property causes something to change in the control. The general way to change the property is to write something like:

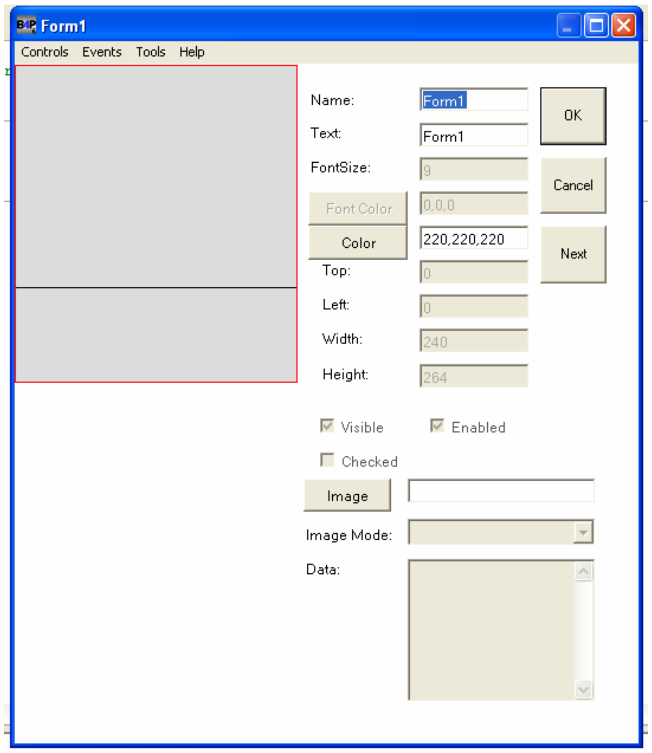
[control name].[property name] = [value]

Note the colon separating the control's name from its property. For example:

button1.Text = "Click to Toggle"

This will show the text "Click to Toggle" (without the brackets) on the button.

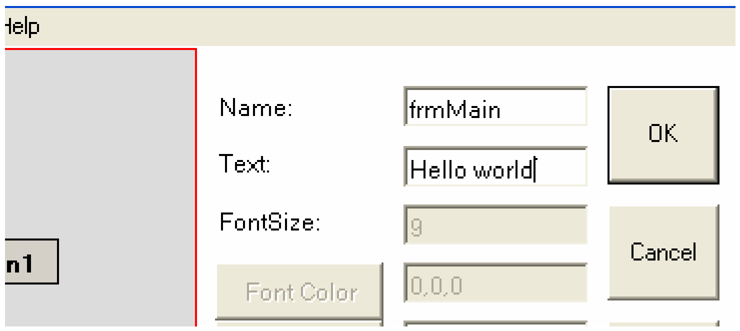
Adding controls, events and properties is possible both on code and on the visual designer window. It is much easier to use the designer, hence we always prefer doing so. However, it is not always possible. Adding a control is done in the designer window, from the Control menu, changing the properties is done by inserting a value to the property to the right, and adding an event is done from the event menu. The grey area to the upper left corner represents the form itself – this is how your screen is going to look like on the device.



Name: Form1
Text: Form1
FontSize: 9
Font Color: 0,0,0
Color: 220,220,220
Top: 0
Left: 0
Width: 240
Height: 264
☑ Visible ☑ Enabled
☐ Checked
Image Mode:
Data:

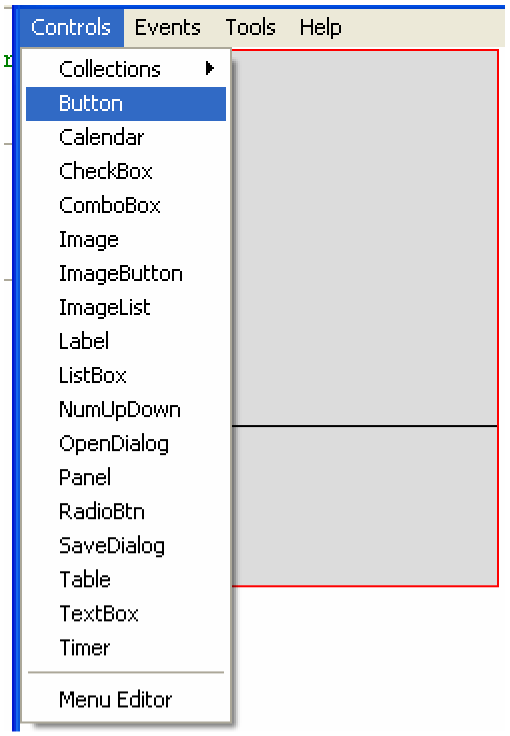
Naming

Giving meaningful names is a good habit when programming. On the designer window, change the default name "Form1" to something meaningful, such as frmMain (frm is short for Form. This naming convention, of preceding the name with a three letters abbreviation of the type of control, is very common). Change the Text property of the form to "Hello world" – this will be the caption on the top bar:

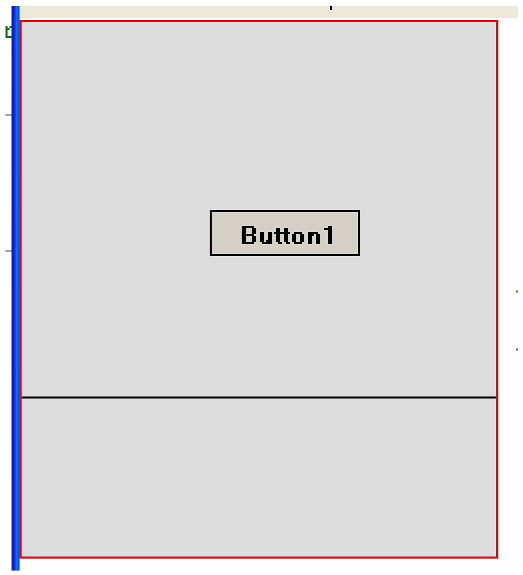

Name: frmMain
Text: Hello world
FontSize: 9
Font Color 0,0,0

Adding controls to the form

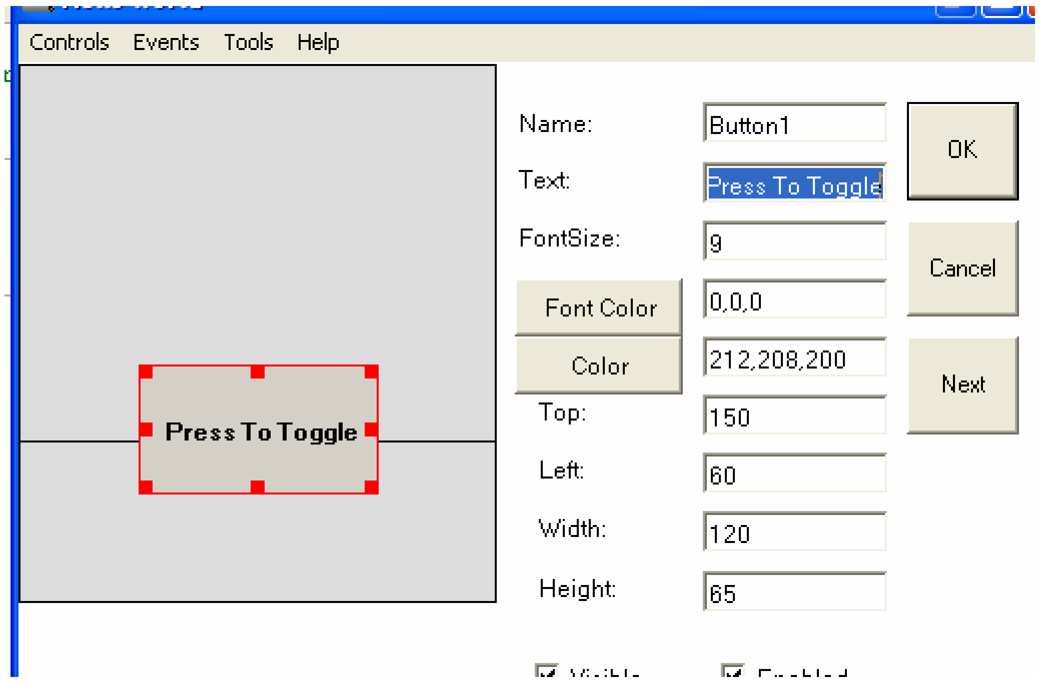
Now, add two controls to the form. Go to the Control menu, and select "Button":



A rectangle with the word "Button1" appears on the grey area representing the form:

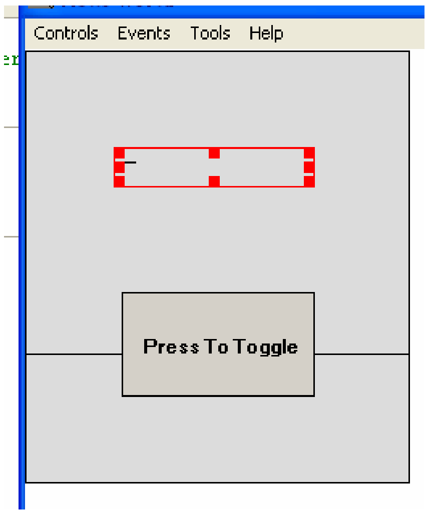


Click the middle of the button. Small red rectangles appear. Click and hold down in the middle of the button, and drag it with your mouse to a lower place on the screen. Then, click one of the rectangles and drag to change the size. When done, change the text property to the right so that it shows "Press to Toggle", and press enter. Resize to fit. Eventually, your form should look like this:



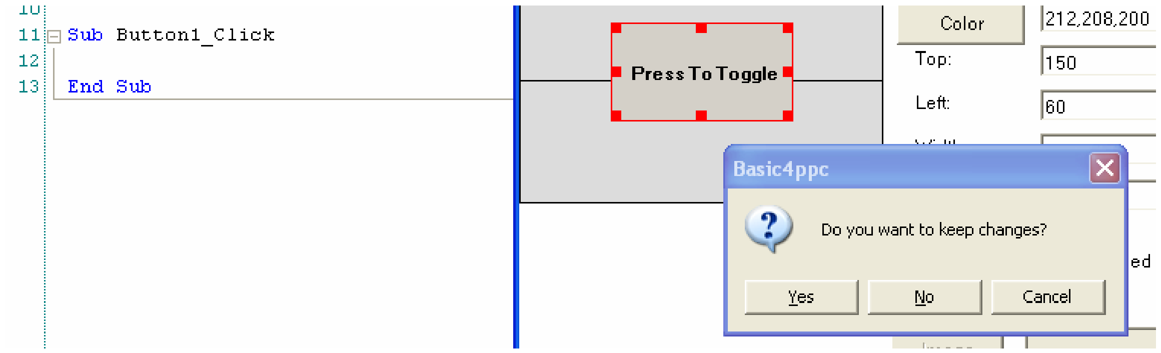
Name: Button1
Text: Press To Toggle
FontSize: 9
Font Color 0,0,0
Color 212,208,200
Top: 150
Left: 60
Width: 120
Height: 65

In a similar way, add a control called label. Set the text property to show "---" (We do it so that we do not lose it by setting an empty text, though we can find it of course):



Note we didn't change the names of the button and the label. What we changed was the text they show – do not confuse these properties.

This is all you need for the first UI you create. The only thing left is to add an event: the code that's run when you click the button. Select the button by clicking it (so that it has red rectangles). Now, click the Events menu, and choose Click (this will add a new Sub, called "Button1_Click", to your code!). When you click it, Basic4ppc assumes you wish to write the code for it, so it closes the designer window, but asking you if you want to save your work first. Click yes.



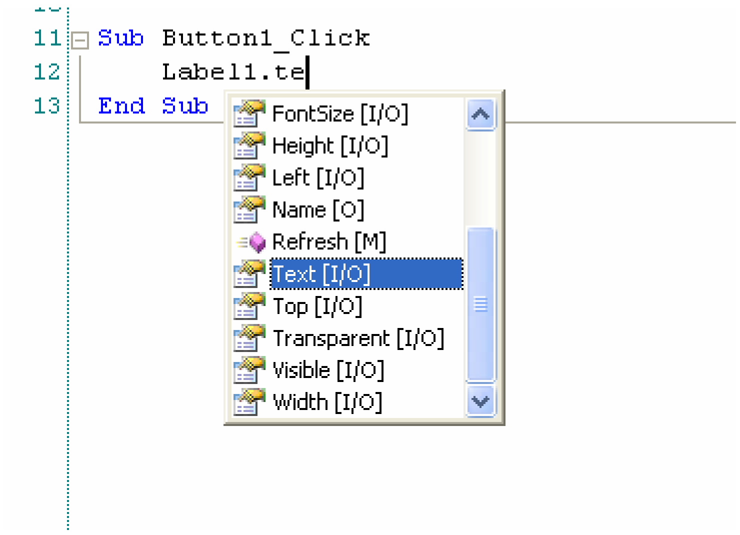
```
Sub Button1_Click

End Sub
```

One more thing to notice is that since you changed the form's name to frmMain, line 7, which was "Form1.Show" is now "frmMain.Show". Basic4ppc does not always change the names you change along the code, but in this case it does.

On line 12, write the actual code of your program:

Label1.Text = "Hello World"

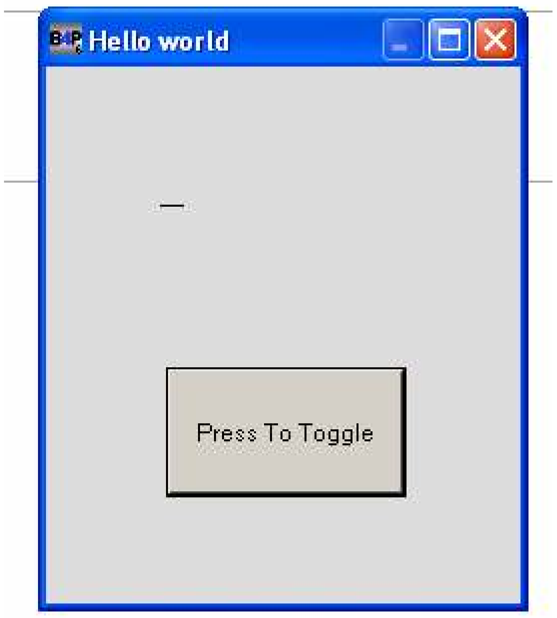
Note, that when you start typing the first letters of the word Text, a list appears. This is the AutoComplete feature:



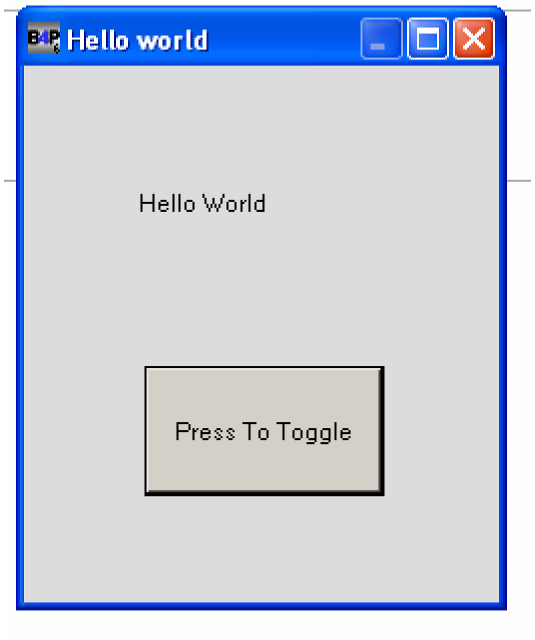
```
Sub Button1_Click
    Label1.te
End Sub
```

Press Tab, or space, or "=" (equals), or most non alphanumeric keys, and the word is displayed on the screen.

Run the program by pressing F5. When asked, save it under Hello 2.A window appears:



Click the big button:



The "Hello World" caption appears.

<u>What happened?</u>

The program started just as the previous one, and there is nothing special to say until line 7:

```
7       frmMain.Show
```

Every control and form has, apart from properties, also <u>methods</u>. Methods are actually Subs owned by the control. By specifying their name you can "call" them – that is, cause them to run their code. In this case we called the Show method of the form "frmMain". This method, as its name implies, shows the form on the screen. Program execution now reached line 8:

```
8       End Sub
```

And as explained earlier, Basic4ppc checks if it can stop the program now. Since there is still a displayed form, the program waits. When you clicked the button, it called the Sub with the name "Button1_Click". This Sub looks like this:

```
11    Sub Button1_Click
12       Label1.Text = "Hello World"
13    End Sub
```

Of course, the interesting is line 12: this line takes the string literal “Hello World” and places it in the Text property of the control with the name Label1 (which happens to be the label you placed on the form). Usually, changing the Text property of a control (for ones who have it) immediately causes the displayed text to change, and this is what you see here.

Compiling and testing on the device

Running your application on the desktop

Basic4ppc offers a unique way to test your application by running it on a form, under normal Windows, that is just small enough to emulate your mobile device. It has some primary properties Windows Mobile forms have, and many times it is more than enough to test your application there. It is especially convenient for testing logic, and user interface design issues, under rapid development conditions (and this is, lets face it, most of the time). Yet, some features cannot be tested on the desktop – for example, the mobile file system is different, there is no Today Screen, you cannot test for different hardware and on top of all, performance, generally, is much better. On the other hand, it is fast, convenient, easy, allows you to develop desktop applications, and, unique to Basic4ppc application, lets you run any application you have written both on the device and on the desktop with no changes.

Emulators

The second way to test your application is to use an emulator. An emulator for your device is a program that emulates many features, otherwise hard to test, on your device. It shows an image of the device and interacts with the computer as if it was a device. You download it on Microsoft website:

- Windows mobile 6.0 developer kit (includes emulator):
http://www.microsoft.com/downloads/details.aspx?familyid=06111A3A-A651-4745-88EF-3D48091A390B&displaylang=en
- Windows Mobile 6.5 SDK (requires the previous):
http://www.microsoft.com/downloads/details.aspx?FamilyID=20686a1d-97a8-4f80-bc6a-ae010e085a6e&displaylang=en

Working with emulators lets you test issues with file system, screen resolutions, and more. Although being a bit slow, it is very helpful. On the Basic4ppc forum there are some tutorials, written by the community member Ghale, about working with emulators. They cover much knowledge about installation and usage:

- Using the Emulator: http://www.basic4ppc.com/forum/tutorials/4707-basic4ppc-windows-mobile-emulator.html
- Using Cell Phone Emulator: http://www.basic4ppc.com/forum/tutorials/4715-windows-mobile-emulator-cellular-emulator.html
- Using GPRS: http://www.basic4ppc.com/forum/tutorials/4716-windows-mobile-emulator-configuring-data-connections-using-gprs.html#post28969

Anyway, using the emulator requires you to compile your program and copy it to your emulator as if it's a device. Go on reading and see some tips for this.

The best place to save your program

Windows communicates with Windows Mobile (in other words, the desktop communicates with the device – be it a cell phone, a pocketPC, or any other device running WM) using one of two software:

- Under Windows XP - ActiveSync, a program that keeps your device synchronized with a specific folder on the desktop. It checks for updates periodically, and if the device is connected (either with a cable or wirelessly) it synchronizes both sides. It works like this: you choose a folder (or it is created for you, usually) and everything in it is copied back and forth.
- Under Windows Vista and later, Windows Mobile Device Center does exactly the same, with a cooler name.

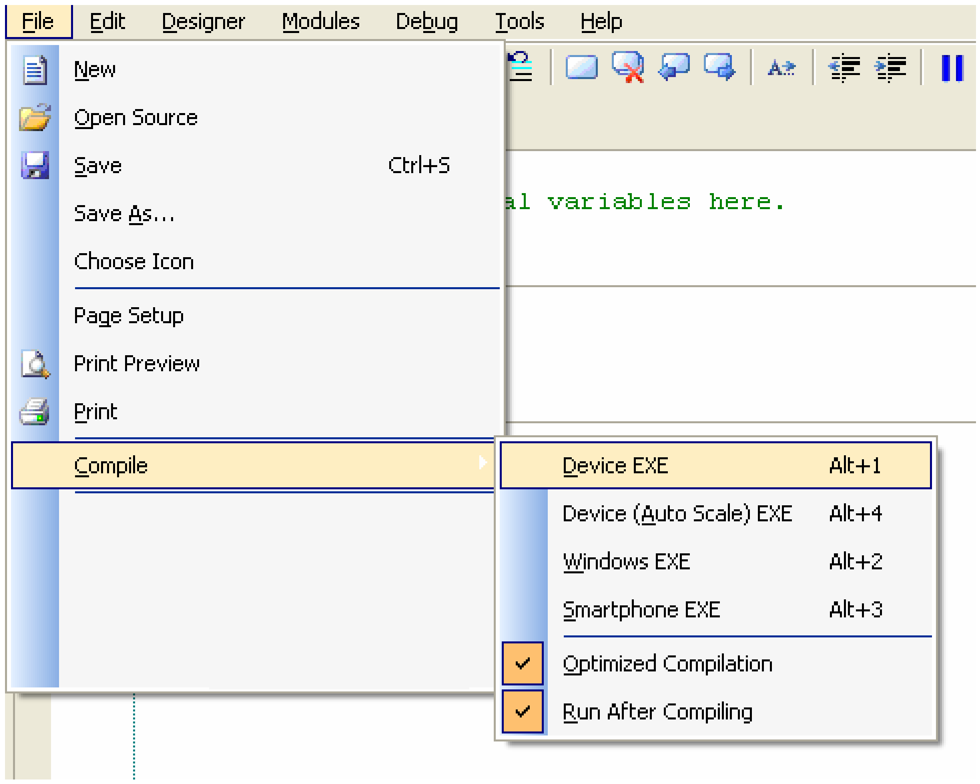
See the tutorials above for screenshots. They are on the forum. Did I mention the forum yet?

When you compile your program (that is, translate it to assembly, the machine language the computer speaks), Basic4ppc creates a copy of your program with the extension .exe for executable. This announces the operating systems it can run this file directly. The file is saved somewhere you choose when compiling (see next section).

So, the best place for you to save your newly created exe, would be on a sub-folder under your shared folder. Basically you might want to save all sources and perform all development under your shared folder. If you don't have a synchronized folder, create it using ActiveSync or Windows Mobile Device Center. As your application grows it depends on external files like libraries, images and data files. Assuming that you build your code from both the desktop and the device, it is very important that the device files and the desktop files are up to date. By working under synchronized folder, the files will be synchronized between the device and the desktop, and when you compile, you save it on this folder and magically it is copied to your device, ready to run!

Compiling

Compilation, as aforementioned, is the process of taking your Basic4ppc code and translating it to machine language. To do it, go to the File menu, choose Compile and select the target of this application:



The differences between the options are obvious: Device EXE is an executable file for the device, Smartphone is for Windows Smartphone edition, Windows EXE is for desktop application. Auto Scale is explained in details on the Graphics section. For now, choose Device EXE (or press Alt+1), or if your device has 480X640 screen, choose Device

(Auto Scale) EXE. Then save the file under your shared folder (a dialog appears for this), and you can run it from the device within seconds.

The Device IDE

Basic4ppc offers a very unique feature: the device IDE. This integrated development environment offers most of the features the Desktop IDE offers: Designer, Editor and so on. There are some limitations – you cannot compile, to name one, and the screen is smaller, but you can develop and run your program right on the phone or pocketPC you are working on – this is good either to program on the go or just to write your code where it is run.

To start the Device IDE make sure you installed “Basic4ppc device.exe” on your device. Then run Basic4ppc from your device and start working. Start by writing the two sample programs above on your device – it’s a good practice and you’ll get the feel. However, along this guide, and for convenience reasons only (mainly because it is easier to understand the screenshots), we will use the Desktop IDE.

Getting Help

When coding, it is very common to need help. The main source for help is the help menu, described below. Apart from it, there are some other sources for help you should know.

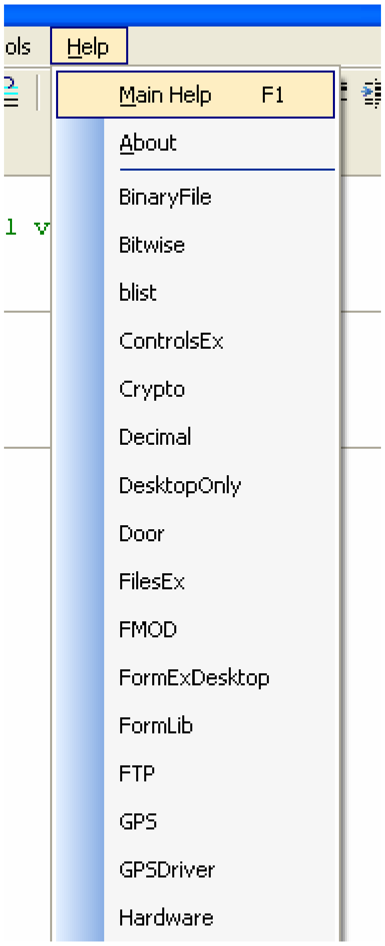
Basic4ppc includes important help sources. In order to keep it as simple as possible, there are conventions you should be aware of as to where to look for help.

A note about help files: If you are not usually a “help reading” guy, this is the place to start. Basic4ppc’s help is written short, straightforward and with only few unnecessary words – just in order to make it easy for people like you to learn.

Help is built in "Help files". A help file is a file with the extension .CHM. It is installed under the Basic4ppc folder. Every help file covers a topic, as will be described below. Additional libraries contain their own help, so that you don't need to browse through irrelevant files.

Help menu: Contains the Main help, and a help topic for each installed additional library (see the libraries chapter for more information about libraries help). Some of these help files are found online as well. Whenever possible, it is recommended to consult the online help (on the Basic4ppc website:
http://www.basic4ppc.com/help/index.html)

- **Main Help**: under the help menu in the IDE (either desktop or device), you will find the main offline help. This help file is the main source for the essentials of Basic4ppc.
- **Libraries help**: each library that is deployed with a help file complying with the Basic4ppc conventions (see Libraries chapter) is listed on the help menu.
  The Basic4ppc help is intended to be the first source for knowledge – use it.
  The figure below shows a list of installed help files.



Help menu

- **Online help**: the main help is found on the web in the aforementioned link. Apart from this link, you can find some more resources in the Basic4ppc documentation center: http://www.basic4ppc.com/Documentation_Center.html
- **Tutorials center:** for Subjects requiring extra learning that is not suitable for the help files, there is a list of tutorials found here: http://www.basic4ppc.com/Tutorials.html. The list is not long but new ones

are built from time to time. However, this guide is supposed to cover most of the topics included in them.

- **Tutorials on the Forum**: an important source for tutorials, including both user-written and official tutorials. As said earlier, the forum is an essential resource. This is the tutorials section: http://www.basic4ppc.com/forum/tutorials
- **Online Reference list**: links to all Basic4ppc keywords, commands, and controls help, and to most important libraries: http://www.basic4ppc.com/specifications.html#list
- **Basic4ppc Frequently Asked Questions**: http://www.basic4ppc.com/faq.html
- **Videos**: for some important topics there are videos to learn from, each 2-7 minutes long (average 4): http://www.basic4ppc.com/Video_Reference.html
- **The Basic4ppc forum** is the very essential place you should know. And again – if you are not a forum kind of guy, this is the time to start. Professional developers share their knowledge and huge amount of code is shared – don't hesitate. The forum is the place to download code samples, to ask questions (about anything – any newbie question is gladly accepted), ask for assistance, for graphics or for reviews of your software, to share what you wrote, and so on. There are forums in English, Russian, Portuguese, Italian, French, Spanish, Chinese and German. Of course, the English one is the most vital and the one with best chances to get a quick answer.

The forum is also the place to download additional libraries. Only registered users can download additional libraries (only those who purchased Basic4ppc) but anyone can post in the forum and get assistance.

These are the important parts of the forum:

Main page: http://www.basic4ppc.com/forum/

Questions and help (English): http://www.basic4ppc.com/forum/questions-help-needed/

Official libraries: http://www.basic4ppc.com/forum/official-updates/

User created libraries: http://www.basic4ppc.com/forum/additional-libraries/

Important: How is a help file built

There is a convention for the structure of a help file. If you know the convention it's easier to read it:

- It starts with an overview – what this help file covers (usually the main object in the file).
- There is a detailed explanation about each procedure or function, either on its own page or altogether.
- Many times some code samples are included.

And just to get you started…

Some newbie questions from the forum:

http://www.basic4ppc.com/forum/questions-help-needed/3959-some-basic-questions-about-basic4ppc.html#post30890

Program Flow

Programs you write are built of **statements** – things you tell the computer to do.
The most basic thing there is to know about how the program works is the order in which your commands – statements - are executed. Basic4ppc is a procedural, event driven language. This means there are two ways by which your code is executed. You write your code in “Code blocks”, and they are executed a line at a time, from top to bottom. Every piece of code you write is held inside a Sub, which is the basic code unit. Some Subs are called “Events” (it is explained later) and are executed by the operating system as a response to something that happened “outside” your code – for example, you can write a Sub that is executed (this means, the code you wrote inside) every time the user clicks a button, or every time a file has finished downloading.

A code block is sequence of lines of code, which is executed one at a time. The following is an example:

```
1   Sub Globals
2       'Declare the global variables here.
3
4   End Sub
5
6   Sub App_Start
7       Msgbox("Line 7")
8       Msgbox("Line 8")
9       Msgbox("Line 9")
10      Msgbox("Line 10")
11  End Sub
```

Look at the code in the App_Start Sub – if you run this program, you will get four messages saying “Line 7”, “Line 8”, “Line 9”, “Line 10”. The reason is that the program (any Basic4ppc program) starts at the first line in the App_Start Sub, and goes line by line until the last line there.

:

Active statement

The line that is currently being executed is called "Active Statement". It is very important to control which statement is executed – this is the way to control what you program does. In the sample above the active statement actually tells you which one it is by popping a message out, but of course this is usually not the case.

Statements and lines

We have discussed statements so far as if each of them is a separate line. But you can:

- Write two statements in one line, if they are separated by a colon:

  ```
  Msgbox ("Line 7"):Msgbox ("Line 8")
  ```

  This will display both messages.
- Write on statement in two lines, using the underscore character at the end of the first:

  ```
          Msgbox _
              ("Line 7")
  ```

  This will act just as if they were on the same line.

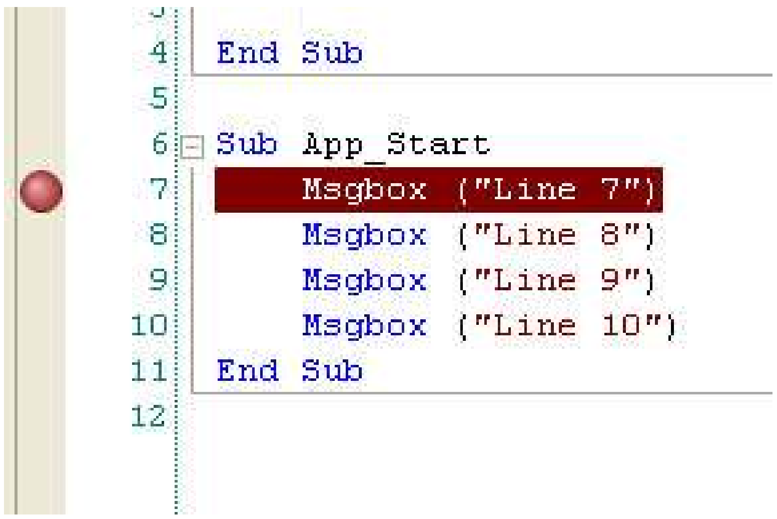
- Comments: preceding the character ' (apostrophe) to a line or a part of a line causes Basic4ppc to disregard everything that is to the right of the character. This is use for comments in the code and for times when you need to check some different alternatives.

Debug example

A good way to see in real time how it works is to trace your program execution using the debugger. The debugger shows you the active statement marked with yellow, and lets you go on statement by statement:

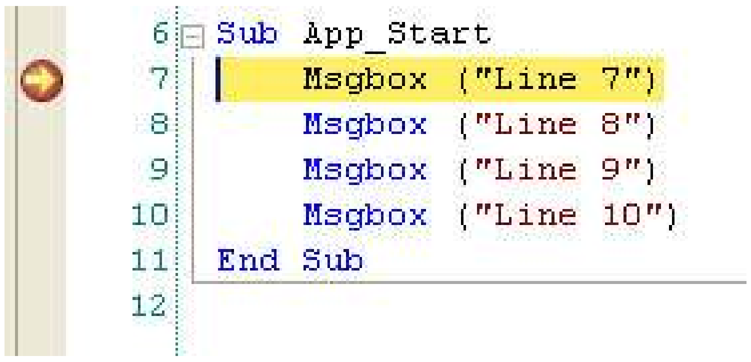
Program

Start by creating the simple program from the screenshot below. I added a **breakpoint** by clicking once on the white tab to the left of the first line in the Sub:



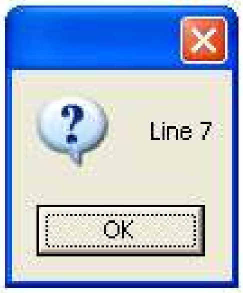
```
End Sub

Sub App_Start
    Msgbox ("Line 7")
    Msgbox ("Line 8")
    Msgbox ("Line 9")
    Msgbox ("Line 10")
End Sub
```

Now, run the program by pressing F5. The brown line is suddenly highlighted with yellow, and there is a small arrow pointing it:

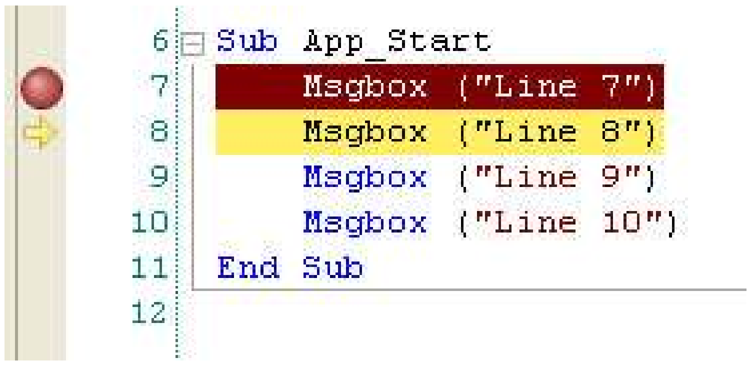


```
Sub App_Start
    Msgbox ("Line 7")
    Msgbox ("Line 8")
    Msgbox ("Line 9")
    Msgbox ("Line 10")
End Sub
```

Press the F8 button – this tells Basic4ppc "Execute this line and move to the next". Of course, you will get this message box:


Line 7

Click OK. The next statement is highlighted:


```
Sub App_Start
    Msgbox ("Line 7")
    Msgbox ("Line 8")
    Msgbox ("Line 9")
    Msgbox ("Line 10")
End Sub
```

And so on. Try yourself – hit F8 several more times to the end of the program and see where the active statement is. This is a very simple example, but this way of tracing the active statement is a very efficient way to track down bugs in your programs when they get complicated.

Everything is in Sub

When writing In Basic4ppc, every piece of code you write must be included in a Sub. A Sub is a set of code lines that has a defined name and can be executed together. It is also recommended to have a distinctive, simple functionality for each Sub, and give it a significant name. Thus, Sup App_Start is the place where your application starts. This is called **the program's Entry Point.**

A Sub's text starts at line 1 and goes to End Sub. The code you write inside the Sub starts executing from the first line and goes a line at a time, as shown above, to the End Sub statement. This statement indicates the place after which a new Sub may start. One advantage of using Subs, is that if you write the Sub's name as a statement (as we wrote "Msgbox"), you cause the lines inside the Sub to be executed (this is called "Calling a Sub"). Actually, **Msgbox** is a Sub predefined for you by Basic4ppc (by Windows, to be correct).

Special Subs

Some Subs are special. This means you cannot name other Subs with their name, and that they have a predefined role in a Basic4ppc program.

- Sub Globals: This is the place where you place **variables** you want to access from every point in your program. Further discussion about it is in the variables chapter.
- Sub App Start:
  This is the Entry Point of the program. The first statement here is the first to execute. When the active statement reaches the End Sub of this Sub, there are two possibilities:
  - If there is nothing else left "alive" (such as user interface shown and waiting for the user to interact with and so on) the program ends.

    - If there is anything left, the program waits for “things to happen”. This is implemented using **events,** which are Subs that are called when “something happens”.

- Events

  An event is a Sub that is called by some “component” in your program. In the second “Hello World” program you wrote the previous chapter you had a button on a form – this button is a good example of a component. When programs became too complex, they started to use this kind of “Sub programs”, sometimes referred to as “components”, “controls” and “libraries”. One common way to know when something happens inside a component is to use the events it has – this is the way external components tell you “something happened”. You used the Click event of the button in your code, to know when the user clicked it. The concept is demonstrated at length along this guide.

Control program flow

Controlling the program flow is one of the most important things to do when designing a program. We will introduce just some of the basic ways to do so in a nutshell: a further discussion is in the following chapters.

- Goto

  One infamous way to control where the next statement is, is using the Goto statement. It has this form:

  ```
  Goto [Name-Of-Label-You-Define]
  ```

  And it causes the next statement to “jump” to the label. A label is just a word you write in a separate line, and it is followed by a colon. These are labels:

Label1:

lblMainProgram:

ErrorHandlingPart:

A label must start with a letter and comply with the rules for variables names, and is followed by a colon. The next example shows how you can use Goto:

```
Sub App_Start
LabelNumberOne:
        Msgbox ("Line 7")
        Goto LabelNumberOne
End Sub
```

This will cause the message “Line 7” to appear endlessly (try…). However, using the Goto statement is not recommended as it tends to cause extreme complication of code that is very hard to maintain. Try to avoid it if you can and use loops, Sub calls and if…then…else structures instead.

Note: Do not confuse the colon following a label with the one used to separate two statements in one line!

- Calling a Sub

  Calling a Sub is a good way to cause something to happen. The following is an example of how to call a Sub in code:

```
Sub Globals
    'Declare the global variables here.

End Sub

Sub App_Start
    ShowMessage
End Sub

Sub ShowMessage
    Msgbox ("Inside Sub")
End Sub
```

Here, in line 7 we call the Sub “ShowMessage” that is declared in line 10.

- IF … Then … Else

  The IF statement is used to control the flow of a program in a conditional way. For example:

```
Sub Globals
   'Declare the global variables here.

End Sub

Sub App_Start
   If 1 = 2 Then
                 Msgbox("Something is wrong.")
   Else
                 Msgbox("Still in control.")
   End If
End Sub
```

The IF statement will be further discussed later. It is presented here only in the context of general flow control statements.

Variables, Values and Expressions

It is one of the most common scenarios that you need to make the computer “remember” a value of something. Even if you want the computer to just “count to 10” you still need it to remember what the last number was. The same way, when you want to remember the user’s name you need to store it somewhere. This is what variables do. A variable is a word in your program that holds a value. You decide what will be its name, and you set the value to it using the assignment operator “= “. For example:

```
 1  Sub Globals
 2      'Declare the global variables here.
 3
 4  End Sub
 5
 6  Sub App_Start
 7      i = 10
 8      j = 300
 9      K = "Hello World"
10      l = True
11      index = 10
12      indexNumber2 = 10
13      indexNumber3 = j
14      indexWithAConfusingName = k
15
16      Msgbox(i)
17      Msgbox(j)
18      Msgbox(k)
19      Msgbox(l)
20      Msgbox(index)
21      Msgbox(indexNumber2)
22      Msgbox(indexNumber3)
23      Msgbox(indexWithAConfusingName)
24  End Sub
```

Here, in lines 7 – 14 I set values to different variables. I gave them the name I wanted: there are some rules but basically you can name them anything. So I started off with just simple, commonly used “indices” names i, j, k, l. Not very meaningful, so I moved on to something with more meaning: line 11 gives the value 10 to the variable with a name "index".

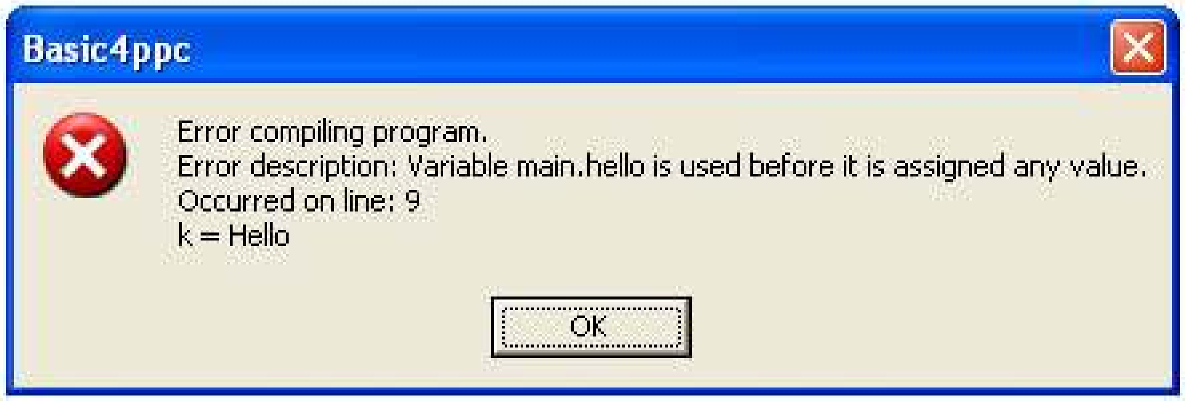
Note, that in line 9 I didn't write

k = Hello world

but

k = "Hello World" (note the brackets!)

If I didn't, Basic4ppc would have thought the word "Hello" is a variable. This is confusing, Basic4ppc would have thought, and remembering it does not allow using variables before they are assigned a value, it them pops a message:



```
Error compiling program.
Error description: Variable main.hello is used before it is assigned any value.
Occurred on line: 9
k = Hello
```

There are some good reasons why this is not allowed, but you can turn of this feature in the tools menu. It is not recommended to turn it off as it makes your programs more human-error-prone. It's there mainly for backward compatibility.

Anyhow, notice I did assign the value "True" to a variable without brackets – this is because "True" and "False" are special words with special meaning (keywords).

What else could we learn from looking at the program above? One thing is that I set values to the variables the same line where I first said "there is a variable with this name" (I actually say so implicitly: I just assign the value to it). One other thing, is that I can assign different types of values to variables: it could be a number, a name (string), or the value

"True" (or "False") which is a word Basic4ppc knows (a "reserved word" used in conditions evaluation). This is because variables in Basic4ppc are **typeless** (starting from v. 7 you can declare types – see at the end of the chapter). They hold any data and (with some limitations) try to figure out what you meant to do when you ask for anything. This is very important: you can assign any type of information to a variable, but since you may want to treat them differently (say, you want to be able to multiply numbers, but not words), there are different ways to use the information inside – as if it represents a word, a phrase, a number, a date or a Boolean (true/false) value.

Names of variables

- Naming

  Naming variables is up to you. Use any word you want, apart from the Basic4ppc reserved words (keywords) which can not be a variable name (they already have meaning. The list of keywords is in the main help.
- Rules for naming

  A variable name must start with a letter. It must be composed only of the following characters: A-Z, a-z, 0-9, and underscore (_). No spaces, no brackets, no weird characters, please.
- Case insensitivity

  Variables names are case insensitive. This means that both "Index", "index" and "indEX" refer to the same thing.

Scope

Every variable you use is "known" only inside the Sub you first declared it in, and nowhere else. Exception: if you want a variable to be known all over your module (about modules, see modules chapter), declare it in **Sub Globals.** This excellent example is taken from the help file. You must turn off the option “Check for unassigned / unused variables” under the tools menu in order for this to run:

```
Sub Globals
	a=20
End Sub

Sub App_Start
	b=10
	CalcVars
End Sub

Sub CalcVars
	Msgbox ("a = "  &  a)
	Msgbox ("b = "  & b)
End Sub
```

The result might surprise you: the first msgbox will show **a = 20.**

The second msgbox will show **b =**

b is empty because it's local and wasn't assigned any value yet in this Sub. The range of code inside which a variable is “known” is called the variable’s **scope**. So we say the scope of any variable is the Sub in which it was first assigned value. If you declared a variable in the Sub Globals, its scope is the entire module. In the modules chapter we will discuss ways to cause the scope to be even wider – cross module.

Declaring

Unlike many other languages, Basic4ppc does not demand you to declare variables explicitly. Declaring is done just by assigning a value to a new (previously undeclared) variable.

Literals (or, how values are written in my code?)

When we say "assign value" to a variable we actually mean, that the value you specify is to be stored under a special "title" – the title being the variable's name. So there must be a way to write the value in the Basic4ppc code in a way that tells Basic4ppc this is a value.

Well, there is. The most common types of information you store in variables (this is called sometimes "simple variables" or "primitives" or "value variables") are numbers, text, and Boolean values (why True/False). A value written in your code explicitly is called "a literal".

- Strings (that is, text) are written in your code enclosed in brackets:
```
UserName = "John Smith"
UserLastName = ""    ' note: now UserLastName is empty!
```
- Numbers appear just as the are:
```
UserAge = 50
Pi = 3.1415
```
- Boolean values appear as the are:
```
IsUserMale = True
DoneProcessing = False
```

Assigning other variable's value

To set the value of a variable to a different one, just write the assignment without any further notations:

```
UserName_Previous = UserName
UserAge = UserPreviousAge
i = j
j = index
```

Note, it all works well if the variables UserName, UserPreviousAge, j and index where declared previously (that is, assigned values). Otherwise, depending on the option aforementioned in the Tools menu, you might get an error message. If this option is turned off, every new variable not previously assigned is considered to be empty.

Expressions

You can assign the value of something more complicated than just a variable or a literal. These are expression:

```
i = j + 5
```

This expression tells Basic4ppc to calculate the value of j + 5 and put this value in the variable i. Say j is 10, so j + 5 = 15, and after this line i is also 15. Applying this logic you can write this as well:

```
j = j + 5
```

Although weird at first glance, this tells Basic4ppc to calculate the value of
j + 5 and put the result back in j.

Expressions can use the operators in Basic4ppc, and can also include values returned from Subs. Say a Sub named Max is written this way:

```
Max (x, y)
```

and returns (that is, equals after it's executed) the value of the higher of the pair (x, y). So the statement

```
x = Max (x, y)
```

will cause Basic4ppc to take the higher value, and put its value inside the variable x. The expression

```
x = Max (x, y) * 2
```

Will do the same, but will multiply the result by 2 before assigning it to x.

But since everything is typeless, if I wrote something like:

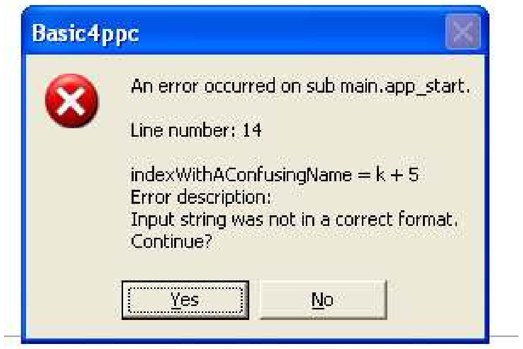
```
Username = “5”    ‘ not very likely, but definitely allowed!
UserAge = 10
```

what happens if I write:

```
x = username + userage    ‘ note the case insensitivity
```

The answer is that **the type of an expression is defined by the type of the operator**, and the result is assigned to the variable. An expression is composed of an operator (the “+” sign, in this case) and operands (the numbers the “+” sign “expected” to get). And the “+” operator returns a numeric value. So let’s see what Basic4ppc does: it takes UserName, which holds a string “5” (not a very good name but yet), and tries to convert it to a number. It succeeds, and gets 5. Then it takes the UserAge variable which holds the value 10, and applies the + operator on both. The result is kept in a variable called x.

But what if there was something else in the UserName variable? Say, “John”? Then Basic4ppc would have tried to calculate “John” + 5 and an error would have happened. The error message says something quite unclear, and this is the reason why you should be familiar with it: “Input string was not in a correct format”. This does not give you much information (though the error message indicates the line you should suspect). Generally, this error pops when something in the expression could not be handled, Basic4ppc tried to treat everything as strings (they are the most basic so if nothing works it tries…) and failed:



An error occurred on sub main.app_start.

Line number: 14

indexWithAConfusingName = k + 5
Error description:
Input string was not in a correct format.
Continue?

In this case, this means one of the variables in the expression (or more than one) could not be dealt with.

Operators you can use in expressions:

Operator	What it does	Example

+	Adds two numbers	X = 2 + 4
		Y = x + 5 ‘now y = 11

-	Subtracts two numbers	X = 2 * 4 ‘8

*	Multiplies two numbers	X = 3 * 5 ‘ 15
		Y = X * X ‘ 225

/	Divides two numbers	X = 8
		X = X / X ‘ 1
		X = 0
		X = X / X ‘ ERROR

^	Power	X = 2 ^ 4 ‘ 16
Mod	Modulus operator: the remainder of a division	X = 10 mod 3 ‘ 1

&	Strings concatenation: adds two string variables to one	X = “A”
		Y = “B”
		Z = x & y ‘ “AB”

		i = 5
		J = "A"
		k = i & j 'Note: since everything is typeless the type of the result is defined by the type of the operator and the operands are converted. So here i is converted to "5" and k = "5A"

And	Boolean And. Boolean operators are used in Conditional statements such as IF and Select.	X = True And False 'False
		
		Y = 5
		X = 2=2 And y=4 'False

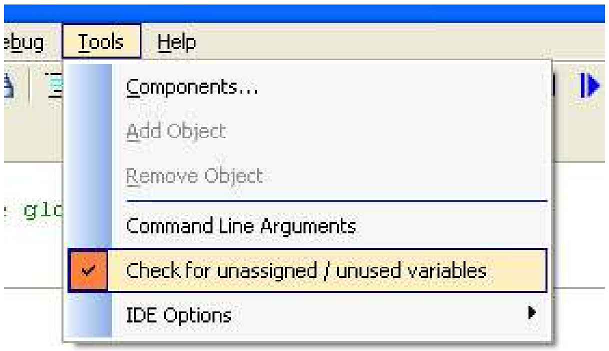
Or	Boolean Or. Boolean operators are used in Conditional statements such as IF and Select.	X = True Or False ' True
		Y = 5
		X = Y = 5 Or Y = 4 'True

Not()	Boolean Not. Boolean operators are used in Conditional statements such as IF and Select.	X = True
		Y = Not (X) 'False
		Y = Not (2 = 2 Or X) 'False

()	Parentheses change the order of precedence inside an expression.	X = 2 * 4 + 8 ' 16
		X = 2 * (4 + 8) ' 24

Initial Values

Any variable that was not assigned any value is empty (it is treated as an empty string if a string is needed, or as 0 if a number is needed). However, if you keep the "Check for unused / unassigned variables" option on, your program will not compile if you try to use an uninitialized variable.



Variables and Computer Memory

This section will not try to cover even a slight part of the topic. It is intended to give a general idea about the implications of storing values in computer memory and the overhead included in converting types.

All variables are stored in the computer memory as sets of binary numbers, composed of 0's and 1's. Basic4ppc takes care of the conversion needed from the sets of 1 and 0 to the meaningful text you need, but you should be aware that:

1. It takes memory and memory is limited. It is not very likely that you run out of memory on the desktop, as today's computers have huge amount of memory, but it is not rare on the device. It is very rare to run out of memory just because you used regular variables, but if you try hard enough you can. Beware the consequences of initializing endless number of

variables (especially on recursive Subs), and beware more when using images, which tend to be memory consumers.

2. It takes time to use variables. The fact that variables are invariant (typeless) has a meaning – they have a conversion overhead which, on intense scenarios, might result in a performance penalty. This is rare and happens on graphics intense games, when writing process-extensive UI and so on). Versions prior to 6.9 must use either typed array or the math recompiler library written by the user Agraham on the Basic4ppc forum here: http://www.basic4ppc.com/forum/additional-libraries/4981-mathrecompiler-performance-enhancer.html

Gaining performance: Declaring variables with explicit values – Basic4ppc v. 7 and up

Starting from v. 6.9 Basic4ppc allows you to declare types for variables and for values returned from subs. Doing so is done with the following syntax:

Dim indNum As Number - indicates indNum is a numeric variable (with decimal places)

Dim strName As String - indicates strName is a string variable.

Dim intInd As Integer - indicates intInd is a whole number.

Note:

- Number indicates a floating-point-precision number, whereas Integer indicates a whole number. Number is faster – use it when you can.
- Declaring a variable as a string has the same meaning as not specifying a type.
- A Sub's returned value type can be indicated the same way:

    - Sub Max (x, y) As Number

- Sub's parameters types can be indicated the same way:
    - Sub Max (x As Number, y As Number) As Number

Using numbers rather than untyped variables could speed up calculations up to 10 times (this is true for compiled programs, not when running in the IDE).

Some more links and questions from the forum about this topic are here:
http://www.basic4ppc.com/forum/beta-versions/5494-performance.html#post32361

http://www.basic4ppc.com/forum/beta-versions/5501-what-variable-type-declare.html#post32390

Subs

A Subroutine ("Sub") is a piece of code. It can be any length, and it has a distinctive name and a defined scope (in the means of variables scope discussed earlier). A Subroutine is called "Sub" in Basci4ppc code, and is equivalent to procedures, functions, methods and subs in other programming languages. The lines of code inside a Sub are executed from first to last, as described in the program flow chapter.

Declaring

A Sub is declared in the following way:

```
Sub [Sub-Name-You-Choose]
    …
    …
    …   code …
    …
End Sub
```

It starts with the keyword Sub, followed by the Sub's name, and ends with the keywords End Sub.

Subs are always declared at the top level of the module – that is, you cannot nest two Subs one inside the other.

Calling a Sub

When you want the lines of code in a Sub to execute, you simply write the Sub's name. For example:

```
1  Sub Globals
2          a=20
3  End Sub
4
5  Sub App_Start
6          b=10
7          CalcVars
8  End Sub
9
10 Sub CalcVars
11         Msgbox (“a = “  &  a)
12         Msgbox (“b = “  & b)
13 End Sub
```

In this example, line 7 calls Sub “CalcVars”. The active statement will jump to line 11 – the first in the Sub CalcVars. **When the active statement reaches line 13 (End Sub), it jumps back to the line after which the Sub was called from –** in this case, line 8. It is very important to understand this part. Consider the following:

```
1  Sub Globals
2          a=20
3  End Sub
4
5  Sub App_Start
6          b = 2
7          CalcVars
8          b=10
9          CalcVars
10 End Sub
11
12 Sub CalcVars
13         Msgbox (“a = “  &  a)
14         Msgbox (“b = “  & b)
15 End Sub
```

In line 7 we call CalcVars, and when its execution ends (that is, the active statement reaches line 15) we then jump back to the place from which we were called + 1 line – the active statement is now line 8. On line 9 we call CalcVars again, and this time when we reach the end the active statement will jump to line 10.

Naming

Basically, you can name a Sub any name that's legal for a variable. It is recommended to name the Sub with a significant name so you can tell what it does from reading the code. There is no limit on the number of Subs you can add to your program, but it is not allowed to have two Subs with the same name <u>in the same module.</u>

What are Subs for

Subs are useful for a couple of purposes. First, they reduce the amount of code you need to write: in the sample above I wanted to display two message boxes after each change to the value of b, so I called a Sub that does just this with one line. Second, if tomorrow I wish to show 5 messages per each change, all I need to do is to change the Sub by adding three more lines to it and the change takes place in multiple places. Third, Subs are the basic unit of a structured program. Structural programming is the breaking of code into small units, each holds a significant name and each carries out a significant, isolated task. Fourth, reading code that is divided into Subs is much easier (the issue of readability is very important once you work on your program more than one week. You see the name of the Sub and you say to yourself, ok, I already know what this means). And fifth, when you have a Sub that's worked well on a program you wrote, you can use it again and again in many other programs thus sparing the time it takes to specify, write and debug.

Returned value

A very useful feature about Subs, is that a Sub can return a value to the calling statement, and this value is then used as if it was a variable. The example below demonstrates this idea:

```
1   Sub Globals
2       'Declare the global variables here.
3       X = 0
4       Y = 0
5   End Sub
6
7   Sub App_Start
8       X = 1
9       Y = 800
10      Msgbox (MultiplyXYAndAdd30 + 10)
11  End Sub
12
13  Sub MultiplyXYAndAdd30
14     Return X * Y + 30
15  End Sub
```

In lines 2 – 3 we declare two global variables so that we can access them from within the Sub. Then we set values to them in lines 8-9, and then, in line 10, come the big thing about this program: we call the message box function (msgbox) with the name of the Sub as a parameter. What should happen?

Well, the active statement jumps to line 14, where I wrote the keyword "Return". This keyword tells Basic4ppc something like: "Calculate the expression following, and go back to where you were called from, where the Sub's name shall be replaced by the value calculated". So it multiplies X and Y and adds 30, to get a result of 830. This value is then returned to the message box and only then, the message box is shown.

Types of returned value

The value a sub returns is typeless by default. Anyway, similar to assigning value types to variables, starting from v. 6.90 you can set the type of the value returned from the sub as described in the variables chapter. The syntax is:

- Sub Max (x, y) As Number

The reason for this is mainly performance – see the variables chapter for a more complete discussion.

Parameters

Using the global variables to pass values into the Subs is a good way, but it is considered "risky". The reason is, that you don't know where else in your program you might have used these values (this is really the case once you have more than these 15 lines of code!) and if you need to change them inside the Sub and only then return a value, you might find yourself in troubles. Some other Sub might have changed the values already. So the best way to pass values into a Sub is using parameters. Parameters are a list of values that a Sub is receiving from outside and that cease to exists when the active statement is no longer inside the Sub. Their names are written in parentheses after the Sub's name in the Sub XXX declaration. The values you put in are written when you call the Sub. This is a much better way to write the same program:

```
Sub Globals
    'Declare the global variables here.

End Sub

Sub App_Start
    X = 1
    Y = 800
    Msgbox (MultiplyXYAndAdd30(X, Y))
End Sub
```

```
11
12 Sub MultiplyXYAndAdd30 (A, B)
13    Return A * B + 30
14 End Sub
```

There are few things to note:

- There are no global variables – this say structured programming rules are kept.
- The Sub's name is the same, but it has two parameters: A and B.
- When calling the Sub on line 9, we set the value that X has (set on line 7) to the parameter A, and the value Y has to the parameter B.
- When calling the Sub, you type "(" for the Sub's parameters. This is why there is "))" at the end of the line – you have two parentheses to close.
- The Sub multiplies these two parameters, adds 30 and returns result.

But what if I called the Sub's parameters X and Y, as the variables in the App_Start Subs are called?

The answer is that parameters have **scope** just like variables. So if instead of A and B I had X and Y, nothing would have happened, because the Sub would refer to its local X and Y rather that to any other external X and Y. This is also true if there **are** global variables X and Y – **when I have parameters with the same names as something else outside, the Sub always uses its parameters.**

Passing parameters by values or by reference

When you pass variables as parameters and change their values inside the sub, the original value of the variable is not changed. For example:

```
Sub Globals
   'Declare the global variables here.

```

```
4   End Sub
5
6   Sub App_Start
7       varX = 10
8       Msgbox (varX)  ' 10 is shown
9       ChangeX(varX): Msgbox (varX)  ' 10 is still shown
10  End Sub
11
12  Sub ChangeX(X)
13     X = 20
14  End Sub
```

Follow the program's flow – even though the variable varX is passed as a parameter to the ChangeX sub (lines 12-14), and the value of the X parameter is changed, this does not affect the value of varX. The reason is that the original value is copied – this is called passing variables **by value.** Had the value been change, we would have said the variable is passed **By Reference**.

Starting from v. 6.90, it is possible to pass variables as parameters to subs either by value or by reference:

- when you pass variables normally, or prefixed by the keyword **ByVal**, the variable is copied as shown above.
- If you prefix the variable name in the sub's declaration with the **ByRef** keyword, and the parameter's value is changed in the sub, the passed variable's value is changed as well. The last program can be rewritten as following:

```
Sub Globals
    'Declare the global variables here.

End Sub

Sub App_Start
    varX = 10
    Msgbox (varX)  ' 10 is shown
```

```
9      ChangeX(varX): Msgbox (varX)  ' 20  is shown
10  End Sub
11
12  Sub ChangeX(ByRef X)
13    X = 20
14  End Sub
```

Note the ByRef keyword in line 12.

Assigning data types to variables

Starting from v. 6.90, it is possible to set the data type of subs parameters just as it is to variables (starting at the same version – see description in the variables chapter). The most common types are Number (recommended), Integer and String. The difference is described in the variables chapter as well as other points of attention. The syntax in Sub declaration is as follows:

```
1   Sub Globals
2     'Declare the global variables here.
3
4   End Sub
5
6   Sub App_Start
7     varX = 10
8     Msgbox (varX)  ' 10 is shown
9     ChangeX(varX): Msgbox (varX)  ' 10 is still shown
10  End Sub
11
12  Sub ChangeX(X As Number, Y As Integer, ByRef T As String)
13    X = 20
14  End Sub
```

Note that the ByRef is added (line 14) only as a sample of the combination and is not necessary.

Recursive calls

A recursive call is the situation where a Sub calls itself. Consider the following:

```
Sub Globals
    'Declare the global variables here.

End Sub

Sub App_Start
    X = 2
    Y = 2
    Msgbox (Power (X, Y))
End Sub

Sub Power (Number, Pow)
    If Pow = 1 Then
                    Return Number
    Else
                    Return Number * Power (Number,
                    Pow – 1)
    End If
End Sub
```

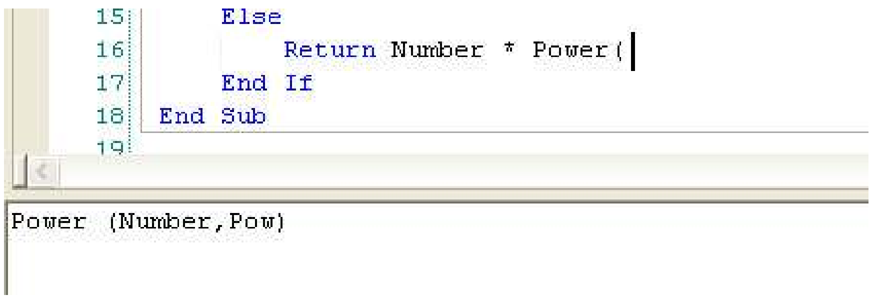
Though we did not discuss the IF statement yet, it is very easy to follow the Power Sub. It takes two parameters, Number and Pow (which stands for power, but the name Power is taken by the Sub…). If pow is 1 it returns the number itself (because $x^1 = x$). If the power is not 1, it takes 1 off the power (Pow – 1) and executes the power calculation again – by calling itself with (Number, Pow – 1). The result is then multiplied by the number (to compensate for the "-1") and returned further. For example, when we call it with 2, 2: Power (2, 2) will return the value "**2 * Power(2, 1)**". **Power (2, 1)** will return 2 (because of the IF statement) and only then the value of the pending expression ("**2 * Power (2,1)**" can be evaluated as **2 * 2 = 4**.

Values that cannot be passed (as parameters) **or returned** (as returned values)

You have seen how to pass and return simple variables to and from Subs. However, Array, Structures, Controls and Objects, all will be discussed later, cannot be either passed or returned from a Sub. You must use the method shown in the beginning of the chapter if you need to use them inside Subs – that is, declare them as globals.

Help when typing the parameters of a Sub

It is sometimes very confusing to figure out the list of parameters you need to pass to a certain Sub. Basic4ppc offers you the "help-on-the-fly" feature, which displays the lists of parameters as you type it at the bottom of the screen:



```
        Else
            Return Number * Power(
        End If
    End Sub
```

```
Power (Number,Pow)
```

Arrays

What is an array

An array is a list of variables, all sharing the same name. Lists are very useful when programming – think of the list of the songs in your mobile phone. You can place them all in an array called Songs, and access each on using an **index**. Referring any of the array's members is done using its index, and from this point on you act as if it is a regular variable. This is the general structure of and array with 10 cells, of which 4 are empty:

Cells holding values	song 1	Song 2	Song 3	O bla di	O bla da	Bra				
Indices	0	1	2	3	4	5	6	7	8	9

Note the index. The index of the first cell is 0, and the index of the 10 cell is 9. This is always the case: in every array, the index of the n-th cell is n-1.

Declaring an array

In order to declare an array you use the keyword "Dim". Unlike simple variables, arrays **must** be declared. The declaration must be done in Sub Globals. Arrays are global and can be referred to just like any other variable that was declared in Sub Globals. You set the array name, and the number of items you want it to have in parentheses:

```
Dim songs(10)
```

Referring to items

When you need to refer an item, you write the array’s name and the item’s number in parentheses:

```
     Songs(0) = “The Future/Leonard Cohen”
     Songs(1) = “The End/The Doors”
```

and so on. The following example demonstrates something a bit simpler. It uses the **FOR loop** which was not yet discussed here, so it will be explained briefly after the code:

```
1   Sub Globals
2       ‘Declare the global variables here.
3       Dim a (10)
4   End Sub
5
6   Sub App_Start
7       a(0) = “AAA”
8       a(1) = “BBB”
9       a(2) = 3
10      a(3) = 4
11      a(4) = a(2) + a(3)
12
13      For i = 0 To ArrayLen(a()) – 1
14                  Msgbox (a(i))
15      Next
16  End Sub
```

- Line 3 declares the array. It declares an array with 10 items. As we said, the first items has index 0. So when the program starts on line 7, it sets the string value “AAA” to the first item. Then, values are set to the next items in the array. On line 11 the values of a(2) and a(3) are added to one another and the result is inserted into the item with index 4 (the fifth item!).

- The FOR loop in lines 13, 14 and 15 is just a way to repeat what's in the middle 10 times. It actually means something like:
    - Take a variable i.
    - Give it the number 0.
    - Perform the lines from here to the "NEXT" statement (line 15) until the variable i equals the expression "ArrayLen (a()) – 1" (This expression equals 9, and this will be explained right away. Meanwhile, treat it as if it was 9).
    - Every time you do it, increment i by 1.

So the result of the whole thing is that the line

```
Msgbox (a(i))
```

Is executed 10 times (i = 0, i = 1, i = 2, i = 3, … i = 9) each with a different value of i. So the first time it displays a(0), then a(1), and so on.
Run this program yourself and see what happens. The items from 5 to 9 are empty and an empty message box is displayed.

Getting the length of an array
The length of the array is the number of items it has (both empty and non empty). If you need to know the length of the array you can use the keyword ArrayLen. It takes one parameter – the name of the array, followed by empty parentheses (thus indicating it's an array).

Note, that this keyword returns <u>the number of items</u> and not the index of the last one. The index of the last is ArrayLen(a()) – 1.

You may wonder why you should ever wish to find this out. Since you declare every array you are supposed to know the size. This is true theoretically, but in real lift you might change the definition so many times, you will get tired (and make mistakes) if you try to find out all places in your code where you wrote 10 and then 12 and then 300 and change them manually. Apart from this, you can resize your array somewhere along the way: read on to Resizing.

Resizing

It is sometimes hard to predict how many items you are going to need. If you need to change the number of items an array has, you can use the Dim keyword again anywhere in your application with a different number of items:

```
Dim a(200)
```

Anyhow, when you redim your array (Dim it again), you lose everything kept inside. So the array a now holds 200 items, all empty.

Explicit types – loop faster, get performance

Arrays can represent a very large amount of data. One of the advantages of an array is the ability to loop through the values easily, as shown in the code sample above (and remember to always write ArrayLen (xxx()) – 1 ! …). But, when dealing with large amount of data, the issue indicated at the "performance" paragraph in the variables chapter becomes relevant: there is a significant performance penalty caused by the fact that Basic4ppc does not know what to expect and tries to convert, carefully, the items in the array to different type. Being negligible when dealing with simple variables each at a time, it becomes quite evident with arrays. To overcome this issue, arrays can be declared as typed arrays – that is, you declare them with an explicit commitment to populate them only with data of a type you specify in advance: numeric, strings or Boolean. And since the computer knows some Sub-types, you need to even specify the exact type of data you are going to use. Basic4ppc takes your word and stops checking. This accelerates things a lot, but from this moment on this is your responsibility to populate only data of the right type in your array. If you don't, the program will stop and an error will occur.

The second advantage of this ability, is that it gives you a good way to communicate with external libraries, that expect to get, or that return, typed arrays. For performance reasons Basic4ppc does not convert a whole array as it does with simple variables when interacting with external libraries, so you should declare a type in this case.

In order to declare a typed array you use the keyword "As". This tells Basic4ppc what kind of data are in there. For example:

Dim a (10) As String

Will force every item in the array to be of type string (general text). The types you can use are (table appears in the help file):

Name	Description	Range
Byte	8-bit unsigned integer	0 – 255
Int16	16-bit signed integer	-32768 – 32767
Int32	32-bit signed integer	-2,147,483,648 – 2,147,483,647
Int64	64-bit signed integer	-9e18 – 9e18
Single	32-bit floating point	-3.4e38 – 3.4e38

Double	64-bit floating point	-1.79e308 – 1.79e308
Boolean	8-bit Boolean value	True, False
Decimal	128-bit floating point	-79e27 – 79e27
Char	A Unicode character	U0000 – UFFFF
String	A text string	0 – 2,147,483,648 chars. This is more a theoretical length as your device's memory is not likely to allow it

Error messages when type not compliant with data

Say we declared the array in the last code sample as

Dim a (10) As Byte

and then try to assign the value "AAA" (which is a string literal) to the first item. What would have happened? The answer is that an error will occur, and an error message stating "Input string was not in the correct format" will appear. As aforementioned, this is the general failure Basic4ppc encounters when it does not know how to convert something to a requested data type.

Multi-dimensional arrays

Basic4ppc supports arrays of up to 3 dimensions. To get an idea what a dimension means, say we have an array declared as

Dim arr1 (10) As Byte

so that it looks like this:

Index 0
…

…
…
Index 9

But sometimes we wish to create more than one column next to the first one. If we declared the array in this way:

Dim arr1 (10, 3) As Byte

Then we would have got this:

Arr (0,0)	Arr (0,1)	Arr (0,2)
…	…	…
…	…	…
…	…	…
Arr (9, 0)	Arr (9,1)	Arr (9,2)

This is called a two-dimensional array. Every item is referred to using a dual-index, as described above. The ArrayLen function can be used to retrieve the length of any dimension, with the optional Dimension parameter, where ArrayLen (a(), 1) is the length of the first dimension (10), ArrayLen (a(), 2) is the length of the second dimension (3), and the default, if the parameter is omitted, is 1.

A three dimensional array is just the same. It can be described as a cube, but it is more important to understand the concept. It is declared with

```
Dim a (10, 3, 5)
```

and every dimension has the length specified. Items are referred to as

```
a(0, 0, 0)… a(9, 2, 7)
```

(first and last, respectively).
ArrayLen (a(), 3) will return 8 in this case.

Assigning values from external objects
When getting an array from an external library, or from an object that returns an array (such is the array of bytes returned as a stream of raw data by the **serial** library), you actually need only to create a “dummy” array (since the “real” array is created by an external object and passed back to you). The only thing that matters here is that Basic4ppc knows you have an array with this name. The size and data are then given by an assignment of the type

```
ArryaName() = ExternalObject.ReturnedArray
```

and you need not take care of anything else. What you should do in such a case is declare an array of size 0:

```
Dim Buffer(0) As Byte
```

and assign the value you got from an external object as follows (this example assumes the existence of an object of type Serial):

```
Buffer() = SerialObject.InputArray
```

Initializing more than one item – the Array keyword

Basic4ppc gives you a special keyword with which you can initialize the values of more than one item in the array at the same line – the **Array** keyword. The example at the beginning of the chapter could be rewritten this way:

```
Sub Globals
   'Declare the global variables here.
  Dim a (0)
End Sub

Sub App_Start
   a() = Array ("AAA", "BBB", 3, 4, 0)
   a(4) = a(2) + a(3)
   For i = 0 To ArrayLen(a()) – 1
              Msgbox (a(i))
   Next
End Sub
```

Note, you could not write "a() = Array ("AAA", "BBB", 3, 4, a(2) + a(3))", because until after the line is executed the array still has the initial length of 0 and the expression a(2) will result in an error "Index was outside of the bounds".

Passing to/from Subs

Arrays in Basic4ppc cannot be passed as parameters to Subs, or be returned from a Sub as a returned value. This forces you to declare them as global variables and set their values directly, and not using parameters as is recommended for simple variables.

Copying arrays

As described before, when you get an array from an external object you can skip the size declaration as it's resized by the owner. When you need to copy an array or a part of an array to another array, the best way to do it is using the ArrayCopy keyword. This function copies part of the source array to the target array.

ArrayCopy (Source Array, Source Start, Count, Target Array, Target Start)

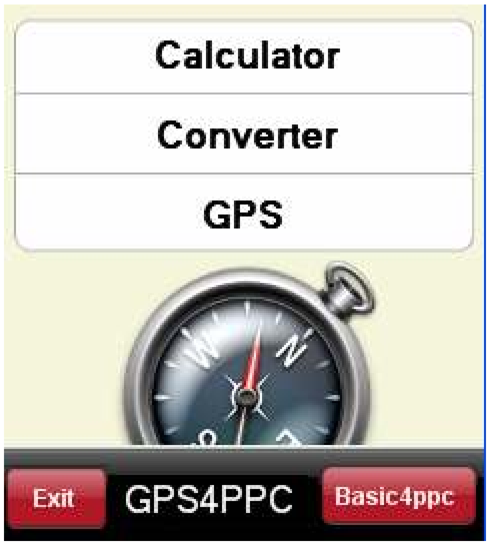
The parameters are:

- Source Array – The items will be copied from this array.
- Source Start – The first item that will be copied.
- Count – Number of items to copy.
- Target Array – The items will be copied to this array.
- Target Start – The position in the target of the first copied item.

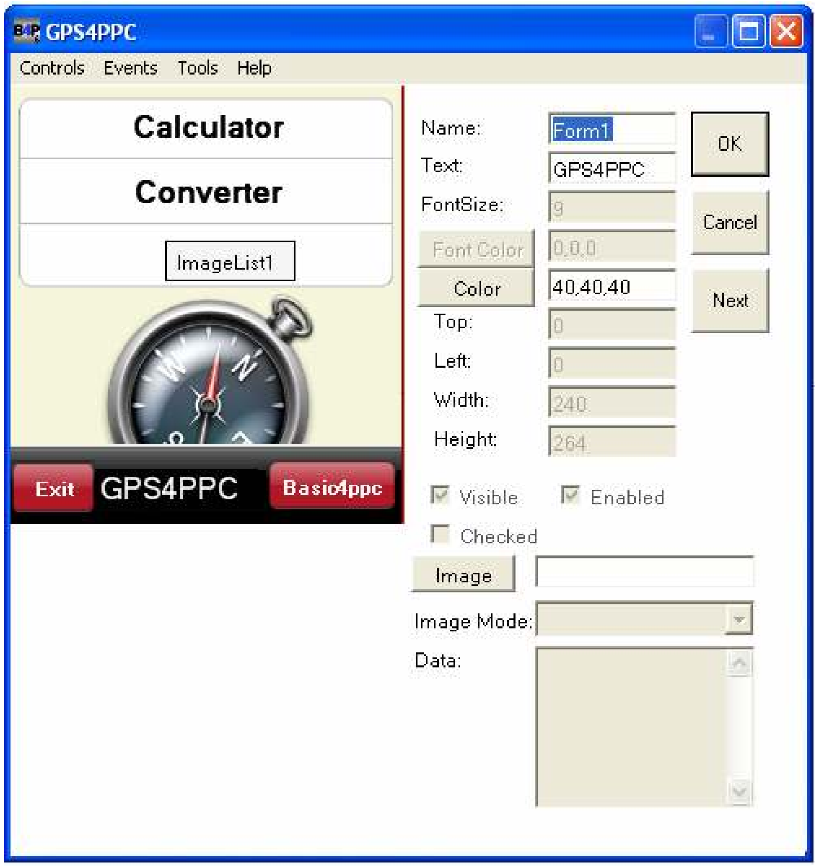
Graphical User Interface and Visual Designer Basics

This chapter is the first one in which we will cover the basic ways to create GUI in Basic4ppc. Though it is basic, the techniques shown are the same used for richer interfaces. Knowing how to display output and get input in a graphical environment will let us go on with the samples in this guide.

Some history first: when Windows Mobile was first released, it was one of the only operating systems for mobile devices, and it tried to resemble the user interface Windows (the big one, I mean) supplies. This was going well for some years but is undergoing a fundamental change during the last years, since more modern devices had appeared. The first UI introduced by Windows mobile, and the easiest to implement, is the one shown in this chapter. The bad news is it is not as neat as more modern UI suites out there. So this chapter draws the basics of designing GUI **with a designer**. A later section, holding some chapters, describes the flexibility of combining both the designer and code for GUI creation. The picture below illustrates the places towards which we are heading – this is the main screen of Basic4ppc's GPS4PPC – a sample program and open source code available on the Basic4ppc website that show you how to create both modern GUI and GPS software. The last chapter in the Advanced Graphics and UI section shows a step by step of how to create this interface.



And this is how it looks on the designer:



Name: Form1
Text: GPS4PPC
FontSize: 9
Font Color: 0,0,0
Color: 40,40,40
Top: 0
Left: 0
Width: 240
Height: 264
[✓] Visible
[✓] Enabled
[ ] Checked
Image Mode:
Data:

What is GUI

A Graphical User Interface is everything you see on the screen of a Windows modern computer. Unlike old interfaces where graphical ability was very limited (for example, when you changed the font size on your word processor there was no indication it has changed), you nowadays see a visual representation of many things. The amount of representation depends on the person who wrote your program. When you open a window, for example, it can either just appear or can, say, gradually grow from the corner of the screen. The amount of work required to create every effect differs, hence the general preference of programmers to choose the simpler solution. In order to make things simpler, there are standard tools and many extensions to help you build your interface. The first thing you would like to know is that on a Windows system, every program shows its GUI in a window.

What is a window

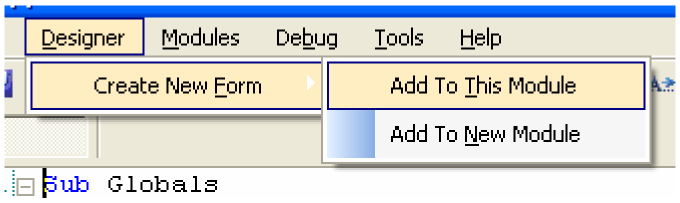
A window is the place where your program displays its GUI, but is also the place where everything happens: message boxes appear, graphs are displayed, input is written. A program can have more than one window, or have none at all. A window is also **a control,** a word used to describe all GUI elements that can be combined to a windows GUI system (and many times not only GUI). Basically, a window is a kind of a "top-level" control, that gives place to all other UI elements (controls) to live in.

In windows programming systems, the basic "window" is called "Form". From now on, we will refer to every window, especially during the design stage, as a form.

The most first window that is opened when your program starts is the program's main form. When you start the program (if it has a form) it waits till this form is closed: only then the program ends.

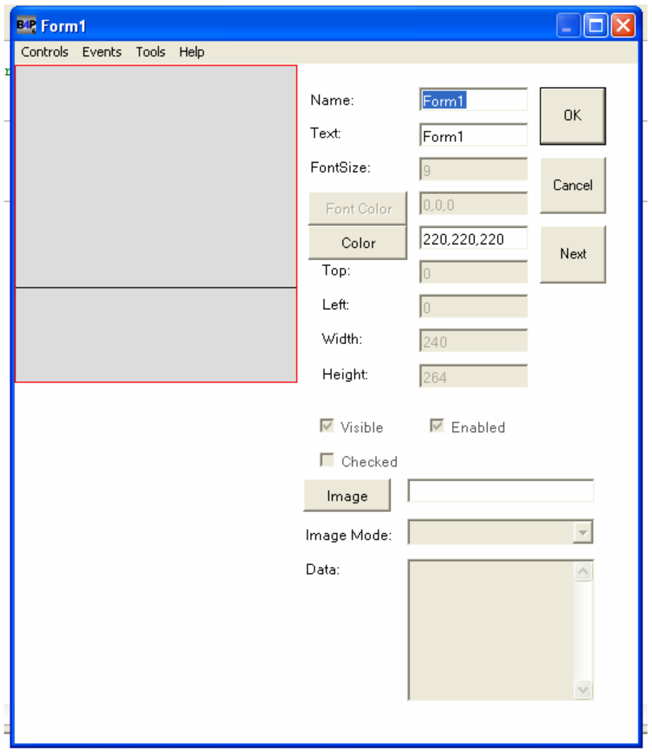
Adding a form to your program

The first thing you need to do in order to create GUI is to add a form to your program. Open the Designer menu, choose "Create new form", and then "Add to this module":



```
Sub Globals
```

The designer window is opened and an empty representation of the new form appears:

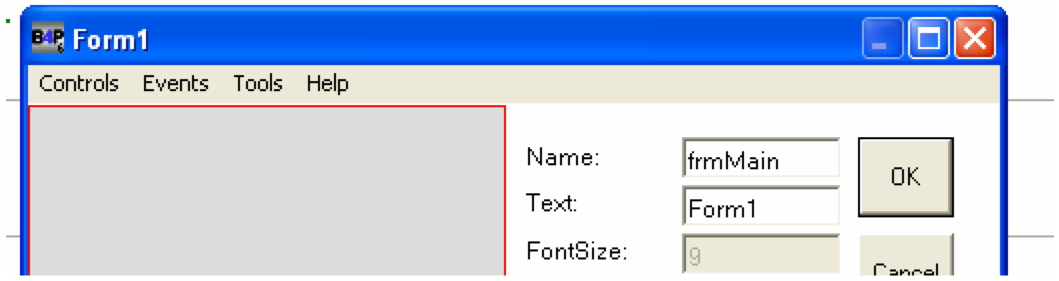


Name:	Form1
Text:	Form1
FontSize:	9
Font Color	0,0,0
Color	220,220,220
Top:	0
Left:	0
Width:	240
Height:	264

Visible ✓ Enabled ✓
Checked
Image Mode:
Data:

The first thing there is to know about controls, and forms amongst them, is that they have names, and you can refer to them using their names. Then, they also have properties, which are values you set to achieve certain purposes.

The form is created by default with the name Form1. This appears at the top-right corner of the designer window, and this is the first thing we change to something a bit more meaningful. It does not matter much now, but it’s a good habit.



Name: frmMain
Text: Form1
FontSize: 9

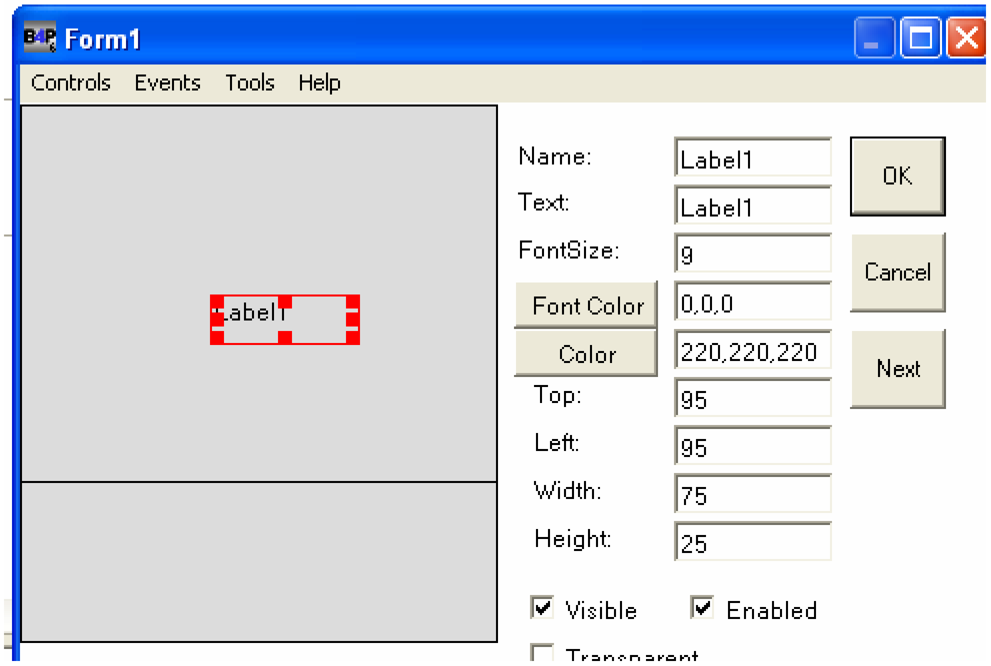
Change the name to frmMain (frm for Form).

Adding controls

The second thing to do when you create the GUI is to add the elements you need. Look at your computer’s screen. Almost every distinct piece of what you see is an element – a control on top of a form. There are dozens, and the designer lets you add only the most basic. The rest are added in code as external objects – see the Additional Libraries chapter for information about libraries and adding controls. Anyhow, the idea is the same.

Controls (GUI elements) are the building blocks of your GUI. Each such component has a defined role and carries out a specific task. For example, there is one that displays text. Just this. You place it on the form, tells it what you want to display and it shows the text on top of itself (it is called **a Label**). There is a different one, which lets you type some text in. This is called a **Textbox**. And there is another one, which draws a button on the screen and simulates pressing it down when your mouse clicks it.

We will start by creating a simple GUI that uses just these elements: the Label, the Textbox and the Button. Open the Controls menu, and Choose Label. The word Label1 appears on the "Form" in your designer. Click it so that it is surrounded with a red rectangle:



Name: Label1
Text: Label1
FontSize: 9
Font Color 0,0,0
Color 220,220,220
Top: 95
Left: 95
Width: 75
Height: 25
☑ Visible ☑ Enabled
☐ Transparent

Now, change the relevant properties of the control.

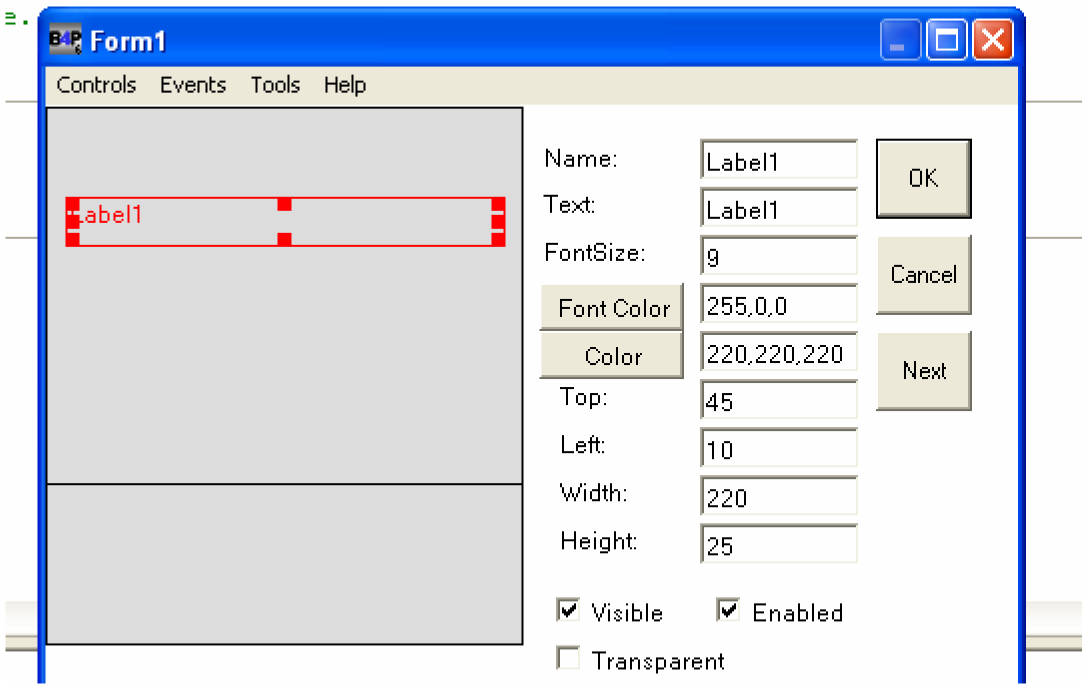
Properties

Properties are special kind of variables: they are variables that are owned by an object or control. You can access them in your code. Changing them affects the control's display and behavior. They can be set programmatically, as we will soon see, or on the designer. Some properties can only be read – that is, you can set their values to other variables or display it, but you cannot change it. Some are read/write (I/O, sometimes) – you can set their values as well.

Set the following properties of your label:

- Change the Name to lblOutput
- Change the Text to "---"(without the brackets).
- Change the font color to red (press the button)
- Change the top to 45, Left: 10, Width: 220, Height: 25.

Your form should look like this:



Name: Label1
Text: Label1
FontSize: 9
Font Color: 255,0,0
Color: 220,220,220
Top: 45
Left: 10
Width: 220
Height: 25
[✓] Visible [✓] Enabled
[ ] Transparent

The same way add a Button control and change its properties:

- Text: "OK"
- Top: 210, Left: 85, Width 75, Height 40 (note, these properties can be changed by dragging the control).

And a Textbox with these properties:

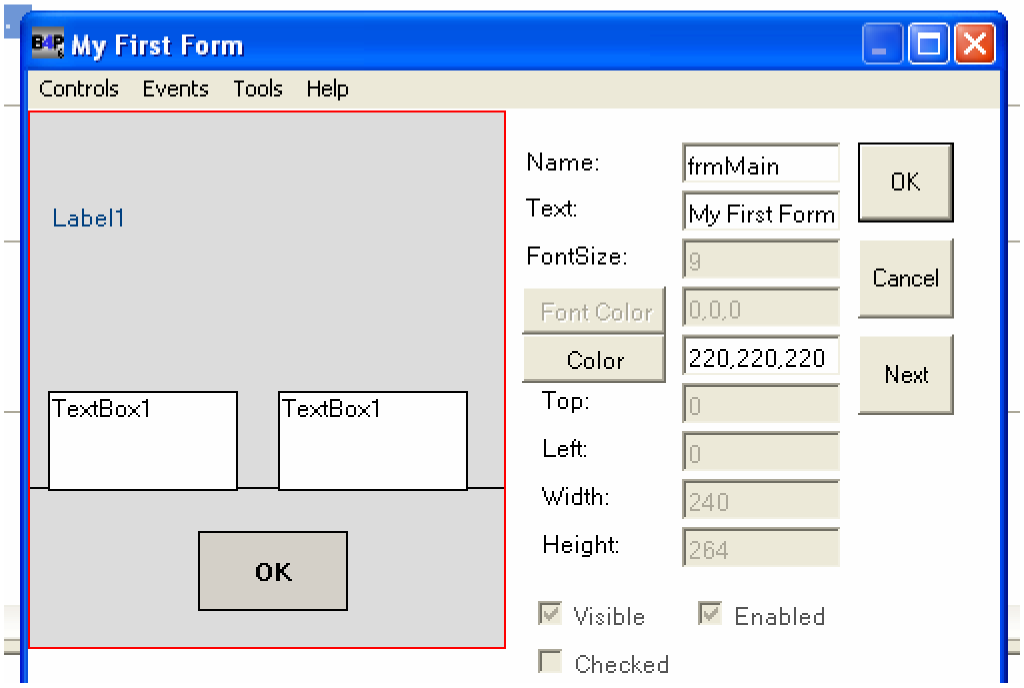
- Name:lblIn1
- Check "Multiline" (it appears when you click the Textbox)

- T/L/W/H: 140, 5, 95, 50

And another Textbox with these properties:

- Name:lblIn2
- Check "Multiline" (it appears when you click the Textbox)
- T/L/W/H: 140, 5, 125, 50

Now click the gray surface of the form and change the Form's Text property to "My First Form" – this is the text at the forms Title bar (the blue line above each window).



Name: frmMain
Text: My First Form
FontSize: 9
Font Color: 0,0,0
Color: 220,220,220
Top: 0
Left: 0
Width: 240
Height: 264
[✓ Visible] [✓ Enabled] Checked

Adding an Event

An event is a Sub that is executed by a control or an external object when something happens. The Button has a useful event called Click. Click the button first to select it, open the Events menu, and select Click. A Sub called Button1_Click is added to the program and you are being asked if you wish to save the changes. Choose yes.

As you see on the code, the new Sub has a name composed of the control's name, an underscore, and the event's name. When Sub comply with this naming convention it is automatically attached to the event with the relevant name for the relevant control (just for the general interest, you can also add events with different names using the AddEvent keyword. This will be discussed later this book).

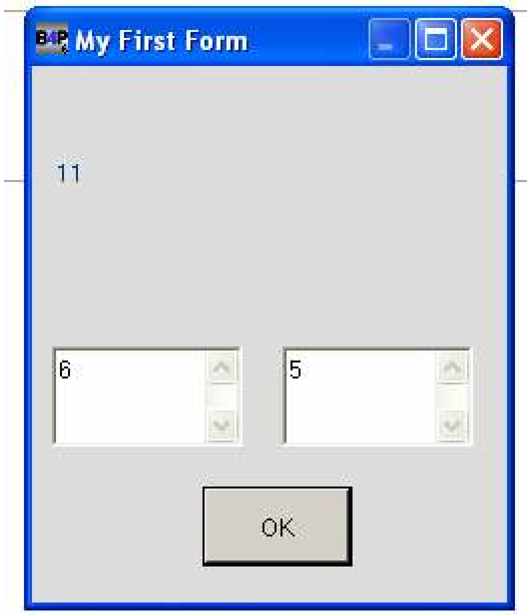
The event should not just sit there – you should add a code to it so that it does something when the button is clicked. The way to do it is to just write the code you need to it as if it is a regular Sub (it actually is). Suppose we need this event to show the result of adding a number you enter into the first Textbox to one you entered into the second. The Textboxes are called txtIn1 and txtIn2 (txt indicate it's a Textbox. "in" stands for input). They both are controls and their properties, then, are accessible in code.
So if you add this code to the Button1_Click Sub, the numbers in both textboxes are added and the sum is displayed on the surface of the label:

```
Sub Button1_Click
        Label1.Text = txtIn1.Text + txtIn2.Text
End Sub
```

The line of code in the Sub above does just what you'd expect: it puts into the label the result of adding the text inside the first textbox to the text inside the second textbox.

Trying out your code

The code you just wrote is ready: hit F5 and run your program. Type 5 in one textbox, type 6 in the second one, and hit OK:



If your device is connected, this is the right time to test your code. It should not be tested each time you change something, just upon major corrections. But it is a good practice. Read in chapter 2, the best place for your source code: follow the instructions there and compile your program (autoscaled if your device has a 480 X 640 screen). You can run it on the device and see how it looks. Don't worry if you feel this UI is dull. We will be creating much more compelling user interface in a following chapter. This chapter is intended to only show you the basics.

Designer behavior summary

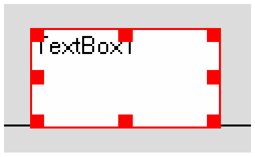
This is a short list of the different things you can do with the designer. Read it briefly, and you can always go back to it when you need a specific task.

- Adding a control

  Add a control by selecting its name from the pull down "Controls" menu to the upper left corner of the designer window. A list of controls appears. Choose amongst them and it is added to the form. Most controls appear visually on the form, but there are some, such as the timer and the data structures, which are invisible.

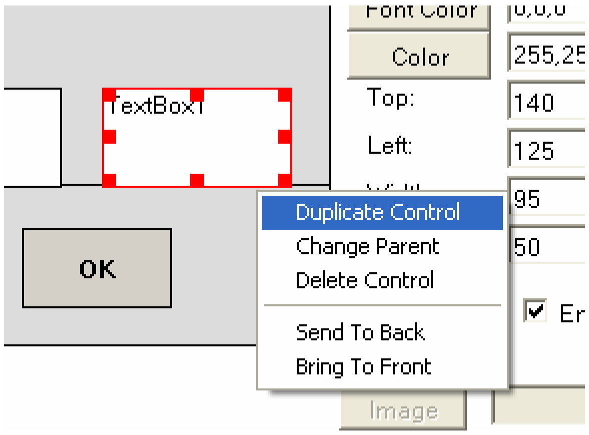
- Selecting

  Select controls by clicking inside it. When selected, the control is marked with red:



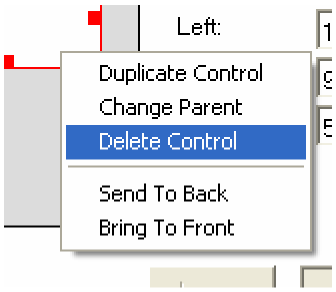
- Duplicating

  Duplicate by right-clicking and selecting "Duplicate control" from the drop down context menu.



- Deleting

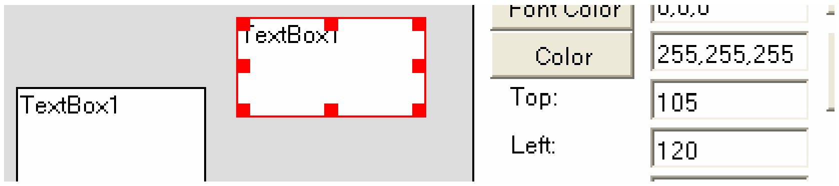
  Delete a control by choosing the Delete Control option in the same menu (above).



- Moving

  Move a control along the form by dragging it with your mouse. You can also change the

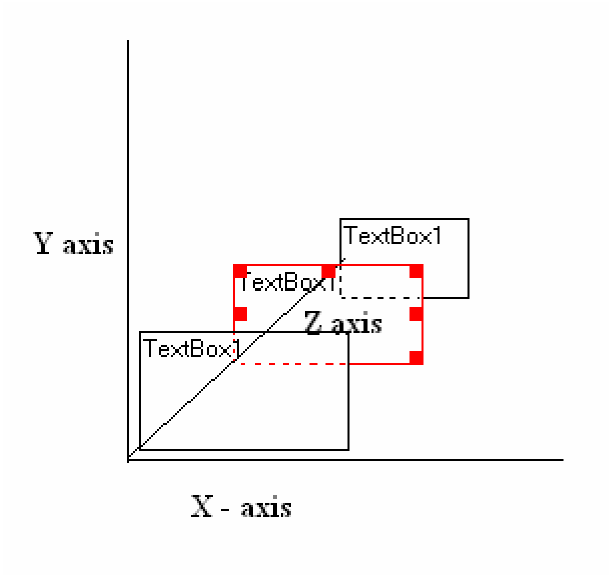
Top and Left property of the control to a different location. Note, that when you drag it with the mouse these properties actually change. You can, of course, set them in code just as we did with the **Text** property in the small code sample above, to the same effect.



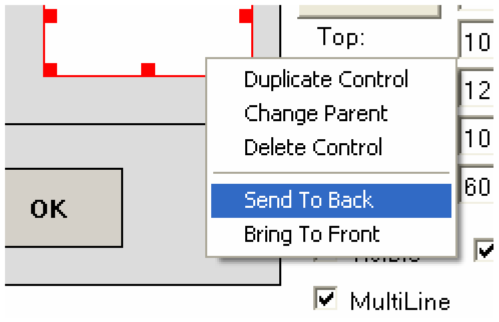
- Resizing

  By dragging with your mouse over one of the bold red rectangles on the border of the control when selected, shown in red in the figure above, you can adjust its size graphically. This is similar to changing manually the Width and Height properties on the designer, or by setting them in code just as we did with the **Text** property in the small code sample above, to the same effect.
- Placing controls one on top of the other: Z-Order

  The term Z-order is commonly used on texts dealing with the orders of controls on a Microsoft form. It describes the “order of controls when placed on top of one another”: or, as the guys in Microsoft like to think about them, the orders of controls along the Z axis. Consider the left-right dimension of your screen to be the X axis, and the Up-Down dimension to be the Y axis, so the “Into-the-Screen”-“Towards you” dimension can be thought of as the Z axis:



In order to change the location of a control in the "Depth" dimension (in other words, to set which control is on top of which), use the "Send to Back" and the "Bring to Front" menu options in the context menu on the designer (or otherwise, use the Methods "Textbox1.SendToBack" and "Textbox1.BringToFront" in code, assuming your control is called Textbox1):



Try it on: place some controls on your form, and change their order.

- Changing properties

  The designer window contains some other properties you might be interested in changing. These properties are the control's Name, Text, Font size, Font color and Back color and other properties that change from one control to another. Note, that you can set in code each of these properties (this is actually what Basic4ppc does for you, in a code section that is hidden from the editor). Some common properties you may find here are:

  - Visibility: when set to false, the control is invisible when the program runs. Note – it IS visible on the designer!
  - Enabled: even when visible, you sometimes prefer the user will not be able to interact with a specific control. Set enables to false and get a grayed out, non-functioning control that is only displayed there.
  - Image: some controls (such is the Image control, to name one…) allow you to set image to them – you can pick it up it here.
  - Image mode: includes three options to display an image on a control:
    cCenterImage: the center of the image and the center of the control are at the same place.
    cNormalImage: the upper-left corner of the image and the upper left corner of the control are at the same place.
    cStrechImage: image is stretched or shrunk so that is exactly fits the control's size.
  - Other: some other properties appear here when you select different controls. You can easily find out what they do from the help or by trying it out.
- Changing parent
  Some controls are "Parent controls". They group other controls and thus allow you to treat a bunch of components as if they were one. Imagine you have three buttons – OK, Cancel, Exit – that you need to show and hide always together. Placing them on a Panel control and then setting the Panel's Visible property to false, in code, is the most elegant, and less error prone, way to do so.

To change a control's parent, from the Tools menu choose Change Parent, and then confirm the message and click once the new parent desired. If you chose a control that can not be a parent control, the action is ignored.

- Adding an Event

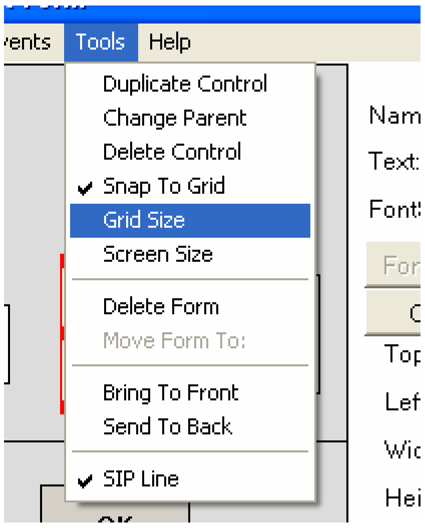
  Adding an event is done as previously explained by selecting it from the Events menu. Each control has different events exposed through the designer. Some other events exist, but they are documented only under Microsoft's .NET CF 2.0 online documentation: they can be referred to using the Door library, as will be explain on the Door library's chapter.

- Grid

  The grid are imaginary lines – horizontal and vertical – spread across the screen. They cannot be seen (hence imaginary) but help you to align your controls in straight lines.

  - Size

    Set distance between the grid lines using the grid size option on the Tools menu.

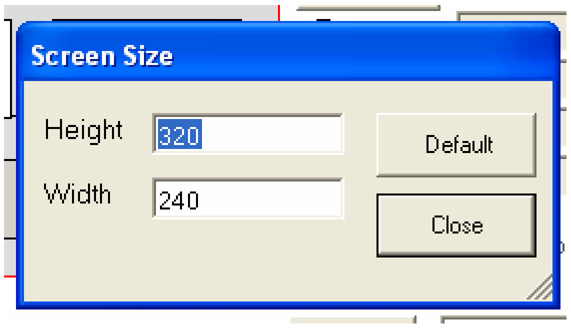


- Snap

  You can force the designer attach your controls to the grid, and it is very useful most of the times. But this can be somewhat annoying when you try to set something very Sublime and find your controls moving in steps bigger than you want them to. Uncheck the Snap to Grid option to stop them from doing so.

- Screen size

  A very important feature in the designer is the ability to change the size of the screen you are working on. When developing for mobile devices, you are often required to work with different screen sizes. Use this option to set the dimensions of the screen you are developing for.



Height 320

Width 240

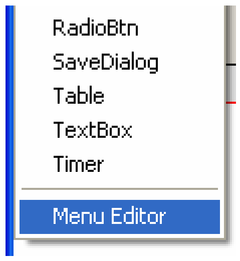
- Deleting a form

  From the Tools menu, select "Delete Form".

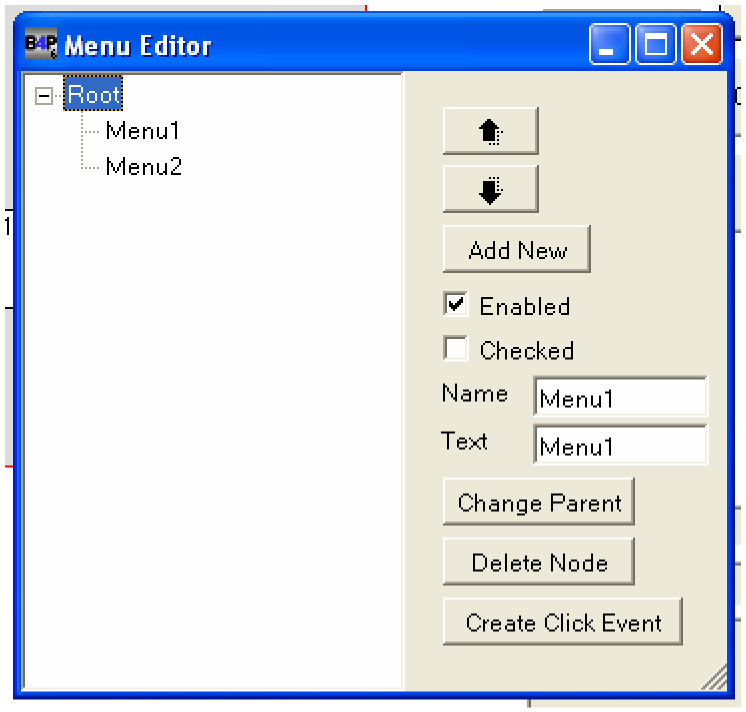
- Editing a menu

  Some applications make use of the Windows mobile ability to create menus. Such is the Basic4ppc Device IDE's main menu – it resembles the main menu on the desktop IDE. Adding a main menu to your application is a very important ability.

  Open the Basic4ppc menu editor



The menu editor appears:

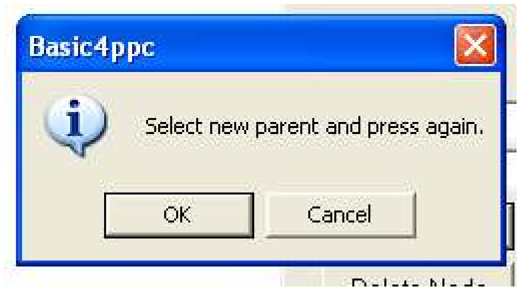


Name: Menu1
Text: Menu1

Most of the editor is very intuitive and will be discussed in a nutshell. The main features are described below, and some extensions follow:

- A menu is described as a tree: there is a root, and every Sub menu is a "node". Each node is called a menu item.
- The Add New button adds a new Sub-menu to the same level you are on.
- You change the name of a menu item using the Name textbox. This can be different than the Text displayed on it, which is set using the Text textbox. So you can have a menu called mnuFileOpen with text "Open".
- The Enabled checkbox describes itself (try it).

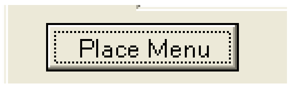
- The Checked checkbox allows you to add a small "v" sign next to the menu item.
- Now the "Change Parent" Button is trickier. Select one of the nodes: then, click the Change Parent button, and get this message:

  Basic4ppc

  Select new parent and press again.

  Now, when you click OK, it seems as though nothing happened! But no, there is a change: the "Change Parent" button has changed its caption:

  … and it now says "Place Menu"! So what you would do now, is select the new menu under which you want to place the menu you moved, and hit "Place Menu" – and there you are.
- The two small up and down arrows at the top do what you would expect, except that they do not change the parent of a Sub menu. For this purpose, you will use the previously described procedure.
- Delete a menu item with the Delete Node button.
- And at last, the Create Event button. A menu item is like every other control when it comes to events. So if you want to add a procedure that is to run when the item is clicked by the user, hit the Create Event button and add you code to the newly created event.

  A note about menus:

  Over the last years, mobile devices menus have changed appearance Substantially. A finger-base UI is taking growing place in the market, and menus intended to be used with stylus are less popular – especially in small, or professional applications. Though today considered "too complex for the typical user", yet many times, when writing a more complex or professionally-specific applications, the use of menus in inevitable.

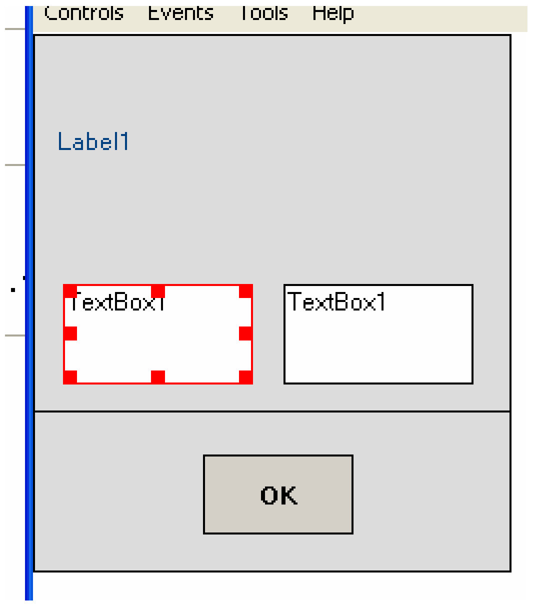
Creating a compelling user interface

On a later chapter of this guide I show how to create a user interface that complies with the modern standards of user interfaces – I will guide you through Basic4ppc'

And yet another note. Many users have written programs using menus that are intended to be used with stylus. There are some noticeable ones, amongst which are the "Scaling maps" discussed in a tutorial by Klaus, and the commercial **CEASER field** – both can be found on the forum.

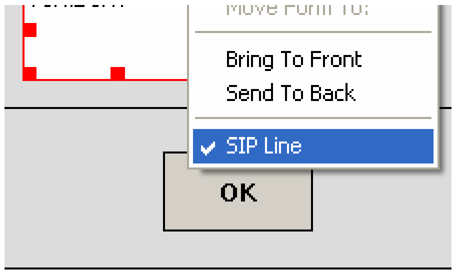
- Sip line

The SIP line (Soft Input Panel – the strange name Microsoft has given its on screen keyboard) is the line that appears to the bottom fifth of the form on the designer:



- On the picture above, the OK button is located just below the SIP line. The reason for displaying this line is simple: this lower part of the screen below the line will be hidden every time the SIP (the keyboard) appears. This applies to the keyboard supplied by Microsoft: there are other keyboards on the market, but you cannot plan for anything there is. Many are compliant with the size of the original one.

  You can hide the SIP line using the Tools-SIP Line menu option:

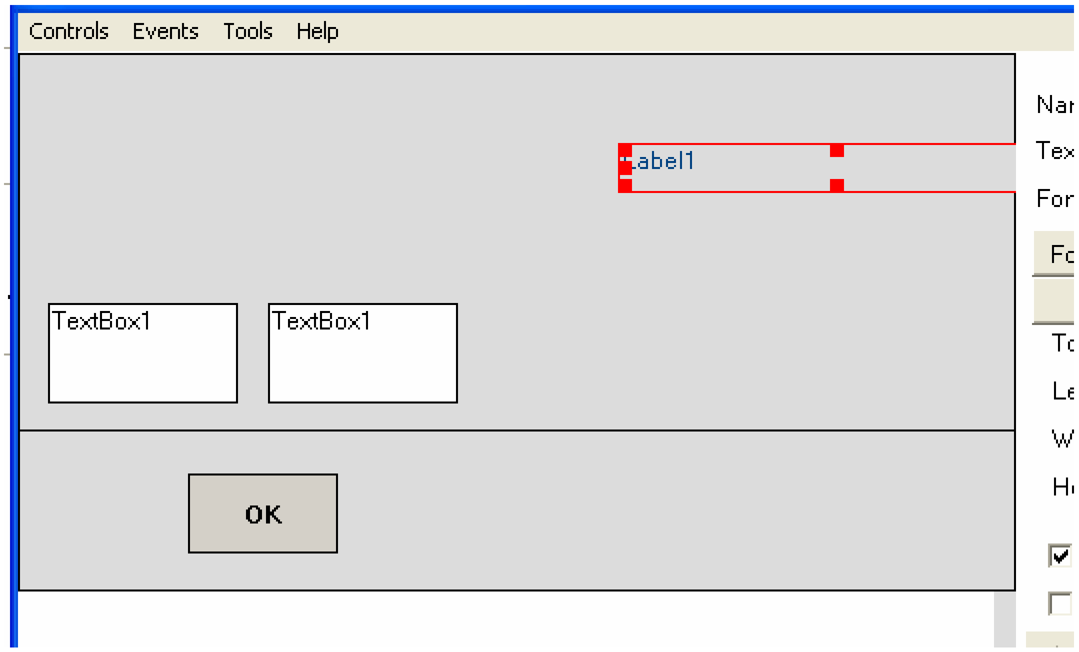


- Finding a lost control

  Sometimes a control you placed on the form gets lost. To resemble the situation, let's do the following small exercise: add a label to the form, name it lbl (short for Label. Don't remember how? See the beginning of this chapter). Now, change the Left property of the label you added to 300 and press Enter. Whoops! The label has disappeared! Where do we find it?

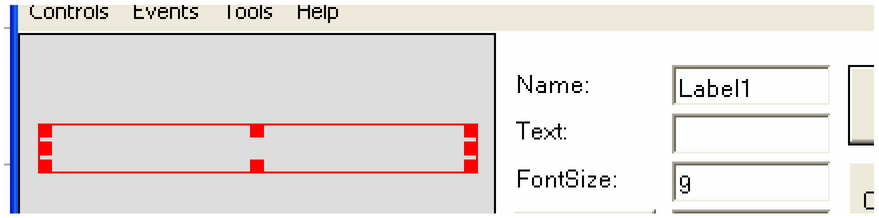
  - Enlarge the form

    Well, obviously it has jumped to coordinate 300: go to the Tools menu, select Screen Size, and insert 500 as the screen's width: there you are:



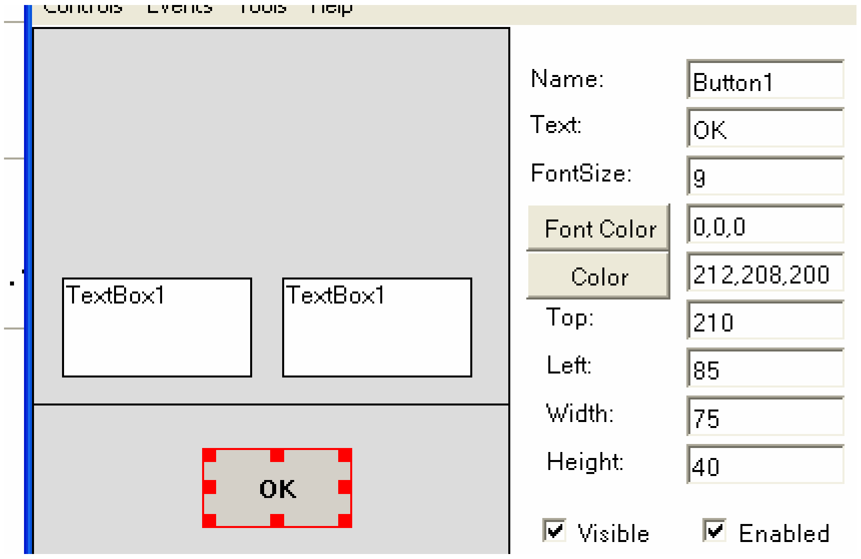
- Select multiple

  No consider another scenario: you got the label ok, but you can't see it. It is actually very common: suppose you wanted the label to have no text at first, and you planned on adding the text to it at runtime. Try it: move the label back to coordinate 10 (the Left property) and clear the Text property so that it is empty:



Name: Label1
Text:
FontSize: 9

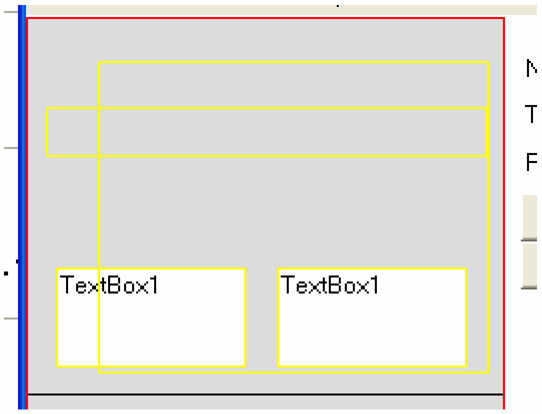
  The label is evident at the picture above, but what if you clicked one of the other controls on the form? Click the OK button to find out:



Name: Button1
Text: OK
FontSize: 9
Font Color 0,0,0
Color 212,208,200
Top: 210
Left: 85
Width: 75
Height: 40
[✓ Visible] [✓ Enabled]

Well, there we are: the OK button is selected, but where is the label?

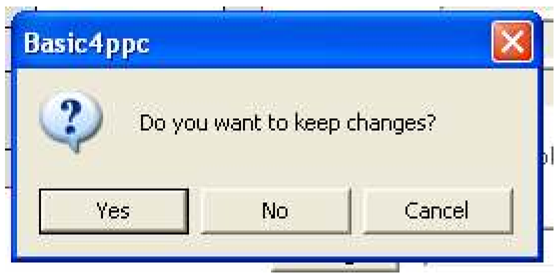
The way to find it is to click and drag somewhere on the form towards the place where the label is:



Notice the yellow lines that allow you to multi-select the controls on the form. The label is now visible and you can re-select it alone and set its properties.

- Save changes?

  When you are done with designing, close the designer window. Basic4ppc is unsure whether to save your work or not, so this message appears:



Do you want to keep changes?

Hit yes. Actually, hitting no is good only when you have messed up something so severely that the only way to get out of the troubles is to cancel it all and start again – often a very good strategy.

<u>What is a method (a bit off topic)</u>

We have spoken quite a lot about the properties of a control. A property can be thought of as a variable that is attached to a control (or an object, as we will later see), from which values can be taken and sometimes, to which values can be set. As I mentioned when discussing the BringToFront method, there is a second kind of "things" attached to a control – the Subs it allows you to call, called Methods. A method is just like a regular Sub you write in code, but it is a predefined one. You can call it as if it's yours, and it usually does something related to its control, as you can see in the example below:

```
Sub Bring_Label_To_Front
    Label1.BringToFront
End Sub
```

- Help on controls

  There is much to know about every control. You will get to know the important ones during this guide, but there will yet be many others. As everywhere else in Basic4ppc, the best way to learn is to read the help file (see "How is a help file built" in chapter 2) and ask on the forum. For the most important ones there is a chapter on this guide and future tutorials may appear.

Flow Control Structures

This chapter covers some basic statements that control the flow of a program. It is the direct continuation of the "Program Flow" chapter. It is delivered here after the chapter about GUI, as this allows us to demonstrate the things needed using some simple GUI tools.
On this chapter you will find the If statement, the For, While, and Until loops, the Select statement and you will learn about calling Subs as a way to control the flow of the program.

What is "Program flow control"
As explained on the Program Flow chapter, one of the most basic tasks to carry out when writing a program is the control over which is the next statement to execute ("Active statement"). Controlling this lets you direct your program to do what you want it to do.
For example, suppose you ask the user to enter his age, and act differently when he is more that 25 than when he is less than 25. You direct your program to a different part in each case: the If statement does just this.

The **If** statement

Not a beginner? If you have a former programming experience and you only need a brief summary of the Basic4ppc If statement, go straight to the end of the section. There you can find a table with Boolean operators and a summary of the differences between the common If statement in other programming languages and that of Basic4ppc.

The If statement is a way to check if a condition "exists" (it's sometimes called "evaluates to true"), and do something in case it is and a different thing in case it is not.

Let’s consider a simple test case: you will write a program that puts a textbox on a form (see the previous chapter for this), the user enters his birth date into it, and presses OK. If it is his birthday, than a message saying “Happy Birthday” appears.

The way to do this is to use the IF statement. Create a new form on a new Basic4ppc program (use the File-New menu to create a new program). Don’t bother renaming it: Form1 is a good name for all reasonable purposes. Add the following controls to it:

Textbox	
Property	Value
Name	txtMonth
Top	60
Left	120
Text	
Width	30

Textbox	
Property	Value
Name	txtDay
Top	60
Left	155
Text	
Width	30

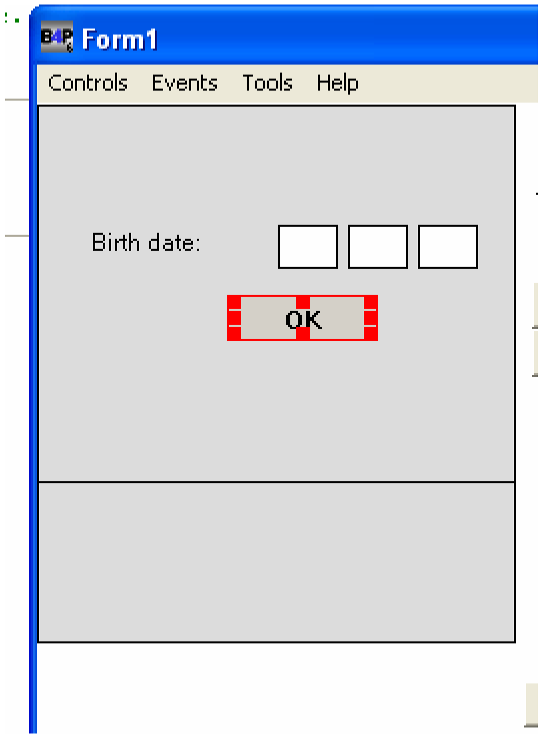
Textbox	
Property	Value
Name	txtYear

Top	60
Left	190
Text	
Width	30

Label	
Property	Value
Name	Label1
Top	60
Left	25
Text	Birth date
Width	30

Button	
Property	Value
Name	cmdOK
Top	95
Left	95
Text	OK
Width	75

Your designer's form should now be looking like this:



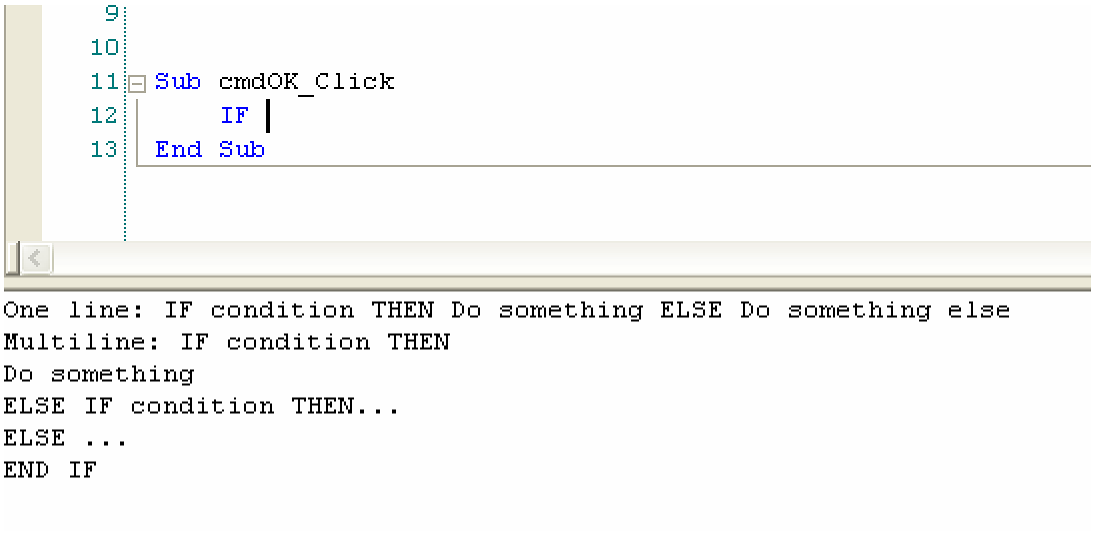
Now, add the Click event to the OK button. Confirm the message box offering you to keep the changes. A new Sub appears – the event. This is where the If code will be shown.

Add the following code to the new Sub:

```
Sub cmdOK_Click
        If txtDay.Text = DateDay(Now) Then
                Msgbox ("Happy Birthday")
        End If
End Sub
```

Since we are now just getting a bit deeper into more "common" way of writing code, I am taking the place to make a point about typing – note two of the typing assistants you should be familiar with when typing:

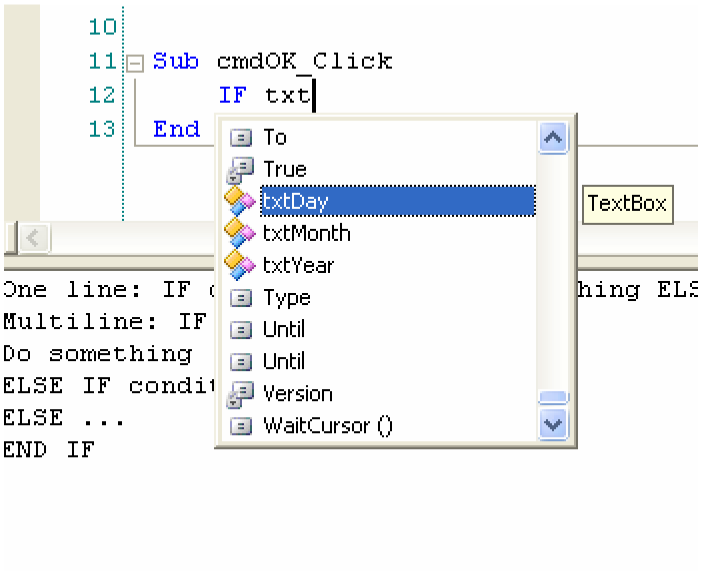
Syntax: as always, when you start typing the On-The-Fly typing assistant shows some code tips at the bottom. Get used to read these notes, as the offer important information about the syntax of things you are now typing.



```
Sub cmdOK_Click
    IF 
End Sub
```

```
One line: IF condition THEN Do something ELSE Do something else
Multiline: IF condition THEN
Do something
ELSE IF condition THEN...
ELSE ...
END IF
```

One other good thing to get used to is using the AutoComplete feature. When you add controls using the designer you gain a very important advantage, in the form of the ability to use AutoComplete to suggest the next word. When you start typing, type the first couple of letters of the control's name ("txt…" for example), and press Ctrl+Space:



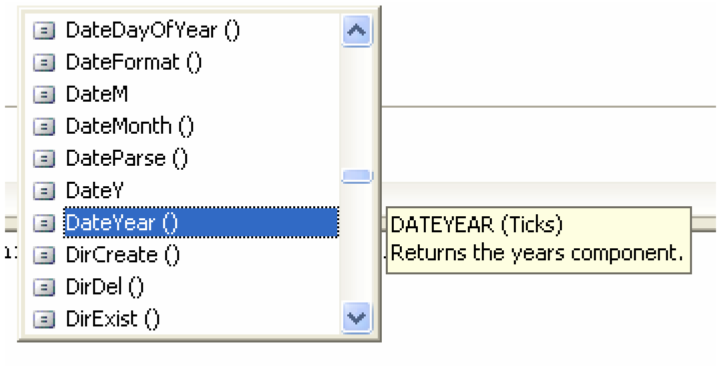
```
Sub cmdOK_Click
    IF txt
End
```

```
One line: IF c                                              hing ELS
Multiline: IF
Do something
ELSE IF condit
ELSE ...
END IF
```

You will learn later, that the AutoComplete feature works only with controls added through the designer: it knows nothing about those added in code. More about adding controls in code is in the future chapters about GUI.

Let's get back to the code. The code you added starts with the conditional statement "If". The If statement is composed of these parts:

- The keyword If.
- A condition. In our case, the condition is txtDay.Text = DateDay(Now). It means that the computer should compare the value of the Text property of the txtDay TextBox (this is where the user has entered his day of birth) with the day of the current date (the keyword "Now" is used to represent the current date. We get the day component out of it using the DateDay function (it's a keyword as well). More about them on the Date and Time chapter. You can get a short descriptions for keywords when you wait a moment selecting them from the AutoComplete list:



DATEYEAR (Ticks)
Returns the years component.

-

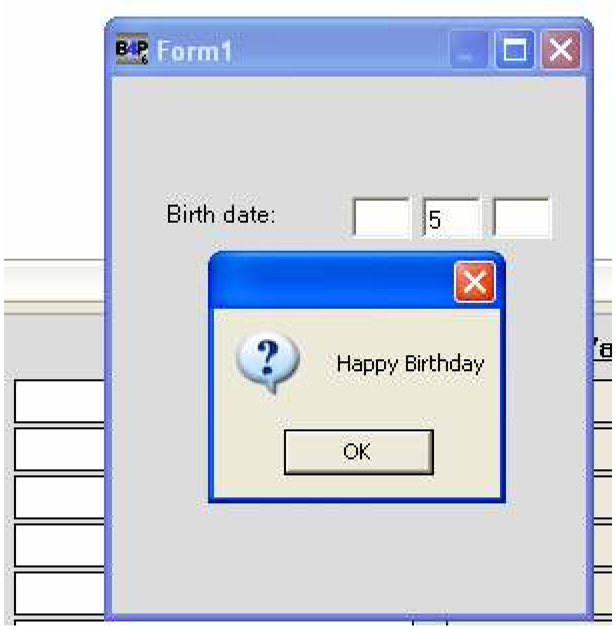
This is a useful feature that can help you recall what the meaning of each keyword is.

The process of comparing the two **expressions** (both txtDay.Text And DateDay(Now)) is called "**evaluating the condition**". At the end of the process the compiler (or the interpreter) decides if the condition evaluates to True or to False: these are the only options. Anything that is true evaluates to True (surprisingly), and vice versa. If it evaluates to True, and only if, whatever you wrote between the If part and the End If statement happens.

- The keyword **Then** must come right after the condition. It tells Basic4ppc that this is the end of the condition and from now on orders follow.
- Commands – normal Basic4ppc code.
- An Optional (omitted in the sample above) **Else** statement, after which a Basic4ppc code appears. This code will be executed if the condition in the If statement evaluates to False.
- The statement **End If**, that tells Basic4ppc this is the end of the If statement.

Note, that in case the If statement evaluates to False (say, the user's day of birth is not the same day as today is), and there is no **Else** statement, nothing of all of this happens – the active statement jumps to the line after the **End If.** This is line 15, and the Sub ends.

Run your program pressing the F5 key. Save it under the name "If sample" – we will use it on a later chapter. Enter the day you are in on the month to the middle textbox and hit the OK button:

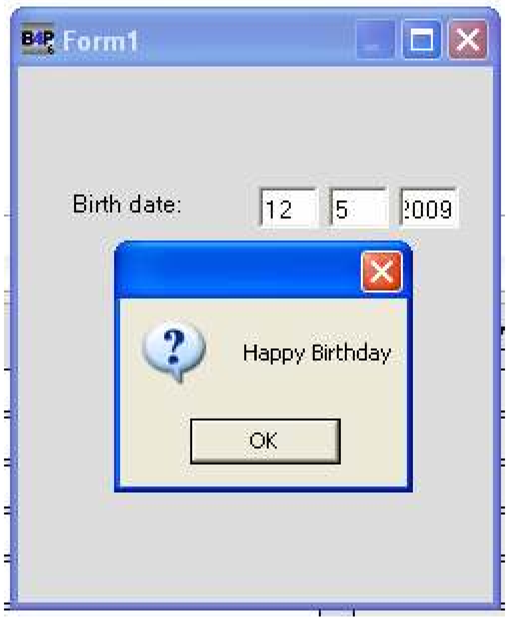


Happy Birthday

Hopefully, everything is now working and you got a happy birthday greeting. But we would like the user to enter the entire birth date, so let's change our code to something like this (you will have to stop the program that's running by either closing it's main form or hitting the Stop button on the main toolbar: ■

```
Sub cmdOK_Click
        If txtDay.Text = DateDay(Now) Then
                If txtMonth.Text = DateMonth(Now) Then
                        If txtYear.Text = DateYear(Now) Then
                                Msgbox ("Happy Birthday")
                        End If
                End If
        End If
End Sub
```

There is nothing new here, except we added some **nested if statements**. The inner If statements will not be executed (that is, they will not even be evaluated) if the previous ones are evaluated to False. So in general, the user must enter today's date in order to get some greeting.



Birth date: 12 5 2009

Happy Birthday

But there is a problem: it really is tedious to have to write so many If statements. It is also not very readable. In order to make things shorter, you can use a combination of the following **Boolean conditions words:** And, Or, and Not. So the same code could be looking like:

```
1   Sub cmdOK_Click
          If txtDay.Text = DateDay(Now) AND txtMonth.Text = DateMonth(Now) AND
2         txtYear.Text = DateYear(Now)
3                   Msgbox ("Happy Birthday")
4         End If
    End
5   Sub
```

Line 2 is squeezed because of the limits of the printed page, but it is one line. It has three conditions, joined by the And keyword. This means, the entire If statement evaluates to True only if each of the expressions following the And keyword does so itself.

Now let’s add an Else statement. If it is not the user’s birthday, let’s check if he is less than 15, and react to that. We saw how to check if two values are equal. How do we check the difference between them and which is bigger? Well, we use the operator > instead of =. This is the new code:

```
11  Sub cmdOK_Click
            If txtDay.Text = DateDay(Now) AND txtMonth.Text = DateMonth(Now) AND txtYear.Text =
12          DateYear(Now) Then
13                  Msgbox (“Happy Birthday”)
14          Else
15                  If txtYear.Text + 15 > DateYear(Now) Then
16                          Msgbox(“You are still young. You must be really expecting it!”)
17                  End If
18          End If
19  End Sub
```

Read carefully lines 14 – 18. Note, they are only executed if the main “if” is not. Only then we enter the “Else” part. Also note, the **Else** keyword is now the place that ends the first part and that the **End If** statement, previously ending the If clause, now ends the Else clause. The Else clause itself contains another If statement that is comparing the value of the year entered by the user added 15 years to the present year. If the user is under 15, he gets another message.

Writing If on a single line

It is many times very space-consuming to write the If statement as described above. An alternative is to write it in a single line:

```
If x = 1 Then Msgbox (“X is 1”)
```

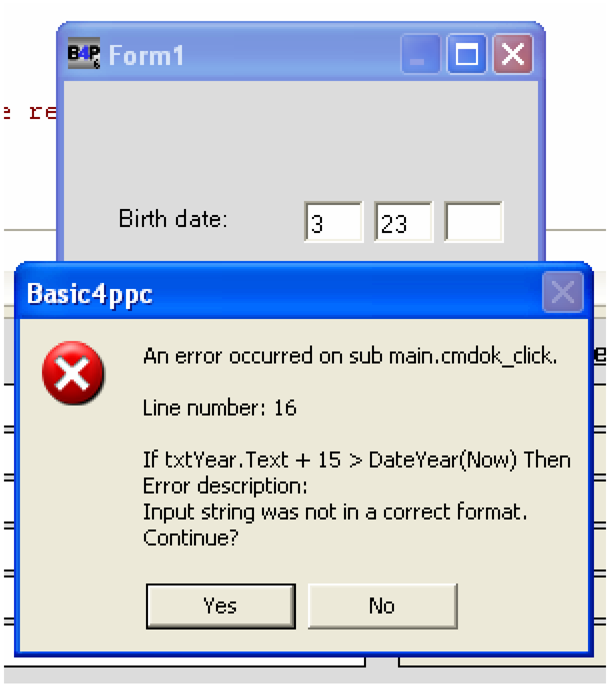
In this case, there is only one command after **Then** and no **End If.** The Else should appear right after the command, and there should be no irregular form of this structure:

If x = 1 Then Msgbox (“X is 1”) Else Msgbox (“X is not 1”)

Note, that any exception here will not compile – use the standard form instead.

Getting to know Basic4ppc

One of the most important things you can learn from this chapter, is what would have happened if you entered values that are unexpected by the software. Try to enter a blank value to the year textbox and press OK:



An error occurred on sub main.cmdok_click.

Line number: 16

If txtYear.Text + 15 > DateYear(Now) Then
Error description:
Input string was not in a correct format.
Continue?

An error message appears, and it says some things you want to read. First, it tells you where the error is, and cites the line. Then it says: “Input string was not in the correct format”. Since you know on which line this happened, you go to line 16 (cited on the message as well, and understand, the text in txtYear (the Year textbox) is blank, so it cannot be added 15. A better practice of programming would have been replacing line 16 with:

```
If txtYear.Text <> “” And txtYear.Text + 15 > DateYear(Now) Then
```

And preventing the error. More about error messages is found on the debug chapter. The

<> operator indicates “different from”. And the expression “” represents an empty string value – a string that has now text in it.

The Not keyword

The Not keyword can be used in Basic4ppc to indicate that the negative of an expression should be evaluated. For example, instead of writing the previous line of sample code, I could have written:

```
If Not(txtYear.Text = “”) AND txtYear.Text + 15 > DateYear(Now) Then
```

The reason I mention it is that unlike other languages, Basic4ppc requires that you use the Not keyword with parentheses following it, thus answering the obvious question “not what?”.

The table below summarizes Basic4ppc’s Boolean operators:

Operator	Meaning
=	Equals
>	Greater than
<	Smaller than
>=	Greater than or equals
<=	Less than or equals
<>	Different from
And	Boolean And (both conditions must be true)
Or	Boolean Of (only one of the conditions must be true)
Not ()	The condition must not be

	true

This is a brief summary of the main things to know about the If statement:

The statement structure:

```
If <Boolean Condition> Then
   <code…>
Else
   <code …>
End If
```

And, Or, Not can be used. **Not** is followed by parentheses, for example: If Not (x = 1) Then…

The statement can be written in a single line:

```
   If x = 1 Then Msgbox (“X is 1”)
```

In this case, there is only one command after **Then** and no **End If. Else** can be added right after at the same line. No exceptions.

The Select Statement

Sometimes, you have to test if a variable (call it x) equals 1, if it is not test if it equals 2, then 3 and so on. It can be done with the If statement as follows:

```
If x = 1 Then
  Msgbox (“one”)
Else
  If x = 2 Then
    Msgbox (“two”)
  Else
    If x = 3 Then
      Msgbox (“Three”)
  End If
End If
```

But obviously this is tedious, error prone and unreadable. The select statement is used just when you need to compare one value to a list of some possible values. The following could be written as:

```
Select x
  Case 1
    Msgbox (“one”)
  Case 2
    Msgbox (“two”)
  Case 3, 4, 5
    x = 2
    Msgbox (“Three or Four or Five”)
  Case Else
     <call a Sub, do whatever…>
End Select
```

This has the same result but is more comfortable. The general structure is:

- **Select** <variable name> starts the statement.

- Each possibility starts with the word **Case** followed by the list of possible values – if the variable is one of them, the code after this Case is executed.
- The statement ends with **End Select**.

The For Loop

Not a beginner? If you have a former programming experience and you only need a brief summary of the Basic4ppc For statement, go straight to the end of the section.

Loops are one of the most basic structures in programming. Suppose you want your program to read an array of numbers, and print the numbers on it to the screen. The easiest way to repeat an action several times is to use a loop, and the **For** loop is the best amongst loops when you can easily calculate the number or repetitions required. Later on this chapter you'll see the While and the Do loop.
This is an example of the For loop structure

```
1  For index = 1 To 10
2    Msgbox (index)
3    Msgbox (index * index)
4    Msgbox (index^index)
5  Next
```

This is just an example. The general structure is this:
First, the loop starts with the keyword **For**. This is line 1 in the example. Then, you must enter the name of a variable that is used as a counter. I chose "index", but it is common to shorten it to just "i". Of course, any variable name is good. Then you tell the program where your index starts and where it ends: mine started at 1 and went to 10: **For index = 1 To 10**. You will write "=" for the starting value, and write the ending value after the **To**

keyword.

This statement utilizes the loop. The lines following it (lines 2, 3, 4 in the sample) are going to execute once each loop, and each time with a different value of the variable “index” attached to them. If you copy this sample to your App_Start, you will get 30 message boxes though only three has been implicitly written. You can follow the execution with the debugger if you wish to further investigate what happens.

The second part of interest here is the essential **Next** statement. This keyword indicates Basic4ppc you mean to repeat only those lines 2, 3 and 4 and not anything else: when the program reaches the **Next** statement it does two things: first, it increments “index” by 1 (this is why it’s called next. So after the first loop it equals 2 and so on) and second, it sends the active statement back to line 1 – right where the For loop started. Index now equals 2, and this will go on until index reaches 10.

If you need index to behave a bit differently you can change the increments by which it is growing. The optional **Step** keyword can be added to the same code like this:

```
1  For index = 1 To 10 Step 2
2    Msgbox (index)
3    Msgbox (index * index)
4    Msgbox (index^index)
5  Next
```

And now, rather than incrementing **index** by 1, it is incremented by the number following the **Step.** If you want to count downwards for some reason, you could have written

```
1  For index = 10 To 1 Step -2
2    Msgbox (index)
3    Msgbox (index * index)
4    Msgbox (index^index)
5  Next
```

Just note that you need to start at 10 and go down to 1 instead of the original values.

Note:

- It is considered a not very good practice to change the value of the index manually inside the For loop. It is allowed in Basic4ppc, but try to avoid it as it tends to cause an error prone code.
- When the loop ends, the value of a variable named index (or whatever you called it) is the last value it had inside the loop.

This is a brief summary of the main things to know about the For loop:

```
The statement structure:
For <variable>  = <value> To <value> [optional: Step <value>]
  <code…>
Next
```

The **While** Loop

Not a beginner? If you have a former programming experience and you only need a brief summary of the Basic4ppc While statement, go straight to the end of the section.

The While loop has the same target as the For loop has. It lets you do something a given number of times, but is more useful when you don't know in advance how many times you are going to do it. A very basic sample, that does just the same the For loop sample on the previous section did, is this:

```
Index = 0
```

```
2  Do While index < 10
3    Msgbox (index)
4    Msgbox (index * index)
5    Msgbox (index^index)
6    Index = index + 1
7  Loop
```

So, what are the differences:

- First, you need to manually set an initial value to your variable: this is line 1.
- Second, line 2 is where the loop starts. It starts with the keywords **Do While** followed by a Boolean expression (just as those used with the If statement, and with the very same rules applied to). This is the major difference between the For and the While loop: the For is a private case of the While, where there is a counter that counts from one value to another: the While loop demonstrated above is nice and working but it would have been much smarter to use a For loop instead. The While loop shows its strength on places where there is no evident counter that counts and instead you are waiting for some other thing to happen – maybe a file to reach its end, maybe an internet data stream to stop and so on. However, for simplicity I chose to stick to the same example, so I used a Boolean expression to count to 10.
- Third, you should increment the variable yourself, if you are using one, or make sure that something happens that sometime will cause your condition to become true (or otherwise, you are on an endless loop).
- And forth, there is no **Next**. Because it does not only count the word "next" may not be correct, so you use the word **Loop** instead.

And as a summary, here is another example (taken from the help file) for using the While loop:

```
Do While Msgbox ("Add another?",,cMsgboxYesNo) = cYes
  i = i + 1
```

```
3  Loop
4  Msgbox (i)
```

This simple code exposes you to some new concepts:

- On line 1, the Boolean condition is **Msgbox ("Add another?", , cMsgboxYesNo) = cYes**. This means that a message box can **return a Boolean value** when activated – it actually always does, but we never check it. It is useful if you ask the user anything and want an elegant code.
- On the same line, there are two words not shown here yet: **cMsgboxYesNo** and **cYes**. These are **<u>system constants</u>**. System constants are somewhat like "predefined variables", which values cannot be changed. They are regular integer constants (unchangeable variables). The **cMsgboxYesNo** tells the message box to add a Yes and a No button to it, and it then return one of the constant values, **cYes** or **cNo**.
- There are many more system constants. Each time you use a control or a method you haven't in the past, read its help file. They are documented there.
- The "i = i + 1" statement in line 2 is not related to the loop at all.
- Of course, the loop will continue until the user chooses No.

This is a brief summary of the main things to know about the Do While statement:

The statement structure:

Do While <condition is true>

  <code…>

Loop

The Do Until Statement

Contrary to the Do While loop, the Do until loops **until** a condition is met. It is a very close relative to the Do While loop, with the only difference the Until loop executes while its condition is False. The last code example on the previous chapter can be rewritten using until as follows, to the same result:

```
Do Until Msgbox (“Add another?”,,cMsgboxYesNo) = cNo
  i = i + 1
Loop
Msgbox (i)
```

Iterating through an array with a loop

It was discussed briefly in the array chapter and an example was given shortly. This may be a good time to go through the major points of the technique and point them out again, as this is a fundamental task carried out often with loops. Consider the following code:

```
Sub Globals
    ‘Declare the global variables here.
    Dim a (10)
End Sub

Sub App_Start
    a(0) = “AAA”
    a(1) = “BBB”
    a(2) = 3
    a(3) = 4
    a(4) = a(2) + a(3)

    For i = 0 To ArrayLen(a()) – 1
            Msgbox (a(i))
    Next
End Sub
```

Main things to note:

* Use the ArrayLen keyword followed by ().

* Loop until the index is the length of array – 1.

* Start the index from 0.

A word of caution about loops

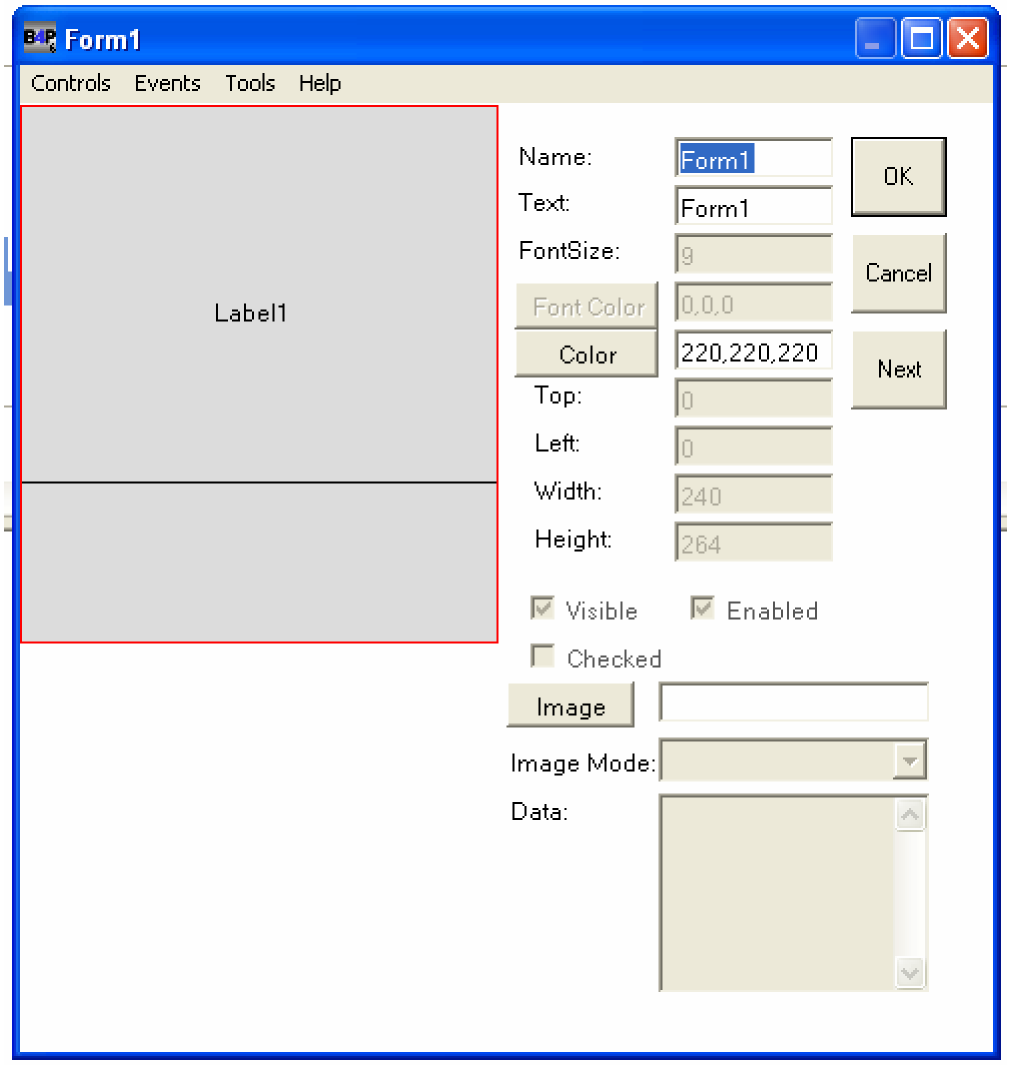
Windows is a multi-task operating system. This means it can let many program operate at once. This does not surprise you, I guess. The thing is, Windows chooses which program should get attention by the intensity of process this program demands. Windows tries to give the program as much attention as needed unless otherwise said in order to improve user experience – this means, when the program needs CPU power Windows gives it so that it can finish what it needs and respond to the user.

When writing a loop, the program repeats an action many times. It is very easy to write a loop that does nothing, but Windows identifies it as its bigger CPU consumer and stops everything else on the computer. Thus, writing an innocent For loop you find yourself with a hung system. Try it – write a loop from 1 to 10,000,000 that does nothing and run the program (save and close anything else before this). You are likely to hang your computer fully or partially. To prevent this, try to make sure you are using only loops that are short enough and have a definitive end, or use the DoEvents procedure. This procedure is used to tell Windows "This is going to take a while, let all other programs tell you their needs". It is not recommended to use it – it is just an offer, a last option kind of solution you should try to avoid and use only when needed. Good programming never uses DoEvents, but it is sometimes inevitable. The better solution is almost always using Timers. More about them on the Timers chapter.

Timers

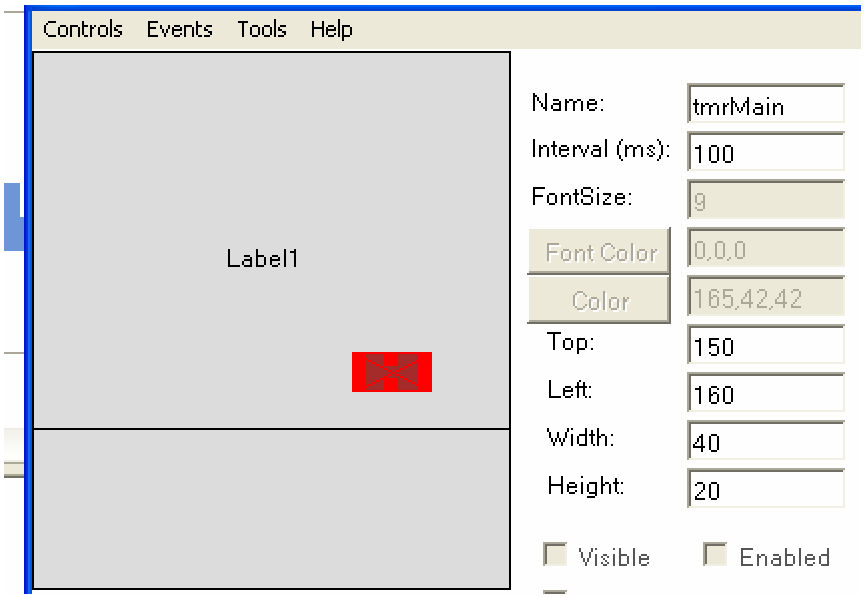
One of the most important, fundamental (yet very easy to use) controls Windows natively supports is the Timer control. A timer is a control that has no appearance: it has no GUI, you cannot click it and there are only few properties to set, despite its name. The only thing a timer does is ticking. A timer, once activated, ticks every short while you set to it. On each "tick" it calls a special Sub – this is the Timer's event, and does whatever is written inside.

Let's start with a simple example. Create a new Basic4ppc program, add a form to it and add a label in the middle of the form called Label1, as shown below:



Name: Form1
Text: Form1
FontSize: 9
Font Color: 0,0,0
Color: 220,220,220
Top: 0
Left: 0
Width: 240
Height: 264
☑ Visible ☑ Enabled
☐ Checked
Image Mode:
Data:

Now, open the Controls menu and select Timer. A box with an X inside appears on your form – this box represents your new Timer. Name it tmrMain and move it a bit aside so it isn't placed on top of your label:



Name: tmrMain
Interval (ms): 100
FontSize: 9
Font Color 0,0,0
Color 165,42,42
Top: 150
Left: 160
Width: 40
Height: 20
Visible
Enabled

Recall that timers do not appear actually on your screen when the program is run, so there is no actual importance as to where the timer is placed. What **is** important, however, is that you tell the timer what to do – you do this, surprisingly, by an Event.

Go to the Events menu, and select the Tick event. Confirm the message box, and a new Sub appears. Now, what we are about to do on this Sub is make your label move around on the form. For this, we will change the value of the Top property (indicating the top coordinate of the control on the form), repeatedly. So write inside the new Sub that appeared:

```
Sub tmrMain_Tick
            Label1.Top = Label1.Top + 5
    If Label1.Top = 220 Then
      Label1.Top = 0
    End If
End Sub
```

Explanation: Line 2 changes the place where Label1 is displayed. We take it down further every tick. If we reached a too low point (this is what line 3 checks) then we make it jump back to the top of the form (Top = 0) on line 4.

If you ran your program now nothing happens (except that the form appears). Try it. The reason is that the timer is not activated yet. There are two things you need to set before any timer starts: the interval value, and then, to set it on. The following code does this. Add it to the App_Start Sub, before the Form1.Show statement.

```
Sub App_Start
        tmrMain.Interval = 500
    tmrMain.Enabled = True
    Form1.Show
End Sub
```

Run your program now and see what happens. The label starts going down. If you stop the program and change the interval to 100 it goes down faster. Save this sample as “timer sample”.

What happened? The timer starts ticking when the Enabled property is set to true. Then, every <interval> milliseconds (500 ms in our case) it calls the Sub “tmrMain_Tick”: the event you added using the designer. The code inside, when executed, takes the label down a bit.

Why not use a loop for this?

Good question. There are two reasons:

- Loops consume a lot attention from the computer’s CPU when running without a defined end (in this case the loop runs forever).

- A loop is synchronous, where timers are asynchronous. This means, that when you run your code inside a loop, you program does just this. So if you are in the middle of a loop you program does not respond to user input (many times the entire system doesn't) and does nothing else. With timers, on the other hand, you can set the timer active, the Sub is being called every 500 seconds and you can go with your program do other things without interrupting yourself with what the timer is doing. If you show a clock with the hour and minutes on the screen, you wouldn't like to have to take care of this minor feature in your main code while dealing with other things and making sure all the time you do it right: set a timer to take care of it periodically and you got the best solution for asynchronous programming, supported right by your OS.
- Loops cannot be timed. There is no way to cause a loop to wait till the next 500 milliseconds have past before the next iteration: it goes over as fast as it can.

Timers overhead

Yet, timers do have some implications you should be aware of. Though they are many times the preferred solution, you should know that:

- **Memory is consumed with each timer**. This is not a problem usually, but on smaller devices and phones, with less memory, there is an overhead that becomes significant if you try to utilize more that a couple of tens of timers. There is no real need for you to do so, but if you did.

- **Speed** of the entire system is reduced with every active timer. When they are not needed anymore, just set the Enabled property to false – this is a good programming practice and may prove vital on devices.

- **Timers are hard to debug**. Later on, you will learn about debugging, and you will notice that a timer – related algorithms, being asynchronous, are harder. You start at a certain point and follow a known route when suddenly some timer-event pops. The solution for this should not be "don't use them". But, as much as possible, use one of these techniques:
    - **Use only the simplest code** needed to be in the event including Sub calls.
    - **Turn the event off.** If you need lot of things to happen in the event, set Timer.Enabeled to False at the beginning of the event, and set it back to On at the end. This way you will prevent the event being called again before all that had to be done is done.

Timers and threading

Though the subject of threading exceeds the breadth of this guide, it is an essential part of understanding timers, so we'll dedicate a word for it. **A thread** is the most basic unit of a program that the operating system knows: this is what is being given time by Windows. Every program has a thread: some have more than one. Most Basic4ppc programs have one (though multiple are possible, using the Threading library by Agraham that will be discussed later). A thread cannot, basically, do two things at the same time. Windows' GUI is no exception, and therefore a question rises as to how timers work. Do they use a separate thread (to which the operating system "gives" different time intervals)? The answer is no. Timers use a separate way to measure the interval – they take advantage of the Windows internal messaging system. This way you can simulate asynchronous programming that is more than enough for 95% of common needs, without the complexity of threaded programming.

This is not anything you should know now, but bear in mind that the Timer control is operated on the same thread as your GUI is on. There is a different kind of timers, much less used, that is operating of a different thread and actually is used as a way to create a different thread easily.

Date and Time

A common task a programmer is required to do is to deal with date and time values. This can be the date when something happens, the date a file was changed, or an attempt to measure the time something takes.

When dealing with time periods, many questions arise. Obviously, there should be a representation of time as a number, and this number should be standardized among all computer system. And different cultures have their ways of displaying dates and times and different people rather use 24 hours rather than 12 or vice versa – a complete mess. The solution implemented in the .NET framework, and which Basic4ppc implements as well, is to use a very small time period as the basic unit (rather than, say, seconds), and count everything in the terms of this unit. The unit is called **Ticks**. A Tick is defined as 1/10,000,000 of a second. The first tick ever (or tick 0) was at midnight, January 1st, 0001 AD. The Basic4ppc Date and Time tutorial on the forum states it was written on September 23, 2007 which is 633261705083378192 ticks. The way it works is that once you have the ticks value you can do whatever you need in order manipulate is, and you can convert it to different **display formats** without any effect of the tick value itself.

What's the time now

A common task is to save the time to some variable. The current tick value in your computer's clock is achieved using the **Now** keyword.

Converting Ticks to Strings and Back

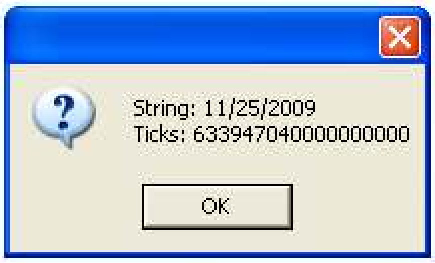
To convert the number of ticks you have at hand to a string a human can read, use this pair: **Date** and **Time.** So if you write this code:

```
Msgbox(Date(Now) & " " & Time(Now))
```

You get a message box with the date and time. The opposite direction is **DateParse** and **TimeParse.** They get a string representing the date or time, and return the relevant tick value. This code demonstrates:

```
D = DateParse("02/03/2004")
Msgbox ("String:" & Date(d) & crlf & "Ticks" &d)
```

And it produces the following message:

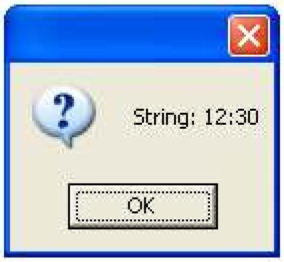


```
String: 11/25/2009
Ticks: 633947040000000000
```

OK

Changing the code to use TimeParse yields:

```
D = TimeParse("12:30")
Msgbox ("String:" & Time(d))
```



```
String: 12:30
```

OK

- **Now on the Device**

When using the Now keyword on the device, take into account that time on the device ticks (when using now, but not with timers – see Timers) every 1 second!

Note: When using the DateParse and TimeParse keywords to convert strings, the strings sent must be in a format compliant with the current DateFormat and TimeFormat settings.

Formatting Displayed Date and Time

In order to change the way date and time appears on your screen when written, use the **DateFormat** and **TimeFormat** keywords to set the date and time format all around your application.

These keyword functions get a string representing the format you wish to set for date and time display. For instance, you can write

```
    DateFormat("dd/mm/yyyy")
```

To force all dates on your app to comply with the "two digits day, two digits month, four digits year, separated by slash" format. The table below illustrates the possibilities:

Time string	Meaning
HH	Hour component in 24 hours display
hh	Hour component in 12 hours display
mm	Minutes component
ss	Seconds component
tt	AM/PM notation
<any character>	Separator between time string

	parts (usually ":" or "-")
Date String	Meaning
mm	Month component
dd	Day component
yy	Year component, 2 digits
yyyy	Year component, 4 digits
<any character>	Separator, usually "/" or "-".

The defaults are:

DateFormat("mm/dd/yyyy")

TimeFormat("HH:mm") – 24 hours format.

Note: When using the DateParse and TimeParse keywords to convert strings, the strings sent must be in a format compliant with the current DateFormat and TimeFormat settings.

Getting part of the date

Many times you need to do something with just the days, or you need to know just the second, for example. The following keywords let you access different parts of the date/time value. Note you should use them with the **numeric ticks value** and not with a string:

DateDay returns the day in the month. For example: on Feb. 3rd , 2052, DateDay(Now) returns 3.

DateMonth returns the month. For example: on Feb. 2nd, 2052, DateMonth(Now) returns 2.

DateYear returns the year. For example: on Feb. 2nd, 2052, DateYear(Now) returns 2052.

DateDayOfWeek returns a string with the name of the weekday for a given date: for example:

```
Msgbox (DateDayOfweek (DateParse("12/1/1900")))
```

Results in "Monday".

TimeHour returns the hour component of the time.

TimeMinute returns the minute component of the time.

TimeSecond returns the seconds component of the time.

Adding and Subtracting time values

A common task you might need to carry out is to add a time period to another date/time value. This can be done using the **DateAdd** and the **TimeAdd** keywords.

DateAdd returns a new ticks-value after adding the required years, months, and days. This code adds 7 days to today's date:

```
NewD = 0
D = Now
NewD = DateAdd (D, 0, 0, 7)
```

The parameters are: DateAdd (OriginalTicksValue, Years, Months, Days) where:

- OriginalTicksValue: the date from which to add.
- Years: number of years to add.
- Months: number of months to add.
- Days: number of days to add.

TimeAdd returns a new ticks-value after adding the required number of hours, minutes and seconds. This code adds 24 hours:

```
NewT = 0
```

```
T = Now
NewT = TimeAdd (T, 24, 0, 0)
```

The parameters are: TimeAdd (OriginalTicksValue, Hours, Minutes, Seconds) where:

- OriginalTicksValue: the date from which to add.
- Hours: number of hours to add.
- Minutes: number of minutes to add.
- Seconds: number of seconds to add.

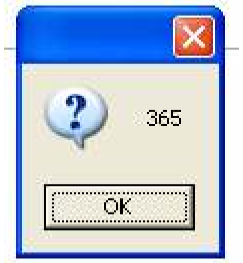
Note: As ticks are just numbers, they could be used inside all kinds of calculations. Furthermore, the number of ticks per day is constant, as is the number of ticks per hour etc.

There are four date and time constants you can use as if you have declared them yourself (as regular variables):

1. **cTicksPerDay**
2. **cTicksPerHour**
3. **cTicksPerMinute**
4. **cTicksPerSecond**

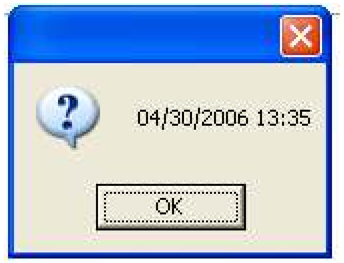
This code calculates the number of days between two dates:

```
d1 = DateParse(“04/30/2006”)
d2 = DateParse(“04/30/2007”)
Msgbox(Int((d2-d1)/cTicksPerDay)) ‘will show 365
```


365

This code gets the value of a specific date and time:

```
d = DateParse(“04/30/2006”) + (TimeParse(“13:35”) Mod cTicksPerDay)
Msgbox(Date(d) & “ “ & Time(d)) ‘will show 04/30/2006 13:35
```


04/30/2006 13:35

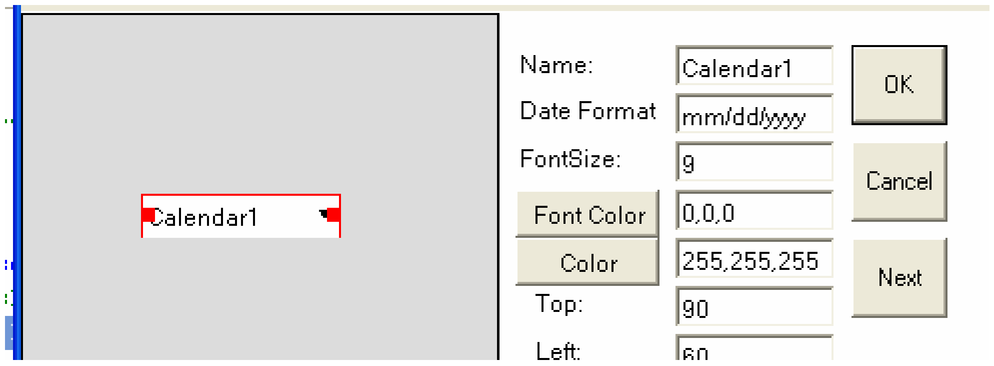
Dates comparison

As dates are stored as regular numbers, all you have to do is to compare them normally. Remember, the only situation in which a date is a string is if you have used the Date or the Time keywords, you have strings. If you did not, you have numbers. So you can use the following code with no problem:

```
If Now > DateParse (“12/12/2010”) Then Msgbox(“bigger”) Else Msgbox (“Smaller”)
```

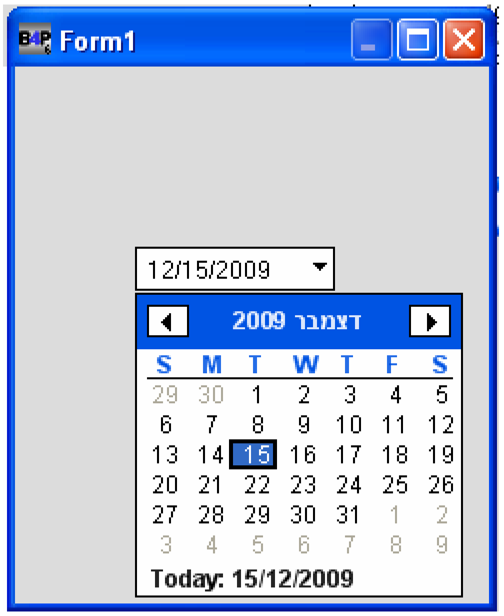
Controls that support dates

The issue of entering a valid date is a complicated task. There are many validity issues and many things to check (just think about the different number of days in February). So the easiest way to let the user enter dates when you are working with GUI is using the **calendar control**. When placed on the designer surface you can set the date/time format to use with it and it looks like this:



Name:	Calendar1
Date Format	mm/dd/yyyy
FontSize:	9
Font Color	0,0,0
Color	255,255,255
Top:	90
Left:	60

When running it looks like this:



12/15/2009

2009 דצמבר

S	M	T	W	T	F	S
29	30	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

Today: 15/12/2009

Usage and formatting

The calendar control allows the user to pick a date, decreasing entering errors. Note, that it also has a **format** property of its own, allowing you to choose a format different than the one used elsewhere in your application. This is not a function as the DateFormat is. This is a property. This means that where you would have used the DateFormat keyword like this:

```
DateFormat (“dd/mm/yyyy”)
```

… trying to do so with the Format property of a calendar control will result in a syntax error. Use this syntax:

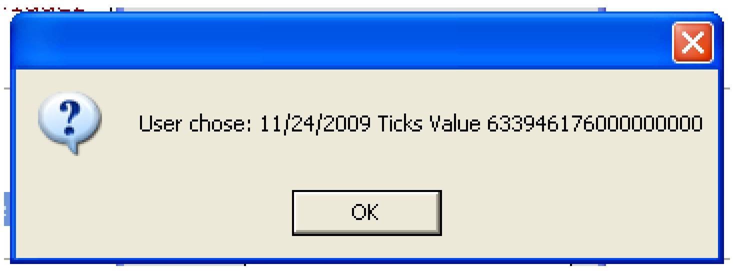
```
Calendar1.Format=“dd/mm/yyyy”
```

to achieve the same effect. However, the format property only works with Date and not with the Time component of the Date/Time tick value. Though sometimes it might display something (depends on your device) it is not recommended to use it for time formatting. See the DateTimePicker control below.

The Value of the date the user chose is stored in a property called Value. This is stored **as ticks**. So if your calendar is called Calendar1, the following code:

```
Sub Calendar1_ValueChanged
  Msgbox(“User chose: “ & Date(Calendar1.Value) & “ Ticks Value “ & Calendar1.Value)
End Sub
```

Will show something like this:


User chose: 11/24/2009 Ticks Value 633946176000000000

You can also set the value of this property to something else to show different date. Calculations carried out with this control and with time values have been discussed many times in the forum. If you are interested, you may want to further read here:

- Displaying partial component of date/time and converting dates: http://www.basic4ppc.com/forum/questions-help-needed/485-calendar-control.html
-

The events supported by the calendar are:

- **ValueChanged:** Fired (that is, the Sub is called) when the value of the calendar is changed. This happens both when the user picks a date, or when the user changes month or year without picking, or when the user changes the manual date entry textbox.
- **Close**: Fired when the dropdown part of the calendar is closed (when the user chooses a date normally).
- **Dropdown:** fired when the dropdown part is opened.

DateTimePicker Control

Using an external control (that is, not a standard part of the Basic4ppc control, but one supplied in an additional library) may be a good improvement of the basic calendar control. This excellent control is provided by Agraham (mentioned earlier – the most influencing, contributing member in the Basic4ppc community, and probably one of the most experienced Basic4ppc programmers (as well as other languages) in the world) as part of his ConrolsExDevice (Ex stands for Extended) library. See the additional libraries chapter for explanation about how to add this, and other, controls from external libraries.

Apart from display enhancements, this control contains an enormous amount of custom date formats to choose from. The help file will do the trick for you.

SysTime library

The SysTime library is an additional library written and supplied to the Basic4ppc community by one of the most productive members, username Filippo. It adds the functionality supplied nowhere else, of changing the system date/time (remember, you can get it with the Now keyword, but you can't set it). See the additional libraries chapter for installation instructions and the help file of this library for details.

OnTime library

Written by the forum user Derez, this library adds some functionality such as time zones, and includes previously described functionality:
http://www.basic4ppc.com/forum/additional-libraries/5515-ontime-library.html#post32456

Errors

Errors and bugs are an integral part of programming. Bugs are usually divided into three types:

- Compile time errors: your program does not compile, and an error message appears when you try to run it.
- Runtime errors: your program compiles, but during the execution it stops and shows an error.
- Bugs that have no error messages: these are the hardest to trap and are usually a result of a failure in the logic implemented.

Compilation Errors in Basic4ppc

Compilation errors in Basic4ppc may appear on two occasions:

a. When you first run the program (remember, when running through the IDE the program is interpreted). Errors appearing at this stage are the bluntest code, syntax or structure errors, often indicating Basic4ppc could not understand what you meant.
b. When you <u>compile</u> your program from the File-Compile menu. When you do so, Basic4ppc goes through a much deeper validating process, as describe in the compilation chapter. This process might return errors even when your program was able to run using the interpreter: this is quire rare. However, the error messages you may encounter here differ in terminology from those you see when running from the IDE.

Generally, understanding error messages Baisc4ppc introduces is the basic thing you have to do to in order to fix the problem. This is no hard thing to do, but take into account that compile time errors indicate sometimes that Basic4ppc could not understand what you have tried to write, and cannot react properly to you. Thus, you may encounter error messages that are not very understandable.

Runtime Errors

Runtime errors indicate the computer was unable to perform what you wanted: this could be because a file you tried to open was not there, or the type of input the user entered was something you did not expect or one of your variables eventually reached a place you did not handle or the hard drive just mechanically failed. These are all events after which your program cannot go on – and it usually crashes.

Error Label

A good program is a one that takes into account all exceptional situations that may occur and handles them in code. In order to do so, Basic4ppc supplies a built in error handling mechanism, called the **Errorlabel Statement**. This statement is written in the beginning (usually, can actually be anywhere) in a Sub and looks like this:

```
'Disconnects the connection.
Sub mnuDisconnect_Click
        Timer1.Enabled = False
        ErrorLabel(ErrorDis)
        If Serial.PortOpen Then Serial.PortOpen = False
        lblCord.FontColor = cGray
        Msgbox("GPS disconnected.")
        Return
        ErrorDis:
        Msgbox ("Error disconnecting")
End Sub
```

This Sub above is taken from the GPS4PPC program that is supplied, as mentioned earlier, as an open source and sample code on the Basic4ppc website. This Sub is the Sub that is called when the user wishes to disconnect from the GPS. Connecting or disconnecting to peripheral devices is always error prone, because it involves communicating with something you are not even sure exist and operating. So it's reasonable to take errors into account: let's see how they are handled.

Line 1 is a comment, describing what this Sub does (This is a good habit in general). Then line 2 is the name of the Sub. The first thing this Sub is doing is to turn off a timer (it's a timer use to check new GPS data periodically). Then, line 4 is the one we are interested in:

Errorlabel (ErrorDis)

And notice the **label** on line 9:

ErrorDis:

So in terms of error handling, the general structure of this Sub is:

```
Sub XXX
  <some code that is apparently not error-prone>
  Errorlabel (ErrorDis)
  <some code where we expect errors might happen>
  <some code>
  <some code>
  Return
  ErrorDis: ' <this is the label name that was specified on the Errorlabel statement>
  <some code that is to be executed in case we get here at all>
End Sub
```

The Errorlabel keyword gets a name of a label that is located somewhere is the same Sub. Recall from chapter 3, that labels are parts in code that "give a name" to the lines following them, and are used also for the Goto statement (the one most programmers deeply recommend not to use). The second place where labels are used is on error handling. The label you specify with the Errorlabel statement is the place to which the program jumps (that is, the active statement jumps) when an error occurs on the code after the Errorlabel. So, in the code above, if an error occurred, the program jumps right to line 9 (the label) and from there goes on: line 10 is a message announcing the user of the problem (he should know there was something going wrong when trying to disconnect), and the Sub ends. Notice another very important part of the code – this is line 8: Return. The keyword "return" causes the program to exit immediately from the Sub it is in and return to where it was. It actually finishes the Sub at once. So, in the Sub above, had no error occurred, the program closes the connection (this is line 5), does another thing or two and reaches line 8, where it exists. When an error occurs, the program jumps right to the label, announces the user of the problem and exits the Sub. Note, that if you did not specify an error label and an error has occurred, then the program crashes.

Limitations of Errorlabels

An error label is a violation of the normal order of program flow. Errors are usually a very exceptional situation (indeed, they are referred to as "exceptions" in the .NET framework). So normal execution is not always possible, and the mechanism of error recovery has some limitations:

- Error labels do not work inside loops. This is not always true – they might, but it is not guaranteed as, depending on how low on the hierarchy between Basic4ppc and Machine Language the error occurred, they may totally break the hierarchy and structure of code inside a Sub, and execution may be forced to jump to the upper level of the Sub. Avoid using error labels inside loops.
- Referring the program to an error label is time consuming. The same reasons mentioned above are the cause that recovering from an error is a process that takes some time – no more that a fraction of a second on normal situations, but much more than what would

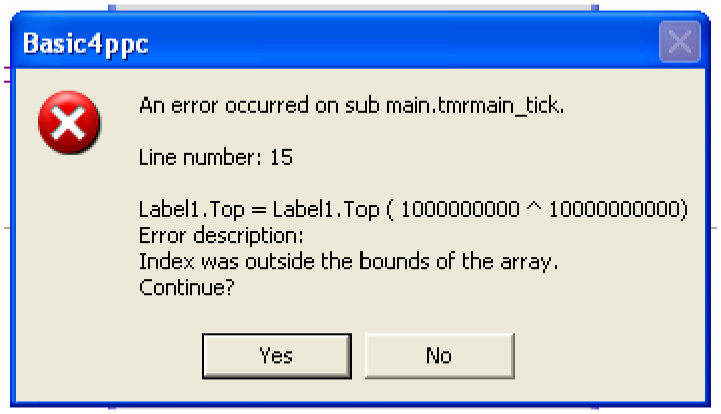
have happened if there was no error. Running a loop that calls a Sub that causes an error on each iteration can thus be significantly slower that handling the different cases in code.

```
Sub tmrMain_Tick
        Label1.Top = Label1.Top ( 1000000000 ^ 10000000000)
        If Label1.Top = 220 Then

        End If
End Sub
```

Error messages

Error messages are the messages you get when something went wrong. Here is an example of a simple error message. Consider the following code:

Which caused the following error message:



```
Basic4ppc

An error occurred on sub main.tmrmain_tick.

Line number: 15

Label1.Top = Label1.Top ( 1000000000 ^ 10000000000)
Error description:
Index was outside the bounds of the array.
Continue?
```

This error message was created easily at runtime by trying to create a number too big for the type of number it was assigned to. It is not the cause of the error that matters for our discussion, but the structure of the error message itself.

a. Always read error messages carefully. Tough (as mentioned above) you might find them confusing, because they are often a result of a failure Basic4ppc could not figure out completely, it is yet crucial to read them.

b. The first line indicates the Sub in which your error happened. In this case it says: "An error occurred on Sub main.tmrmain_tick.". As you can find on the Modules chapter, your program, even if never explicitly declared, contains a module called Main that includes everything you write by default (this is the main program file). Thus, Basic4ppc lets you know on which module the error occurred: main. Then, it tells you on tells you the name of the Sub where it happened: tmrmain_tick on this case. Note, that since Basic4ppc is case insensitive, the original way you wrote tmrMain_Tick is lost – this makes it a bit harder sometimes to understand where the error is.

c. Then, there is the line number on which the error has occurred (the code sample above is cut off the entire program, so line numbering is different).

d. Then, a quote of the line is shown so that you can look at the line isolated. This is where the program failed, and probably the source of the error (though it could be somewhere above, and this is just the place where error has become significant enough to crash the program).

e. Then, there is a short description of the error. Unfortunately, this is not always the best description ever: for instance, on the error above the description was: "Index was outside of the bounds of the array." But hey, you did not use any array. So someone else probably did? Actually yes, but no one told you about it: Basic4ppc uses an array to represent something, but you are not exposed to where and when. So the right thing to do here is to try and to understand what could have been the cause from the "general spirit" of the message – not a very scientific advice but a bit of experience, common sense and forum searching can lead you to the right place.

f. At last, if you have no clue what went wrong, go directly to the forum, and ask. Give as many details as you can (if you can add some code it's best, and the forum members may ask you to upload some source code (zipped). Don't hesitate to use it: the forum is your friend.

Error messages on compiled programs

After compiling your program, error messages you get on the device get changed a bit. Since there is no more Basic4ppc to wrap the messages, the underlying .NET error messages (and on some occasions, event the native operating system's messages) may appear.

Error messages on the device

When working on the device IDE, note that by default the .NET CF is installed without everything available. This includes, strangely, the complete error messages text. There are two files you should download and install on your device before you can actually get proper error messages when writing your programs there. Follow the instructions in the forum thread below to download them. Especially note post number 9.
http://www.basic4ppc.com/forum/questions-help-needed/870-error-messages-device.html

Debugging fundamentals

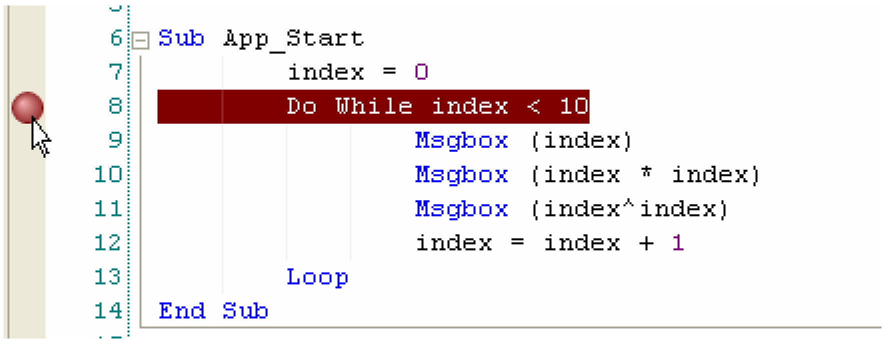
Debugging is the essential process of finding what you did wrong in the program you wrote. Basic4ppc is very forgiving, and there are many ways in which you may cause your program to do something different than what you meant.

Breakpoints

The most basic thing you do when you start debugging your program is to set a breakpoint somewhere. When the program reaches your breakpoint, it stops and lets you see what's going on inside.

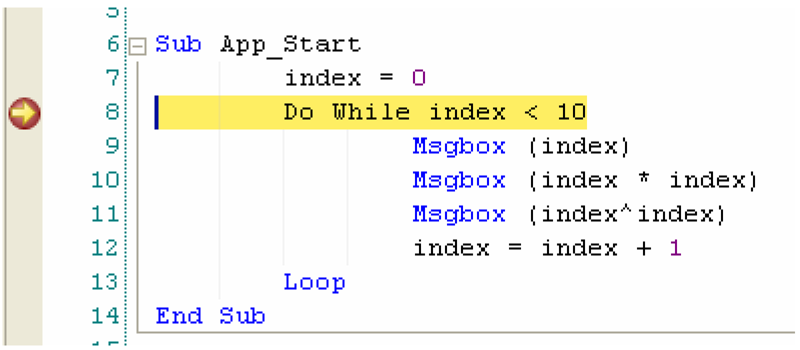
Note: you can set as many breakpoints as you like. For simplicity, I show everything here with just one. There is no limit on the number, and the program stops at each of them.

In order to set a breakpoint, hover your mouse to the left of the line where you need it and click:



```
Sub App_Start
        index = 0
        Do While index < 10
                Msgbox (index)
                Msgbox (index * index)
                Msgbox (index^index)
                index = index + 1
        Loop
End Sub
```

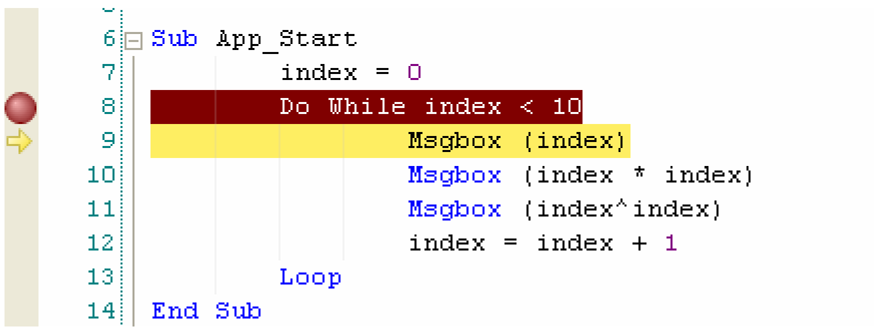
The line on which you clicked is shown with brown background and a brown ball appears to its left. This means that when you run the program pressing F5, Basic4ppc stops at this point and waits:



```
Sub App_Start
        index = 0
        Do While index < 10
                Msgbox (index)
                Msgbox (index * index)
                Msgbox (index^index)
                index = index + 1
        Loop
End Sub
```

Active statement

This is how it looks: the line is highlighted with yellow, and there is a yellow arrow pointing at it where the brown ball is. The highlighted line is the active statement: the next line to execute. You can execute it, and move the yellow highlighting to the next line, by hitting F8:

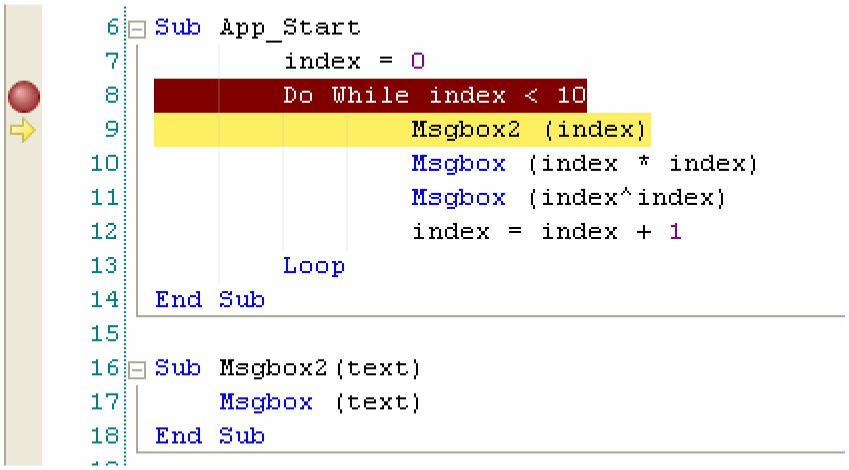


```
Sub App_Start
        index = 0
        Do While index < 10
                Msgbox (index)
                Msgbox (index * index)
                Msgbox (index^index)
                index = index + 1
        Loop
End Sub
```

Different options of stepping

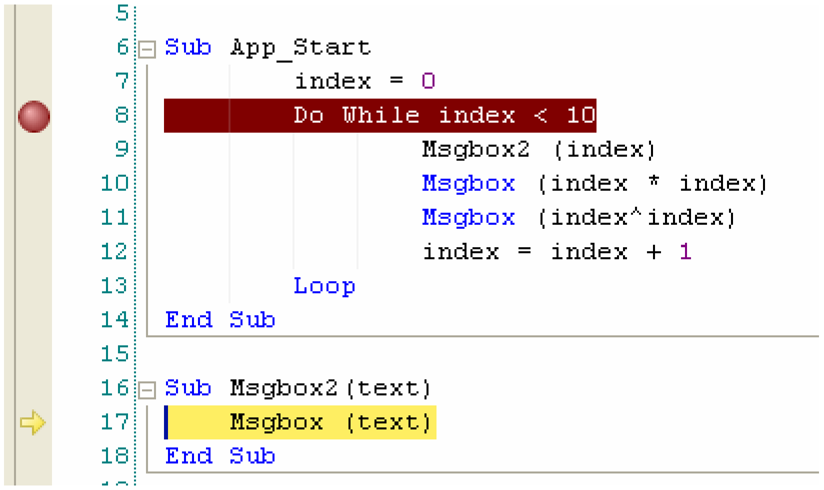
Advancing from one line to the following one is a process called stepping. There are different kinds of stepping you should be aware of when you come to debug your program:

Step (or Step Into) is what you did – by pressing F8 you go from one line to the one after it. What's special about it is that if one of the lines calls a Sub, then you go right into this Sub. Look at the following image, where the previous program was changed a bit – I added a Sub called Msgbox2 that does just the same as the original one. The whole point is to make clear the effect of the Step action:



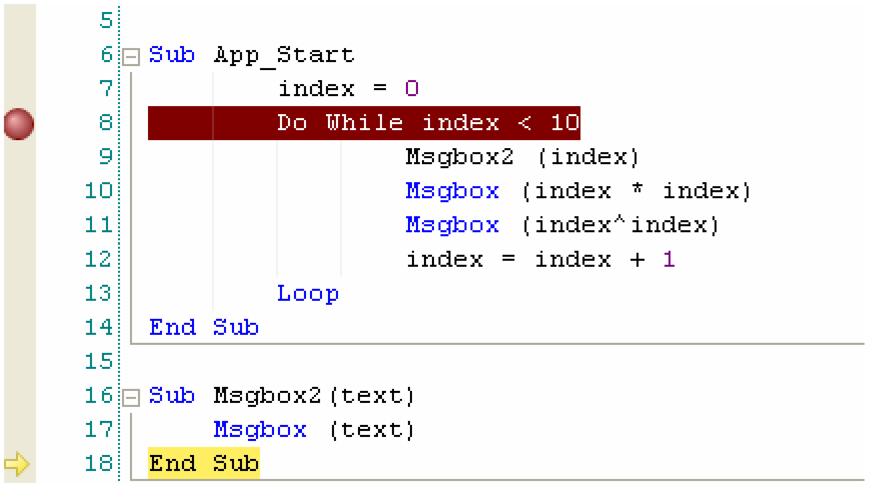
```
Sub App_Start
        index = 0
        Do While index < 10
                Msgbox2 (index)
                Msgbox (index * index)
                Msgbox (index^index)
                index = index + 1
        Loop
End Sub

Sub Msgbox2(text)
    Msgbox (text)
End Sub
```

Note that the following line to execute (in yellow) is a call to the new Msgbox2 Sub. This Sub appears at the bottom of the picture. After pressing F8 again, the active statement is now the first line in Msgbox2:



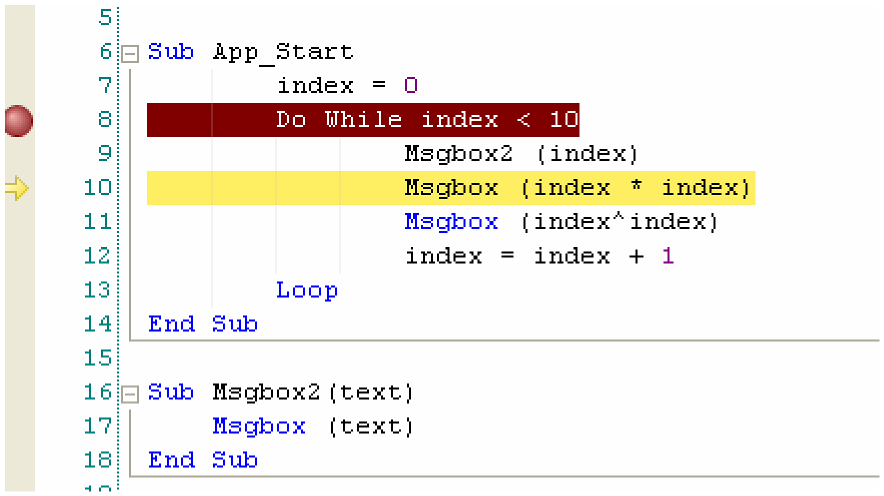
```
Sub App_Start
        index = 0
        Do While index < 10
                Msgbox2 (index)
                Msgbox (index * index)
                Msgbox (index^index)
                index = index + 1
        Loop
End Sub

Sub Msgbox2(text)
    Msgbox (text)
End Sub
```

And, when I press F8 again, it goes on the next line:



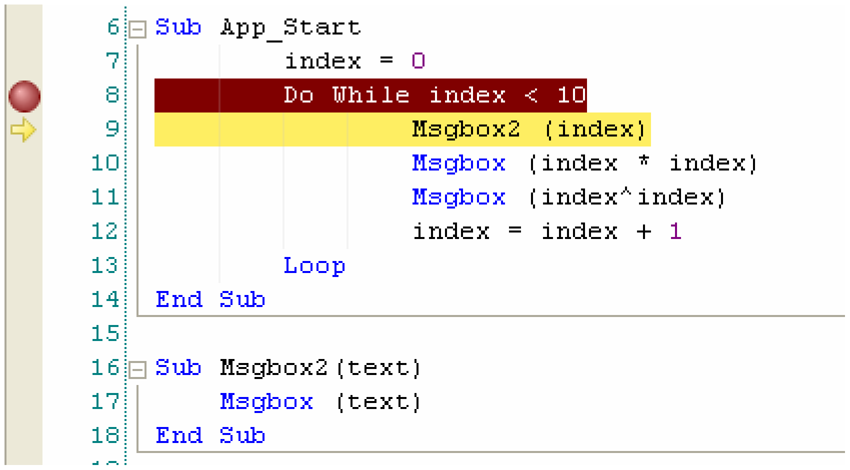
```
Sub App_Start
        index = 0
        Do While index < 10
                Msgbox2 (index)
                Msgbox (index * index)
                Msgbox (index^index)
                index = index + 1
        Loop
End Sub

Sub Msgbox2(text)
    Msgbox (text)
End Sub
```

And from here back to the one after the caller:

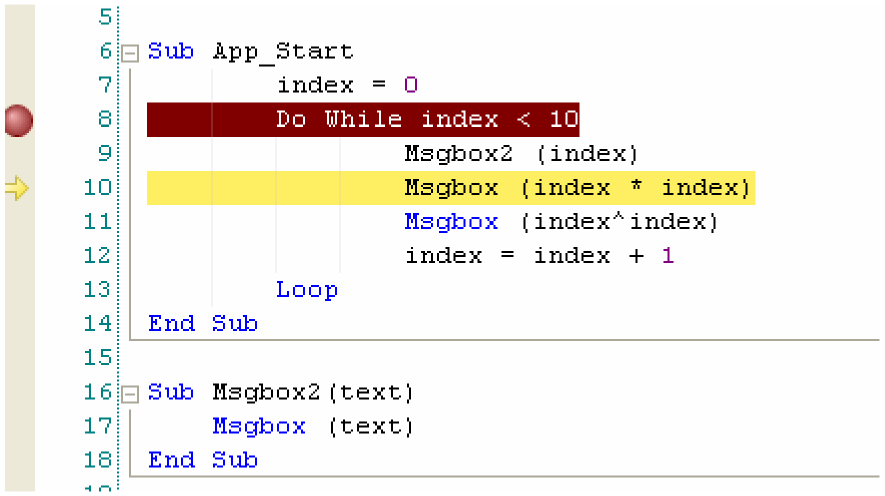


```
Sub App_Start
        index = 0
        Do While index < 10
                Msgbox2 (index)
                Msgbox (index * index)
                Msgbox (index^index)
                index = index + 1
        Loop
End Sub

Sub Msgbox2(text)
    Msgbox (text)
End Sub
```

Step over is what happens if you press the F9 key. This does the same thing, but it does not go into Subs. This is very useful when you want to skip the details of Subs you do not need to debug. So if I pressed F9 rather than F8 on the previous example, this is what that would have happened: First, the active statement was the Sub call:


```
Sub App_Start
        index = 0
        Do While index < 10
                Msgbox2 (index)
                Msgbox (index * index)
                Msgbox (index^index)
                index = index + 1
        Loop
End Sub

Sub Msgbox2(text)
    Msgbox (text)
End Sub
```

And from there we jump right to the next line:


```
Sub App_Start
        index = 0
        Do While index < 10
                Msgbox2 (index)
                Msgbox (index * index)
                Msgbox (index^index)
                index = index + 1
        Loop
End Sub

Sub Msgbox2(text)
    Msgbox (text)
End Sub
```

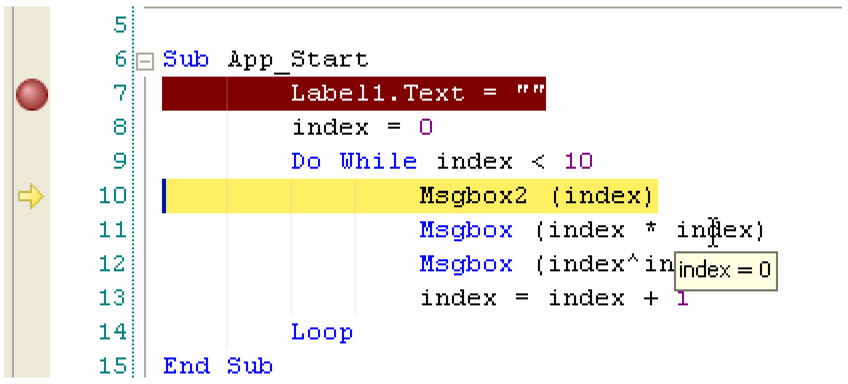
Keep this in mind – it is very useful. The entire process is very helpful when you need to understand what you have done, and the explanation is not complete until we have described the functionality of the F10 key, that does the same thing, except that if you are inside a Sub and press the F10 key, the next line you stop at is not the next inside this Sub but the one after the call to this Sub: this is useful when you are done with debugging a Sub and you want to jump back quickly to the calling Sub. It is called **Step Out**.

Stop is what happens when your program runs and you hit the Ctrl+Q keys or click the stop button. The program basically stops at once, closing every window that is left opened. On some occasions this is not the case, and this includes the case where a message box is displayed. On such cases the program first needs you to confirm the message, or do something else that actively closes an opened window, before it closes.

Watches

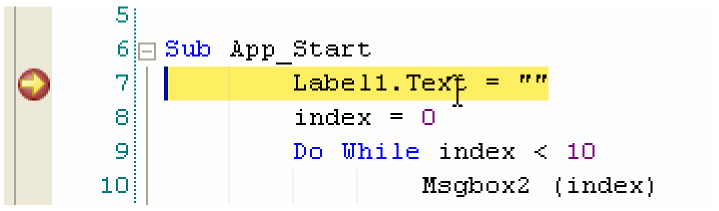
The main idea behind debugging is not quite just tracing where the active statement is. Much more than this, it is the issue of knowing which value is inside which variable. There are basically two ways to do this:

Hover your mouse over a variable name when the program is paused to see its value instantly. This works only with “native”, inner variables you declared: this does not work with controls’ properties or with the values stored inside objects of a library. I have changed the previous program a bit so that it has one more line at the beginning – it does nothing but to demonstrate this issue. When I hover my mouse cursor over one of the variable I declared, its value appears (the variable is “index” in the example):



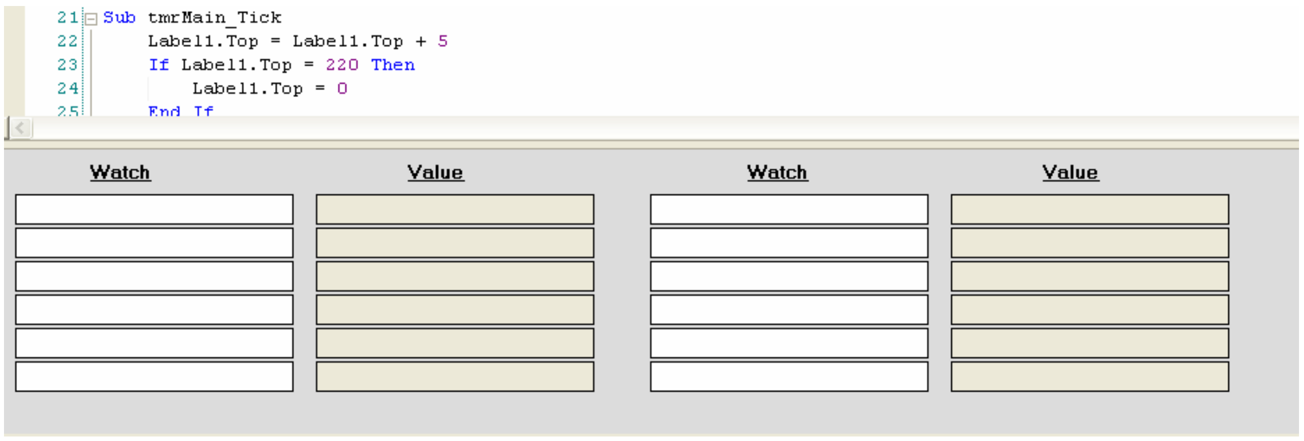
```

Sub App_Start
        Label1.Text = ""
        index = 0
        Do While index < 10
                Msgbox2 (index)
                Msgbox (index * index)
                Msgbox (index^in
                index = index + 1
        Loop
End Sub
```

But if I hover over a property of the Label1 control, nothing happens:



```
Sub App_Start
        Label1.Text = ""
        index = 0
        Do While index < 10
                Msgbox2 (index)
```

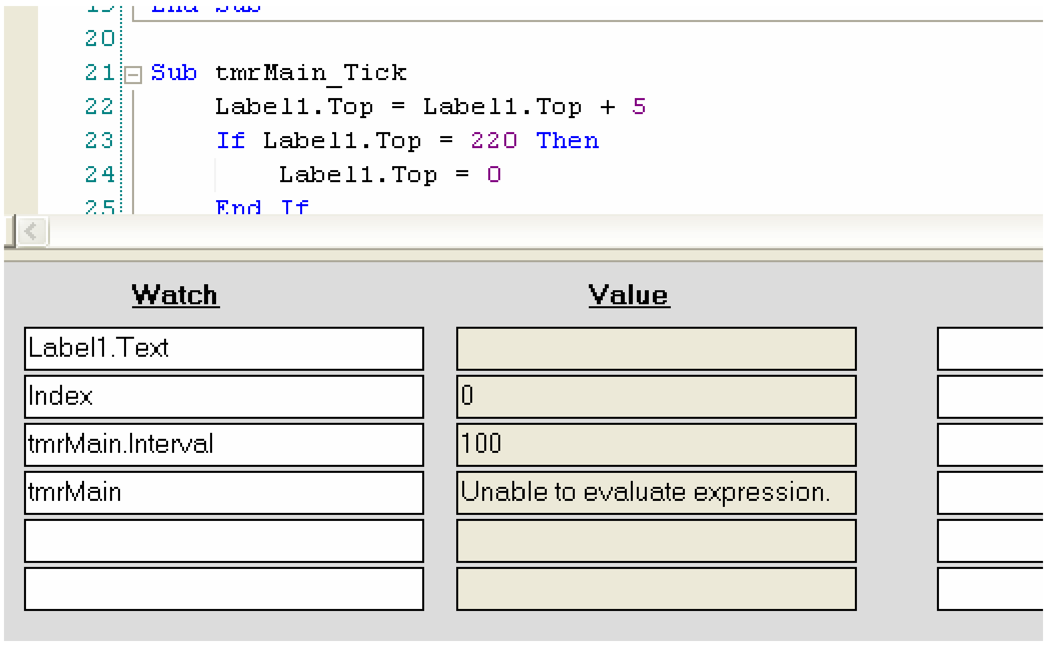
Watches are the right way to spot the values of variables constantly without having to hover you mouse over them or to check the values of variables that you cannot by just hovering. This is the watches table, located at the bottom of your screen:



```
Sub tmrMain_Tick
    Label1.Top = Label1.Top + 5
    If Label1.Top = 220 Then
        Label1.Top = 0
    End If
```

Watch	Value	Watch	Value

There is a column for the expression you wish to watch and a column for its value, and you can write anything you need in the watches column. It's a very powerful tool and it can be used to check values like this:

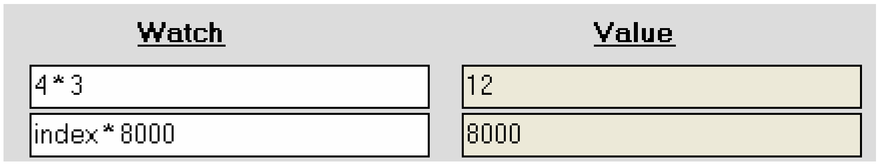


```
Sub tmrMain_Tick
    Label1.Top = Label1.Top + 5
    If Label1.Top = 220 Then
        Label1.Top = 0
```

Watch	Value
Label1.Text	
Index	0
tmrMain.Interval	100
tmrMain	Unable to evaluate expression.

This time I typed some values into the watches column. The first one is the Text property of the Label1 control. The value column next to it is empty, and this means the value of the property is the empty string (“”). The next one is the **index** variable – the same one I watched by hovering. The third one is the interval of the timer **tmrMain**: its interval is currently 100 ms. The last one is the value of tmrMain itself: since this has no clear meaning Basic4ppc displays the massage “Unable to evaluate expression”.

The watches are a very useful tool for debugging and can be used to calculate the value of many expressions while working on Basic4ppc. For example, on a different trial I placed the following into the watches boxes to evaluate:



Watch	Value
4*3	12
index*8000	8000

Debugging on the Device IDE

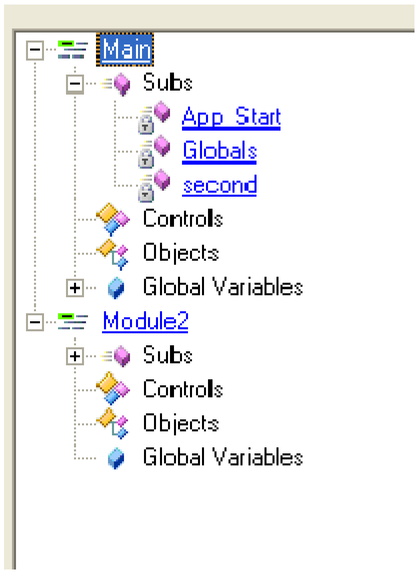
Debugging on the device IDE is a different issue. Though in many ways the Device IDE lets you do almost anything you can do on the desktop, this one aspect is of a grate difference. In order to overcome this issue, an excellent tool called “Debug Recompiler” was developed by who is probably the most important, productive and supportive community member in the Basic4ppc forum – username Agraham. This amazing work, offered as (one of his many) a contribution to the Basic4ppc community, can be downloaded with complete documentation from the link below. The idea behind what he did was, that in cooperation with the developer of Basic4ppc (username Erel in the forum), Agraham wrote a program that re-compiles the applications written for the device. The process is described in details on the excellent help files accompanying the Debug Suite for IDE and is effective for both “legacy” and “optimized” compilation methods. It is highly recommended to uses this software if you need to debug on the device – leave your comments on the forum or ask Agraham for assistance if, after reading the documentation (it is not long at all) you have issues getting it to work. This is where you get this product:
http://www.basic4ppc.com/forum/additional-libraries/4724-debugging-suite-ide-legacy-optimised-applications.html

Modules

This chapter covers one of the topics needed in order to read and write long programs. It is based on the Modules Tutorial in the Basic4ppc website.

What modules are

A Basic4ppc module is the fundamental building block of a project. A module is a code file, holding code, controls, objects and forms. All Basic4ppc programs contain a main module called "**Main**", held in a file with the name of the project and the extension ".sbp". Additional modules are held in files named by the programmer – the file has the same name as the module does, with the extension ".bas". A Basic4ppc project is composed of the **main** module and any number of additional modules. The program execution starts at the **main** module, but other Subs and variables can be stored in any other module. Modules are displayed in the Project tree as follows:



The project in the picture above is divided into two modules – the main module's name is Main (though the file name is different) and the second module's name is Module2, with the file name "Module2.bas".

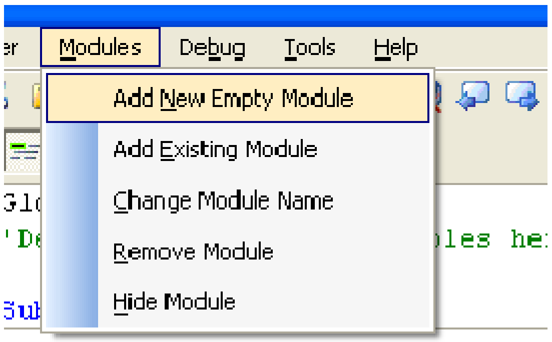
What are modules for

Main advantages of modules are:

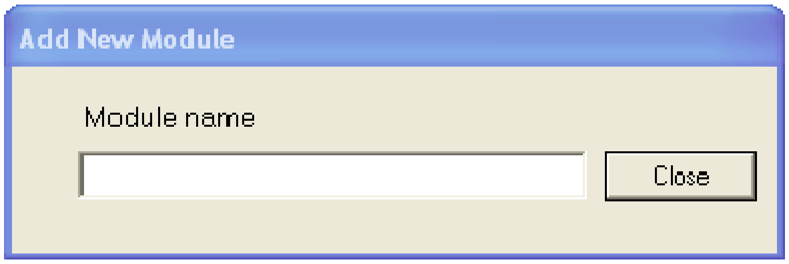
- Reuse: a module written for one project can be used in a second project.
- Generalization: modules can be written as general libraries of procedure with specific Subject, used by many programs and distributed unrelated to any project.
- Encapsulation: a module writer controls which part of the module is exposed to the outer world, thus allow the user to add certain functionality to a project without being bothered by the implementation. Modules can share parts of the code and hide other parts. This is usually achieved by the use of access modifiers, indicating which part of the module is accessible from where.
- Convenience: modules allow splitting large projects into small pieces which are easier to deal with, making them more understandable and easy to manage.

Modules Manipulating using the IDE

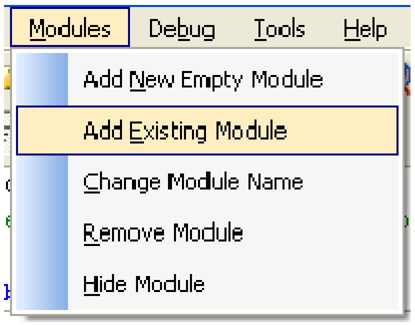
- **Creating a new** module: in the main menu choose **Modules >> Add New Empty Module.**



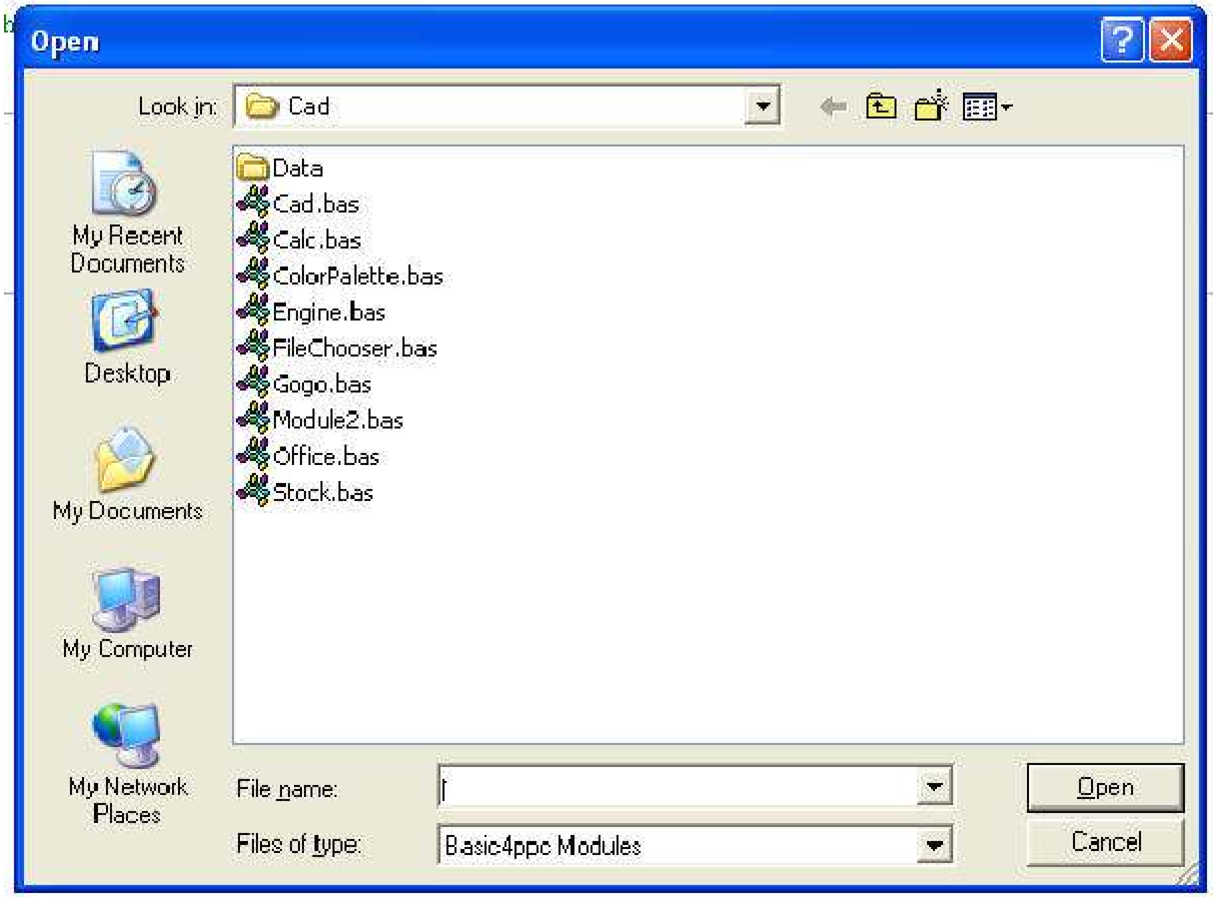
A dialog box appears. Name the new module and click OK:



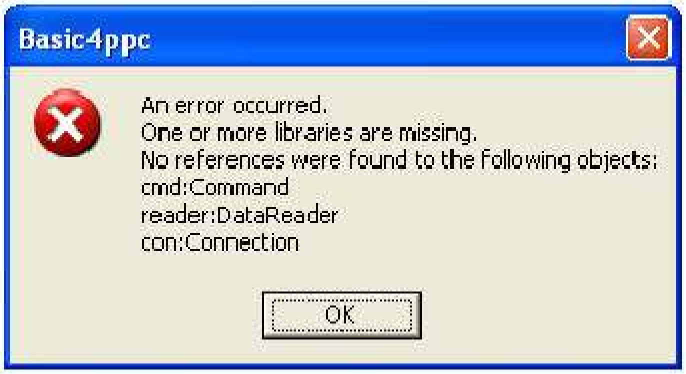
- Adding an existing module to a project: in the main menu choose **Modules >> Add Existing Module:**



Select the module name from the dialog box and click Open:

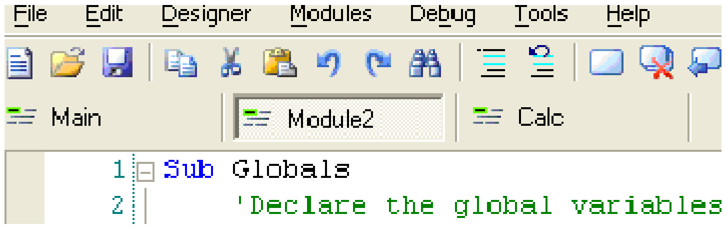


Note: when adding a module that uses libraries, you should manually copy the required libraries into the project's folder (you'll learn about libraries and how to handle them on the libraries chapter). An error appears when adding a module indicating missing libraries, if objects were added to the module using the IDE:



```
An error occurred.
One or more libraries are missing.
No references were found to the following objects:
cmd:Command
reader:DataReader
con:Connection
```

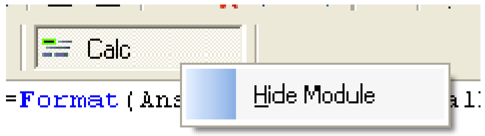
- **Switching** between modules (editing different modules): under the main toolbar, a list of tabs with the names of the modules included in your project is displayed. The one that is active for editing is sunken. Click the one you would like to edit, or press **Ctrl+Tab** to move to the next module or **Ctrl+Shift+Tab** to move to the previous one:



```
Sub Globals
    'Declare the global variables
```

Using **the Device IDE, switching** the active module is done by selecting its name from the modules list under the Modules menu.

- **Hiding a module** from this list is done by right-clicking its tab, and selecting “**Hide Module**”. The main module cannot be hidden:



- **Changing a module’s name**: from the main menu, choose **Modules >> Change Module Name**. In the dialog box give a new name. The name of the module that is currently active will be changed.
- **Removing a module from the project**: from the main menu, choose **Modules >> Remove Module**. The module that is currently active will be removed from the project.

Coding with Modules

Variables Scope

Generally, all variables in Basic4ppc are accessible from within their parent Sub only. Only variables declared in the Global Sub are accessible from anywhere inside their module. This is called the variable's **scope**.

Access modifiers

By default, all variables' and Subs' scope is limited to the containing module. In order for them to be accessible from other modules, their declaration must be preceded by the **public** access modifier. Variables can become public when declared in the **Globals** Sub of any module with the **public** access modifier, for example, if the **main** module contains this declaration:

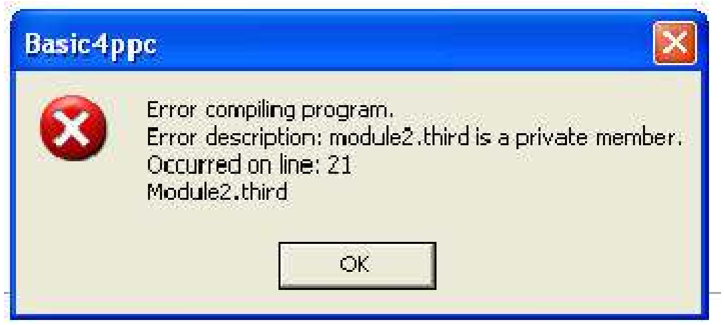
```
Sub Globals
    Public someFlag
End Sub
```

In this example, the someFlag variable is public, hence accessible from different module. Accessing it will be done by writing the containing module's name followed by a period and the variable's name as follows:

```
Sub someSub
    Main.someFlag = 10
End Sub
```

Error message when trying to access a private Variable or Sub

When trying to access a variable or Sub that was not declared with the **public** access modifier, you might encounter the following error message. In this case, add the public keyword to the relevant declaration. Note that the term "member" in this message is used to indicate either a variable, or a Sub. See details about Subs' scope later this page.



```
Error compiling program.
Error description: module2.third is a private member.
Occurred on line: 21
Module2.third
```

Private access modifier

When declaring variables in the Globals Sub using the Dim keyword rather than the Public access modifier, variables declared will be treated as private – they will be accessible only from within their own module. The same behavior can be achieved using the **private** access modifier instead of the **dim** keyword. The following two code segments have the same exact functionality:

```
Sub Globals

    Dim someFlag

End Sub
```

```
Sub Globals

    Private someFlag

End Sub
```

The **private** access modifier is supported for readability only.

Duplicate names

Two modules can have variables with identical names. Accessing other module’s variable when it has the same name is done by explicitly indicating the owner module’s name. For example, in a Sub-module of a project (not the Main module), we can write:

```
Public Sub Third
  Msgbox(Main.b)  ‘show the Main module’s variable “b” value
  Msgbox(b)      ‘show this current module’s variable “b” value
End Sub
```

Accessing Subs

Everything written above about variables applies for Subs as well. Subs do not have to be declared in the Globals Sub, of course, but apart from it behave the same as variables, regarding scope:

- By default, Subs are accessible only from within their module.
- Adding the **public** access modifier to a Sub declaration will allow you to access it from a different module, thus exposing functionality. Having the following code in the **Main** module, for example:

```
Public Sub someSub
    MsgBox (“Some Sub Calling”)
End Sub
```

… will allow you to access someSub from within different module as follows:

```
Sub otherModuleSub

    Main.someSub

End Sub
```

- Subs can be preceded by the **private** access modifier: this has the same functionality as not writing any access modifier, as Subs are private by default.

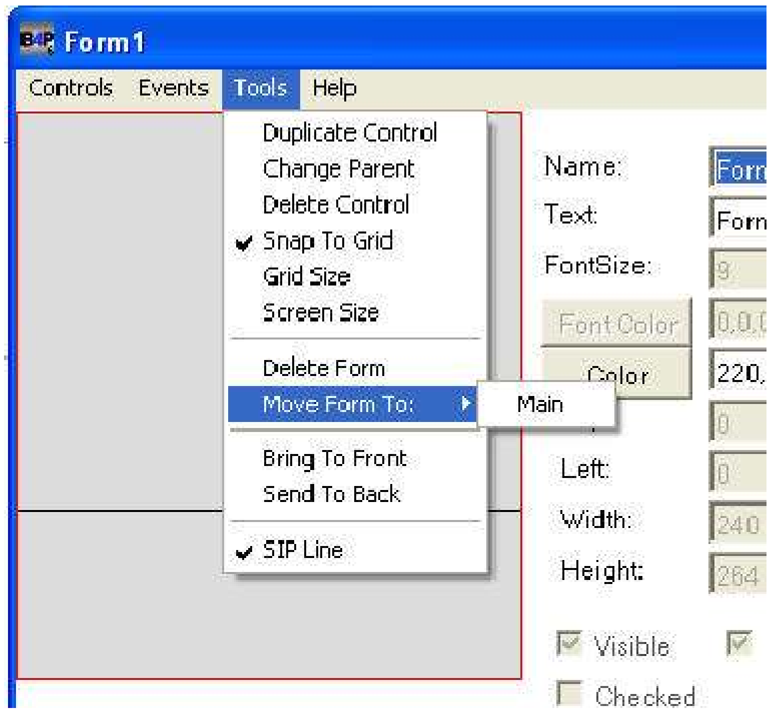
Accessing Controls and Objects

Controls and Objects are always Public. They cannot have access modifier and are accessible explicitly from within other modules using the standard syntax: [module name].[control name].

Forms and modules

A module can contain any number of forms. Like other controls, forms are always public. When adding a new form to a project, you can choose either to add it to the current module, or to a new one. A good habit is to keep each form in its own module. This makes it easier to replace the entire code later when demands change and improves readability.

After selecting the module in which the new form is added, you can always change its module from the Visual Designer window, by selecting from the main menu **Tools >> Move Form To:**


Form1
Controls
Events
Tools
Help
Duplicate Control
Change Parent
Delete Control
✓ Snap To Grid
Grid Size
Screen Size
Delete Form
Move Form To:
Main
Bring To Front
Send To Back
✓ SIP Line
Name:
Text:
FontSize:
Font Color
Color
Left:
Width:
Height:
Visible
Checked

Additional Libraries

This chapter covers a most important concept in Basic4ppc – the use of additional libraries lets you extend the ability of the language to an unlimited breadth. It is based on a tutorial by the same name in the Basic4ppc website.

contents of this chapter

This online reference holds two main sections – a tutorial for quick start, and a full reference article.

- **Basic4ppc libraries – developer's tutorial** use this section for an easy, step by step tutorial of how to add and use libraries in your application.
- **Basic4ppc libraries – a complete reference** - this is the section given here.

Use this section for a comprehensive discussion about the topic, containing references to external resources, examples and detailed explanations of all aspects.

- Basic4ppc libraries basics
- Using libraries in your application – a comprehensive practical discussion
- Analogy
- Concept
- Three basic steps
    - Adding a library to your project
        - Dependency
        - Merging
    - Adding an object
        - At design time
        - At runtime

        - Instantiating: making room for the object
            - Constructors
            - Overloading
- Getting help using a specific library
- External and Internal controls and the differences in usage
- Common error messages you may encounter
- Existing libraries – list and links
- Deploying libraries with your application
- Frequently asked questions and links to answers
- See Also

A Basic4ppc library

Generally, A library is a file that contains software code and data that can be used by various programs. For example, a library that holds functionality for Internet-connections might be used by several programs running and accessing the Internet at the same time. A Basic4ppc library is the same: a collection of Subs contained in a separated file which can be used by many programs. In order to use the code held in such a file, you add it to your Basic4ppc project, then add an object from the library's objects (which is similar to declaring a variable), and then create an instance this object, using a constructor. Then you call the Subs encapsulated inside this file. For example: suppose two different programs need the ability to draw charts. It is then reasonable for both to add the Charts library to the project, thus importing a large amount of new Subs otherwise unsupported. Changes carried out later inside the Charts library will affect all next compilations of the using applications, and there will be no need to maintain duplicate code.

What are libraries made of?

Libraries are just like any other software code. The main difference from a regular Basic4ppc program is that libraries cannot be run separately (they lack the App_Start Sub), as a stand-alone program. Their Subs must be called by a caller program. This can be compared to taking out of the main source file a big group of Subs and use them as "external" code. Basic4ppc libraries are made of Microsoft's .net dll files that are built especially for accessing them from a Basic4ppc program.

What are libraries good for?

Using libraries enables you to:

- Reuse code: write once, use many times.
- Use other people's code.
- Enjoy extended functionality.
- Save time and maintenance efforts.
- Simplify upgrades.

Are there drawbacks?

There are:

- Dependency: in order to deploy your software you should deploy the relevant library as well. Same is true with compiling and running. The program is no longer self contained, and might experience problems if the needed library is absent. Basic4ppc usually overcomes this issue through the practice of Merging libraries.
- Versioning issues: even when found, a library might be of an inaccurate version. This is not always known until too late.
- Complexity: rather than having everything wrapped together at your project you use external code, and you have to find out how to use it.
- Documentation: being developed by community members and not only by Anywhere software, Libraries documentation depends on the developer's good will.

Types of libraries:

Basic4ppc introduces two main types of libraries: desktop libraries and device libraries. The difference is a result of the differences between the Microsoft Mobile .NET compact framework (CF) and the full version. Not all functionality found in the full version is supported in the compact framework's version. In some cases, those differences led to the development of two files with a similar name. Such a case is the Serial library. There are two files for it – SerialDesktop.dll and SerialDevice.dll.

Using a library in your Basic4ppc project

This section covers in details all knowledge required to add and use a library in your project.

The coffee cup analogy: if you are new to libraries

Using libraries includes some topics usually not needed for development with Basic4ppc. Such topics are creating and instantiating "objects", a method of joining components to your program without having either to deal with how they are built, nor add too much of them as "native" Basic4ppc commands. The following section covers the concept of adding objects to a program through an analogy.

Using libraries can be thought of as analogous to the process of making a cup of coffee in your kitchen. Your kitchen is probably a great place, but unless you have some products in it, it lacks the ability to "create" a cup of coffee. Basic4ppc's IDE is the same: your can do anything, you just have to have the right "products" inside.
Imagine standing at your kitchen yearning for coffee. The first thing you do, is purchase a new jar of coffee. This can be of any brand, and can be compared to step 1: adding a library to a project. After you have done this, you now have a kitchen with a jar full of coffee: just like a project with a .dll file added to it.

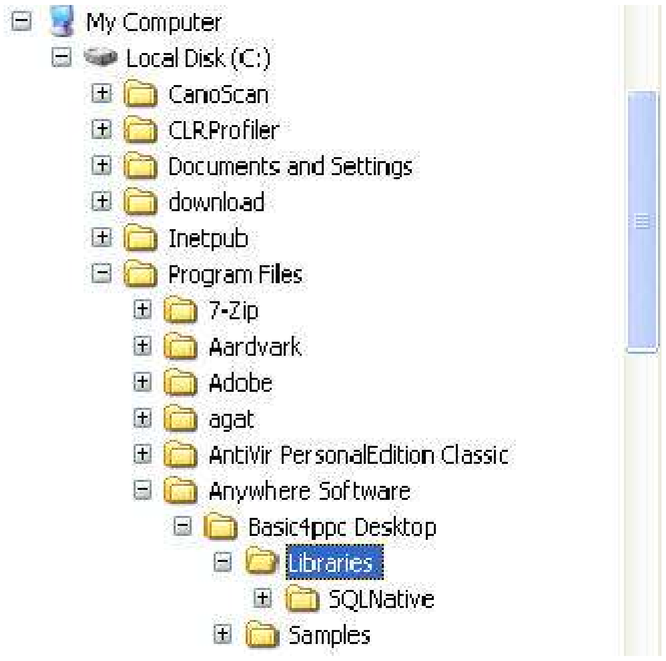
Second, you use a teaspoon and to take some coffee out of the jar. This teaspoon full of coffee will be used immediately, blended with hot water, sugar and cream. The coffee spoon can be thought of as step 2: creating an object of your library. The jar of coffee you've bought isn't the right thing to use for your coffee cup: it's just the container. Use a spoon to "create" a usable "coffee component" for your cup. Same applies for your library: a library file added to a project is a container. Create an object, a spoon, to be able to use it. Think of a library file as an endless jar of coffee: your can take out as many spoons ad you need, the amount left inside doesn't change. The same way, you can have as many libraries' objects as you want: just as if you would have made more than one cup of coffee. Blending the coffee with its ingredients is what gives the final product its taste – sweeter or less sweet, stronger and so on – many cups of coffee started with the same teaspoon taken out of a jar.

Third, you place the coffee from your spoon into a cup. The cup is the place where things are mixed, and a real coffee is created from the ingredients. This is analogous to step 3: use the library's NEW() Sub to start working with it. Just like placing the coffee inside the cup dedicates a special three-dimensional space for your coffee, the same way using the NEW() Sub allocates a special place in memory for your library's object. You must spare this place for your object, just the same way you must place your coffee inside a cup.

Basic4ppc libraries – Concepts

All Basic4ppc libraries are .dll files contained in a special folder under the application's folder (note: the terms "DLL file" and "Library" will have the same meaning in the documentation). Suppose your Basic4ppc is installed in C:\Program Files\Anywhere Software\Basic4ppc, then the libraries will be held under C:\Program Files\Anywhere Software\Basic4ppc\Libraries. It is important, because Basic4ppc will look for updates for the loaded libraries in both the project's folder, and the [application-path]\Libraries (C:\Program Files\Anywhere Software\Basic4ppc\Libraries in our example) (This applies to versions prior to V6.8. In earlier versions, if you choose to add a library from a different location, Basic4ppc will attempt to copy the library to the \Libraries folder).

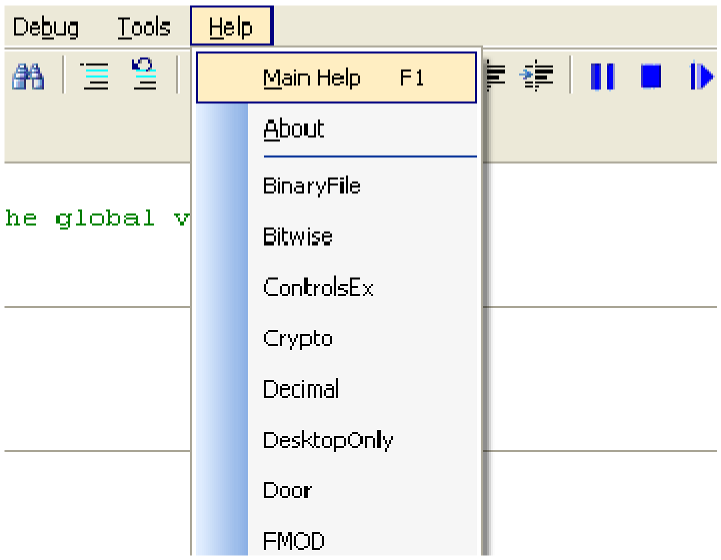
Note: holding libraries under the \Libraries folder is recommended, but is not obligatory.



Default location of the Libraries' folder

The libraries folder contains not only the library itself (a .dll file). Although optional, for most of the libraries it includes 3 files with identical names and different extensions. If a library is called, for example, lib.dll, then the \Libraries folder shall include the following files:

1. Lib.dll – the library file itself. This file is required in order to use the library's functionality in your code.
2. Lib.chm – a help file associated with the library. When found under the \Libraries folder, its name will be added to the list of available libraries' help under the Help menu, as in the following picture (note: this applies only to the desktop IDE and not to the device IDE):



A list of available libraries under the help menu. This list is built when running Basic4ppc, and contains all files with the .chm (windows help files) found under the \Libraries folder and associated to a .dll file.

3. A .cs file (c sharp, .NET's technology main programming language). When compiled, Basic4ppc programs are first translated into c# programs and then compiled using standard c# complier. Including the .cs file in the libraries folder enables **Merging.** Merging will be discussed in details later this chapter.

Using DLL (libraries) in your project

Using DLLs in your Basic4ppc project has 3 steps. They are discussed in details below:

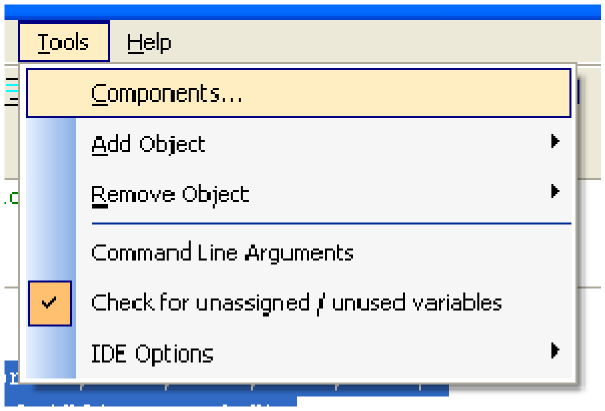
1. Add library to the project: attach the .dll file(s) to your project.

2. Declare the variable name: create an object from within the library added.

3. Save room in memory for the object: call the object's NEW Sub (exists in every library).

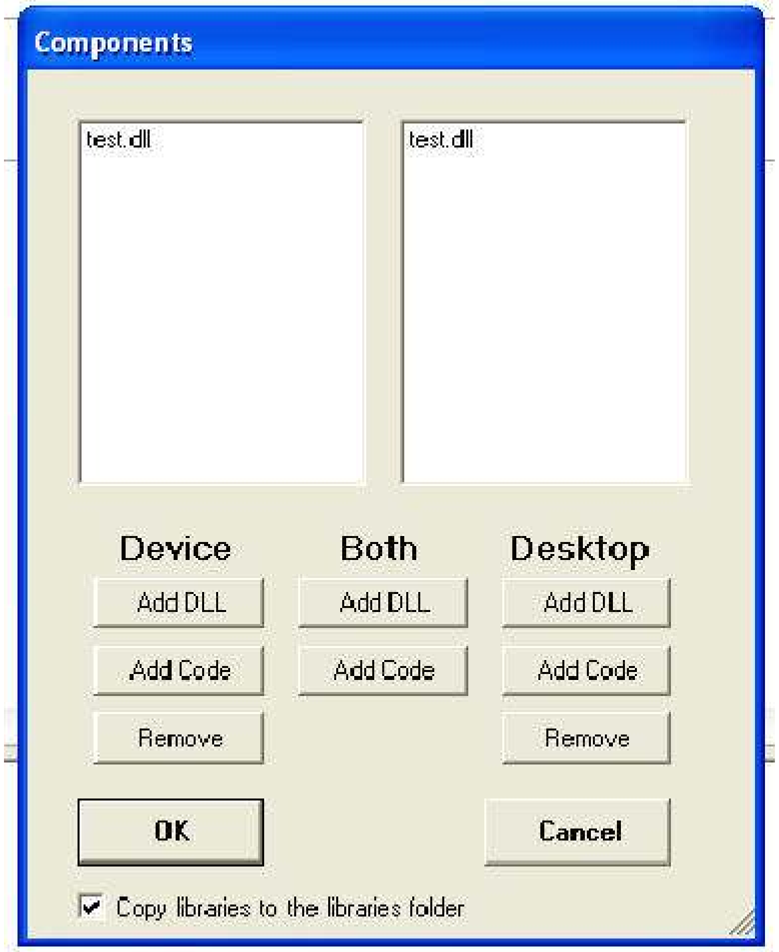
4. Start working…

In order to start using a library you must first add it to your project. This is done from the IDE. Open the Tools menu and select Components:



Tools/Components menu

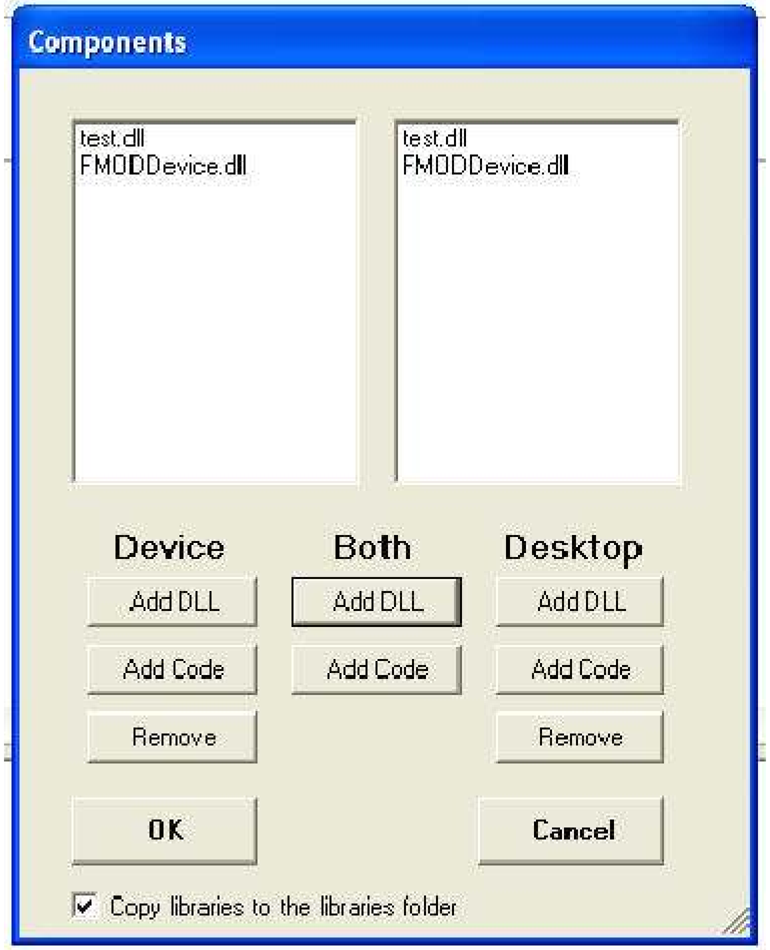
The following window appears, allowing you to choose the libraries file to be added to your project:



Components dialog box

Adding a library is done by clicking the “Add DLL” button. The white columns represent the files added in your project for both your desktop application and your device application. Sometimes, .dll files are different between the two. A convention is to name the .dll with the target platform’s name attached in these cases, such as “serial**Desktop**.dll” and “serial**Device**.dll”. Follow this convention when deploying dll’s you have written yourself, to prevent user mistakes. In case each application (desktop/device) uses its own .dll, use the “Add DLL” button found under the target-device’s name to select the wanted file. In case the desktop and the device use the same library, use the middle button to select the file. It is your responsibility to know whether or not to use the same file for both .

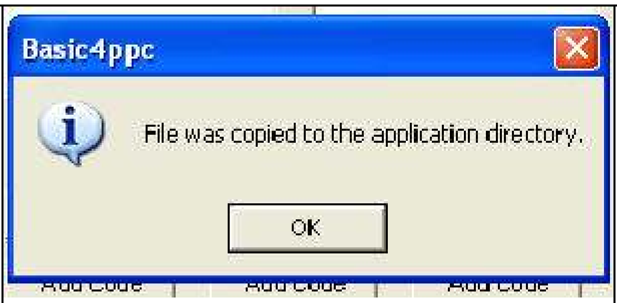
Basic4ppc will not find this out for you and the following situation is possible, resulting in disability to run the desktop application:



Libraries adding dialog box, in which the same .dll file was added for both desktop and device applications. This might cause unexpected behavior.

Copying the libraries into the application folder and working with a recent version

When you add a library to your project, Basic4ppc will copy the libraries file to your project folder. This is the reason why you should save your project prior to adding a library (otherwise there is no project folder). From this moment on, Basic4ppc uses the copy held under the project's folder (Versions prior to 6.8: a message will appear indicating success or failure to copy the files.).

	 Basic4ppc File was copied to the application directory. OK	
	A message indicates the successful copy of a selected library (V 6.5 and below only)	

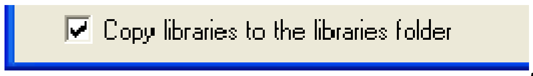
Removing a previously selected file from a list is done by first selecting the file name, and then clicking the "Remove" button under the relevant list.

The Add Code buttons

The two Add Code buttons below the libraries lists both function as a special way to manage differences in application's code easily. Similarly to the way modules work in Basic4ppc, this option allows you to integrate to your program an external file containing Basic4ppc code. This code is merged "as is" into the main module at compile time, thus allowing you to call its Subs and so on. Unlike with modules, this way easily allows you to include **totally different basic code** inside your Device application than this included in your desktop application. This is a powerful feature beyond the scope of this article. However, it's worth mentioning.

Copy libraries checkbox

If you leave the "Copy libraries to the libraries folder" checked:



, Basic4ppc will copy the library's file to the default libraries folder as described above.

Note: starting from V6.8, Basic4ppc no longer copies the files to the default libraries folder and this checkbox does not exist. This has a number of consequences, mainly a difficulty in finding a library not in the default folder, and the lack the library's name under the Help menu. You should take care yourself of placing your libraries where you want them.

When finished with the Components dialog box, click the OK button to finish the DLL selection part.

Notes

- Starting from V6.8, in order to prevent versioning conflicts, each time you open a project Basic4pcc checks for a newer version – based on its modification time –in the libraries folder, and copies the new file, if found.
- Exception when working on the device IDE:
  - Note that **libraries are not automatically copied to your device when installing Basic4ppc.** Make sure you manually copy all files required for your application to the app folder.
  - It is a good habit to keep your Basic4ppc source files entirely under the synchronization folder of your ActiveSync. Many versioning issues are solved just by keeping dlls synchronized. Keep everything in Subfolder there during development, and ensure you are always on sync.

Dependency

When working with DLLs, your application becomes dependent on its libraries in order to compile and run. Some dependencies issues can be eliminated by merging the DLLs into your application upon compile (see details below), but not all libraries can be merged. DLLs themselves may be dependent on other DLL files, and so on. Basic4ppc automatically tracks dependency issues when copying the library into your project's folder, and warns you if a dependency problem rises.

Merging

As stated, libraries must be found in the application folder for the application to be able to find them when needed: failure to find a .dll file will result in a runtime error and execution will stop. It is the developer's responsibility to make sure that all needed files are deployed and copied correctly to the destination machine. Either by building a setup program or by any other method of instructing the user, files should be where needed when application is run.

In order to eliminate to minimum the need to manually handle additional files issues, Baisc4ppc allows for merged files into your application's executable. If a library's source code (c# code) can be found under the \Libraries folder, the code will be inserted as is into the application's main executable upon compilation. This way the .dll is merged into the application and becomes a part of it, thus making deployment significantly simpler.

All official Basic4ppc libraries can be merged (that is, they all contain a [dll-name].cs file with their source code). Not all external libraries support this feature, from various reasons. If you do not have the .cs file, you have to take care yourself of the deployment issues.

Step 2 of 3: Creating an object of a library you've added (all steps)

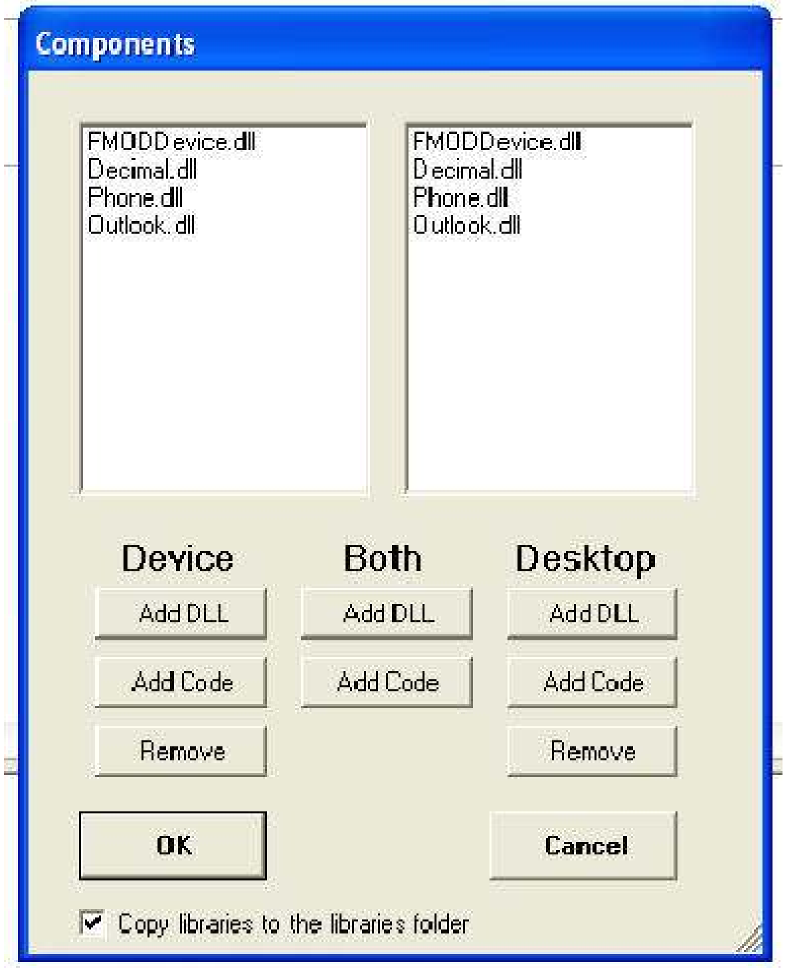
After you've added desired libraries to your project, you must first create an object of each one. Going with the coffee cup analogy, this is analogous to taking a teaspoon full of coffee out of a magical, endless coffee jar. Although have different meaning in other programming languages, in Basic4ppc the term Object is used to indicate the "usable variable held in a library". Each library can contain more than one object type. You create your objects from the object-type, as if you take a spoon full of coffee out of a coffee jar. You can create as many objects of a single library file as you want. Each one can have slightly different characteristics. For example, you can have 3 different TreeView objects, each with different nodes in it, different colors ect., but they all are objects created from the TreeView library.

Put simple, creating an object is equivalent to declaring a variable. In Basic4ppc, apart from arrays you don't <u>have</u> to declare any variable, but it is allowed to do so using the Dim keyword. Adding an object is actually declaring a variable, with the type of the object (referred to as "object type" later).

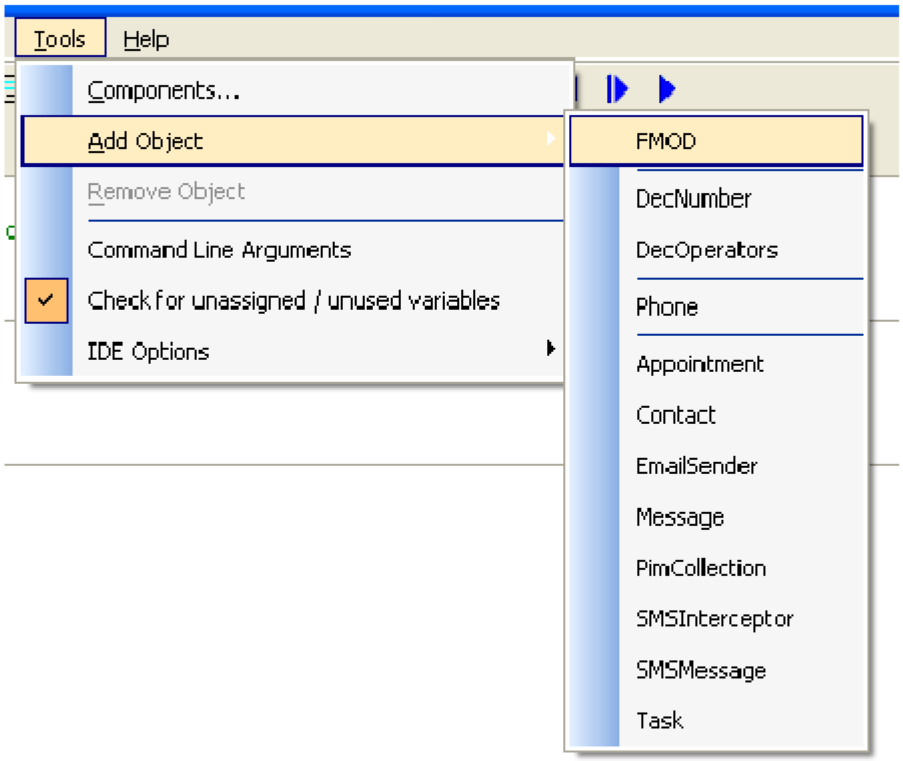
In place of the Dim keyword, there are two ways to create a new object of a library to your project:

1. At design time, using the Tools menu – this is the preferred way.
2. At runtime, using the AddObject keyword.

<u>Adding an object at design time</u>: Use the Tools menu, and select "Add Object", to create a new object from your library. Note, that a library may contain more than one object you can create. For example, If you add these libraries to your project:



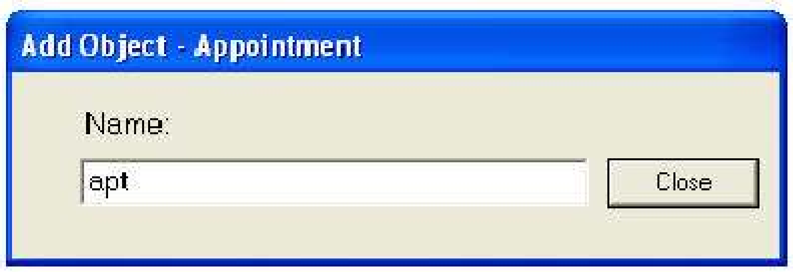
…you will have the choice of much more objects you can create. As seen below, objects in each library appear in the "Add Object" menu, and a line separates each libraries' objects from one another:



Objects that can be added when adding the libraries above to your project.

Use only the objects you need. There is no point in adding too many objects to your program. In order to find out which objects to add to your program, use the help file that is usually associated with the .dll itself, as described above. For more details, read Getting help about a library.

After selecting the object you want to add, you must first name it. A dialog box appears, asking you to name your object:



Name:
apt

An object named “apt” is created from the basic appointment object found in the Outlook.dll file

Adding an object at runtime:

Many times it’s impossible, or impractical, to add all your objects at design time. Although this lets you use the properties list while typing, you may want to create your object only at runtime. Doing so is possible using the AddObject keyword. This keyword replaces in one line the multiple-stages process described above. After adding the .dll file to your project (see step 1), use the AddObject keyword anywhere in your code to create a new object from your added library. The highlighted line, when executed, will cause your program to do exactly the same thing described above:

```
Sub App_Start

AddObject(“tvMain”, “TreeView”)

Form1.Show

End Sub
```

This program starts by adding an object of type “TreeView” to the program and naming it “tvMain”, just like we did using the menus with the Appointment object. At this stage the object was not added to any form.

Actually, at this stage, we cannot yet use our new object. Of course, it was not added to any form, but what’s more important: its NEW Sub was not yet called.

Step 3/3 – call the object’s NEW Sub (all steps)

Consider your new object as the spoon of coffee you took out of the jar: now, you know it’s a coffee, you know you placed the right amount in a spoon, but you yet don’t know where to place it – where is the cup. A new object’s cup is its NEW Sub.

Calling an object’s NEW Sub is the way to allocate a place for it. Actually, this is exactly what happens when you call the NEW Sub: a place in the computer’s memory is allocated for the new object, taking into account all future changes it may undergo.

Apart from allocating place in memory, the object’s NEW Sub may also take some data as parameters. These parameters are used for initialization of some object’s properties. Not all NEW Subs do that.

Calling the object’s NEW Sub is called instantiating (creating an instance). The reason is that when you call the NEW Sub you actually spare memory for the object and name this memory portion. This creates an instance – a copy – of your object. The “object type”, TreeView in the example above, is not changed at all – all actions you will perform from now on will affect only the instance (saved in its **new** place in the memory). This way two goals are achieved:

1. Many instances of the same object can be created, all with the default properties of the object type.

2. Any change to an object instance doesn't affect other instances of the same object, thus allowing you to have, for example, many TreeViews with different properties (that is, different color, different nodes and so on).

Constructor

Every object must have a Sub called NEW (actually, it is called NEW1). The NEW Sub of an object is called constructor. A constructor builds a place in memory and assigning the name to it. Constructors can have further functionality, such as initializing properties to default values.

The following example calls the "tvMain" object's constructor (of type TreeView, created programmatically a line earlier) and supplies some required parameters.

```
Sub App_Start

  AddObject("tvMain", "TreeView")

  tvMain.New1("Form1", 0,0,50,100)

  Form1.Show

End Sub
```

We will focus on the highlighted line again. The "New1" constructor is called. The parameters passed to it describe needed information: to which form to add the new instance, and what size should it be (where to place its corners relative to the form). Once you've called an object, you can start using it. The following example demonstrates how to add two nodes to a TreeView object, called tvMain again:

```
Sub App_Start
```

```
    AddObject("tvMain", "TreeView")

    tvMain.New1("Form1", 0, 0, 200, 200)

    tvMain.AddNewNode("first node")

    tvMain.AddNewNode("second node")

    Form1.Show

End Sub
```

Overloading – more than one constructor

An object can have more than one constructor (that is, more than one NEW Sub). To distinguish between them, each Sub called new has a number attached to it: NEW1, NEW2 and so on. For example, the BinaryFile library has two constructors:

- New1 (ConnectionName As Stream, ASCII As Boolean):

Parameters:

ConnectionName – The name of a file connection opened using FileOpen (must be of random type).

ASCII – If false, strings will be encoded using UTF-8 (Unicode) format, otherwise strings will be encoded using ASCII format.

- New2 (ConnectionName As Stream, CodePage As Int32)

Parameters:

ConnectionName – same as above.

CodePage – a code indicating the desired encoding for the file (a list of encoding is available at Microsoft's site, see Basic4ppc help for the link)

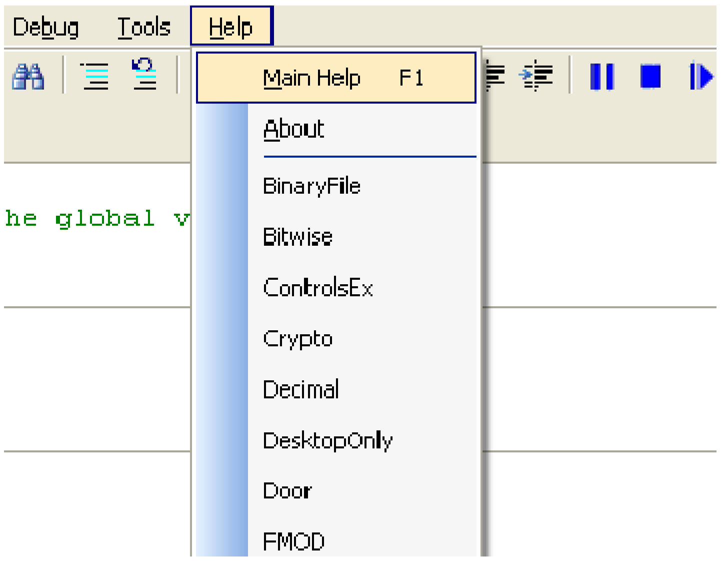
As you can see, both New Subs are similar. Both will create an instance of the BinaryFile object, used to read from and write to files. But they take a different second parameter, this way allows more flexibility.

Getting help with a library

Getting help about a specific object/property

Using a library encounters a new problem: getting help about the objects included in it, their Subs and methods and the way to work with them. This part describes the commons solutions.

The first thing to look for is the help file, usually associated with a library. If you write your own library, make sure you supply decent documentation. Basic4ppc lists all libraries help files in its help menu, when found under the libraries folder:



Orientation in a library's help file

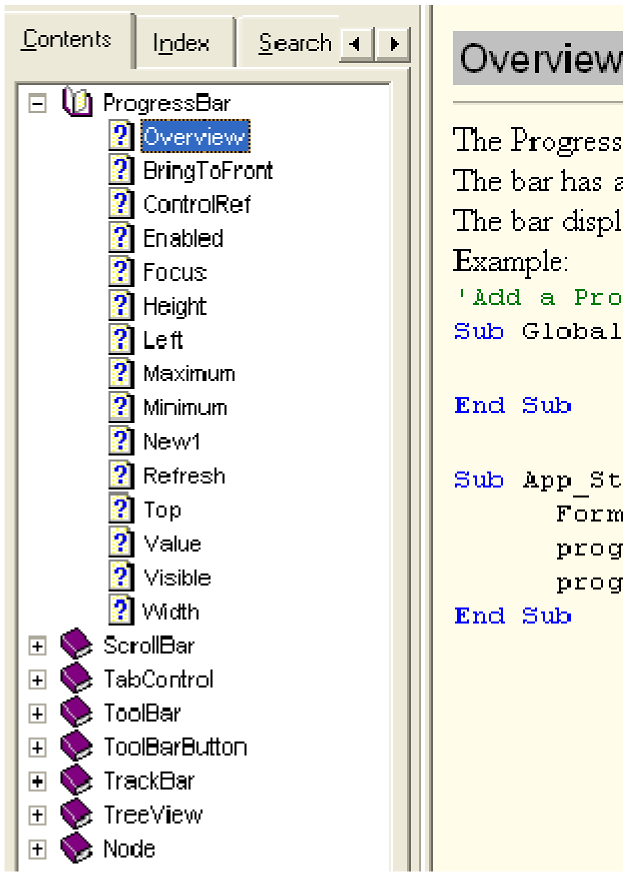
The convention about library's help file declares the following structure:

1. A help chapter is dedicated to each object in the library.

2. The first topic under each chapter is an overview about the object.

3. A help topic is dedicated, separately, for each property, method, Sub and variable under each object.

The following picture illustrates this structure using the ControlEx library.



Overview

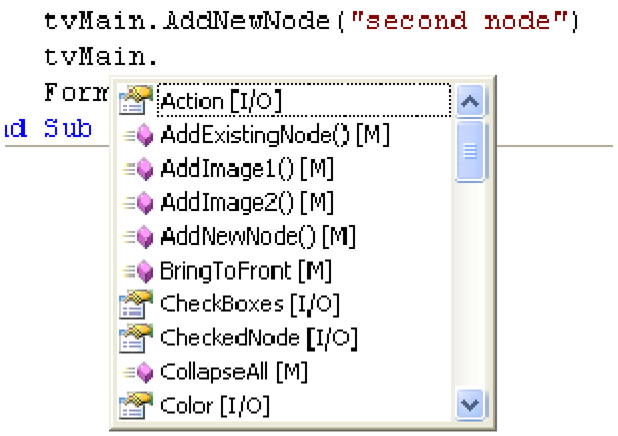
The Progress
The bar has a
The bar displ
Example:

```
'Add a Pro
Sub Global

End Sub

Sub App_St
        Form
        prog
        prog
End Sub
```

- It is a good developing practice to always start by reading the help chapter of any new object you did not yet use. This can save a lot of time dealing with compile-time and runtime errors.

Working with the object you have created

When an object is instantiated, you can start using it just as you would have used any other Basic4ppc component. Note that if you add an object using the AddObject keyword, Basic4ppc will not display a list of properties after you type the new object's name and a period (.). This is a good reason to always use the **Tools** menu – **Add Object** option. Objects added this way will be much easier to deal with, especially when dealing with libraries, that present new objects you may not be familiar with. The following picture shows the AutoComplete feature and demonstrates the help it had supplied when the examples in this reference were built:

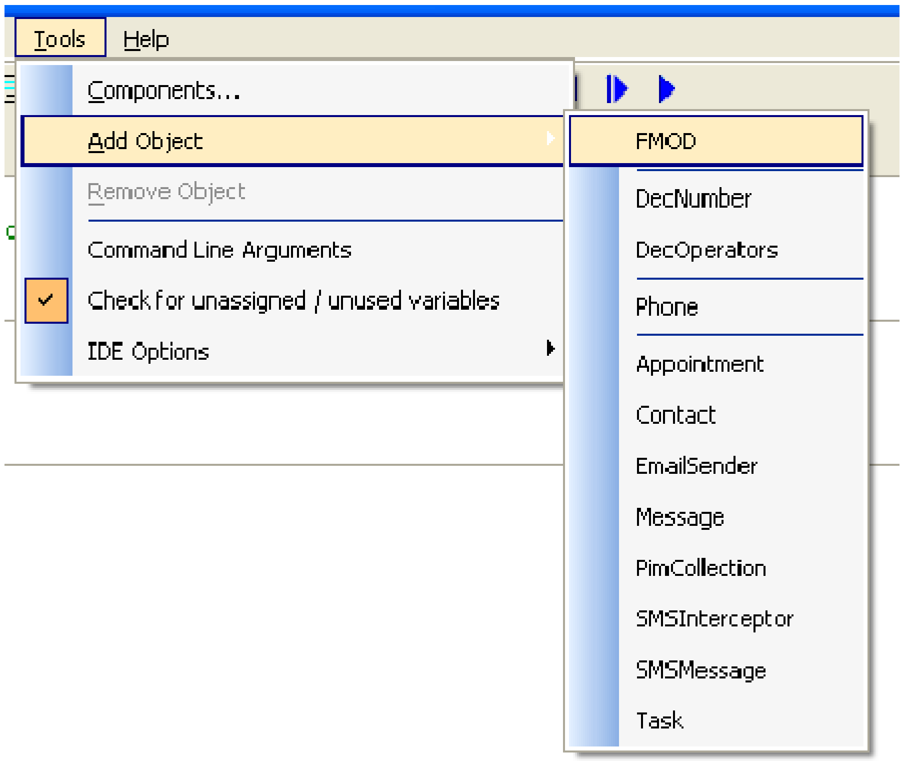


```
tvMain.AddNewNode("second node")
tvMain.
Form
nd Sub
```

```
Action [I/O]
AddExistingNode() [M]
AddImage1() [M]
AddImage2() [M]
AddNewNode() [M]
BringToFront [M]
CheckBoxes [I/O]
CheckedNode [I/O]
CollapseAll [M]
Color [I/O]
```

Finding out which objects are included in a library

The easiest way to find out which objects are included in a library, apart from reading the help file, is looking at the **Tools** menu, under the **Add Object** Sub-menu. After adding a library, you will find there a list of all available objects, as seen in the picture below:


Tools
Help
Components...
Add Object
Remove Object
Command Line Arguments
✓ Check for unassigned / unused variables
IDE Options
FMOD
DecNumber
DecOperators
Phone
Appointment
Contact
EmailSender
Message
PimCollection
SMSInterceptor
SMSMessage
Task

Internal vs. External controls

Using a control added from a library ("External control") is different than using a control internally included in Basic4ppc ("internal control"). There are some minor differences you should be aware of:

With internal controls, when you pass a control **to a Sub in an external library,** that needs a control as a parameter, you specify the controls name. For example (see full example below):

```
tvMain.New1("Form1", 0, 0, 200, 200)
```

When working **with controls added from a library**, Basic4ppc compiler will not recognize the name you gave the control when you added it. When passing such controls as parameters, **use the ControlRef** property found in (almost) every control. This property references the control itself in a way that allows external method to take it as a parameter. Consider this property as a link between Basic4ppc and objects imported from outside world, ad in the following example:

```
Sub App_Start
tvMain.New1("Form1", 100, 100, 200, 200)
tvMain.AddNewNode("first node")
tvMain.AddNewNode("second node")
AddObject("TabControl1", "TabControl")
TabControl1.New1("Form1", 0, 0, 200, 200)
TabControl1.AddTabPage("first page")

TabControl1.AddControl(tvMain.ControlRef, 0, 0, 0)

Form1.Show

End Sub
```

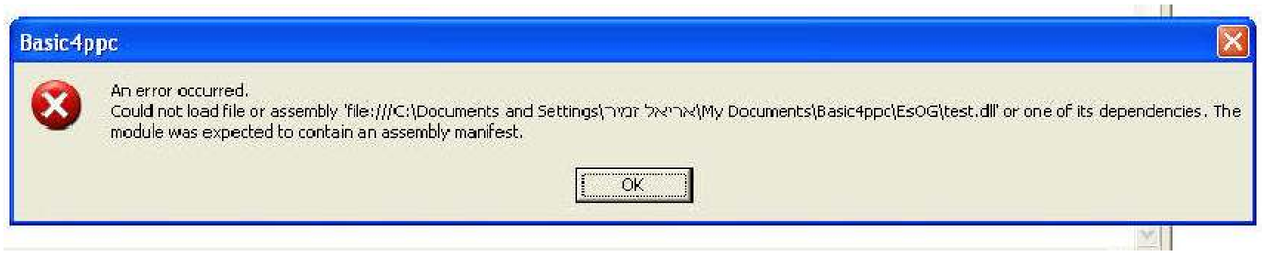
Note, that after adding a TabControl in line 5, we add it to a form called Form1 using the New1 constructor. Form1 is an internal control and can be passed as a parameter to the external control literally, using its name. But both tvMain (that was added using the **Tools** menu) and TabControl1 are external controls. If we wish to add the TreeView tvMain to our TabControl (TabControl1), we cannot pass the literal string "tvMain" as a parameter to the AddControl method. Instead, we use the ControlRef property.

External controls cannot be visually added to the designer, hence their properties must be programmatically set.

Error messages you might encounter

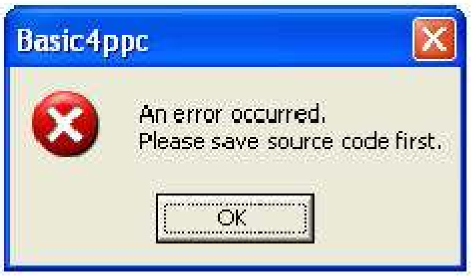
When trying to add a library it is common to encounter some error messages. This part discusses the most common amongst them.

Files could not be loaded, or files are missing assembly manifest: Your files are probably defected. Try re-installing your libraries files, and make sure you are referencing a Basic4ppc dll.


Basic4ppc

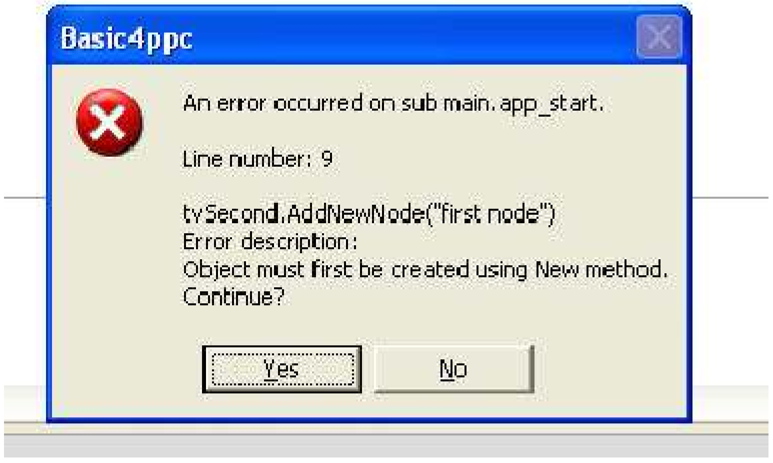
```
An error occurred.
Could not load file or assembly 'file:///C:\Documents and Settings\אריאל זמיר\My Documents\Basic4ppc\EsOG\test.dll' or one of its dependencies. The
module was expected to contain an assembly manifest.
```

Please save source code first: Basic4ppc cannot add a library if your project is not saved.


Basic4ppc

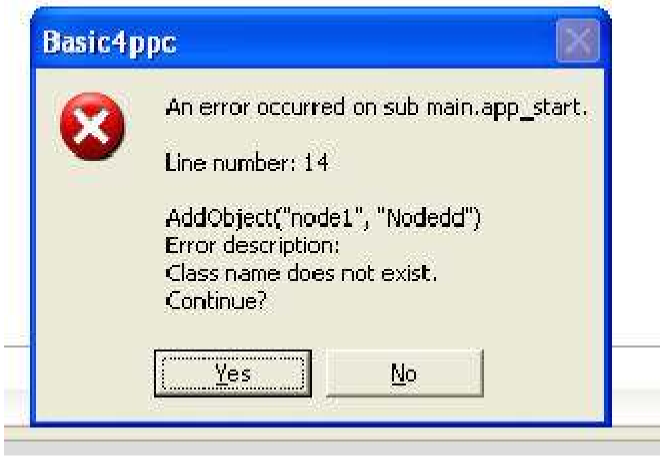
```
An error occurred.
Please save source code first.
```

Object must first be created using New method: You did not use the object's constructor.



```
An error occurred on sub main.app_start.

Line number: 9

tvSecond.AddNewNode("first node")
Error description:
Object must first be created using New method.
Continue?
```

Class name does not exist: You have tried to add an object specifying a non-existing object type (sometimes called "class"). See detailed explanation about how to find out which object types are included in your libraries, and make sure the object's type name you specified is correct. Prefer adding objects from the Tools menu.



```
An error occurred on sub main.app_start.

Line number: 14

AddObject("node1", "Nodedd")
Error description:
Class name does not exist.
Continue?
```

A list of existing libraries

This list is partial. A complete list can be found online at the Basic4ppc forum here (you need to be connected to the internet in order to follow this link)

Libraries supplied with Basic4ppc

BinaryFile – manipulate binary (rather than text) files.

Bitwise – perform bitwise operations.

ControlsEx – extends the amount of control at your aid when developing user interface.

Crypto – data encryption solution.

Decimal – perform very accurate mathematics operations on very big numbers

DesktopOnly – use functionality supported only on the desktop.

Door – allows you to access .Net objects, methods and properties from Basic4ppc code without creating new libraries.

FMOD – enables access to Firelight Technologies FMOD libraries, which support many common audio file formats including MP3, WAV and more.

FormLib – adds support for full screen applications.

FTP – allows access to Internet (or other network) resources using FTP protocol.

GPS – connect to your device's GPS receiver.

Hardware – supports some Device-only functionality.

HTTP – allows access to Internet resources using HTTP protocol

ImageLib – extends support for images and drawings.

Network – allows connection between computers over a network, using TCP protocol.

Outlook – allows access to Microsoft Pocket Outlook data.

Phone – allows using phone functionality if the Device supports it.

RAPI – a desktop-only library, which allows desktop-device connection using ActiveSync.

Regex – allows usage of Regular Expressions.

Registry – allows access to the registry.

Serial – adds support for the serial port.

Sprite – adds support for animation

SQL – allows SQL database access.

Each of these libraries is fully documented.

Deploying an application with libraries

Deploying an application written with libraries is done the same way you deploy every application. An issue to be aware of is the issue of the libraries files (dlls) themselves. As mentioned earlier, dll files should be located the same place as your main application.
Merged libraries will automatically be merged into your main application file. In this case there is only one file to deploy.
Non-Merged libraries should manually be taken care of. Either use SetupBuilder to add them to your .cab file, of otherwise copy them and deploy with your application. You have to make sure you copy them to the same folder from which your application is run later, otherwise the application will fail to run.

Copyright
You should be aware of copyrights applied to the libraries you use. Unless otherwise stated, all Basic4ppc libraries are redistributable, and you may use them freely in your application for any legal purpose (including commercial projects).

Part II - Data management with Basic4ppc

Data management overview

Background

Many modern computer applications involve managing data in various ways. It can be the user settings saved to a simple local text file or a whole database of customers and orders – managing data is a common task.

This chapter gives <u>an overview</u> of how to carry out this task with Basic4ppc. The chapter is not intended to be a complete reference. Some topics are covered thoroughly and some are just references to other sources. However, it aims at giving you a good idea of the tools at hand and where to find more information. Topics covered in this chapter are:

- What do we mean by data management
- Which tools are available in Basic4ppc
- How to choose the tool you need
- Where to find information

What is data management

Data management is the process of saving data to a persistent storage and retrieving it later. It also refers to the management of data in data structures.
There are many levels of data management as mentioned above. It can be though of comprised of the follwing:

- not necessarily persistent: Data Structures
- Basic and simple: Text files
- Strong yet very simple – Table control
- Professional
    - XML
    - SQL
    - Binary files

Basic4ppc offers the following methods of data management:

Data Structures:

Data structures are a basic way to manage data. Use the collections below when you need quick sorting or retrieving of data from a list.

This section is not intended to give background in data structures, but you should know that the following are available:

- Array: a basic structure in Basic4ppc: described in the Arrays chapter.
- ArrayList: internal control added from the designer. Manages a list of elements like an array. Unlike array, allows adding and removing elements in the middle of the list and lets you sort and search in the list.
- Hash table: internal control added from the designer. Lets you save data attached a unique key and search for it by the key.
- Stack: internal control added from the designer. Items are inserted in LIFO order.
- Collections from the Collection library, developed by the most significant community member Agraham in the forum: ArrayEx (Extended), ArrayListEx, HashTableEx, QueueEx, SortedListEx, StackEx.

Files

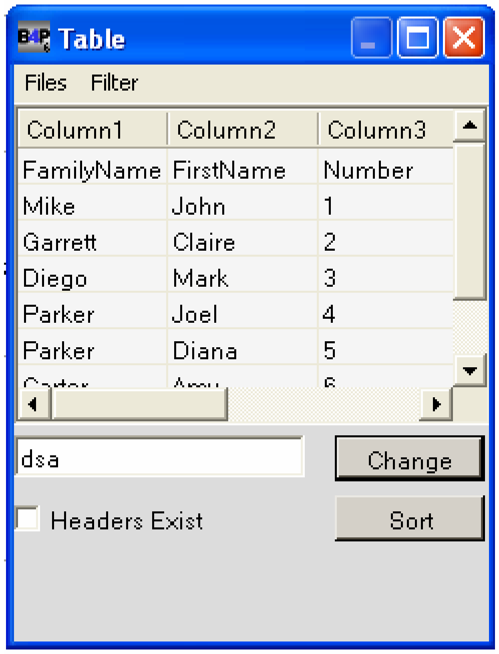
Files are data saved to a disk (or other persistent media). There are two main types of files:

- Text files: described at length in the Text Files chapter.
- Binary Files: usually handled with the BinaryFile Library – a standard Basic4ppc library. See here:

http://www.basic4ppc.com/forum/additional-libraries/3314-dll-version-listings.html

The Table Control

The table control is a very special control. It is used to manage data and the basic appearance is this:



Column1	Column2	Column3
FamilyName	FirstName	Number
Mike	John	1
Garrett	Claire	2
Diego	Mark	3
Parker	Joel	4
Parker	Diana	5

dsa

The Table control is meant to store a set of records (rows) and to allow access to the data, manipulating the data, saving and loading it. Table control is also a grid control and allows showing data in a grid. You can use the Table control with Visible set to false as a data reference or display the data directly on it, as in the picture above.

The Table control holds data in records – rows. Each row consists of several columns. Rows are accessed by their index (starting from 0) and columns are accessed by their names. Each column can store either strings or numbers only.

The control allows you to load XML files, SQLite files (using the DataReader, see the SQLite chapter) and Microsoft Excel CSV files. It also allows sorting, filtering (with SQL expressions) and performing such table related operations.

Learning to work with the table control

At the moment there are few resources for this important control. The best place to start is to read this forum thread entirely:

http://www.basic4ppc.com/forum/questions-help-needed/5367-table-control.html

and then:

- Information about the Table control can be found in the Help file under Table and under database.
- Under the Basic4ppc Samples folder is a sample program named "Table.sbp".
- Search the forum for Table Control.

Databases

Basic4ppc has libraries for the following common databases server:

a. SQLite library – see the next chapter for a full tutorial.
b. MSSQL library, developed by community member citywest:
http://www.basic4ppc.com/forum/additional-libraries/3082-ms-sql-library.html

Excel files

You can read Microsoft Excel files directly to the Table Control (ReadCSV method).

XML

XML files are handled with Agraham's XML library here:

http://www.basic4ppc.com/forum/additional-libraries/3385-xml-library.html

Text Files and basic files/folders manipulation

Manipulating files is one of the most basic tasks carried out when programming. This chapter covers the topic of dealing with text files, considered the most basic files handling there is. First, let's start by setting the fundamentals and the terminology.

A file

A file is a collection of data stored on a persistent storing device, such as a disk, a flash card or any other kind of media. This is the basic, classic definition: nowadays, borders between files and other kinds of data agglomerations are become indistinct, mainly because of the rapid progress in the storage media types and volumes and in the network's ability to store and transfer data. So if we try to declare what file is ideally, we could have said that it is "a bunch" of data – usually stored in a way that's meaningful to someone, that can be used as a long "stream" of bytes (the basic memory units). The difference between this "stream" and regular data located in memory is that only what is located properly in memory by the operating system can be accessed by the means of variables as we are used to, while the stream of bytes, though holds the same value, is meaningless until somehow parsed.

Consider the following scenario: you have three variables, stored in memory, representing the age of the user, his name and his date of birth. The age is and date take each 4 bytes of memory and the name is a string of 20 bytes. So your memory might look like this (in the upper line, each cell represents one byte in the computer's memory):

0	0	3	0	J	O	H	N	S	M	I	T	H												2	3	4	4	F	F
Address 0000				**Address 0004**																				**Address 0024**				**Addr. 0028**	

Variable name: Age	Variable name: Name	Variable name: DateB	Not used

Schematic memory dump

Your memory is basically just the uppermost line, but the operating system holds a mapping of variables and addresses that give meaning to the memory stream. If you store the same variable structure to a file, your file will have this structure:

0	0	3	0	J	O	H	N	S	M	I	T	H												2	3	4	4	F	F

With no additional information. Regardless of the question if you plan to store it to a disk, send it over the internet or just keep it in your computer's memory, this byte stream is just a list of bytes.

Text vs. Binary

Traditionally, software developers distinguished two file types from one another: Text files and binary files. They both represented a means of saving chunks of memory to the disk, but whereas binary files where just the memory chunk itself, text files where a textual representation of the data – usually meaning the programmer had to convert everything to strings before saving, losing efficiency (data is bigger), adding programming overhead (converting), but gaining the advantage that humans could have read the file (and some other advantages). So, instead of saving the date of a something as a number of 4 bytes, you would have converted it to a string such as "01/01/1900". Each character in a string consumes two bytes (in the common Unicode encoding) resulting in a 20 bytes consumption where only 4 where needed earlier. On the other hand, if speed wasn't that important and storage volume is no limit, this file can be opened with every notepad-like program instantly, edited easily and in general is much easier to handle during debugging, development and software lifetime cycle in general. On the other hand, reading data from both text and binary files requires some parsing – usually a bit simpler with text files, especially for beginners or in small to medium programs. Windows itself used to save initialization data for programs in text files with the extension .ini, a method later abandoned for the registry system. Nowadays a common usage of text file is XML files, basically text (for human readability) with special notation system (tagging) that allows software to read the data without having to know in advance what the structure is. There are some other characteristics that are described below and differs text from binary files. For example, binary files have the big advantage of being "random access" – you can start reading them in the middle, while text files force you to read from the beginning to reach where you need.

Other file types

During the years many standards and formats have become popular. The word "format" here actually says there is a special structure for each file type: for instance, a file with the extension .jpg is supposed to comply with the standard for saving jpeg images, which means it follows certain rules of what comes after what and when: say, first 4 bytes are version number, next 2 bytes are date, next 2 bytes are empty and so on and so forth. This is called a file format. Thousands of formats exist, each with its purpose. Most of them are using binary files, but some, such as HTML and XML, use text files (Basic4ppc itself saves the designer data and code of a module in a text file with a special format, that is private yet human readable).

Some of the major standards have become so popular that there are functions that read them instantly. For example, Basic4ppc lets you read and write the Microsoft's Excel .CSV files instantly into the Table control, or the SQLite file format into the same control. Additional libraries of high importance let you easily read and write SQL data of different providers (MySQL, MSSQL, SQLite), XML, JPEG and more – you do not really have to parse them yourself, a tedious progress that's error prone and, mainly, needless.

Text files

Yet many times you want to be able to save your own data to a file without too much hassle. You just need to put in a couple of lines, maybe the names of the photos you took with your cell phone or the dates of coming events. The simplest way to do it is using text files. Many applications use text files to store textual data, store user settings and so on. It is pretty simple to work with text files, but still there are some points that you should be aware of.

Sample program

This chapter creates an example: a small application with some user data. The user data will be loaded when the application starts (Sub App_Start) and saved when the application ends (Sub Form1_Close). It is based on the forum tutorial created by Erel and can be found in the forum, though it is more comprehensive than the tutorial.

Before we start – basic concepts and methods

A text file is a file that can be opened in notepad. It stores text and this means that lines are separated with the line-break character. This convention allows programs that expect text to know where to break to a new line.

Opening and closing files: A file in your system, apart from being the stream of bytes described earlier, is also a resource supplied by the operating system to access this bytes stream. These resources are limited and managed – for example, the operating system will not allow two programs to access the same file at the same time, because one of them might change data the other one needs. For this reason, each file is either opened or closed at a given point – and if it's opened by your program, any other program that tries to access it fails. On the other hand, if you left it opened, you deny access for anyone else: not very respectful.

FileOpen

To open a file, use the **FileOpen** keyword.

Syntax:

```
FileOpen (ConnectionName, FileName, AccessType (see below), [, cAppend [, cASCII])
```

Parameters:

- ConnectionName: a variable representing the file – think of it as a special kind of variable. You can't assign anything else into it later, but it will be used as the connection "handle" to your file – when you need to read or write to the file you specify this name as the target.
- FileName: A string representing the file name to open. If you do not specify a path the system assumes you meant the folder from which the program is run.
- Access type: must be one of the following: cRead, cWrite, cRandom. These are constants pre-defined in Basic4ppc, and define the way the system uses the file:
    - cRead: you can only read from this file, starting from byte number 0 and reading each byte at a time. You cannot jump to somewhere in the middle.
    - cWrite: you can only write to this file, always appending to its end.
    - cRandom: you can do both, accessing any point in the file randomly by its byte number. This can be applied only to binary files.
    - **Note**: text files use only cRead and cWrite.

- cAppend (optional): use only when you use cWrite, and you do not want to overwrite and existing file. If the file exists already and you did not use cAppend, the file is overwritten when opened (practically deleted of its content and left empty). If you did use cAppend, the file is opened and any data you write to it is added to its end.

  Note: it's a Basic4ppc quirk that you should leave an empty place for cAppend if you use cAscii after it without cAppend. For example:

  ```
  FileOpen (c1,"Data.txt",cWrite, , cASCII)
  ```
- cASCII (optional): set file encoding to ASCII rather than Unicode (default). Use this encoding if you plan to work with software that does no support Unicode. See encoding below.

Example:

```
FileOpen ( c1,"TextFiles.ini", cWrite)
```

FileClose

When you are done with your file, it is important to close the connection to it in order to release the file lock (let others use it and free resources). FileClose does this. The syntax is very simple:

```
FileClose (ConnectionName)
```

Where ConnectionName is the same variable you set as a name to access your file in the FileOpen command. Note that only one connection is permitted per file, even inside the same application.

Example:

```
FileClose (c1)
```

Writing and reading – FileWrite

Text files are usually written and read a line at a time in a raw. **FileWrite** writes a text string to a file with the following syntax:

FileWrite (ConnectionName, Text)

Where:

- ConnectionName is the one used with the FileOpen command.
- Text is the text to write to the next line in the file. If the text has more than one line (suppose a multiline textbox), then they are all written to the file.

Remarks:

If the file is not open, an error will occur.

Example:

```
FileWrite (c1, txtFirstName.Text)
FileWrite (c1, txtLastName.Text)
FileWrite (c1, chkValidAccount.Checked) 'writing a Boolean value.
```

FileRead

Reads and returns the next line from a file. Syntax:

FileRead (ConnectionName)

Where ConnectionName is the one used with the FileOpen command. Note, that this command returns a value, so it should be used like this (if c1 is the connection name):

r = FileRead (c1)

Now, r holds the next line in the file, and an imaginary "pointer" that pointed to this line is advanced and now points to the next line. The next time you read from the file with the FileRead command, the next line is returned and so on. When you reach the end of the file

(the pointer points to a place after the last line), you get the End Of File constant **cEOF** returned by the command. See the example below to get a further understanding of how to use this command exactly.
Note: If the file is not open, an error will occur.

Example:
(Assuming you have an opened connection named c1, opened as cRead)

```
1    r = FileRead (c1)
2    Do Until r = EOF
3            sum = sum + r
4            r = FileRead (c1)
5    Loop
6    Msgbox (sum)
7    FileClose(c1)
```

In line 1, we read the next line. If the next line is the end of the file, the cEOF constant is returned. We check this in line 2, and if this is what we have we do not enter the loop. In line three we count the loop iterations: this is the number of lines read. In line 4 we read the next line, and in line 5 we jump back to line 2, checking again if End Of File was reached. If so, we jump to line 6 to show the counter: otherwise, we go on with the same process. This is the right way to read data in sequentially from a text file – the process inside the loop may vary according to your needs, but this is the basic structure.
Note: line 7 closes the file…

FileReadToEnd
If, for some reason, you need to read the remaining data from a file, you can use this command. Usually text files are handled a line at a time, but for some purposes (you entered a big chunk or you are just not interested in processing data) this could spare time.
Syntax:

FileReadToEnd (ConnectionName)

Example:

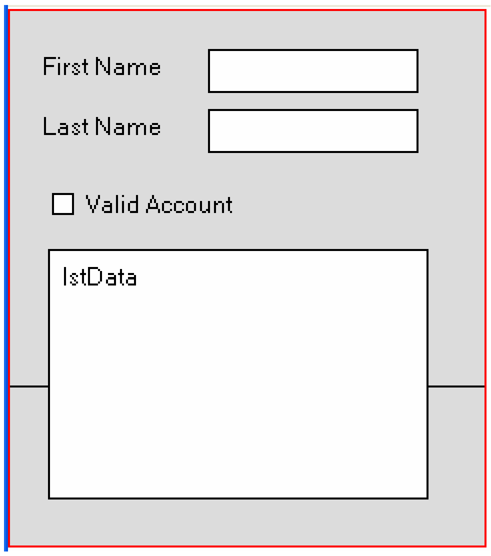
R = FileReadToEnd (c1)

FileExist

It is a good habit to make sure a file is there before you try to open it. File operation is error prone, and trying to access as non existing file is one of the most common things that might cause your application to crash. The FileExist function returns True is a file exists and False otherwise.

Sample

The sample below demonstrates how to build a small program that opens a text file, writes and read data to and from it and closes it. The text file data itself is display below the source code. This program requires that you have a form that looks like this in the designer (note that the complete sample, with the form and the text file can be downloaded here: http://www.basic4ppc.com/forum/tutorials/943-text-files.html)



Controls names are:

- Upper textbox is txtFirstName (it is important so that to be consistent with the code below)
- Second textbox is txtLastName.
- Checkbox is chkValidAccount.
- A listbox named lstData.
- Form name is Form1.

Code:

```
Sub Globals
        ‘Declare the global variables here.

End Sub

Sub App_Start
        Form1.Show
        LoadINI
End Sub

Sub SaveINI
        ErrorLabel(errSaveINI)
        FileOpen( c1,”TextFiles.ini”,cWrite)
        FileWrite(c1,txtFirstName.Text)
        FileWrite(c1,txtLastName.Text)
        FileWrite (c1, chkValidAccount.Checked) ‘writing a Boolean value.
        For i = 0 To lstData.Count-1
                FileWrite(c1,lstData.Item(i))
        Next
        FileClose(c1)
        Return ‘If there were no “Return” here, the error message would have shown.
        errSaveINI:
        Msgbox(“Error writing INI file.”,””,cMsgboxOK,cMsgboxHand)
        FileClose(c1)
End Sub

Sub LoadINI
        ErrorLabel(errLoadINI)
        If Not(FileExist(“TextFiles.ini”)) Then Return
```

```
            FileOpen(c2,"TextFiles.ini",cRead)
            txtFirstName.Text = FileRead(c2)
            txtLastName.Text = FileRead(c2)
            chkValidAccount.Checked = FileRead(c2)
            s = FileRead(c2)
            Do Until s = EOF 'Read the ListBox items.
                        lstData.Add(s)
                        s = FileRead(c2)
            Loop
            FileClose(c2)
            Return
            errLoadINI:
            Msgbox("Error reading INI file.","",cMsgboxOK,cMsgboxHand)
            FileClose(c2)
End Sub

Sub Form1_Close
            SaveINI
End Sub
```

The text file attached contains the following lines:

```
Mike
Rogers
 FALSE
Item 1
Item 2
Item 3
ABC
DEF
```

Explanation – step by step

```
Sub Globals
            'Declare the global variables here.

End Sub

Sub App_Start
            Form1.Show
            LoadINI
End Sub
```

Lines 1 – 4 are Sub Globals.

Line 7 displays the form.

Line 8 calls the LoadINI Sub (reading the data from the file. Note, that though it appears in code after the SaveINI (saving data to the file) Sub, I prefer discussing it the order the calls are made):

```
27  Sub LoadINI
28          ErrorLabel(errLoadINI)
29          If Not(FileExist(“TextFiles.ini”)) Then Return
30          FileOpen(c2,”TextFiles.ini”,cRead)
31          txtFirstName.Text = FileRead(c2)
32          txtLastName.Text = FileRead(c2)
33          chkValidAccount.Checked = FileRead(c2)
34          s = FileRead(c2)
35          Do Until s = EOF ‘Read the ListBox items.
36                  lstData.Add(s)
37                  s = FileRead(c2)
38          Loop
39          FileClose(c2)
40          Return
41          errLoadINI:
42          Msgbox(“Error reading INI file.”,””,cMsgboxOK,cMsgboxHand)
43          FileClose(c2)
44  End Sub
```

Sub LoadINI is the interesting part:

- Line 28: File operations could fail – for example, if the file is used by another application.
  We are using **ErrorLabel** to handle unexpected errors and show a message to the user (see the error handling chapter).
- Line 29: Check if there is a file to open. If there isn’t, exit the Sub.
- Line 30: we open the file with **FileOpen**. The first parameter is the name that we give to this connection. The name c2 is chosen here just to prevent confusion with the SaveINI Sub – this could have been the same name. The

second parameter is the path and file name: In this case the file is located in the same folder of the source code (or compiled executable). The third parameter is cRead – we open it for reading only. The optional cASCII is omitted here.

- The settings file is built of several known fields and an unknown number of fields as the ListBox items. **FileRead** reads a single line from the connection: a line at a time.
- Lines 31, 32, 33: In the same order we previously saved the data, we now read the first three known lines.
- Line 34, 35, 36, 37, 38: the remaining data is now fetched and a new item is added to the ListBox for each line. The same technique shown before is used. Each line is compared to the **EOF** constant which symbols the End Of File. When the value equals to EOF we know that there are no more items left (note: another approach could have been to write the number of items before the items on a separate lines and use a For loop).
- Line 39: close the file.
- Line 40: exit the Sub to prevent from the error message from showing.
- Line 41: handle errors: display error message.
- Line 43: close the connection in case of an error as well!

When you exit the program, the data is saved to the file. Lines 46 – 49 are the Form1_Close event. It simply calls the SaveINI Sub.

```
46  Sub Form1_Close
47          SaveINI
48  End Sub
```

The SaveINI Sub is where data is saved to the disk:

```
11  Sub SaveINI
12          ErrorLabel(errSaveINI)
13          FileOpen( c1,"TextFiles.ini",cWrite)
14          FileWrite(c1,txtFirstName.Text)
15          FileWrite(c1,txtLastName.Text)
16          FileWrite (c1, chkValidAccount.Checked) 'writing a Boolean value.
17          For i = 0 To lstData.Count-1
18                  FileWrite(c1,lstData.Item(i))
19          Next
20          FileClose(c1)
            Return 'If there were no "Return" here, the error message would have
21          shown.
22          errSaveINI:
23          Msgbox("Error writing INI file.","",cMsgboxOK,cMsgboxHand)
24          FileClose(c1)
25  End Sub
```

- Line 12: Again, since file operations could fail, we use ErrorLabel here.
- Line 13: Open the c1 connection (name is arbitrary: could have been anything) for writing.
- Lines 14, 15, 16: Writing strings (14, 15) and then a Boolean value to the file.
- Lines 17, 18, 19 – writing the lines of the listbox called lstData to the file. The listbox control was not discussed in this guide, but using the item(n) property returns the line number n (zero based numbering) as a string.
- Line 20: close the connection.
- Line 21: avoid reaching the error label.
- Lines 22, 23, 24: handle errors that might occur. Beware to close the file in this case either.

Encoding

The default encoding for text files is UTF8. This allows you to read or write any Unicode character into them. Instead, You could choose to use ASCII encoding. ASCII encoding supports only the lower values of the ASCII table (0-127). Note that UTF8 encoded files will start with a special Unicode marking. Most applications recognize this marking and do not show it. However, some older applications will show something like ";fe" at the beginning of the text. In such cases use ASCII encoding instead. If you write a DOS batch file (on the desktop) then you should use ASCII encoding for that reason.

Other file-related commands

Basic4ppc offers some other commands for file handling. Others are available using external libraries. The next sections cover some of the important functions you should know and where to find them. There are more and many others are created.

Files

- **FileReadToEnd** reads the entire remaining data (unlike FileRead which only reads the next single line) and returns a string with it.
- **AppPath** returns a string with the path of the current application. This is useful when you need to refer to an external program or when you want to know or to show the user where your program is run from.
- **FileCopy** (<SourceName>, <TargetName>, [<Overwrite>]) copies a file from the source location to the target name and location. If overwrite (optional Boolean) is set to true, overwrites the target.
- **FileDel** (<name>) deletes a file.
- **FileName** (<file_with_path>) gets a string of a file and a path and returns the filename itself.
- **FileDirName** (<file_with_path>), on the other hand, gets a filename string and returns the full name of the folder.
- **FileLength** (<filename>) gets a string with the file's name and return the length in bytes.

Folders

- **DirCreate** (<name>) creates a new folder with the name you specify for <name>if it isn't yet there. If it is, the command is ignored. Unless an explicit path is specified, path is relative to the application path (see AppPath).
- **DirDel** (<name>, [<IncludeFiles>]) deletes a folder. IncludeFiles is optional, Boolean value with default (if omitted) of False). If set to true, the folder is deleted even if there are files in it.
- **DirExists** (<name>) returns true if the directory you asked for exists.

FilesEx library – extended files manipulation

Note: as all external community-created libraries, available for registered users only, download from the forum (use the dll listing in the ling below to find the last version). You should consult the first page of the help file for some details regarding this library:
http://www.basic4ppc.com/forum/additional-libraries/3314-dll-version-listings.html

The FileEx (Extended) library, written by **Agraham**, supplies the following functionality:

- **DirectoryStringInfo**(Path As String) : Returns a string array of length 4 containing information about the directory specified by Path. Index 0 is the directory attributes, 1 is the creation time and date, 2 is the last access time and date, 3 is the last write time and date.
- **DirectoryDoubleInfo**(Path As String) : Returns a double array of length 4 containing information about the directory specified by Path. Index 0 is the directory attributes as a bit pattern, 1 is the creation time and date in ticks, 2

is the last access time and date in ticks, 3 is the last write time and date in ticks.

- **DirectoryCopy** (SrcPath As String, DestPAth As String): SrcPath is the source directory. DestPath is the name and path to which to copy this directory. The destination can be an existing directory to which you want to add this directory as a Subdirectory. Returns false if SrcPath is not valid otherwise returns true.
- **DirectoryCopyTree** (SrcPath As String, DestPAth As String): SrcPath is the source directory. DestPath is the name and path to which to copy this directory and its' Sub-directories. The destination can be an existing directory to which you want to add this directory as a Subdirectory. Returns false if SrcPath is not valid otherwise returns true.
- **DirectoryMoveTree** (SrcPath As String, DestPAth As String): SrcPath is the source directory. DestPath is the name and path to which to move the contents of this directory and its' Subdirectories. Note that the original directory remains empty. The destination cannot be another disk volume or a directory with the identical name. It can be an existing directory to which you want to add this directory as a Subdirectory. Returns false if SrcPath is not valid otherwise returns true. This method throws an IOException if, for example, you try to move c:\mydir to c:\public, and c:\public already exists. You must specify "c:\public\mydir" as the destDirName parameter, or specify a new directory name such as "c:\newdir".
- **DirectoryRename** (SrcPath As String, NewName As String): SrcPath is the source directory. NewName is the name for this directory. The destination cannot be another disk volume or a directory with the identical name. Returns false if SrcPath is not valid otherwise returns true. As there is no

real Rename function in .NET this is actually a **DirectoryMoveTree** to the old path with a new name.

- **FileStringInfo** (FilePath As String): Returns a string array of length 5 containing information about the file specified by FilePath. Index 0 is the file attributes, 1 is the file length, 2 is the creation time and date, 3is the last access time and date, 4 is the last write time and date.
- **FileDoubleInfo** (FilePath As String): Returns a double array of length 5 containing information about the file specified by FilePath. Index 0 is the file attributes as a bit pattern, 1 is the file length, 2 is the creation time and date in ticks, 3 is the last access time and date in ticks, 4 is the last write time and date in ticks.
- **FileGetAttributes** (FilePath As String): Returns an Int32 containing the attributes of the specified file as a bit pattern.
- **FileSetAttributes** (FilePath As String, Attributes As Int32): Sets the attributes of the given file. FileGetAttributes or FileDoubleInfo should be used to get the existing file attributes and that information modified to set or clear the required attributes.
- **FileMove** (FilePathOld As String, FilePathNew As String): Moves the file at FilePathOld to FilePathNew. FilePathNew must include the name of the file which may be different from the original. Returns false if FilePathOld is not valid otherwise returns true.
- **FileRename** (FilePathOld As String, FileNameNew As String): Renames the file at FilePathOld to FileNameNew. Returns false if FilePathOld is not valid otherwise returns true. As there is no real Rename function in .NET this is actually a FileMove within the same directory.

- **GetDirectories** (Path As String, Pattern as String): Returns a string array containing the names of all directories within the directory specified by Path and matching the search pattern.
- **GetFiles** (Path As String, Pattern as String): Returns a string array containing the names of all files within the directory specified by Path and matching the search pattern.
- **DllVersion**: Double [Read only property]: Returns the version number of the library.

Drive (DriveInfo library)

Note: as all external community-created libraries, available for registered users only, download from the forum (use the dll listing in the ling below to find the last version). You should consult the first page of the help file for some details regarding this library: http://www.basic4ppc.com/forum/additional-libraries/3314-dll-version-listings.html

The DriveInfo external library, written, as so many others, by **Agraham**, supplies the following methods (should be added as and external library. Description is taken from the help file included with the library):

- **GetAvailable** (volumename As String): Returns the number of bytes of free storage available to the current user on the specified volume. This method works on both desktop and device.
- **GetFree** (volumename As String): Returns the total number of bytes of free storage on the specified volume. This method works on both desktop and device.

- **GetSize** (volumename As String): Returns the storage capacity of the specified volume in bytes. This method works on both desktop and device.
- GetStorageCardNames: Returns an array containing the names of the storage cards present on a device. These names may be used with the other GetXxx methods. On the desktop this returns an empty array.
- **Dllversion**: Double [Read-only property]: Gets the version number of this library.
- **Valid:** Boolean [I]: Gets a value indicating whether the result returned by the last GetXXX methods is valid or not.

More Sample code

Additional samples can be found in the Basic4ppc Sample programs (under program files/Anywhere software/Basic4ppc by default). Especially note the **FolderChooser** program (working with folders) and the **Album** program that searches for photos in a folder you specify and displays them.

Deprecated file functions

FileGet, FileGetByte, FilePut and FilePutByte are obsolete functions used to be the way to handle binary files. Binary files are now handled solely by the BinaryFile library.

Basic4ppc and SQLite

This chapter is based on a tutorial with the same name from Basci4ppc.com

Contents for this chapter

Introduction

download sample code

This chapter is an article, or a tutorial that covers the basics of using SQLite open-source database engine with Basic4ppc. It focuses on simple scenarios and is accompanied by a full sample code, built gradually along the tutorial.

Goals, Scope and Examples in this article

The goal of this article is to give a new user the tools to build, from scratch, a data-driven application with Basic4ppc and SQLite. It covers a simple scenario (in order to focus on the fundamentals of SQL with Basic4ppc). It supplies references to external resources for a complete learning process. Although it does not cover everything, following all links supplied will lead you through all knowledge resources needed for a complete understanding of SQL, SQLite and data driven application development with Basic4ppc.

What is needed to use this tutorial:

- **Basic4ppc** can be downloaded here.
- **Links** in this tutorial may refer to resources **on the web**. If you read this article using the offline help you might not be able to follow all links unless you are connected to the internet. However, some other links link to places within this article.
- **A .net library named System.Data.SQLite** that can be downloaded at http://sqlite.phxsoftware.com.
- **Basic4ppc SQLite libraries** (.dlls) are distributed with Basic4ppc.
- **SQLite Browser** is optional. One such browser can be downloaded here. We will use screenshots from this browser in the manual demonstration part of this reference. A detailed example of the browser usage is found there.

- **.Net Compact Framework (CF) 2.0** should be installed in order to use the most recent version of SQLite libraries. See the Basic4ppc requirements here.

Examples accompany this reference, combining together the fully-functional program that manages a small restaurant's order taking system. Each part of the reference discusses different part of the program hence a partial code is attached. The design of the sample is described in details here. The complete source code is downloadable from the Basic4ppc website, see link above.

What is SQL

SQL (Structured Query Language) is a programming language designed for the management of data in a relational database management system (RDBMS). Many database systems support SQL, and it is the most common language in use in business database systems.

SQL syntax and logic are simple and easy to understand. Yet they exceed the scope of this article. Many free tutorials exist on the web. One such tutorial is at the w3school site.

What is SQLite

SQLite is an open source library that implements a self-contained, serverless, zero-configuration, transactionalSQL database engine. SQLite homepage supplies all information needed for a complete understanding of the concepts and the extent of usage worldwide. SQLite documentation will supply you with extensive knowledge, yet will not explain how to build, from scratch, an data-driven application with Basic4ppc.

How to use this reference

- Read through the Table of Contents and see if you are familiar with any of them.

- Download the source code attached.
- Skip what you know.
- Read the rest. Run the source code in debug mode.

Getting started with SQLite

This part is for users new to SQLite, or to SQL at all. It covers in details visually building database and a table. This helps for developers never done this before.

Browser

Although SQLite databases are usually created programmatically, it helps to have some SQL-browser-usage experience. If you are new to the concept of SQL databases, it may be helpful to create and modify a bit some databases manually. To do that, download (or use any) SQLite browser, such as SQLite database browser downloaded for free here. Samples and screenshots in this part are taken using this browser. (New to SQL?)

Basic4ppc SQL viewer

As a good source code example of the Basic4ppc usage, you may also download Basic4ppc SQL viewer, written in Basic4ppc. This tool shows the contents of SQLite database and is suitable for the device.

SQLite datatypes

Although not intended to cover all syntax and functions of SQLite, this reference will not be complete without mentioning the basic SQLite datatypes. The method of assigning types to data is very dynamic and flexible, thus being ideal for Basic4ppc. Basically you can place any data type to any column. Nevertheless, the following types can be declared along with the column name when creating the table:

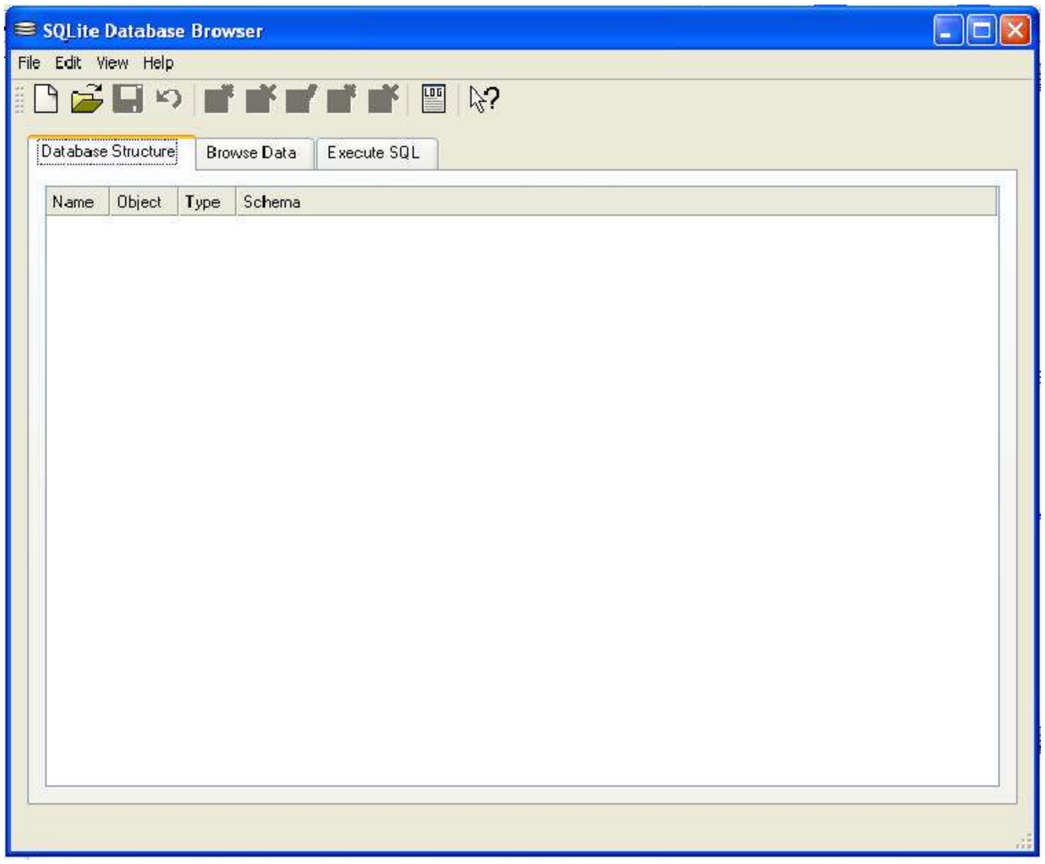
- TEXT – textual data.
- NUMERIC – numbers that may include decimal point. The data is stored in the most suitable way.
- INTEGER – holds integers.
- REAL – real numbers with decimal point.
- NONE – data is kept exactly as inserted.
- BLOB – although not really a data type, BLOB is a unique way of storing data without knowing what it contains. A data stored as BLOB is stored exactly as it is read – as a set of bytes. This strong type is usually used to store images, music and streams of bytes.

SQLite type conversion system is a thing you may want to learn more about here.

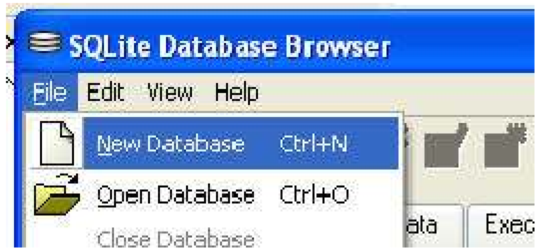
Create a Database Manually

It is a good idea to get to know SQLite by manually creating a database.

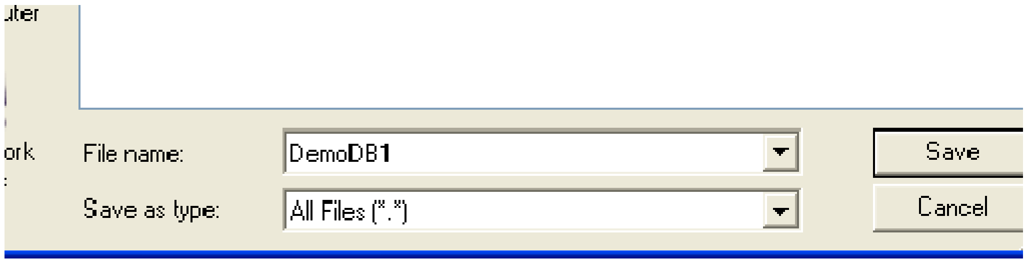
Open the browser. This screen appears:



Choose New Database from the File menu.



SQLite databases are kept in a single file. This simplifies portability a lot, making it comfortable for use with portable devices. Name your file "DemoDB1" and click Save.



The “Create Table” dialog box appears. A database must contain at least one table to have any meaning. We will create a quick phonebook table example. Name your table t_phoneBook as shown below. We will next add some fields to the table.

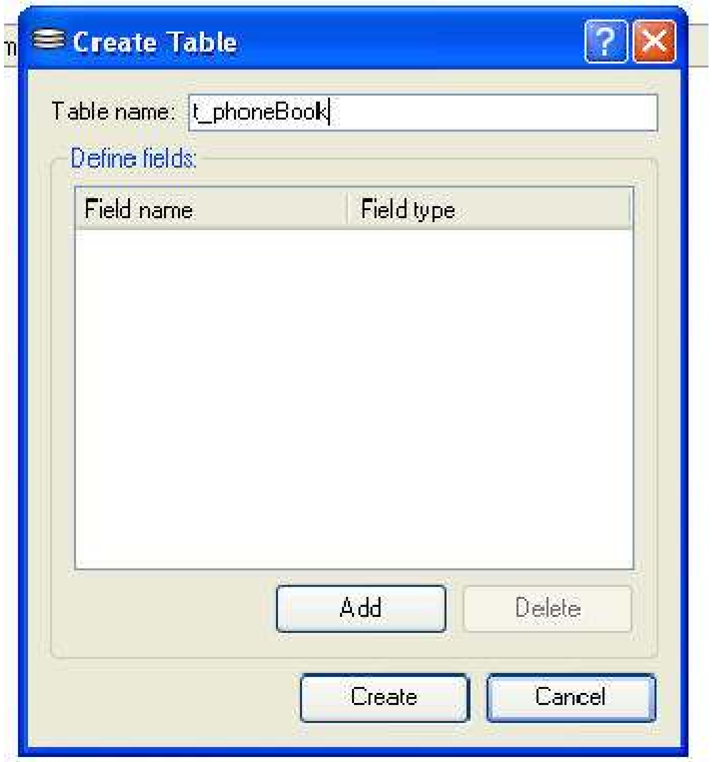
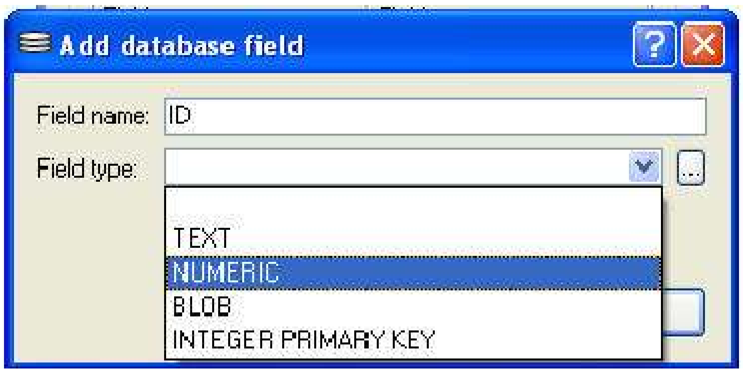


Table name: t_phoneBook

Now click the Add button. The “Add database field” dialog appears. Name the field ID and choose Numeric as the field type. This will be used as a unique ID for each record in the database.



Field name: ID

Click Create. Continue adding the fields Name and Number, until you've added the fields as below:

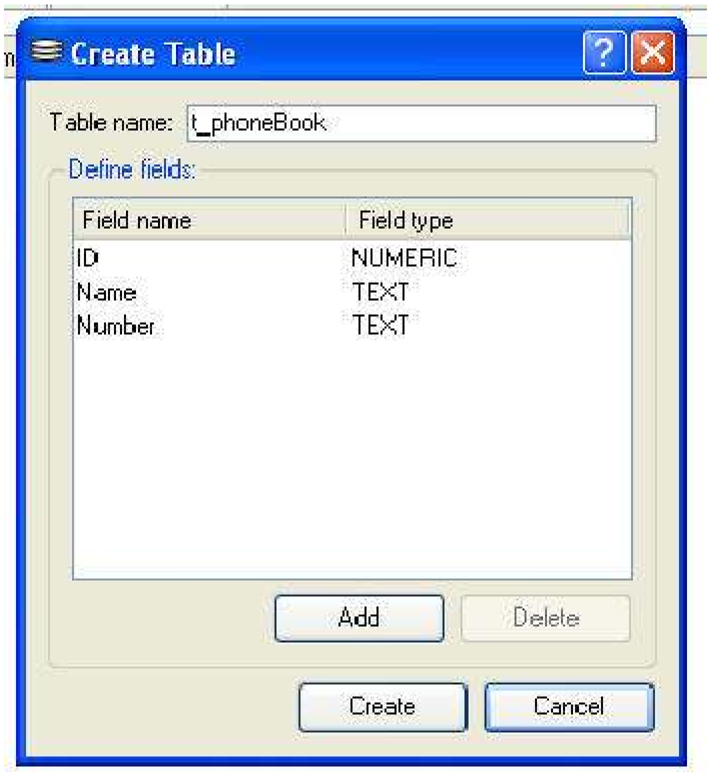
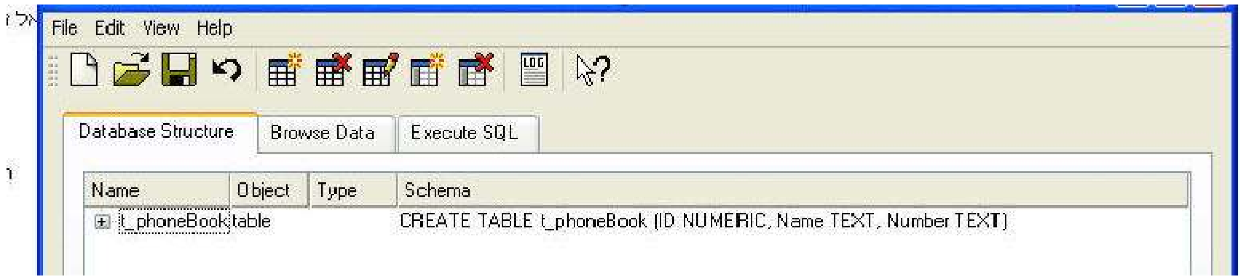


Table name: t_phoneBook

Field name	Field type
ID	NUMERIC
Name	TEXT
Number	TEXT

Click Create to create your table.



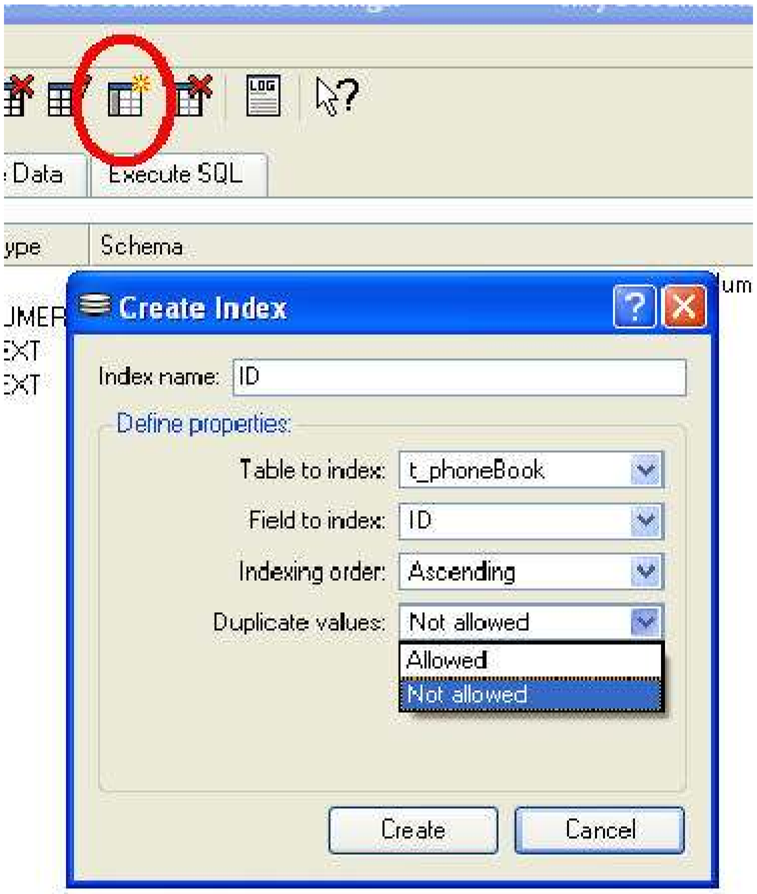
Name	Object	Type	Schema
t_phoneBook	table		CREATE TABLE t_phoneBook (ID NUMERIC, Name TEXT, Number TEXT)

A new line appears under the "Database Structure" tab. This line shows the SQL statement required to create the table. The browser does this for you automatically.

Since we wanted the ID field to be a unique index, click the Create Index button, circled in red below, and enter the details shown below.

Make sure you choose Not allowed as the Duplicate values property's value.

Press Create.



Index name: ID

Define properties:

Table to index: t_phoneBook

Field to index: ID

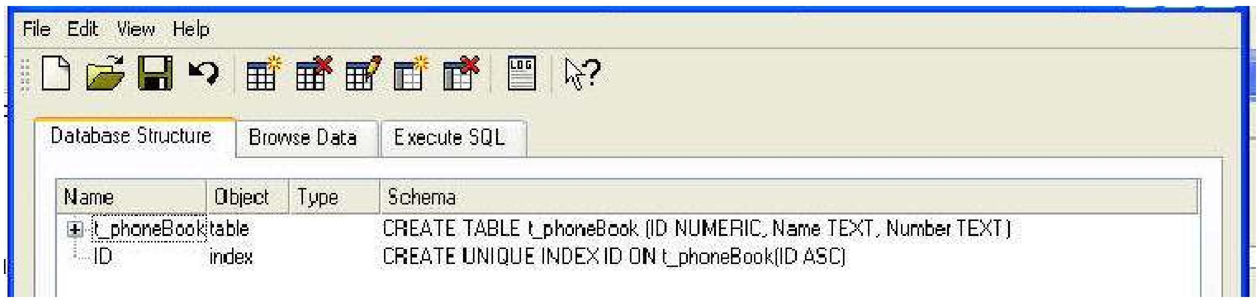
Indexing order: Ascending

Duplicate values: Not allowed

Allowed

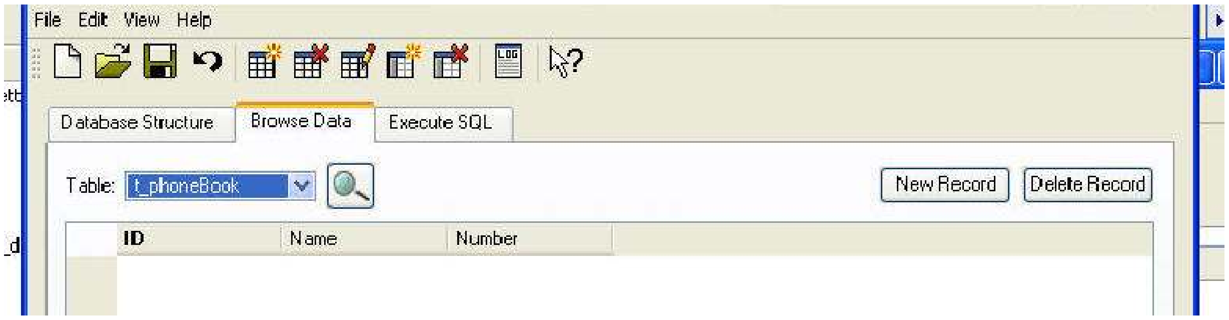
Not allowed

Note how this had changed the SQL required to define your table:

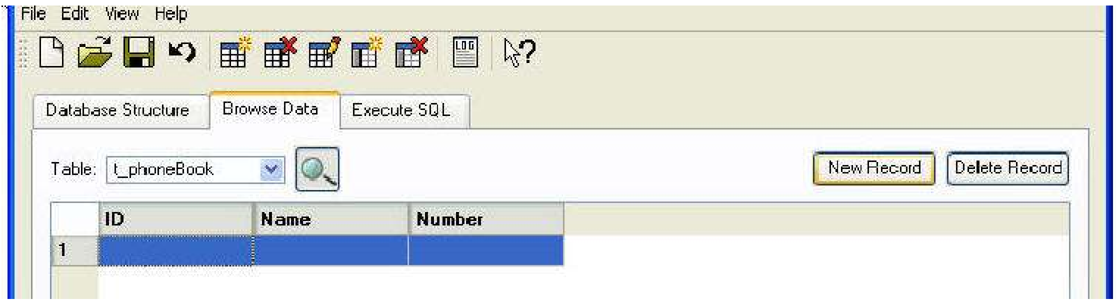


Name	Object	Type	Schema
t_phoneBook	table		CREATE TABLE t_phoneBook (ID NUMERIC, Name TEXT, Number TEXT)
ID	index		CREATE UNIQUE INDEX ID ON t_phoneBook(ID ASC)

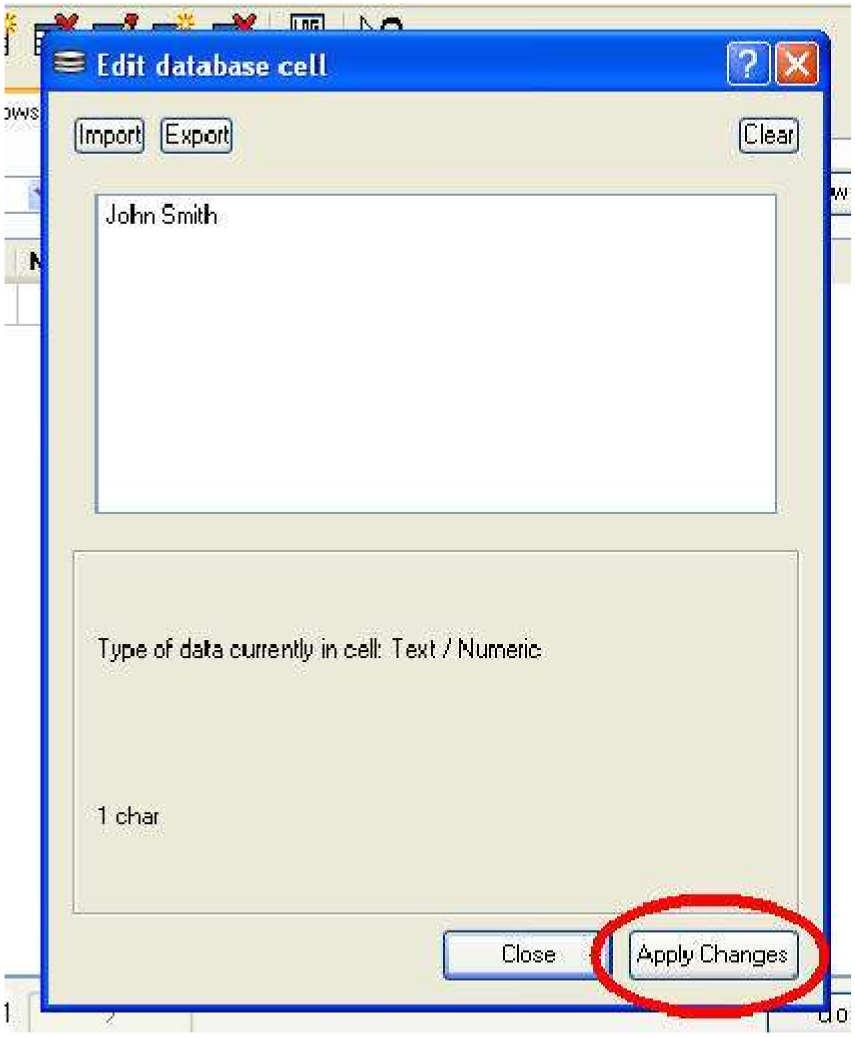
Now let's browse the table. Click the Browse Data tab.



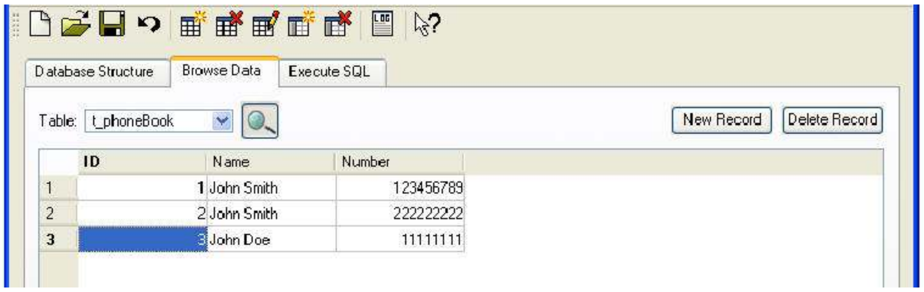
A table appears with columns ID, Name and Number. These are the fields you have created a minute earlier. Click the New Record button to start adding data to your table:



Double click the empty cell under the Name column header. The Edit database cell dialog appears. Enter a name, John Smith in this example, and click Apply Changes.

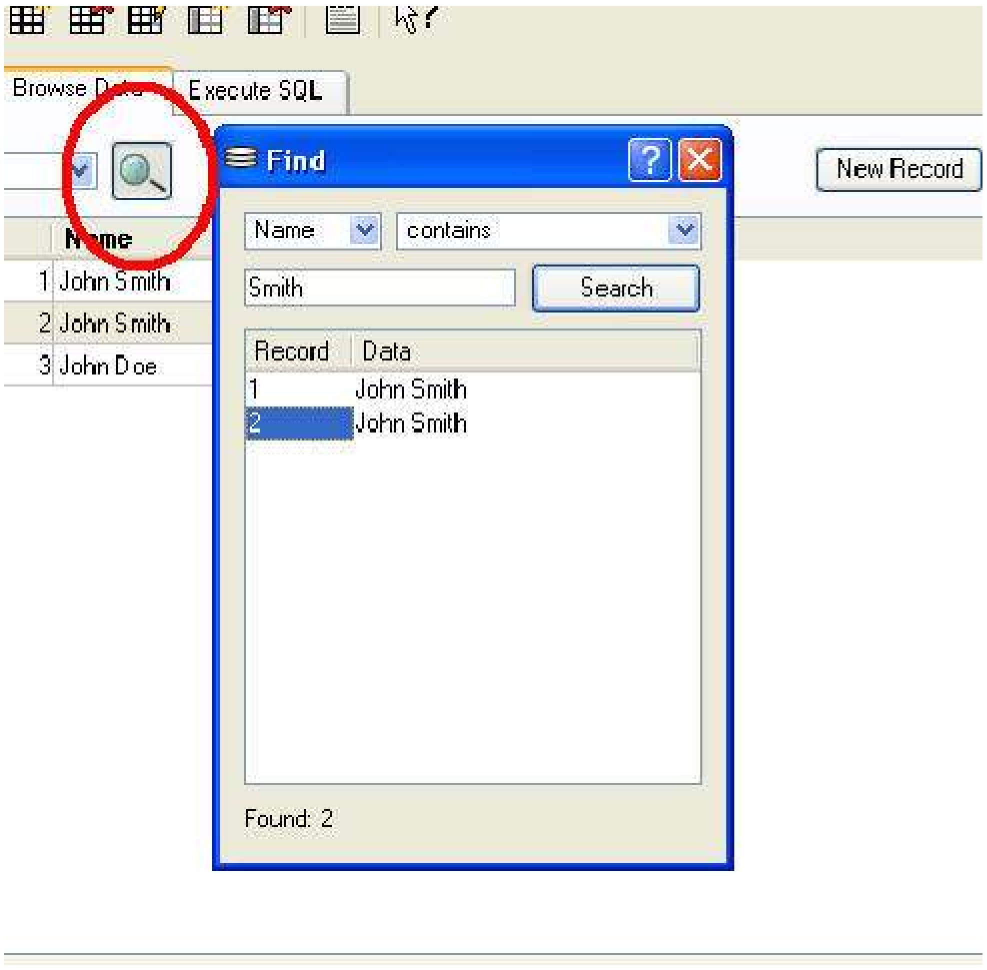


The same way, enter 1 as the first record's ID, and continue adding data as follows:



	ID	Name	Number
1	1	John Smith	123456789
2	2	John Smith	222222222
3	3	John Doe	11111111

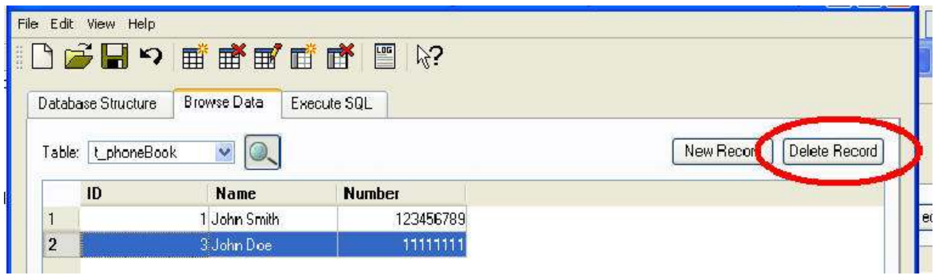
Now, we will take a look at some basic database operations. Start with Searching a record. Press the Find button, and fill the details as shown below. Click Search when done. The browser finds a list of the matching records.



Record	Data
1	John Smith
2	John Smith

	Name
1	John Smith
2	John Smith
3	John Doe

Next, exit the Find dialog box, click the second row of your table and click Delete Record. The row disappears and we are left with the other two.

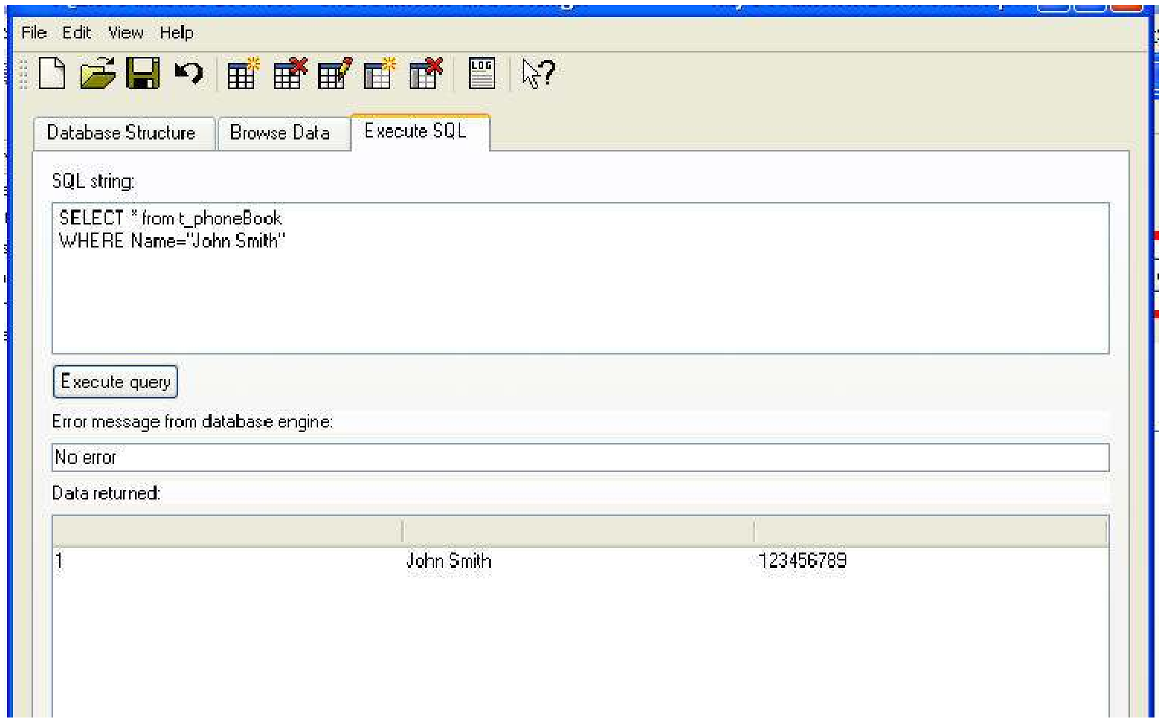


	ID	Name	Number
1	1	John Smith	123456789
2	3	John Doe	11111111

Last thing to check, we will look at the Execute SQL tab. This powerful tab allows you to execute arbitrary SQL statements, thus manipulating the database as if from within your code. It may be useful later, when you will need to check your SQL statements visually. Click this tab, and enter the SQL statement:

```
SELECT * from t_phoneBook
WHERE Name=”John Smith”
```

and click Execute Query. A list containing only one row (the only one left after we deleted the second “John Smith” row) appears.



```
SELECT * from t_phoneBook
WHERE Name="John Smith"
```

No error

1	John Smith	123456789

Sample program

This reference is accompanied by a sample program (download source). This part describes a complete plan of what this program should do.

The program is an order taking management system for a small restaurant. In order to keep things simple, only one table is used. We also assume there is one database hand-held in the device.

The program we describe below has been widely simplified in order to keep the example as limited to the goals as possible. The word **simplicity** indicates places in which we have considerably exceeded realistic design, or otherwise a note under the title "simplicity considerations".

Scenario and Analysis

- The restaurant with which we deal is a small one containing 6 tables.
- It is a fast-food, family-owned place, in which one waiter is working.
- There is also one cook, making the dishes.
- There are 4 optional items to choose from: a hamburger, ummon-fries, Coke and mineral water.
- Each table has 4 seats next to it.
- Work flows as follows:
  1. Waiter takes the order from the customers. He indicates into his Pocket PC what each one has ordered.

2. When done, the waiter goes to the kitchen and tells the cook what was ordered. He tells the cook what was the table number. **Simplicity** at least we could have print the order. We will skip this now.
3. Cook makes the dishes and places them on the bar, with a note indicating table number.
4. The waiter in turn serves them to the customers, and indicates in the database this order was served.

- **simplicity Note**: of course, stage 2 here is a bit awkward. It is much preferred to have data automatically synchronized with a central server and have the cook watching a screen hang on the wall indicating exactly which order should now be cooked, with its number. As this involves synchronizing between two copies of the database, it exceeds this reference's scope. Same is true about the diversity of choices: we have simplified it a lot and limited ourselves to 4 items, without duplications, so that we will not have to use more than one database table.
- Yet this simple example is pretty realistic. Such a restaurant, serving a very limited choice of dishes yet is crowded can be seen here. Note especially this photo, where the waitress' hand-held device is clearly shown.

Program Design

- Database:
  - One database will be used and called Rest.
  - One table will be used, and called t_orders ("t_" stands for table).
  - t_orders will contain the following fields:
    - ID (integer) – each record has a unique ID. This will be the table's primary key.
    - Sum (decimal numeric) – the total sum of this order. **Simplicity**: Currently this is calculated manually.

        - TableNum (integer) – table number (1 to 6). **Simplicity**: we will not check validation of input here.
        - SeatNum (integer) – the seat number next to the table. **Simplicity**: no validation checks here either.
        - Hamburger (ummon) – did this customer order a hamburger.
        - French (ummon) – did this customer order ummon fries.
        - Coke (ummon) – did this customer order a Coke.
        - Water (ummon) – did this customer order mineral water.
        - isServed (ummon) – is this order served yet? The waiter a box when serving the order.
        - Time – time this order was accepted.
    - **Note:**
        - Since SQLite does not contain Boolean nor date/time data types, all Boolean data will be kept as integer, where 0 indicates False and 1 indicates True. Time will be kept as text.
- This program is about to be built using the multitier architecture. It will contain a business logic layer that will implement the following methods:
    - Get all data of a single order with a given ID.
    - Add an order to the order list.
    - Update an order with a given ID in the order list.
    - Delete an order with a given ID from the list.
    - Return a list of all orders containing given values in the TableNum or the isServed field.
- User Interface (UI)
    - Order details: a form containing all order data. This form contains Save, Cancel and Delete buttons.

- Orders list: a list of orders, filtered as described in the filtering form. The list contains an option to show Order details form for each order.
- Filtering form: the place to indicate which orders should be displayed in the orders list. Contains a “Go” buttons which displays the list. Possible filters are:
    - Table number
    - Served/Not served
    - Date served
- Main menu: containing the buttons:
    - New order
    - Orders list

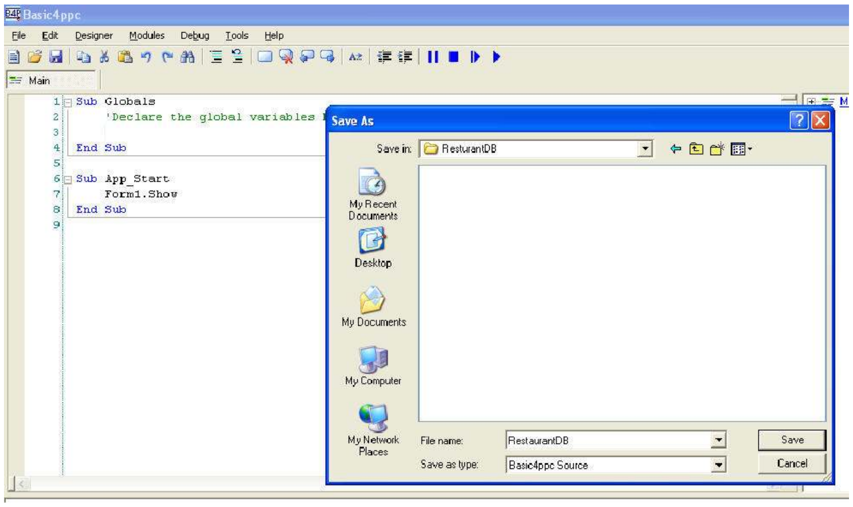
Source code

Each part of the source code is thoroughly explained in the relevant part.

Step 1 – Adding SQL support to your Basic4ppc application

This part demonstrates how to add the SQL libraries to your application. For more information about adding libraries, see the libraries chapter of the guide.

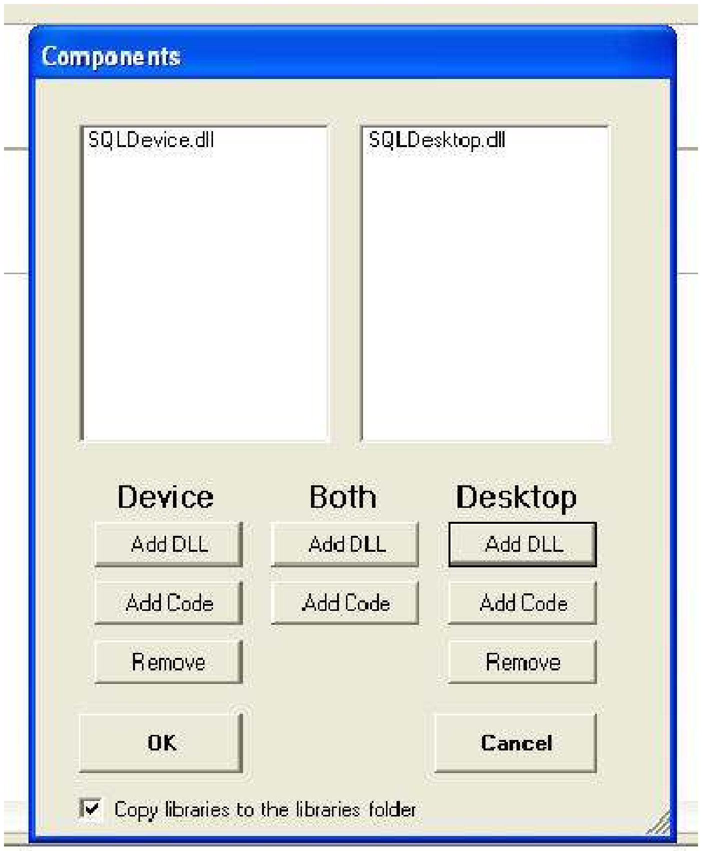
- Start a new Basic4ppc project. Save it under RestaurantDB.



```
Sub Globals
    'Declare the global variables
End Sub

Sub App_Start
    Form1.Show
End Sub
```

Save in: ResturantDB
File name: RestaurantDB
Save as type: Basic4ppc Source

- Add a new Form to your project and name it Form1.
- Click Tools – Components…, and add the .dlls SQLDevice and SQLdesktop to your project (device and desktop columns respectively) (read more about adding components and objects to your projects). Click OK:

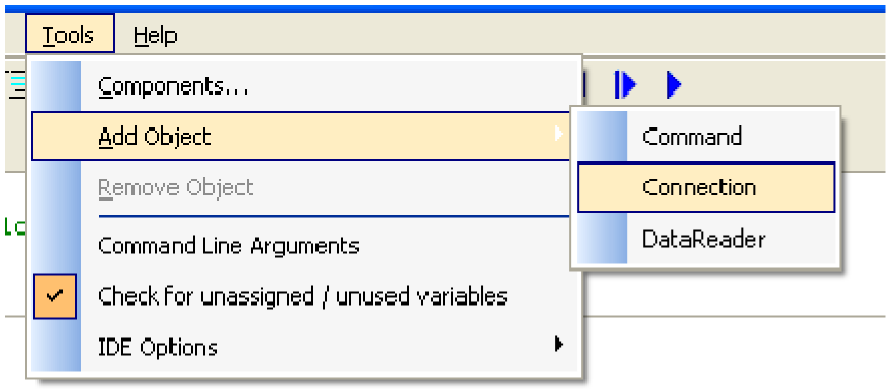


The last action is actually all that is needed to add SQL support to your application. Note that deploying an application with SQLite requires you to deploy the SQLite library with your application as described in details under **Deploying data driven application with Basic4ppc** at the end of this chapter.

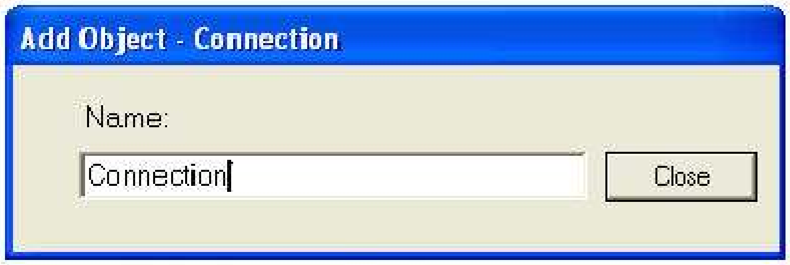
Step 2 – create and connect to a database

This part assumes you have added SQLite support to your application as described in the previous page. It goes on with developing the sample application.

- Add a connection object to the project, using the Tools – Add Object menu:

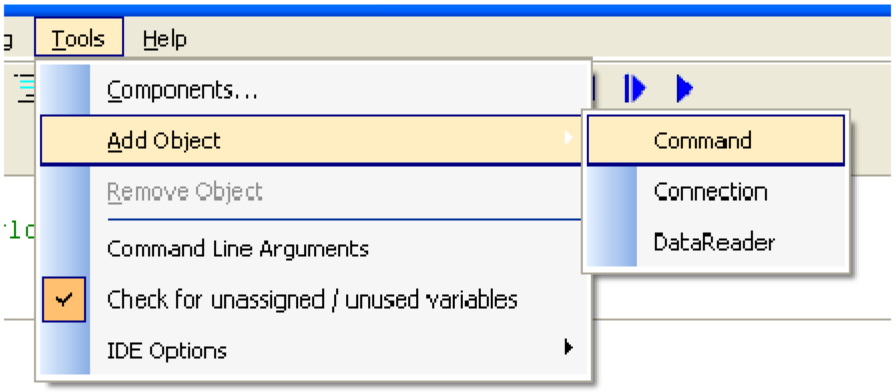


And name it Connection

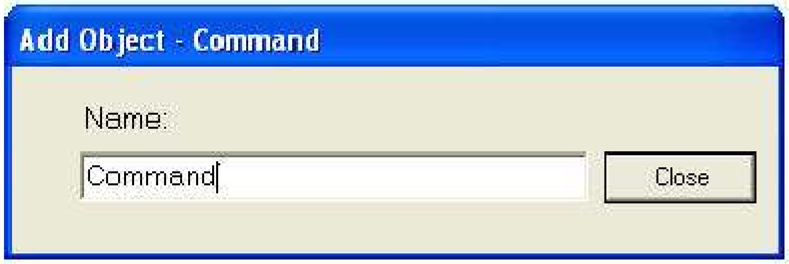


The "connection" object is an object found in the SQLite library that handles details regarding the connection with a specific database. Since SQLite uses a single file as a database, the connection object receives a filename as an argument, indicating the name of the database to which to connect.

- Add a command object the same way…



… and name it Command:



Name:
Command

- The first thing to do is to establish connection to the database. We will do this in a separate Sub called CreateConnection. Add the following lines to the programs code, below the last line of code:

```
9.  Sub CreateConnection
10. Connection.New1
11. Command.New1("", Connection.Value)
12. Connection.Open("Data Source = " & AppPath &
    "\Rest.lit")
13. End Sub
```

Add the code above at the end of your program, below the End Sub of the Sub App_Start

These lines of code form the basic connection to the database:

- In line 2 we instantiate a new instance of the Connection object we have previously added. We use the New1 constructor, which takes no parameters.

- In line 3 we instantiate a new instance of the Command object. Its New1 constructor takes two parameters: a default command text, left empty in this example, and a connection to be attached to. The command text property will be manually filled later. At this moment we are only interesting in creating the instance, and opening the connection.
- In line 4 we open the connection. The Open Sub takes a string as a parameter. This string is called Connection String and is used by many databases as a unique identifier of the database location relative to the application and other database related information. Using SQLite we only need to specify the filename (since it is a single-file, serverless database) to use. In this example we have named the file "Rest.lit", and located it in the application folder using the keyword AppPath. The Open Sub actually tells the connection object which file to use. This file represents the entire database. **If the file does not exist, Open will create it.**
- **Notes**:
  - We could have   ummon   line 3 at this stage. It is for convenience only that we have included it the same place where other objects and connection are initialized.
  - When creating the command, we use the Value property of the connection object. This Sub returns a reference to the connection object in a format expected by the command object.

Of course, after the first time you have created the database there is no need to create it anymore. This code either creates a database and connects to is, or just connects to it, if it already exists.

After writing this Sub, add a call to the Sub in the App_Start Sub as follows:

```
Sub App_Start

  CreateConnection

  Form1.Show

End Sub
```

This ensures the newly-created Sub (CreateConnection) will be called when the program starts.

We also have to make sure we close the connection when the program ends, typically when the main form (Form1 in our example) is closed.

Add the following code:

```
Sub Form1_Close

  Con.Close

End Sub
```

These are the actions required to open a connection. Next we shall create the tables.

Step 3 – Creating the database tables, and basic queries

This part assumes you have opened a database and connected to it as described it step 2. As shown in the manual-database-manipulation section, after creating the database you must create the tables resides in it. It is your responsibility to check if they already exist: you should only create them the first time you connect to the database.

The first thing to do is to check if the table we wish to create exists already. We do this by executing an SQL "SELECT" statement on a special table, contained in each SQLite database. This is the "SQLite_master" table.

SQLite_master table

Every SQLite database has an SQLITE_MASTER table that defines the schema (that is, the inner structure) for the database.

For tables, the **type** field will always be **'table'** and the **name** field will be the name of the table. So to get a list of all tables in the database, use the following SELECT command

```
SELECT name FROM sqlite_master
WHERE type='table'
```

Similarly, to get a list of one table or less with the name **'t_orders'** (that is, to check if a table exists), use the following SELECT command:

```
SELECT name FROM sqlite_master WHERE type='table' AND name='t_orders'
```

Executing SQL commands

SQL commands are executed from Basic4ppc using the Command object. We use the one previously instantiated and set value to the CommandText property. This is the SQL statement itself. Create a new Sub called CreateTableIfNotExists, and add the following line of code to it:

```
Command.CommandText = “SELECT name FROM sqlite_master WHERE type = ‘table’
AND name=’t_orders’”
```

The Command object now holds our SQL statement. In order to execute it, there exist two Subs in it: ExecuteReader, and ExecuteNonQuery:

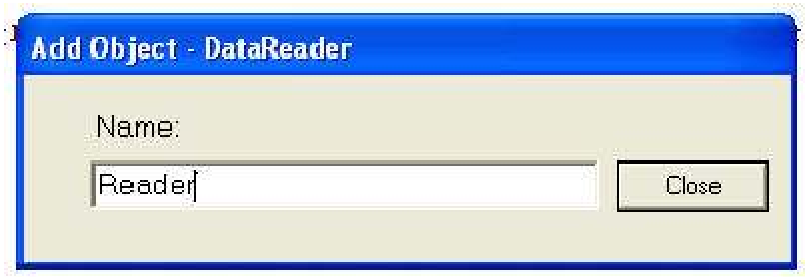
14. ExecuteReader is called when you execute a command that is about to return a Subset of the rows in a table (including an empty list of rows, and a list containing all rows in the table). This Sub returns an object of type DataReader. There must be a DataReader object in your application ready to be assigned to and instantiated (using the New1 constructor: see Instantiating – Libraries tutorial).
15. ExecuteNonQuery is called when you execute a command that manipulate the data in a way that does not return a list as a result. For example, if you delete a row from the table, you would have set the CommandText to be something like “DELETE t_order WHERE ID=1”. No rows are returned as a result. Instead, a number is returned, indicating the number of rows affected by the command.

Since the “SELECT” statement we use is intended to return results, it is essential to create the place to hold them. This place is the DataReader object.

The DataReader Object

When receiving a list of records from the database, you need a DataReader in order to access them. DataReader objects allow fast, forward only reading of a given list of records.

Start off with adding a DataReader object to your application, the same way you added the Connection and Command objects. Name the new object Reader. It is going to be used to hold the data returned by the command, when executed:



Add the following line below the previous one:

```
Reader.Value = Command.ExecuteReader
```

This will execute the command as described below. A list of data is now held in the Reader object (of type DataReader). Think of it as a big list:

Pointer	Column 1	Column 2	Column 3	Column 4	Column 5
▶	[data]	[data]	[data]	[data]	[data]
	[data]	[data]	[data]	[data]	[data]

The number of columns differs according to the result of the SELECT statement. A column is generated for each field in the statement. We later access these columns using the GetValue Sub. For now, we are only interested in the number of rows. In order to find out the number of rows we use the pointer.

Fetching the list from the DataReader

DataReaders use a pointer that indicates which is the "current row". Starting with the first row, this pointer points to one of the rows and can be moved forward to the next row using the DataReader's ReadNextRow Sub. The row to which the pointer points, is the one from which data is returned when accessing specific column, but this comes later. For now, only the number of rows we can promote the pointer is of interest. ReadNextRow returns True when succeeds and False if there is no more rows. We will use this feature and try to move the pointer one row forward. If it moves, this means we got at least one table in the list we got back: no need to create a new table. If it does not move, we shall create the table. The code looks like this:

```
                                                  'instantiate reader

Reader.New1

Reader.Value = Command.ExecuteReader

If Reader.ReadNextRow = False Then

  ' No table with this name in the database.

  ' Create one.

   Command.CommandText = "CREATE TABLE t_orders (ID INTEGER PRIMARY KEY, 
Sum REAL, TableNum INTEGER," & _
    "SeatNum INTEGER, Hamburger INTEGER, French INTEGER, Coke INTEGER, 
Water INTEGER, " & _
    "isServed INTEGER, Time TEXT)"

    Reader.Close

    Command.ExecuteNonQuery

End If
```

If the ReadNextRow Sub returned false, we set a new SQL command into the CommandText property of the Command object. This SQL command tells the database to create a new table with the list of fields specified.

Note: we used the INTEGER PRIMARY KEY as the data type of the ID field. In SQLite, this causes an auto-increment of this field when no value is entered.

Close the reader before you leave…

You are almost ready to test your code. One last thing is missing – closing the DataReader session. You must close the session before trying to execute another command: this resets the reader. If you did not, an error will occur. Change the End If above to else so that you add the following line of code at the end of the Sub:

```
Else

  Reader.Close

End If
```

And add a call to the CreatTableIfNotExists at the App_Start Sub:

Your code should now be looking like this:

```
Sub Globals

  'Declare the global variables here.

End Sub

Sub App_Start

  Form1.Show

  CreateConnection

  CreateTableIfNotExists

End Sub

Sub CreateConnection

  Connection.New1

  Command.New1("", Connection.Value)

  Connection.Open("Data Source = " & AppPath & "\Rest1.lit")
```

```
End Sub

Sub CreateTableIfNotExists

  ‘Find all the tables in this database

  Command.CommandText = “SELECT name FROM sqlite_master WHERE type = ‘table’
AND name=’t_orders’”

  ‘instantiate reader

  Reader.New1

  ‘Fill reader

  Reader.Value = Command.ExecuteReader

  If Reader.ReadNextRow = False Then

    ‘ No table with this name in the database.

    ‘ Create one.

    Command.CommandText = “CREATE TABLE t_orders (ID INTEGER PRIMARY KEY,
Sum REAL, TableNum INTEGER,” & _
    “SeatNum INTEGER, Hamburger INTEGER, French INTEGER, Coke INTEGER,
Water INTEGER, “ & _
    “isServed INTEGER, Time TEXT)”

    Command.ExecuteNonQuery

  Else

    ‘Close reader anyhow

    Reader.Close

  End If

End Sub
```

Note: the method illustrated here is a bit clumsy. The DataReader part could be skipped, and the SQL statement “IF NOT EXISTS” could have been used. We chose to present the somewhat longer way in order to introduce the reader with both the DataReader object (at this early stage of the reference) and the SQLite unique “sqlite_master” table. This gives a better understanding of manipulating the database scheme.

Before Step 4: Adding Parameters to SQL Commands

A very common scenario with SQL commands is that you do not know at design time which values you will need. Consider the following case: you wish to get a list of all orders made in the restaurant to table number 4. The proper SQL command is:

```
SELECT * FROM t_orders WHERE TableNum = 4
```

Dangerous use of SQL Command Strings

But if you do not know orders from which table you will have to display? Assume the user enters the number at run time. And say you stored it in a variable called strTableNumber. Then you can write something like:

```
‘ ---- WRONG WAY ---

Command.CommandText = “SELECT * FROM t_orders WHERE TableNum=” &
strTableNumber ‘WRONG WAY!

Reader.Value = Command.ExecuteReader
```

The code above will work fine if you entered a number to the strTableNumber variable. But this is very dangerous to use this code in a “real world” application. This code is exposed to “SQL injection” – the most common way to destroy your database by inserting malicious script into the data input. This method takes advantage of the fact the character “;” is used in SQL to separate two adjacent statements. So if, as a response to your request for input, the user entered the string “4; DELETE FROM t_orders”, all records in the t_orders table will be deleted.

In order to protect against this kind of violence, SQL commands use parameters to indicate data that is not known at development time.

Advantages of parameters

- Protect against SQL injection.
- Faster then concatenated string query (significantly faster).
- Eliminate the need to use escape characters for special characters.

Using parameters in an SQL statement

If an SQL statement contains a word preceded by the character “@”, the word is interpreted as a parameter:

```
Command.CommandText = “INSERT INTO [table1] (col1, col2) VALUES (@ID, @Name)”
```

In this code, the values that will be inserted into the tables are the values stored in the parameters named ID and Name. You declare the parameters using the Command object’s AddParameter Sub, and you assign a value to it using the SetParameter Sub:

```
Command.AddParameter(“ID”)
Command.AddParameter(“Name”)

Command.SetParameter(“ID”,”strNewID”)
Command.SetParameter(“Name”,”strNewName”)

Command.CommandText = “INSERT INTO [table1] (col1, col2) VALUES (@ID, @Name)”
```

Note: you must NOT add the “@” character when calling the SetParameter Sub. If you wrote something like the following code, an error message will pop, indicating **line 4** as the cause to the error. This will not help you much (see common errors) in figuring out you should not have added the “@” at lines 1 and 2.

```
1    Command.AddParameter(“@ID”)

2    Command.SetParameter(“@ID”,”strNewID”)
```

```
Command.CommandText = "INSERT INTO [table1] (col1) VALUES (@ID)"
Command.ExecuteNonQuery
```

Step 4 – Implementing the Business Layer

This part assumes you have followed the instructions up to step 3 (creating the tables) and "Before step 4 – parameters". This part uses the information presented earlier and the objects created and code written, to implement the functionality described in the program design part.

What is the Business Layer?

The "business layer" is a set of Subs the user interface calls and are logically located in between the user-interface and the database. Since a complete explanation of the multitier (multilayer) architecture is out of the scope of this guide, here is a link to the relevant Wikipedia article. For our purpose it is enough to know that the "Layer" is actually a set of Subs. We call these Subs from the UI, and they call the database. This way we can isolate database from the UI, Thus

1. Creating a more readable code
2. Allowing later replacement of each side easily
3. Simplifying debugging by isolating database accessing from UI events.

Methods contained in the Business layer

The business logic layer is the part of the program with which the user interface connects. Subs implemented there are:

- getOrderData (ID) – retrieves all data of a single order with a given ID.
- addOrder(sum, tabNum, seatNum, isHamburger, isFrench, isCoke, isWater, isServed, timeTaken) – adds an order, with the given parameters, to the order list.
- updateOrder(id, sum, tabNum, seatNum, isHamburger, isFrench, isCoke, isWater, isServed, timeTaken) – Update an order with a given ID in the order list.
- deleteOrder(id) – Delete an order with a given ID from the list.
- getListOpenedOrdersToTable(isServed) – Return a list of all orders containing given value in the isServed field.
- getListOrdersToTable – Returns a list of all orders in the database.

Points of interest in the code

A few points below worth mentioning even if you are not about to read it all. If you are in a hurry, pay attention especially to the transactions part, and to the ExecuteTable Sub, described below. Apart from these two new things, there are a few tips and a demonstration of using SQL main statements: SELECT, INSERT INTO, UPDATE, DELETE and WHERE.

Transactions

Before we go on to the implementation, we introduce an important feature to be aware of. The Connection object demands that when executing any SQL command, there must be a "Transaction" opened for the command. If there is not one, the Connection object creates one. When executing several commands in a row, it is faster to create the transaction manually. Start with Connection.BeginTransaction and end your code-block with Connection.EndTransaction. For instruction reasons, we will follow this rule in each Sub here.

Code for the Business Layer

getOrderData(ID) – notes:

1. We start the transaction at line 3, and end it at line 26.

2. Adding parameters is done as described earlier.

3. In line 13 we assign the data to a global variable, defined in Sub_Globals, called order. This is a structure that holds the order data. It is used as a bridge between the Subs and the User Interface: later these values are copied to the form's controls.

4. Do not forget to close the reader.

```
1 Sub getOrderData(ID)
2       'Start transaction
3       Connection.BeginTransaction
4       'Set parameter values
5       Command.AddParameter("ID")
6       Command.SetParameter("ID", ID)
7       Command.CommandText="SELECT * FROM t_orders WHERE ID=@ID"
8       'Fetch data
9       Reader.Value = Command.ExecuteReader
10
11      'Assign data
12      If Reader.ReadNextRow = True Then
13            order.ID = ID
14            order.sum = Reader.GetValue(1)
15            order.tableNum = Reader.GetValue(2)
16            order.seatNum = Reader.GetValue(3)
17            order.hamburger = Reader.GetValue(4)
18            order.french = Reader.GetValue(5)
19            order.coke = Reader.GetValue(6)
20            order.water = Reader.GetValue(7)
21            order.isServed = Reader.GetValue(8)
22            order.timeTaken = Reader.GetValue(9)
23      End If
24      Reader.Close
25      ' End transaction
26      Connection.EndTransaction
27 End Sub
```

addOrder – notes:

1. Note the way the SQL string is split into two lines for readability.

2. Note the way time is inserted into the database (yellow line). This way simplifies readability of the database records later when injected into a table. It is not always the best way.

```
Sub addOrder(sum, tabNum, seatNum, isHamburger, isFrench, isCoke, isWater, isServed,
timeTaken )
            ‘ Start transaction
            Connection.BeginTransaction
            ‘ Create parameter
            Command.AddParameter(“sumMon”)
            command.AddParameter(“tabNum”)
            command.AddParameter(“seatNum”)
            command.AddParameter(“isHamburger”)
            command.AddParameter(“isFrench”)
            command.AddParameter(“isCoke”)
            command.AddParameter(“isWater”)
            command.AddParameter(“isServed”)
            command.AddParameter(“timeTaken”)

            command.SetParameter(“sumMon” ,sum)
            command.SetParameter(“tabNum”, tabNum)
            command.SetParameter(“seatNum”, seatNum)
            command.SetParameter(“isHamburger”, isHamburger)
            command.SetParameter(“isFrench”, isFrench)
            command.SetParameter(“isCoke”, iscoke)
            command.SetParameter(“isWater”, isWater)
            command.SetParameter(“isServed”, isServed)
            command.SetParameter(“timeTaken”, Time(timeTaken))
            Command.CommandText = “INSERT INTO t_orders
(ID,Sum,TableNum,SeatNum,Hamburger,French,Coke,Water,isServed,Time)” & _
             “ VALUES
(null,@sumMon,@tabNum,@seatNum,@isHamburger,@isFrench,@isCoke,@isWater,@isServed,
@timeTaken)”

            Command.ExecuteNonQuery
            ‘ End transaction
            Connection.EndTransaction
End Sub
```

updateOrder

```
Sub updateOrder(id, sum, tabNum, seatNum, isHamburger, isFrench, isCoke, isWater,
isServed, timeTaken )
            ‘ Start transaction
            Connection.BeginTransaction
            ‘ Create parameters
            Command.AddParameter(“id”)
            Command.AddParameter(“sum”)
            Command.AddParameter(“tabNum”)
            Command.AddParameter(“seatNum”)
            Command.AddParameter(“isHamburger”)
            Command.AddParameter(“isFrench”)
            Command.AddParameter(“isCoke”)
            Command.AddParameter(“isWater”)
            Command.AddParameter(“isServed”)
            Command.AddParameter(“timeTaken”)
            Command.SetParameter(“id” ,id)
            Command.SetParameter(“sum” ,sum)
            Command.SetParameter(“tabNum”, tabNum)
            Command.SetParameter(“seatNum”, seatNum)
            Command.SetParameter(“isHamburger”, isHamburger)
            Command.SetParameter(“isFrench”, isFrench)
            Command.SetParameter(“isCoke”, isCoke)
            Command.SetParameter(“isWater”, isWater)
            Command.SetParameter(“isServed”, isServed)
            Command.SetParameter(“timeTaken”, timeTaken)
            Command.CommandText = “UPDATE t_orders SET Sum=@sum,
TableNum=@tabNum, SeatNum=@seatNum, Hamburger=@isHamburger,
French=@isFrench, “ & _
                                                                  “Coke=@isCoke,
Water=@isWater, isServed=@isServed, Time=@timeTaken WHERE id=@id”
            Command.ExecuteNonQuery
            ‘ End transaction
            Connection.EndTransaction
End Sub
```

deleteOrder

```
Sub deleteOrder(id)
            ‘ Start transaction
            Connection.BeginTransaction
            ‘ Create parameters
            Command.AddParameter(“id”)
            Command.SetParameter(“id” ,id)
            Command.CommandText = “DELETE FROM t_orders WHERE id=@id”
            Command.ExecuteNonQuery
            ‘ End transaction
            Connection.EndTransaction
End Sub
```

getListOpenedOrdersToTable – notes:

The main thing of interest here is the new Sub under the Command object: ExecuteTable. This part breaks the rules of the multitier model as it allows you to fill a table control directly. Yet, this is very convenient and easy to do and we used it. Note that the table does not necessarily have to be visible, and thus can be used as a virtual storage for your records in memory. The ExecuteTable gets two parameters: the name of the table control to fill, and the maximal number of rows to use. Generally speaking, it is not recommended to use too many rows in the table, because this has a significant performance penalty.

```
Sub getListOpenedOrdersToTable(isServed)
            ‘ Start transaction
            Connection.BeginTransaction
            ‘ Create parameters
            Command.AddParameter(“isServed”)
            Command.SetParameter(“isServed” ,isServed)
            Command.CommandText = “SELECT * FROM t_orders WHERE
isServed=@isServed”
            Command.ExecuteTable(“tblOrdersList”, 1000)
            ‘ End transaction
            Connection.EndTransaction
End Sub
```

getListOrdersToTable

```
Sub getListOrdersToTable()
            ‘ Start transaction
            Connection.BeginTransaction
            Command.CommandText = “SELECT * FROM t_orders”
            Command.ExecuteTable(“tblOrdersList”, 1000)
            ‘ End transaction
            Connection.EndTransaction
End Sub
```

Step 5 – Building the User Interface

As this is not intended to be a User Interface development guide, we will focus on three main issues: Table control, Common techniques, and a brief description of the sample program's user interface.

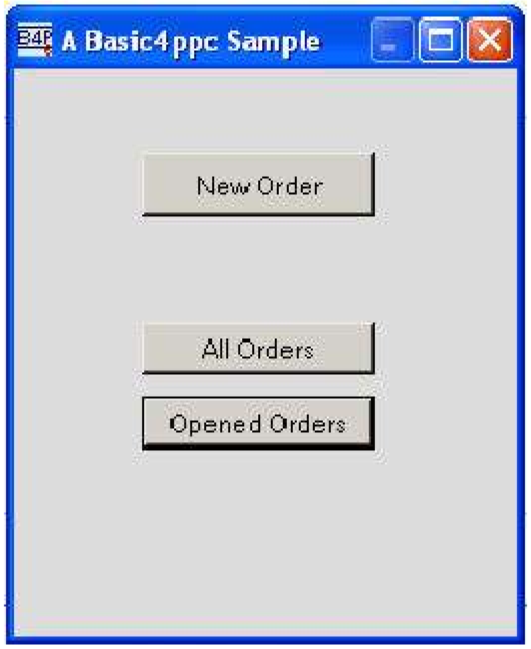
Table Control Dynamic SQL link

The table control can be filled directly from a Command object. To do this, use the ExecuteTable method. You have to have a table control in your code. This command deletes everything from the table, and then fills it. Column names are the names of the fields in the database. Cell values are the values entered to the database. A common mistake is to use this table with a huge amount of rows: this control is not planned to work smoothly with more than a couple of thousands rows in it.

Sample code UI design

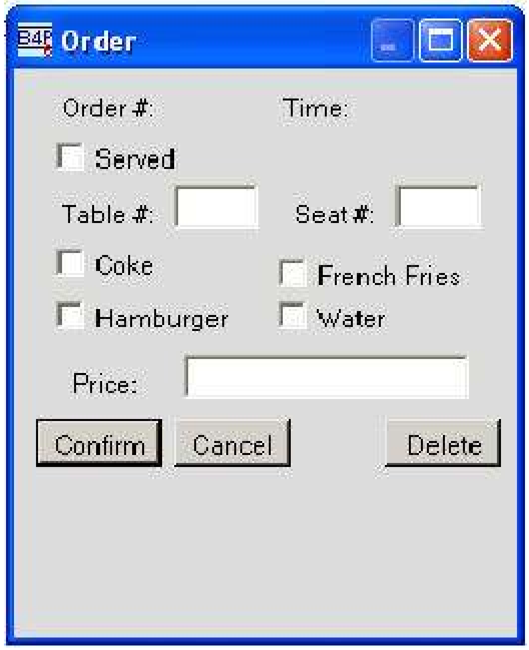
The User Interface in the sample code consists of one module, and three forms:

<u>Form1: entering the program</u>



Form1 holds three buttons, acting as a menu: New order, calling the Order form, and two buttons causing a list to show: the first displays all orders, and the second displays opened orders only (not served). Both direct the application to the List form.

<u>frmOrder: the Order form</u>

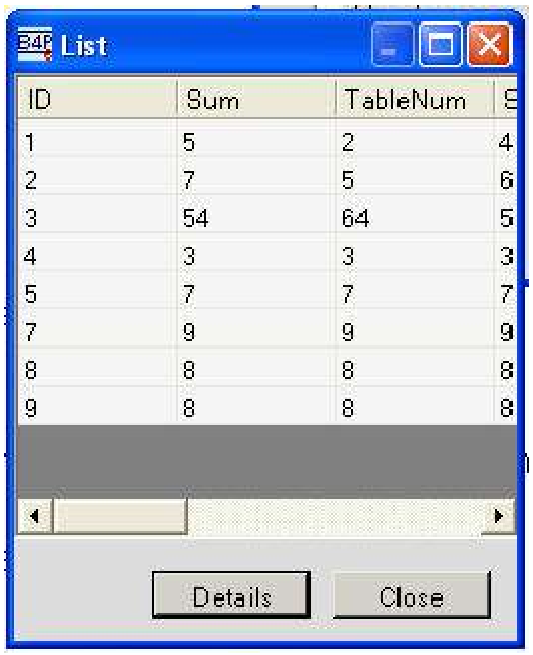


The order form holds the user interface required to enter order data.

Note:

1. Same form is used for updating and adding records. The decision whether to call **add** or **update** is made when closing the form. The decision whether or not to fill the controls with data is made when entering the form. A global structure, named order, is used to save the state of the application at each point.
2. Deleting is done from here by raising the **delete** Sub.

<u>frmList: the lists form</u>



ID	Sum	TableNum	S
1	5	2	4
2	7	5	6
3	54	64	5
4	3	3	3
5	7	7	7
7	9	9	9
8	8	8	8
9	8	8	8

This form is used in order to select the desired order for extra details. User interface here is very basic, but very easy to implement. Pressing the Details button moves us to the order form, this time with details filled.

Common UI techniques

Common tasks implemented here are:

1. Using a global variable – named order – as the application “working order”. This variable is used to return data from the Subs to the forms.

2. Calling the Business logic Subs with all details for code saving.

3. Updating the database only when clicking “Confirm”. Do not try to follow minor changes the user does with each of the fields: instead, display an input form, wait until he makes up his mind and update everything at once.

Deploying an application that uses SQL

Deploying an application that uses SQLite requires you to **manually include** the SQLite open source library.

This topic has earned its right to have a page of its own in this reference being the Subject of many questions in Basic4ppc forum on the web. Follow the steps carefully. **The required files differ between versions 6.5 and earlier of Basic4ppc to the latest ones.**

Latest Basic4ppc version:

There are **three** .dll files for SQLite:

1. For a Desktop only application, copy the file named "System.Data.SQLite.DLL" and deploy with your application.
2. For a Device only application, copy the **files**:
   a. System.Data.SQLite.Device.DLL
   b. SQLite.Interop.**060**.DLL – note that the **"060"** may change in future versions. Use the latest: for example if you have SQLite.Interop.070.dll, use this rather than the 060.
16. For a dual (desktop/device) application, use all three files.

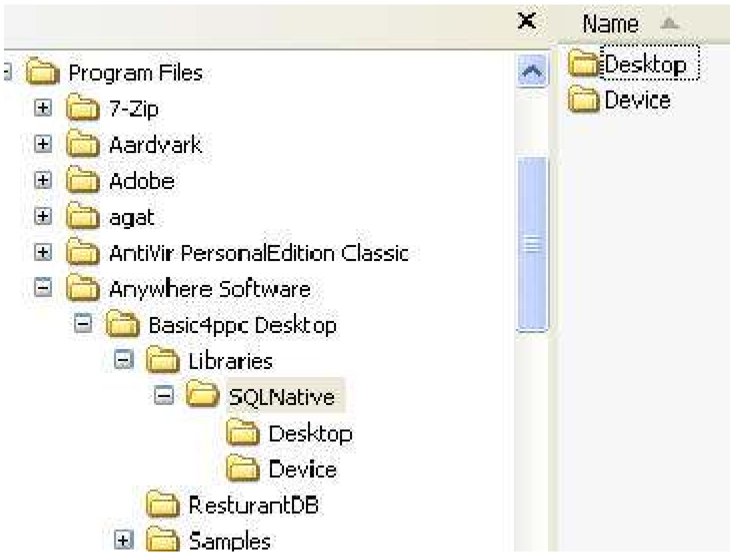
Basic4ppc version 6.5 and earlier:

"System.Data.SQLite.DLL"

Note: there are **two libraries with the same name** – a device library and a desktop library. They differ from one another. It is the developer responsibility to make sure he deploys both with the compiled application. You can distinguish them from one another using their size: the device library's size is 450K, whereas the desktop one is 568K.

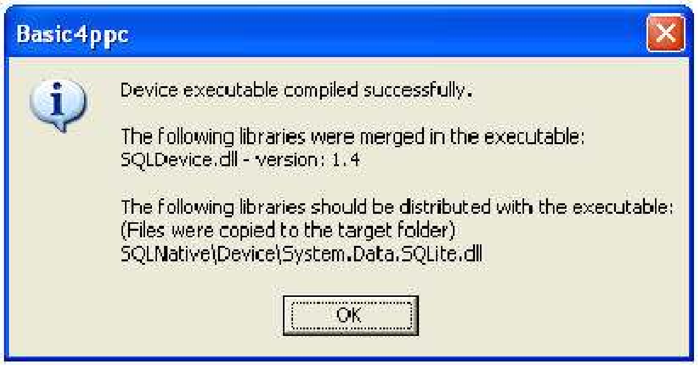
Where to find these files:

Both files are located at Basic4ppc Desktop\Libraries\SQL Native under either Desktop or Device:



Typical location of System.Data.SQLite.DLL

A message indicating what should be copied will appear after compiling. Note, there is no indicating as to which of the two files to copy. Try it out on your device before you deploy…:

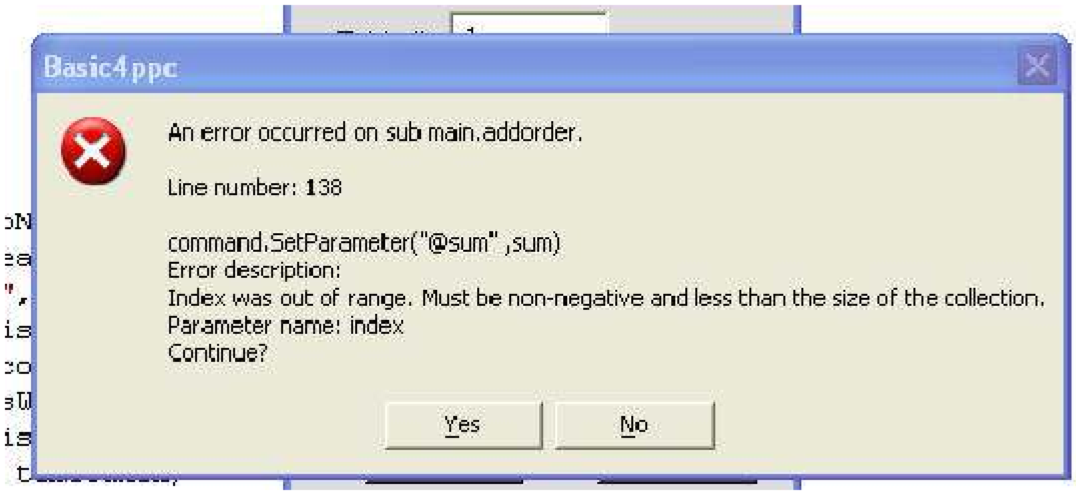


```
Device executable compiled successfully.

The following libraries were merged in the executable:
SQLDevice.dll - version: 1.4

The following libraries should be distributed with the executable:
(Files were copied to the target folder)
SQLNative\Device\System.Data.SQLite.dll
```

Error Messages

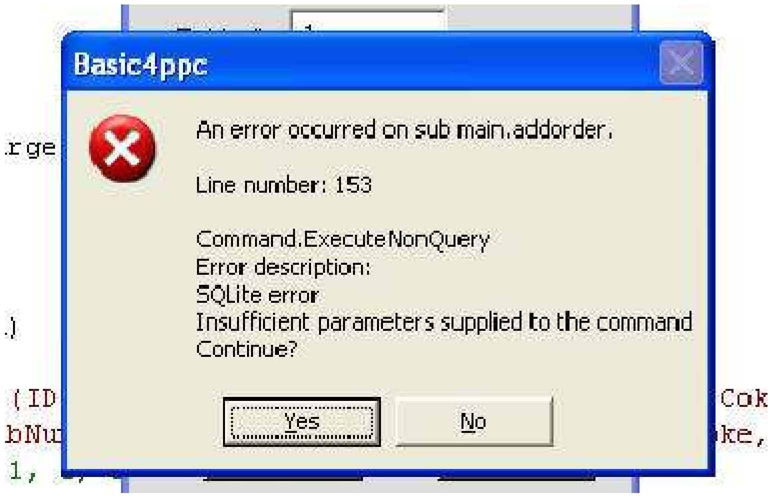
This part gathers some of the most common error messages you may run into.

17. "Index was out of range" when trying to set a parameter's value: you probably had a typo in the parameter name. Basic4ppc could not find a parameter added by that name.



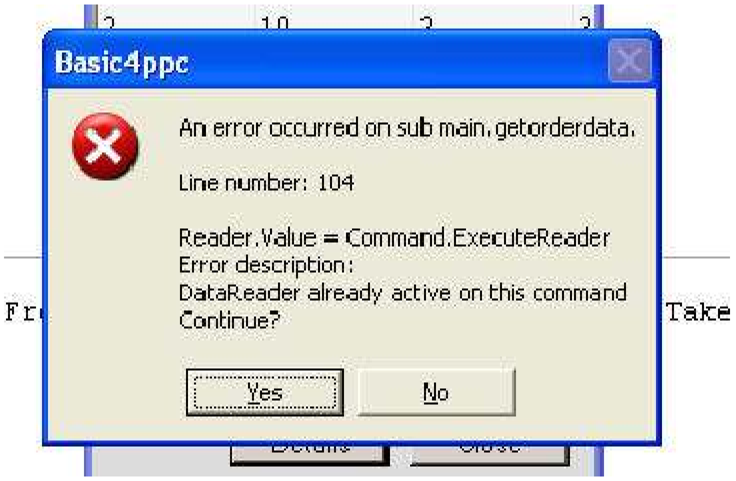
```
An error occurred on sub main.addorder.

Line number: 138

command.SetParameter("@sum" ,sum)
Error description:
Index was out of range. Must be non-negative and less than the size of the collection.
Parameter name: index
Continue?
```

2."Insufficient parameters supplied to the command" message when executing one of the execute Subs. Most probably you did not set value to one of the parameters specified in the SQL statement. You may have forgotten to add them at all. You may have a spelling mistake.



```
Basic4ppc

An error occurred on sub main.addorder.

Line number: 153

Command.ExecuteNonQuery
Error description:
SQLite error
Insufficient parameters supplied to the command
Continue?
```

18. DataReader already active on this command: you did not use the DataReader.Close Sub before trying to execute another command.



```
Basic4ppc

An error occurred on sub main.getorderdata.

Line number: 104

Reader.Value = Command.ExecuteReader
Error description:
DataReader already active on this command
Continue?
```

Some more errors, including errors forum user had encountered, are added at the end of the FAQ chapter.

FAQ

This part contains new issues as well as issues discussed earlier. It is gathers technical details that are either unique to Basic4ppc, or are easy to forger, or are good to have at hand, and some important questions from the Basic4ppc Forum.

I'm new to SQL / to SQLite. What should I do?

Where do I get the latest version of SQLite interop.060?

Questions and errors from the forum

Simple example

http://www.basic4ppc.com/forum/questions-help-needed/5558-sql-error.html#post32628

Disk IO Error

Q: I/O Error (v. 6.8 and up): When I run a program using SQLite on a device emulator, I get an error message: **Disk I/O error in SQLite**. What should I do?

A: See this thread in the Basic4ppc forum: Error: Disk I/O with SQLite

Cannot commit transaction

http://www.basic4ppc.com/forum/questions-help-needed/4732-encountered-confusing-error-message-please-post-here-5.html#post32296

Troubles with .dlls and SQL

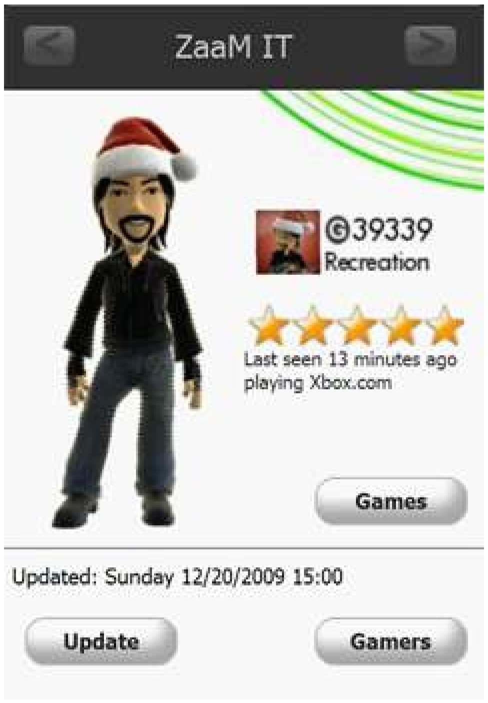
http://www.basic4ppc.com/forum/questions-help-needed/5465-pinvoke-sqlite.html#post32287

My question wasn't listed here. Where is there a link to the Forum?

Part III - Graphics and Graphical User Interface

Background

Creating GUI and displaying graphics is a complicated task. These modern days, you encounter this task almost each time you create an application. It is getting even more complicated on a small, resources-limited computer such as a mobile device. This section addresses the issue and shows you ways to create real world applications - from the very basics through the analysis of complicated GUIs.



Last seen 13 minutes ago playing Xbox.com

Updated: Sunday 12/20/2009 15:00

A sample GUI created by the forum user ZaaM

This chapter assumes the knowledge of the first section. You are assumed to have read it, especially the designer part. Explanations and code samples rely a lot on this assumption. Yet, if you have previous programming experience there is no need that you read both – you will have no problem understanding this chapter. This is especially true is you have ever experienced programming with graphical user interface designer, ever worked with controls as UI components and experienced working with Windows Forms.

The issue of creating a modern looking, easy to use, intuitive user interface has gained growing attention during the last years, as described in the GUI Basics chapter. Along with this, the demand for better graphics ability on handheld devices is increasing.

In this section, you will learn how to:

- Create basic user interface with the standard controls
- Modify user interface for advanced usage using external standard libraries
- Create a modern, finger-based, touch-screen interface
- Use the additional, open source, user created libraries to add controls to your applications
- Creating graphics, images, lines, moving shapes and more
- Work with images
- Combine graphics in your user interface

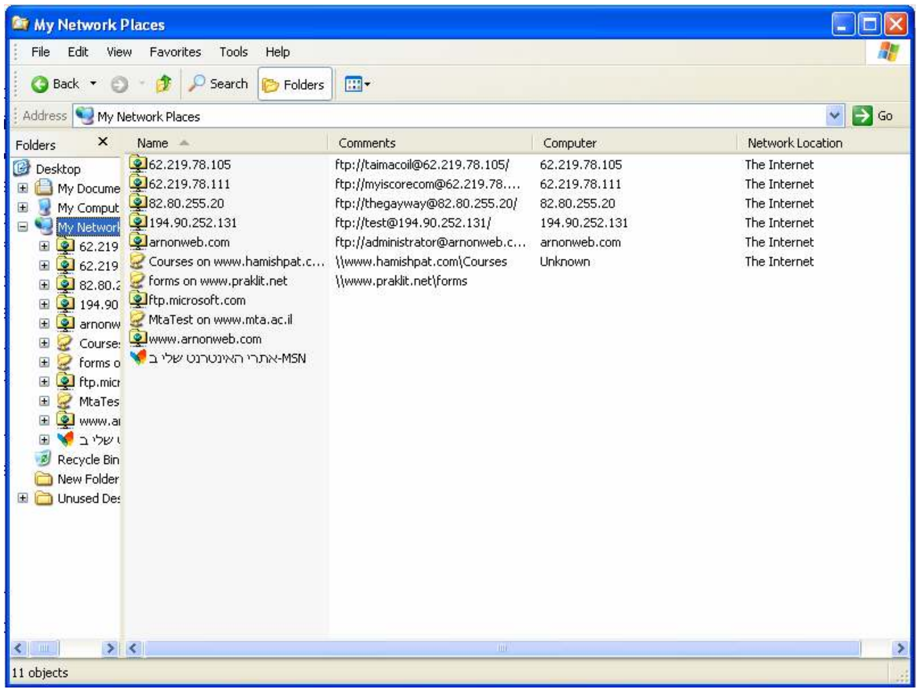
A very important source of knowledge you <u>should</u> read is the Graphics Tutorial written by a graphics-expert community member – Klaus, published in the Basic4ppc forum. This tutorial is referenced to later, with a description of what's inside – but you should read it as the details and the level of preciseness Klaus demonstrates are second to none.

Graphics and UI - what's the difference?

The distinction between "GUI" (Graphical User Interface) and "Graphics programming" is not as distinctive as you may think. Basically, the creation of graphics on your screen is quite a complicated task to carry out. Take into account you computer has to handle each pixel on your screen (pixel, short for Picture Element, is the smallest "dot" a screen can show. They are organized in rows and columns. Typically you have around 1,000,000 on your screen, each can have about 65000 colors). And worse, the programmer has to tell the computer how to do it! Obviously there is no way to manually set the value (color) of every pixel. What you can do, however, is use pre-programmed "components", libraries, controls etc, each carrying out a specific task. For example, you have seen that when you want to display a button you place a component called button on the screen, and it draws by itself everything you need. Further more, it communicates with Windows so that when you move the window where it resides it goes with it, and when you click it, an event is fired. These components are divided roughly into two sections: Controls, which are user interface components, and other external libraries and objects.

Controls

Controls are mainly designed to create part of the user interface on your screen. From the point of view of a programmer, you compose the UI from various controls – for example, the known Windows Explorer is actually built of the following: a **TreeView** control composing the left hierarchical view, a **Separator** composing the bar that separates between the left and the right part, a **ListView** control, creating the list to the right, a **Main Menu** and various **Toolbars**. Each of them supplies different functionality: for example, the listview to the right automatically allows you to select one or more of the items on the list.



Name	Comments	Computer	Network Location
62.219.78.105	ftp://taimacoil@62.219.78.105/	62.219.78.105	The Internet
62.219.78.111	ftp://myiscorecom@62.219.78....	62.219.78.111	The Internet
82.80.255.20	ftp://thegayway@82.80.255.20/	82.80.255.20	The Internet
194.90.252.131	ftp://test@194.90.252.131/	194.90.252.131	The Internet
arnonweb.com	ftp://administrator@arnonweb.c...	arnonweb.com	The Internet
Courses on www.hamishpat.c...	\\www.hamishpat.com\Courses	Unknown	The Internet
forms on www.praklit.net	\\www.praklit.net\forms		
ftp.microsoft.com			
MtaTest on www.mta.ac.il			
www.arnonweb.com			
אתרי האינטרנט שלי ב-MSN			

All the programmer had to do is to fill the list with the names of the file in the current folder. So, apart from displaying the graphics required for them, controls supply different behaviors, which save a time and offers standardization. Furthermore, they implement functionality you might find very difficult to implement yourself: Consider, for example, the Basic4ppc bList control, creating a finger-scrolled list. Creating a control yourself, or worse, implementing the functionality each time you need it, is impossible.

There are some drawbacks, of course:

- Performance issues – being planned for user interactions, controls are usually not planned for very fast graphic reactions, and if you place too much of them on a form you may run into performance issues, as well as memory problems.
- Flexibility - You have only what's ready made – you can't customize everything.

In most cases these are not a significant problem and the advantages in efficiency are significant, but in these rare cases when you need something not implemented as a control, you need to use the somewhat more "native" graphics ability.

Graphics

Graphics is basically everything you display on the screen that is not a Control. This include lines, shapes, gradients, color fills, moving sprites, images (sometimes) and more: much of the ability is partially available through the various controls and some isn't, and usually it's a combination – you do what you can with regular controls, and you combine them with things needed to be precisely done with graphics.

How this section is built

This section covers many different topics, all part of the general issue of Graphics and UI. In order to let you decide how you wish to use the section, we shall start with describing briefly how it is built. This way you can decide whether it's best for you to read it all, skim trough parts or pinpoint a specific topic you need.

Basic graphics and GUI overview

- Theory about graphics – we start off with giving general background for the beginners. This is not intended to give a complete discussion, but to make sure you know what pixels, resolution, and image file types are.
- Basics of graphics and GUI: covers everything there is to know about how to create the native graphics and create GUI at runtime. Covers the techniques without getting too much into controls types.
- Advanced graphics: a brief description of some more-advanced topics with graphics and UI, including AutoScale.
- UI basics – this part aims to give a comprehensive reference of the most important controls you have at hand. It covers both Basic4ppc native controls and those created by users.
- Modern GUI – a step by step tutorial for creating a finger-based UI.
- Code Samples – Code samples are invaluable source of knowledge. I tried to include references to some of the most important. Some detailed codes are available in the tutorial by Klaus. Samples differ in level and complexity and are usually written by users and are published for all in the Basic4ppc forum under the share your creations section.
- **How to learn from this chapter**

- If you are a beginner, or if you have no experience in creating GUI, you should read this chapter entirely and run the samples.
- If you have experienced GUI creating and graphics under Windows OS, skim through and see what you need. Sections are independent but previous knowledge is assumed.

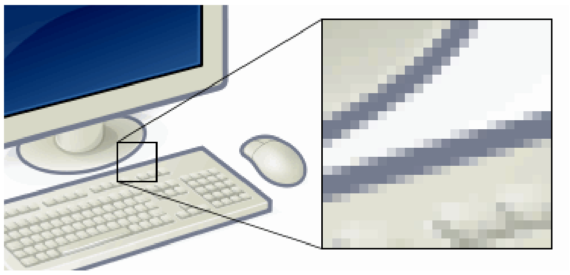
Fundamentals of graphics and GUI

Graphics basics

This chapter is intended to give background for graphics programming. It is not intended to be a complete graphics tutorial or a comprehensive guide, just to make sure you know the basic terms.

<u>Pixels – the basic unit</u>

A pixel is generally thought of as the smallest single component of a digital image. When displaying anything on the screen the computer uses small dots different colored light. These dots are arranged in rows and columns and when you display an image, the computer turns on with the right color the right pixels so that the image appears properly:



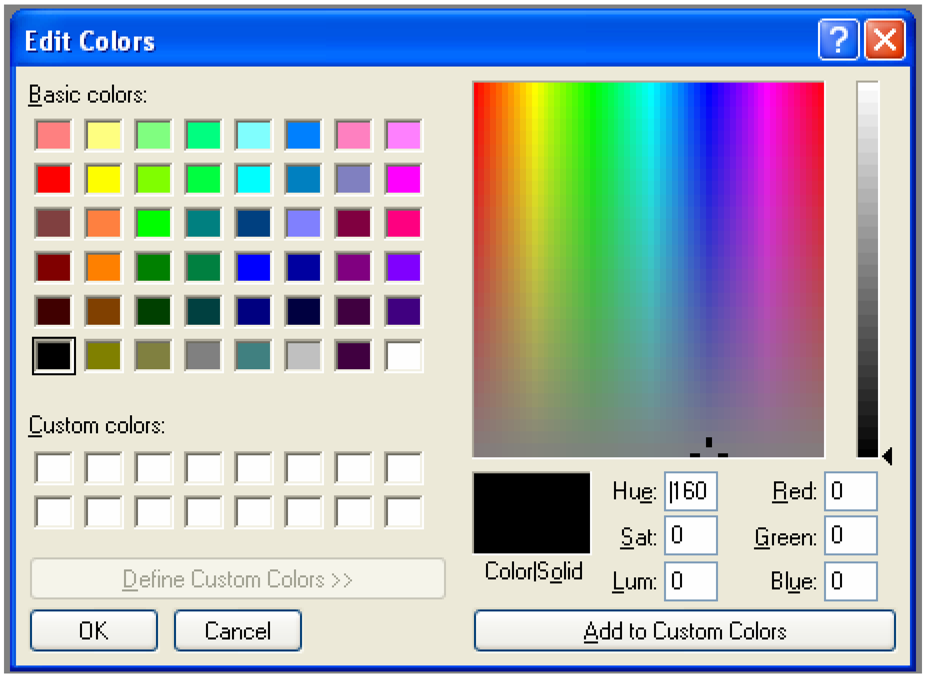
This image, taken from Wikipedia, shows a portion of the image enlarged, so that the pixels are visible

The smaller the pixels are, the better the picture you get. If the pixels are too big you get a pixilated image (such is the enlarged portion above). If they are small, the image appears smother, but smaller.

<u>Color</u>

In order to display color, the computer gives every pixel a color value (expressed as a number) composed of the color intensity in three base colors ("channels") – Red, Green and Blue – hence RGB. Using this method, the computer tells each pixel the intensity level for each channel. For example, if the Red channel gets the maximal intensity, pixels are **shown red**. If both red and blue channels get half the maximal intensity, pixels are **shown purple**. Maximal value is 255, so this result was achieved by setting these values to 128.

You don't have to memorize this: there are various methods to find out the color you need – the easiest and cheapest is to use Microsoft Paint, installed with Windows, select colors – edit colors – define custom colors:



Hue: 160 Sat: 0 Lum: 0 Red: 0 Green: 0 Blue: 0

The values of Red, Green and Blue are shown to the right and you can change them and check out what happens.

The system of color representation used by Basic4ppc (and Microsoft .NET) is by combining the three values with a fourth value indicating the total opacity of the pixel into one number. Each color is represented with an eight bit number, ranging from 0 – 255 (hex 00-FF). this allows you to describe all colors with a single 32 bit number, in the format ARGB where A is Alpha, describing the opacity channel (this channel is not natively supported by the .NET CF. Basic4ppc user eww245 has created a library that adds this functionality and will be described later). The other channels are combined to one number.

The RGB keyword

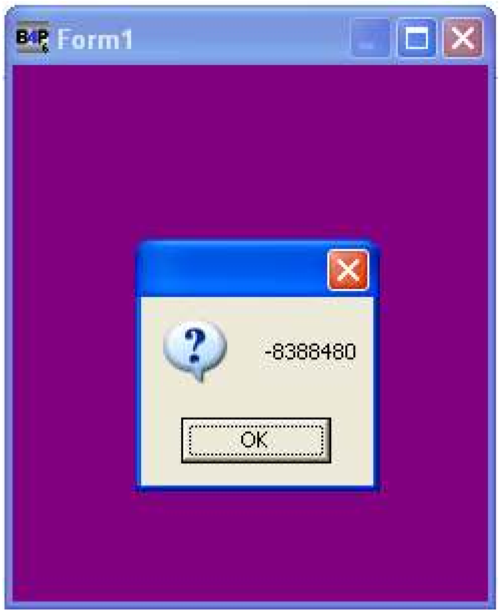
In order to get the number you need from the values of colors use the Basic4ppc Rgb keyword. The syntax is

    Rgb (RedValue, GreenValue, BlueValue)

where the values range from 0 to 255. The result is a number. For instance, with the same values described earlier (R=128, G=0, B=128) and the following code produces the form below:

```
Sub Globals
    'Declare the global variables here.

End Sub

Sub App_Start
    Form1.Show
    Form1.Color = Rgb(128, 0, 128)
    Msgbox(Rgb(128, 0, 128))
End Sub
```

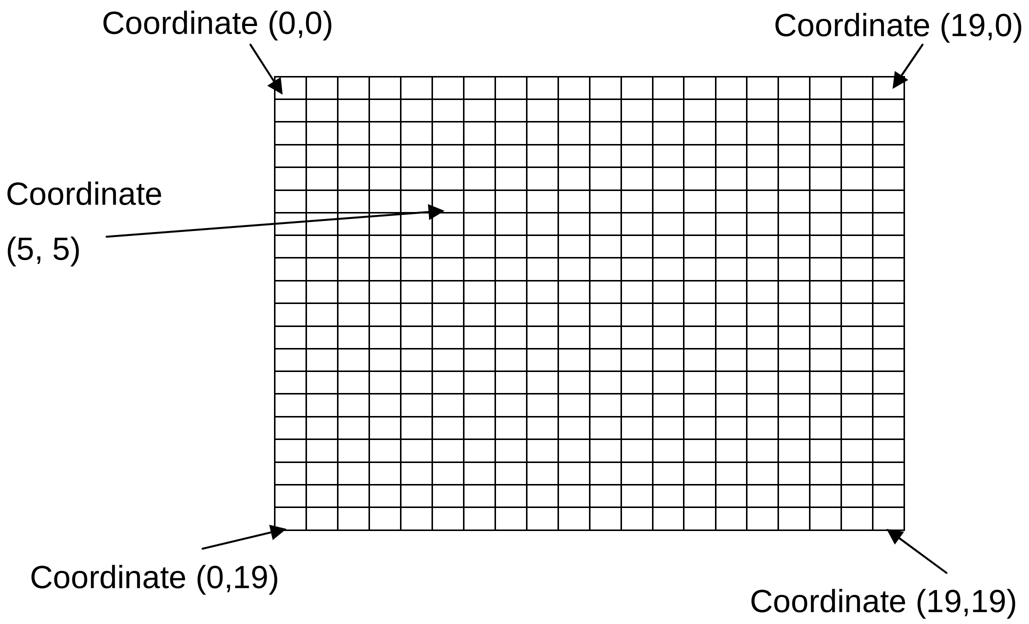
Note especially line 8 and 9. When you run it, you get a form painted with purple with a message showing the color number. Again, you do not have to be able to understand this number – it is enough that Basic4ppc understands it.



The number in the Message Box is nothing you should remember…

Coordinates

The screen is composed of a couple of hundreds of pixels each line and a couple of hundred lines. Typical numbers in desktop computers are 1024 pixels a line and 768 lines total, but this differs (see resolutions later). This way you can reference each pixel with a pair of coordinates. The upper left pixel is (0,0) and the lower right pixel in this example is (1023,767) (note there are 1024 pixels but the rightmost coordinate is 1023). The coordinates are referred to in the format (x, y) where x is the column number and y is the row. The first column and row have the index 0, and on a 1024 columns wide screen the last one (rightmost one) has index **1023** (because we started with 0!). Following is a small example of a 20X20 grid – just to give you the idea:



For example, the method **Line** draws a line on a form between a pair of coordinates.

The syntax is

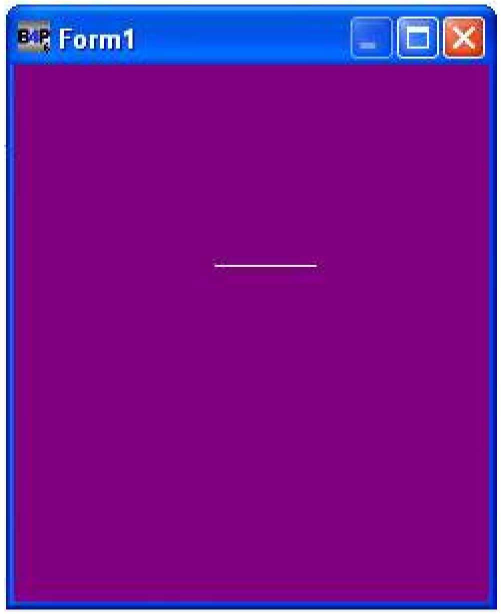
Line (x1, y1, x2, y2, color [optional additional parameters – see later])

This code is a slight change of the previous:

```
Sub Globals
    'Declare the global variables here.

End Sub

Sub App_Start
    Form1.Show
    Purple = Rgb(128, 0, 128)
    White = Rgb (255, 255, 255)
    Form1.Color = Purple
    Form1.Line(100, 100, 150, 100, White)
```

```
12  End Sub
```

There are two interesting changes here: Line 11 draws a line on the form in the 100th line between the 100th and the 150th column:



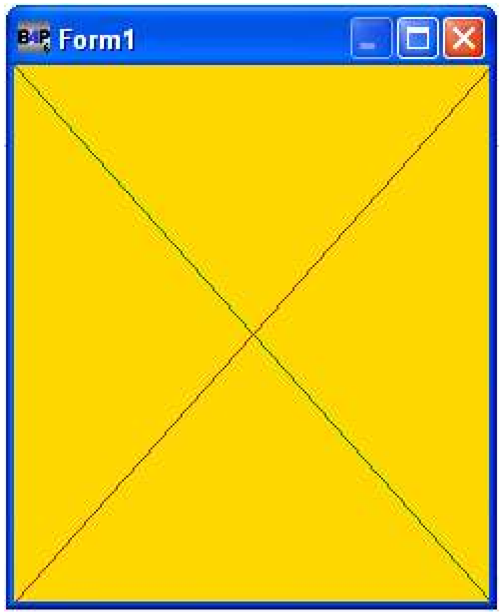
A tip about colors: I assigned the values returned from the Rgb keyword to variables containing them for easier usage (and future usage). This is so effective that the main colors are already predefined in Basic4ppc as constants with the name of the color preceded with "c": cRed, cWhite, cBlack and so on. So instead of defining myself Purple = Rgb… and so on, I could have used cPurple which is predefined.

This is another example of using coordinates in code:

```
1  Sub Globals
2      'Declare the global variables here.
3
4  End Sub
5
```

```
Sub App_Start
    Form1.Show

    Form1.Color = cGold
    Form1.Line(0, 0, Form1.Width, Form1.Height, cGreen)
    Form1.Line(Form1.Width, 0, 0, Form1.Height, cBrown)
End Sub
```

This demonstrates the usage of the form **Width** and **Height** properties in graphics methods.



Screen Resolutions

The changes in screen sizes and pixel sizes led to different amount of pixels displayed on the screen, with different density. The term “resolution”, although claimed sometimes to be formally defined different, usually refers to the dimensions of the screen in both aspects: a resolution of 1024 X 768 is what described earlier. Typical **desktops** resolutions are:

Name	X (columns)	Y (rows)
XGA	1024	768
WXGA	1280	768
WXGA	1280	800
SXGA	1280	1024
WSXGA	1680	1050
HD-1080	1920	1080
WUXGA	1920	1200

Typical **Mobile phone\device** resolution

Name	X (columns)	Y (rows)
VGA	480	640
QVGA	240	320
WVGA	480	800 *see below

* a sample question from the forum about this screen res. with bList: Working with a non-standard resolution – WVGA with bList and AutoScale http://www.basic4ppc.com/forum/questions-help-needed/5490-problem-blist-wvga.html#post32357

Note, that the two upper resolutions are the one that were traditionally in use by Window Mobile devices (both with and without phone) and are called VGA and QVGA (Quarter VGA). The QVGA is the default resolution of a new form you create with the designer. The AutoScale mode compiles application so that they look almost the same on both. However, modern phones support more and more different resolution (for example, the Samsung Omnia has 240 X 400. This is not any standard resolution). Applications written for screen resolutions such as 480X640 will appear with an empty rectangle at the bottom of the screen when run on a bigger screen. Of course, you can avoid it if you want by checking the actual screen size you have when the program starts and adjust locations of your controls and graphics respectively.
This code sets Button1 position to the lower left corner.

```
Sub Globals
    'Declare the global variables here.

End Sub

Sub App_Start
    Form1.Show

    Button1.Left = 0
    Button1.Top = Form1.Height - Button1.Height
End Sub
```

The tables above are used to decide the right place to place things on the screen when you program for an unknown resolution. This is often the case with mobile device, and sometimes you need to adjust the forms you design to fit the screen.

Find the resolution you need

If you are using Basic4ppc to write programs for a mobile phone with specific screen resolution, you might want to consult the list of screen resolutions for mobile phones below. This is a link to a website that from which users download mobile phone wallpapers – the list is there for the users' benefit: http://cartoonized.net/cellphone-screen-resolution.php.

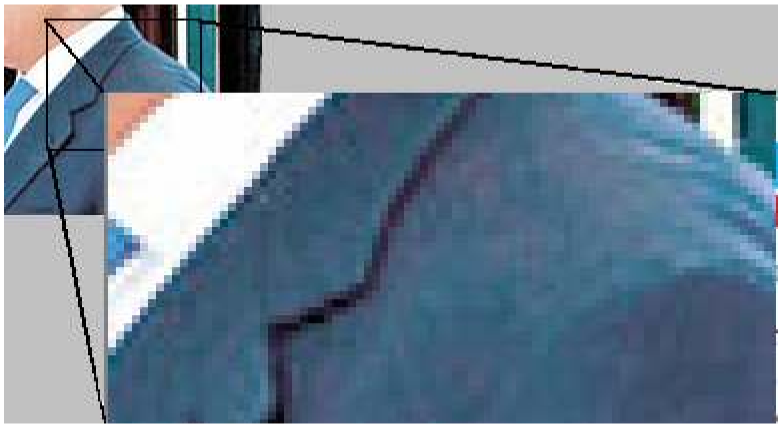
Mouse and Touch Screens

Usually, there is not much to know about mouse and touch screens when dealing with graphics, but it is important to understand the concept. The basic input unit we are all used to is a mouse, or a replacement of a mouse. The mouse is a pointing device – it's basically a way to keep track of a certain position on your screen. The mouse cursor is displayed on a specific position (x, y). When you click the mouse button, the system knows it should take a certain action with respect to the current location of the mouse cursor. It checks what's there at the same location on the screen currently and (usually, on a .NET, Basic4ppc and similar systems) sends a message to the object underneath, expecting it to **handle the mouse click event.** This terminology explains the term "event" used to "acknowledge" a software component of something that occurred.

When using a touch screen, the same idea is applicable, only the system does not keep track all the time of the place where the cursor is. The "click" event raised normally by the mouse is replace here with a stylus or a finger touch, but apart from this it is exactly the same – the system gets informed by the hardware (touch screen instead of mouse) of the location of the touch and raises the click event the same way.

Images

The last concept you should be familiar with is the way a computer analyses and displays images. As we said, an image is composed of many pixels, set in proper order one next to the other. This forms the layout of the picture as you can see below, as an example:



The jacket on the original picture is enlarged so that you can see the pixels. Each individual pixel has to be stored as a number representing the color. This consumes memory and disk space (depends on where you save the image to). An image is one of the most memory-consuming media types, so various methods have been invented in order to reduce the memory consumption. These methods differ in some aspects, the most important of which is the amount of size reduction they achieve and the loss of quality resulting. Usually, these methods are called after the files extension they are associated with. The most common are:

- BMP: the basic Windows image format, supported by most Microsoft products. Applies no compression hence results in very big image files. On the other hand, very simple to handle.
- JPEG – a lossy compression format, widely used and producing relatively small files. On the other hand, usually causes some quality loss (hence "lossy") which differs with compression ratio.

- GIFF – a lossless compression format that supports animation within the image file itself (a unique feature not to be found in almost any other common format). On the other hand, supports relatively few colors and best compresses when there are large areas of the same color.
- PNG is another common format, with compression based on both GIFF and JPEG. It can be lossy or lossless, and can support translucency, though not every platform can display it. It produces relatively small files and is common over the web along with JPEG.

This lists just the most common raster formats – that is, pixel-mapping-based. There are few other vector formats, but they are out of the scope here. The important thing is to know that images occupy a lot of space – this means that handing them without considering the effect on memory might cause problems: lack of memory, lack of disk space (when saved to disk), and communication slowdown (when sent over the internet).

Creating UI with Basic4ppc

Why GUI

You may be wondering, since everything is eventually graphics, why do we start with teaching you how to create GUI using controls, which naturally limit your flexibility, rather than give you everything you need to create a free graphical GUI using plain graphics. There are good reasons:

- **Wrapping "low level graphics"**: controls do the work for you by saving you the time to adjust your graphics. Instead, they translate it to terms of locations on forms, width and other human understandable properties rather that just x and y.
- **You could draw yourself** but this is very complicated to optimize and program.
- **Events, properties, standardization** are all integral part of controls. They support events and properties you know already, and they are standard over every program thus ensuring your users can predict how you program reacts with them.

Forms

The basic control, to which you have already been acquainted, is the Form. The form is said to be the "Parent" of all controls. Every window in Windows is a form. This is the only control you can draw on in Basic4ppc without using external libraries. It serves as a container to your user interface. On a mobile device, where windows have only the size of the screen, it employs the entire screen your user sees at a given time – in short, it is a very important control.

The best way to get a feeling is to add a form to a Basic4ppc program, like you did many times before. Look at the designer chapter if you can't remember how, and if you can just add a form called frmMain to a new application. This is the form we shall start with in the following chapter.

The way this chapter teaches you about forms is by creating an example. If you follow the entire example, you will get a thorough understanding of the basic things forms let you do. Yet, since actually almost everything has a form related to it (if the user has to see it, it has to be on a form), this will only give you the basics – next chapters will enhance your knowledge and understanding. However, this section is essential in order to be able, either now or later, to achieve more knowledge about graphics and UI in Basic4ppc, and about working with the libraries (shown later) that use forms. Though the sample program is long, it is recommended to follow it entirely. It covers the following fundamentals:

- **Basics**: Opening, closing, placing controls and using common events
- **Basic graphics**: lines, shapes, pixels, images
- **Using the fore layer** – a unique feature Basic4ppc supplies to draw in two layers
- **Runtime controls manipulation** – adding and using controls and subs at runtime – a very important, fundamental topic that is one of the most powerful features of the language.

Note the next chapter, Advanced forms operations, adds some advanced yet fundamental features you can't afford skipping….

Eric the Camel

Since we are dealing also with images, we shall use the assistance of our home camel, Eric. You can either chop off the image from this guide (electronically), or use the image of your own camel – as long as there is an image named Camel.jpg on the application folder, we are good.

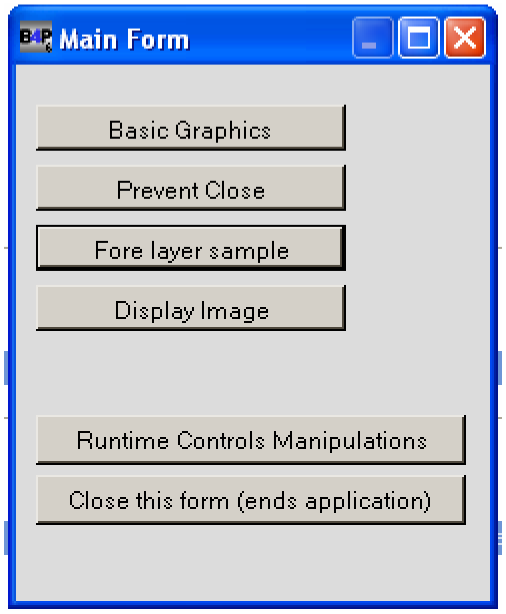
Source code

Source code for the entire sample program is at the end of the chapter. But first, we describe the basic concepts one at a time.

Source code is also downloadable from the Basic4ppc website, so that you have all the designer-generated parts as well: http://www.basic4ppc.com/files/Form_Sample_2.zip

The main screen

The main form (screen) this application has is a form with some buttons on it. It looks like this:



The purpose is to create a simple way to choose what you want to see – a primitive menu that is created easily (recall the menu designer described in the designer chapter). Each button demonstrates a different part in this application. The buttons refer you to what this small program does (see details and code below):

1. Basic graphics
2. Prevent close – demonstrates how to handle the close event of a form.
3. Fore layer demo.
4. Image display technique.
5. Runtime control manipulation.
6. Close the form. When closing the main form, the application ends.

- On the main form there is one invisible control – an imagelist called ImageList1. This is an invisible list of images and it's discussed later in details.

Parts in the application that are related with this form are the Globals sub, App_Start, and the click events of the buttons.

```
2   Sub Globals
3           Dim xPolyArr (0)                        ' x coordinates for sample polygon
4           Dim yPolyArr (0)                        ' y coordinates for sample polygon
5           strPreviousName = ""            ' stores the pressed control name
6           Dim ControlsOnForm (0)                  ' an array to get all controls
            flgGettingPixel = False         ' a boolean indicator - set to true
7                   'if we are in "get pixel color" mode
8   End Sub
9
```

Sub Globals sets values for some variables – they are explained later.

```
10  Sub App_Start
11          frmMain.Show
12
13          ' set the values for global vars
14
15          ' x coordinates for sample polygon
16          xPolyArr() = Array (20, 200, 10,  105, 200)
17          ' y coordinates for sample polygon
18          yPolyArr() = Array (40, 10,  200, 240, 200)
19
20  End Sub
```

Sub App_Start:

- Line 11 shows the main form.
- Line 16 sets values to the array that holds the coordinates of a polygon. These are only the x coordinates, as the name implies. Y coordinates are set on line 18.

```
21
22  Sub randomColor                         ' create randomal colors
23    Return Rgb (Rnd (0, 256), Rnd (0, 256), Rnd (0, 256))
```

```
End Sub
```

Sub randomColor is a small utility sub that returns a random color when called. It uses two interesting functions:

- Rgb (described in the introduction) returns a number representing a color when you give it the ingredients of Red, Blue and Green, in the range 0 – 255.
- Rnd (short for random) returns a random number in the range you specify.
- The combination creates a number that is a blend of three random ingredients.

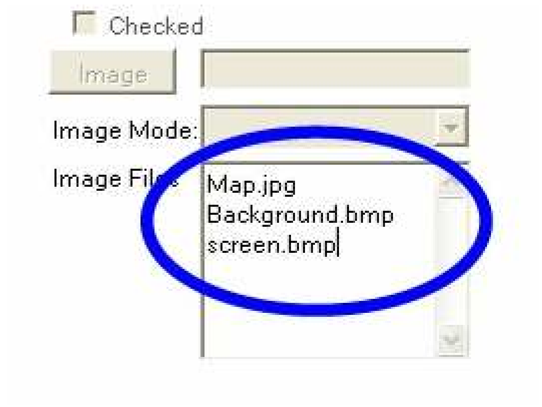
```
Sub cmdImage_Click
        frmImage.Show
End Sub

Sub cmdShapes_Click
        frmShapes.Show
End Sub

Sub cmdClose_Click
        frmCancelClose.Show
End Sub

Sub cmdCloseApp_Click
        frmMain.Close
End Sub

Sub cmdRuntime_Click
        frmRuntime.Show
End Sub
```

The methods (events, actually) above serve as general referrals to the relevant form – the names let you know which is which (the form that show "Basic Graphics" is called frmShapes).

ImageList

The main form (frmMain) contains one control you have not yet met. This is the ImageList. An ImageList is basically just a list of images, loaded from a file. What is the great need for a list of images, if you could have just loaded each from a file when you need it? Well, there are good reasons to use it.

- When an image is loaded to an image list, it is kept there and there is no need to reload it when you need it again. If all your buttons have an image of a shining gradient (this is how it's done many times), you wouldn't like to load each at a time, when it's the same image – this takes a lot of time. It's much better to load it into an ImageList and use it from there.
- ImageList gives you a place to store images. You should get some experience with graphics so read on – it is explained under the Image showing form sub-section.
- Interactions with other applications: image lists store images in a standard way (the .NET image object). This way, you can interact with external functions that expect to get a memory piece that has the standard representation of an image.
- Image lists let you load images of various file types.
- Images stored in image lists at design time (through the designer) are placed <u>inside the application file</u> when it is compiled. They are then treated by operating system as <u>resources</u>, the general name for media and data that is deployed in the same file as your application does. This has many advantages – the main one, of course, is that your user can never have a missing file when it's placed inside an ImageList at design time. Note, however, that you can populate the list at runtime as well, from external files: there is no way to guarantee that they are there when you then look for them. The picture below shows the way to specify images at design time, in the visual designer:



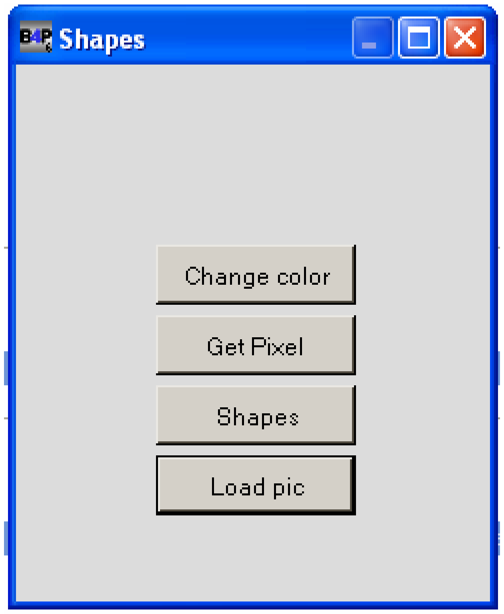
Map.jpg
Background.bmp
screen.bmp

-

The ImageList control is invisible. Its only usage is to serve as a dynamic list, a container for other controls or usages. The ImageList in the sample is called ImageList1 – the default name.

<u>Basic Graphics</u>

When you click the Basic Graphics button (its name is cmdShapes), the form named frmShapes appears:



The buttons do these actions:

- Change the form's color.
- Get the color of a pixel the user touches.
- Draw some shapes on the form
- Set a background picture.

<u>Change the form's color:</u>

```
Sub cmdColor_Click
	frmShapes.Color = randomColor
End Sub
```

The Form.Color property sets the color of the form. Gradients can be achieved with either images or additional libraries.

- Get the color of a specific pixel on the form

```
Sub cmdGetPixel_Click
	flgGettingPixel = True
	Msgbox("Next time you touch" & CRLF & "the screen, the pixel's" & CRLF & "color # will appear")
End Sub
```

```
244
245 Sub frmShapes_MouseDown (x,y)
246         If flgGettingPixel Then
247                 Msgbox("Pixel: " & frmShapes.GetPixel(x,y))  ' x, y from the event
248
249                 flgGettingPixel = False
250         End If
251 End Sub
```

The way to get the color of a given pixel is very simple: you call the form’s GetPixel function with parameters x and y (indicating the coordinates on the form, of course. Yet, note that it takes two subs to do so in the sample. The reason is that there is no way to know which pixel the user meant to indicate for color checking when he clicks the Get Pixel button. So, instead of responding immediately, we display a message saying the next click yields the clicked pixel color, and we then set a global flag (Boolean variable) to true (lines 241 – 242). Then, we wait for the next click on the form. Any click on the form raises the Mouse_Down event, and we handle this event in line 245. The first thing we do is to check if the flag is on: remember that the event is raised every time the user touches the screen when this form is shown, not necessarily with any relation to the Get Pixel button – so we must check. If the flag is on, this means the user has just clicked the button and is willing to see the color of the pixel. The Mouse_Down event supplies you with the X and Y coordinates via two parameters with these names. We use them in line 247 to display the color. Of course, the displayed number has little, if at all, meaning for any user – it can be used for color manipulation inside the program when you need it. The last thing we do in line 249 is to set the flag to false, so that the user does not get a color each time he touches the screen.

Draw some shapes

```
52  Sub cmdShowShapes_Click                          ' draw some sample shapes
53          frmShapes.Line(0, 0, 50, 50, cWhite)
```

```
54          frmShapes.Circle(50, 50, 30, randomColor)
55          frmShapes.Polygon(xPolyArr(), 0, yPolyArr(), 0, 5, randomColor)
            frmShapes.DrawString("Sample String. This is drawn right on the form.", 12, 20, 60,
56  frmShapes.Width - 20, 60, randomColor)
57  End Sub
```

- Line 53 **draws a line** on the form – it uses the line method. Parameters are X1, Y2 (starting position), X2, Y2 (ending position) and color.
- Line 54 **draws a circle** – parameters are X, Y (center point), radius in pixels, and color. Additional parameter, omitted above, is simply the letter "F", indicating "Full" shape – this draws a full shape.
- Line 55 **draws a Polygon** – a shape with an arbitrary number of lines. This is done by supplying two 1-dimensional arrays to the functions, X's and Y's, where each pair of numbers indicate the position of the next vertex: the first item in the x's array is the x position of the first vertex, where the first y-item is it's matching y position. The parameters are: X-array, and then X-array-item-to-start-from (0 in the sample), Y-array, Y item number, number of vertices (5 in this example), and color (random color is used again). Note the way by which we set values to the arrays in sub App_Start, and the way we pass arrays as parameters to methods and functions (with the parentheses following the name).
- **DrawString** is the last method shown here. This is used as a basic method of drawing text to the screen – actually, everything in the operating system that draws text on the screen uses this as an underlying function. This method is quick and has no over head. On the other hand, it is plain graphics – you cannot treat it as if it was a control, you cannot move the text and if you resize the form it is not resized – it is plain pixels. Use it when speed is the main consideration. Parameters are the text, font size, location and color.
- Note that drawing ability is extended significantly with Andrew Graham's (Agraham) ImageLibEx library. It will be discussed later.

Load a picture to the background of the form

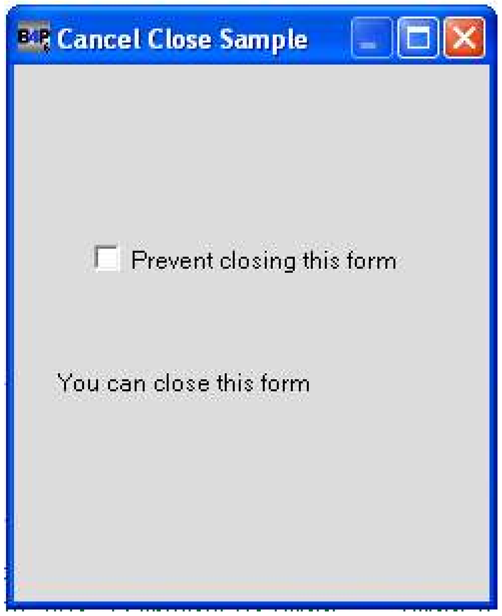
```
Sub cmdLoad_Click                          ' loads an image to the form's background
            frmShapes.LoadPicture("camel.jpg")
     ' equivalent to: frmShapes.Image = ImageList1.Item(0)
            ' the difference is that with ImageList the image is
            'loaded on app start
End Sub

```

The Form.LoadPicture method is used to load a picture from a file and set it to be the form’s background picture. A picture that is set to be the background has special meaning for some controls, especially some of these with transparency. There are some older controls still in frequent use that show transparency by copying what appears to be the form’s background in the transparent places. These controls (for instance, the ImageButton, very commonly in use) may be “fooled” if you place the background in a different way (for instance, if you used the Image control rather than setting image as background, an ImageButton placed on top of the Image misbehaves).

Prevent Close

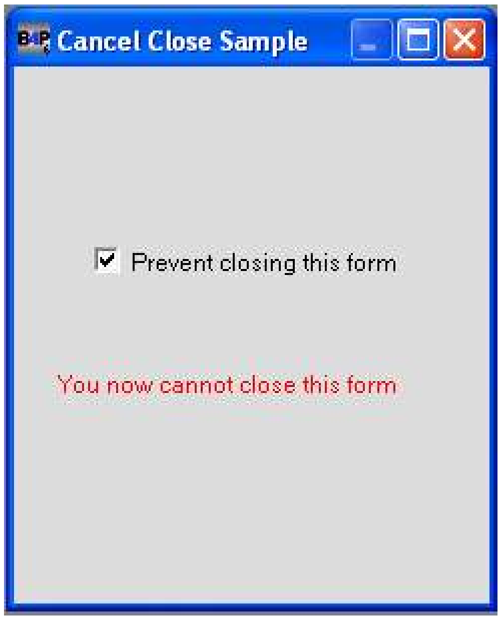
This button takes you to a very simple form with a label and a checkbox on it. This form is called **frmCancelClose** and the program handles its **Close** event.

It looks like this:



You can close this form

and when you check the checkbox it looks like this:



You now cannot close this form

and then, you cannot close the form.

This is the code (the form is opened in line 40 shown above):

```
75  Sub frmCancelClose_Close
76          If chkCancel.Checked Then frmCancelClose.CancelClose
77  End Sub
78
79  Sub chkCancel_Click
```

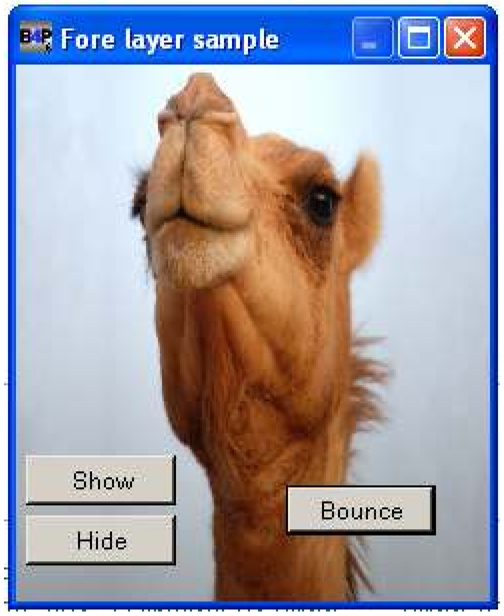
```
80          If chkCancel.Checked Then
81                  lblMsg.Text = "You now cannot close this form"
82                  lblMsg.FontColor = cRed
83          Else
84                  lblMsg.Text = "You can close this form"
85                  lblMsg.FontColor = cBlack
86          End If
87  End Sub
```

The first sub, in line 75, is the Close event of the form frmCancelClose (recall that you know it's an event by the name convention with the underscore!). As its name implies, it is called prior to the final closing of the form. The sub checks (line 76) if the checkbox is checked. If it is, it cancels the closing of the form using the Form.CancelClose method.

Sub chkCancel_Click is a second event. This one here is associated with "chkCancel" – this is the checkbox in the middle of the form, of course. The click event is "fired" when the user – surprise – clicks the checkbox. In this sub we show the label with the proper color and text – it is quite self explanatory.

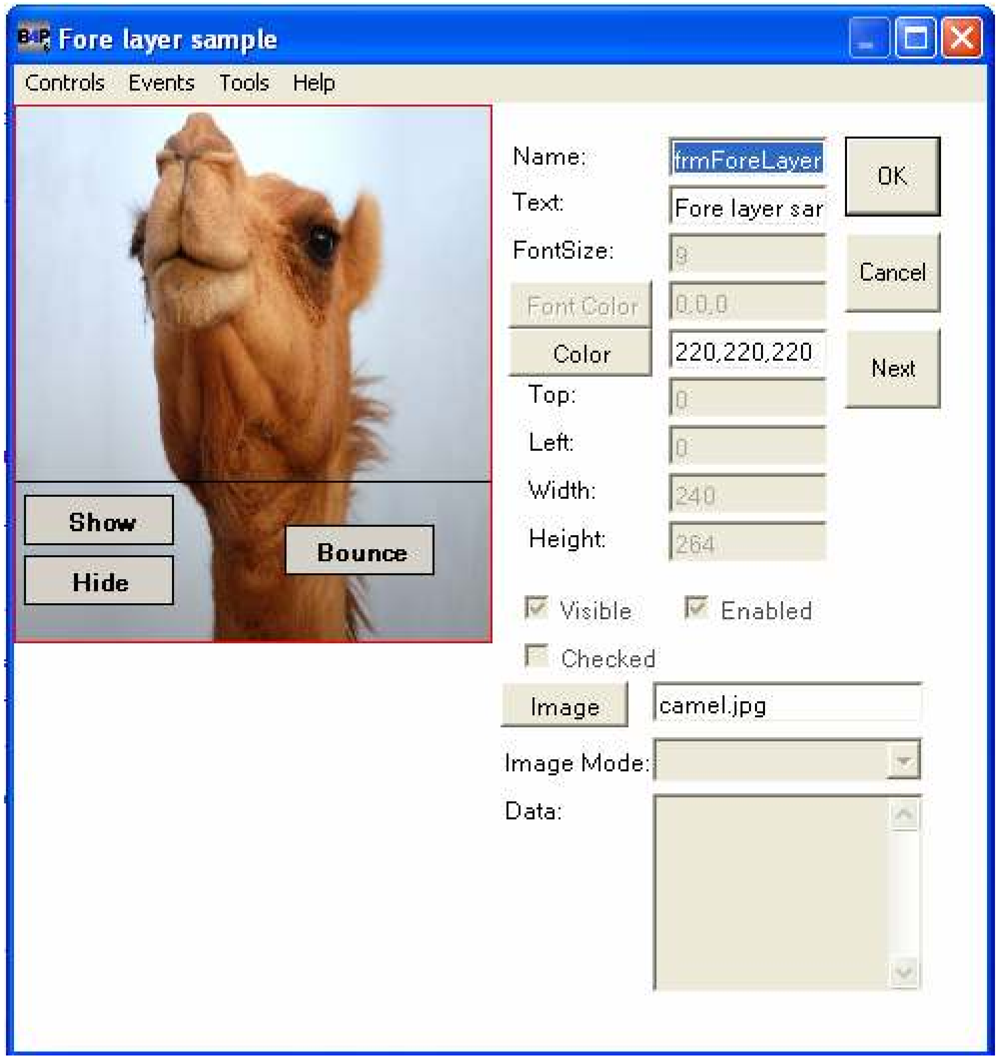
Fore Layer

The fore layer is a Basic4ppc unique feature that gives you an ability to easily draw graphics on a background, without hurting the background. The main advantage is that the form layer has transparency – define a color, and it appears transparent when drawn on the fore layer. This is what the form looks like:



It has three buttons: Show – demonstrates drawing on the fore layer. Hide, which hides what was drawn on the fore layer. And bounce, that bounces a ball.

One more thing this form demonstrates is how to place a picture to the form's background at design time. In the designer, the form looks like this:



Name: frmForeLayer
Text: Fore layer sar
FontSize: 9
Font Color 0,0,0
Color 220,220,220
Top: 0
Left: 0
Width: 240
Height: 264
Visible
Enabled
Checked
Image camel.jpg
Image Mode:
Data:

Note the Image button to the middle right of the form, where the word Camel.jpg is written. This is the file that holds the picture of Eric, our home camel, of course.

Working with fore layer involves these simple steps:

- Use the SetTransparentColor([color name]) to set the color that will be shown transparent when drawn on a fore layer. In the sample, I set it to cWhite – the system **c**onstant **White** color (actually a number you can get with Rgb(255, 255, 255) – check it out). Now, drawing in white on the fore layer in white will erase what's on the fore layer, and show what's underneath. This must be done prior to the next step:

- Set Form.ForeLayer to true. This allows you to draw on the fore layer.
- Use one of the Fore Layer functions to draw on the form. You identify them by the letter F prefixing them:
    - FLine is identical to Line.
    - FCircle, FPolygon, FDrawString, FDrawImage and FGetPixel all have the same functionality as their respective methods with no F.
    - FErase erases a given rectangle – identical to drawing the same rectangle with the transparent color.

The following code does just this:

```
Sub cmdFore_Click

        SetTransparentColor(cWhite)
    ' White is now transparent on forms with forelayer=true.
    'You must call this function prior to forelayer=true.


        frmForeLayer.ForeLayer = True

        frmForeLayer.Show
End Sub


Sub cmdShow_Click
        frmForeLayer.Circle(20, 20, 10, randomColor, F)
        frmForeLayer.Circle(40, 70, 10, randomColor, F)
        frmForeLayer.Circle(70, 40, 10, randomColor, F)
        frmForeLayer.Circle(120, 220, 10, randomColor, F)

        frmForeLayer.FLine(0, 0, 50, 50, cWhite)
        frmForeLayer.FCircle(50, 50, 30, randomColor)
        frmForeLayer.FPolygon(xPolyArr(), 0, yPolyArr(), 0, 5, randomColor)
        frmForeLayer.FDrawString("Sample String. This is drawn right on the form.", 12, 20, 60,
frmForeLayer.Width - 20, 60, randomColor)
End Sub

```

```
116 Sub cmdHide_Click
117         frmForeLayer.FErase(0, 0, frmForeLayer.Width, frmForeLayer.Height)
118 End Sub
119
120 Sub cmdBounce_Click
121         y = 0                                   ' ball start y
122         speed = 10                              ' ball start speed vector down
123
124         Do While speed <> 0
125
126             frmForeLayer.FCircle(100, y, 30, cBlue, F)  'on fore layer
127
128             DoEvents ' force drawing in loop. Omit this - app. stucks!
129             Sleep(100)                          ' delay
130
131             If speed > 0 Then                   'way down
132                         speed = speed * 1.1         'accelerate
133             Else                                'way up
134                         speed = speed * 0.8         'decelerate
135
136                         If Abs(speed) < 2 Then
137                                 speed = 0
138                         End If
139             End If
140
141                 If y > frmForeLayer.Height - 30 Then    'change direction
142                         Speed = -speed
143                 End If
144                 'delete ball - use the transparent color
145                 frmForeLayer.FCircle(100, y, 50, cWhite, F)
146                 y = y + speed                   'change ball position
147
148         Loop
149 End Sub
```

Let’s look at the code:

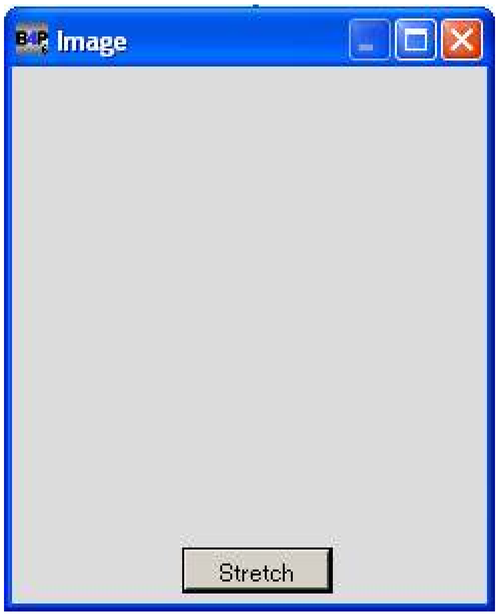
- Lines 93 - 101 are the sub that shows the form. It also performs the first two steps above – set transparent color (to white in this case), and set the fore layer to true.
- Lines 104 – 114 are the sub that is called when the user asks for a fore layer demonstration. Some circles are first drawn on the background, and then some shapes are drawn on the fore layer.

- Lines 116 – 118 are the sub that is called when the user wishes to delete what's on the fore layer. It uses FErase to delete the entire fore layer.
- Lines 120 – 149 are interesting, as they give us opportunity to look at some "real" code. This sub bounces a ball down and up. The X position of the ball is constant: 100. The Y coordinate is changing, but we change it indirectly – we change a variable called **speed**, and add its value to Y. Then, when we set negative values to speed, y actually decreases. We could have changed y directly, but changing the speed lets us control, surprisingly, the speed by which Y is changed, what creates the bouncing effect.
    - Lines 121 – 122 set initial values to y and speed. Speed is indicated in the "down" direction – thus, negative values create upwards movement.
    - Line 124 starts a loop, going on while the speed is not 0.
    - Line 126 draws a ball on the fore layer (FCircle).
    - Line 128 calls DoEvents. This prevents a system crash or stuck that might occur when you are in the middle of a loop. Another thing this initiates is the redraw of the ball: the FCircle (as well as any other drawing method) is only an order to the operating system. The operating system understands a drawing is requested and when it allots time to the graphical engine it redraws. But when in a loop, no time is allotted to anything else but the loop and some "life crucial activities". Try to omit the DoEvents and see – nothing is displayed.
    - Line 129 holds the system for 100 milliseconds. This slows the animation a bit – otherwise it's too fast.
    - Lines 131 – 139 calculate the speed of the ball in the next step. Nothing very complicated about it – on the way down the ball accelerates 10% a step, on the way up it decelerates 20% a step, and if the speed is less then 2 it is set to 0.

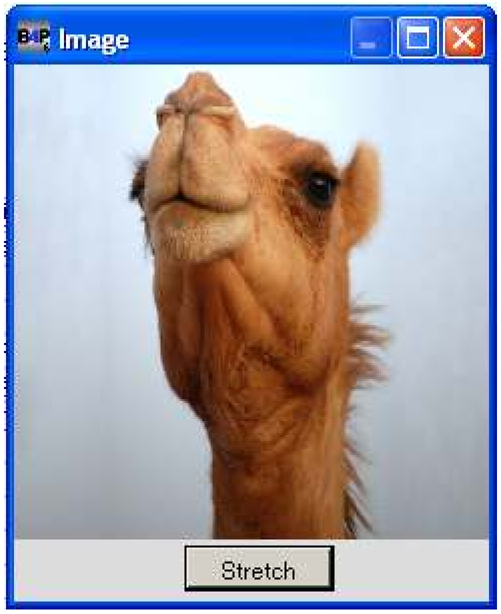
- Lines 141 – 143 change the direction of the ball when it reaches the bottom.

<u>Displaying an image to the screen</u>

The fourth button of the main form shows different ways to display image on the background. The main interest in this code is in lines 59 – 73 – Sub cmdStrech_Click. The form looks like this when you first open it:



and when you press the Stretch button the picture of Eric is stretched from the upper left corner (pixel 0,0) to fill the screen.



This is what you get eventually. Let's look at the code and then discuss some important issues:

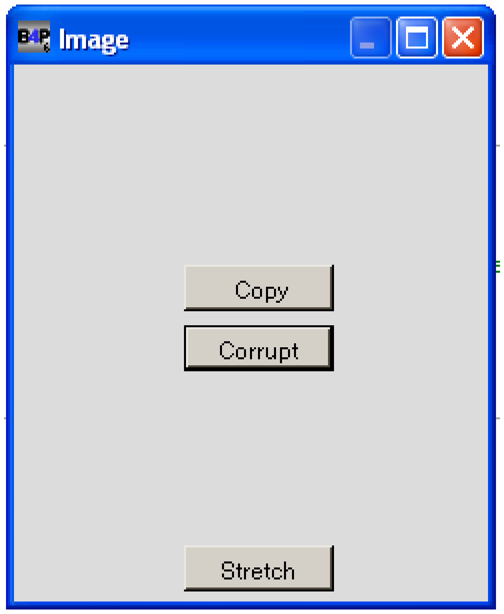
```
59  Sub cmdStretch_Click
60          ErrorLabel(errImage)            'mainly in case the image is corrupt
61          j = 0
62          i = 0
63          Do While i < frmImage.Width
64                  frmImage.DrawImage(ImageList1.Item(0), 0, 0, i, j)
                    DoEvents        'refreshes the form: allows the operating
65              'system to actually draw the changes.
66                  i = i + 1
67                  j = j + 1
68          Loop
69          Return              'avoid the error label
70
71          errImage:               'something's wrong - announce the user
72          Msgbox("Image error")
73  End Sub
```

The event is fired when the button is clicked.

- Line 60 sets an error label.
- Lines 61, 62 set initial values to i and j.

- Line 63 – 68 is a loop that enlarges the picture:
    - First, the picture is drawn at the current size, defined from the upper left corner (0,0) to the current values of i and j. We use "DrawImage" to do this
    - Then, the values of i and j are increased by 1, and the loop goes on.
    - Line 65 is very important: it calls the DoEvents method which orders Windows to check if any other events should be triggered. Read the "**Events may be asynchronous"** paragraph below for an important discussion about this issue.
- Line 69 avoids getting into the error label, defined in line 71 and only shows an error message.

There are some important issues you should be aware of when you deal with graphics. Now that you have seen the basics of form graphics, this simple image example is a very good opportunity to demonstrate some <u>fundamental</u> concepts about images and Basic4ppc. But in order to do this, first make a change to the program. Add the following buttons to the form, named btnCopy and btnCorrupt. Place them roughly as shown below, in the center of the form:

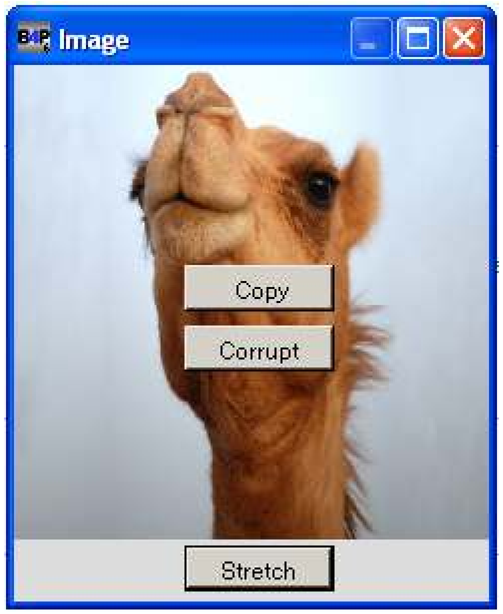


Now, add the following code to the bottom of the program:

```
Sub cmdCopy_Click
        frmMain.Image = frmImage.Image
End Sub


Sub cmdCurrupt_Click
        frmImage.Circle(30, 30, 30, cBlue, F)
End Sub
```

Image as graphics is at the background.

Now, run the program again and hit Stretch. This is what you get:



The reason is that image is drawn (with DrawImage) on the background of the form. This means that in the **Z-Order** (the by which things are drawn on the form, the order along the z axis) the image is at the bottom of everything. If you want an image to be displayed on top of something, you should use the **Image Control**, described later in the controls section – anyhow, this is not a usage of image as graphics, but inside a control. When drawn at the background, every other control, including the buttons in the sample, are on top of it.

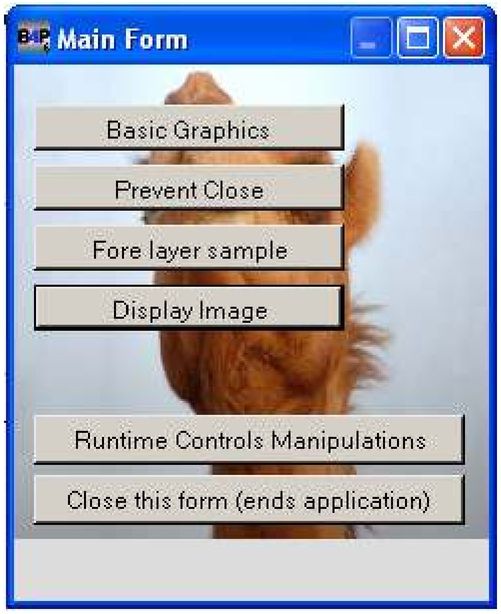
Image is an object and can be set.

Like other “things” (numbers, strings, Boolean values and more), an image is kept in the memory of the computer as an object. Referring the object is possible if a variable holds its value, but in case of an image (and all objects that are not “regular” variables) the variable that holds the “object” (images and other, not discussed here – usually from external libraries) must be of an appropriate “type”. The concept of types of variables is not native in Basic4ppc and anyhow exceeds the scope of this guide. What you should know, is that you can assign the value of an image to an object that can receive images. You create objects like this explicitly with an **Imagelist**, with an **Image Conrol** and with some additional libraries (all described later). You create them implicitly, however, every time you create a form - the Image property is such.

The concept is very simple – you can set an image between two “image objects”, and in our case we use the objects declared for us under each form. It is demonstrated in the small Sub you have added with the cmdCopy button:

```
Sub cmdCopy_Click
        frmMain.Image = frmImage.Image
End Sub
```

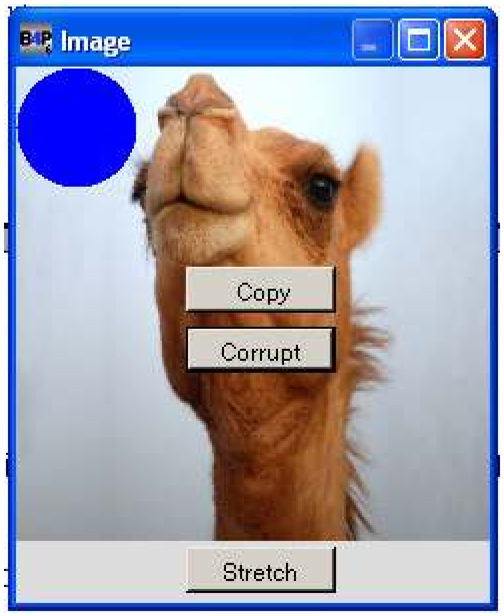
When you click the Copy button, the sub is executed and the Image property of the main form (frmMain) is set to the same value as the Image property of the Image form (the one with Eric!). Try this out – click the Stretch button to display Eric, click Copy, and close the form – the image is copied to the main form!



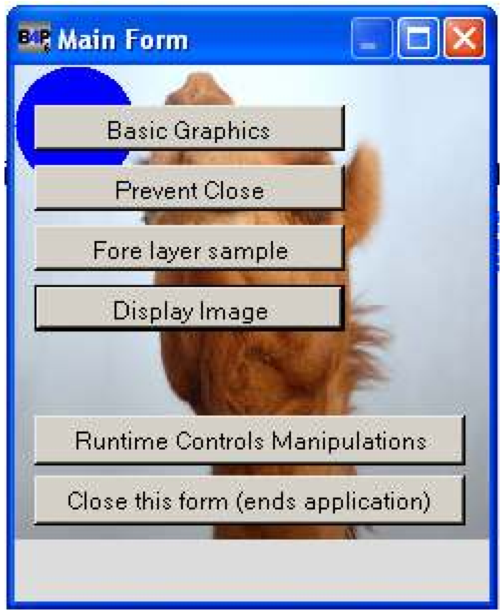
Graphics is drawn on the same "object" as the image. The best way to show this, is to use the "corrupt" button, that fires this event:

```
Sub cmdCurrupt_Click
        frmImage.Circle(30, 30, 30, cBlue, F)
End Sub
```

When you click the button, a circle is drawn on the image:



Now click the Copy button, and copy the image to the main form. Remember, the only thing we do is to copy From.Image property that represents the entire background graphics. Close the form and get the result:



The image is copied together with what you drew on it.

Form is not changed when closed. You have probably noticed this, that when you close the form and re-enter it, the image is displayed as you left it.

Events may be asynchronous.

Events give you significant power being a way to react to something (who's timing may not be predictable) easily. But how is it really happening? The only "real" way to run two different tasks simultaneously is to run them on separate "threads" (A Thread is the basic unit of software Windows handles. Treading exceeds the breadth of this book, but can be implemented in Basic4ppc using the Treading library written by Agraham). But event do not use threading – they run on the main thread. The way windows mimics multitasking here is by utilizing a "**message loop**". This is a loop that is run by Windows for each thread and is repeatedly checking if there are any "messages" it should pass to components that are planned to receive them. Without getting too much into details, you can assume that each control registers itself so that it receives "messages" such as mouse clicks etc. When it gets a message and recognizes it as relevant it triggers the appropriate event.

So the basic idea is this: there is a loop somewhere in your program handled by Windows. If anything of interest (mouse click etc.) happens Windows sends a message to the control where it happened. If nothing happened, Windows keeps checking if anything happened. When a control receives this message, it takes an action – fires the event. The event is the code you wrote in the appropriate Sub. When the Sub ends or if you used the Return keyword, Windows goes back to the message loop and waits there till something new happens. But what if you wrote an endless loop in your event? Or not endless (this is relatively rare) but just something taking too long (this is relatively common)?

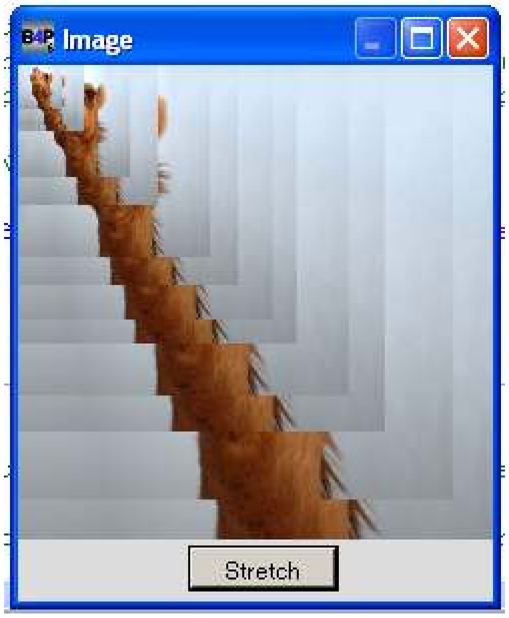
Well, if you did, your program is stuck. It will normally stop being responsive. No other events are fired. No other code is executed. The active statement remains inside the Sub you wrote for a long time (maybe forever) and the message loop is never being called again. It might happen even if you wrote something like the harmless Sub cmdStretch_Click below:

```
59 Sub cmdStretch_Click
60         ErrorLabel(errImage)           'mainly in case the image is corrupt
61         j = 0
62         i = 0
63         Do While i < frmImage.Width
64                 frmImage.DrawImage(ImageList1.Item(0), 0, 0, i, j)
                   DoEvents        'refreshes the form: allows the operating
65             ‘system to actually draw the changes.
66                 i = i + 1
67                 j = j + 1
68         Loop
69         Return          'avoid the error label
70
71         errImage:               'something's wrong - announce the user
72         Msgbox("Image error")
73 End Sub
```

Lines 63 – 68 form a loop that enlarges the image. But the way Windows handles a size change of any of its controls (such a size forces Windows to redraw parts of the screen) is using the Message Loop. When something changes its size a message is sent and some component of Windows gets it and tells everybody to redraw themselves. Obviously, if you just create a loop of this kind changing the size of the image on each iteration. you see nothing on the screen until the Sub ends: the Message Loop is never called.

The **DoEvents** keyword is the solution for this problem. This keyword actually tells windows "Stop everything you are doing right now and run the message loop once: handle every waiting message there. Then, continue here." What happens then is that windows goes back and checks for messages, it finds the message about the size change, and redraws the image.

Using DoEvents is usually an indication that you have created a process which is too long. Good programming practices try to avoid the usage, but it is sometimes useful. Anyhow, note that calling DoEvents inside an event causes events to be asynchronous: An event can start again while the previous action **initiated by the same** event is not complete yet! Generally, when you use it take into account that events can now run in response to some unexpected action (event) happening, and you cannot always predict when, neither their consequences – event when your code is executing. For example, consider the <u>very simple</u> example here: open the form, click Stretch, and before it reaches the final position, click Stretch again. If you do this quick enough, you get this result:



Although nice, this is not always what you want. Particularly, this shows you that the event can fire <u>even when this very same event is executed!</u> In this case there is nothing wrong, but consider a timer control, with a tick event firing once a second, calling DoEvents somewhere: if the operation the event performs is, for example, access to a file, and it takes more than a second, the event might try to access the file when the file is open and get an error, or corrupt the file: this is very dangerous. The way to handle such "asynchronous programming challenges" is to use a Boolean flag (variable) that is set to false at the beginning of the program. Check the value when the event Sub starts and go into the sub only if it's false. When you are in the sub, set it to true, and back to false upon exiting (note error handling). The general structure should look like this:

```
Sub Timer1_Tick
        ErrorLabel(err)

        If flgEventStarted = True Then Return
        flgEventStarted = True

        ' ... some code

        flgEventStarted = False            ' exiting
        Return

err:
        flgEventStarted = False            ' turn false anyway!
End Sub
```

<u>Runtime controls manipulation</u>

Runtime controls manipulation is one of the most important and fundamental parts of Basic4ppc UI programming and object handling. Though it is a somewhat “advanced” topic, getting to understand it is easy and may improve your productivity.

Basic4ppc introduces a unique way of accessing objects without explicitly knowing their names when writing the code. This is a very strong feature that allows you to pass controls as parameters, access unknown controls, save the control’s reference to disk and even call Subs whose names you don’t know when you write the code.

The main idea behind manipulating controls at runtime without knowing their names is that you have certain keywords, that when you give them the control’s name, passed as a regular string (“cmdOK”, for example), they act as if they were the control itself. Such is the Control keyword. For example, instead of writing

```
cmdOK.Color = cWhite
```

you can write

Control(“cmdOK”).Color = cWhite

to the same effect. It seems longer, but is actually very useful, as will be demonstrated here. Starting from version 6.9, you can use the syntax

Button(“cmdOK”).Color = cWhite

as well (of course any control and object can be referenced this way, not only a button). It is shorter and supports AutoComplete.

The simple code below (four lines of code, 154-157, and some error handling wrapping), add as many buttons to a form as you want at once. This is a very simple example, so it’s good to learn from it – on the Basic4ppc forum you can find a more useful example in the runtime controls manipulation tutorial (the tutorial shows how to create a simple notepad with almost no code). The code in this chapter is based on this tutorial, simplifying and explaining things a bit more.

```
151  Sub frmRuntime_Show
             ErrorLabel(errShowing)
152  ' Direct name referencing (AKA runtime controls manipulation) is error prone.
153  ‘ For instance, if you reopen the form with the controls on it you get an error.
154          For i = 1 To numButtonsToRedraw.Value
155                  AddButton("frmRuntime","Btn" & i, 100, 22 * i, 50, 20, "#: " & i)
156                  AddEvent("Btn" & i,Click, "Btn_Click")
157          Next
158          Return
159  
160          errShowing:
161                  Msgbox ("An error was trapped")
162  End Sub
```

Adding controls at runtime – the simplest way

Dealing with controls at runtime means we add them without using the designer. This can be done in one of two ways:

- Add them explicitly – specify full name and location
- Add them implicitly – using some special keywords.

Obviously, we are much more interested in the second way. But in order to achieve this, let's look at the first way – the standard way to add controls at runtime.

The simplest way to demonstrate it is with a small program that just adds a button to an empty form. But assume that we did not add the form via the designer. So we add both form and button. It is so simple, that I omitted Sub Globals:

```
1  Sub App_Start
2          AddForm("Form1", "Adding form at runtime!")
3          AddButton("Form1", "cmdOK", 20, 30, 60, 30, "OK")
4  End Sub
```

- Line 2 adds a form using a new keyword: **AddForm.** There are "AddXXXX" keywords for each control, and for a general "Object" (AddObject – recall, objects come from additional libraries). For our purpose, the AddForm keyword is doing the job by getting the form's name (Form1 in this case: I used the default name. Not very creative, but simple), and a text for the title.
- Line 3 adds a button with the AddButton keyword. Parameters are the name of **the form to which to add the button** (by the way: this could be a panel, another container control), the **location** and **size** (x, y, width, height: see the typing assistant on the bottom of the screen for the list of parameters when you type it) and a **text**. This list is a convention in many **AddXXX** keywords.

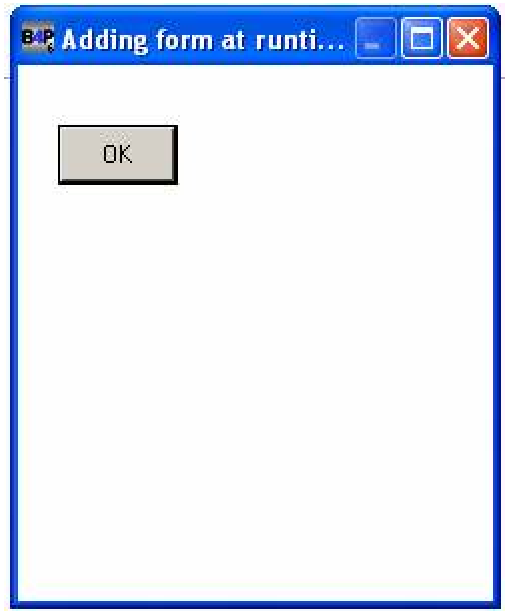
Copy the code and run the program. Nothing happens! No form is shown!

Why? You should be able to answer it yourself at this point.

Well, the reason is, of course, we did not tell Basic4ppc to show the form. Recall that Basic4ppc creates an automatic line **Form1.Show** in every program that you create – but I omitted it here. It is not enough to create the form: you have to show it. Add the line:

```
Sub App_Start
        AddForm("Form1", "Adding form at runtime!")
        AddButton("Form1", "cmdOK", 20, 30, 60, 30, "OK")
    Form1.Show
End Sub
```

And run.



Now you get a form as above. Click the button – obviously nothing happens. No one declared a sub for the Click event of this button!

Adding an event at runtime

There are two ways to add an event:

- If you can cope with the standard names of events in your code (usually the case), use the event name convention: add the sub cmdOK_Click to your code, and it works, for instance:

```
Sub cmdOK_Click
        Msgbox ("Conventional name")
End Sub
```

- If for some reason this name is taken or you are a member of a secret sect vowed to avoid conventional names, declare the event explicitly using the AddEvent keyword. Add this line to the App_Start sub:

```
AddEvent("cmdOK", Click, "WeirdName")
```

Note the irregular way of passing the event name here! There is no variable or a constant called **Click**, but you just write the word here, and if there is an event by this name – it works! Just remember to add a sub with the same name:

```
Sub WeirdName
  Msgbox("Weird name")
End Sub
```

Adding a control the implicit way

The tasks we need to carry out when dealing with controls at runtime are the following (all of them are explained right below the code):

- **Adding** them: we use the **AddXXXX** keywords described above.
- **Referring** a control whose name you don't know is done using the "**Control**" keyword.
- Adding **events** is done using the AddEvent keyword, usually.

- **Sharing events**: an event can be connected to more that one control. In order to find out which control fired the event, use the **Sender** keyword.
- **Deleting** a control is done with the Dispose keyword.
- **Getting the controls** included in a certain form (the "sons" of this form) is done using the "**GetControls**" function
- **Calling a Sub** whose name you don't know is done with the **CallSub** keyword.

Let's get into some details about these tasks.

Adding controls at runtime

As shown above, add controls at runtime with the AddButton keyword. The code below shows the idea:

```
151  Sub frmRuntime_Show
             ErrorLabel(errShowing)
152  ' Direct name referencing (AKA runtime controls manipulation) is error prone.
153  ‘ For instance, if you reopen the form with the controls on it you get an error.
154          For i = 1 To numButtonsToRedraw.Value
155                  AddButton("frmRuntime","Btn" & i, 100, 22 * i, 50, 20, "#: " & i)
156                  AddEvent("Btn" & i, Click, "Btn_Click")
157          Next
158          Return
159
160          errShowing:
161                  Msgbox ("An error was trapped")
162  End Sub
```

- Line 152 declares an error label. Adding controls at runtime is error prone (because you lack the AutoComplete feature) and should therefore be preceded by the ErrorLabel declaration. The label itself is in line 160.
- Line 154 starts a loop with an unknown number of iterations (it depends on what the user specified in the numeric-up-down control named "numButtonsToRedraw").

- Line 155 adds a button the form “frmRuntime”. All buttons are named **Btn1, Btn2, Btn3** and so on, and all have the text **#1, #2….** They are located at column 100 and row starting from 22 and adding 22 to each, this way leaving 2 pixels between one another.
- Line 156 connects all buttons to the same event (“Btn_Click”). The logic was explained above.

Runtime control manipulation basics

Take a look at the Btn_Click sub. This sub is connected to the click event (see line 155):

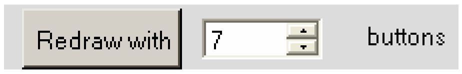
```
164 Sub Btn_Click
165         If strPreviousName <> "" Then ' verify previously pressed button saved.
166
167                 Control(strPreviousName).Color = cGray ' name of previously pressed
168                         ' using the Control keyword: set color to gray
169         End If
170
171         strPreviousName = Sender.Name                   ‘ save the last pressed button's name
172
173         Sender.Color = cRed      ' use the Sender keyword again to set color to red
174                                          ‘note: sender must be an object (anything) with a
175                                  'property "Color", or nothing happens, see next line
176
177         Sender.Name_Of_Some_Non_Existing_Property = cRed' No error-nothing happens
178
            If Msgbox("Do you want to see the control type?", Sender, cMsgboxYesNo, cMsgboxQuestion) = cYes
179 Then
180                 Msgbox (ControlType(Sender))
181         End If
182
183 End Sub
```

- Line 165 – 169 are executed only if the value of a variable called **strPreviousName** is not empty – this variable empty when the program starts and is set when a button is clicked, so the meaning is that the condition is evaluated to true only after a button is clicked once. Then, the name of the button is kept in this variable, and line 167 changes its color back from red (every button clicked is set to red later in line 173) back to gray. Note how we use the Control keyword in line 167 with a string holding a name of a control without knowing which one it is.

- Line 171 sets the value of the variable to hold the name of the button clicked. We use the Sender keyword – **<u>it's a very important one!</u>** There is no other way to know which controls sent this event, when you think about it...
- Line 173 sets its color to red.
- Line 177 demonstrates the easiness by which you can write bugs rather than useful code. Specifying an erroneous property the Sender keyword is as easy as mistyping, and there is no indication something went wrong, apart from a (sometimes minor) property that did not change!
- Line 180 shows how to use the **ControlType** keyword. On the same occasion, line 179 demonstrates how the standard MsgBox returns a value is you ask it to.

Deleting controls at runtime

It is very rare to delete at runtime a control you added using the designer. However, it is not that rare to do so when adding controls at runtime, if you need, for some reason, to add them again. The reason is that you cannot add a control with the same name. Since we demonstrate how you can add a varying number of buttons each time we call the adding sub again and again when you click the Redraw button.



This is the deleting Sub. It uses mainly the Dispose method that every control has by default and that cancels the existence of its owner (hence deleting it from the form where it is displayed).

```
185 Sub cmdClear_Click
186         ErrorLabel(errClear)      ' using direct name referencing is error prone!
```

```
187         For i = 1 To numButtonsToRedraw.Value
188                 Control("Btn"& i ).Dispose        'it's rare to use dispose in Basic4ppc
189         Next
190         Return
191
192         errClear:
193
194 End Sub
```

- Line 186 sets an error label as before.
- The loop in lines 187 – 189 uses the Dispose method without knowing which control it is.
- Line 190 is actually unnecessary, and I wrote it here for elegance and as a good programming practice more than for a good need.

Getting a list of all controls

You can get an array with all the controls of the form with the GetControls keyword. This also works with panels (Panels are containers like form inside a form - see panels next chapter).

```
196 Sub cmdShowControls_Click
            s = "Controls on the form are: " & CRLF
197                         ' CRLF - carridge return, line feed: go to next line.
            ControlsOnForm() = GetControls("frmRuntime")
198         ' note the parameter is a string!! this adds flexibility
199         'and is more error prone.
200         If ArrayLen(ControlsOnForm()) > 0 Then
                    For i = 0 To ArrayLen(ControlsOnForm()) – 1
201         ' always loop to 1 less then length!
202                         s = s & ControlsOnForm(i) & CRLF
203                 Next
204
205                 Msgbox(s)
206         End If
207 End Sub
```

- Line 197 creates the text to be shown. Note the **crlf** global constant used – it is a way to tell Basic4ppc to go one line down.

- Line 198 sets the global array ControlsOnForm (note the syntax, the name should be followed by parentheses when referring an array as a whole) to hold the list of the controls on the form. There are few things to know about the GetControls keyword:
    - Pass an empty string "", to get all the controls available.
    - Note that Timer, ImageList, OpenDialog, SaveDialog, MenuItem and ArrayList controls will not be returned unless you pass an empty string as the argument.
    - GetControls returns the module of each control followed by a period and the control's name.

Calling a sub at runtime

A very strong ability Basic4ppc gives you is to call a sub whose name you don’t know at runtime. This is done with the **CallSub** keyword. It is demonstrated in the sample code when clicking the cmdCallSub button.

```
Sub cmdCallSub_Click
	ErrorLabel(errCalling)

	CallSub (txtSubName.Text)
	Return

	errCalling:

End Sub
```

The importance of this ability is not in this simple scenario – imagine the ability to pass the sub’s name as a parameter to another sub, or to keep a list of changing actions that should be carried out when something happens.

One step further – an advanced word about the Control keyword

This is a basic example so I preferred not to show how to use the Control keyword with **objects** from **external libraries.** Yet, you can use it all the same **event when the control to which you refer is not a native one!** Say the control is a **TreeView** from the **ControlEx** library (a standard library that ships with Basic4ppc. See TreeView the next chapter). And say a variable named strContName holds the name of a certain TreeView. Then you want to change its color to blue. You can use one of the following to refer to it:

- Control (strContName, “TreeView”).Color = cBlue
- **Starting from version 6.9,** you can use the new syntax:

  TreeView(strContName).Color = Blue

  which is shorter and supports AutoComplete.

Although it’s a bit off the concept of this chapter, I’ve included a small sub to demonstrate what I mean. It is important, because it shows a different aspect of runtime controls manipulations – adding them at runtime from external libraries.

```
Sub frmTreeView_Show
		AddObject("tvTree", "TreeView")
		tvTree.New1("frmTreeView", 10, 10, 100, 200)

		tvTree.AddNewNode("1")
		tvTree.AddNewNode("2")
		tvTree.AddNewNode("3")

		strContName = "tvTree"

		Control(strContName, "TreeView").Color = cBlue
End Sub
```

Source code of the sample program in this chapter

```

Sub Globals
```

```
        Dim xPolyArr (0)                        ' x coordinates for sample polygon
        Dim yPolyArr (0)                        ' y coordinates for sample polygon
        strPreviousName = ""            ' stores the pressed control name
        Dim ControlsOnForm (0)                  ' an array to get all controls
        flgGettingPixel = False         ' a boolean indicator - set to true
                'if we are in "get pixel color" mode
End Sub

Sub App_Start
        frmMain.Show

        ' set the values for global vars

        ' x coordinates for sample polygon
        xPolyArr() = Array (20, 200, 10,  105, 200)
        ' y coordinates for sample polygon
        yPolyArr() = Array (40, 10,  200, 240, 200)

End Sub

Sub randomColor                         ' create randomal colors
  Return Rgb (Rnd (0, 255), Rnd (0, 255), Rnd (0, 255))
End Sub

Sub cmdLoad_Click                       ' loads an image to the form's background
        frmShapes.LoadPicture("camel.jpg")
    ' equivalent to: frmShapes.Image = ImageList1.Item(0)
        ' the difference is that with ImageList the image is
        'loaded on app start
End Sub

Sub cmdImage_Click
        frmImage.Show
End Sub

Sub cmdShapes_Click
        frmShapes.Show
End Sub

Sub cmdClose_Click
        frmCancelClose.Show
End Sub

Sub cmdCloseApp_Click
```

```
        frmMain.Close
End Sub

Sub cmdRuntime_Click
        frmRuntime.Show
End Sub

Sub cmdShowShapes_Click                                    ' draw some sample shapes
        frmShapes.Line(0, 0, 50, 50, cWhite)
        frmShapes.Circle(50, 50, 30, randomColor)
        frmShapes.Polygon(xPolyArr(), 0, yPolyArr(), 0, 5, randomColor)
        frmShapes.DrawString("Sample String. This is drawn right on the form.", 12, 20, 60,
frmShapes.Width - 20, 60, randomColor)
End Sub

Sub cmdStretch_Click
        ErrorLabel(errImage)           'mainly in case the image is corrupt
        j = 0
        i = 0
        Do While i < frmImage.Width
                frmImage.DrawImage(ImageList1.Item(0), 0, 0, i, j)
                DoEvents        'refreshes the form: allows the operating
             'system to actually draw the changes.
                i = i + 1
                j = j + 1
        Loop
        Return          'avoid the error label

        errImage:               'something's wrong - announce the user
        Msgbox("Image error")
End Sub

Sub frmCancelClose_Close
        If chkCancel.Checked Then frmCancelClose.CancelClose
End Sub

Sub chkCancel_Click
        If chkCancel.Checked Then
                lblMsg.Text = "You now cannot close this form"
                lblMsg.FontColor = cRed
        Else
                lblMsg.Text = "You can close this form"
                lblMsg.FontColor = cBlack
        End If
End Sub

Sub cmdColor_Click
        frmShapes.Color = randomColor
```

```
End Sub

Sub cmdFore_Click

            SetTransparentColor(cWhite)
        ' White is now transparent on forms with forelayer=true.
        'You must call this function prior to forelayer=true.


            frmForeLayer.ForeLayer = True

            frmForeLayer.Show
End Sub


Sub cmdShow_Click
            frmForeLayer.Circle(20, 20, 10, randomColor, F)
            frmForeLayer.Circle(40, 70, 10, randomColor, F)
            frmForeLayer.Circle(70, 40, 10, randomColor, F)
            frmForeLayer.Circle(120, 220, 10, randomColor, F)

            frmForeLayer.FLine(0, 0, 50, 50, cWhite)
            frmForeLayer.FCircle(50, 50, 30, randomColor)
            frmForeLayer.FPolygon(xPolyArr(), 0, yPolyArr(), 0, 5, randomColor)
            frmForeLayer.FDrawString("Sample String. This is drawn right on the form.", 12, 20, 60,
frmForeLayer.Width - 20, 60, randomColor)
End Sub

Sub cmdHide_Click
            frmForeLayer.FErase(0, 0, frmForeLayer.Width, frmForeLayer.Height)
End Sub

Sub cmdBounce_Click
            y = 0                                   ' ball start y
            speed = 10                              ' ball start speed vector down

            Do While speed <> 0

               frmForeLayer.FCircle(100, y, 30, cBlue, F)  'on fore layer

               DoEvents ' force drawing in loop. Omit this - app. stucks!
                        Sleep(100)                              ' delay

               If speed > 0 Then                    'way down
                           speed = speed * 1.1          'accelerate
               Else                                 'way up
                           speed = speed * 0.8          'decelerate

            If Abs(speed) < 2 Then
                speed = 0
            End If
        End If

        If y > frmForeLayer.Height - 30 Then    'change direction
            Speed = -speed
        End If
        'delete ball - use the transparent color
        frmForeLayer.FCircle(100, y, 50, cWhite, F)
        y = y + speed           'change ball position

    Loop
End Sub

Sub frmRuntime_Show
    ErrorLabel(errShowing)
' Direct name referencing (AKA runtime controls manipulation) is error prone.
' For instance, if you reopen the form with the controls on it you get an error.
    For i = 1 To numButtonsToRedraw.Value
        AddButton("frmRuntime","Btn" & i, 100, 22 * i, 50, 20, "#: " & i)
        AddEvent("Btn" & i,Click, "Btn_Click")
    Next
    Return

    errShowing:
        Msgbox ("An error was trapped")
End Sub

Sub Btn_Click
    If strPreviousName <> "" Then ' verify previously pressed button saved.

        Control(strPreviousName).Color = cGray ' name of previouly pressed
            ' using the Control keyword: set color to gray
    End If

    strPreviousName = Sender.Name           ' save the last pressed button's name

    Sender.Color = cRed     ' use the Sender keyword again to set color to red
                                'note: sender must be an object (anything) with a
                            'property "Color", or nothing happens, see next line

    Sender.Name_Of_Some_Non_Existing_Property = cRed' No error-nothing happens

    If Msgbox("Do you want to see the control type?", Sender, cMsgboxYesNo, cMsgboxQuestion) = cYes
Then
        Msgbox (ControlType(Sender))
    End If

End Sub

Sub cmdClear_Click
    ErrorLabel(errClear)     ' using direct name referencing is error prone!
    For i = 1 To numButtonsToRedraw.Value
        Control("Btn"& i ).Dispose       'it's rare to use dispose in Basic4ppc
    Next
    Return

    errClear:

End Sub

Sub cmdShowControls_Click
    s = "Controls on the form are: " & CRLF
                    ' CRLF - carridge return, line feed: go to next line.
    ControlsOnForm() = GetControls("frmRuntime")
    ' note the parameter is a string!! this adds flexibility
    'and is more error prone.
    If ArrayLen(ControlsOnForm()) > 0 Then
        For i = 0 To ArrayLen(ControlsOnForm()) – 1
    ' always loop to 1 less then length!
            s = s & ControlsOnForm(i) & CRLF
        Next

        Msgbox(s)
    End If
End Sub

Sub cmdRedraw_Click
    cmdClear_Click
    frmRuntime.Show
End Sub

Sub myFirstSub
    Msgbox(Sender)
End Sub

Sub aSubILike
    Msgbox(Sender)
End Sub

Sub WOW
    Msgbox(Sender)
End Sub

Sub Camel_Please
```

```
        frmImage.Show
End Sub

Sub cmdCallSub_Click
        ErrorLabel(errCalling)

        CallSub (txtSubName.Text)
        Return

        errCalling:

End Sub

Sub cmdGetPixel_Click
        flgGettingPixel = True
        Msgbox("Next time you touch" & CRLF & "the screen, the pixel's" & CRLF & "color # will appear")
End Sub

Sub frmShapes_MouseDown (x,y)
        If flgGettingPixel Then
                Msgbox("Pixel: " & frmShapes.GetPixel(x,y))  ' x, y from the event

                flgGettingPixel = False
        End If
End Sub

```

Extending graphics and forms: advanced forms operations

This chapter holds a brief description about some advanced topic related to UI creation. It is not intended to be a complete reference, but to give you a comprehensive overview and to announce you with the places to look for more information.

Note: for an excellent tutorial and some advanced topics including scaling images, maps, GPS and more, see the tutorial written by forum user Klaus: http://www.basic4ppc.com/forum/tutorials/5191-graphics-tutorial.html

This chapter covers:

- Extensions
- Transparency
- Screen size
    - Different resolutions
    - Landscape and portrait
    - Desktop applications
- AutoScale
- Using camera
- Image Processing

Extensions to Basic4ppc Forms

FormLib

A standard Basic4ppc library that adds support for:

- **Full screen** applications.
- Handling **screen rotations**.
- Setting or removing the **minimize button**.
- Changing **fonts and fonts' styles**.
- Adding **context menus**.
- **Text alignment** of Labels and TextBoxes.
- Change **controls parents – panel and forms** (move a control from one form to another) – using the ChangeParent method.

-

FormsExDesktop library by Agraham – for desktop applications
This library adds important functionality to forms. It has some more advanced features that the native Basic4ppc ability and is mainly used for more complicated programs.

Transparency and Alpha transparency

A topic recently gaining lots of focus, transparency is a crucial part of modern applications' UI. The idea is to be able to place to photos on top of one another with parts of the top on invisible. The term **Alpha-Transparency** refers to a semi transparent image that actually blends both colors from the top and the bottom image. The best way to learn about it is probably Agraham's help file attached to his ImageLibEx library (alpha blending topic): http://www.basic4ppc.com/forum/additional-libraries/3171-imagelibex-library.html.

There are many ways to achieve transparency in Basic4ppc UI development.

- ForeLayer supports full transparency as shown above.
- fgLabel and Label control both support transparency.
- ImageButton control supports transparency with the limitations specified above.
- ImageLibEx lets you display semi-transparent (alpha-blended) images. It is a very comprehensive library that can be downloaded here: http://www.basic4ppc.com/forum/additional-libraries/3171-imagelibex-library.html.
- Alpha library, aimed at displaying alpha transparent images (it is more flexible than the previous one, but less documented) can be found here: http://www.basic4ppc.com/forum/additional-libraries/4952-alpha-image-device.html. Unfortunately, there is no help file – but there are good samples to learn from.

Screen Size and orientation issues (or QVGA, Portrait and Landscape):

The development of many different screen resolutions forces us to consider the screen size and orientation when developing. Further more, today's devices often support both landscape and portrait applications. In order to manage issues related with your application's screen, use the following:

- **Form Size**
    - Find the screen size you are working with using the Form's Width and Height properties. Mobile forms are displayed on the entire screen, so the form size is the same as the screen size.
    - Set the size of form during design as described in the designer chapter.
    - Some devices change the screen orientation automatically when the user tilts the device (usually phone). The FormLib library (a standard Basic4ppc library) can be used to trap it with the Resize event (this event is raised both

when the screen is tilted and when the FullScreen method is used). FormLib is also required in order to handle the screen orientation properly.

- **Desktop applications**

  Full screen applications are developed easily with Basic4ppc. Set the form size at design time and use the FormExLib and FormLib to write applications for the desktop.

- **Tilting the device**

  If you are developing for either HTC popular phones or Samsungs popular Omnia, you might want to take advantage of their special sensors, including the tilt sensor:

  - HTC: use the HTC sensors library by Agraham:
    - http://www.basic4ppc.com/forum/additional-libraries/3237-htcsensors-library-htc-diamond.html
  - Samsung: use SamLib: http://www.basic4ppc.com/forum/official-updates/4938-samlib-basic4ppc-samsung-mobile-phones.html

- **Docking**

  Docking is the ability to affix a control to a certain end of the screen, that is, to attach it so that no matter how the form is changed the control holds the same relative position. It is mainly supported by Filippo in his fgControls Library on various controls (most described the next chapter, but here is the link: http://www.basic4ppc.com/forum/additional-libraries/1238-fgcontrols-library.html.

- QVGA / VGA

  The different resolutions can be changed using Filippo fgChangeToVGA. See the documentation for details.

- AutoScale

  AutoScale is the feature that automatically changes the place of the controls and images on your forms so that they fit to different screen sizes. It is described the next section.

AutoScale

As different devices have different screen sizes and different resolutions, creating an application that runs fine on all devices is not a simple task. AutoScale is the feature that automatically changes the place of the controls and images on your forms so that they fit to different screen sizes. AutoScale was added in version 6.8. Compiling an application with AutoScale is done by selecting this compile option from the File menu.

AutoScale tries to take care of the most common two different possible screen sizes:

QVGA - 240 * 320, dpi: 96 * 96

VGA - 480 * 640, dpi: 192 * 192

Basically, there are two ways to adjust the layout:

Manually:

Form.Width / Form.Height and ScreenScaleX / ScreenScaleY can be used to find the screen parameters at runtime and change the layout accordingly. When an application is autoscaled there is no change in the values returned by Form.Width and Form.Height, so you should use the other two, or compile without AutoScale.

AutoScale

AutoScale compiled applications are regular optimized compiled application.
The difference is that the application will try to automatically do all the required conversions between the lowest common denominator - QVGA screen, and the actual running screen.

<u>How AutoScale works</u>

When the program starts it checks the device's screen dpi. It saves the values of the width and height DPI in two variables: screenX and screenY. QVGA screens will return 1 for screenX and screenY and VGA screens will return 2 for screenX and screenY although other values are also possible. Two other relevant variables are fixX and fixY: In the regular compilation mode these variables are constants and equal 1. In AutoScale compilation these variables are equal to screenX/Y variables.

Controls are relocated on the screen so that the keep relative position. When a control is created its location and size values are multiplied with the above values. For example:

Button1.Width = 70 will be translated to Button1.Width = 70 * fixX.

Button1.Left = 20 will be translated to Button1.Height = 20 * fixY.

So the result is that the QVGA layout will appear identical on high resolution screens as well.

Images are also relocated and **stretched**. This may cause some quality loss (it should not if the image is “dense” at the first place) but on most common scenarios will have insignificant effect. If you wish, you can always set images manually according to the actual screen resolution (prepare 2 sets of images).
Note: When AutoScaled, Form.Width and Form.Height will return the same values as when running on a QVGA device. Same is true for all other "screen related" methods (including graphic functions). Use the variables ScreenScaleX and ScreenScaleY the get the real form’s dimensions.

Additional libraries have added some functionality to the AutoScale feature. Some interesting abilities are from the ControlsExDevice:

- **Display** returns the size of the screen in pixels. It supports device AutoScale mode by providing the native size of the screen at runtime. The ratio of the NativeHeight and Height properties or the NativeWidth and Width properties provides the scale factor of AutoScaling if it is in use. These ratios are returned as the FixX and FixY properties and if either of these is not equal to one then the AutoScaled property returns True. This is very similar to the native Basic4ppc constants related with AutoScale.

- **NativeFormImage** assigns new native sized bitmaps to a Form's forelayer and backlayer when AutoScale is in use so allowing the full native resolution of the device screen to be used to provide a higher quality of image than that produced by AutoScaling using the standard QVGA sized bitmaps. However this does require that the application has been coded to be aware of AutoScaling. Use this object to prevent the quality loss associated with autoscaled images.

- **NativeImage** behaves identically to the standard Basic4ppc Image control in non-AutoScaled applications. When AutoScale is in use this control allows the full native resolution of the device screen to be used to provide a higher quality of image than that produced by AutoScaling the standard control. However this does require that the application has been coded to be aware of AutoScaling.

AutoScaling issues – links to the forum

Working with a non-standard resolution – WVGA with bList and AutoScale
http://www.basic4ppc.com/forum/questions-help-needed/5490-problem-blist-wvga.html#post32357

Camera

Scarcely a UI issue, the camera capture ability is worth mentioning in a word. Two additional, users created libraries support camera capture:

Agraham's videocam library

http://www.basic4ppc.com/forum/additional-libraries/2362-desktop-videocam-image-capture-library.html

dzt's Camera Capture library:

http://www.basic4ppc.com/forum/additional-libraries/1155-ccd-camera-library.html

Image processing

This part lists a small list of additional libraries to help you work with images, on top of everything written aforementioned:

- Image processing library with extensive abilities: http://www.basic4ppc.com/forum/additional-libraries/1640-image-processing-library.html
- Big images - Jpeg Library : http://www.basic4ppc.com/forum/additional-libraries/5152-jpeg-library-display-large-images-device.html
- ImageLibEx holds a verity of tools for image manipulations: http://www.basic4ppc.com/forum/additional-libraries/3171-imagelibex-library.html.
- Alpha library, aimed at displaying alpha transparent images (it is more flexible than the previous one, but less documented) can be found here: http://www.basic4ppc.com/forum/additional-libraries/4952-alpha-image-

device.html. Unfortunately, there is no help file – but there are good samples to learn from.

Basic4ppc controls

This chapter is intended to give you an idea about the controls you have at hand. It does not give a full explanation, because you can find the best explanations in the help files and in the forum samples. It contains:

- A list of all standard controls with a brief example and an explanation when needed. Only these controls can be added using the visual designer.
- A list of controls that are included in the ControlsEx library and are part of the standard Basic4ppc package.
- A partial list of important additional libraries, developed by users.
- Common UI graphical effects
    - Gradient
    - Transparency
    - Alpha transparency

Learning to work with the controls

Learning to work with the controls below is not fully described here. Most of the information is given in the help files associated with Basic4ppc or with the relevant libraries (see the Libraries chapter about libraries help file if you are not familiar with the concept), and in the Forum.

Native controls

General word about native controls:

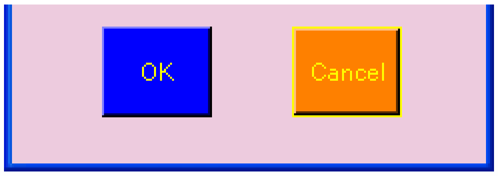
All native controls in Basic4ppc are getting old. It is no secret that the last years have shown great development in mobile UI. Creating modern user interface in Basic4ppc is demonstrated **in details** later this section, and evolves mainly the usage of controls that can be used without stylus: Buttons (mainly ImageButton), bList, images, labels, timers, fgGradientButtons, fgKeyboard and more (they are all described here, no worries!...). User interfaces that require stylus are getting more and more a niche of professional or business specific applications (Basic4ppc's mobile IDE is one, and so are application for a restaurant's order management systems, car rental agency and more). However, they are more than sufficient to start with.

All controls are displayed here and you should be familiar with them – native first.

1. Collections

Collections are a special kind of controls. They are actually data structures that can be added at design time. Since they are not a GUI element, they are not described here (refer to the data structures chapter).

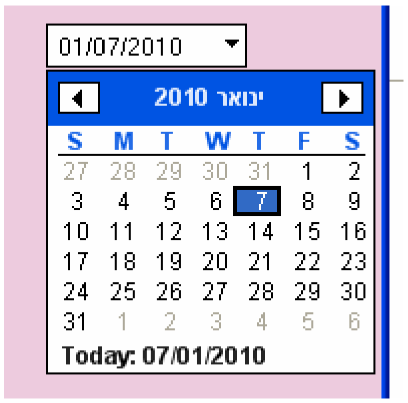
2. Button



Buttons were described thoroughly in previous examples. There is not much new to say: refer to some example above.

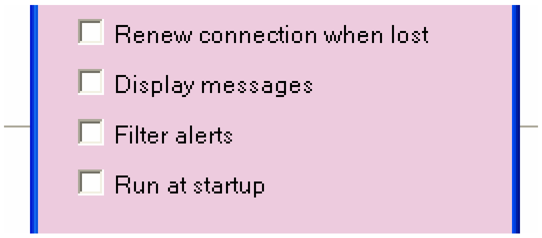
Note: Since buttons are one of the most common controls you have, there are some important improvements you should be aware of. The most common ones are the ImageButton native control, and the fgButton developed and supplied (open source) as part of the fgControls library by Filippo of the Basic4ppc community. It is described further later – it is recommended that you read it and note the special designer developed by the forum user Derez for it.

3. Calendar



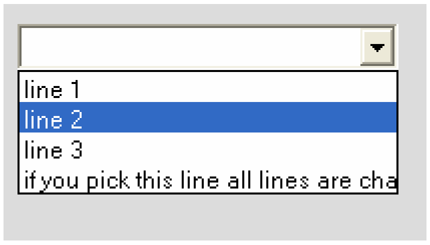
The calendar control lets you choose a date: it is described in details in the Date/Time chapter, together with some improvements

4. Checkbox



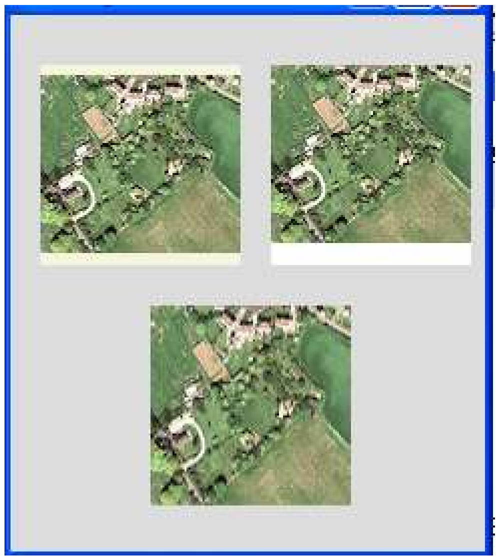
Checkboxes allow you to pick multiple options from a list.

5. Combo Box



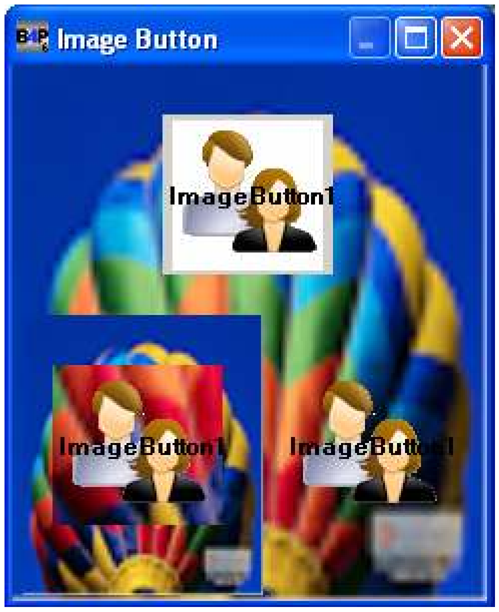
A ComboBox lets you pick one option from list without consuming too much of the screen's expensive real estate area.

6. Image



The image control was presented in the previous chapter. It allows you to display an image on the screen with different alignment settings and to change the image to a different one set from an imagelist or from a file.

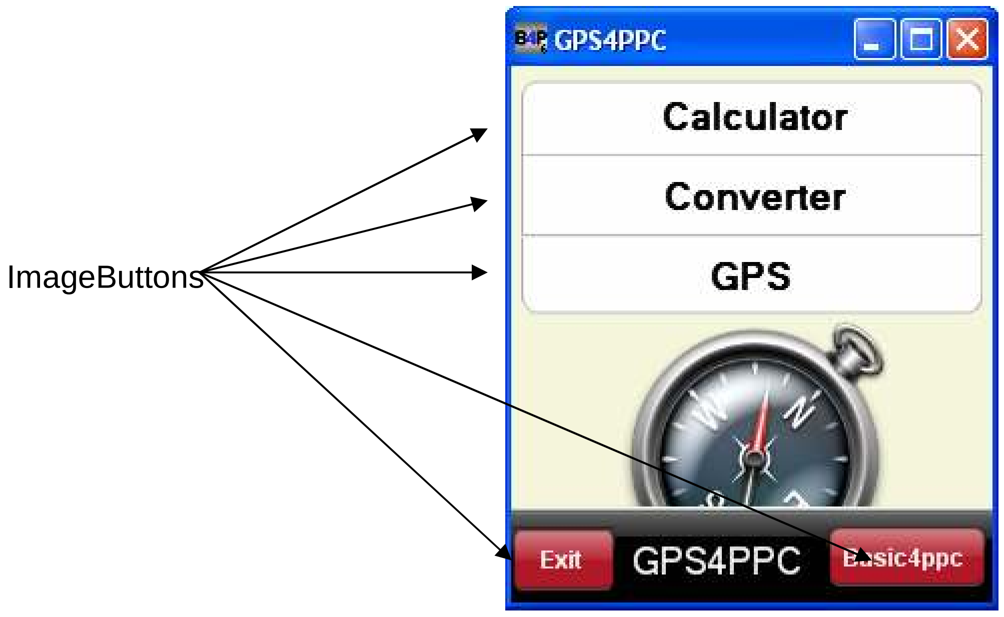
7. Image Button



An ImageButton is a special kind of a button that lets you draw an image on it. What makes it very useful is the ability to set a transparent color, thus allowing it to display a non-rectangle image. The picture above shows the different possibilities. The upper button has not transparent color. The lower right one has the transparency property set to true, with the very same picture as the one above. Generally, when transparent color is set to true, the upper left pixel's color (pixel 0,0) is set as the transparent color – everything with this color is not displayed. However the image button has an important quirk you should be familiar with: it gives transparency a unique interpretation, treating only the background of the form (**or the panel it is placed on**) as the transparency's reference. That is, every control it is places on top of is being disregarded: The lower left ImageButton demonstrates this issue. It is placed with transparency on top of a second Image control, but the button shows the **background images as if there was nothing in between – because this is the background of the form.** Note, that if you placed it on a **panel** with a different color or image it would treat the panel as its background, and this is the best way to overcome this issue (see the GPS4PPC chapter later for demonstration).

Transparency in the mobile device is very limited by default. Microsoft's .NET CF does not natively support transparency or alpha transparency. The Basic4ppc community has developed some libraries to overcome this issue: see details at the previous chapter.

The ImageButton is an important control when designing finger operated UI. For example, in the GPS4PPC application (free on the Basic4ppc site with full source code), the ImageButton is used in the following opening screen:



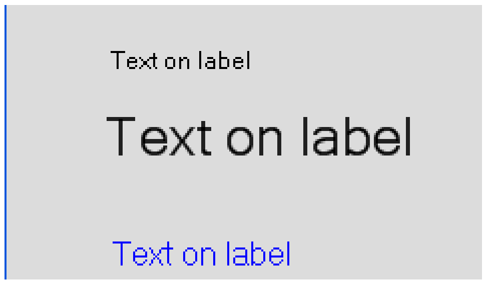
The reason to use an image though there is no real picture on these buttons is the special gradient (4 colors gradient) at the bottom ones and the special rounded shape at the top ones. This is not the only solution, but many times you have to design the ImageButton completely including graphics, and sometimes you can use better solutions, like the fgGradientButtons shown later.

8. Image List

(no graphical representation – see discussion previous chapter)

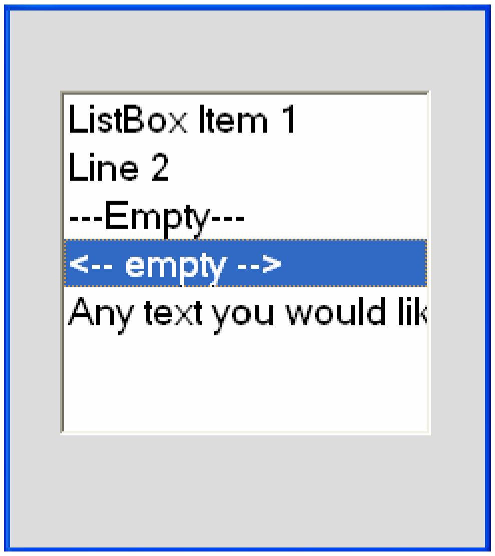
Discussed earlier, ImageList gives you the ability to keep many images efficiently. It is highly recommended to use it whenever you use images.

9. Label



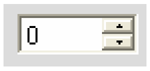
The text that appears on the label is one of the common ways to interact briefly with the user: labels are used for captions and for general short text display.

10. List Box



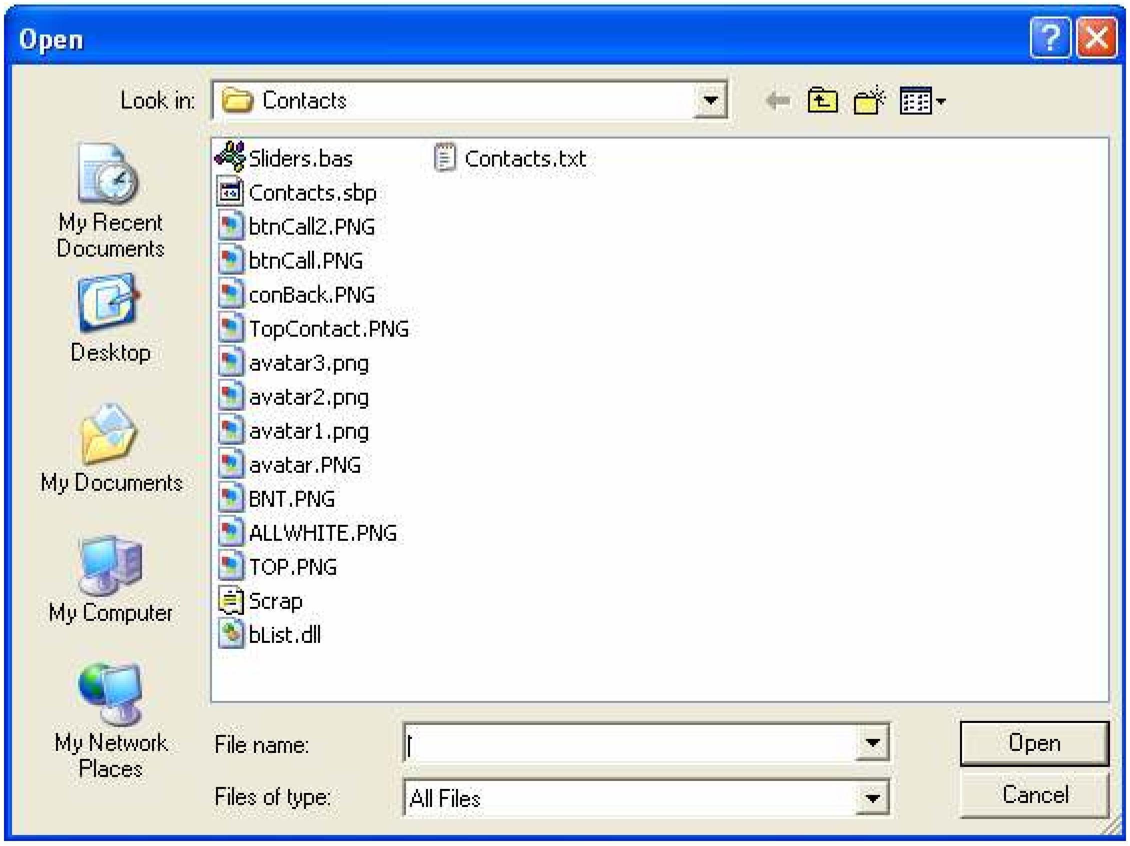
A listbox displays the content of a list of items and lets you pick one of them. It supports scrolling but is mainly designed for stylus. Listbox is a simple control that allows no "table" display – for these display types use the ListView control or better, the Table control. For finger driven, modern style list that support also images, use the bList control.

11. Numeric Up-Down input (NumUpDown)



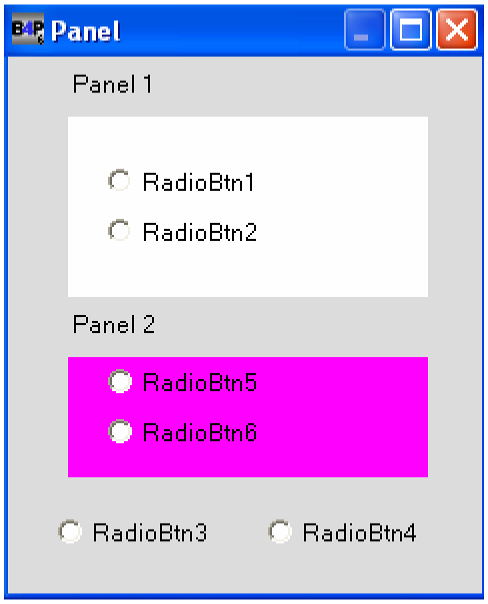
Lets the user enter numeric input with small arrows to enter the input graphically.

12. Open File Dialog Box (OpenDialog)



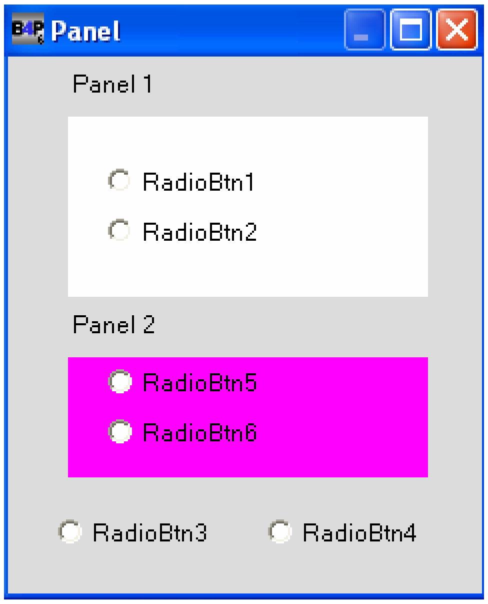
The standard Open File/Save File dialog box (the one shown is Windows XP, of course Vista/Windows 7 and all mobile editions display the proper one) is supplied as a control in Basic4ppc for easy file choosing.

13. Panel



A panel is a grouping control – it allows grouping multiple controls with under the same "parent", so that they share some of their properties. For example, hiding the panel completely hides all of its controls.

14. Radio Button

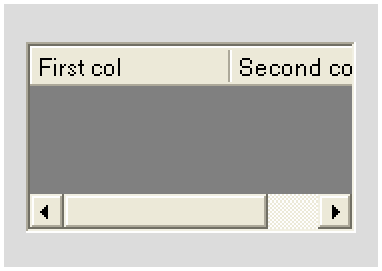


Radio buttons are used when you want the user to select one of many optional choices. When radio buttons share parent (that is, they are placed on the same form or panel) they act as a group – when one is selected the others are deselected. A good forum thread with code samples is here: http://www.basic4ppc.com/forum/questions-help-needed/4857-grouping-radio-buttons.html

15. Save File Dialog Box (SaveDialog)

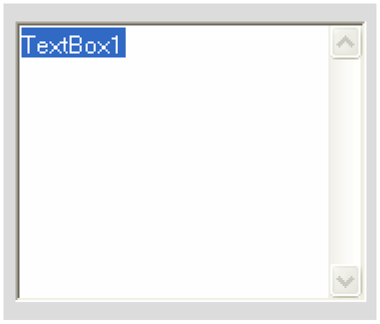
    See the **Open File Dialog** for details.

16. Table



    The table control is a special control for small data and database operations. It displays a table of data and lets you connect to various data sources, manipulate the data and save them (such are SQLite database, Microsoft Excel and more). There is a lot of information about the Table control – use the samples and the forum to learn how to work with it.

17. Textbox



One of the most basic controls, the Text Box is used to get textual input from the user. Text boxes can have one line or more and are useful for short input.

18. Timer

The timer control was discussed at length in a special chapter.

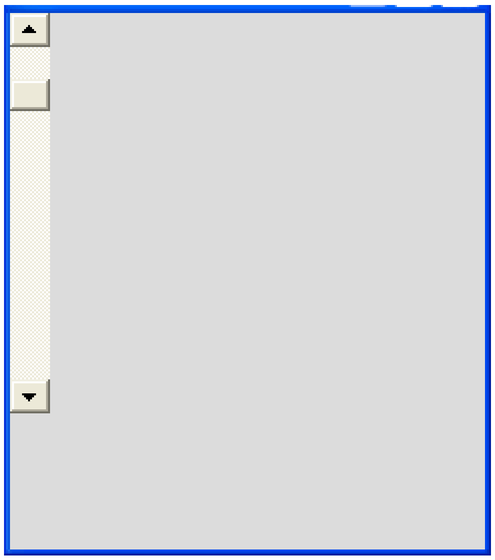
Controls included in the ControlEx library (part of the standard Basic4ppc edition)

The controls shown below are basically subject to the same preface given under "native controls". The difference is that they are not added using the designer, they are added manually (as shown in the sample program).

19. Progress bar

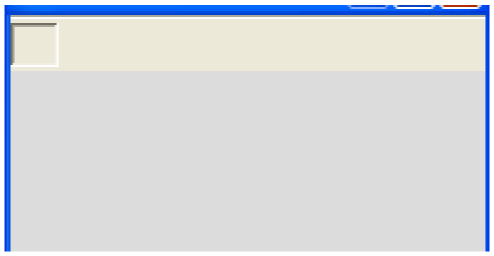
The progress bar display a bar that is gradually filled with a different color: it is a common way to indicate the user of the percentage of progress that has been carried out so far.

20. Scroll bar



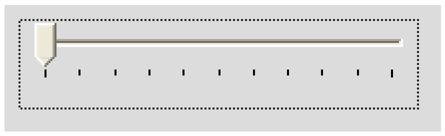
A scroll bar is the basic scrolling control. Being independent of the control it scrolls, it allows you to take control over the exact scrolling method and actually supplies you just the graphics and the values of the desired position.

21. Toolbar and Toolbar button



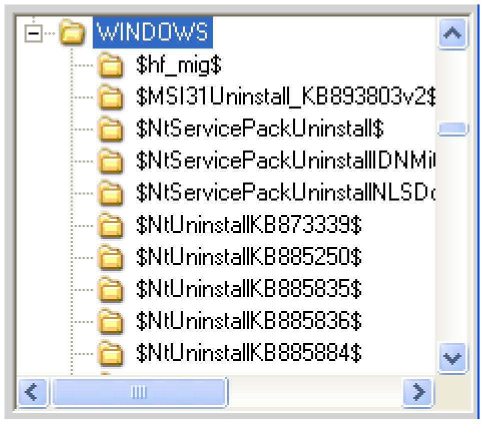
The toolbar is a bar with buttons at the top of the screen. Unlike the sample above, each can hold a picture. It is usually used for shortcuts.

22. Track bar



A track bar allows the user to select a value on a scale: it can be displayed either vertically or, as in the sample, horizontally.

23. Tree (Treeview control and the Node component)



Displays the common view of a tree with symbols next to it. An extension written by forum user Derez is available here: http://www.basic4ppc.com/forum/additional-libraries/5452-treeviewplus.html.

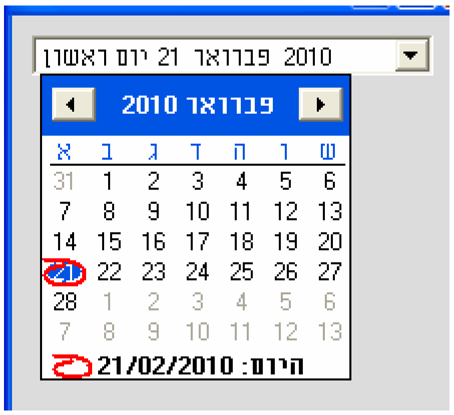
Controls from User-Created libraries

User created libraries play substantial role in the development of Basic4ppc. The controls listed below are only a small part of a huge diversity found in the forum (under "Additional libraries").

ControlsExDevice library by Agraham (All rights reserved)

Note – there is also ControlsExDummy, ControlsExDesktop. Read the help file's first page before using!

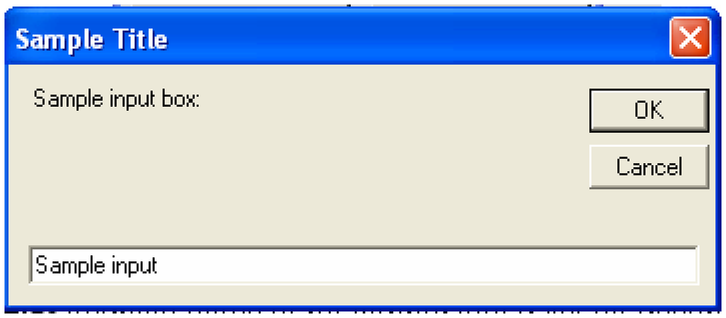
24. **DateTimePicker** control provides an enhanced means of selecting dates and times, compared to the native Basic4PPC Calendar control.



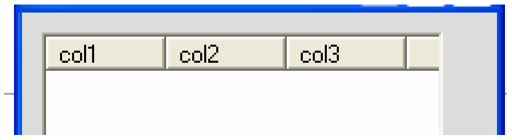
The difference from other calendar controls is flexibility and appearance. See the Date/Time chapter for more information.

25. **IconList** enables Icons to be loaded into a list for assignment to controls that can accept Icon properties. Icon list is similar to ImageList described earlier this chapter and has similar purpose, only with icons.

26. **InputBox** provides an easy way to get input from a user.



27. **ListView** control allows you to display a list of items as text and, optionally, an icon to identify the type of item and a checkbox for each item. The items may be displayed together with an optional icon, in three views. These are small icon view, large icon view and list view. Each item may have sub-items which may be displayed together with the item in a details view.



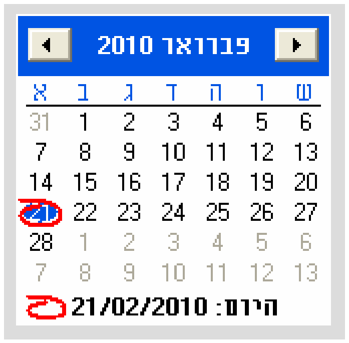
Databases connection to ListView

List view can be used as an alternative to the table control, when using the ListView provided by the forum user Filippo. This advanced control adds some important functionality as described in the post:

http://www.basic4ppc.com/forum/additional-libraries/1749-listview-dll-without-sqlite-interface.html

There are many other posts in the forum about the listview control – use the search option.

28. **MonthCalendar**



Month calendar control represents a Windows control that enables the user to select a date using a visual monthly calendar display. A range of dates may be selected and particular dates emboldened if required. This control is more capable than a DateTimePicker or the Basic4ppc Calendar control but takes up more screen space.

29. **Notification** enables PocketPC notifications to be displayed.

30. **NotifyIcon** puts an icon, determined by the application in the Notification area of the device. The ControlsExDummy version of NotifyIcon is non-functional. However the ControlsExDesktop library includes a fully functional but differently implemented NotifyIcon.

31. **Splitter** enables the user to resize controls that are docked to the edges of the Splitter control at run time. See the Splitter topic for more information.

32. **StatusBar** shows a single line status message on the screen.

33. **WebBrowser**

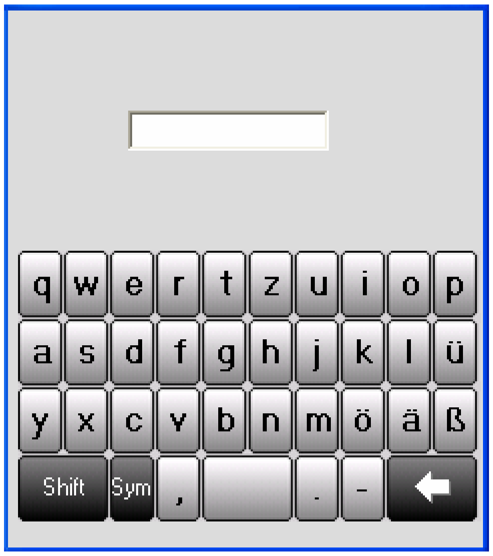
For desktop applications only.

http://www.basic4ppc.com/forum/additional-libraries/1862-web-browser-desktop-device.html

34. **RichTextBox for device**

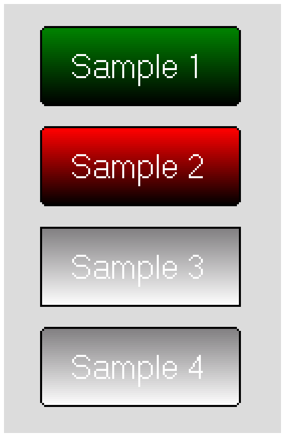
http://www.basic4ppc.com/forum/additional-libraries/3955-htmlpanel-rich-text-device.html

35. **fgKeyboard**



This unique control displays a soft keyboard that is suitable for finger usage rather than stylus. This keyboard, when displayed, mimics the functionality of the usual keypad in a windows mobile device. The keyboard is included in a library with the same name available on the forum. Here is a link to the thread – there are also some users' questions on the same thread: http://www.basic4ppc.com/forum/additional-libraries/5227-fgkeyboard-3.html

36. fgGradientButtons and designer



fgGradient buttons are extending the basic ability supplied by the button control. They are very useful and together with the fgKeyboard, bList, ImageButton and Image form some basic modern UI elements.

Converting buttons in an existing application to gradient buttons is very easy and can be easily be learnt from the following example, crated by the forum user Derez: http://www.basic4ppc.com/forum/share-your-creations/5449-gradient-buttons.html. Derez also wrote a utility program that actually translate existing Basic4ppc application into new ones with gradient buttons: http://www.basic4ppc.com/forum/share-your-creations/5449-gradient-buttons.html

fgControls library

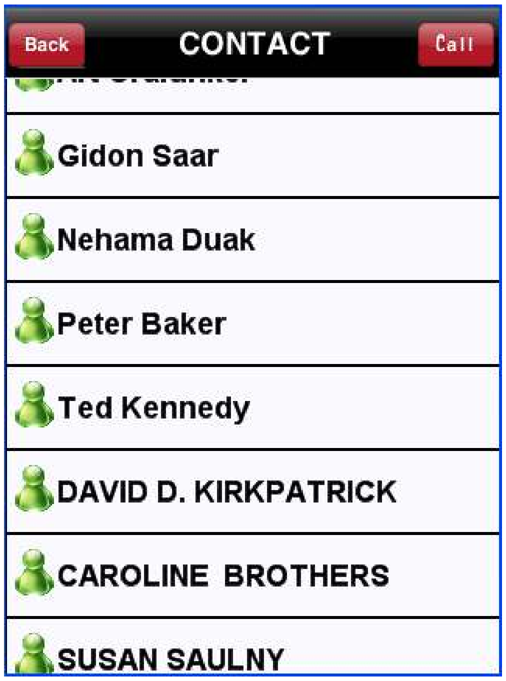
The fgControls library, created by forum user Filippo, is a remarkable extension to Basic4ppc's libraries. The controls are described here briefly – a detailed example is included for each of them. Note: there is an extension to it named fgControls3d:

http://www.basic4ppc.com/forum/additional-libraries/5182-fgcontrols3d-controls-shade-effects.html). Here is the link to the original one: http://www.basic4ppc.com/forum/additional-libraries/1238-fgcontrols-library.html.

It includes the following:

37. fgTextBox – extends normal Textbox abilities. Lets you select border style, type only numbers and set different docking style.
38. fgCalendar – similar to DataTimePicker.
39. fgMouseEvents – handles mouse events that occur in the application.
40. fgMaskedEdit – allows masked editing (e.g. only dates and so on).
41. fgImageList – extends the imagelist abilities.
42. fgSystemInfo – supplies system information.
43. fgSystemIo – supplies information about the file system.
44. fgToolBar – extends toolbar options.
45. fgToolBarButton – for fgToolbar.
46. fgSystemColor – allows setting system color for all objects.
47. fgGetAsyncKeyState – determines if a key is up or down.
48. fgScreenOrientation – automatic display orientation.
49. fgDirChooser – directory chooser.
50. fgFileChosser – file chooser.
51. fgChangeToVGA – changes automatically all controls on a form to match VGA mode.
52. fgChangeToQVGA – changes automatically all control to match QVGA mode.

53. bList



bList is a modern UI element used as a list viewer. bList is intended to help creating user interface for touch screens and is specifically suited for Basic4ppc. bList allows finger-based operation and is intended to be used for multi-choice elements, lists, and the like. Such is the contacts list displayed above.

> Moving the list is done by dragging the finger (or a stylus) over the screen. The faster dragged the faster the list moves. Lifting the finger will cause the list to reduce speed to a full stop. Clicking an item fires an event and the programmer can handle it in code.

bList is a very advanced control. It lets you display images and text, it is very easy to use (the basic sample supplied with the library is called "bList in 4 minutes)" and is very strong. The zip file, downloadable from the forum, includes various samples and demo programs that show you every single feature the control has.

bList is available here:

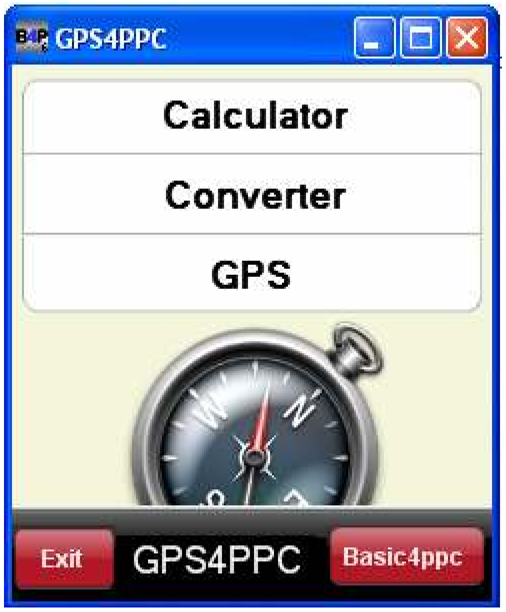
http://www.basic4ppc.com/forum/official-updates/4881-blist-v0-9-beta-released-7.html#post32171

Modern GUI Tutorial: How to create UI such as the GPS4PPC

GPS4PPC is a sample program that shows you how to work with GPS in Basic4pc and how to create a nice graphical user interface. It is downloadable from the Basic4ppc webpage here: http://www.basic4ppc.com/gps4ppc.html

In this chapter I will try to give a step by step explanation of the way I created the GPS4PPC GUI, so that it is easy for others to reproduce.

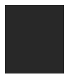
We will start with the main screen and go step by step. The application starts with the following screen:



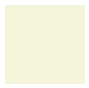
Generally, there are the following graphical elements: a small menu with three options, an image of a compass and a bottom bar with a label and two buttons.

As we take a look at it, there are several things to notice (each point is discussed later):

- As you already know, the entire screen is one form. The back color of the form is not the default gray – it is actually the color (40, 40, 40), which is **absolutely not** the color you see as background! Color (40, 40, 40) is something like this:



while the background here is (245, 245, 220):

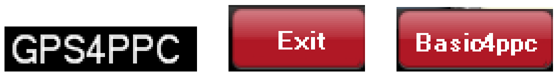


- The small “menu” is actually a composition of three ImageButtons.
- The compass is displayed on an image control.

Note: The compass image is by courtesy of Marcelo Novaes de Oliveira, "KDEWolf", www.mnovaes.eu, who gave us his kind permission to use it for the open source project GPS4PPC.

- The bottom bar is an image of a bar:

on top of which I placed a label and two buttons:

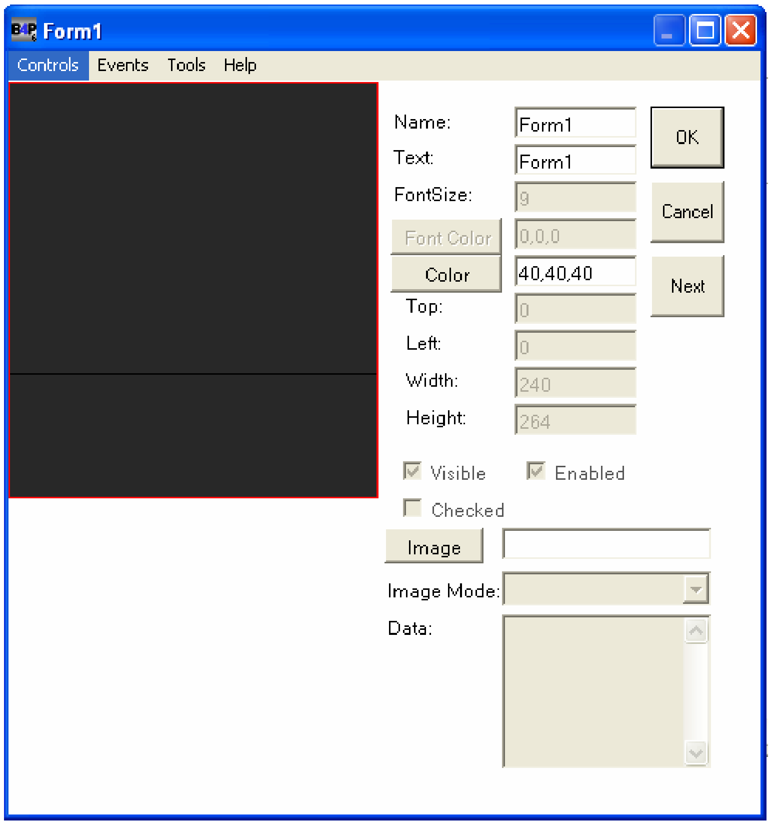


Creating the starting screen

<u>Step 1 – create the form</u>

Create a new Basic4ppc application and save it.

Create a new form, and set its back color to 40, 40, 40.



Name:	Form1
Text:	Form1
FontSize:	9
Font Color	0,0,0
Color	40,40,40
Top:	0
Left:	0
Width:	240
Height:	264

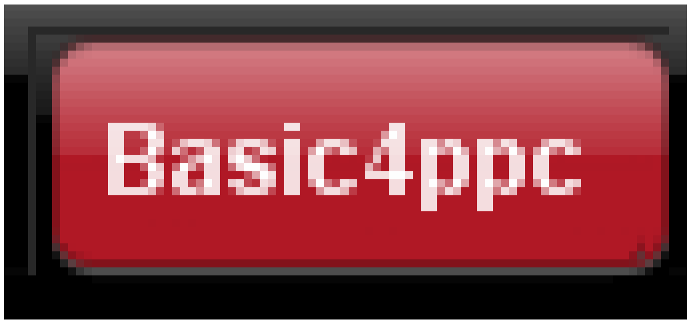
[✓ Visible] [✓ Enabled]

Checked

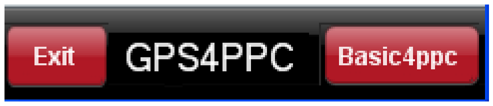
Image Mode:

Data:

The reason we change the back color is due to another weird behavior ImageButton has: when you click it (or touch it) it moves one pixel down and to the right to indicate the click, and fills the top and left line of pixels with the color of the form below it. Here is a close up of the button when pressed:

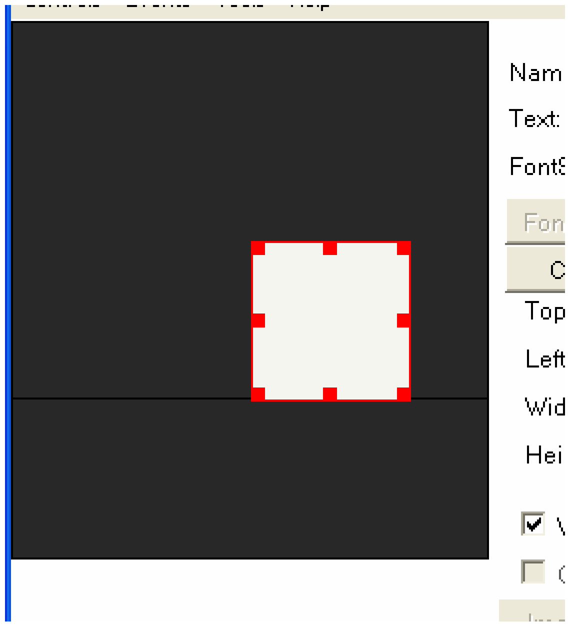


The color is set to 40, 40, 40 so as to be as close as possible to the colors of the bottom bar: this is hardly noticeable when clicking it, if you do not really focus:

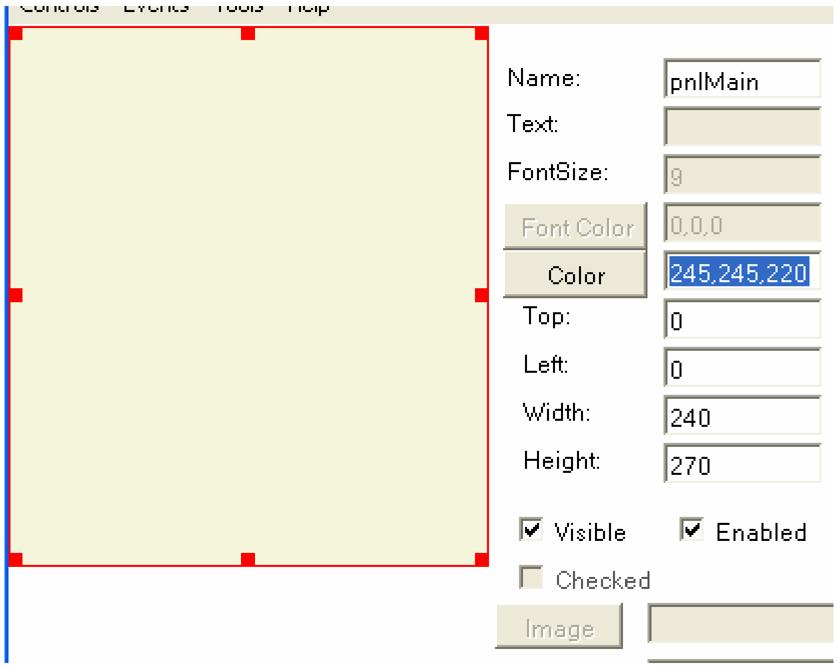


Step 2 – set proper display background color

Anyhow, we need the form to have the desired color. Add a panel named pnlMain, color (245, 245, 220



and stretch it so that it covers the entire form:

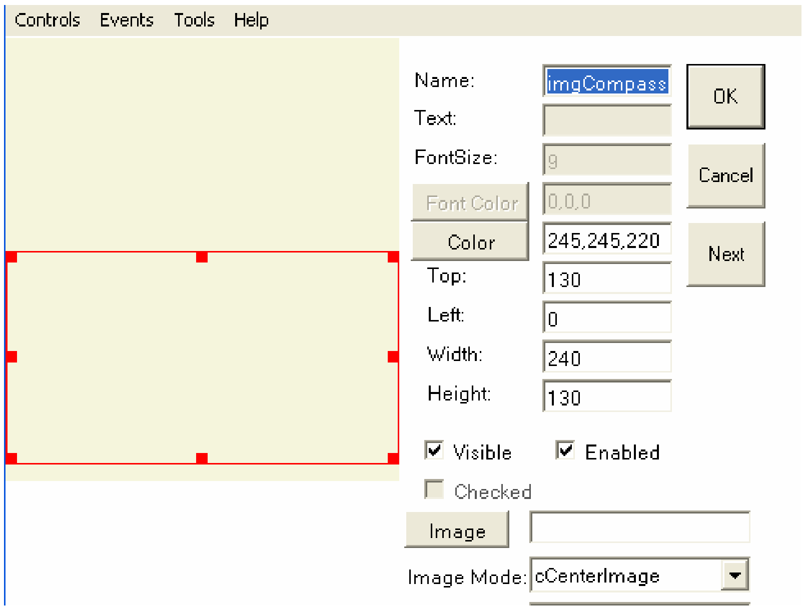


Name:	pnlMain
Text:	
FontSize:	9
Font Color	0,0,0
Color	245,245,220
Top:	0
Left:	0
Width:	240
Height:	270

☑ Visible ☑ Enabled
☐ Checked

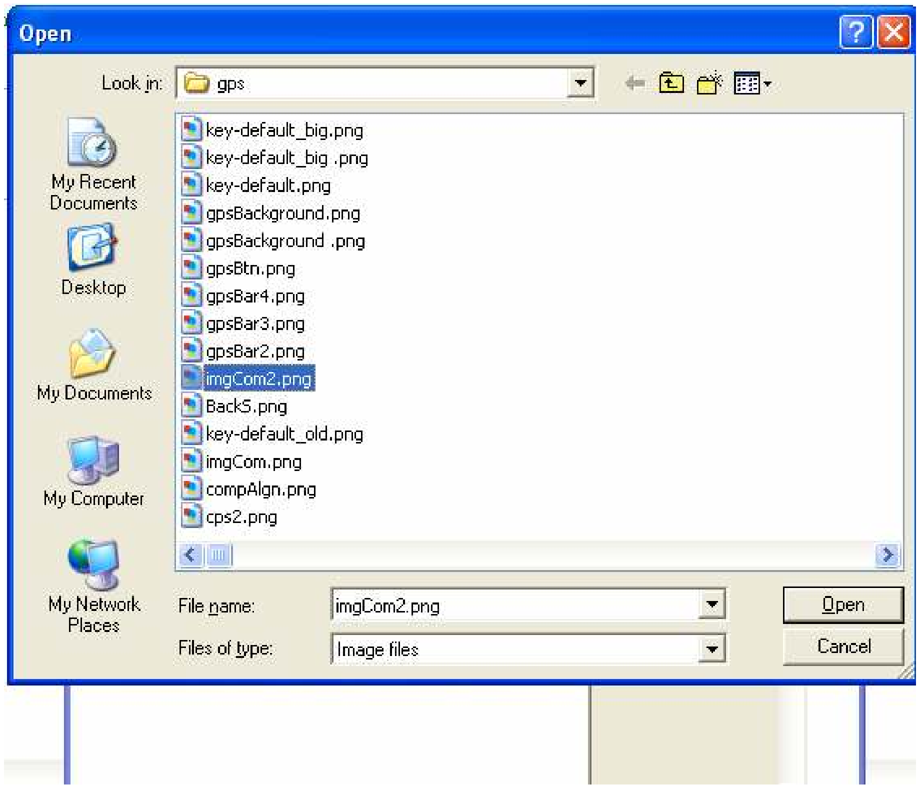
Step 3 – add compass image

Add an image to the panel:



Name: imgCompass
Text:
FontSize: 9
Font Color 0,0,0
Color 245,245,220
Top: 130
Left: 0
Width: 240
Height: 130
☑ Visible ☑ Enabled
☐ Checked
Image Mode: cCenterImage

and set its image property to the compass image:

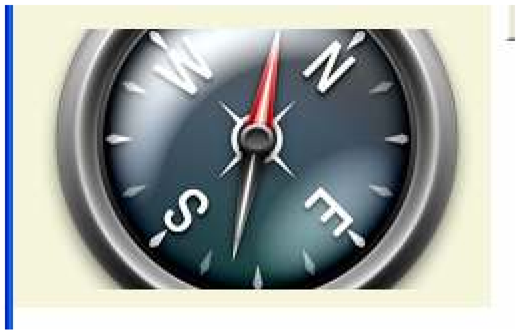


Look in: gps

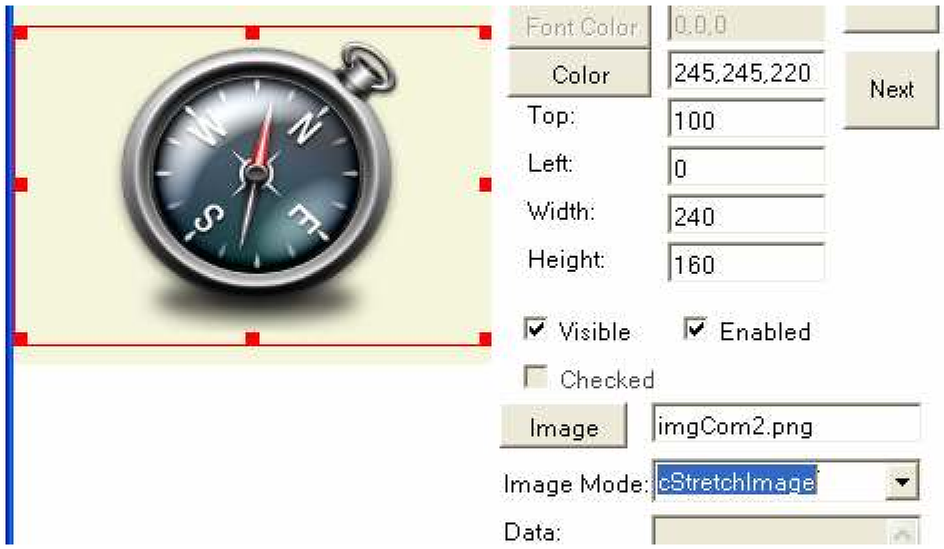
key-default_big.png
key-default_big .png
key-default.png
gpsBackground.png
gpsBackground .png
gpsBtn.png
gpsBar4.png
gpsBar3.png
gpsBar2.png
imgCom2.png
BackS.png
key-default_old.png
imgCom.png
compAlgn.png
cps2.png

File name: imgCom2.png
Files of type: Image files

so that it looks like this:



and then set the Image Mode property to cStretchImage and adjust the height:

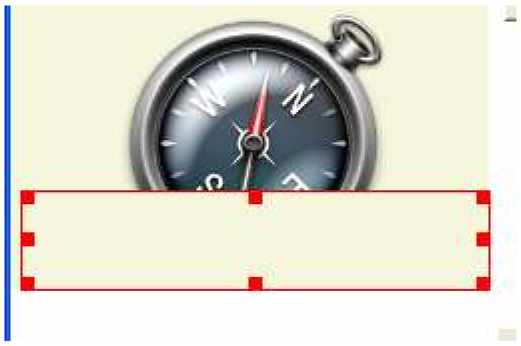

Font Color: 0,0,0
Color: 245,245,220
Next
Top: 100
Left: 0
Width: 240
Height: 160

☑ Visible ☑ Enabled
☐ Checked

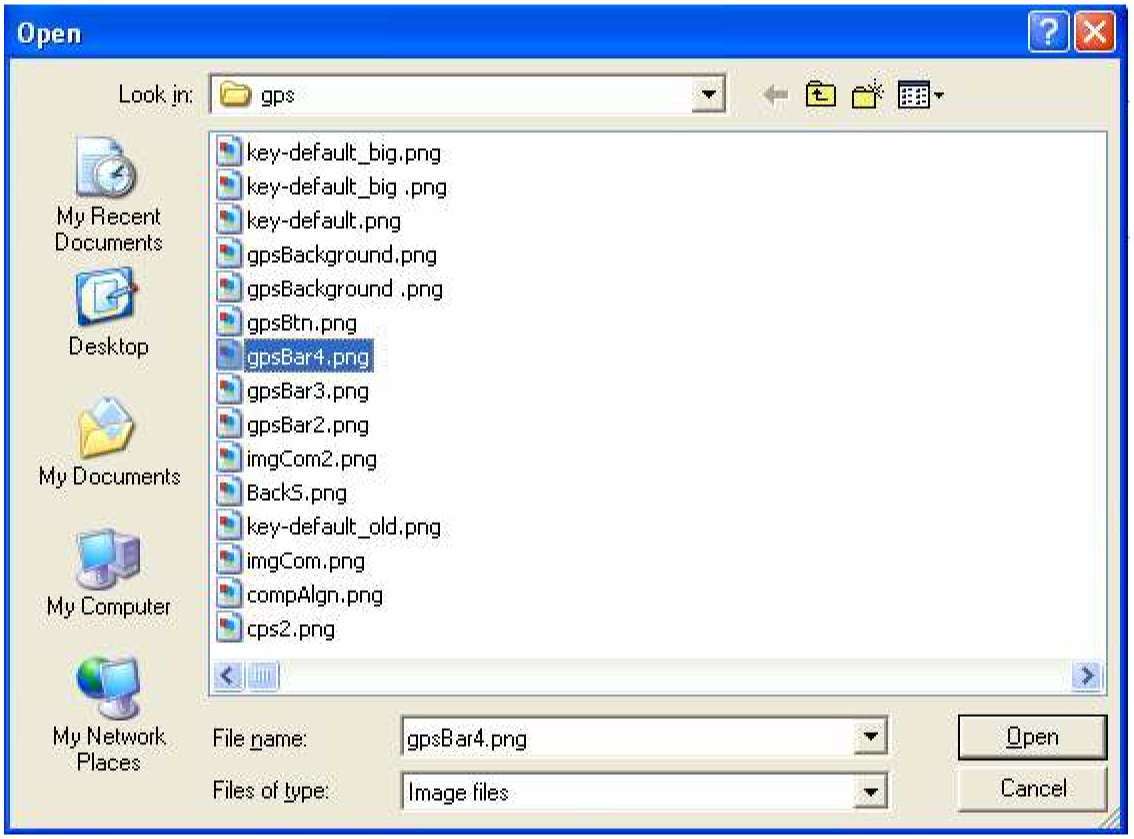
Image: imgCom2.png
Image Mode: cStretchImage
Data:

Step 4 – add bottom bar

Add an image at the bottom (I've also changed the compass image's dimensions):



Now, we have to use another custom image. I had to do some experiments so it is called gpsBar4:



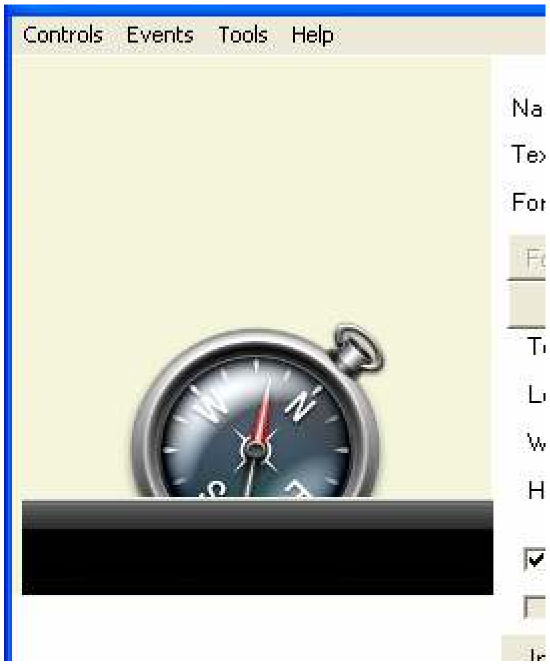
key-default_big.png
key-default_big .png
key-default.png
gpsBackground.png
gpsBackground .png
gpsBtn.png
gpsBar4.png
gpsBar3.png
gpsBar2.png
imgCom2.png
BackS.png
key-default_old.png
imgCom.png
compAlgn.png
cps2.png

File name: gpsBar4.png

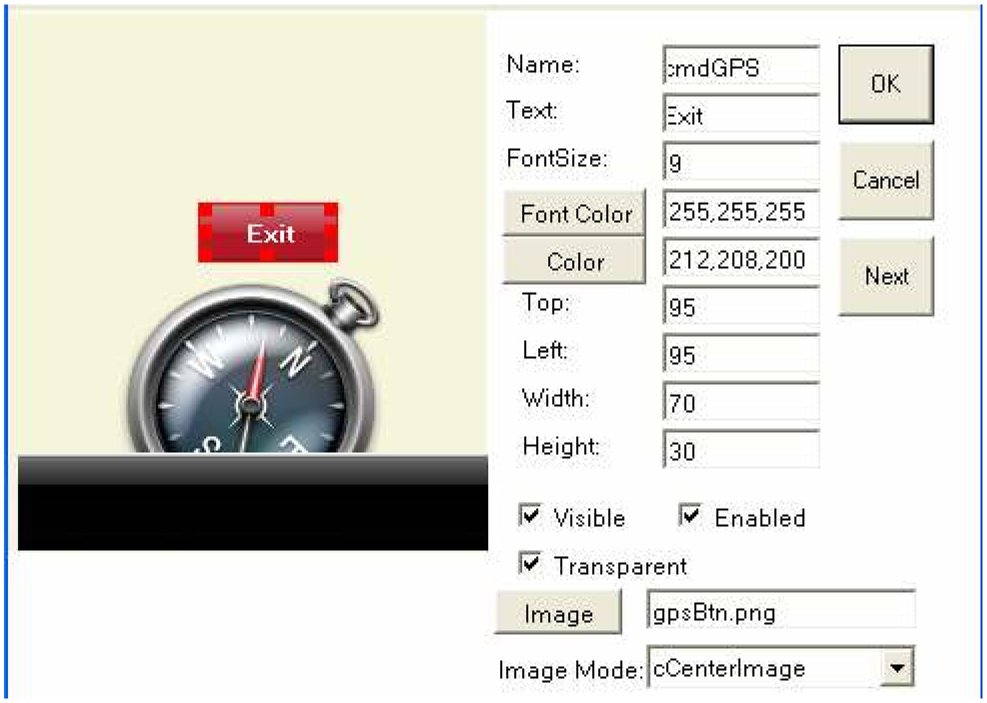
Files of type: Image files

And it looks like this:

There is some very interesting thing to notice about this image: it has a gradient composed of **four colors** to create the shine effect. Now your form looks like this:

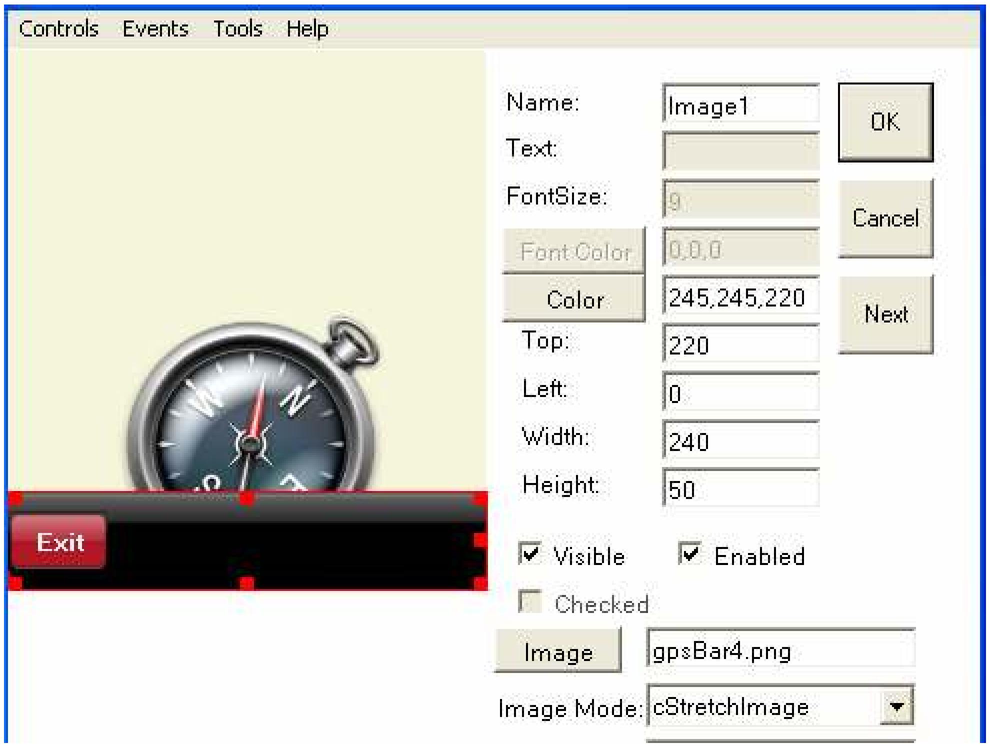


Create a new ImageButton, call it cmdGPS, set font color to white (255, 255, 255) and text to Exit:



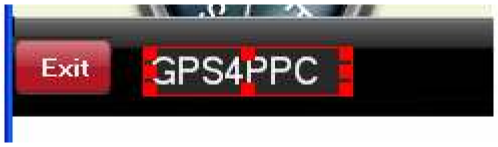
Name: cmdGPS
Text: Exit
FontSize: 9
Font Color 255,255,255
Color 212,208,200
Top: 95
Left: 95
Width: 70
Height: 30
✓ Visible ✓ Enabled
✓ Transparent
Image gpsBtn.png
Image Mode: cCenterImage

Place it at (230, 0) and set both the ImageButton and the Image of the bar **ImageMode** property to cStretchImage:

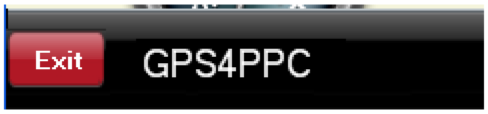


Name: Image1
Text:
FontSize: 9
Font Color: 0,0,0
Color: 245,245,220
Top: 220
Left: 0
Width: 240
Height: 50
Visible ✓
Enabled ✓
Checked
Image: gpsBar4.png
Image Mode: cStretchImage

Create a label the same way (font size 14, color white):

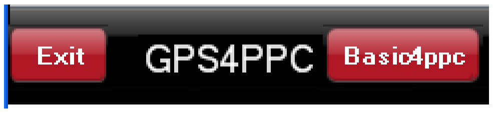


and set its color to black:



note, you should place it so that the black color at the bottom of the bar unites with the color of the label.

Now, add another ImageButton the same way you added the other one:

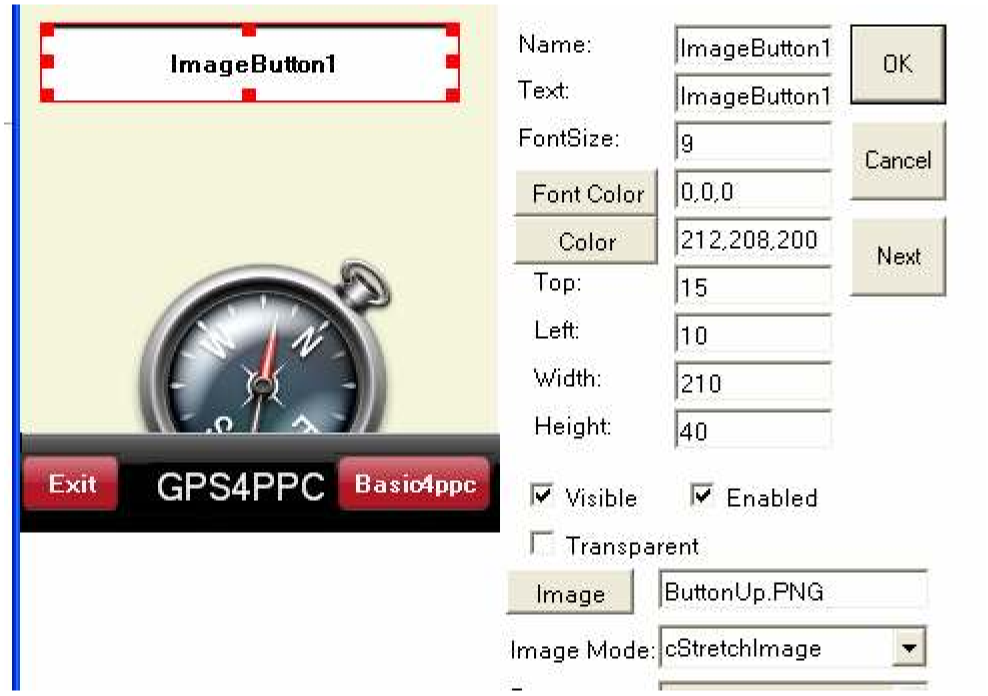


Step 5 – add the menu

The menu is composed of three ImageButtons with images I created with a photo processing application. You will have to have three of your own. Make sure the top one has a black line of pixels at the very top, and the bottom one has it at the bottom.

The top one is called ButtonUp.png. Add an ImageButton and set it to show this image with image style of cStretchImage. Since you need it to have a yellowish background, be sure to first select the panel and not the form so that it is placed on top of the panel.

Note what happens if you set transparency to False:

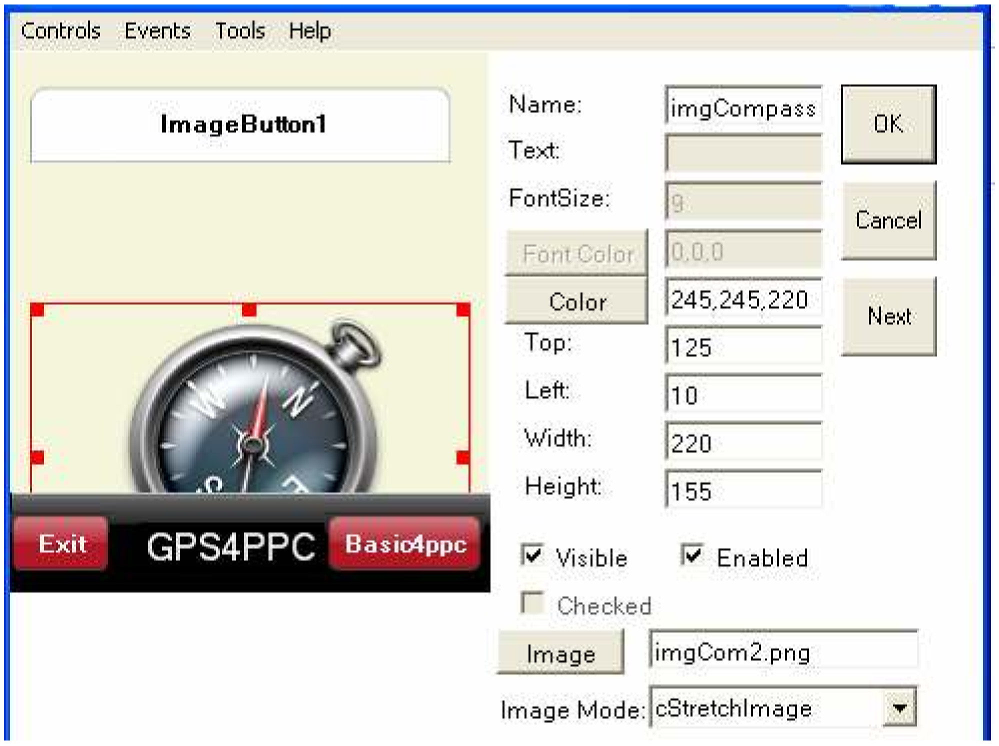


Name:	ImageButton1
Text:	ImageButton1
FontSize:	9
Font Color	0,0,0
Color	212,208,200
Top:	15
Left:	10
Width:	210
Height:	40

☑ Visible ☑ Enabled
☐ Transparent

Image: ButtonUp.PNG
Image Mode: cStretchImage

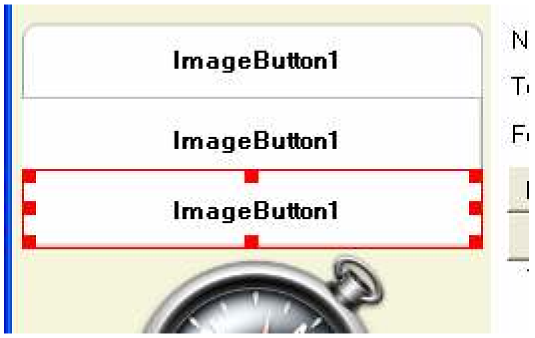
So you better set it to True:



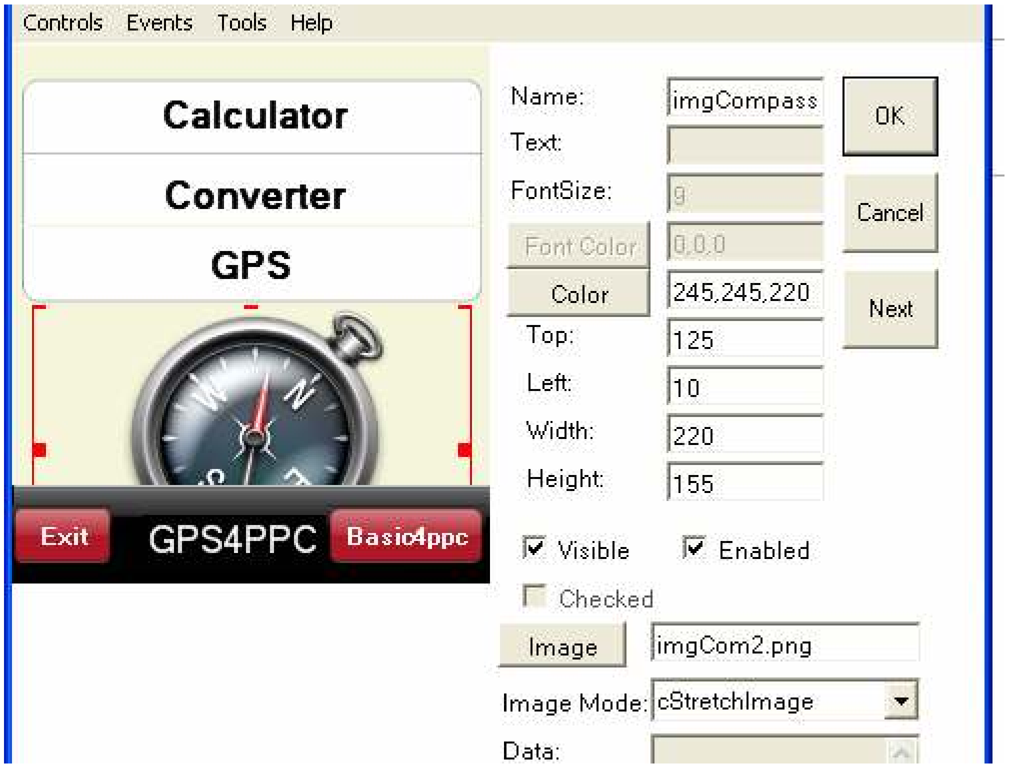
Name: imgCompass
Text:
FontSize: 9
Font Color: 0,0,0
Color: 245,245,220
Top: 125
Left: 10
Width: 220
Height: 155
☑ Visible ☑ Enabled
☐ Checked
Image: imgCom2.png
Image Mode: cStretchImage

Note: if it does not help and the black line is still there you probably did not place it on the panel, but rather you placed in on the form. If this is the case, right click it, choose "change parent", press OK and click the panel.

Now add two buttons below. The middle one need not be transparent:



And set the texts and font size:



Name: imgCompass
Text:
FontSize: 9
Font Color: 0,0,0
Color: 245,245,220
Top: 125
Left: 10
Width: 220
Height: 155
[✓] Visible [✓] Enabled
[ ] Checked
Image: imgCom2.png
Image Mode: cStretchImage
Data:

You are done – save your work and run!

Note: there are probably some small adjustments required to fit perfectly. We leave them to you.

Creating the lists

The GPS4PPC application contains simple usage of the bList control. The best way to learn how to work with the bList control is to download it from the link in the controls chapter (above) and to follow the very detailed help file attached. It takes minutes!

Other points of interest

* The GPS4PPC uses a specially created keypad for numbers – it is very easy to create it – follow the code, which is quite self-explanatory.

- The application does extensive usage of the FormLib – especially the ChangeParent method. See the FormLib's help file in order to understand the usage – it is very simple.
- It is not the purpose of this chapter to explain the GPS part in the application but only the UI. Anyhow, there are two very simple tutorials in the forum and on the website to start from and many threads with questions that had been asked.

Creating the rest of the screen in the application has the same principles, so it is a good practice. You can always download the source code at http://www.basic4ppc.com/gps4ppc.html