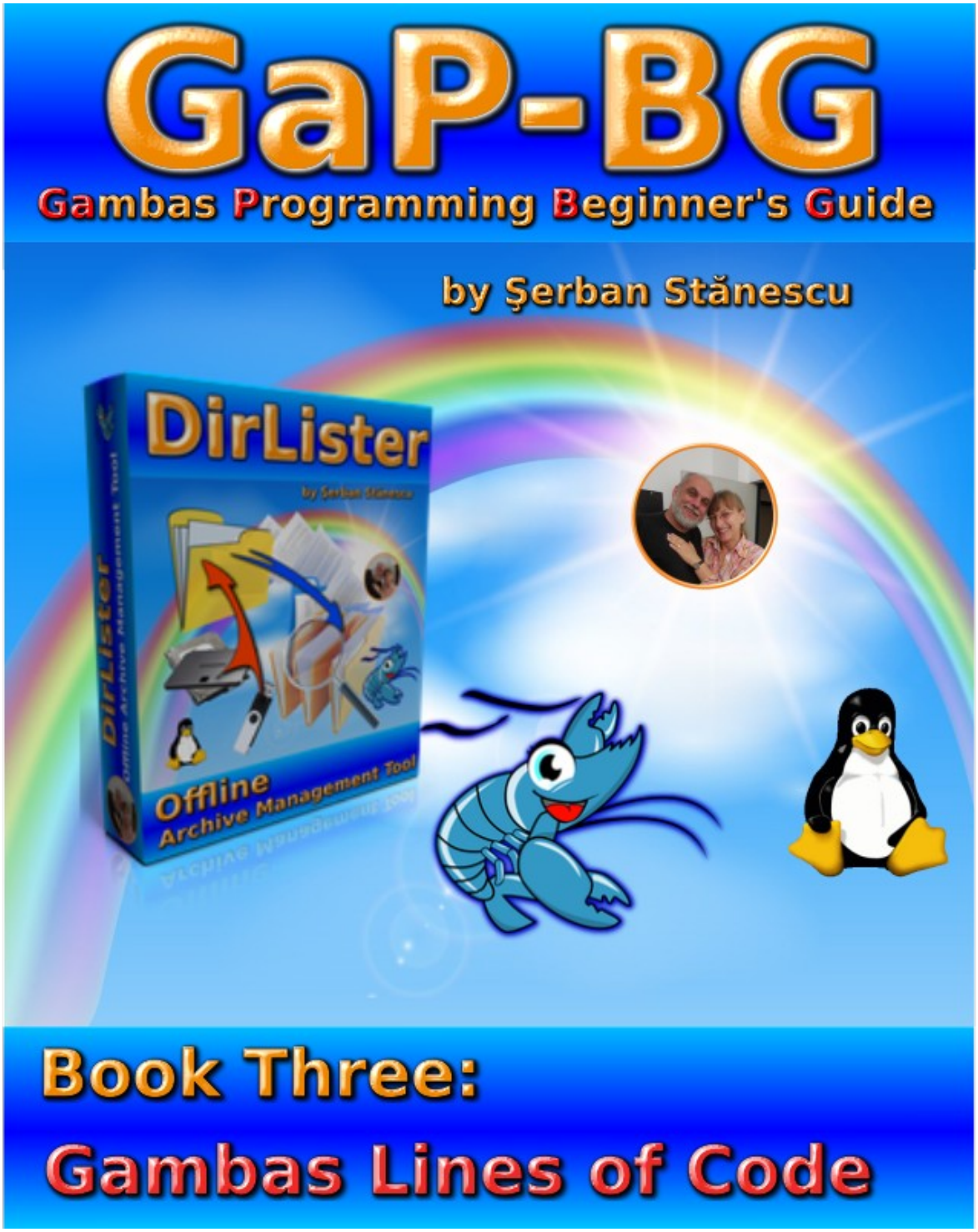

GaP-BG
Gambas Programming Beginner's Guide
by Şerban Stănescu
DirLister
Offline
Archive Management Tool
Book Three:
Gambas Lines of Code

Book Three: Gambas Lines of Code

Gambas Programming Beginner's Guide, Book Three

Contents
* Contents * EOF *

[D 05-04-2020 h 09:40]

[L 06-04-2020 h 11:30]

—

Before we begin

This book is written from ***a very different perspective*** than a mathematician's one or a programmer's one.

The love of my life is **Personal Development.**

That is: "**Know Thyself**" or "**Gnothi Seauton**" or "**Temet Noscem**" or "**Nosce te ipsum**" or "**Connaâtre toi-meme**". It's the same thing. Just a different **language**.

From that perspective, or paradigm, computers are a fancy toy; a tool that might be used to help you know yourself better and nothing more. A tool that tries to mimic the human nervous system and it does that, ***poorly***. Any computer, even supercomputers, are nothing else than a pale shade of the human nervous system. And the human nervous system, has 10,000,000 CPUs that are networked and work together. Far more than that, as scientists already have proven, the system is scalable, through what is called "***myelinic shell***" if your activity requires more "computing power" and for that reason, the real number and the real processing capacity of the human brain, the nervous system capacity is, at least virtually, unlimited. So any comparison of the human computing power and the technologic toy called "computer", is, at its best, inappropriate.

Why then use a computer, instead of learning how to harness the human internal resources, expanding human possibilities in unknown yet ways?

My answer is a simple one: "**One step at a time!**"

Over the millenia, "***Know thyself***" proved to be far more difficult than building various toys such as buildings, cars, planes and other vessels (navy, cosmic vehicles, orbital stations) and the latest gadget of the last two decades, obviously, the "*pocket computer*": the so-called "**Smart Phone**".

At this point, I can only say that I use a computer to help me develop my inner skills, either existing ones or new ones. ***Your skills***, my skills, are the only real assets that really mean something and it worths going as far as we can on that path.

IF a computer can help me achieving this goal, than it is useful. Otherwise, it's useless. Even worse: it's a waste of energy, of time...

Why write a computer program

Being organised, having a clear mind, being able to write a clear plan that you can follow and that leads you to the achivements you desire, **is a key skill.**

Programming a computer, requires at least **three skills**:

- Having the skill to ***identify and analyse a problem***, a situation that potentially drains your resources — like energy, time, money — and you want to get it out of your way. You want it done better, with the least resource consumption and best possible outcome;
- The skill to ***create a scenario*** (plan, project) based on the previous analysis, where the problem gets solved in the best available way;
- The skill to **ACT UPON THAT SCENARIO.** That is obvious! Zero actions get zero results!

So the ***question*** at this point is:

Does computer programming actually HELP ME or anyone else DEVELOP THOSE SKILLS?

Now, let's imagine a simple tool: **a traffic light** (semaphore, if you like). It shows three colors:

RED, means "*If you go you'll get hurt*" or in programming terms, "**Syntax error**". If a syntax error occures, that means that the traffic light is on RED. The program has at least *one syntax error* and you need to correct it, if you want to go any further. A syntax error, means that you still need to learn THE PROGRAMMING LANGUAGE.

The previous two books of this series, cover exactly this aspect: ***the available resources for learning the language and even more.***

As for the "Red Light", it is the same in Life: if you fail, you have to learn something you thought you know, but practice proved the opposite.

The next color, is **YELLOW:** "***Get prepared for the next step!***"

In programming, that means you passed the "Language test": the internal tools (parser, compiler, interpreter) were unable to find any syntax error so they said "*Good to go!*" and your program is running, BUT...

...It requires a lot of work. You were able to create the skeleton and it's working, which is a very good thing, but there are still a lot of things to do. Lots of features to implement, test, improve. However, you're on the go!

You now have something that WORKS and that CAN BE IMPROVED. *That is the best thing, so far.*

Last color, is **GREEN:** "***You're good to go!***".

Well, in programming that means that you already learned a lot about the language, you got quite skilled in getting the most of it, you even passed the ***program logic*** test. Your program works, it does what it was meant to do. **You made it!**

Well... ***Did you?***

In Romania there is a common saying: "**Better, is the enemy of good.**" I guess it's widely spread in any culture.

Your program *might do what you meant it to do.* Still, by the very fact that you created it, by the very fact that you use it, YOU GREW, and you keep GROWING. You developped something INSIDE YOU. **That changes your perspective.** For that reason, YOU'LL SEE THINGS DIFFERENTLY.

And you'll descover that *you can add a new feature* that enhances the program. And there you are, working again at it.

Only this time, everything goes faster. And the next time, even faster.

IF and WHEN that happens, that means that you developped some skill to a level high enough to help **YOU** solve many problems very fast.

So, **IF the computer can help you achieve THIS, than IT IS USEFUL.**

Otherwise, it's a waste of resources.

Three fundamental questions

For me, Life and computers are very much the same.

There are ***three questions that I ask myself*** in any situation and I recommend you do the same. I seek and find as many answers as I can, using any available resource to do that.

1. **WHY.** Why write a computer program? That is critical to understand. IF the program you plan to write brings you a certain benefit, than it worths writing it. Otherwise, skip the idea.
2. **HOW.** After finding some answers to the previous question and you're certain about writing the program, you need to answer this question. So, "***How do I write the program?***". While an acceptable number of answers to this question are provided by any programming language's documentation, they can only cover at most 40% of the required answers. The rest of 60%, depend on too many factors such as the efficiency of writing it (the answers to the first question), the programming logic, your creativity — there is more than one solution to do the same task — and many other aspects, such as the ability to break the final goal into smaller pieces, "tasks", "actions", usually called "Procedures" or "Subs" or "Functions" that will be easier to build and then assembly those in a whole, "***The Program***".
3. **WHAT.** In this context, "***What***", referes to ***what control*** or class, or whatever tool offers the language, am I suppoosed to use, for ***what specific task.*** Some of the controls, offer a combination of features that might be the best way to solve a certain task; some others, are very specific to a certain task, such as for an example, the TextArea control, that is specifically designed for handling multiline text and string manipulation. And as such, it is likely that you will choose to use a TextArea if you need to open text files and manipulate the contents, like in text editing tasks.

A little about designing an application

My approach to programming in this book, is a practical one:

- **Describe WHY I wrote DirLister;**
- **Describe HOW I wrote it;**
- **Describe WHAT DOES WHAT, and HOW it does.**

While there are endless variations of "*How to write a programmig book*", my idea is to provide an application [**DirLister**] that actually does what it says it does, provide the source code that was compiled and generated the binary, and then, explain at least one mechanism or, one **CHAIN OF EVENTS**. We'll get into the "Chain of Events" paradigm in the following chapters.

An important step in writing a program is **THE DESIGN.**

Two aspects are important here:

- **Computing tasks design.** While the GUI is a shell between the computer and the human user, there are procedures that actually DO the task. It is important to understand that designing the computation flow is one thing (writing the code to process data and return the results), and ***designing the GUI*** (Graphic User Interface) ***is a totally different thing.***
- **GUI design.** This is, I suppose, the most attractive part of programming graphic applications. However, designing a GUI, is a very difficult task and you need to know many things about each graphic control, because it might interfere with the computation tasks and break the program. This happens often, especially when the controls have bugs or undocumented specifics. Here is *a common situation:* A ListBox control, has a property called "Index". Usually it is used to set the current item of the ListBox, to a certain item, or to set the current item to the first one. The fact you need to know and is *extremely important and relevant for the program flow* is that almost all ListBox controls in any language,

when setting the item to a value say, [**ListBox1.Item=17**], will trigger the [**ListBox1_Click()**] event. So, you have to design your [**ListBox1_Click()**] event in a manner that can distinguish if there really was a Click event, or the event was a fake, meaning it was triggered by setting the selected item to a certain value, or else, the code contained by the [**ListBox1_Click()**] will be executed everytime you issue [**ListBox1.Item=27**] or any other value within the range of elements contained by ListBox1. Since there are many situations when you really want to avoid triggering the [**ListBox1_Click()**], you'll be facing the situation to find somehow a solution. This kind of problem is common to almost all item based controls (ComboBox, ListView, etc.). The bad thing is, the most documentation I read, says nothing about that, or leaves you without any clue on how to do it, tellying you to write code that can be executed if the item has to be changed.

Since I persented you a problem, let's see if there is any solution to it.

Step 1. Define a flag. Meaning, a variable to tell you IF there was a [**ListBox1_Click()**] OR, there was an assignment for a specific item:

```
Dim booListBoxClicked As Boolean  '--- This has to be global, so we can set it from anywhere in the program
'---
booListBoxClicked=False  '--- We put it here, otherwise when the Click() event is triggered, it will be executed.
ListBox1.Item=0    '--- That's very common, select the first item. That trriggers the  Click() event.
'---
Public Sub ListBox1_Click()
'--- Now, check what happened:
  If booListBoxClicked=True Then
    '---Do something or just ignore
  Else
    '--- Definitely, IGNORE EXECUTION!
    Return  '---We skip the event.
  EndIf
End
```

Step 2. Check **where** and **IF** we need the [**ListBox1_Click()**].

Since we setup a global flag and we changed its value to "**False**", now, each time we click inside the ListBox, the event will be ignored. So, we need to set the flag to "True", otherwise, the event will be skipped although it occures.

So, how do we do that?

This is how:

```
'---
Public Sub ListBox1_Click()
'--- Now, check what happened:
 If booListBoxClicked=True Then
   '---Do something or just ignore. We can write the code outside the conditional.
 Else
   '--- Definitely, IGNORE EXECUTION!
   '--- Before exiting the event's procedure, we change the flag's value:
   booListBoxClicked=True    '--- Now, we can exit the procedure! If it occures again, the flag is TRUE.
   Return  '---We skip the event.
 EndIf
 '--- The procedure's code might go here, or inside the If... Outside, it's easier to write/read the code.
End
```

Actually, this code is a PATTERN that can be used to filter any weird situation you encounter during development. For instance, before a Form actually shows up on the screen, the [**FMain_Resize()**] event, occures **THREE TIMES.** Did you know this?

NOW, you do!

So, if you write code on this event, bear in mind that it might get executed three times. Unless... You DESIGN IT otherwise!

The DirLister Application

The following chapters will describe the things DirLister does and how it does what it does.

While "***What it does***" you can read in the "***DirLister User's Guide***", **HOW IT DOES, goes here.**

This is how you can see the code and get a good idea on how to get from a set of requirements, to something functional, both in data processing and in GUI design.

DirLister is available for installation within the IDE, via the startup screen, "***Gambas Software Farm***", search filter "***DirLister***", "***Download and Install***" option, OR, on Gambas Farm, at:

https://gambas.one/gambasfarm/?id=771&action=search

While I'm very much aware of the fact that there might be many better solutions to get the same outcome, this is how I figured it out after less than one month of practice with Gambas. Maybe, **if I decide to go ahead**, I'll find different solutions, better ones or who knows what.

So far, it does what I need, I am using it and I am pleased how it works. Theare are some features I'd like it to have, but since for now it works, I'll be thinking of those some other time. Then decide if they are really important and it worths working on them.

Why DirLister

As I previously said, it is important to understand this "**WHY**".

Here it goes:

- ***I lost 7 drives*** in less than 6 years. A total amount of 7,918 GB of storage space. Money IS a problem. At least, for me. But the most painful loss is THE ARCHIVE. Losing an archive that has more than 5 years worth of work, well, belive me, IT IS PAINFUL.
- ***Wasting time in searches.*** While searching the main drive might return results for the most recent parts of your work, it is highly unlikely that *all of your work* is on the main drive. So, if it is on an external drive, you'll have to use it. ***IF you spent hours*** loking for something you barely remember "what's the name of...", it will wear off sooner than it should.

For me, that was reason enough to get to work: **Save money and save time, sounds good enough!**

Should I remind you the scenario where, what you were looking for say... 2 hours, actually proved to be missing from the drive you searched in and you have to move to the next? It happened to me, more than once...

The DirLister How

When the "Why" was clearly outlined, the next step was to find out "***How am I going to achieve what I want?***"

The most obvious answer, is "**Read the docs!**"

Well, that proved to be just a theory...

After spending well over 250 (Yeah... Two hundred and fifty +) hours digging the internet for a few usable lines of code, I figured out the basics and started to write some lines of code, to get a little warmed up...

First versions, were mostly sketches. I wanted to see how properties work, what events and how to use those, what controls are available and what do they offer and the like.

While I knew what was the way I needed to look like, the way to get there, proved to be longer than it was supposed to be. I'll skip the details.

The unpleasant fact is, that from an average of 180 lines of commented and tested code written daily in RapidQ, in Gambas, I suffered a severe fall down, to a merely 30 lines of code per day. Frustrating! Nevertheless, I went on. That is why I wrote and published the two previous books: to spear the next person that aproaches Gambas, from wasting 250 hours +, just to get some basic knowledge on using Gambas.

Versions from 0.1.x to 0.4.x, were only meant for this sole purpose: ***to learn how to use the different controls available, statements, commands and other language stuff.***

The design ideas, started to evolve, as I saw the means to get better results, using different controls.

From version 0.5.x, I switched from the Form control as a container for all things, to the TabPanel control. This offered me the room I needed for all the other controls and features.

As you can see, the TabPanel, has four tabs and each one is in fact a small application that does a different set of tasks, with its own interface.

And it takes only a click to move from one to the other. A big step

forward!

Now, having the looks and feel clearly outlined both in my mind an on the IDE designer, I could move to the less obvious part: connecting the various properties, events, methods and procedures, into something that actually "*does the thing*".

Gambas, like many other programming languages, is an EVENT DRIVEN environnement. Meaning that the application, "listens" continuously and if the user interacts with the computer, depending on the type of event, it will execute the code assigned to that event.

Since doing anything complex is something that requires more than one simple action, like pressing a key on the keyboard or clicking a mouse button or whatever other action, I had to figure out:

- The SHORTEST WAY to get to where I want;
- What can I do wrong on that path, that might cause the program to hang or give unexpected results? This is where things get really complicated since it is very difficult to figure out what was my mistake if any, or... ***was it a bug*** of a control or some other language element? Besides, my mistakes, are my mistakes. How am I supposed to know what someone else might do wrong? Different people, different knowledge, different level of practice, too many variables!... Complicated!

The DirLister What

Since the documentation was of little help if none, I had to figure out by myself how to connect "this with that" to get to a certain result. Eventually, a stable result.

This was a step by step process that lead me to design the interface in a certain way; then I figured out the USER actions, or "*External Events*" that I was supposed to link to the "*Internal Events*", such as the example I gave before, [**ListBox1_Click()**].

At this point, I had to figure out how can I get what I want, on the shortest path, if there is at least one. Usually, there are more ways to do the same thing and how you do it, largely depends on factors such as how many lines of code you need to write, how much CPU will it use, how much RAM will it use, how easy and intuitive would it be for the user to get things done.

And here, we get to the concept of "**Chain of events**".

Each and every action is a link in a chain of actions that is meant to lead to the desired outcome. However, this chain of events is less obvious for the user, if obvious at all...

That is why I decided to walk on this path: presenting the chain of events, from both perspectives. **User's perspective and programmer's perspective.**

This way, you will be able to ponder yourself each and every solution and if you think there is a better way to do the same thing, well, you have the code, you can test the idea. If it works, that's great!

The Chain of Events

I spoke earlier a little about the chain of events.

It would have been very nice if there was only one chain of events.

As you can guess looking at the interface, there are more chains of events. Describing all of them in one book, will make it difficult to read so I guess if I will be able (the TIME problem!) to cover all the chains of events, it is likely that there will be more than one book, just for presenting DirLister's "what and how".

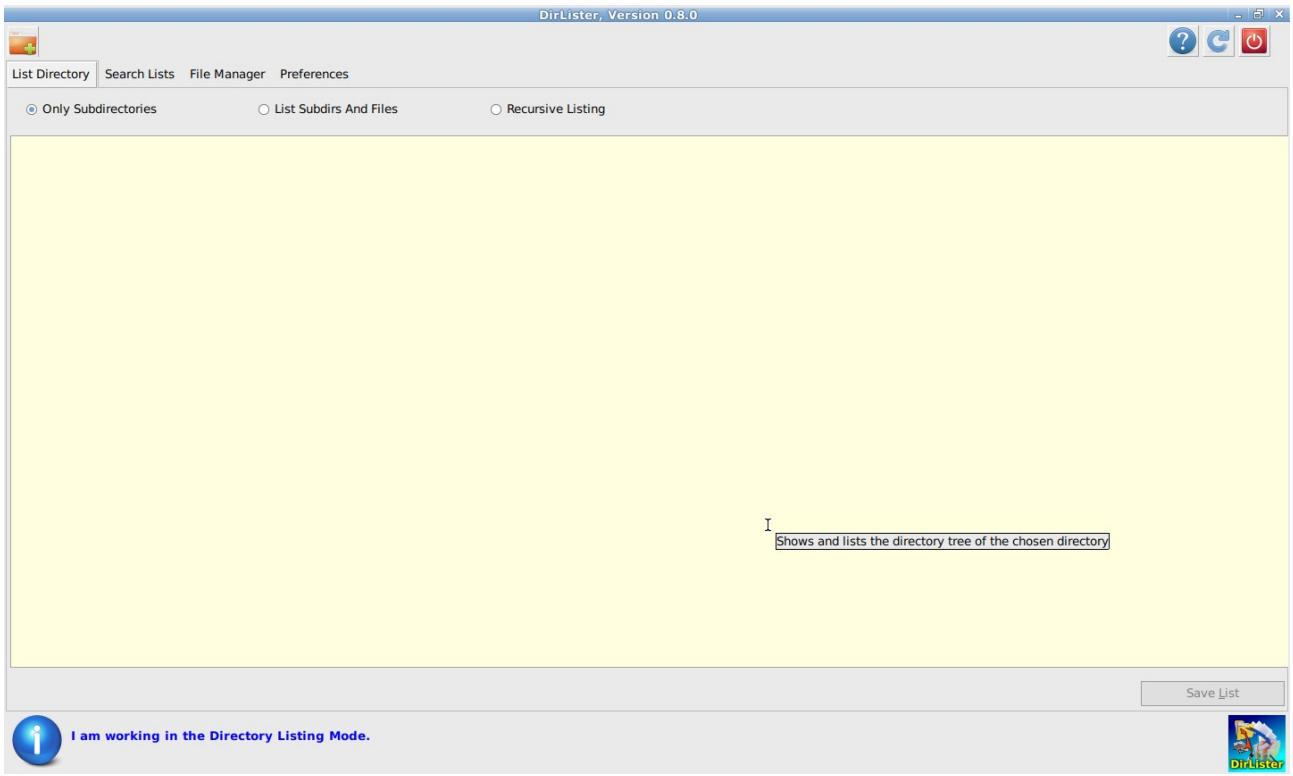
In this book, I will present **THE FIRST CHAIN OF EVENTS**, which is related to the first tab, and the basic functionality of DirLister: ***listing directories and the files within those directories, if any, working with files, how to read and write from and into a file, how to use some string manipulation assets provided by Gambas.***

So, let's move to the "**Gambas Lines of Code**" part.

DirLister: Listing directories and files

I explained in the previous chapter, "**Why DirLister**" the reasons why I decided to write this application.

Now, let's look at the first GUI element, the first tab, "**List Directory**":



The upper part, has a toolbar with ***four buttons***. From left to right, they are: [**ListDirBtn**], [**HelpBtn**], [**RefreshBtn**], [**CloseBtn**].

I wish I could say it is only one chain of events, but as you already figured out I guess, there is more than one.

Let's see them:

1. **User wants to list a directory of his choice:** Clicks the [**ListDirBtn**]. The listing procedure should start at once. BUT! What if...? We'll se the catch here, below.
2. **User wants Help:** Clicks the [**HelpBtn**]. This is strightforward and simple. We'll see the code for that.

3. **User got, for some weird reason, an ugly interface** and wants to fix this: Clicks the [**RefreshBtn**]. This is easy, too.
4. **User wants to leave the application:**. Clicks the [**CloseBtn**]. That was easy, too.

Now, let's see the code for those ***three*** buttons:

```
Public Sub HelpBtn_Click()
  Dim strHelpFile, strXDGString As String

  '--- We'll use here xdg-open "FileName"...
  strHelpFile = DirListerDefaults & "/P-055-01-DirLister-Help-1.0.pdf"
  '---
  If Exist(strHelpFile) Then
    strXDGString = "xdg-open"
  Endif
  If System.Find(strXDGString) = Null Then
    Message.Warning("The Application <" & strXDGString & ">, is missing. You need to install it, if it is
available and supported by the system!")
    Return  '-- Abort nicely...
  Else
    Exec [strXDGString, strHelpFile]
  End If
  Catch
    Message.Info(Error.Text)
End
```

As you can see, while in words and user actions things are easy, in code, all things get more complicated!

After the User clicked the Help button, we need to make sure a few conditions are met:

1. ***Do we have the file we need to open?*** While for me, ***it is supposed*** the User has it, simply ***because I put it there***, in computers, guess-work is bad practice so, *we need to test that.*
2. ***Are we on a system that uses xdg-open to open the default application for a certain file MIME type?*** We need to test that too.

3. **IF, and ONLY IF all went smoothly**, we launch the file using the [xdg-open] tool.

In the end, the only thing we can do if the unexpected shows up, is catch the error and show an appropriate message, if any available.

While this code covers all predictible events that might occure, who really knows what else might happen, that is out of reach even for a highly skilled programmer? So, even if it looks great, still something might happen to defy this "*looks great*" paradigm... However, at least I got peace of mind: I did my best to prevent what I could figure out.

[**RefreshBtn_Click**]

```
Public Sub RefreshBtn_Click()

  TabPanel1.Refresh
  LeftFManPane.Reload
  RightFManPane.Reload
  FMain.Refresh
  Wait 0.00001

End
```

As you can see, this was really easy: I called some control's methods and we're done. First versions were only with [**FMain.Refresh**] but for reasons beyond my understanding, that worked bad: did nothing with the [**FileChooser**] and [**TabPanel**] controls, so I changed to this. Ugly, but it does the job. And... frankly, the User... How does he know the code is ugly?!

There is ***another solution*** though... Using the [**gb.dbus**] and [**gb.inotify**] components to trigger the [FileChooser.Reload] whenever a file is added or deleted to the current directory shown by the control, or a subdirectory was created/deleted from a File Manager, or another external appplication...

[**CloseBtn_Click()**]

That's even easier:

```
Public Sub CloseBtn_Click()

 If TextArea1.Text <> "" Then
   TextArea1.Clear
 Endif
 If TextArea2.Text <> "" Then
   TextArea2.Clear
 Endif

 PictureBox1.Picture = Picture[strIconsRelativePath & "Warning-60x60.png"]
 InfoBox.Text = "Closing all threads, freeing all resources, leaving you... Have a great day! :) "
 FMain.Refresh
 Wait 0.000001
 Quit
```

Initially the only line of code was [**Quit**], but while testing I came to understand that if the two controls were loaded with text, especially with large amount of data, closing took a long time so, I fgured out how to say a nice "Good bye" instead of confusing the user that expected to see the app closing and apparently, nothing happened.

Since the list files used in many tests were above 75 MB in size, it seems that it took a long time to free the system resources so, I "squeezed" a little the process.

The listing process

In the previous chapter, I presented four buttons and the corresponding events.

As you could see, I described only three of the events.

Now, we'll get into the fourth, and the most important in this stage:

[ListDirBtn_Click()]

This event, does the "heavy weights lifting" job so I left it for a dedicated chapter, since there are more things to explain here.

The code is larger than the others and many things are done here, so let's see the code (on the next page).

Until next page, let me say a few words about what happens.

(1) User clicked the [**ListDirBtn**] button. Now, we need to know what type of list he wanted: just the subdirectories of a certain directory or drive? That is the default option but maybe, he forgot to make a choice and wanted a different result. Since tests proved that this is fast even on large drives, best choice is to leave this option as DEFAULT OPTION.

The other options, are triggered too by [**RadioButton**] controls. Since there are three of them, one will be selected by default so we're done here.

Nothing bad happens if User forgets to select a different option.

Next time, he will remember and click the radio button to select the appropriate option.

The [**lngItemsFound**], is a global variable to store the number of the items found on the drive and we need to reset it to zero.

The **Dialog class has a bug** and if i use once a call [**Dialog.OpenFile()**] or [**Dialog.SaveFile()**], using a filter for a speficific file type, the class freezes as "File Dialog", disabling the directories, regardless the fact that the call is [**Dialog.SelectDirectory()**] and this should NEVER happen, since the filter was set for the OPEN or SAVE FILE.

```
Public Sub ListDirBtn_Click()
  Dim FirstDirPath As String

  lngItemsFound = 0
  Dialog.Filter = [""]
  Dialog.Title = "Choose a Directory to list:"
  If IsDir("/win") Then
    Dialog.Path = "/win/" '--- Normally, would have to be "/", or "/media"
  Else
    '--- We are on a different computer. Leave the default path to /media
    Dialog.Path = User.Home '--- Normally, would have to be "/", or "/media"
  Endif
  DirTextArea.Text = ""
  'Message("[ListDirBtn_Click())]: Dialog Path = <" & Dialog.Path & ">._.")
  Select Case ListingType
    Case "DirOnly"
      '---Message.Info("You chose <DirOnly>")
    If Dialog.SelectDirectory() Then Return
      FirstDirPath = Dialog.Path
      InfoBox.Text = "[ListDirBtn_Click()] FirstDirPath: " & "<" & FirstDirPath & ">"
      '--- This is where we change behaviour according to <ListingType> parameter
      ListDirOnly(FirstDirPath)

    Case "DirAndFiles"
      '---Message.Info("You chose <DirAndFiles>")
    If Dialog.SelectDirectory() Then Return
      FirstDirPath = Dialog.Path
      InfoBox.Text = "[ListDirBtn_Click()] FirstDirPath: " & "<" & FirstDirPath & ">"
      '--- This is where we change behaviour according to <ListingType> parameter
      ListDirAndFiles(FirstDirPath)

    Case "DirRecursively"
      '---Message.Info("You chose <DirRecursively>")
    '---Return
    If Dialog.SelectDirectory() Then Return
      FirstDirPath = Dialog.Path
      InfoBox.Text = "[ListDirBtn_Click()] FirstDirPath: " & "<" & FirstDirPath & ">"
      '--- This is where we change behaviour according to <ListingType> parameter
      ListDirRecursively(FirstDirPath)
      DirTextArea.Insert("================== Found " & Str(lngItemsFound) & " items so far.
==================" & Chr$(10))

  End Select
```

```
  PictureBox1.Picture = Picture[strIconsRelativePath & "Info-60x60.png"]
  'InfoBox.Text = "I'm done! :) Please, save the list now!"

Catch
  Message.Info(Error.Text)
End
```

This is the first code sequence and what am I doing here, is to test if I'm home. Meaning ***if DirLister runs on my computer.*** If so, *I want a specific partition to be the default choice.* Else, I will default to User's Home.

After setting the default directory, we clear the [**DirTextArea**] control and get ready to fill it with our data.

The commented line is a message dialog template that I can use when something goes sideways. Here, if uncommented, it will show me the choice made with the Dialog control. The head of the message, tells me where it comes from: [**ListDirBtn_Click()**]. If there is more than ony "Spy" working as an "operational field agent", then I need to know its location.

I use this method to pause the execution if something it's unclear, or seems to go wrong for some reason. Say, a variable that was supposed to have a value, is actually empty for some reason.

```
Public Sub ListDirBtn_Click()
  Dim FirstDirPath As String

  lngItemsFound = 0
  Dialog.Filter = [""]
  Dialog.Title = "Choose a Directory to list:"
  If IsDir("/win") Then
    Dialog.Path = "/win/" '--- Normally, would have to be "/", or "/media"
  Else
    '--- We are on a different computer. Leave the default path to /media, or /home/User
    Dialog.Path = User.Home '--- Normally, would have to be "/", or "/media"
  Endif
  DirTextArea.Text = ""
  'Message("[ListDirBtn_Click()]: Dialog Path = <" & Dialog.Path & ">._.")
```

Making the choices happen

Below, we have a [**Select Case**] statement. Since there are only three cases, a conditional would have done the same thing, but the code is complicate to follow when nesting IFs.

If somehow a structure gets crippled (missing endif, or an end or whatever), will be a hell to find what is missing and where. Might be just a misplaced quote sign from some string. I'll skip the comments here...

The Select Case Strings

They come from a different place, as a global variable, [**ListingType**].

User changes the value of this variable, when clicks on a RadioButton.

The default value is [**DirOnly**].

```
Select Case ListingType
  Case "DirOnly"
  If Dialog.SelectDirectory() Then Return
    FirstDirPath = Dialog.Path
    InfoBox.Text = "[ListDirBtn_Click()] FirstDirPath: " & "<" & FirstDirPath & ">"
    '--- This is where we change behaviour according to <ListingType> parameter
    ListDirOnly(FirstDirPath)

  Case "DirAndFiles"
  If Dialog.SelectDirectory() Then Return
    FirstDirPath = Dialog.Path
    InfoBox.Text = "[ListDirBtn_Click()] FirstDirPath: " & "<" & FirstDirPath & ">"
    '--- This is where we change behaviour according to <ListingType> parameter
    ListDirAndFiles(FirstDirPath)

  Case "DirRecursively"
  If Dialog.SelectDirectory() Then Return
    FirstDirPath = Dialog.Path
    InfoBox.Text = "[ListDirBtn_Click()] FirstDirPath: " & "<" & FirstDirPath & ">"
    '--- This is where we change behaviour according to <ListingType> parameter
    ListDirRecursively(FirstDirPath)
    DirTextArea.Insert("================== Found " & Str(lngItemsFound) & " items so far.
==================" & Chr$(10))

 End Select
 PictureBox1.Picture = Picture[strIconsRelativePath & "Info-60x60.png"]
Catch
 Message.Info(Error.Text)
End
```

As you see from the code above, the internal events are branched. We have three choices and for each one, we do a slightly different job, both behind the scenes AND on the GUI, namely, in the [**DirTextArea**] control.

What is going in the background?

There are three procedures: [**ListDirOnly(FirstDirPath)**], [**ListDirAndFiles(FirstDirPath)**], [**ListDirRecursively(FirstDirPath)**].

[**FirstDirPath**], contains the previously selected directory. We pass it to the procedure, and will use it inside each procedure, but in a different manner each time.

The last line of code in the [Select Case], [**DirTextArea.Insert(...)**], writes some useful data at the end of the list: how many items were found in the selected directory.

If there are many items in the list, you need to scroll at the end or hit [**Ctrl+End**] to see the line. So, there is an example here on *how to convert numeric values into strings*, then show those somewhere, or maybe write them into a file, behind the scenes and also, how to put some data at the end of a file, using the [**DirTextArea.Insert(...)**].

Obviously, the code is different if you want to do it directly in a file, since you need to use the [**Stream**] class.

There are other procedures where I used this class, such as [**OpenListBtn_Click()**], but they are on a different chain of events, on the second Tab, [**Search Lists**] and on a different interface.

Maybe I'll cover that in another book.

Anyway, you can study the code and you can find where it appeares, using the SearchBox in the IDE, with the "**Stream**" keyword. It will list all procedures where the [**Stream**] class is used and you'll be able to see HOW.

Now, we spread some light on "How" we get to list based on user input, handled by the three ***RadioButtons***.

Let's cover this too!

The controls are: [**DirOnlyRadio**], [**DirAndFilesRadio**], [**DirRecursivelyRadio**] and the events used to change the value of the [**ListingType**] global variable, are [**DirOnlyRadio_Click()**], [**DirAndFilesRadio_Click()**] and [**DirRecursivelyRadio_Click()**].

```
Public Sub DirOnlyRadio_Click()
  ListingType = "DirOnly"
End

Public Sub DirAndFilesRadio_Click()
  ListingType = "DirAndFiles"
End

Public Sub DirRecursivelyRadio_Click()
  ListingType = "DirRecursively"
End
```

It's as simple as that!

The value is then passed to the procedure [**ListDirBtn_Click()**] when the event occures.

The value is tested in the [**Select Case**] statement and if the match is found, the subsequent code is executed.

This is why the "**Case Else**" was ignored here: we have a default value for [**ListingType**]. It is set on the line 194, in the [**InitializeDirLister()**] procedure. So, that means that we already have a value for it, when the form shows up and it's OK to proceed.

Now, we almost got to the end of the [**ListDirBtn_Click()**] event.

We called the appropriate procedure and after the procedure is executed, the event call is done.

The only code after exiting the [**Select Case**] is the precaution of catching the error if any, and show the error, if it is a known one and is internally handled somehow by the Gambas environnement. While most errors

can be handled, some weird collisions might lead to an error number with many figures, a LONG type variable. This usually signals an unknown error and even if there is nothing to do, it keeps the program running, instead of crushing it.

Since I still have to cover the error handling section when I'm writing this, that is all I can say for now.

The next chapter, is about the three procedures that we launched from within the [**Select Case**] statement.

As you can see, the "**Chain of Events**" approach, is slightly different from what the user might expect and although it includes the User's perspective, we go way beyond that. "*A larger picture*", so to speak.

Now, let's move to the next three pieces of the puzzle: [**ListDirOnly(...)**], [**ListDirAndFiles(...)**] and [**ListDirRecursively(...)**].

The Listing Procedures

[ListDirOnly(...)]

This is the default procedure in the [**Select Case**] statement.

While this might look useless now, I needed it to do another job, in the [File Manager] mode: to set up some bookmarks. As I explained in the User's Guide, I have a complex directory structure. For this reason, I want to be able to set up a set of directories for any of the panels of the File Manager. It spears me of lots of clicks to get where I want.

Another reason is that when I work a lot in a project and the files I create get too many, I need to re-structure the contents of a certain directory so I create a number of subdirectories. If this solves the problem, OK, but sometimes I need to go on more than one level of depth and here is the explanation.

Instead of clicking 10 times to get to the last subdirectory, I click on the best match in the bookmarks list and there I am!

Besides, maybe I need to see how exactly looks the directory tree after say, one year or maybe 10 years later from the day I created it and it's easier with such a list. I just load it into the internal editor and that's all.

So, let's see now how is it done.

The code on the next page, starts with defining some variables we'll use here, locally **Path1, FileList1, FileName, DirLine**.

If you remember from the previous chapters, I set up a global variable to be able to pass the User's choice: [**MainDirPath**]. It stores the path given by the [**Dialog.SelectDirectory()**] call.

Now, we use it to list the subdirectories in this [**MainDirPath**].

Next, I set up an icon to be shown in the StatusBar, in case the procedure takes longer that a few seconds. It seldom shows, but anyway, if there is a long list of subdirectories, might show up. Looks nicer!

The Loop

As you can see, we enter a loop, where each item provided by the [**Dir(MainDirPath)**] is analysed and if it is a directory, it is listed in the [**DirTextArea**] control, as a new line.

Next, there is **a filter** there. Since I was unable to get more information on how to use the [**Dir()**] command, if a directory or file belonging to Root or eventually another user on the same computer that has restricted rights gets on the list, the procedure triggers an "Access violation" error. As I said before, I still have to study error handling so, I got what I could, through guess-work.

The [**FMain.Refresh**] call, is here to force the display of the images. I probably forgot to change, because I later replaced it with [**StatusBar.Refresh**] that seems to work better in this case. Glitches...

```
Public Sub ListDirOnly(MainDirPath As String)
  Dim Path1, FileList1, FileName, DirLine As String
  FileList1 = ""
  PictureBox1.Picture = Picture[strIconsRelativePath & "Docs-Search-64x64.png"]
  FMain.Refresh
  Wait 0.001
  For Each FileName In Dir(MainDirPath)
    Path1 = MainDirPath &/ FileName
    FileList1 = MainDirPath &/ FileName & CRLF
    '--Test if [Path1] is a directory:
    InfoBox.Text = "INFO! Listing directory: " & Path1
    If IsDir(Path1) = True Then    '--It is a directory
      DirLine = Path1 & "/" & CRLF
      DirTextArea.Insert(DirLine)
      If InStr(Path1, "lost+found", gb.IgnoreCase) Then
        '--- Found the <lost+found> directory! Skipping to avoid <Access denied> error!!!
      Endif
      lngItemsFound = lngItemsFound + 1
    Else    '--It is a FILE. We skip it.
    Endif
  Next
  DirTextArea.Insert("================== Found " & Str(lngItemsFound) & " items so far.
==================")
  PictureBox1.Picture = Picture[strIconsRelativePath & "Info-60x60.png"]
  InfoBox.Text = "Found " & Str(lngItemsFound) & " items so far."
End
```

The items counter. In order to be able to show the number of items found and listed, I used the same global [**lngItemsFound**] variable, then converted it into a string. After the procedure finishes its main job (the loop ends), we write the number of items found, both in the [**StatusBar**] and in the [**DirTextArea**] control, and we're done!

If the user decides to save the results, now we can do that too. The button [**SaveDirListBtn_Click()**] will do the job and when the procedure is ready, the button becomes available = **Enabled**. This is triggered from the [**DirTextArea_Change()**], with the lines below:

```
Public Sub DirTextArea_Change()
  If SaveDirListBtn.Enabled = False Then
    SaveDirListBtn.Enabled = True
  Endif
End
```

So, whenever the contents of the [**DirTextArea**] changes, the button gets enabled. Otherwise it remains disabled.

[ListDirAndFiles(...)]

This is the next procedure. Simpler than the first, since we list all items found. A tweak was made here though, to make the directories easy to distinguish from the file entries. I added a forward slash at the end, so I can see at a glance that the item in the list IS A DIRECTORY. Otherwise, it's almost the same procedure as the previous.

```
Public Sub ListDirAndFiles(MainDirPath As String)
  Dim Path1, FileList1, FileName, DirLine As String
  FileList1 = ""
  PictureBox1.Picture = Picture[strIconsRelativePath & "Docs-Search-64x64.png"]
  FMain.Refresh
  Wait 0.001
  For Each FileName In Dir(MainDirPath)
    Path1 = MainDirPath &/ FileName
    FileList1 = MainDirPath &/ FileName & CRLF
    '--Test if [Path1] is a directory:
    If IsDir(Path1) = True Then    '--It is a directory
      DirLine = Path1 & "/" & CRLF
      DirTextArea.Insert(DirLine)
      If InStr(Path1, "lost+found", gb.IgnoreCase) Then
        '--- Found the <lost+found> directory! Skipping to avoid <Access denied> error!!!
      Else    '--- Do nothing. It is a subdirectory.
      Endif
    Else    '--It is a FILE
     DirTextArea.Insert(FileList1)
    Endif
    lngItemsFound = lngItemsFound + 1
  Next
  '---
  PictureBox1.Picture = Picture[strIconsRelativePath & "Info-60x60.png"]
  InfoBox.Text = "Found " & Str(lngItemsFound) & " items."

End
```

Although I use it seldom, sometimes I need to know all the files that are in a certain Project's directory. I use a "***Project based***" approach to all my work, so everything on the computer's storage, is organised as a Project, having a "***Main Directory***" and then subdirectories and files.

For someone who uses Blender, or Inkscape, or any video editing software, audio editing software, let alone Gambas, sounds very familiar.

The Gambas directory, already has level 3 subdirectories, and it might go even further, depending on the amount of time I can assign to Gambas programming and the complexity of the subprojects.

[ListDirRecursively(...)]

This one, is a bit tricky...

I wrote something like this some 20 years ago, for the "Father" of DirLister, "**CD Manager**". By the time I wrote it, the [**Dir()**] command was the only one I knew. Later, I found [**RDir()**] but since the procedure was already in place and working, I skipped the rewriting. Maybe someday I will test this too. It has a lower priority on my tasks list.

The idea of making it work, is to filter the type of item the current iteration found: is it a file? We list it. Is it a directory? We list it, but what if it has a subdirectory?

While the problem looks difficult to solve, the solution is to call the procedure itself, if we have a directory with "children". That approach is a pattern (template, archtype) for many situations when you need to do something that branches on a tree pattern model.

What actually happens is that the current directory has changed to [**Path1**] and if [**Dir(Path1)**] is invoked, will restart the loop, but this time, with a different item. It will go as deep as needed, since we called it for all directories found. It will go to the level x-1 when it finds only a file, and so on, until the whole tree is listed.

```
Public Sub ListDirRecursively(MainDirPath As String)
  Dim Path1, FileList1, FileName, DirLine As String
  FileList1 = ""
  PictureBox1.Picture = Picture[strIconsRelativePath & "Docs-Search-64x64.png"]

  For Each FileName In Dir(MainDirPath)

    Path1 = MainDirPath &/ FileName
    FileList1 = MainDirPath &/ FileName & CRLF
    '--Test if [Path1] is a directory:
    InfoBox.Text = "INFO! Listing directory: " & Path1
    If IsDir(Path1) = True Then    '--It is a directory
      DirLine = Path1 & "/" & CRLF
      DirTextArea.Insert(DirLine)
      '--Recursive call of ListDirRecursively()
      If InStr(Path1, "lost+found", gb.IgnoreCase) Then
      Else
        ListDirRecursively(Path1)
      Endif
    Else    '--It is a FILE
     DirTextArea.Insert(FileList1)
    Endif
    FMain.Refresh
    Wait 0.000001
    lngItemsFound = lngItemsFound + 1
  Next
  PictureBox1.Picture = Picture[strIconsRelativePath & "Info-60x60.png"]
  InfoBox.Text = "Found " & Str(lngItemsFound) & " items so far."
End
```

As you see, the code is pretty simple and the only weird thing is the recursive call, that is less transparent. The key to this is that the value of [**Path1**], changes at each iteration (pass) of the loop chain. The procedure will be released only when the last item found by [**Dir(Path1)**] has been listed.

If you look closely, in fact there are two loops running one inside the other: the first loop is the one that called the procedure when the item was a

directory; the second loop, is the one that starts within the recursive call and this mechanism goes on and on, until the maximum level of depth is reached so if we have a 6 level tree, we'll also have a 6 level loop running, one inside the other. This is less obvious and might lead to confusion. Using a signaling method (the StatusBar) and a depth counter, you can check this visually, with a global variable that gets increased at each recursive call [**intLoopDepth**]:

```
Public intLoopDepth As Integer  '--- Global
'---
'--- I skipped the irrelevant lines...
      '--Recursive call of ListDirRecursively()
      If InStr(Path1, "lost+found", gb.IgnoreCase) Then
      Else
        '--- This gets increased only if a recursive call appears.
        intLoopDepth= intLoopDepth+1  '--- We need it here, to count the calls
        ListDirRecursively(Path1)
      Endif
      '---
```

Theoretically, this is an infinite loop: the procedure calls itself and if the ***exit condition, last item IS a file,*** somehow fails to appear, will call itself indefinitely. Since even if there are hundreds of thousands of items, or even millions, this is a limited (finite) amount of "something" and at some point, the loop ends, even if that takes longer.

This is why I chose to show the current item in the [**StatusBar**] at the cost of speed: the user ***can see if the loop actually does something.*** Otherwise, it might look like frozen.

Showing the Results and saving

We described so far the first part of the interface and the undelying code that does the computational part. I explained the buttons on the upper toolbar, then the dedicated toolbar with the Radio Buttons. Below the Radio Buttons, there is a TextArea, [**DirTextArea**].

We use it to build our results list.

The lines of code are almost the same line. They are different only in what's regarding the content written in the control, which is different, depending on what User chose to list and if the request was to list both files and directories, the variable that contains the current item, is different:

```
'--- This contains a Directory:
DirTextArea.Insert(DirLine)
'--- This contains a File:
DirTextArea.Insert(FileList1)
'---
'--- Adding the statistics:
DirTextArea.Insert("================== Found " & Str(lngItemsFound) & " items so far.
==================" & Chr$(10))
```

If you check the code, you'll se the same lines in all three procedures.

The only different lines of code, are those where we add the statistics, at the end of the list. It's done on the [**ListDirBtn_Click()**] event, ***after*** calling the appropriate procedure.

Now, if you remember, we spoke about saving the results into a file.

As you can see, before hitting the [**ListDirBtn**] to trigger the [**ListDirBtn_Click()**] event, the [**DirTextArea**] is empty and the [**SaveDirListBtn**], is disabled (grayed).

However, when the first item is found, it is written into the [**DirTextArea**], which means that its contents changed, and the [**DirTextArea_Change()**] event was triggered. When this happens, the value of the [**SaveDirListBtn.Enabled**] property is changed to [**True**]. It will

remain [**True**], as long as the listing goes on.

When finished, IF User chooses to save, than will click on the [**SaveDirListBtn**], triggering [**SaveDirListBtn_Click()**] event.

This kind of procedure is called "Event handler" because it allowes us to put some code inside it, to handle the workflow at this very specific point.

Now, we want to save the results we got.

This is how we do it:

```
Public Sub SaveDirListBtn_Click()
  Dim FileExt As String

  Dialog.Path = RTrim$(DirListerUserDir & "/")
  '---Message("SaveDirListBtn_Click(): DirListerUserDir: " & Dialog.Path)
'=================================================================
    '--- This is where we change behaviour according to <ListingType> parameter
    Select Case ListingType
    '------------------------------ .dirlisterbmk (13)
    Case "DirOnly"
      Dialog.Filter = ["*.dirlisterbmk", "Directory Bookmarks"]
      If Dialog.SaveFile() Then Return
      FMain.Refresh
      Wait
      If Right$(Dialog.Path, 13) = ".dirlisterbmk" Then
        File.Save(Dialog.Path, DirTextArea.Text)
        CrtListFile = Dialog.Path
      Else    '--- User ignored the extension; we add it.
        FileExt = Dialog.Path & ".dirlisterbmk"
        File.Save(FileExt, DirTextArea.Text)
        CrtListFile = FileExt
      Endif
      SaveDirListBtn.Enabled = False
      PictureBox1.Picture = Picture[strIconsRelativePath & "Info-60x60.png"]
      InfoBox.Text = "INFO! Directory list saved as: " & CrtListFile
      FileExt = ""
      FMain.Refresh
      InfoBox.Text = "DirLister V " & Application.Version
      FMain.Refresh
      Wait
    '------------------------------ .subdirfileslist (16)
    Case "DirAndFiles"
      Dialog.Filter = ["*.subdirfileslist", "SubDirectory And Files List"]
      If Dialog.SaveFile() Then Return
      FMain.Refresh
      Wait 0.00001
      If Right$(Dialog.Path, 16) = ".subdirfileslist" Then
```

```
      File.Save(Dialog.Path, DirTextArea.Text)
      CrtListFile = Dialog.Path
    Else    '--- User ignored the extension; we add it.
      FileExt = Dialog.Path & ".subdirfileslist"
      File.Save(FileExt, DirTextArea.Text)
      CrtListFile = FileExt
    Endif
    SaveDirListBtn.Enabled = False
    PictureBox1.Picture = Picture[strIconsRelativePath & "Info-60x60.png"]
    InfoBox.Text = "INFO! Directory list saved as: " & CrtListFile
    FileExt = ""
    FMain.Refresh
    InfoBox.Text = "DirLister V " & Application.Version
    FMain.Refresh
    Wait 0.00001
  '----------------------------- .dirlist (8)
  Case "DirRecursively"
    Dialog.Filter = ["*.dirlist", "Recursive Directory List"]
    If Dialog.SaveFile() Then Return
    FMain.Refresh
    Wait 0.00001
    If Right$(Dialog.Path, 8) = ".dirlist" Then
      File.Save(Dialog.Path, DirTextArea.Text)
      CrtListFile = Dialog.Path
    Else    '--- User ignored the extension; we add it.
      FileExt = Dialog.Path & ".dirlist"
      File.Save(FileExt, DirTextArea.Text)
      CrtListFile = FileExt
    Endif
    SaveDirListBtn.Enabled = False
    PictureBox1.Picture = Picture[strIconsRelativePath & "Info-60x60.png"]
    InfoBox.Text = "INFO! Directory list saved as: " & CrtListFile
    FileExt = ""
    FMain.Refresh
    Wait 0.00001
    InfoBox.Text = "DirLister V " & Application.Version
    FMain.Refresh
    Wait
  '----------------------------- That's it! Finish Select Case Statement...
  End Select
  Dialog.Filter = [""]
'====================== [SaveDirListBtn_Click()] ==========================

Catch
  Message.Info(Error.Text)

End
```

What the code does

Probably the code looks complicated but in fact, is very simple.

What it does is to *add different extensions* to the generated lists, in order to avoid the messing in the other functionalities of the **DirLister**, such as directory bookmarks, in the [**File Manager**] mode.

Since I'm a lazy kind of person, I chose to check the extension and if it's missing, to add it automatically.

That is why the code *looks* complicated.

The only relevant lines of code here, are where we **save the data into a file:**

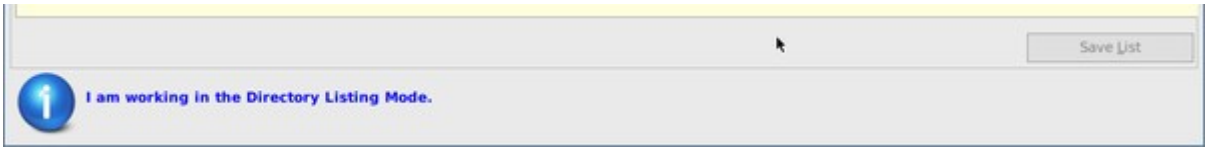
```
File.Save(FileExt, DirTextArea.Text)
```

To be able to use the same line of code, I used a variable for the file+extension, instead of the string itself. So, the call [**File.Save()**] has the "*Where to save*" part, the path of the file **AND** the desired extension, checked before. The "*What to save*" part, is contained in the [**DirTextArea**] and we get it from there, using the [**.Text**] property call.

The Status Bar

We went through the code, following the interface shown to the user and the last item we described, was the [**SaveDirListBtn**] control.

We still have some events in the same chain of events, that occure here, in the [**StatusBar**]



As you can see in the screen capture, we have an image, than a text.

The [**StatusBar**] itself, consists of three controls. Very simple!

[**StatusBar**] is a [**Panel**] control, which is a **container.**

Inside this container, I placed the [**PictureBox1**] control that shows the images, depending on the context of the program flow, then beside it, the [**InfoBox**], which is a [**TextLabel**]. This is how relevant images and the status messages are shown.

This is how the program "talks" to the User, while doing something that takes a longer time than would be acceptable to wait.

As you can see studying the whole code, there is a large number of assignments, both for images and for text displaying.

Each change, is triggered by the actions that are running or were performed by DirLister, at some point of the workflow.

Some messages, are for fun, to get the user know that something happened (Info icon), some others are warnings (Warning icon) some others, just show that somekind of listing is running (Docs-Search icon). When some kind of job has been fulfilled, sometimes the DirLister logo appears.

This is a way to add some interactivity to the program, when time

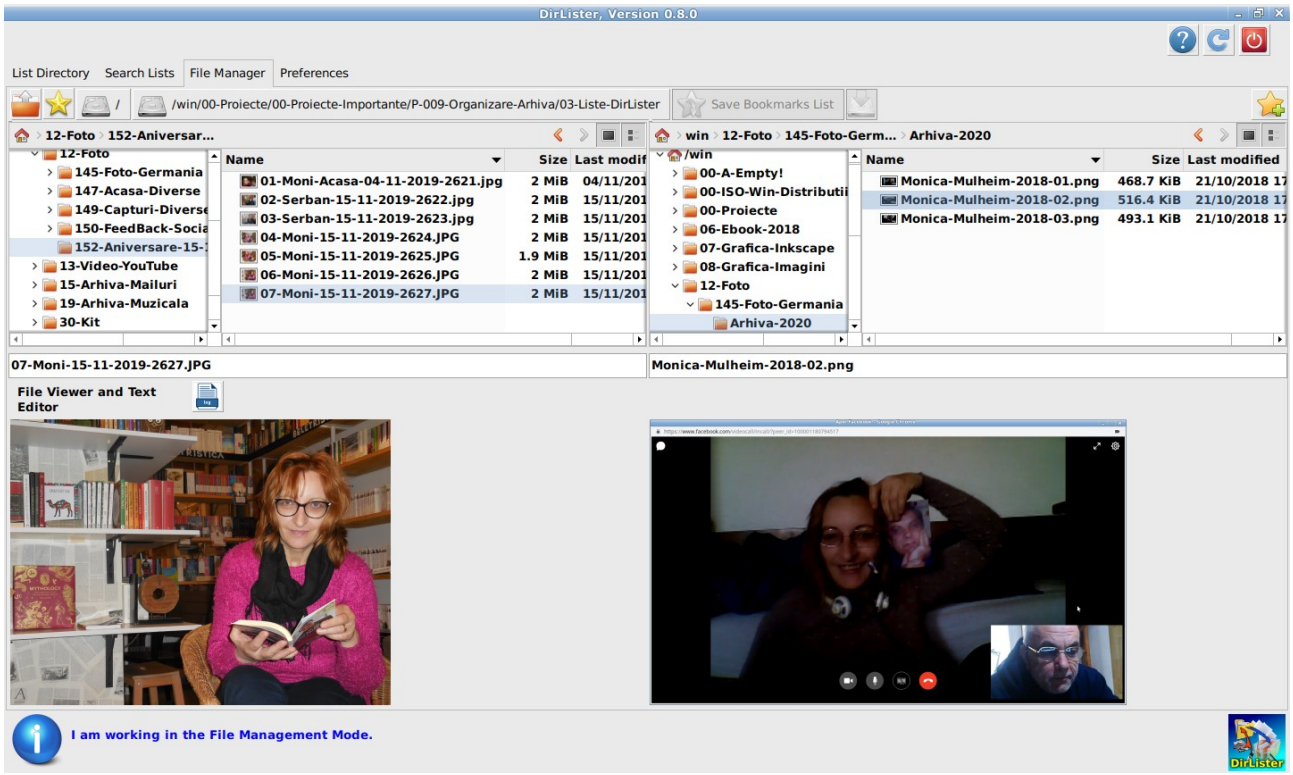
consuming operations are running.

Obviously, if you want to add more fun, you can create a procedure where you change randomly a funny text read from a file, and show it there.

Or maybe, a "Did you know?..." or "Tip...", or "Famous quotes...".

Further more, adding a Timer control, allows to show the message on a predefined amount of time, than change it when the interval is consumed.

That gets the User distracted in a nice way, from the very fact that the current operation takes a long time.



File Manager, Image Compare feature. It's my wife...

Designing the Help System

I left for the end of this book this critical part of writing, testing and deploying an application, since it is a very general theme. It applies to all kind of software so, it has nothing specific to **DirLister** or any other particular application, written in Gambas or whatever programming language.

Designing the Help system is critical, mostly in those days since the User needs **to know "at a glance" what and how.** While this might look easy at first thought, as in "***Write a PDF and you're done!***", facts are a little different.

It's all about **PERCEPTION.**

While this might be less obvious, the Help System is actually ***a marketing tool.***

It will decide largely if your application will be embraced by a larger or a smaller number of users. And this is the critical part!

While writing an application for your own needs might be suitable for someone who has a fortune consisting of hundreds of bank accounts filled with countless zeroes following some "1" up to a "9" figure, for the most of us, writing an application *just for personal use*, can mean only one thing: the losses caused by its absence make the development of the app cheaper than assuming the losses.

This is my case, by the way... I already lost about 800 (more, but it's irrelevant) euros in storage media. Preventing it happen, ***cuts down my storage media expenses with at least 65% of the above figure.*** That means about 520 euros. And it took me less than 6 weeks to write and test **DirLister** (regardless the documentation phase). The trick is that *it cuts the losses every year from now on.* Assuming an average yearly cost of 150 euros, the savings are almost 100 euros/year. Meaning that in the next 10 years, I'll save 1000 euros. That's all. Figures; basic maths.

For all other reasons, *marketing defines the success or failure* of any product or service. There are countless examples on the internet, I'll point you

only at one: **Simon Sinek's TED**, "***Why...***":

How great leaders inspire action:

https://www.ted.com/talks/simon_sinek_how_great_leaders_inspire_action

Means to offer Help

There are basically three approaches when it comes to offer help:

1. **ToolTips;**
2. **Manuals (PDF, or whatever, HTML maybe);**
3. **Contextual Help.**

The last choice, is the worst from the programmer's perspective and the best from **the User's perspective.**

From the programmer's perspective offering contextual help, involves a very complex mechanism that is needed to track each move on the screen and having a very detailed documentation, strictly bounded by the context. That's taugh work to do for a single person. Becomes way too expensive since it eats tons of resources and produces only a fancy (yet delightful...) result. And worst of all, the underlying mechanism, involves database manipulation. Is it worth?

Probably the worst part of contextual help is that tracking everything that happens on the screen is *the easiest part*, although it sounds already awful. The ugliest part though, is that you need a very precise mechanism to clearly separate the events from the HELP CALLS. And this complicates the code endlessly. If the app is a complex one, maintaining the code becomes a hellish job.

So what can we do?

Here is what I did:

Compromising among solutions

I chose something in-between: **Write meaningful ToolTips AND** writing a User's Guide that is **easy to navigate and search**.

If you look at the User's Guide, you'll see enough links that allow you to go back and forth wherever you want, from the beginning to the end of the document. EOF stands exactly how you might have already guessed, for "***End Of File***".

Studying a common user's behaviour (I was doing computer service for some years), I came to understand that almost all users want to go on the shortcut (BTW: Is there a shortcut to a shortcut? I'd love it!), if any available. Meaning that reading the manual, would usually be the last choice...

Under such circumstances, even contextual help, fails to provide what it is supposed to.

This is where the ToolTip feature comes in: I wrote enough information for almost every interface element, to avoid sending the User to the User's Guide. *I know it's there*, but — guess what? — **the User knows that too!**

So, how elegant would it be to push him to open the User's guide, all around the App?

While I know it's difficult to keep balance between being verbose in a tool tip and being too concise, the best approach to care for User's TIME and ENERGY, is to use ToolTips.

And that is what I did. Maybe I forgot to write one here and one there, but as a general rule, all important interface elements, have their own ToolTip so, it is enough to hover the interface to get a good idea of "***What does what***".

Well, enough at least for a quick overview of the Application.

THIS little, tiny amount of time, the few seconds spent at first launch, decide almost 100% the fate of the Application. Will I use it or am I supposed to look further for a better one? **A more INTUITIVE one?**

Final words

Well, we got to the end of the first "***Chain of Events***". I went through each part of the interface, the user interactions and what those interactions change into, behind the scenes.

I hope that this approach will help you figure out how to approach **your own projects.**

This time, it seems I got lucky, I managed to stick to the idea of "***Short book***".

If I decide to go to the next "***Chain of Events***", it will be, obviously, the next interface, [**Search Lists**].

A lot of stuff happens there, there are more options and more procedures to describe so keeping it short, is impossible, unless I cripple it. Assuming a relative degree of predictibility, the next book might have some 100 pages or so, which has nothing to do with "***short book***"...

Anyway, **thanks for reading** and I hope this book helped you figure out some solutions on how to use strings, and some basic controls to build a simple application, that you can further enhance, **gradually**!

It is exactly the path I followed: I started from a Form, then added some controls, tested properties, methods, events, played around with those, combined them.

I moved then to ***File Operations***, ***String Manipulation*** and then did my best to gather the knowledge into a project I was interested to make it happen: **DirLister.**

My best wishes and successful Gambas coding!

Şerban Stănescu

Contents	EOF		End Of File