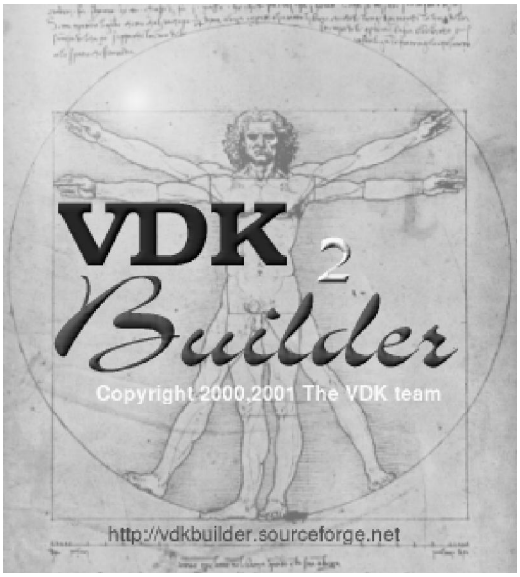


VDK −VDKBuilder series 2 Tutorial

Version 1.1 March 2002

intentionally left blank page

Version 1.1
March 2002

By Mario Motta, VDK Team
mmotta@guest.net

intentionally left blank page

INDEX

Introduction page 7

System requirements page 7

How to build VDK and VDKBuilder page 8

Getting help and technical support page 9

The VDK team page 9

The skeleton project
- Getting started page 10
- Skeleton project source files page 13
- Skeleton project GUI source files page 17
- Skeleton application first run page 20
- How to nicely close skeleton application page 21
- VDKBuilder project options page 23
- Making skeleton form nicer page 25
- Adding widgets to skeleton form page 27
- Speaking about signals and events page 31
- Go on with skeleton application page 33

The menu project
- Menus and child forms page 51
- About child forms page 55

Using a fixed container page 63

Widgets for drawing page 65

How use unsopported widgets (placeholders) page 71

VDK−Libsigc++ signal system extension page 77

APPENDICES

A − Signals and events page 91

B − About properties page 99

C − Autoconf and Automake support to build GNU packages page 103

D − NLS − How to make a Native Language Support application using VDKBuilder page 105

E − Vdkxdb library, using VDKBuilder with data−aware widgets page 109

F − Vdksdl library, using VDKBuilder with SDL library page 127

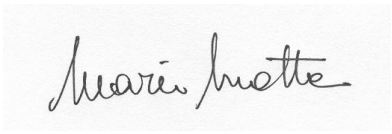
G − VDKInput tutorial by Jonathan Hudson page 135

H − Going on Jonathan Hudson work page 145

Author's note

- This tutorial has the purpose of helping newbies to quickly learn how use VDK and VDKBuilder.[1] However also more experienced users should find it useful since some advanced tecniques and programming tips are covered. Topics follow a planned logic, the reader should follow this flow, jumping from one topic to another can be somewhat confusing. The tutorial is divided in two parts, the former shows how to use VDK and VDKBuilder, the latter contains several appendices where some topics are discussed more indepth. We hope that you will find this tutorial useful, readers are encouraged to send comments, suggestion and even complaints to Mario Motta mmotta@guest.net. A final note: I apologize for my terrible english, mother−tongue readers please don't be too pretentious.

Thanks



- Author thanks George Boutwell for his proofreading of this tutorial.
- May be that some figures into this tutorial are slightly different since at the time of this writing VDKBuilder work was in progress, this shouldn't be a problem however, when necessary figures have been updated.
- This tutorial is a live document, i hope to make it better using users feedbacks.

1 Even if this tutorial was made for VDK/VDKBuilder series 2 it is largerly applicable to VDK/VDKBuilder 1.2.5 as well.

INTRODUCTION

VDK is a C++ binding of the famous GTK+ widget set libray, strictly speaking it is not just a wrapper rather a framework that hides in the background as much GTK+ work as possible. VDK was designed to be used by newbies with little or no knowledge of GTK+, while at the same time more experienced users can take advantage of their knowledge of GTK+ calls and conventions. Using VDK one will realize that there is not always a one−to−one relationship with GTK+, this is by design, however it is alway possible to access and use underlying GTK+ objects. This hybrid nature of VDK permits it to be useful to both newbies and more experienced users.

VDKBuilder is a RAD (Rapid Application Development) tool based on VDK that helps programmers in constructing GUI interfaces, as well as editing, compiling, linking, and debugging code all within an integrated environment. Using VDKBuilder dramatically reduces development time since all code related to GUI construction and signal processing is automatically generated, maintained and updated.
VDK is licensed under LGPL, VDKBuilder under GPL

VDKBuilder features

- **Project Manager**
  Permits the creating of new projects or the opening of already existing ones. It takes care of the editing, compilation and linking phases of the whole project, and automatically generates and maintains updated GUI source files and the related makefile.
- **Source Editor**
  A multi−file editor with C/C++ syntax coloring, search & replace, undo, jump to errors, word completions, hints and more.
- **GUI Designer**
  Permits construction of Graphical User Interfaces by simply "dragging" widgets onto forms while VDKBuilder automatically generates all GUI related code for you.
- **Widget Inspector**
  Allows editing of widgets' properties using a very convenient field attributes interface
- **VDKBuilder Maker**
  Interacts with some external programs such as compiler, debugger and others tools simplifying project compiles, links and debugging within VDKBuilder itself. External programs outputs are handled by VDKBuilder Maker and used by VDKBuilder, for instance all warnings and errors from the compiler are included into the source editor and features such as jump to error work seemlessly.
- **Plugins**
  VDKBuilder can be extended by user developed plug−ins. These plug−ins are developed as shared libraries and can be used to extend the supported widgets. VDKBuilder by default has plugins with additional widgets including data−aware widgets based on xBase database library. (see http://xdb.sourceforge.net)
- **Others features**
  A fully customizable environment, Automake/Autoconf support for generating GNU packages, Emacs support (the ability to make Emacs the default source editor) and more..

SYSTEM REQUIREMENTS

To build and install VDK and VDKBuilder you need:

1. A Linux box, VDK/VDKBuilder have been tested on RedHat, Mandrake, Slackware distributions but have been nicely built on many others[2] systems.
2. X libraries with include files, normally all modern Linux distributions install them by default
3. GTK+/Glib libraries, you need version 2.0 or later, be sure to have installed the development version with include files and others tool.
   GTK+ can be downloaded from: http://www.gtk.org
4. Optionally you can have also libsigc++ version 0.85 or later. Libsigc++ can be downloaded from: http://libsigc.sourceforge.net/
5. Optionally (but recommended) doxygen installed, this nice tool allows you to build a complete reference manual in html format with many hypertext links.
   Doxygen can be downloaded from: http://www.doxygen.org/download.html
6. A c/c++ compiler, normally Linux boxes come with GNU gcc.

2 VDK/VDKBuilder are know to compile and run also in FreeBSD, Solaris, HPU−Irix, Mac−Linux Yellow Dog

HOW BUILD AND INSTALL VDK/VDKBUILDER

Here we discuss how to build and install from source code tarballs, rpm or rpm−like distributions aren't covered and not officially supported by VDKTeam.

1. Have VDK/VDKBuilder tarballs: vdk−2.0.0.tar.gz and vdkbuilder−2.0.0.tar.gz
   They can be downloaded from http://vdkbuilder.sourceforge.net
2. VDK
   Unpack:
   ```
   $ tar xvzf vdk−2.0.0.tar.gz
   ```
   I suggest you read INSTALL, NEWS and README files before proceeding, they can contain last minutes changes and other useful information.

   Build: [3][4]
   ```
   $ cd vdk−2.0.0
   $ ./configure  −−prefix=<your prefix> [others options]
   $ make
   ```

   Install: (you need root permission)
   ```
   $ su
   # <enter password>
   # make install
   # exit
   ```

   Test:
   I suggest to take a look at vdk test program into vdk−2.0.0/testvdk directory:
   ```
   $ cd testvdk
   $ ./testvdk &
   ```
   Testvdk program assures that all widgets work well, gives you a complete overview of VDK features and lets you browse the source code as well. Furthermore it is a good source of information how to use VDK widgets.

3. VDKBuilder
   Unpack:
   ```
   $ tar xvzf vdkbuilder−2.0.0.tar.gz
   ```
   I suggest you read INSTALL, NEWS and README files before proceeding, they contain last minutes changes and other useful information.

   Build: [5]
   ```
   $ cd vdkbuilder−2.0.0
   $ ./configure  −−prefix=<your prefix> [others options]
   $ make
   ```

   Install: (you need root permission)
   ```
   $ su
   # <enter password>
   # make install
   # ldconfig −v | more
   ```
   This assures that the plugin libraries will be correctly inserted in loader search path.

3 To see all configuration options try $ ./configure −−help

4 The default installation prefix is `/usr/local`, RedHat and RedHat like system users are encouraged to use `/usr` instead. Those users that are upgrading to VDK 2.0.x from VDK 1.2.x and want to maintain both versions must use a different prefix, VDK 1.2.x and VDK 2.x are incompatible and should reside in different paths (I suggest `/usr` for vdk 1.2.x and `/usr/local` for vdk 2.0). If you have installed libsigc++ library add `−−enable−sigc=yes` and `−−enable−testsigc=yes` to your configure options. This will enable the vdk extended sygnal system. Also `−−enable−static=no` is reccommended, this disables building of static libraries, linking vdk applications with static libraries is not well tested and you loose some LGPL license advantages..If you link your application against static libraries you must distribute it under the same license, linking with shared libraries instead permits you to distribute it under a less restrictive license.

5 The default installation prefix is `/usr/local`, RedHat and RedHat like system users are encouraged to use `/usr` instead. Those users that are upgrading to VDKBuilder 2.0.x from VDKBuilder 1.2.x and want maintain both versions must use a different prefix, VDKBuilder 1.2..x and VDKBuilder 2.x are incompatible and should exist in different paths (I suggest `/usr` for VDKBuilder 1.2.x and `/usr/local` for VDKBuilder 2.0)

Test:
At this point you should be able to run:

```
$ vdkb2
```

At first run VDKBuilder will attemp to create a new directory: /home/your_account/.vdkb2/res where it will put some files:

− *cpphints*

Is a file that contains source editor hints, you change this file using "Tools −>Hints editor" menu

− *lruprojects*

This file contains last recently used projects that can be reopened using "File −>Reopen" menu

− *tokens.db*

This file contains keywords and syntax pattern information used by source editor syntax highlighting.

− *vdkbrc*

This file contains some instructions regarding default fonts to be used and backgrounds for tooltips. File format is more or less self explanatory, edit this file if you want to change them but keep in mind that many gtk+ themes will override your settings.

− *last.session*

This file stores informations on the last session and is used to restore it on the next run. This feauture is deactivated by default, use "Tools−>Set builder environment" menu to activate it.

− *plugins.db*

Is the plugins database, it contains informations about available plugins and where they can be loaded from. Use "Components" menu to edit this file.

− *vdkbide.defaults*

Contains all VDKBuilder defaults, do not edit directly this file, use "Tools−>Set builder environment" instead.

You will be given a dialog upon successful creation of the /home/your_account/.vdkb2/res files.

I suggest you then use VDKBuilder to open the project vdkbuilder−2.0.0/example/hello.prj, this will demonstrate a very simple program made with VDKBuilder. Since project paths are hard wired you will be prompted to update, answer yes to prompts and reopen the project again. To open this project use "File−>Open−>Project" menu which opens the "Open project" dialog, and browse through your directories to find vdkbuilder−2.0.0/example/hello/hello.prj project file, select it and click on the "Open" button (or double click on selected file).

GETTING HELP AND TECHNICAL SUPPORT

If you plan to use VDK/VDKBuilder to develop or study I strongly suggest you subscribe to the vdkbuilder−list, as you can then get the best response to requests for help.
Visit VDKBuilder site at http://vdkbuilder.sourceforge.net and follow mailing list link.

THE VDK TEAM

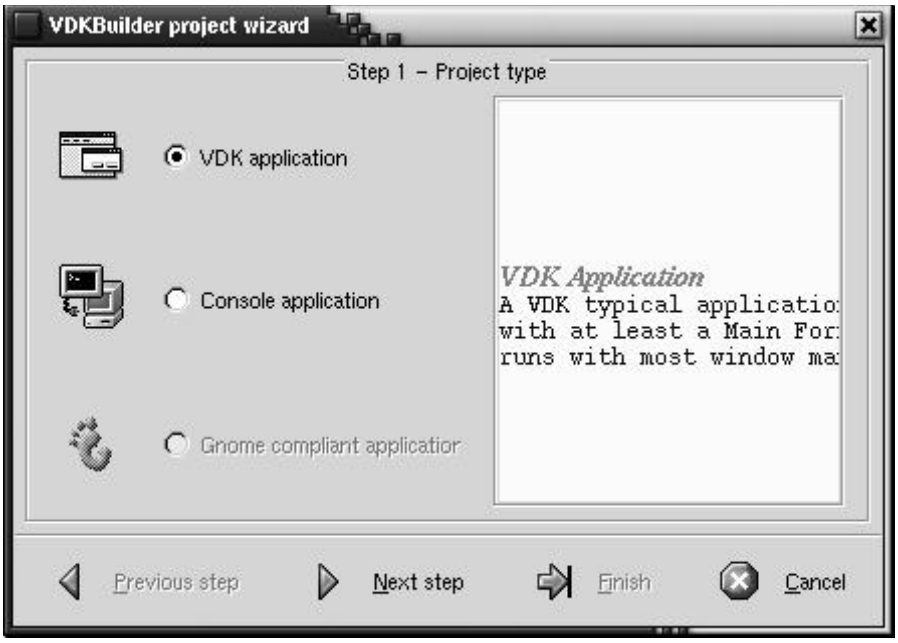
VDK Team is the pool of programmers that created and now maintain both VDK and VDKBuilder:

- **Mario Motta** <mmotta@guest.net> from Italy
  The original author, he wrote initial versions of VDK and the VDKBuilder
- **Ionutz Borcoman** from Romania
  He is the first co−author of VDK, taking care of interface design and a lot of tests. His complains and requests contributed to make VDK better and better. Actually isn't an active member, we hope soon he will join us again.
- **Mile Lazarovski** <daxml@freemail.com.mk> from Macedonia
  Developer and mantainer, he wrotes a nice class browser for VDKBuilder.
- **Tim Lorenz** <tim@lorenz.nu> from Germany
  Developer and mantainer, he wrote VDK signal system extension for libsigc++ and the new VDK properties system.
- **Pierre Louis Malatray** <pierrelouis.malatray@free.fr> from France
  Developer and mantainer, he wrote useful extensions to VDKString class.
- **George Boutwell** <gboutwel@yahoo.com> from USA
  Developer and documentation writer
- Many other contributors devoted their work to VDK/VDKBuilder, we thank them all.

GETTING STARTED

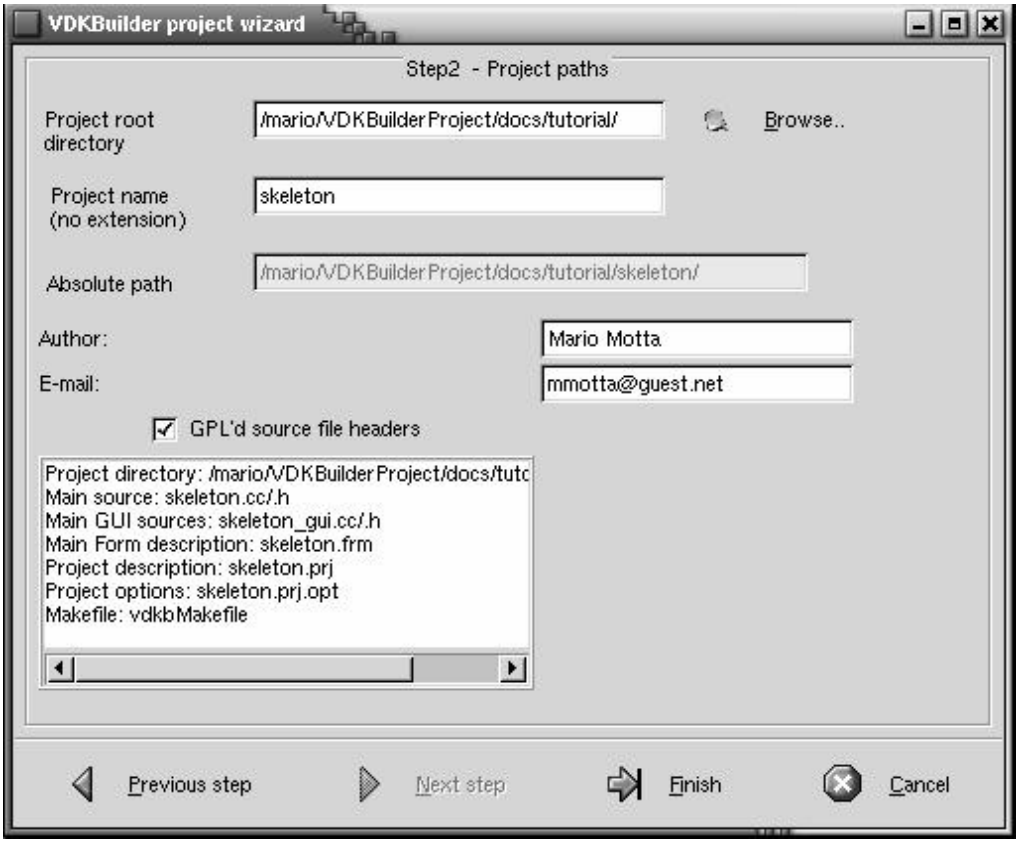
The best way to learn a tool is to begin to use it, so let's make the simplest program we can with VDK. It will be just an an application with an empty form, nothing more, but is enough to demonstrate many of VDK features.
The simplest, most bare bones, program that you create consists of an application and it's associated form (or window) called "application main form", closing this form leads to program termination as well. VDK accomplishes this with two abstract classes: VDKApplication and VDKForm; these are both abstract classes, you cannot construct an instance of them, instead you have to subclass them for your purposes. VDKApplication has a pure virtual method called Setup() that you must override in your derived class which at some point must construct the main form, also VDKForm has a pure virtual Setup() method that you must override as well. Both methods will be automatically called at program start up, this is the place where you can make all necessary initializations and setup information, but that will discussed more in depth later.

Now lets use VDKBuilder (thereafter called simply "builder"), it will do most of the work for you.
- Run vdkb2
- Click on "File−>New..−>Project" menu (or click on "New project" icon on speed bar)
- A VDKBuilder new project wizard will appear:



as you can see you have three different choices:
- **VDK application**
  A plain VDK application with at least a main form that does not contain anything desktop specific (specifcally for KDE or Gnome) and should display on almost all window managers.
- **Console application**
  An application made to be executed within an X terminal with the shell. This application does not require VDK or a GUI Interface at all.
- **Gnome compliant application** (has not been enabled at the time of this writing)
  A VDK application that is Gnome aware and uses one or more Gnome widgets, it requires Gnome libraries to be installed and typically requires the Gnome desktop, as well.

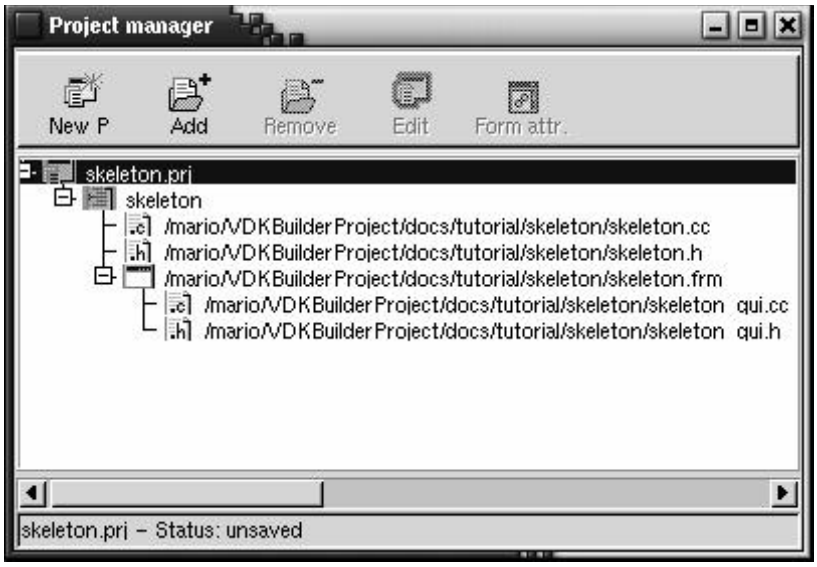
Since we want to make a plain VDK application, click on the “Next step” button. The next page of the project wizard will require you to choose a project root path and a project name to construct a new directory named: “project−root−path/project name”. Be sure not to put any extensions on the project name, builder will add all needed extensions for you. By default the project wizard assumes "project−root−path" is your “home” directory, you can change it using the “Browse” dotted button to the right of the project root field. You can also insert author name, author email and check for a GPL license, all this informations will be inserted into source headers. (You won’t see these informations into source files on this tutorial however)



- browse to your project root directory
- enter the project name “skeleton” in project name field
  as you can see the project files tree is displayed in the lower panel:
  - *Project directory: /mario/VDKBuilderProject/docs/tutorial/skeleton*
    is the directory that will contains all project files
  - *Main source: skeleton.cc/.h*
    These files are initially created by builder, any further changes you make to this file builder will leave untouched.
  - *Main gui sources: skeleton_gui.cc/.h*
    This file contains the code written by builder during interface design, any changes you make to this file will be overwritten by builder each time you make a project build.
  - *Main form description: skeleton.frm*
    This file is written by builder and describes how to make the main form interface, how many widgets it contains, which signals are connected with what code and so on. Even if file format is plain text, do not edit it or edit at your own risk, as builder maintains this file.
  - *Project description: skeleton.prj*
    This file describes all the projects files and their type, do not touch it, builder maintains this file.
  - *Project options: skeleton.prj.opt*
    It contains project only options. Do not edit it, this file is initially created by builder with default values, use “Project−>Options” menu to add or change what you need.
  - *Makefile: vdkbMakefile*
    Is the makefile that builder creates each each time you do a project build, it is useless to edit it. It can be used to do a make from the command−line.[6]

6 e.g. $ make −f vdkbMakefile

– Now you can click on “Finish” button.
  After you have been prompted to confirm the construction of the new directory (answer “yes”), you can see that both Project manager and source editor are now showing the project files tree and the main source: skeleton.cc.
– Project files tree is not complete, however, so use “File−>Save All” menu to save the whole project.
  Now Project Manager will show the complete files tree (click “+” icon to expand nodes if necessary) as you can see on the next picture.



Above you see all of the “skeleton” project’s files. Selecting a node and clicking on the “Edit” icon on upper project manager bar will open the file for editing depending on what type of file it is. If it’s a source file, it will be opened in the source editor, or if the node is a form it will be opened in the GUI designer. Double−clicking the node, will, also, open it ofr editing. Try to select skeleton.frm node and click on icon, GUI designer will appear:

before starting a discussion on the above source files, let’s instruct builder to save the current work session so it can remember where you were next time you run builder:
– select “Tools−>Set builder environment” menu
– check “Saves last work session” check box, click on “Close and save” button.

SKELETON PROJECT SOURCE FILES

Now we will see what builder has created for you, lets go to the source editor, skeleton.cc should be already there, now double click on skeleton.h node in the Project Manager, the file should appear in source editor, let's look step by step at the contents of both skeleton.h and skeleton.cc files.

Skeleton.h

```
/*
skeleton Plain VDK Application
Main unit header file: skeleton.h
*/
```

/* include sentinel, assures that this file won't be included twice
*/

```
#ifndef _skeleton_main_form_h_
#define _skeleton_main_form_h_
```

/* this conditional compilation is related to Automake/Autoconf support,
and will be discussed later
*/

```
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif
```

/* this file contains all VDK declaration, enums, types etc..
*/

```
#include <vdk/vdk.h>
```

/*Here builder has derived the application main form class, naming it with the project name (capitalized) as prefix and "Form" as suffix.
*/

```
class SkeletonForm: public VDKForm
{
private:
```

/*this method will be written/rewritten by builder each time you edit/change the form with GUI designer, GUISetup() code is on skeleton_gui.cc file (will be discussed later) and called by SkeletonForm::Setup().
*/

```
void GUISetup(void);

public:
```

/* since this is the application main form it is created passing a pointer to associated application. In most cases a form is a child of another form, in such case it is created by passing a pointer to the "owner" form, application main form can be considered as a "special" child with its'owner set to NULL. In VDK there are two differents concepts: ownership and parenthood, ownership is related to memory management, parenthood instead refers to signals/events flow, in forms ownership and parenthood clash, in other widgets the owner of a widget is always a form while its parent the container that contains it. This topic will be discussed later together with signals/events flow.
*/

```
    SkeletonForm(VDKApplication* app, char* title);
    ~SkeletonForm();
```

/* here builder has declared Setup(), making SkeletonForm a real class (recall that VDKForm is an abstract class), this method will be invoked just before form creation during application start−up.
*/

```
void Setup(void);
```

/* here skeleton_gui.h file will be included (this file is written and overwritten by builder during GUI design)
*/

```
/*
 gui setup include
 do not patch below here
*/
#include <skeleton_gui.h>
};
```

/* here builder declares a derived application class, using project name (capitalized) as prefix and "App" as suffix.

```
      */
  // Skeleton APPLICATION  CLASS
  class SkeletonApp: public VDKApplication
  {
  public:
      /* Application constructor takes same arguments as main(), note the int* argc instead
      int argc as in main()
      */
  SkeletonApp(int* argc, char** argv);
  ~SkeletonApp();
      /* here builder has declared Setup(), making SkeletonApp a real class (recall that
      VDKApplication is an abstract class), this method will be invoked during application start−
      up.
      */
  void Setup(void);
  };

  #endif

  // do not remove this mark: #!#
  // end of file:skeleton.h
```

That’s all for skeleton.h, this file, initially created by builder will be left untouched after that and is for you to modify as needed.

Now let's switch to skeleton.cc using source editor page tabs.

skeleton.cc

```
/*
skeleton Plain VDK Application
Main unit implementation file:skeleton.cc
*/
#include <skeleton.h>
/*
main program
*/
int main (int argc, char *argv[])
{
      /* constructs skeleton application
      */
SkeletonApp app(&argc, argv);
      /* and runs it (note the & in argc for the int * in SkeletonApp.)
      */
app.Run();
      /* that's all folks!
      */
return 0;
}

// Skeleton MAIN FORM  CLASS
      /* main form constructor. By default DisplayType is set to GTK_WINDOW_TOPLEVEL,
      however it can be customized to GTK_WINDOW_POPUP that provides a form with no
      decorations. An application's main form should be a GTK_WINDOW_TOPLEVEL however.
      */
SkeletonForm::SkeletonForm(VDKApplication* app, char* title):
       VDKForm(app,title,v_box,DisplayType)
{
}
      /* main form destructor
      */
SkeletonForm::~SkeletonForm()
{
}
      /* main form setup, this method is called during application setup, here it calls
      GUISetup() written by builder, this should be the first call in Setup(), so leave it where is
      and add your code after.
      */
void
SkeletonForm::Setup(void)
{
       GUISetup();
// put your code below here

}

// Skeleton APPLICATION  CLASS
      /* application constructor
      */
SkeletonApp::SkeletonApp(int* argc, char** argv):
       VDKApplication(argc,argv)
{
}
```

```
/* application destructor
*/
SkeletonApp::~SkeletonApp()
{ }
/* application setup, called at program start−up, creates, sets up and shows application main form.
InitialPosition is set by default to GTK_WIN_POS_NONE, can be customized to be set under mouse or centered on the screen.
*/
void
SkeletonApp::Setup(void)
{
      MainForm = new SkeletonForm(this,NULL);
      MainForm−>Setup();
      MainForm−>Show(SkeletonForm::InitialPosition);
}

// do not remove this mark: #!#
// end of file:skeleton.cc
```

again skeleton.cc will be initially created by builder and left untouched for your coding.

SKELETON PROJECT GUI SOURCE FILES

There are two more source files, these are under builder responsibility: skeleton_gui.h and skeleton_gui.cc, these files contain source code for constructing the application interface, they are rewritten by builder each time you do a build and you have changed something in the GUI designer. Those files are written getting informations from the file: skeleton.frm that describes the form and it's contents.

So the path is:

- you change the gui design
- builder updates skeleton.frm
- when you do a project build builder reads skeleton.frm to rewrite skeleton_gui.h and skeleton_gui.cc.

You may be asking why you can browse the contents of these files even if it's useless to edit them? The answer is: nothing should be hidden from you, even if it is redundant and useless, personally I dislike those tools that hide thing in the background without notice to the user. If something goes wrong you cannot see where and why, with builder you can see all that's being done for you in an easy and plain way. It may be inelegant but it works. Further more seeing these files is useful for this this tutorial, and you can see them growing as you add useful widgets and signal handling to the interface. So let's take a step−by−step look at these files.

skeleton_gui.h

/* recall that this file is included into skeleton.h, it contains all declarations for interface: widgets, signals and signal response methods. Since we have constructed an empty form you do not see any widget declaration here.
*/

```
/*
skeleton gui header
*/
public:
/*
declaring signal and events
dynamics tables
*/
```

/* VDK has three different ways to handle signals and events emitted by widgets:

- using static signal/event tables, the easiest way, handled directly by builder, user only needs to fill signal response method. Since we do not have widget, no signals/events to handle, they are not shown in code here.
- using dynamic tables, handled by user who needs to then connect the widget with the signal and response method, a bit more complicated but more flexible. Here you see the declaration of signal/event dynamic table, user connects signal/events using SignalConnect()/EventConnect() methods.

*/

```
DECLARE_SIGNAL_LIST(SkeletonForm);
DECLARE_EVENT_LIST(SkeletonForm);
```

/*

- There is a third way that VDK can use, the extended signal system and needs libsigc++ library (recall that you must have built vdk with −−enable−sigc=yes option), user then needs to connect the widget with signal and response method, more complicated but, again, much more powerfull and allows the user to connect with all signals/events.
- Furthermore it is possible, as demonstrated later on, to use native gtk+ signal system using lower level gtk+ calls. In this case we loose a bit of flexibility since we are forced to connect with global or class static callbacks, connecting directly with class member functions is denied.

*/

```
// declares two static used to initialize
// form display type and initial position
static GtkWindowType SkeletonForm::DisplayType;
static GtkWindowPosition SkeletonForm::InitialPosition;
// do not remove this mark: #!#
// end of file:skeleton_gui.h
```

you can find a complete discussion about VDK signal system in tutorial appendices.

skeleton_gui.cc

```
#include <skeleton.h>
//define static display type and initial form position
GtkWindowType SkeletonForm::DisplayType = GTK_WINDOW_TOPLEVEL;
GtkWindowPosition SkeletonForm::InitialPosition = GTK_WIN_POS_NONE;
```

/* since in skeleton_gui.h dynamic table were declared, this is their definition
*/

```
/*
defining signal and events
dynamics tables
*/
```

/* Note that the two macros that define dynamic tables require the name of the class and his ancestor, this is a key point, signals/events will flow not only from widget through their containers up to outermost widget but also along widget class hierarchy, this feature makes VDK signal system “broadcast” signals/events in a way that allows them to be answered at any level of the widget hierarchy. You can find a more detailed discussion on signal/event flow strategy and how you can make the best of it in tutorial appendices.
*/

```
DEFINE_SIGNAL_LIST(SkeletonForm,VDKForm);
DEFINE_EVENT_LIST(SkeletonForm,VDKForm);
```

/* this method is called by SkeletonForm::Setup(), here it prepares an empty form setting initial size (width and height respectively) and title. Obviously you can change them using both GUI designer and Widget Inspector. Recall that this is the minimum size, form cannot be shrunk below.

```
/*
main form setup
*/
void
SkeletonForm::GUISetup(void)
{
      SetSize(400,300);
      Title = "skeleton Main Form";
}

// do not remove this mark: #!#
// end of file:skeleton_gui.cc
```

Now take a look at skeleton.frm, the file that describes how a form is made and what it contains. Recall that even if viewed in a textual form editing this file is not only useless but can be dangerous since the parser is not well protected against syntax errors, so do not edit it unless you are an expert and at your own risk :−)

skeleton.frm

```
[skeleton]
{
      class:form;
      skeleton.this:skeleton;
      skeleton.NormalBackground:nihil;
      skeleton.Foreground:nihil;
      skeleton.Font:"nihil";
      skeleton.Cursor:nihil;
      skeleton.BackgroundPixmap:nihil;
      skeleton.FocusWidget:nihil;
      skeleton.DisplayType:0;
      skeleton.InitialPosition:0;
      skeleton.Usize: 400, 300;
      skeleton.Title:"skeleton Main Form";
      skeleton.OnFormActivate:nihil;
      skeleton.OnChildClosing:nihil;
      skeleton.OnConfigure:nihil;
      skeleton.OnExpose:nihil;
      skeleton.OnIconize:nihil;
      skeleton.OnMove:nihil;
      skeleton.OnRealize:nihil;
      skeleton.OnResize:nihil;
      skeleton.OnRestore:nihil;
      skeleton.OnShow:nihil;
      skeleton.CanClose:nihil;
```

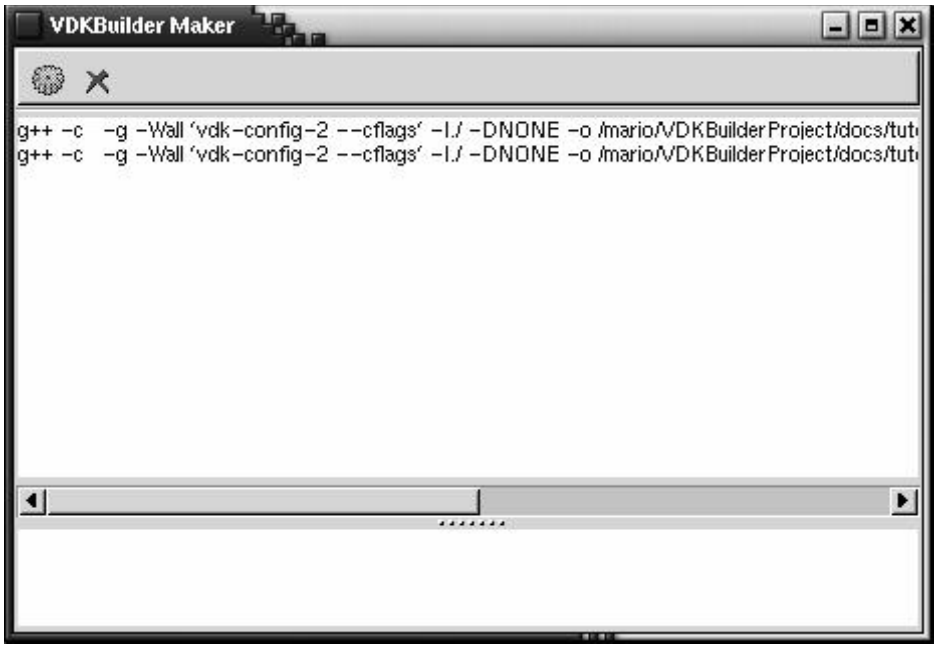
```
}
```

the file contains a form's description in terms of a list of properties and their values:
<property name>:<value>;
the list is enclosed into braces and has a label that uniquely identifies the form.
Since the form is actually empty no other information is written, skeleton.frm will grow as user adds widgets and signal/event handling to the GUI interface. As you can see most of properties are set to special value "*nihil*", which means the user has not set any particular value. Depending on the property nothing will be done or default value is used. For instance since the form's normal background property is set to nihil, the form will use default gtk+ theme background color. Same thing for form events, since none are handled by user, all related properties are set to nihil. However, these notes are here only to server as documentation, should there be the rare case that you have to directly patch this file and only in case of builder failure (unlikely but not impossible :−))

SKELETON APPLICATION FIRST RUN

Now we are ready to make skeleton application, and then run it for the first time.

- Use "Project->Make" menu (or use icon on source editor upper bar)
  VDKBuilder Maker will appear and starts builder's maker, which is an interface with some external programs such as GNU compiler. It is divided into two panes, while the upper one shows what maker is doing (in this case you see the compiler outputs referring to compilation process), the lower pane will eventually show (in red foreground) compilation errors if any. Before starting maker, builder has peformed a few tasks, however:
  - builder has checked if one of the project files is changed, in such case will prompt you for saving it[7]
  - if you have modified the interface builder reads skeleton.frm and rewrites skeleton_gui.h/.cc files
  - builder writes vdkbMakefile used by GNU "make" external program
  - builder calls GNU "make" program and show its outputs on builder maker panes.
  - builder maker will disappear as the compilation process finishes and the overall result is prompted to you and reported on source editor messages pane.



  Hopefully a message box should have prompted "VDKBuilder Maker terminated succesfully", this means that skeleton application has been built and ready to be runned.

- Use "run" icon located on main window lower speedbar or on source editor upper bar.

  Again, builder maker will appear together with skeleton application, I agree it is an useless empty form, but soon we will make it nicer.
- Now close skeleton application using window manager "close" button on one of the window upper corners (depends on windows manager you are using)
- Close also builder maker clicking on "kill" icon on upper toolbar.

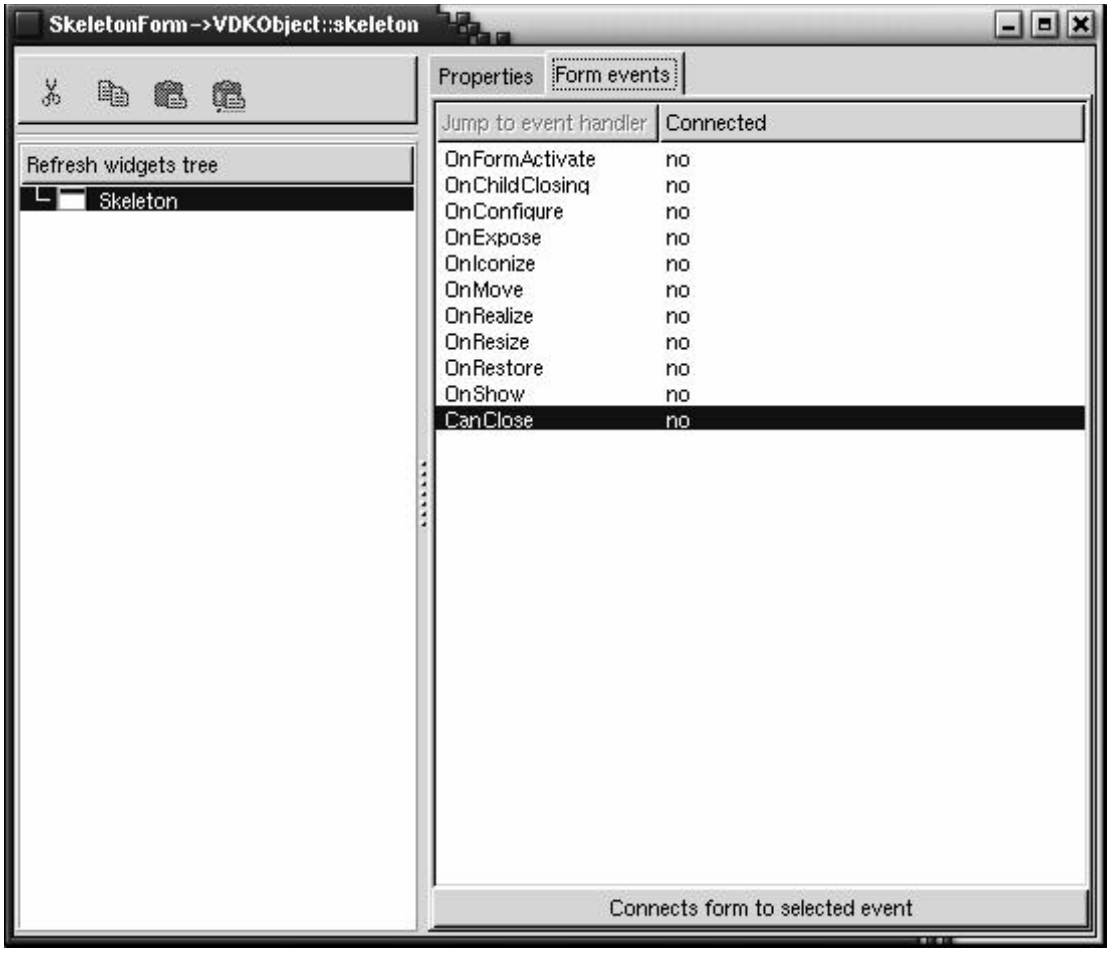
Often it is necessary to do some clean-up before terminating an application or closing a form, as said before closing the main form lead to application termination. How we can know when a form is being closed in order to make necessary clean up and/or eventually prompt the user?
Let's take a look at the next section which explains how.

7 This is the default behaviour, if you want save changed files without be prompted, use "Tools->Set Builder Environment" menu and check "Auto save modified files" checkbox, then click on "Close and save" button.

HOW TO NICELY CLOSE SKELETON APPLICATION

VDK handles window manager “window delete” event calling VDKForm::CanClose virtual method that returns a boolean, a true value means “yes you can proceed to close the form”, a false value will interrupt form closing procedure. Obviously at VDKForm class level this method return always true, but if you override this method in SkeletonForm class you can control wheter the main form (and then application) should be closed or not. Again builder can help you in this task

- Select “skeleton.frm” node on project editor and click on “edit” icon
  Widget inspector togheter with the gui designer will appear
- Select Skeleton on widgets tree and click on “Form events” page tab
- Select CanClose on form event list
- Click on “Connects form to selected event” button



you will see that source editor move to the end of skeleton.cc, where builder has written for you a SkeletonForm::CanClose method.

```
//asks user before closing
bool
SkeletonForm::CanClose(void)
{
return true;
}
```

Leaving CanClose() method as is, is what makes main form and application close without further notice. So let’s change CanClose() a bit in order to prompt the user for a closing confirmation.

```
bool
SkeletonForm::CanClose(void)
{
  int answer = Application ()->MessageBox (
                            "Skeleton application",
                            "Really close application ?",
                            MB_YESNO | MB_ICONQUESTION,
                            NULL,
                            NULL,
                            5000);
  return answer == IDYES;
}
```

now CanClose() return a true value only if user answers “yes” to the Message dialog.

MessageBox() is declared as:

```
int VDKApplication::MessageBox (  char * caption,  char * text,
                                  int mode = MB_OK,
                                  char * oktext = (char*) NULL,
                                  char * canceltext = (char*) NULL,
                                  unsigned int wait = 0 )
```

This method belongs to VDKApplication class and provides a modal dialog window for messages to the user. Application object can be accessed from form using VDKForm::Application() method.
Parameters:

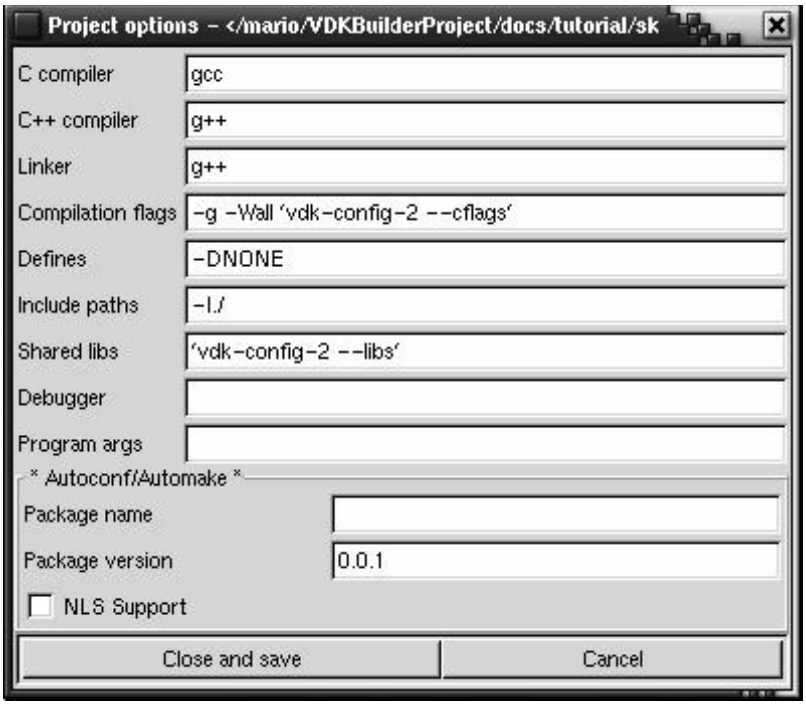
- mode
  MB_OK provide only one button with a "Ok" default caption
  MB_YESNO provides two button with "Yes" and "No" default captions.
  MB_OKCANCEL provides two button with "Ok" and "Cancel" default captions.
  Mode can be ored with:
  - MB_ICONSTOP provides a warning icon
  - MB_ICONINFORMATION provides a "information icon"
  - MB_ICONQUESTION provides a question mark icon
  - MB_ICONERROR provides an error icon
    MessageBox returns an integer that depends on modes and user response:
    - IDOK, user pressed OK button in MB_OK or MB_OKCANCEL mode
    - IDYES, user pressed YES button in MB_YESNO mode
    - IDNO, user pressed NO button in MB_YESNO mode
    - IDCANCEL, user pressed NO button in MB_OKCANCEL mode
- oktex
- canceltext
  Args no longer used, mantained for compat with vdk series 1.2.x
- wait
  if set to anything other than 0, makes MessageBox automatically closed after <wait> msecs with IDCANCEL or IDNO result.
  MessageBox accepts CR as "yes/ok" and ESC as "no/cancel" default answers.

Now let’s build again the project, you will be prompted to save both skeleton.cc and skeleton.frm, that’s normal since you changed the interface adding a form event handling, also skeleton.cc was changed by adding CanClose() overridden method. Again builder maker will appear starting compilation process that hopefully ends with success. Running the application and closing it a message box will appear

close skeleton application and builder maker as well.
As an aditional exercise you could take a look at how skeleton_gui.h and skeleton.frm have been changed by builder.

VDKBUILDER PROJECT OPTIONS

Before proceeding with our skeleton project let's take a look at how we can change project options and how they change the building process.
Use "Project−>Options" menu for this, a dialog form will appear:



The first three fields, that refer to compiler and linker to be used , are filled by default and in a Linux box should be left as they are unless you do want to use another compiler different from GNU gcc (i don't see any reason for it).

Compilation flags

This fields, filled by default with −g −Wall, permits a compilation with debug information and a satisfactory level of warnings, furthermore compiler will receives switches from the output of vdk−config−2 script that tells to the compiler where to search all includes files that a vdk application needs.
Just for exercise try to open an xterm and enter:

```
$ vdk-config-2 --cflags
```

you should have something like this output:

```
-I/usr/local/include/vdk2                    -I/usr/local/include/gtk-2.0
-I/usr/local/lib/gtk-2.0/include            -I/usr/local/include/glib-2.0
-I/usr/local/lib/glib-2.0/include          -I/usr/local/include/pango-1.0
-I/usr/local/include/atk-1.0                    -I/usr/lib/sigc++/include
-DUSE_SIGCPLUSPLUS
```

your output may be different depending on where (prefix) you installed vdk and if it has been compiled it with libsigc++ extension.
You can use this field to change/add compiler switches as you need.

Defines

Use this field to change/add your defines if needed

Include paths

Use this field to change/add include paths if needed

Shared libs

This field is filled by default with "vdk−config−2 −−libs" script output, and tells to the linker which shared libraries should be linked with your program. Again as an exercise open an xterm an try:

```
$ vdk-config-2 --libs
```

you should have something like this output:

```
-L/usr/local/lib  -L/usr/X11R6/lib  -lvdk  -lgtk-x11-1.3  -lgdk-x11-1.3
-lXext -lgdk_pixbuf-1.3 -lm -lpangox -lX11 -lpango -latk -lgobject-1.3
-lgmodule-1.3 -ldl -lglib-1.3 -lsigc -lpthread
```

your output should look different however since at the time of this writing gtk+2.0 is not officially

released yet and we are using gtk+1.3 developing version.

Debugger

Into this field (left blank by default) you have to enter the name of the debugger that you want use to debug your program.[8]

Actually not all known graphical debugger have been tested, builder has been tested with:

- **ddd**
  The Data Display Debugger, based on gdb, is in my opinion the best ever seen, you can find it at: http://www.gnu.org/software/ddd
- **kdbg**
  The KDE Debugger, you can find it at: http://members.nextra.at/johsixt/kdbg.html
- **gvd**
  The GNU Visual Debugger, a very nice tool, you can find it at: http://libre.act−europe.fr

Program args

If your programs needs some arguments to run here is where you would put them.

Autoconf/Automake

This section set options for GNU autoconf/automake support for your projects, and will be discussed later in tutorial appendices.

8 Whichever debugger you use, it should be found either on /usr/bin or /usr/local/bin

MAKING SKELETON MAIN FORM NICER

Making a nicer background

Bored with that flat−grey form background ? Now I will show you how to have a nicer fuzzy−grey background.

First of all we need a nice little pixmap, get it from the <where−builder−is>/vdkbuilder−2.0.0/example/hello directory:

```
$ cd <where−skeleton−is>/skeleton
$ cp <where−builder−is>/vdkbuilder−2.0.0/example/hello/fuzzy.xpm .
```

Now let's change SkeletonForm::Setup() method a little bit:

```
/*
main form setup
*/
void
SkeletonForm::Setup(void)
{
  GUISetup();
  // put your code below here
  BackgroundPixmap = new VDKRawPixmap(this,"./fuzzy.xpm");
}
```

We use Form::BackgroundPixmap property set to a newly constructed VDKRawPixmap.
VDKRawPixmap is a raw image that does not belong to any widget,
Constructor is declared as:

```
VDKRawPixmap::VDKRawPixmap( VDKObject *owner, char *pixfile );
```

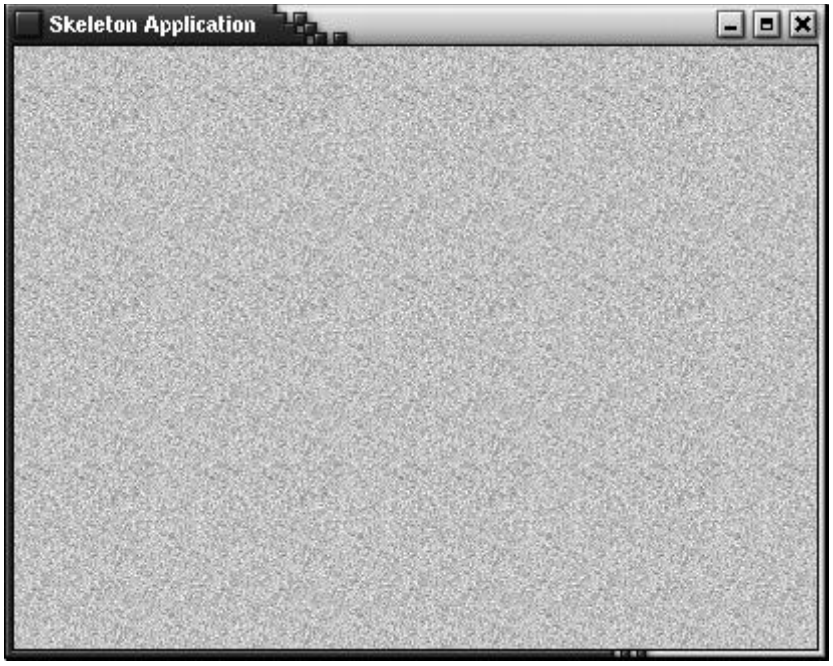
and takes two arguments:

- owner: the object that will became raw pixmap owner
- pixfile: the file name that contains pixmap data

Setting form title

- Select Widget Inspector (hereafter called simply WI) skeleton node, edit "form title" field to read: "Skeleton Application" and click on "Set form title" button. Form title will change on GUI designer.

Now try to build again the project, here the result



The pixmap will be repeated to fill the whole form surface creating a nice texture effect. Recall that you cannot see the fuzzy background at design time, obviously builder know nothing about what you wrote in SkeletonForm::Setup(). However (and left as reader exercise) you can achieve same effect using GUI designer and Widget inspector.

VDK pseudo garbage collection

Now you could ask, why did we construct a VDKRawPixmap with the “new” operator and there is no corresponding “delete” operator is shown somewhere? Good news, with VDK you can forget the time consuming and error prone procedure of freeing objects constructed onto the heap. VDK implements a sort of limited garbage collection, the “owner” is responsible for deleting all it’s owned objects when it is destroyed. So you construct new objects as needed and forget them, they will be deleted when the owner dies. Garbage collection isn’t set by default, however, even if not set, garbage collection will be triggered just before program termination. You can set a timed garbage collection using VDKApplication::SetGarbageCollection() method, that takes a “tick” argument, timed garbage collection will be invoked each “tick” milliseconds. However this useful feauture gives some limitations:

- all VDKObjects must constructed on the heap with “new” operator, no static or automatic VDKObjects are allowed, any copy−initializing and/or assignements between VDKObjects will be flagged as compilation error.[9]
- you will never invoke “delete” operator on a VDKObject or the result will be a program crash, to explicitely delete a VDKObject you have to use VDKObject::Destroy() method that is safe.

9 Notice that VDKObject class is a subclass of VDKNotCopyAble where copy operator is inaccesible.

ADDING WIDGETS TO SKELETON FORM

Now we will begin to add useful widgets to our skeleton form, this is a typical interface design activity that requires a little planning and a knowledge of how VDK arranges widgets on a form. When speaking of "adding widgets" we must think of "containers". Containers are widgets that contain other widgets, however they are special since:

- they are transparent or more precisely they use the parent background
- can be nested at any level, a container can contain another container and so on
- they do not receive or emit signals, just let signals flow from contained object to their parents.
- No widget can be placed on the for unless it is planed in a container

So before adding a widget we must add containers to the form, this concept is so strong that builder won't allow you to add widgets to the form if you don't put at least one conatiner on the form. There are several type of containers, if you look on builder's tool palette you will see that it is the most crowded one. Basically they belong to two classes:

- containers with variable geometry
- containers with fixed geometry

The most of the containers fall into the former class, while the latter has only VDKFixed class. It may be that the reader is a Linux newbie coming from Windows and knows fixed containers only.
What is the difference between the two types?
While fixed containers mantain a fixed size and position of contained widgets, others, much more powerful, arrange both contained widgets size and position depending on the container size, if the container grows, all the contained widgets will grow to occupy all available space, if several widgets are in a container they will share the container space. A key point is the concept of "minimum size", the size needed at minimum to show the entire widget, a container cannot shrink below the size resulting from the sum of the contained widgets minimum sizes. For instance, if the font of a button changes to a bigger size, the minimum size is recalculated and the container resized. However this behaviour can be largerly customized by the user. This strategy can be somewhat confusing for a new user, but soon he/she will realize its power. For now remember that changing the size of a widget means that you are not changing the "actual" size but the minimum size for that widget.
Most common containers are boxes (VDKBox class), they can be either vertical and horizontal, the former arranges widgets in rows, the latter in columns. We will use them for our skeleton form.
A little bit of design here:

- we will put a vertical box into the form (called main_box)
- main box will have three rows where we willl put in order:
  - a vertical box (called upper_box)
  - a separator
  - an horizontal box (called lower_box)

The upper box will contain several widgets that we will be constructing and destroy as needed
The lower box will contain three buttons, which are there to let user create and show widgets on upper box.

Let's begin.

- select skeleton node on WI (GUI designer will appear as well).
- select a vertical box from "Containers" tool palette, you will see the the pointer became an cross−hair to warn user that GUI designer has switched into "drop mode", you can reset "drop mode" any time by clicking on "cancel selection" icon on speed bar.
- drop the selected container into GUI designer, "drop mode" resets and pointer reverts to arrow shape.

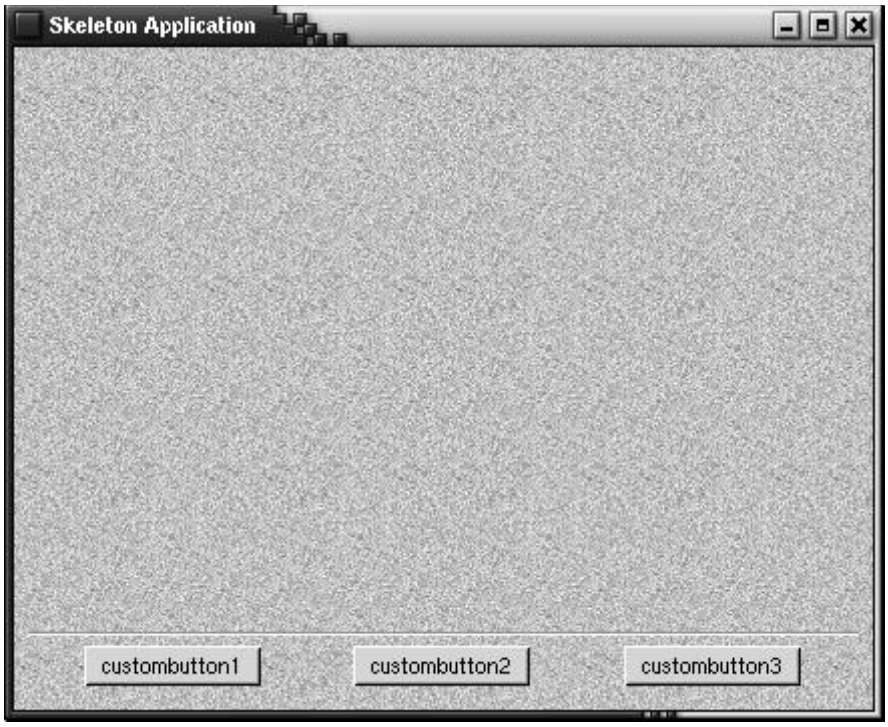
  There are several ways to drop a widget from tool palette into GUI designer:
  - you can either select the widget and click or drag the widget icon into the GUI designer.
  - alternatively you can select the container on the widget tree in WI and use the right−click pop menu option.
- You will see that a box will be dropped onto the form and the widget tree in WI shows the newly inserted container named *vbox1*. GUI designer shows the box in dark grey.

  Now let's change vbox1 name:
  - select vbox1 on WI
  - edit the "widget name" field in WI to read "main_box" and click on "Set widget name" button, widget name will change both in WI and GUI designer.

  Now we will set a border around the box, contained widgets will be lined up inside the border:
  - Set "border width" field to 5 and click on "Set border width" button, box will shrink showing a 5 pixel border.

- Now drop another vertical box into main_box , you will see another box nested under main box in the widget tree on WI, which grows to shows the newly inserted widget named “vbox2”. Now change it’s name to “upper_box”.
  Notice that upper box occupies all available space in main box, that’s normal, there is a drwaback, however, you cannot select main box on GUI designer since it’s completely covered by upper box. In addition any other widget dropped into GUI designer will be added to upper box. If you would drop a widget to main box how you can? There are several way but the easiest is probably to select main box node into WI and use right click option menu, let’s show how:
- Now we add an horizontal separator to main box:
  - select an horizontal separator from the “Misc” tool palette and select main_box node in WI
  - right click on mouse, an option menu will show you:
    “Drops a <VDKSeparator> into main box”, click on this option and you will see that it drops the new widget under main box.
  - Now select “separator0” node on WI, uncheck both “Expand” and “Fill” checkboxes and click on the “Repack” button.
    Let’s explain a bit: Depending on the box’s packing strategy all contained widgets occupy and
    share all available space, so separator has been expanded to fill all available space into main box, unless directed differently. By unchecking box “fill” and “expand” fields we have told the separator to occupy its minimum size.
- Now we add an horizontal box to main box:
  - select an horizontal box from “Containers” tool palette and select main_box node into WI
  - right click on mouse, an option menu will show you:
    - “Drops a <VDKBox into main box>, click on this option and you will see the new widget “hbox3”dropped under main box.
    - Now change its name to “lower_box”
      Notice that now upper_box and lower_box share the same space in main box.
    - Since we want to leave the most space to upper_box let’s select in GUI designer lower box and uncheck on WI both “Expand” and “Fill” checkboxes and click on the “Repack” button. You’ll see that lower box shrinks to its minimum size and gives more space to upper box (and separator0).
- Now we’ll add three buttons to lower box:
  - select a button from “Buttons” tool palette, drop the widget on lower box, a dialog box will appear letting you choose from different button types, just click on “Accept” button and the button will be dropped on lower box
  - repeat above procedure two more times, you will see that as buttons are added to box they shrink and reposition to leave space for their siblings.
  - Bored with these three buttons stuck togheter?
    - Select lower box and set border to 5
    - Select the buttons, uncheck “Fill” checkbox, click on “Repack” button and you will see the buttons equally distributed into lower box.[10]
- Now let’s build and run the program, the result is on the next page.

10 Multiple selection is not yet inplemented, you have to repeat each single action on each widget.



Refining the design

Interface design is a loop−like activity, often you need to change, add and move widgets over and over, builder has a lot of features to make this task as painless as possible. Even if you cannot directly move a widget from a container to another you can use the widget stack, where builder puts a cut widget and from where you can paste the same widget to another container. So to move widgets you cut them into stack and paste them from stack. Selecting a widget on GUI designer and right clicking lets you access a context pop menu.

Let's make an example with skeleton form, we would like divide lower box into two horizontal box, left box will contain two button (not three) and right one a combo box.

We proceed like this:

- select custombutton1, right click to have pop menu and use "Cut" option
- repeat as above on custombutton2
- select custombutton3, right click to have pop menu and use "Remove" option, removing an object really deletes it, you cannot recover it anymore.

Custombutton1 and custombutton2 are now on the widget stack.

- select lower box and drop on it an horizontal box from "Containers" tool palette
- change its name to "lower_left_box"
- select lower box on WI (recall, as said before, you cannot select lower box in GUI designer since it is completely covered by lower left box) and drop another horizontal box
- change its name to "lower_right_box"

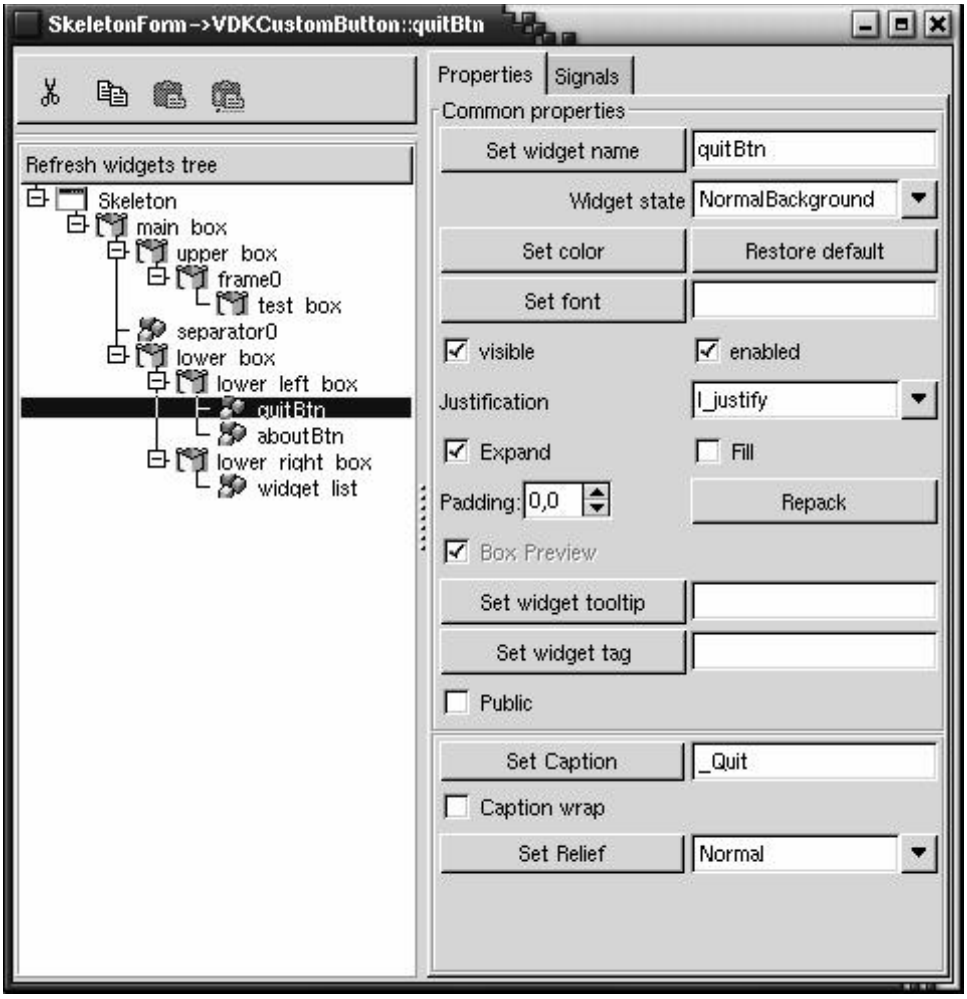
You will see lower box with two nested box inside.

- Now select lower_left_box and righ click to have widget context popup menu, you will see two menu
  now enabled: "Paste VDKCustomButton::custombutton2" and "Paste others..". Paste menu shows the widget on top of the stack, if in the stack there is more than one widget, this menu also will have "paste others" enabled.
- Choose "Paste others.."
  A dialog form will appear showing widget stack
- Select vdkcustombutton1 and click on "Paste" button. (If needed you can make a multiple selection, but this is not the case). You will see custombutton1 dropped on lower left box.
- Select custombutton1 and right click on popup menu, now "Paste others" is disabled since just one widget is left on widget stack. Click on "paste" menu, you will see custombutton2 added to lower left box.
- Now select a combo box from "Misc" tool palette and drop it on lower_right_box, change its name to "widget_list".

Upper box appear a little bit flat and bare, it will look nicer if would have a beveled aspect, we can achieve this effect by adding a frame to upper_box, so:

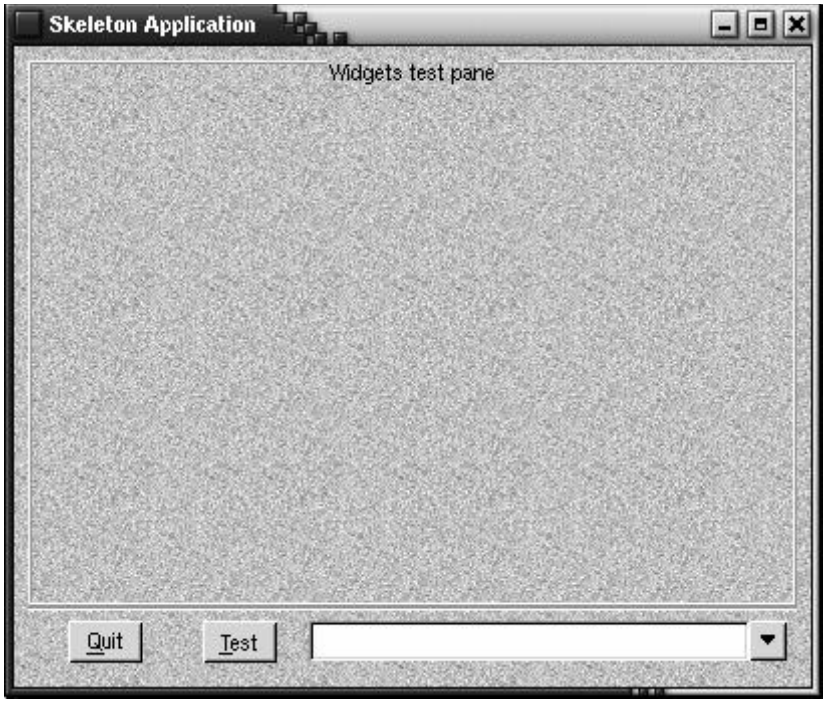
- select a frame from “Containers” tool palette and drop it into upper box
- you will see a frame added to upper box, frames come with another box inside by default, change box name to “test_box”.
- select on WI frame0 and edit “label” field entering “Widgets test pane” and hit return
- set “shadow” and “align” fields respectively with “shadow_etched_out” and “c_justify” and click on “Set shadow/align” button.
  Now we are almost satisfied with the interface, it only remain for user to change the buttons names and captions, so:
- select custombutton0 and change its name to quitBtn and his caption to “_Quit” (note the underscore, this will activate a key accelerator on the button, pressing alt q, will give the same result as having clicked the button.
- select custombutton1 and change its name to testBtn and his caption to “_Test”

Now widget tree on WI begins to be a bit crowded isn’t it? It should look something like in next picture:



Lets build and run the program now to see the effect of our changes.
The result is on the next page.

Cool, isnt'it :−)



SPEAKING ABOUT SIGNALS AND EVENTS

Let's explain a bit what signals and events are.

- Signals are synthethized events coming from widgets, for instance when you click a button it emits a signal "clicked", behind the scene however several other things happened like intercept mouse button press, mouse button release, etc.
- Events are simpler than signals, for instance a mouse move or a key press or release are events.

Signals are much more powerful, since they can express more than one event, for instance the button emits clicked signal only if you press/release mouse button over it, if you just press inside and release mouse button outside nothing happens since this action is understood as giving up on the button click.
How we can use signals?
Let's explain with an example. We want when user clicks over the button "Quit" for the skeleton program to quit. So:

- select quitBtn on GUI designer
- select Signals tab

You will see on upper panel a list of buttons to connect signals for that widget,on lower pane a list of connected signals that represents the VDK static signal table:

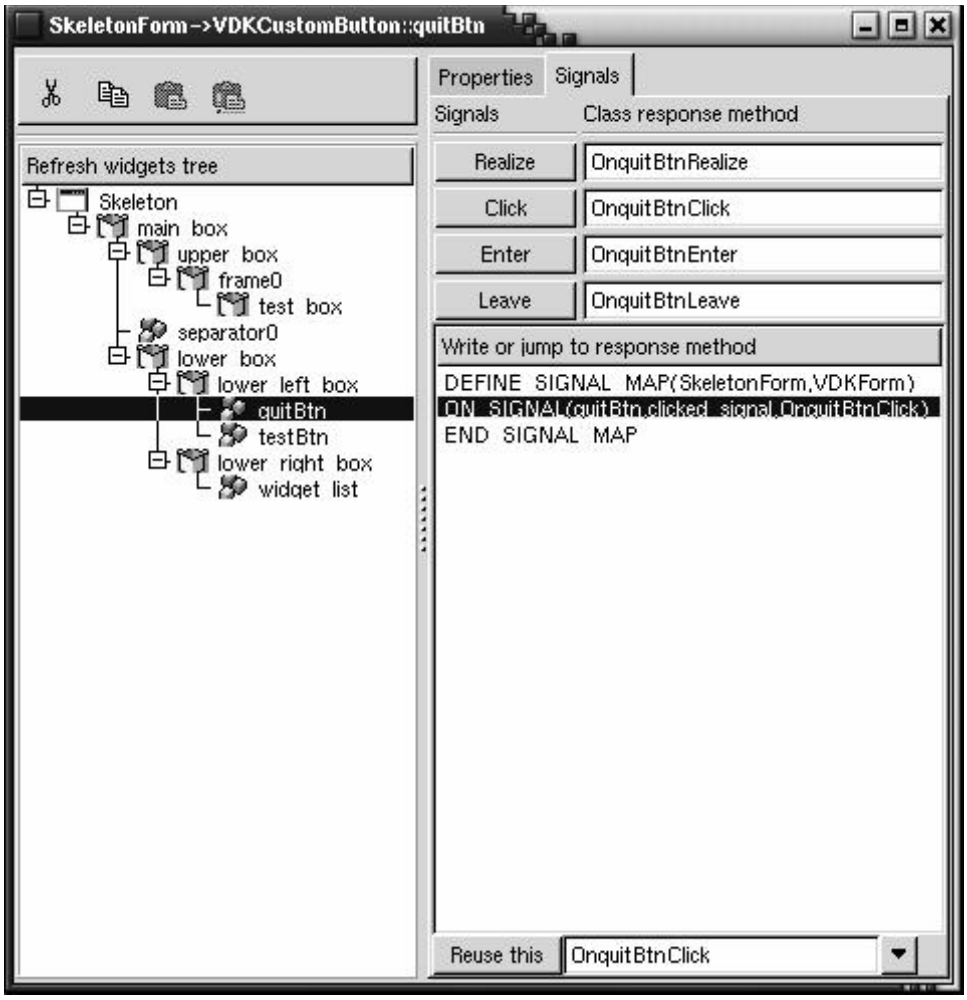
```
DEFINE:SIGNAL_MAP(SkeletonForm,VDKForm)
END_SIGNAL_MAP
```

that is empty at the moment

- click on button labeled "clicked"

Signal map will grow:

```
DEFINE:SIGNAL_MAP(SkeletonForm,VDKForm)
ON_SIGNAL(quitBtn,clicked_signal,OnquitBtnClick)
END_SIGNAL_MAP
```



Each table entry is a signal connection, it binds the object <quitBtn> with the signal<clicked_signal> and the response function < OnquitBtnClick>, thus meaning that when quitBtn is clicked, SkeletonForm::OnquitBtnClicked response method will be called. What you to do now is just fill that response method.

– now select the list entry “ON_SIGNAL(quitBtn,clicked_signal,OnquitBtnClick)” and click on list column header labeled “write or jump to response method” and you will see that source editor will appear:

```
//signal response method
bool
SkeletonForm::OnquitBtnClick(VDKObject* sender)
{
 return true;
}
```

builder has written a basic function body.

– Simply change above method to read:

```
bool
SkeletonForm::OnquitBtnClick(VDKObject* sender)
{
 Close(); // closes main form and therefore the application
 return true;
}
```

VDKForm::Close() method closes the form and since this is the application main form it will quit the program as well. However before closing SkeletonForm::CanClose() will be invoked.

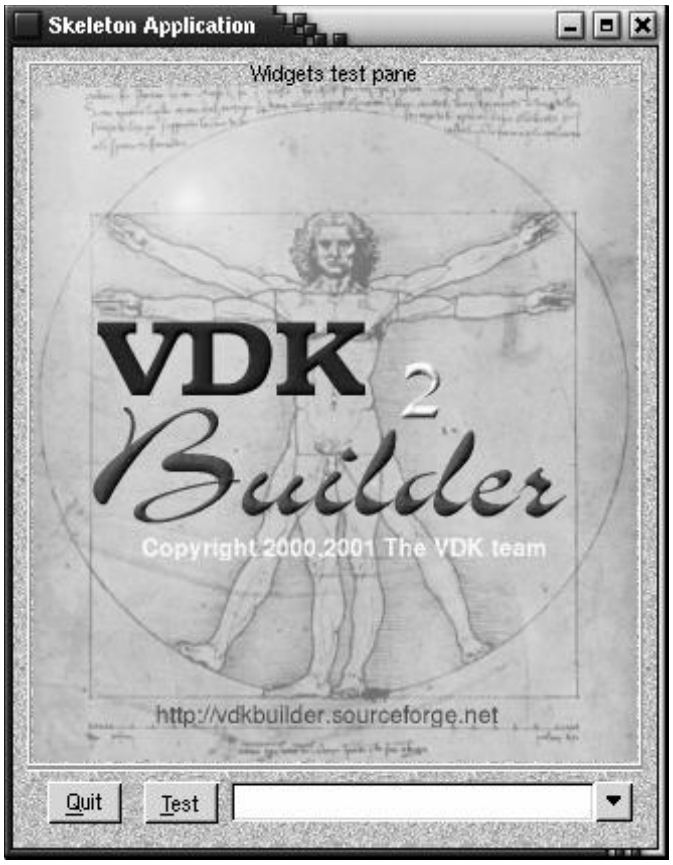
– Now build the program and test it to see the result, you should verify that hitting Alt−q or clicking on button labeled “Quit” skeleton application will terminate after having prompted the user. It is interesting to see how builder has changed GUI source files skeleton_gui.cc and skeleton_gui.h. (Left as reader exercise)

GO ON WITH SKELETON APP

Let's finish with skeleton form initial setup. Now we will put VDKBuilder 2 logo on test box.

- Select a VDKImage from "Misc" tool palette and drop it on test_box.
  An image will be placed into test_box showing a little icon as image placeholder, widget will be named image0
- Select on WI image0 and click on "Set glyph" button
  A File open dialog will appear with "File type" set by default to "*.png"
- Browse into directory until find <prefix>/share/vdkb2/res, where prefix is where you have built vdkbuilder (most likely either /usr/ or /usr/local)
- Select logo.png and click on "Open" button.
  VDKBuilder logo image will be set into image0
- Click on WI ".file −> project dir" and logo.png will be copied from source directory into skeleton project dirctory.
- Change form dimensions to fit image into test_box.

Build the program and see the result:



Admittely so far we have written very few lines of code since builder did most of the job for us, now is the time to begin our coding since what we want to do is beyond builder's own capabilities. What we plan to do is a program to test various widgets available on VDK, all tested widgets should be showed into test_box one at time, user will choose which widget has to be tested selecting it from a drop down list in the combo box named "widget_list".
So lets proceed with filling the combo box drop down list. One can say that the best place to do this is SkeletonForm::Setup() function, this is the natural place to put initial widget settings. This is not always true, however, not all widgets are ready to be manipulated when Setup() function is called.
Let's explain a bit more: a gtk+ widget needs to be fully created on screen to be fully functional, this can happen only when the window itself has a window id assigned (which is assigned when windows is created, ie Setup() function finishes).
For instance in the VDKNotebook pages are fully created only when they are activated and thus viewed for the first time, if you want to initialize a widget contained in a page that will be activated by the user you cannot do this in Setup() function. However, there is the possibility to force a widget be fully

created using a gtk+ lower level call[11] at any moment. Fortunately when a widget got fully created it emits the signal “realize” giving us the possibility to solve the problem in an elegant way. So what we are going to do, is connect with widget_list “realize” signal and make the necessary initilizations there (instead of in Setup() member function). So let’s return to builder and

- select widget_list on GUI designer
- switch on Signals page on WI
- click on “realize” button
- select correspondent entry on signal map
- click on list column header labeled “jump or write to signal response method”

Source editor will write for you the function body:

```
//signal response method
bool
SkeletonForm::Onwidget_listRealize(VDKObject* sender)
{
 return true;
}
```

now let’s change the response method a bit in order to correctly initialize widget_list.

```
//signal response method
static char* widgets_list_entries[] =
{
  "VDKCustomList",
  "VDKEntry",
  "VDKSCrolled",
  "VDKNotebook",
  NULL
};
bool
SkeletonForm::Onwidget_listRealize(VDKObject* sender)
{
// here we get a copy of combo box popdown string list,
// most likely an empty list since combo box was just realized
  StringList sl = widget_list->PopdownStrings;
// add all prompts to string list
  for (int t = 0; widgets_list_entries[t]; t++)
    sl.add (widgets_list_entries[t]);
// substitute combo box popdown string list with our modified copy
  widget_list->PopdownStrings = sl;
  return true;
 }
```

Build the program and you will see widget_list filled with above strings.
Take note that in this case initialization could have be done also in SkeletonForm::Setup() since widget_list should be fully created at that time, however we recommend this method instead since it will help avoid a crowded Setup() function and isolate potential sources of errors better.
Our design needs to insert and remove widgets from test_box, for this we need to keep track of the widget that is being tested at the time and should be removed from test_box at next test request.

- To do this edit skeleton.h to add a private member like this:

```
// Skeleton FORM  CLASS
class SkeletonForm: public VDKForm
{
// gui object declarations
private:
// ....
    VDKObject* tested_widget; // store presently tested widget address
public:
// ...
};
```

- And edit skeleton.cc

```
void SkeletonForm::Setup(void)
{
  GUISetup();
  // put your code below here
  tested_widget = NULL;
  BackgroundPixmap = new VDKRawPixmap(this,"./fuzzy.xpm");
}
```

to initialize tested_widget to NULL (at setup no widget is under test).

11 The call is gtk_widget_realize(GtkWidget*)

Now the procedure should be more clear:
- connect with “Test” button click
- when the button is clicked we get which item of combo box is selected
- remove presently test widget (if any)
- replace this with the widget asked by the user.

Let’s go on
- select testBtn node on WI
- switch on page tab labeled “Signal”
- click on “clicked” button
- select corresponding signal map entry
- click on list column header labeled “jump or write to signal response method”

Source editor will show response method on skeleton.cc:
```
//signal response method
bool
SkeletonForm::OntestBtnClick(VDKObject* sender)
{
 return true;
 }
```
To know which item is selected into combo box we use VDKCombo::Selected property that returns an integer >= 0 if something is selected, −1 if no item is presently selected. So we are able to write our response method to “Test” button click limiting it, for the moment, to testing VDKCustomList.
Even writing a limited part of this response function will give us the occasion to show other ways to use VDK signal system including the use of lower level gtk+ calls.
So let’s take a step by step look at OntestBtnClick() and some companion functions with a discussion about signals propagation.:
```
//signal response method
static int destroy_connection = 0;
```
/* we store connection to “destroy” signal in order to disconnect when the widget is really destroyed
*/
```
bool
SkeletonForm::OntestBtnClick(VDKObject* sender)
{
```
/* response function will always receive an argument that is the sender widget (testBtn in our case), It returns a boolean value, “true” means signal was handled and no need to propagate it anymore, a “false” value means let’s proceed with propagation. Signals are emitted by objects to a recursive "pattern visiting" dispatching algorithm. A first recursion is made into the object hierarchy (so called class level) , another is made into object parent hierarchy (so called parent level). Signal flow will be stopped as soon as signal is flagged as "handled" , that is the dispatching algorithm finds a defined signal response table entry (so called slot) for that signal on that object that answered "true".
Here a typical signal path:
- Signal emitted by a widget
- Signal goes to widget class, if handled stops, otherwise recursively goes up to object hierarchy until handled or widget root class (namely VDKObject).
- If not yet handled and the widget has a parent (thus a container which it is contained in) goes to parent class
- if handled it stops otherwise goes up to parent hierarchy until handled or reaches parent root class (again VDKObject class)
- if parent has a parent above it, steps are repeated until outermost parent or it is handled whichever occurs first. Into widget hierarchy there is only one widget that has no parents, this is the application main form, so signal can reach this point if needed.
- If not yet handled signal is "lost".

  Now let’s take a look to our case to see which is the path that “clicked” signal follows from testBtn until SkeletonForm.
  - Signal emitted by testBtn
  - goes into VDKCustomButton
  - up to hierarchy until VDKObject
  - up to parent’s chain:
    - goes on nearest testBtn parent that is lower_left_box and up to his hierarchy
    - to lower_box and his hierarchy (lower_left_box parent)
    - to main_box and his hierachy (lower_box parent)

– to skeleton form where handled and stops. (main_box parent)

This mechanism permits a wide broadcast strategy and allows to inherited signal response among class hierarchy.

*/

/* store which item is selected (form 0 to n−1 where n is items number)

*/

```
int selected = widget_list->Selected;
```

/* if logo image is visible hides it

*/

```
if (image->Visible)
 image->Visible = false;
```

/* otherwise check if there is a non null tested_widget, in this case removes it from test_box When a widget is removed from a container it gets explicitly destroyed as well.

*/

```
else if (tested_widget)
  test_box->RemoveObject (tested_widget);
```

/* checks which item was selected

*/

```
switch (selected)
  {
  case 0: // VDKCustomList
  {
char* titles[] =  { "Column one",  "Column two", "Column three"};
  // creates a VDKCustom list
    VDKCustomList* list  = new VDKCustomList(this, 3, titles);
  // stores it on tested_widget
    tested_widget = list;
```

/* here we demonstrate the use of native gtk+ signal connection calls, since we are in a "C" environment we cannot connect with a class member function but forced to use a class static function, passing it the list address as 2nd arg. Remember, to declare OnWidgetListRealize() into skeleton.h as static. We connect here with "realize" signal to initialize some list properties and fill it with text rows. (this is not necessary however, just for teaching purposes.)

`gtk_signal_connect` wants three args:

– a `GtkWidget*`, so we will obtain it using gtk+ widget access function `VDKCustomList::CustomWidget()`, `VDKCustomList` and `VDKCustomTree` classes are the only case where more general `VDKObject::WrappedWidget()` cannot be used, for all others use the latter.
– A `char*` that is the signal name
– the callback function pointer
– an `void*` (or better a `gpointer`) that is up to user to code and will be passed to callback function. In our case we pass to `OnWidgetListRealize()` the list address itself.

*/

```
  gtk_signal_connect(GTK_OBJECT(list->CustomWidget()),
        "realize",
        GTK_SIGNAL_FUNC(SkeletonForm::OnWidgetListRealize),
        (gpointer) list);
    }
    break;
  case 1:
  case 2:
  case 3:
  default:
    ;

    //tested_widget = NULL;
  }
```

/* now we check it tested_widget is still NULL, in this case we reshow logo image

*/

```
if (!tested_widget)
      image->Visible = true;
```

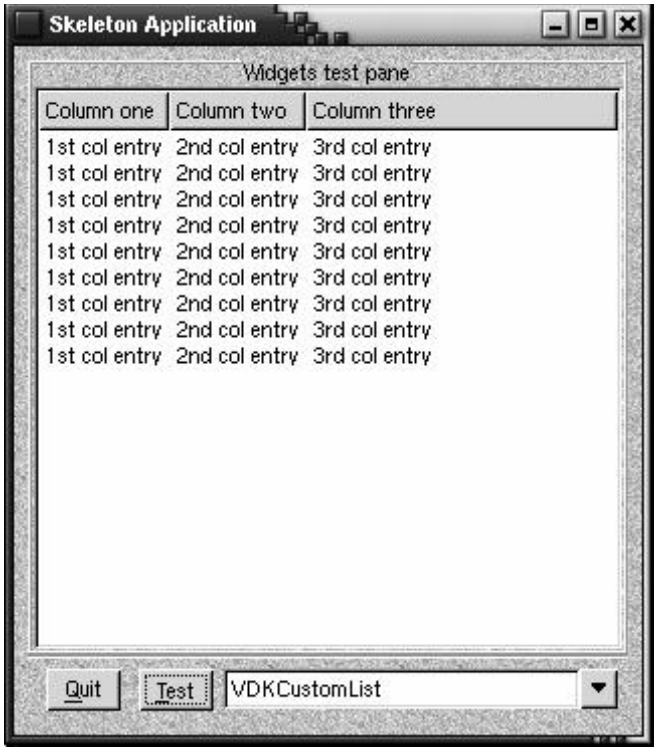
/* otherwise we add the widget to test_box and connect with "destroy" event. Here we demonstrate the use of dynamic signals tables in VDK, in this case we are in "C++" environment and we can connect with class member function passing to SignalConnect his class−offset address, vdk signal system will take care to convert it into an absolute address.

SignalConnect wants three args:

- the widget to connect with
- the signal name
- the class member function that should answer to signal, all response function must be declared as bool SomeClass::ResponseMethod(VDKObject*)

```
    */
    else
    {
      test_box->Add (tested_widget);
      destroy_connection = SignalConnect (tested_widget, "destroy",
                    &SkeletonForm::OnDestroyTestWidget);
      }
    return true;
  }
/*
answers to widget_list realize signal (ala GTK+)
recall that is a class static function
*/
void
SkeletonForm::OnWidgetListRealize (GtkWidget* wid,  gpointer gp)
{
char*  entries[]  =  {  "1st  col  entry","2nd  col  entry",  "3rd  col
entry"};
    /* Here we have to cast from a void* to the correct  pointer type passed on connecting.
    Cast should be safe since if this function was called surely a VDKCustomList* address
    was passed to it.
    */
   VDKCustomList* list  = reinterpret_cast <VDKCustomList*>(gp);
   /* Setting this VDKCustom property allows us to force all columns  to resize width to best fit
   */
   list->AutoResize = true;
   /* we load the list with some texts
   */
   for (int t = 0; t < 9; t++)
        list->AddRow (entries);
}
/*
Answers to a tested_widget destroy signal
*/
bool
SkeletonForm::OnDestroyTestWidget (VDKObject* sender)
{
// disconnect before widget destroy
SignalDisconnect(destroy_connection);
// we reset tested_widget to NULL to mean that no widget
// are contained on test_box
  tested_widget = NULL;
  return true;
}
```

As usual build the program and see the result, selecting “VDKCustomList” on combo box a partially filled list should appear on test box, selecting another item should make logo image to appear again as show on next page.



To see the all the source code generated so far, you'd want to look at all of the following pages:

- skeleton.h
- skeleton_gui.h
- skeleton.cc
- skeleton_gui.cc
- skeleton.frm

We will leave skeleton project as it is so far, we will bring it later to make some enhancements.

skeleton.h

```
/*
skeleton Plain VDK Application
Main unit header file: skeleton.h
*/
#ifndef _skeleton_main_form_h_
#define _skeleton_main_form_h_
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif
// vdk support
#include <vdk/vdk.h>
// Skeleton FORM  CLASS
class SkeletonForm: public VDKForm
{
// gui object declarations
private:
// gui object declarations
  void GUISetup(void);
  VDKObject* tested_widget;
  bool  OnDestroyTestWidget (VDKObject* sender);
  bool OnRealizeCustomList (VDKObject* sender);
  static void OnWidgetListRealize (GtkWidget* wid,  gpointer gp);
  bool OnWidgetListRealize1 (VDKObject *sender);
public:
      SkeletonForm(VDKApplication* app, char* title);
      ~SkeletonForm();
      void Setup(void);
/*
 gui setup include
 do not patch below here
*/
#include <skeleton_gui.h>
};

// Skeleton APPLICATION  CLASS
class SkeletonApp: public VDKApplication
{
public:
SkeletonApp(int* argc, char** argv);
~SkeletonApp();
void Setup(void);
};

#endif

// do not remove this mark: #!#
// end of file:skeleton.h
```

skeleton_gui.h

```
/*
skeleton gui header
*/
protected: VDKBox* main_box;
protected: VDKBox* upper_box;
protected: VDKFrame* frame0;
protected: VDKBox* test_box;
protected: VDKImage* image;
protected: VDKSeparator* separator0;
protected: VDKBox* lower_box;
protected: VDKBox* lower_left_box;
protected: VDKCustomButton* quitBtn;
protected: VDKCustomButton* testBtn;
protected: VDKBox* lower_right_box;
protected: VDKCombo* widget_list;
bool CanClose();
public:
DECLARE_SIGNAL_MAP(SkeletonForm);
bool OnquitBtnClick(VDKObject* sender);
bool Onwidget_listRealize(VDKObject* sender);
bool OntestBtnClick(VDKObject* sender);
/*
declaring signal and events
dynamics tables
*/
DECLARE_SIGNAL_LIST(SkeletonForm);
DECLARE_EVENT_LIST(SkeletonForm);
// do not remove this mark: #!#
// end of file:skeleton_gui.h
```

skeleton.cc

```
 /*
skeleton Plain VDK Application
Main unit implementation file:skeleton.cc
*/
#include <skeleton.h>
/*
main program
*/
int main (int argc, char *argv[])
{
// makes application
SkeletonApp app(&argc, argv);
// runs application
app.Run();
return 0;
}

// Skeleton MAIN FORM  CLASS
/*
main form constructor
*/
SkeletonForm::SkeletonForm(VDKApplication* app, char* title):
      VDKForm(app,title)
{
}

/*
main form destructor
*/
SkeletonForm::~SkeletonForm()
{
}

/*
main form setup
*/
void
SkeletonForm::Setup(void)
{
      GUISetup();
  // put your code below here
  tested_widget = NULL;
  BackgroundPixmap = new VDKRawPixmap(this,"./fuzzy.xpm");
}

// Skeleton APPLICATION  CLASS
/*
application constructor
*/
SkeletonApp::SkeletonApp(int* argc, char** argv):
      VDKApplication(argc,argv)

{
}

/*
application destructor
*/
SkeletonApp::~SkeletonApp()
{ }

/*
application setup
*/

void
SkeletonApp::Setup(void)
{
      MainForm = new SkeletonForm(this,NULL);
      MainForm->Setup();
      MainForm->Visible = true;
```

```
}

// do not remove this mark: #!#
//signal response method
static int destroy_connection = 0;
bool
SkeletonForm::OntestBtnClick(VDKObject* sender)
{
  int selected = widget_list->Selected;
  if (image->Visible)
    image->Visible = false;
  else if (tested_widget)
    test_box->RemoveObject (tested_widget);
  switch (selected)
    {
    case 0: // VDKCustomList
      {
      char* titles[] = { "Column one",  "Column two", "Column three"};
      VDKCustomList* list  = new VDKCustomList(this, 3, titles);
      tested_widget = list;
      gtk_signal_connect(GTK_OBJECT(list->CustomWidget()),"realize",
                 GTK_SIGNAL_FUNC(SkeletonForm::OnWidgetListRealize),
                 (gpointer) list);
      }
      break;
    case 1:
    case 2:
    case 3:
    default:
      ;
      //tested_widget = NULL;
    }
  if (!tested_widget)
    image->Visible = true;
  else
  {
    test_box->Add (tested_widget);
    destroy_connection = SignalConnect (tested_widget, "destroy",
                  &SkeletonForm::OnDestroyTestWidget);
    }

  return true;
 }
/*
*/
void
SkeletonForm::OnWidgetListRealize (GtkWidget* wid,  gpointer gp)
{
   char*  entries[]  =  {  "1st  col  entry","2nd  col  entry",  "3rd  col
entry"};
  VDKCustomList* list  = reinterpret_cast <VDKCustomList*>(gp);
  list->AutoResize = true;
  for (int t = 0; t < 9; t++)
       list->AddRow (entries);
}
/*
*/
bool
SkeletonForm::OnDestroyTestWidget (VDKObject* sender)
{
  SignalDisconnect (destroy_connection);
  tested_widget = NULL;
  return true;
  }
//signal response method
static char* widgets_list_entries[] =
{
  "VDKCustomList",
  "VDKEntry",
  "VDKSCrolled",
  "VDKNotebook",
  NULL
};
```

```
bool
SkeletonForm::Onwidget_listRealize(VDKObject* sender)
{
  StringList sl = widget_list->PopdownStrings;
  for (int t = 0; widgets_list_entries[t]; t++)
    sl.add (widgets_list_entries[t]);
  widget_list->PopdownStrings = sl;
 return true;
 }

//signal response method
bool
SkeletonForm::OnquitBtnClick(VDKObject* sender)
{
  Close ();
  return true;
 }
//asks user before closing
bool
SkeletonForm::CanClose(void)
{
  int answer = Application ()->MessageBox (
                              "Skeleton application",
                              "Really close application ?",
                              MB_YESNO | MB_ICONQUESTION,
                              NULL,
                              NULL,
                              5000);
  return answer == IDYES;
}
// end of file:skeleton.cc
```

skeleton_gui.cc

```
#include <skeleton.h>
/*
defining signal and events
dynamics tables
*/
DEFINE_SIGNAL_LIST(SkeletonForm,VDKForm);
DEFINE_EVENT_LIST(SkeletonForm,VDKForm);
/*
defining signal static table
*/
DEFINE_SIGNAL_MAP(SkeletonForm,VDKForm)
ON_SIGNAL(quitBtn,clicked_signal,OnquitBtnClick),
ON_SIGNAL(widget_list,realize_signal,Onwidget_listRealize),
ON_SIGNAL(testBtn,clicked_signal,OntestBtnClick)
END_SIGNAL_MAP
/*
main form setup
*/
void
SkeletonForm::GUISetup(void)
{
      SetSize(293,400);
      Title = "Skeleton Application";
main_box = new VDKBox(this,v_box);
Add(main_box,0,1,1,0);
main_box->BorderWidth(5);
upper_box = new VDKBox(this,v_box);
main_box->Add(upper_box,0,1,1,0);
upper_box->BorderWidth(0);
frame0 = new VDKFrame(this,"Widgets test
pane",v_box,shadow_etched_in);
frame0->Shadow = shadow_etched_out;
frame0->Align = c_justify;
upper_box->Add(frame0,0,1,1,0);
frame0->BorderWidth(0);
test_box = new VDKBox(this,v_box);
frame0->Add(test_box,l_justify,1,1,0);
test_box->BorderWidth(0);
image = new VDKImage(this,"logo.png");
image->SetSize(259,353);
test_box->Add(image,0,1,1,0);
separator0 = new VDKSeparator(this,h_separator);
main_box->Add(separator0,0,0,0,0);
lower_box = new VDKBox(this,h_box);
main_box->Add(lower_box,0,0,0,0);
lower_box->BorderWidth(5);
lower_left_box = new VDKBox(this,h_box);
lower_box->Add(lower_left_box,0,1,1,0);
lower_left_box->BorderWidth(0);
quitBtn = new VDKCustomButton(this,(char*)
NULL,"_Quit",16,(GtkPositionType) 1);
lower_left_box->Add(quitBtn,0,1,0,0);
testBtn = new VDKCustomButton(this,(char*)
NULL,"_Test",16,(GtkPositionType) 1);
lower_left_box->Add(testBtn,0,1,0,0);
lower_right_box = new VDKBox(this,h_box);
lower_box->Add(lower_right_box,0,1,1,0);
lower_right_box->BorderWidth(0);
widget_list = new VDKCombo(this);
lower_right_box->Add(widget_list,0,1,1,0);
}

// do not remove this mark: #!#
// end of file:skeleton_gui.cc
```

skeleton.frm

User shouldn't care about this file, is here just for completeness.

```
[skeleton]
{
      class:form;
      skeleton.this:skeleton;
      skeleton.NormalBackground:nihil;
      skeleton.Foreground:nihil;
      skeleton.Font:"nihil";
      skeleton.Cursor:nihil;
      skeleton.BackgroundPixmap:nihil;
      skeleton.FocusWidget:nihil;
      skeleton.Usize: 293, 400;
      skeleton.Title:"Skeleton Application";
      skeleton.OnFormActivate:nihil;
      skeleton.OnChildClosing:nihil;
      skeleton.OnConfigure:nihil;
      skeleton.OnExpose:nihil;
      skeleton.OnIconize:nihil;
      skeleton.OnMove:nihil;
      skeleton.OnRealize:nihil;
      skeleton.OnResize:nihil;
      skeleton.OnRestore:nihil;
      skeleton.OnShow:false;
      skeleton.CanClose:true;
}
[object]
{
      this:main_box;
      class:VDKBox;
      parent:nihil;
      _justify:0;
      _Expand:1;
      _Fill:1;
      _Padding:0;
      Tag:nihil;
      declare_public:nihil;
      Usize:nihil;
      BorderWidth:5;
      mode:v_box;
      event-aware:nihil;
}
[object]
{
      this:upper_box;
      class:VDKBox;
      parent:main_box;
      _justify:0;
      _Expand:1;
      _Fill:1;
      _Padding:0;
      Tag:nihil;
      declare_public:nihil;
      Usize:nihil;
      BorderWidth:0;
      mode:v_box;
      event-aware:nihil;
}
[object]
{
      this:frame0;
      class:VDKFrame;
      parent:upper_box;
      _justify:0;
      _Expand:1;
      _Fill:1;
      _Padding:0;
      Tag:nihil;
      declare_public:nihil;
      Usize:nihil;
      BorderWidth:0;
      Label:"Widgets test pane";
```

```
      Shadow:4;
      Align:1;
}
[object]
{
      this:test_box;
      class:VDKBox;
      parent:frame0;
      _justify:l_justify;
      _Expand:1;
      _Fill:1;
      _Padding:0;
      Tag:nihil;
      declare_public:nihil;
      Usize:nihil;
      BorderWidth:0;
      mode:v_box;
      event−aware:nihil;
}
[object]
{
      this:image;
      class:VDKImage;
      parent:test_box;
      NormalBackground:nihil;
      PrelightBackground:nihil;
      InsensitiveBackground:nihil;
      ActiveBackground:nihil;
      SelectedBackground:nihil;
      Foreground:nihil;
      Font:"nihil";
      Enabled:true;
      Cursor:nihil;
      Visible:true;
      Tip:"nihil";
      _justify:0;
      _Expand:1;
      _Fill:1;
      _Padding:0;
      Tag:nihil;
      declare_public:nihil;
      Usize:259,353;
      Glyph:logo.png;
      GlyphBydata:nihil;
}
[object]
{
      this:separator0;
      class:VDKSeparator;
      parent:main_box;
      NormalBackground:nihil;
      PrelightBackground:nihil;
      InsensitiveBackground:nihil;
      ActiveBackground:nihil;
      SelectedBackground:nihil;
      Foreground:nihil;
      Font:"nihil";
      Enabled:true;
      Cursor:nihil;
      Visible:true;
      Tip:"nihil";
      _justify:0;
      _Expand:0;
      _Fill:0;
      _Padding:0;
      Tag:nihil;
      declare_public:nihil;
      Usize:nihil;
      mode:h_separator;
}
[object]
{
      this:lower_box;
```

```
	class:VDKBox;
	parent:main_box;
	_justify:0;
	_Expand:0;
	_Fill:0;
	_Padding:0;
	Tag:nihil;
	declare_public:nihil;
	Usize:nihil;
	BorderWidth:5;
	mode:h_box;
	event-aware:nihil;
}
[object]
{
	this:lower_left_box;
	class:VDKBox;
	parent:lower_box;
	_justify:0;
	_Expand:1;
	_Fill:1;
	_Padding:0;
	Tag:nihil;
	declare_public:nihil;
	Usize:nihil;
	BorderWidth:0;
	mode:h_box;
	event-aware:nihil;
}
[object]
{
	this:quitBtn;
	class:VDKCustomButton;
	parent:lower_left_box;
	NormalBackground:nihil;
	PrelightBackground:nihil;
	InsensitiveBackground:nihil;
	ActiveBackground:nihil;
	SelectedBackground:nihil;
	Foreground:nihil;
	Font:"nihil";
	Enabled:true;
	Cursor:nihil;
	Visible:true;
	Tip:"nihil";
	_justify:0;
	_Expand:1;
	_Fill:0;
	_Padding:0;
	Tag:nihil;
	declare_public:nihil;
	Usize:nihil;
	Caption:"_Quit";
	CaptionWrap:nihil;
	Relief:nihil;
	Glyph:nihil;
	GlyphBydata:nihil;
	pixmapped:nihil;
	cbtype:16;
	labelpos:1;
}
[object]
{
	this:testBtn;
	class:VDKCustomButton;
	parent:lower_left_box;
	NormalBackground:nihil;
	PrelightBackground:nihil;
	InsensitiveBackground:nihil;
	ActiveBackground:nihil;
	SelectedBackground:nihil;
	Foreground:nihil;
	Font:"nihil";
```

```
	Enabled:true;
	Cursor:nihil;
	Visible:true;
	Tip:"nihil";
	_justify:0;
	_Expand:1;
	_Fill:0;
	_Padding:0;
	Tag:nihil;
	declare_public:nihil;
	Usize:nihil;
	Caption:"_Test";
	CaptionWrap:nihil;
	Relief:nihil;
	Glyph:nihil;
	GlyphBydata:nihil;
	pixmapped:nihil;
	cbtype:16;
	labelpos:1;
}
[object]
{
	this:lower_right_box;
	class:VDKBox;
	parent:lower_box;
	_justify:0;
	_Expand:1;
	_Fill:1;
	_Padding:0;
	Tag:nihil;
	declare_public:nihil;
	Usize:nihil;
	BorderWidth:0;
	mode:h_box;
	event-aware:nihil;
}
[object] {
	this:widget_list;
	class:VDKCombo;
	parent:lower_right_box;
	NormalBackground:nihil;
	PrelightBackground:nihil;
	InsensitiveBackground:nihil;
	ActiveBackground:nihil;
	SelectedBackground:nihil;
	Foreground:nihil;
	Font:"nihil";
	Enabled:true;
	Cursor:nihil;
	Visible:true;
	Tip:"nihil";
	_justify:0;
	_Expand:1;
	_Fill:1;
	_Padding:0;
	Tag:nihil;
	declare_public:nihil;
	Usize:nihil;
	Editable:"nihil";
	Hidden:nihil;
	Sorted:nihil;
	CaseSensitive:nihil;
}
[connect]
{	sender:quitBtn;
	signal:clicked_signal;
	slot:OnquitBtnClick;
	declare:1;
}
[connect]
{	sender:widget_list;
	signal:realize_signal;
	slot:Onwidget_listRealize;
```

```
      declare:1;
}
[connect]{  sender:testBtn;
      signal:clicked_signal;
      slot:OntestBtnClick;
      declare:1;}
```

Intentionally left blank page

MENUS AND CHILD FORMS

In this section we will learn how to create menus and child forms. Implementing menus was one of the hardest tasks we faced while making the GUI designer, even with it being the best we could come up with, it isn't completely satisfactory. The main reason is that menus are treated as separated windows, furthermore gtk+ has a very complicated way to handle them. We are forced to accept some limitations, or the "point and click" strategy would have to have been abandoned. We decided for the former even if the way builder constructs menus is somewhat counterintuitive and requires a bit of training. To moderate this drawback we decide to provide an alternative way that uses WI only. So you can make/edit menus:

- using GUI Designer with point and click
- using Widget Inspector, in this case you cannot see immediately the result of your changes on GUI Designer.

Here we will discuss the former only, the latter is more or less equal and requires basically the same process. Some key points should be discussed before proceeding:

- since menus are treated as separate windows you cannot select and mantain a selection on them using a simple mouse lick, as with other widgets, you must select and "freeze" the menu with a double click. To see your changes you have to "unfreeze" the menu since during freezing GUI Designer is locked (otherwise menus won't stay on the screen) and doesn't display the changes.
- you must bear a in mind: there is a difference between a menu and a menu item, the former is just a container that contains menu items, furthermore a menu item can be linked with a menu making submenus. As containers menus can't handle signals, you will connect always with menu item signals.

In VDK there are two types of menus:

- VDKMenubar: is a bar that normally stays on top of the form and that contains menu items
- VDKMenu: a more general menu items container that can be linked with a menu item in a recursive fashion to create sub−menus and so on. VDKMenu can be popped (dragged) as stand alone menu as well.

So let's begin:

- Select in builder "File−>New−>Project" and complete the information for the new project, calling ti "menu', like you did for the "skeleton" project.
- Select in Project manager "menu.frm" and double click on it, you should have the default 400x300 empty form now shown on the screen
- Select "Menu bar" from Containers tool palette and drop it onto the empty form, a menubar will appear stuck on the top of form, named "menubar0"
- Now select menubar0 node on WI and uncheck both "expand" and "fill" check boxes
- Click on "Repack" button. We did this because we want the menubar to always stay at it's minimum size, otherwise it will try, according to variable geometry, to expand on the form.
- Now right click on GUI Designer on menubar to have context pop−menu and choose "Append a menu item", a "menuitem0" will appear on leftmost part of the menubar.
- Repeat above action adding another menuitem to menubar, named "Menuitem1"
- Change "Menubar0" widget name to "main_menubar" and change menu items names "menuitem0 " to "child_menu" and "menuitem1" to "about_menu".
- Change child_menu and about_menu captions to "_Childs..."[12] and "_About" respectively
- Select "about_menu" and select "r_justify" in the "Justification" combo box. This fields let's you control wheter a new widget is packed at the start (l_justify) or at the end (r_justify) of the container. Click on "Repack button", now about_menu is place on rightmost part of the menubar.
- Now right click on GUI Designer on about_menu to have context pop−menu, choose "Set min size", a dialog window will appear, click on "Reset to min size" button and close the dialog. This instructs about_menu to shrink to its minimum size.

Now we will add an icon to child_menu.

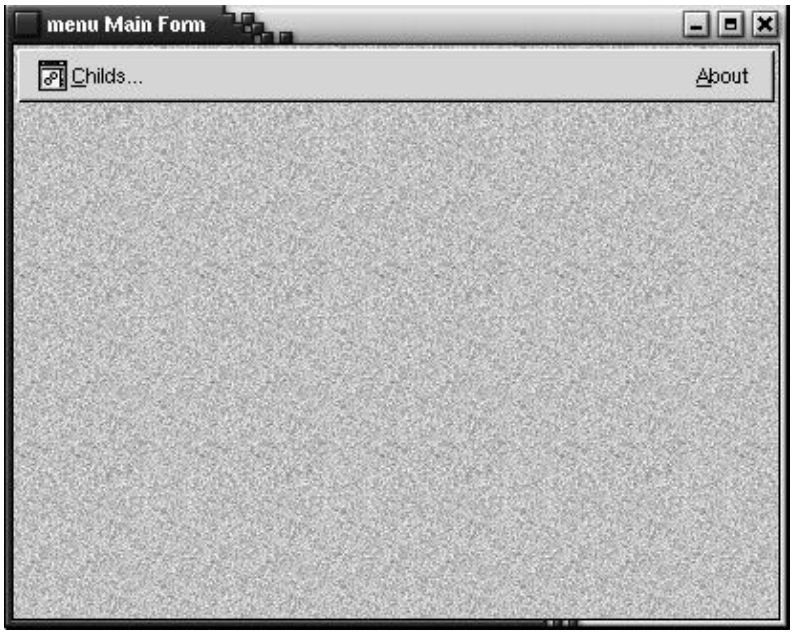
- Select child_menu node on WI and click on "Set Glyph" button, a file open dialog will appear, browse to find <where_vdkbuilder_is>/vdkbuilder/pixmaps and click on "formprops.xpm" file, click on "Open" button, now child_menu will contain a little icon
- click on ".xpm−> project dir" to have formprops.xpm file copied into your project directory.

12 It is just a convention to fill the menu item caption with three trailing dots if it is linked with a submenu, also, note the _in front of Childs telling the UI to treat that as a 'hot key' for the menu.

Now let's create a better background on the form:

- Select "menu" node on WI and click on "Set back map" button, a file open dialog will appear, browse to find <where_vdkbuilder_is>/example/hello/, select "fuzzy.xpm" and click on "Open" button.
- Click on ".xpm−> project dir" to have fuzzy.xpm file copied into your project directory. The fuzzy background will be showed during program run.

Now build and run "menu" program, here the result:

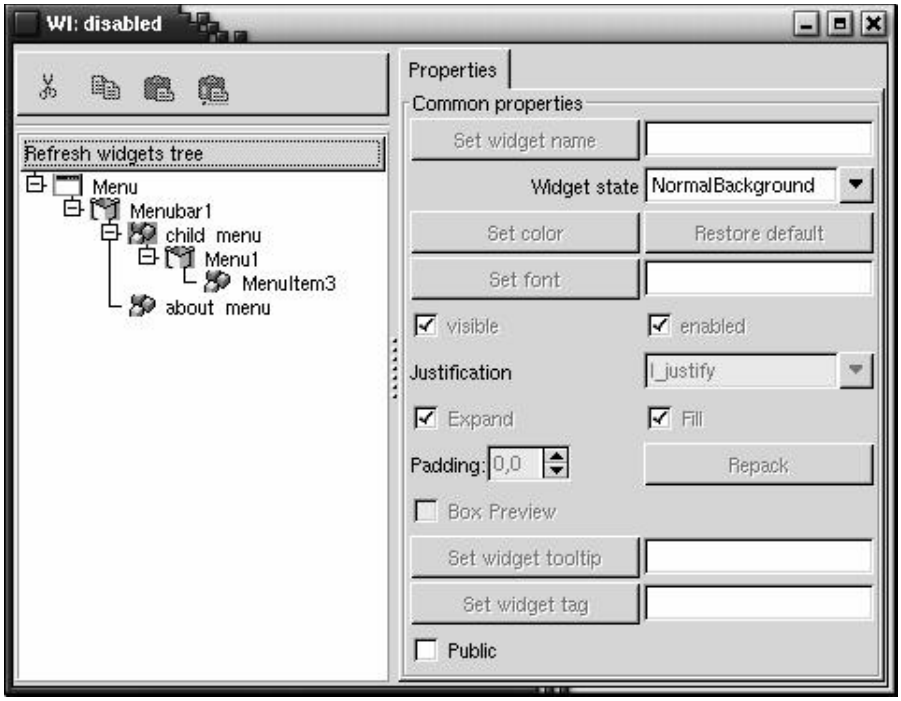


Now we want link "Childs..." menu item to a submenu that contains another three items:

- Modal Dialogs..., Child forms, Quit

We proceed like this:

- Select on GUI Designer child_menu item, right click to access to pop−menu and choose "Add a menu", recall that we have to insert the three menu items into a container (a menu) You will see that a new "Menu1" with a "MenuItem3" added by default. Looking into WI you should have a situation like below



As discussed before we have linked child_menu with a new menu (that becames a sub−menu).

- Now let's change Menu1 name to "child_submenu"
- To insert two more menu items into child_submenu you have to "freeze" it selecting child_menu item and double clicking, otherwise it won't remain selected .
  Remember that double clicking on a menu item will freeze the linked menu if any and if you want to change this menu you have to select it's parent menu item, otherwise you select the contained menu items not the container itself.
- Right click on child_menu to have pop−menu and choose again "Append a menu item"
- Repeat above action

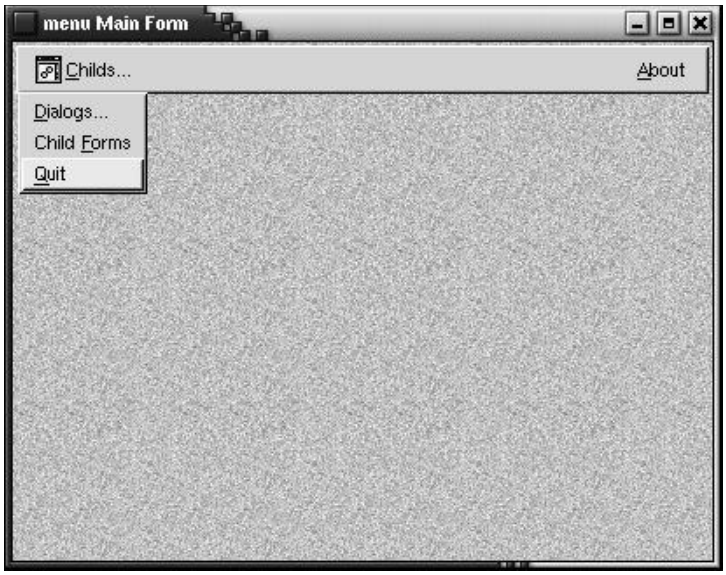
You do not immediately see the appended menu items since menu is still in "freeze" state.

- double click on child_menu to "unfreeze", now selecting it, again, you will see the change you made.

Now let's change child_submenu items name and caption in order

- dialogs_item − "_Dialogs...", childs_items − "Child _Forms", quit_item − "_Quit"

For this it is probably easier and quick to select item nodes in WI.
Here the result at runtime:



Now we want link a sub−sub−menu to dialogs_items adding to new sub−items:

- again select childs_menu and freeze it
- now select dialog_item item
- right click to get pop−up menu and choose "Add menu".
  This is the ame thing you did before adding a sub menu to main_menubar but now you are adding a sub−menu to dialog_item . You will see a new sub−menu with a default menu item linked to dialog_item
- unfreeze with a double click
- select childs_menu and double−click to freeze linked sub−menu
- right click to add two new menu items
- unfreeze to see the result.
- Now use WI to select what you added and change names and caption respectively:
  - Menu2 to dialog_submenu
  - MenuItem6 to fileopen_item and "File Open"
  - MenuItem7 to filesaveas_item and "File Save as"

Now we will add some icons to newly inserted items:

- select fileopen_item and click on "Set Glyph" button, browse in File open dialog until you find <where_vdkbuilder_is>/vdkbuilder/pixmaps and click on "openfiledlg.xpm" file, click on "Open" button, now child_menu will contain a little icon
- Click on ".xpm−> project dir" to have openfiledlg.xpm file copied into your project directory.
- Repeat above steps on filesaveas_item to set savefiledlg.xpm file as menu item glyph.

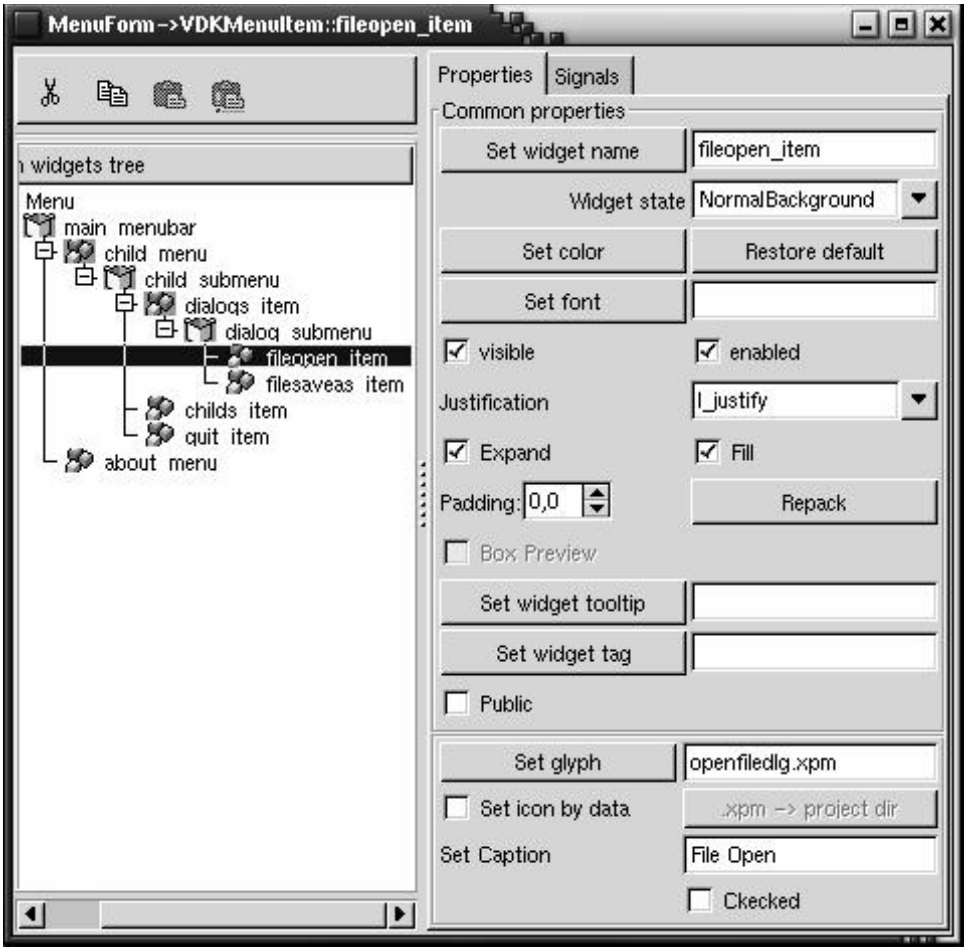
We should have completed the menu now, just let's set all to their minimum size:

- freeze child_submenu, right click to popup menu and choose "Set min size"
- click on "Reset to min size" button on dialogs

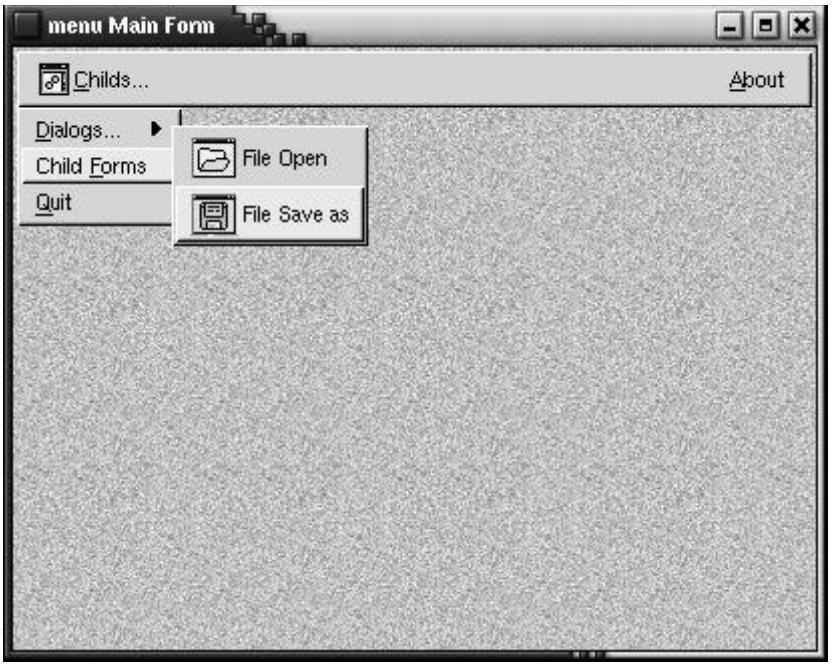
– repeat above also for dialog_submenu

All menus will be adjusted to their minimum size.

Here is what WI should more or less look like:



Admittely the whole procedure is somewhat complicated and counterintuitive but has its own logic and is recursively applicable at any level of nested menu−submenus. Furthermore it is the easier and less error prone than the gtk+ hard way that requires a complicated table be filled. However it is always possible to use WI for working with menus and menu items, with right clicking. The only drawback is that you cannot see the result in GUI Designer (unless you switch each time to−from WI).
What I suggest is to use the GUI Designer "freeze"/"unfreeze" procedure to quickly add all items and menus you need and use WI later on to set/changes all properties as needed. Again you may find it interesting to run the project and to inspect menu_gui.cc and .h to see what builder has done for you.
Here the result at run time:



ABOUT CHILD FORMS

Child forms are those that link their life to their parents in the sense that when the parent dies they should die as well. Obviously a child form can have a shorter life than its parent and can be constructed several times during application execution. Since they are often used to dialogue with the user, they also called “Dialog” forms.
There are two categories of dialog forms:

- Modeless forms
- Modal forms

To the former class belong all child forms that do not halt application execution letting their parent operate, to the latter those that actually stop parents execution until they are closed.
In VDK there are no differences between these two classes other than their behavior when they are displayed, they both belong or are subclasses of VDKForm class. Life management and behaviour is transparent to the user, when a parent gets closed it closes all its child forms as well, showing a modal dialog halts parents operations, a modeless dialog lets parents operate concurrently. The procedure to have a child form is equal to that used for main form:

- construct the child
- setup
- show (either modal or modeless, by default child is modeless however).

In VDK there are some stock modal dialogs that are used to open or save files, send messages to user etc. We have already seen a special modal dialog when we used Application−>MessageBox(). Others provided by gtk+ such as color and font selection aren’t wrapped by vdk since it’s use is so easy that it wasn’t worth the effort (you are welcome to do it and contribute it back.)
What we want do from now on is to use various menu items to demonstrate how to construct and operate on/with various types of child forms, again builder will be making it easy for us to accomplish this task.
Before showing how to make a child form with builder let’s demonstrate how to display a pop−menu on a form. To accomplish this we have to make main form able to answer both keystrokes and mouse button presses, (by default it doesn’t) after that we can connect to these events and handle them.

So edit MenuForm::Setup() like this:

```
/*
main form setup
*/
void
MenuForm::Setup(void)
{
  GUISetup();
  // put your code below here
      /* You use a gtk+ call not wrapped on VDK. It is a one−to−one statement we guess
      isn't worth wrapping since we can access the underlying gtk+ widget easily using
      VDKObject::WrappedWidget(). These gtk+ calls make the form able to answer to the
      button and key presses. (note that mask values can be ored togheter)[13]
      */
  gtk_widget_add_events (WrappedWidget(),GDK_BUTTON_PRESS_MASK |
                                    GDK_KEY_PRESS_MASK);
  // connects form to button click and key press
  EventConnect("button_press_event",&MenuForm::OnButtonPress);
  EventConnect("key_press_event",&MenuForm::OnKeyPress);
}
```

Now let's write response methods code (remember to declare them into menu.h as well)

```
/*
  answers to alt−p or alt−P key strokes on main form
*/
bool
MenuForm::OnKeyPress (VDKObject* sender, GdkEvent* e)
{
      /* here we down cast from a generic event class to key event class. Cast is safe since if
      this function has been called, surely <e> is a key event
      */
  GdkEventKey* event = (GdkEventKey*) e;
  bool isCtrl = event−>state & GDK_CONTROL_MASK;
  bool isKey = (event−>keyval == GDK_P) || (event−>keyval == GDK_p);
  if(isCtrl && isKey)
    child_submenu−>Popup ();
      /* note that here we answer false, letting key event be further propagated, otherwise main
      form menu key accelerators wont work.
      */
  return false;
}
/*
  answers to button press on main form
*/
bool
MenuForm::OnButtonPress (VDKObject* sender, GdkEvent* e)
{
  GdkEventButton* event = (GdkEventButton*) e;
  if (event−>button ==  3)
      /* right button,
      1 = left, 2=middle(if any),5,6 = mouse wheel
      (if any) */
  child_submenu−>Popup ();
  return true;
}
```

Now it's the time to connect various menu items with signals to activate them.
So:

- select quit_item, switch to Signals page and connect with "activate" signal as explained before.
- repeat as above also for :
  - childs_item
  - fileopen_item
  - filesaveas_item

You will see in menu.cc that builder has wrote all response method bodies:

13 You can find more about GDK enums and types into <prefix>/include/gtk−2.0/gdk/gdkevents.h, <prefix>/include/gtk−2.0/gdk/gdktypes.h, <prefix>/include/gtk−2.0/gdk/gdkkeysyms.h where <prefix> is where you have gtk+ installed (either /usr or /usr/local most likely)

```
//signal response method
bool
MenuForm::Onfilesaveas_itemActivate(VDKObject* sender)
{
 return true;
 }

//signal response method
bool
MenuForm::Onfileopen_itemActivate(VDKObject* sender)
{
 return true;
 }

//signal response method
bool
MenuForm::Onchilds_itemActivate(VDKObject* sender)
{
 return true;
 }

//signal response method
bool
MenuForm::Onquit_itemActivate(VDKObject* sender)
{
 return true;
 }
```

Just modify `Onquit_itemActivate()` to read:

```
bool
MenuForm::Onquit_itemActivate(VDKObject* sender)
{
 Close();
 return true;
 }
```

build and run menu project and you can see that hitting alt−Q or alt−q or selecting “Quit” menu item application will quit.

Now let’s make a menuForm child using VDKBuilder.

- Use File−>New−>Form−>Default menu or simply click on “New form” icon on speed bar.
- A File Save As.. dialog will appear, now edit “Save file as” field entering “child.cc”
- Click on on “Save” button, you will be prompted that the new file was correctly saved and you have to add it to Project manager. In the background builder made these files:
  - child.cc
  - child.h
  - child.frm

  These are the basic files needed to make a new form.
- Now select on Project Manager the menu.prj node and click on “Add” icon.
  An “Add a unit to project” dialog will appear, select child.cc and click on “Open” button (or double click on selected file), the new form will be added to the project. Looking at child.cc and .h you will see that builder has named the child form using usual conventions explained before, in this case the new child class name is ChildForm. Also the remaining part of the code is more or less equal to the code that you saw when you newly constructed MenuForm. The major difference is in the ChildForm’s constructor since this is a child form and it will be constructed with a VDKForm as owner and not an application object like main form is constructed with.
- Double clicking on child.frm node on Project Manager you can access to GUI Designer for ChildForm; now there are two GUI Designers on the screen, in general you can have as many GUI Designers as forms and they can be used concurrently, Widget Inspector will switch from a form to form as needed; obviously if the screen is too crowded you can close some GUI Designer, it will be reopened as you click on related .frm node in Project manager.
- Now manage to shrink ChildForm a little bit, drop on it a vertical box and add to the box just a button, name it child_button and set its Caption to “Child Button”as shown on next picture.

- Select on WI child button and connect it with “clicked” signal as explained several times before.
- Now check “Public” checkbox, this will make child_button a public member of ChildForm class, the reason for this setting will be more clear later.

Now let’s prepare to construct and show ChildForm from MenuForm.

- Edit `MenuForm::Onchilds_itemActivate()` on menu.cc to read:

```
  //signal response method
bool
MenuForm::Onchilds_itemActivate(VDKObject* sender)
{
  ChildForm* child = new ChildForm (this, NULL);
  child->Setup ();
  child->ShowModal (GTK_WIN_POS_CENTER);
  return true;
 }
```

- Remember to include child.h at the top of menu.cc to properly declare ChildForm class:

```
  #include <child.h>
```

Building and running the project you will see that activating “Child forms” menu item a child form will be displayed centered on the screen, child form will stop parent execution until it is closed since it was displayed by calling ShowModal(), a simple Show() call would have displayed it as modeless dialog. Now lets see a demonstration of VDK signal system flexibility and broadcast strategy. What we plan to do is to answer to child_button click not only in child form but, also, in its parent. To achieve this result we have to connect with child_button “clicked” signal also in MenuForm, that’s the reason why we declared it as a public ChildForm member. We know that returning false from a signal response lets the signal be propagated to widget parents until outermost one, the outermost parent of child_button is ChildForm but since this is a child of MenuForm the signal should reach this level, also.

- So edit ChildForm::Onchild_buttonClick() on child.cc to read:

```
  //signal response method
  bool
  ChildForm::Onchild_buttonClick(VDKObject* sender)
  {
     printf ("\nChildForm::Onchild_buttonClick() called - sender:%p",
      sender);
    fflush (stdout):
  // returning false lets the signal proceed to parents
    return false;
   }
```

- Edit `MenuForm::Onchilds_itemActivate()` on menu.cc to read:

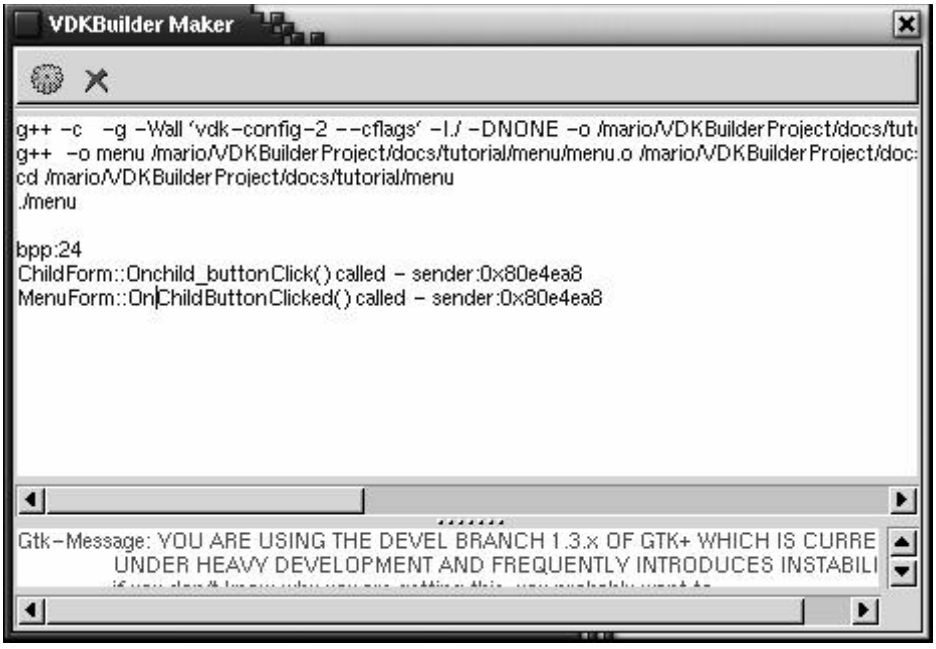
```
  //signal response method
  bool
  MenuForm::Onchilds_itemActivate(VDKObject* sender)
  {
    ChildForm* child = new ChildForm (this, NULL);
    child->Setup ();
          /* since child_button is a public ChildForm member is accessible
          from here
          */
    SignalConnect (child->child_button, "clicked",
                &MenuForm::OnChildButtonClicked);
    child->ShowModal (GTK_WIN_POS_CENTER);
    return true;
   }
```

Now will write the code for signal response into MenuForm:

```
/*
  answers to child_button clicked signal
  into child form
*/
bool
MenuForm::OnChildButtonClicked (VDKObject* sender)
{
  printf ("\nMenuForm::OnChildButtonClicked() called - sender:%p",
                                sender);
  fflush (stdout);
  return true;
}
```

Don't forget to declare `bool OnChildButtonClicked (VDKObject* sender)` in menu.h as well.

Building and running the program should verify that displaying a child form and clicking on "Child Button".you will see this output on VDKBuilder Maker:



Demonstrating that "clicked" signal, after have been answered at ChildForm level was propagated and answered also by MenuForm level. This tecnique gives you a powerful tool that can be used to communicate between forms, even sending your own signals.VDK signal system isn't constrained to managing only gtk+ signals but can emit and handle its own 'custom' signals.
Let's explain how and suppose we want to emit our own signal to parent form when child_button is clicked, let's name this signal "my_own−signal".

– So edit ChildForm::Onchild_buttonClick() on child.cc to read:

```
//signal response method
bool
ChildForm::Onchild_buttonClick(VDKObject* sender)
{
   printf ("\nChildForm::Onchild_buttonClick() called - sender:%p",
    sender);
  fflush (stdout);
  child_button->SignalEmit ("my_own_signal");
  return false;
 }
```

– And edit `MenuForm::Onchilds_itemActivate()` on menu.cc to read:

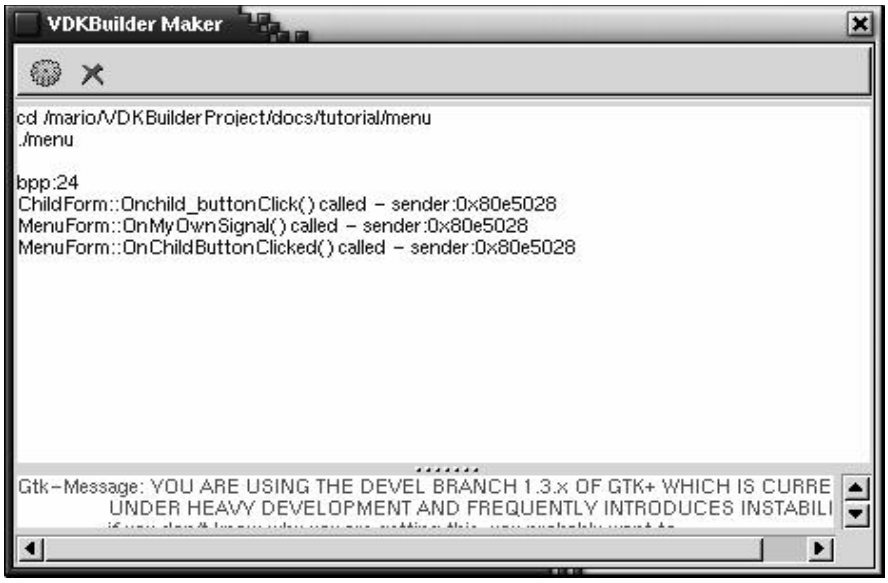
```
bool
//signal response method
MenuForm::Onchilds_itemActivate(VDKObject* sender)
{
  ChildForm* child = new ChildForm (this, NULL);
  child->Setup ();
  SignalConnect (child->child_button, "clicked",
              &MenuForm::OnChildButtonClicked);
  // connects with an user signal
  SignalConnect (child->child_button, "my_own_signal",
              &MenuForm::OnMyOwnSignal, false);
  child->ShowModal (GTK_WIN_POS_CENTER);
  return true;
 }
```

/* here we connect also with child_button “my_own_signal”, note last argument set to “false” value, this will tell VDK that this is an user signal, gtk+ signals will be overridden and VDK will manage it by its own.
*/

– Now we will write the code for signal response into MenuForm:

```
/*
answers to “my_own_signal”
*/
bool
MenuForm::OnMyOwnSignal (VDKObject* sender)
{
  printf ("\nMenuForm::OnMyOwnSignal() called – sender:%p", sender);
  fflush (stdout);
  return true;
}
```

Remember to declare `bool OnMyOwnSignal (VDKObject* sender);` in menu.cc
Building and running the program you should verify that showing a child form and clicking on “Child Button” you will see this output on VDKBuilder Maker:



Demonstrating that the user signal was treated as if it were a normal gtk+ signal, keep in mind that user signals take precedence over gtk+ ones. Just a final note, most likely is more interesting to connect with user signals using the child form instead of an inner widget (parent). Iin this case it isn’t necessary to have a public member in parent form, just remember to connect it with child form and make the child form emit the signal, so these lines should be corrected:

```
// connects with an user signal
    SignalConnect (child, "my_own_signal",&MenuForm::OnMyOwnSignal,
    false);
SignalEmit ("my_own_signal");
```

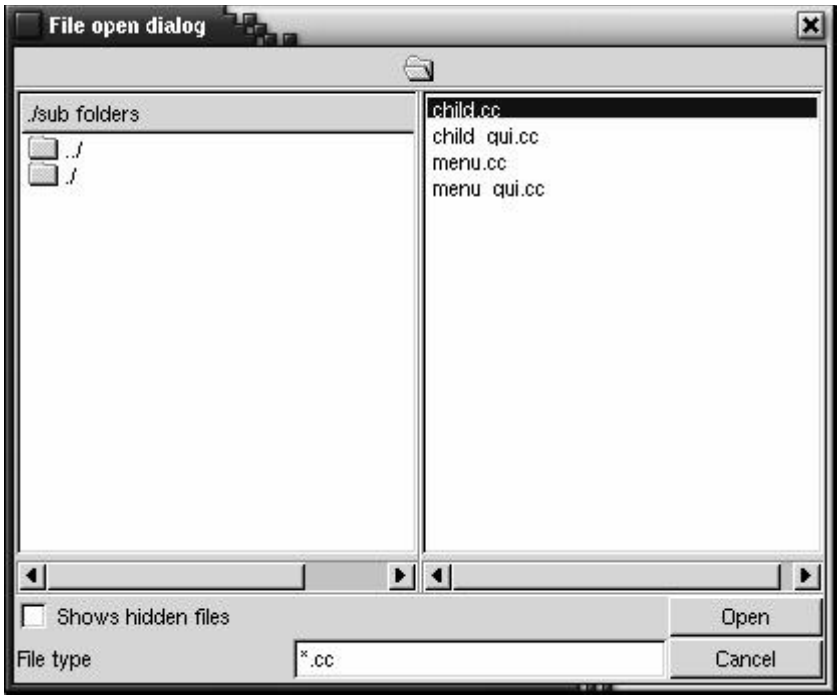
To complete our menu project we have to fill other response methods. Let's begin with fileopen_item:

– edit `Onfileopen_itemActivate()` on menu.cc to read:

```
//signal response method
#include <vdk/Filedialog.h>
bool
MenuForm::Onfileopen_itemActivate(VDKObject* sender)
{
  FileStringArray selections;
  VDKFileDialog* dlg = new VDKFileDialog (this,
            &selections, "File open dialog");
  dlg->Filter = "*.cc";
  dlg->ShowModal ();
  for (int t = 0; t < selections.size (); t++)
    printf ("\nselection:%s", (char*) selections[t]);
  fflush (stdout);
  return true;
 }
```

The code is more or less self explanatory, VDKFileDialog is a "stock" dialog that let's the use browse his directories and choose a file to open.

Build and run and see the result when you activate "File Open" menu item:



Similar dialog shows for the stock dialog to save files.

– edit `Onfilesaveas_itemActivate()` on menu.cc to read:

```
//signal response method
#include <vdk/FileSaveAsDialog.h>
bool
MenuForm::Onfilesaveas_itemActivate(VDKObject* sender)
{
  FileStringArray selections;
  VDKFileSaveAsDialog* dlg = new VDKFileSaveAsDialog (this,
                    &selections, "File open dialog");
  dlg->Filter = "*.cc";
  dlg->ShowModal ();
  if (selections.size () > 0)
      printf ("\nFile to be saved:%s", (char*) selections[0]);
  fflush (stdout);
  return true;
 }
```

Saving file dialog do not handle multiple selections and warns user if he/she attempts to overwrite selected file.

Intentionally left blank page

USING A FIXED CONTAINER

In this section we will explain how use a container with a fixed geometry. As stated earlier this type of container leaves widgets at a fixed size and position, resizing the container and/or changing fonts does not shrink/grow the contained widgets.

It's not recommended to use this type of container since there are several reasons that support for window resizing is necessary, such as different themes, fonts, screen resolution and so on. Nevertheless they can be useful in some cases and especially when coming from a Windows environment, where this container exhibits the behavior you would be used to.

So let's make a simple project to demonstrate the use of a fixed container.

- Let's create a new project that we call "fixed"
- Once you have the main form on the screen select from "Containers" tool palette a "Fixed" container and drop it into the main form.
- Resize the form to make it bigger than 400x300 (which is the default).
- Clicking on the fixed you will see that a dotted grid is showed. By default it has a grif with 8x8 steps but can be customized to have fewer or more steps using WI. (Obviously this grid won't be show at run time, it is just there to help you to align widgets.)

A few words about moving/resizing widgets in a fixed container, it can be done either using mouse or keyboard, dragging the widget to move it or clicking on the widget corners to drag its dimensions. However, using mouse does not work very well, some widgets does not move at all, others show a "gummy" behaviour and other drawbacks. So it is recommended to use the keyboard that works well and assures a finer tuning.

Here the key bindings:

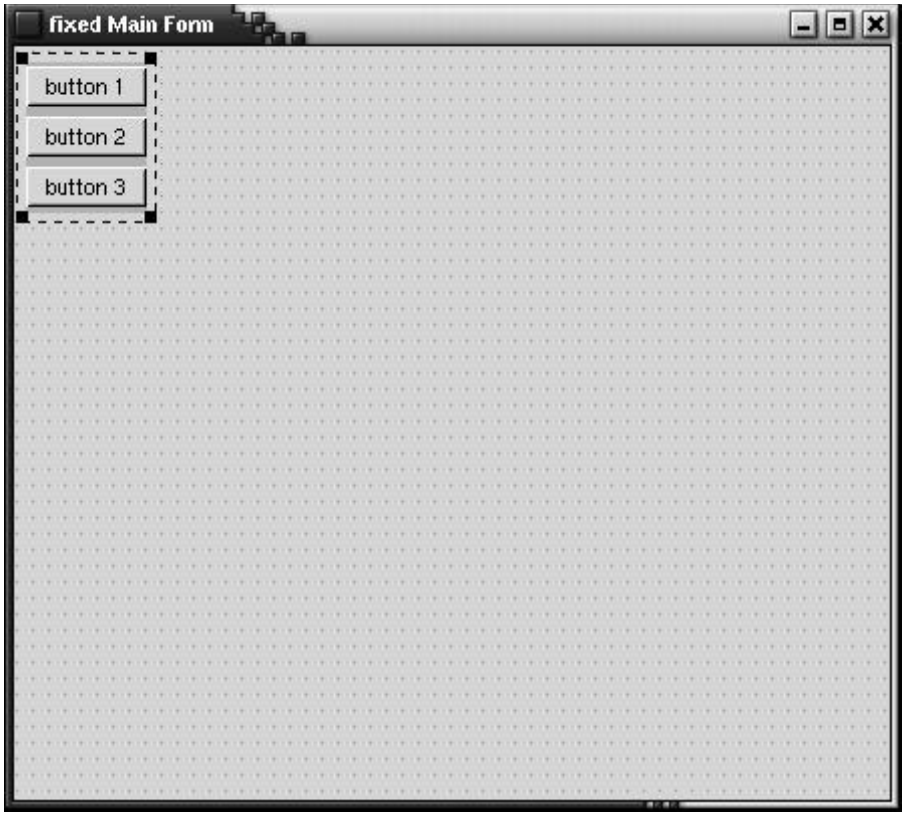
- Moving widgets
  - up,down,left,right arrow keys will move the widget 1 pixel toward related direction
  - Shift+arrow keys will snap the widget toward the nearest grid step in the related direction
- Resizing widgets
  - Ctrl+arrow keys resizes the widget toward related direction by 1 pixel
  - Ctrl+Shift+arrow keys resizes the widget toward nearest grid step.

Same behavoir (included Shift/Ctrl/Ctr+Shift) can be seen with the mouse dragging it into the position or the desired size. Alignment tools do not exist since (here a good news) is possible insert a variable container, move and resize it. For instance a box inserted into the container even if remains into a fixed size/position is able to apply his variable geometry inside himself.

So

- select a vertical box from "Containers" tool palette and drop it into fixed
- resize it to accomodate 3 buttons, let's say a 100x100
- move it to left upper form corner
- set box border to 5
- now drop into the box 3 buttons
- change their captions to "button 1", "button 2" and "button 3" .
- select each button and uncheck "Fill" checkbox leaving "Expand" one checked, then click on "Repack" button
- resize again the box until satisfied and you will see all three buttons aligned and equally spaced into the box. (If you have difficulty selecting the box, right click on a button and use "Select parent container" option.)

The result is shown on next page.



There isn't much more interesting about fixed containers, appart from their geometry difference, they are the same as other containers.
Just a final note, a fixed container could be useful when the window can't be resized, such as an always maximized one or other cases. Making a form not resizeable is related with window managers, some of them will honour this request, others don't.
However let's have a not−resizeable form, into FixedForm::Setup() edit to read:

```
/*
main form setup
*/
void
FixedForm::Setup(void)
{
GUISetup();
// put your code below here
gtk_window_set_resizeable (GTK_WINDOW(WrappedWidget ()),false);
}
```

again here is an example of VDK policy, where it's probably useless to wrap a one−to−one statement, better go ahead and use the lower level gtk+ call accessing the underlying widget, in this case a GtkWindow. Build and run the project, if your window manager is clever enough, the form can't be resized.

WIDGETS FOR DRAWING

VDK has one widget that is specialized for drawing, it's class name is VDKCanvas. Mainly it is a drawing area that provides you with all the necessary methods to draw usual geometrical figures and show pixmaps as well. Drawing operations on canvas are not done directly on the screen but using an offscreen backing buffer. When necessary relevant portions of (or all of) backing image is copied to the screen. This tecnique makes drawing operations an order of magnitude faster and avoids screen flickering. So all drawing should be followed by Redraw() to force drawing to be seen. When applicable the best way is to establish an expose_event response function, where all drawing takes place followed by a call to Redraw().[14]
For those interested in displaying 2D graphics also VDKChart and his subclasses are provided.
In this section you will see these two widgets in action and also how to use VDKBuilder with widgets not directly supported by builder or even our own developed widgets.

- Let's make a new project naming it "draw"
- As usual insert a vbox on the form. (vbox1)
- Insert into vbox1 a vertical box (vbox2) and an horizontal box (hbox1)[15]
- Set vbox2's border to 2 pixel
- Select hBox1 and uncheck both "fill" and "expand" check boxes, click on "Repack" button. Notice that selecting a geometry property to a container will apply also to contained widgets if any.
- Insert into hbox1 4 buttons and set names and captions respectively:
  - sinBTn, "_Sin"
  - cosBtn, "_Cos"
  - sin2Btn, "_1−sin2"
  - quitBtn, "_Quit"

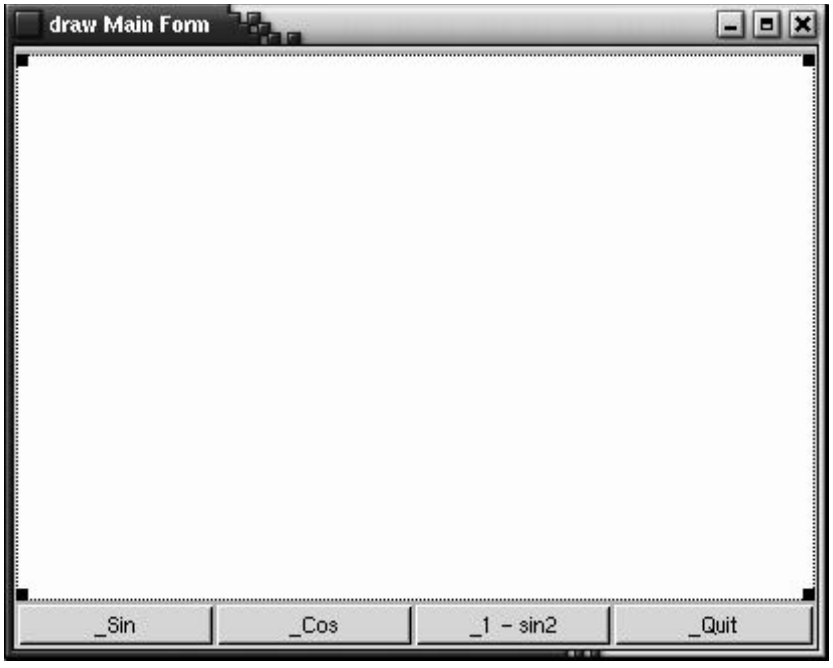
As their captions say we want plot into the canvas the functions: sin,cos and $1-\sin^2$

- Select from "Misc" tool palette a canvas and drop it into vbox1 (canvas0)
- Select canvas0 and name it simply canvas.
- Select "NormalBackground" on "Widget state" combo box and click on "Set color" button.

A color dialog will appear:

- click on "Ivory" color and click on "Accept" button

canvas background will appear in ivory white. Here the result:



Since we want to plot trigonometrical functions we need some convenience functions, let's say:

- a degree to radians conversion functions (standard C lib maths operate in radians)
- a sin,cos computing function in degrees

14 Recently a new widget was added to VDK: VDKDrawingArea, it has a very similar API and does not need a Redraw() call since double−buffering is implemented by Gtk+ at lower level. VDK test program contains an example of its use.

15 Since object name are auto−numbered by default may be names won't match, isn't a problem however.

– since we desire scale function values to canvas dimension we need a function that returns the "actual" size of canvas (recall that user can resize form)
– Since our coordinate system has the origin at point(0,canvas height/2) we will draw x axis as well.
– We need also a simple object that describes a geometrical point with double precision values. We will use a VDKArray template class to store computed points.
– We also keep track of the button that was clicked last (the reason will be clear soon).

So let's edit draw.h to read:

```
class DPoint
{
  public:
  double x,  y;
  DPoint (double x = 0.0,  double y = 0.0):x (x), y (y) {}
  ~DPoint () {}
  // these operators are needed by VDKArray class
  // here unused so let's return always false for sake of simplicity
  bool operator == (DPoint& d) { return false; }
  bool operator < (DPoint& d) { return false; }
};
typedef VDKArray <DPoint> PointArray;
// Canvas FORM  CLASS
class CanvasForm: public VDKForm
{
// gui object declarations
private:
// gui object declarations
  void GUISetup(void);
  PointArray pArray; // stores computed points
  DPoint CanvasSize (); // returns canvas actual size
  VDKCustomButton *active; // stores active button
public:
 // ...
};
```

and draw.cc to read:

```
/*
draw Plain VDK Application
Main unit implementation file:draw.cc
*/
#include <draw.h>
#include <math.h>

#ifndef PI
#define PI 3.141592653
#endif
// converts degrees to radians
inline double deg2rad(double d)
{
    return d*PI/180.0;
}
// returns canvas actual size
inline DPoint
DrawForm::CanvasSize ()
{
      /* Gtk+ GtkWidget class has a structure with an "allocation" field that holds actual widget
      size, here we can access the underlying Gtk+ widget and read it's information.
      The macro GTK_WIDGET(address) is the Gtk+ way to safe cast from a widget class to
      another. (Even if written in C, Gtk+ was designed in an OO fashion).
      */
 double w = GTK_WIDGET (canvas->WrappedWidget ())->allocation.width;
 double h = GTK_WIDGET (canvas->WrappedWidget ())->allocation.height;
 return DPoint (w, h);
}
```

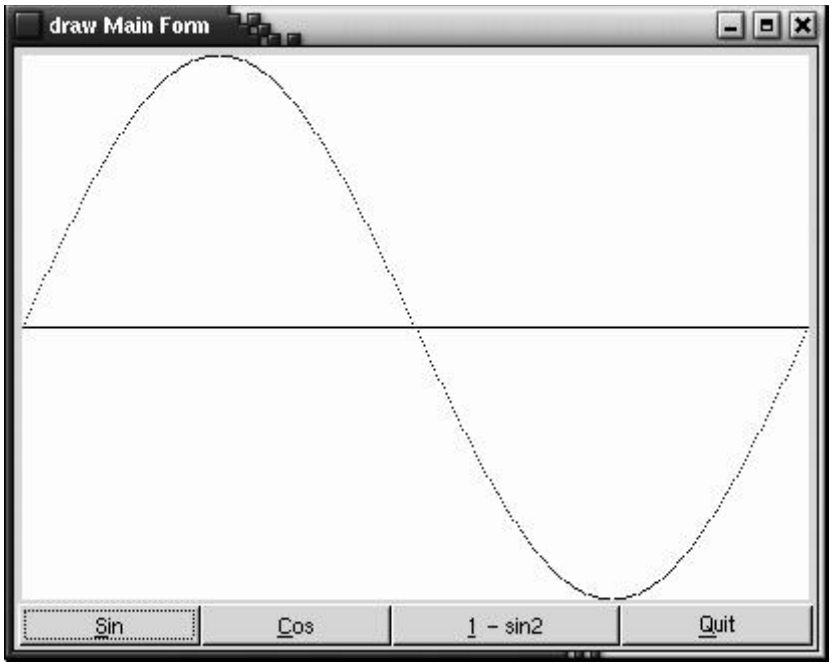
Now going on with "draw" application is more easy, we plan to connect with sinBtn "clicked" signal, compute and plot sin function scaled into the domain represented by canvas actual size.

− Select sinBtn and connect with clicked signal

Now edit DrawForm::`OnsinBtnClicked()` method generated by builder and and chage it to read:

```
bool
DrawForm::OnsinBtnClick(VDKObject* sender)
{
 DPoint size = CanvasSize ();
 canvas->Clear ();
 canvas->Foreground = VDKRgb ("black");
 // draw x axis
 canvas->DrawLine (0,(int) size.y/2, (int) size.x, (int) size.y/2);
 // compute x step to bound 0 - 360°
 double xstep =  size.x/360.0;
 // resize and fill array with scaled values
 pArray.resize (360);
 int z;
 for ( z = 0; z < 360; z++)
      {
      double x = z*xstep; // x values are degrees
      pArray[z] = DPoint (x, size.y/2-((size.y/2)*sin (deg2rad (z))));
      }
 canvas->Foreground = clNavyBlue;
 // plots all computed points on back buffer
 for ( z = 0; z < pArray.size (); z++)
    canvas->DrawPoint ((int) pArray[z].x, (int) pArray[z].y);
 // this call is necessary to shot the back buffer on the screen
 canvas->Redraw ();
 return true;
 }
```

Now build and run the project and see the result obtained by clicking on “Sin” labeled button or pressing alt-s on the keyboard:



Most likely you have already figured out that the graph will disappear if you cover the form, or resize it. That’s normal since VDK is responsible for redrawing the canvas itself but not its content, it’s your responsibility to redraw the content each time there is a need. However there is an event that you can use to redraw the contents of the canvas (or a portion of it) if it needs to be redrown, this event is the “expose_event”, so lets look at how to connect with it:

```
/* main form setup */
void
DrawForm::Setup(void)
{
      GUISetup();
// put your code below here
  active = NULL; // no active button at setup
  EventConnect (canvas, "expose_event", &DrawForm::OnCanvasExpose);
}
```

So `DrawForm::OnCanvasExpose()` is the function called whenever canvas needs to be redrown and there we will write the code. How can we know what is presently on the canvas in order to correctly plot it? Several solutions are possible, such as to store all points in an array, the best is to reuse the code, if we can simulate a button click we get the right one.
Before that, let's modify a bit `DrawForm::OnsinBtnClick()`:

```
bool
DrawForm::OnsinBtnClick(VDKObject* sender)
{
      /* Here we downcast from VDKObject* to VDKCustomButton*, if cast is correct,
      <button> contains a valid address otherwise something went wrong better abort
      */
 VDKCustomButton* button = dynamic_cast <VDKCustomButton*>(sender);
 if (button)
  active = button; // set the last pressed button
 else
  return true;
 DPoint size = CanvasSize ();
 canvas->Clear ();
 //...
}
```

Now let's code `DrawForm::OnCanvasExpose()` (Don't forget to declare it into DrawForm class)

```
/*
*/
bool
DrawForm::OnCanvasExpose (VDKObject* sender,  GdkEvent *e)
{
  // makes active emit clicked signal
  if (active)
    active->SignalEmit (clicked_signal);
  return true;
  }
```

Building and running the project you will see that even if the form is resized or covered, plotting will be mantained and scaled when necessary.
Now let's connect quitBtn with clicked signal to let user quit the application.

- Select quitBtn and connect it with clicked signal
- Write `DrawForm::OnquitBtnClicked()` to read:

```
  //signal response method
  bool
  DrawForm::OnquitBtnClick(VDKObject* sender)
  {
    Close ();
   return true;
   }
```

Now connecting others button to clicked signal and writing sin(x) and $1-\sin^2(x)$ plotting should be relatively simple. Astute reader should have already seen that these function are equivalent to `DrawForm::OnsinBtnClick()` unless computing cos(x) instead sin(x).

- Is it possible to connect different widgets with the same signal on the same response function? Answer is yes.
- Is It possible to know wich widget send the signal? Answer is again yes.

So the solution is to connect all three buttons to `DrawForm::OnsinBtnClick()` and compute sin(x), cos(x) or even $1-\sin(x)^2$ depending on the sender.
Obviously `DrawForm::OnsinBtnClick()` must be modified with this logic, so edit it to read:

```
bool
DrawForm::OnsinBtnClick(VDKObject* sender)
{
 VDKCustomButton* button = dynamic_cast <VDKCustomButton*>(sender);
 if (button)
  active = button; // set the last pressed button
 else
  return true;
 DPoint size = CanvasSize ();
 canvas->Clear ();
 canvas->Foreground = VDKRgb ("black");
 // draw x axis
```

```
canvas->DrawLine (0,(int) size.y/2, (int) size.x, (int) size.y/2);
// compute x step to bound 0 - 360Â°
double xstep =  size.x/360.0;
// resize and fill array with scaled values
pArray.resize (360);
int z;
for ( z = 0; z < 360; z++)
     {
     double x = z*xstep; // x values are degrees
     if (sender ==  sinBtn)
           pArray[z] =
                 DPoint (x, size.y/2-((size.y/2)*sin (deg2rad (z))));
     else if (sender ==  cosBtn)
           pArray[z] =
                 DPoint (x, size.y/2-((size.y/2)*cos (deg2rad (z))));
      else if (sender == sin2Btn)
             {
             double s = sin (deg2rad (z));
             double y = 1-(s*s);
             pArray[z] = DPoint (x, size.y/2-((size.y/2)*y));
            }
     }
canvas->Foreground = clNavyBlue;
// plots all computed points on back buffer
for ( z = 0; z < pArray.size (); z++)
   canvas->DrawPoint ((int) pArray[z].x, (int) pArray[z].y);
// this call is necessary to shot the back buffer on the screen
canvas->Redraw ();
return true;
}
```

What remains to do is connect cosBtn,sin2Btn to `OnsinBtnClick()`, builder let's us do it quickly

- select cosBtn and switch on Signals page
- on the bottom you see a combo box with already connected signal response function
- select "OnsinBtnClick" entry and click on "Reuse this" button
- a new entry will be appended to signal map that shows cosBtn connected with clicked signal and the already written response function.
- Repeat as above also for sin2Btn

Now we have finished, build and make a test run.

A final note about VDKCanvas widget, it isn't a very powerful drawing widget, but it is simple to use. There are better Gtk+ widgets are available like GtkGlArea or and GtkCanvas to make more complex drawing (including in 3D), unfortunatley they weren't wrapped in VDK. Another VDK opiton for this kind of drawing is the VDKSdlCanvas based on the SDL library that requires a separate VDK library named "vdksdl" (and that the SDL library itself). This library provides a powerful drawing widget and sound management tools as well. If you are interested in 2D graphs there is VDKChart class and it's subclasses that provide you with a simply yet powerful tool for plotting 2 dimensional graphics. Using VDKSdlCanvas with builder will be discussed later on appendices.

Intentionally left blank page

HOW USE UNSUPPORTED WIDGETS (PLACE HOLDERS)

There are a few widgets that VDK provides but are not supported by builder, others widgets can be developed by users for their own purposes. How do we use these widgets with VDKBuilder?
There are mainly two solutions:

- make a plugin library and plug them into VDKBuilder
- use a " place holder" widget.

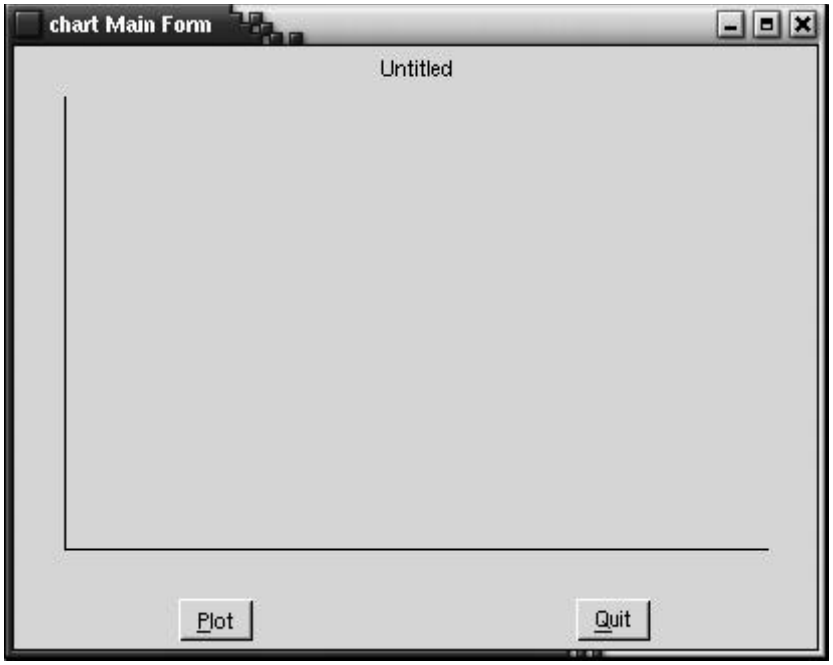
What is a placeholder? As the name itself says, it is a widget that keeps the place of another widget in the GUI Designer. During design time you do not see the real widget but the placeholder one, at run time the situation is reversed (you see the real widget, not the place holder). In this way you can set the widget size/position and even some general properties for the real widget using the GUI Designer and write the code that uses it it with the real widget. What you have to do is let builder know how to construct the widget and access to the real widget code (either in source or object format).
In this example we will use a VDKChart subclass that builder does not support.
So let's begin:

- Make a new project naming it "chart"
- Insert into main form a vertical box
- Insert into main vertical box another vertical box and an horizontal box
- Uncheck horizontal box "Fill" property and click "Repack" button.
- Select from "Misc" tool palette a placeholder widget, this one with a little "ghost" as icon and drop it into the inner vertical box, name it "chart"
- Set inner vertical box "Border" property to 3
- Insert into horizontal box two buttons, name them plotBtn and quitBtn
- set their captions respectively to "_Plot" and "_Quit"

Now let builder know how to construct the real widget that will be a substitute for the placeholder at run time.

- Select chart and edit the field below "Def constructor" button to read : `VDKLineChart(this)`
- Click on "Def constructor" button.

That's all that there is to do with the builder, other widget initialization should be done by coding.
Build and run here the result:



It is interesting to look at chart_gui.h and chart_gui.cc to see what builder wrote for you, it used your VDKLineChart definition to declare the widget and write the constructor. Obviously writing the right definition is your responsibility, builder just use without told it to use.

Now let's write widget initialization code, as shown before the best place is in a "realize" signal response method, so:

- click on chart and select on WI the signal page
- connect with realize signal
- edit `ChartForm::OnchartRealize()` to read:

```
//signal response method
bool
ChartForm::OnchartRealize(VDKObject* sender)
{
  chart->NormalBackground = VDKRgb ("white");
  chart->Foreground = VDKRgb ("navy blue");
  chart->ChartBorder = 50;
  chart->LabelXDigits = 0;
  chart->LabelYDigits = 2;
  chart->LabelX = "deg";
  chart->Title = "red:sin(x) blue:cos(x) green:sin2(x)-cos2(x) ";
  return true;
 }
```

I suspect the above code is pretty self explanatory.

It's time to connect the two buttons:

- select quitBtn, connect it with clicked signal
- Write the code to close the form (left as exercise for the reader)
- select plotBtn, connect with clicked signal and edit OnPlotBtnClick() to read:

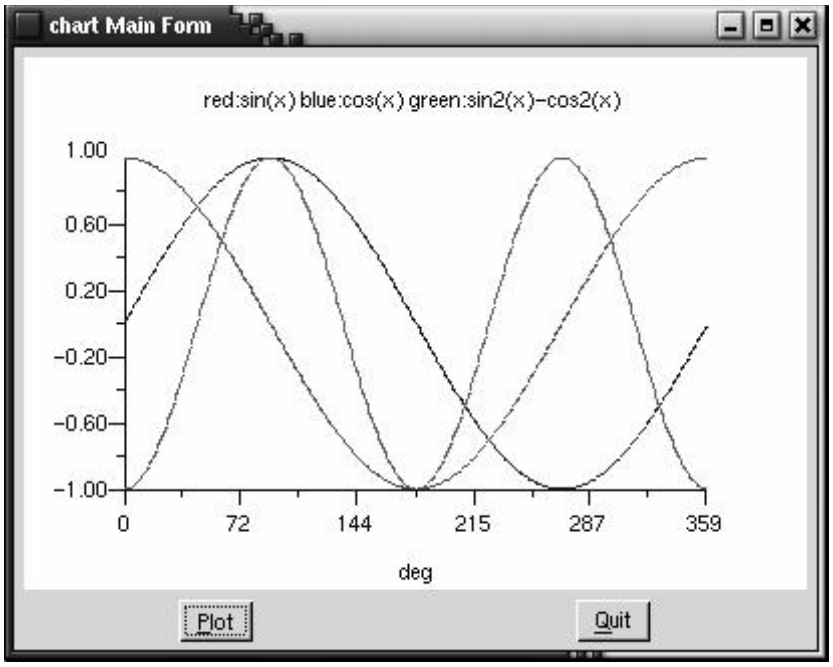
```
#ifndef PI
#define PI 3.14159265358979323846
#endif
inline double deg2rad(double d)
{
    return d*PI/180.0;
}
//signal response method
bool
ChartForm::OnplotBtnClick(VDKObject* sender)
{
  Series* vsin = new Series ("Sin");
  Series* vcos = new Series ("Cos");
  Series* diff = new Series ("sin2-Cos2");
  for (int z = 0; z < 360; z++)
    {
    double s = sin (deg2rad (z));
    double c = cos (deg2rad (z));
    vsin->Add (z,s);
    vcos->Add (z,c);
    diff->Add (z,s*s-c*c);
    }
  vsin->Color = VDKRgb ("blue");
  vcos->Color = VDKRgb ("red");
  diff->Color = VDKRgb ("forest green");
  chart->AddSeries (vsin);
  chart->AddSeries (vcos);
  chart->AddSeries (diff);
  return true;
}
```

- VDKChart subclasses can plot a series of points in different formats:
  - VDKLineChart plots lines between points
  - VDKScatteredChart plots points in x,y coordinates with little squares
  - VDKBarChart plots bars

The use of a VDKChart is straightforward:

- just construct a Series, fill it with values and add the series to a chart.
- the good news is that you do not have to scale values against chart dimension, just fill series with raw computed values and chart will scale them using "actual" chart size as domain, if you resize the chart all values are rescaled with the new widget size.
- Chart can be titled and labeled as needed.
- Series have several properties such as colors and various line styles
- As you can see that you do not need to delete series since they will be freed with the widget, for this reason series should be constructed on the heap with the new operator.
- Adding a new Series with the same name overwrites the old one.
- Clicking on the chart will display more exacts coordinates within a tooltip.

Build and run the project and see the result:



As you can see using lines to plot such tight points doesn't look to good, what you would need is a point plotter instead. VDKChart class has a virtual function Plot() that receives all series points
ready to be plotted, so is would be enough to subclass VDKChart and override Plot() to draw our own graph (i.e. points only.) At this point, we will make a new unit named pointchart.cc and pointchart.h that will contain our PointChart subclass.

- use File−>New−>Unit menu or simply click on "New unit" icon on speed bar
- Source editor will show two new files named "unit1.cc" and "unit1.h"

Edit unit1.h and change it to read:

```
//
#ifndef _point_chart_h
// put your code below here
#define _point_chart_h
#include <vdk/vdk.h>
class VDKPointChart: public VDKChart
{
  public:
  VDKPointChart(VDKForm* owner): VDKChart(owner) {}
  virtual ~VDKPointChart() {}
  // overriddes VDKChart::Plot()
  void Plot(VDKPoint& p, int t,  Series*s);
};
//
#endif
// do not remove this mark: #!#
// end of file:unit1.h
```

- Save the file as pointchart.h

Edit unit1.cc and change it to read:

```
#include <pointchart.h>
/*
Plot a point chart
 */
void VDKPointChart::Plot(VDKPoint& p, int t, Series* series)
{
// each time a new series should be plotted
// set canvas color and line attributes.
// This happens whenever t == 0
  if(t == 0)
    SetColor(series->Color);
  else  if(pixmap)
    // pixmap is a protected member of VDKCanvas class
    gdk_draw_point(pixmap, GC(), p.X(),p.Y());
}
// do not remove this mark: #!#
// end of file:unit1.cc
```

– Save the file as pointchart.cc

Now let's add pointchart unit to our project:

– select chart root node into Project Manager
– in the open file dialog select pointchart.cc and click on "Open" button, or double click on selected file. Pointchart unit will be added to project.

Now you need to change the constructor definition of the place holder:

– select chart and edit "Def constructor" field to read: `VDKPointChart(this)`
– click on "Def constructor" button.

Build and run:

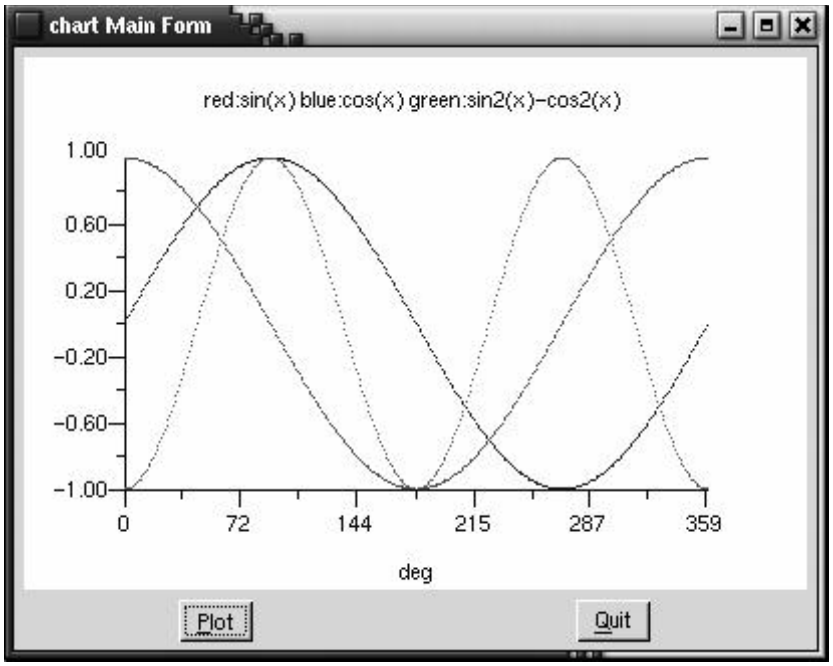


Chart should look nicer.

Once you are convinced that your new widget is ready for use, you can insert it into the project in object format:

– select pointchart node into Project Manager
– click on "Remove" icon
– select chart root node into Project Manager
– on the open file dialog edit Filter field to read "*.o"
– hit Return
– select pointchar.o and click on "Open" button, or double click on selected file.
Pointchart object file will be added to project, from now on it will be linked without beeing recompiled.

Some final notes on using place holders:

- besides the fact that they can be used for unsupported widgets they can be used also to quickly develop and test user developed widgets derived from those provided by VDK, however once finished it is highly recommended that you convert them into shared libs and plugged into the builder.
- They can be used to wrap and test Gtk+ widgets not included in VDK set
- I used this technique while initially developing VDK and VDKBuilder itself.

Intentionally left blank page

VDK − LIBSIGC++ SIGNAL SYSTEM EXTENSION

Since VDK version 1.2.0, the signal system used by VDK was extended to support libsigc++. This section will give you a short introduction of the underlying concepts and how to use it with builder.

- To enable libsigc++ extension you should have libsigc++ installed either from your Linux distribution or downloading it from the web site at url: http://libsigc.sourceforge.net.
  You will need libsigc++ version 0.85 or newer, at the time of this writing latest version was 1.0.1
- If not already done you have to build vdk configuring it for libsigc:

```
$ ./configure [others options]−−enable−sigc=yes −−enable−testsigc=yes
$ make
# make install
```

A few words about libsigc++

Libsigc++ (LS) is a standalone general purpose typesafe signal system. The basic idea is the same for all signal systems: you have a *signal−sender*, a *signal−receiver* and *connections* to establish information flow. In such a template there would have to be #+1 template parameters, the first indicates the return type, others are the parameter types for the function call. LS provides templates for signal up to seven values, but code for more parameters can easily be generated with a m4 script. A connection between a signal and function is established by calling to connect() method for the signal with a handler called slot. This slot is made by a factory named “slot”. This call will return you a connection you can use for further control (e.g. disconnect parts). The callback function can either be a normal global function or a class member function leading to different versions of the slot factory. To get slot from an object you must inherit from SigC::Object base class.

Now let’s see an example showing basic operations. For this you will use the builder making a “console ” application since we do not need any GUI.

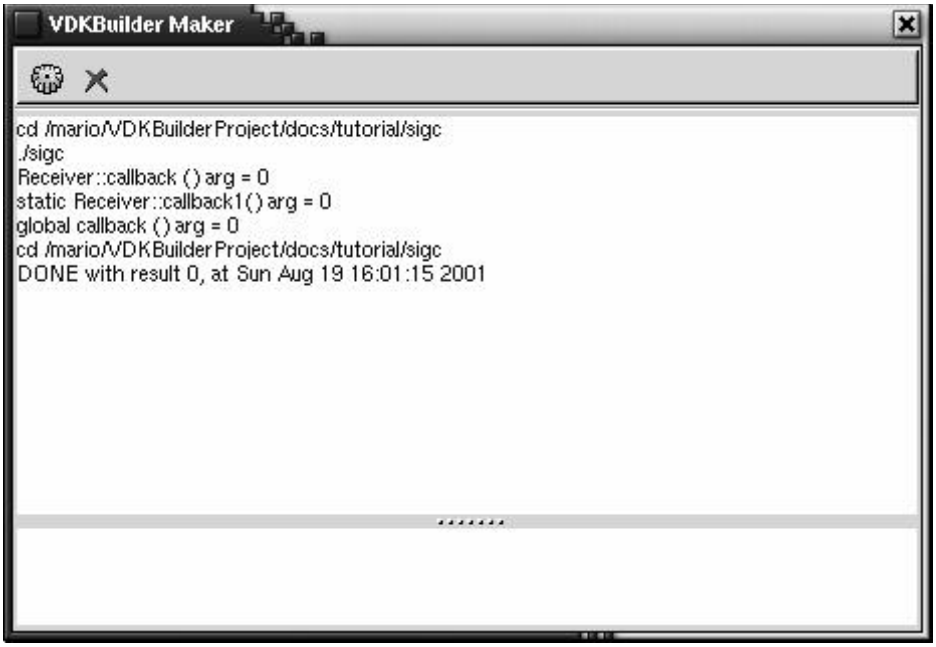
- Select File−>New−>Project menu and check “Console Application” radio button.
- Click on Next button
- Set the project name as “sigc”
- Now edit sigc.cc file to read:

```
/*
sigc ConsoleApplication
Main unit implementation file:sigc.cc
*/
#include <sigc.h>
#include <sigc++/signal_system.h>
#include <iostream>
using namespace SigC;
using namespace std;
class Receiver: public SigC::Object
{
  public:
  void callback (int x)
  { cout << "Receiver::callback () arg = " << x << endl;}
  static void callback1(int x)
  { cout << "static Receiver::callback1() arg = " << x << endl; }
};
void callback (int x)
{    cout << "global callback () arg = " << x << endl; }
/*
main program
*/
int main (int argc, char *argv[])
{
  // make the signal and receiver
  Signal1<void, int> sig1;
  Receiver r;
  // get connection to the global callback() function and to
  // Receiver static  callback1() function
  sig1.connect (slot (&callback));
  sig1.connect (slot (Receiver::callback1));
  // get connection to Receiver class callback() member function
  sig1.connect (slot (r, &Receiver::callback));
  // signal emission
  sig1.emit (0);
  return 0;
}
```

In order to compile and link correctly this simple example we have to tell builder to link with both libsigc++ and pthread library so:

- select Project->Options menu
- edit "Shared libs" field to read : `-lsigc -lpthread`
- click on "Close and save" button.

Build an run, builder maker should show you this output:



Note that callback functions are called in reverse order.

VDK signal extension to LibSigc++

LS signal exists in VDK without interfering with the other signal systems (static, dynamic tables or native gtk+), so you can use them concurrently. LS signals in VDK are handled like native LS signals altough they are not real LS signals. There are two reasons for this: first all VDK_LS signals have an hybrid signature and second in case of certain signals there is an extended API to handle dynamic disconnects better (more on this later on).

A VDK-LS signal of type `VDKSignal#<R,P1,...P#>` can be thought as :

- `VDKSignal#<R,P1,...P#>`

and

- `VDKSignal#<R,VDKObject*,P1,...P#>`

What explains his hybrid nature? This was done to accomodate to the signal pattern where called functions do not need to know the signal source, so our receivers aren't expected to have a parameter for the source. However there are cases where this knowledge is useful so both signatures are accepted to connect with. Obviously the emit-method for a VDKSignal only accepts the full signature. As a general rule all instantiated VDKSignals have a return type of void and begin with a prefix "On". For a list of available signals on VDK classes see the reference generated from sources with Doxygen.

A simple Hello program that uses Libsigc++ extension

We will show the code that is used in a simple hello program, a form containing just a button that when clicked displays a tooltip under mouse position.

- So create an "hello" project
- main form should contain just a vertical box with inside a button named helloBtn and labeled "_Hello" (our reader should be now able to do it without further explanations)

Here what should be the result:

I hope that the commented code will be of help, the project implementation is left as exercise to the reader. We show only hello.cc and hello.h, others: hello_gui.cc and hello_gui.h should be created by builder for you.

Hello.h

```
/*
hello Plain VDK Application
Main unit header file: hello.h
*/
#ifndef _hello_main_form_h_
#define _hello_main_form_h_
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif
// vdk support
#include <vdk/vdk.h>
// Hello FORM  CLASS
class HelloForm: public VDKForm
{
// gui object declarations
private:
// gui object declarations
  void GUISetup(void);
  void OnButtonClicked (VDKObject* sender); // answers to button click
  void ShowTipForm(char* tip); // show a tooltip under mouse
public:
  HelloForm(VDKApplication* app, char* title);
  ~HelloForm();
  void Setup(void);
/*
 gui setup include
 do not patch below here
*/
#include <hello_gui.h>
};
// Hello APPLICATION  CLASS
class HelloApp: public VDKApplication
{
public:
HelloApp(int* argc, char** argv);
~HelloApp();
void Setup(void);
};
/* a simple tooltip class, a form that contains:
– an eventbox (so it can have its own background)
– a label that contains the tip text
– a timer that starts when tip is constructed and auto closes it after
   2 secs.
We didn’t use builder to make this widget, as it is so simple that can
be easily written by hand.
*/
class TipForm: public VDKForm
{
  VDKLabel*  label;
  char* tip;
  VDKTimer* timer;
public:
  // constructing as window pop up (GTK_WINDOW_POPUP mode)
  // makes the form without frames nor decorations
  TipForm(VDKForm* owner, char* tip):
    VDKForm(owner,NULL,v_box,GTK_WINDOW_POPUP),tip(tip)
    {timer = NULL;}
 // ovverrides VDKForm::Close()
  void Close ();
  ~TipForm() {  }
  void Setup(void);
  };
#endif
// do not remove this mark: #!#
// end of file:hello.h
```

hello.cc

```
/*
hello Plain VDK Application
Main unit implementation file:hello.cc
*/
#include <hello.h>
/*
main program
*/
int main (int argc, char *argv[])
{
// makes application
HelloApp app(&argc, argv);
// runs application
app.Run();
return 0;
}
// Hello MAIN FORM  CLASS
/*
main form constructor
*/
HelloForm::HelloForm(VDKApplication* app, char* title):
      VDKForm(app,title)
{
}
/*
main form destructor
*/
HelloForm::~HelloForm()
{
}
/*
main form setup
*/
void
HelloForm::Setup(void)
{
  GUISetup();
  // here we establish a connection with button click
  helloBtn->
      OnButtonClicked.connect(slot(*this,&HelloForm::OnButtonClicked));
  }
/*
      answers to button click showing tooltip window
*/
void
HelloForm::OnButtonClicked (VDKObject* sender)
{
  ShowTipForm ("VDKBuilder tutorial - Hello by VDK Team\nhave fun");
}
/*
 shows tip window under mouse position
 */
void
HelloForm::ShowTipForm(char* tip)
{
  GdkModifierType mask;
  int win_x,  win_y, mouse_x, mouse_y;
  // access to underlying GdkWindow
  GdkWindow* win = Window()->window;
  // gets mouse coordinates relative to upper left corner of
  // the form
  gdk_window_get_pointer (win, &mouse_x, &mouse_y, &mask);
  // gets window coordinates relative to desktop
  gdk_window_get_deskrelative_origin(win,&win_x,&win_y);
  // makes and shows tip window
  TipForm* tip_form = new TipForm(this,tip);
  tip_form->Setup();
  tip_form->Position = VDKPoint(win_x + mouse_x, win_y + mouse_y);
  tip_form->Show();
}
```

```
// Hello APPLICATION  CLASS
/*
application constructor
*/
HelloApp::HelloApp(int* argc, char** argv):
      VDKApplication(argc,argv)
{
}
/*
application destructor
*/
HelloApp::~HelloApp()
{ }
/*
application setup
*/
void
HelloApp::Setup(void)
{
      MainForm = new HelloForm(this,NULL);
      MainForm->Setup();
      MainForm->Visible = true;
}
/*
        TIP FORM CLASS
 */
/*
      initializes tip window
*/
void
TipForm::Setup(void)
{
  // adds an event box
  VDKEventBox *vbox = new VDKEventBox(this,v_box);
  Add(vbox,0,1,1,0);
  // sets background
  vbox->NormalBackground = VDKRgb("ivory");
  label = new VDKLabel(this,tip);
  vbox->Add(label,0,1,1,0);
  label->Foreground = VDKRgb("indian red");
  label->Caption = tip;
  // makes a timer starting at construction
  // timer ticks are established each 2 secs.
  timer = new VDKTimer (this, 2000, true);
  // connects with timer “tick” signal
  timer->OnTimerTick.connect(slot(*this,&TipForm::Close));
}
/*
      ovverrides VDKForm::Close since needs to stop the timer
      before closing tip window
*/
void
TipForm::Close ()
{
  if (timer)
      timer->Stop ();
  VDKForm::Close ();
}
// do not remove this mark: #!#
// end of file:hello.cc
```

USING VDK SIGNAL EXTENSION WITH EVENTS

The above signals all dealt with changes of the objects state, now we inspect signals covering events. All signals dealing with events have the signature: `VDKSignal<void, VDKEventType>` with `VDKEventType` being the type the event is represented by.

Disconnecting

As events are mostly often relatively big and typically emitted several times there comes a new feature into play: these signals have an additional method: `Disconnect(&Connection)`. This will not only destroy visible connection but also destroy the underlying gdk−sigc++ connection.

Filters

Very often under a lot of different circumstances there is the need for some signals to have an additional parameter to the connect routine, this additional paramter is a filter for this event. E.g. When you are interested in button events you must specify the filter from one of the following filters:

- BE_RELEASE, a button was released
- BE_PRESS, a button was pressed
- BE_CLICK2, a double click occurred
- BE_CLICK3, a triple click occurred

Events

Following is a short description of the available objects covering events. Note that a single eventtype can be emitted by various eventsignals. This is just here to give you an overview, if you want to know more
about a certain event look in the reference.

- *VDKMouseEvent*, this covers nearly everything that can happen with your mouse, so it provides information about button presses and pointer position.
- *VDKKeyEvent*, covers the state of your keyboard, so you can ask for what key (if any) is pressed and what modifiers (like Ctrl,Meta) are active. You can ask for a representation as string or integer.
- *VDKMapEvent*, tells you what parts of the whole underlying widget go (un)covered, beside this there is no additional information.
- *VDKFocusEvent*, this is just an interface for events dealing with focus information. It can tell you if a widget has focus or not.
- *VDKKeyFocusEvent*, the widget has got (or lost) the keyboard focus
- *VDKMouseFocusEvent*, the widget has got (or lost) the mouse focus
- *VDKPaintEvent*, a certain area of the widgets needs to be repainted. The area is contained in this event.
- *VDKGeometryEvent*, indicates a change of the widgets geometry (like on resize)

Eventsignals

Now the question is: what signals are there to emit these eventobjects? In contrast to our normal signals which are defined in a certain subclass these signals are all defined in VDKObject class. This is because events only cover the aspect of a specific region on your display (of course they are originally generated by the X11 server).

- **VDKButtonSignal**
  Handles common mousebutton events. These events are emitted somewhat iterated so don't be confused.
  Instantiating member name: `OnButtonEvent`
  Emitted eventtype: `VDKMouseEvent`
  Filter names:
  - BE_RELEASE a button was released
  - BE_PRESS a button was pressed
  - BE_CLICK2 a double click occurred
  - BE_CLICK3 a triple click occurred

- **VDKKeySignal**
  Handles keyevents, is the only source for VDKKeyEvent
  Instantiating member name: `OnKeyEvent`
  Emitted eventtype: `VDKKeyEvent`
  Filter names:
  - KE_RELEASE a key was released

- KE_PRESS a key was pressed

- **VDKKeyFocusSignal**
  These signals cover a VDKKeyFocusEvent so it tells you about the keyboard focus state. It's instantiated member in VDKObject is called OnKeyFocus. You can use the specifiers declared in VDKFocusEventFilter.
  Filter names:
  - FE_IN object got focus
  - FE_OUT object lost focus

- **VDKPointerFocusSignal**
  Very similar to VDKKeyFocusSignal but this one deals with the pointer focus, it uses the same filter specifiers.
  Instantiating member name: `OnPointerFocusEvent`
  Emitted eventtype: `VDKPointerFocusEvent`

- **VDKPointerSignal**
  This one tells you about pointer movements. The movements are divided into two types: general movements, which are every motion or only modified movements. In the latter case these events only occur when a modifier (like a mouse button or Meta) was also pressed.
  Instantiating member name: `OnPointerEvent`
  Emitted eventtype: `VDKMouseEvent`
  The following filters are vailable and declared in VDKPointerEventFilter:
  - PE_ALL, report all
  - PE_PRESSED, report modified only

- **VDKMapSignal**
  Reported when your widget was covered or uncovered. You can specify if you are interested in events where the widget got completely visible, completely invisible or partially hidden. You do this with the help of *VDKVisibilityState*. This enum is also used by the emitted event itself to indicate its state.
  Instantiating member name: `OnMapEvent`
  Emitted eventtype: `VDKMapEvent`
  The following filters are available and declared in VDKPointerEventFilter:
  - FULLY_VISIBLE
  - PARTIALLY_VISIBLE
  - NOT_VISIBLE

- **VDKGeometrySignal**
  This signal represented by OnGeometryEvent and tells you that the geometry was changed by delivering you a *VDKGeometryEvent*. There are no filters available.

- **VDKPaintSignal**
  When you connect to OnPaintEvent a VDKPaintEvent will be delivered you each time the widget got exposed (that means you have to redraw some area). There are no filters available

- **VDKRawSignal**
  This signal is a very special one. When you connect to the representing member named `OnRawEvent` you will get a plain GdkEvent every time the underlying widget receives any event. The signature is of the type is `VDKSignal<void, const GdkEvent*>`. Normally you shouldn't use this one but instead one of the nicer VDKEvent−typed ones.

As usual we will make a simple program to demonstrate how to use eventsignals with libsigc++ extension.

– Make a new project naming it "events"
– Insert into main form a vbox
– drop into vbox a VDKCanvas naming it <canvas>, we will use it to show how handle all events.
– Set canvas background to white (or other light color as you prefere)

Here the complete and commented events.cc and events.h code:
events.h

```
*/
events Plain VDK Application
Main unit header file: events.h
*/
#ifndef _events_main_form_h_
#define _events_main_form_h_
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif
// vdk support
#include <vdk/vdk.h>
// Events FORM  CLASS
class EventsForm: public VDKForm
{
// gui object declarations
private:
// gui object declarations
  void GUISetup(void);
  void CanvasWrite (char* s,  int x,  int y);
  void HandleMouseEvents(VDKObject* obj,const VDKMouseEvent& eve);
  void HandleButtonPress(VDKObject* obj,const VDKMouseEvent& event);
  void HandleDblClick(VDKObject* obj,const VDKMouseEvent& event);
  void HandleKeyEvents (VDKObject*, const VDKKeyEvent& event);
public:
EventsForm(VDKApplication* app, char* title);
~EventsForm();
void Setup(void);
/*
 gui setup include
 do not patch below here
*/
#include <events_gui.h>
};

// Events APPLICATION  CLASS
class EventsApp: public VDKApplication
{
public:
EventsApp(int* argc, char** argv);
~EventsApp();
void Setup(void);
};
#endif
// do not remove this mark: #!#
// end of file:events.h
```

events.cc

```
/*
events Plain VDK Application
Main unit implementation file:events.cc
*/
#include <events.h>
static char buff[256];
/*
main program
*/
int main (int argc, char *argv[])
{
// makes application
EventsApp app(&argc, argv);
// runs application
app.Run();
return 0;
}
// Events MAIN FORM  CLASS
/*
main form constructor
*/
EventsForm::EventsForm(VDKApplication* app, char* title):
VDKForm(app,title)
{
}
/*
main form destructor
*/
EventsForm::~EventsForm()
{
}
/*
main form setup
*/
void
EventsForm::Setup(void)
{
GUISetup();
// put your code below here
}
// Events APPLICATION  CLASS
/*
application constructor
*/
EventsApp::EventsApp(int* argc, char** argv):
VDKApplication(argc,argv)
{
}
/*
application destructor
*/
EventsApp::~EventsApp()
{ }
/*
application setup
*/
void
EventsApp::Setup(void)
{
MainForm = new EventsForm(this,NULL);
MainForm->Setup();
MainForm->Visible = true;
}
```

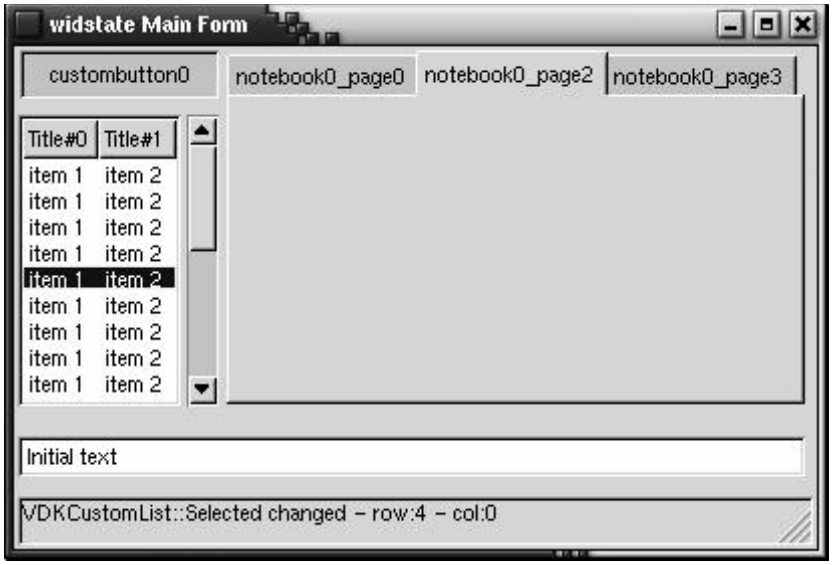
```
// do not remove this mark: #!#
/*

*/
void
EventsForm::CanvasWrite (char* s, int x,  int y)
{
  canvas->Clear ();
  canvas->DrawString (x, y, s);
  canvas->Redraw ();
}
//signal response method
bool
EventsForm::OncanvasRealize(VDKObject* sender)
{
  // set background
  canvas->NormalBackground = VDKRgb ("ivory");
  canvas->Foreground = VDKRgb ("indian red");
  canvas->Clear ();
  VDKFont* font = canvas->Font;
  if (!font && ((font = new VDKFont (this, "courier Bold 12"))))
      canvas->Font = font;
  canvas->Redraw ();
  // connect with mouse events
  canvas->OnButtonEvent.connect (
    slot (*this, &EventsForm::HandleButtonPress),BE_PRESS);
  canvas->OnButtonEvent.connect (
    slot (*this, &EventsForm::HandleDblClick),BE_CLICK2);
  canvas->OnPointerEvent.connect(
    slot(*this, &EventsForm::HandleMouseEvents),PE_ALL);
  // connect with keyboard events
  OnKeyEvent.connect(
    slot(*this, &EventsForm::HandleKeyEvents),KE_RELEASE);
 return true;
 }

void
EventsForm::HandleKeyEvents (VDKObject*,
        const VDKKeyEvent& event)
{
  sprintf (buff, "\non key:%d", event.Key ());
  CanvasWrite (buff, 10,10 );
  }

void
EventsForm::HandleDblClick(VDKObject* obj,
          const VDKMouseEvent& event)
{
  sprintf (buff, "Double click ()");
  CanvasWrite (buff, 10,10 );
}

void
EventsForm::HandleButtonPress(VDKObject* obj,
          const VDKMouseEvent& event)
{
  sprintf (buff, "Button press:%d", event.Button ());
  CanvasWrite (buff, event.Position ().x,event.Position ().y );
}

void
EventsForm::HandleMouseEvents(VDKObject* obj,
          const VDKMouseEvent& event)
{
  VDKFont* font = canvas->Font;
  sprintf (buff, "%d,%d",  event.Position ().x, event.Position().y);
  CanvasWrite (buff,
    event.Position ().x - (font ? font->Width (buff): 0),
    event.Position ().y);

}
// end of file:events.cc
```

As usual here what should be the final result:

A new paradigm as consequence of LibSigc++ signal extension.
Tim Lorenz (VDK signal extension author) did a very good job, but his work become even better when he made a relation between signals and VDK properties. Each property emits a `OnValueChanged()` signal whenever its state changes. This give us a new paradigm based on widget state change rather than a signal emission. We are interested only in widgets state changes, if any. Let's make a simple example with the builder:

- Make a new project naming it "widstate"
- Drop into main form a statusbar
- Drop into main form a fixed container adding it:
  - a VDKCustomButton choosing a toggled one
  - a VDKNotebook with 3 pages
  - a VDKCustomList (2 rows)
  - a VDKEntry

As usual below what should be the result:



In the following code (widstate.cc and widstate.h) we demonstrate how use this brand new paradigm, all the important stuff is commented into WidstateForm::Setup().

widstate.h

```
/*
widstate Plain VDK Application
Main unit header file: widstate.h
*/
#ifndef _widstate_main_form_h_
#define _widstate_main_form_h_
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif
// vdk support
#include <vdk/vdk.h>
// Widstate FORM  CLASS
class WidstateForm: public VDKForm
{

private:
  void GUISetup(void);
  void OnToggledButtonCheckedPropertyChange (bool checked);
  void OnNotebookActivePagePropertyChange (int page);
  void OnCustomListSelectedPropertyChange (VDKPoint p);
  void OnEntryTextPropertyChange (char* text);
  void OnCustomListSelectedTitlePropertyChange (int column);

public:
WidstateForm(VDKApplication* app, char* title);
~WidstateForm();
void Setup(void);
/*
 gui setup include
 do not patch below here
*/
#include <widstate_gui.h>
};

// Widstate APPLICATION  CLASS
class WidstateApp: public VDKApplication
{
public:
WidstateApp(int* argc, char** argv);
~WidstateApp();
void Setup(void);
};
#endif
// do not remove this mark: #!#
// end of file:widstate.h
```

widstate.cc

```
/*
widstate Plain VDK Application
Main unit implementation file:widstate.cc
*/
#include <widstate.h>
static char buff[256];
/*
main program
*/
int main (int argc, char *argv[])
{
// makes application
WidstateApp app(&argc, argv);
// runs application
app.Run();
return 0;
}
// Widstate MAIN FORM  CLASS
/*
main form constructor
*/
WidstateForm::WidstateForm(VDKApplication* app, char* title):
VDKForm(app,title)
{
}
/*
main form destructor
*/
WidstateForm::~WidstateForm()
{
}
/*
main form setup
*/
void
WidstateForm::Setup(void)
{
GUISetup();
// put your code below here
// loads some items on vdkcustomlist
char* items[] = { "item 1",  "item 2"};
for (int t = 0; t < 20 ; t++)
  customlist0->AddRow (items);
// selects first item on list
  customlist0->SelectRow (0, 0);
// set initial text to entry
entry0->Text = "Initial text";
// connect with widget state changes via properties
// reacts to changes on VDKCustomButton::Checked property
custombutton0->Checked.OnValueChanged.connect(
  slot(*this, &WidstateForm::OnToggledButtonCheckedPropertyChange));
// reacts to changes on VDKCustomList::Selected property
notebook0->ActivePage.OnValueChanged.connect(
  slot(*this, &WidstateForm::OnNotebookActivePagePropertyChange));
// reacts to changes on VDKCustomList::SelectedTitle property
customlist0->SelectedTitle.OnValueChanged.connect(
  slot(*this, &WidstateForm::OnCustomListSelectedTitlePropertyChange));
// reacts to changes on VDKCustomList::Selected property
customlist0->Selected.OnValueChanged.connect (
  slot (*this, &WidstateForm::OnCustomListSelectedPropertyChange));
// reacts to changes on VDKEntry::Text property
entry0->Text.OnValueChanged.connect (
  slot (*this, &WidstateForm::OnEntryTextPropertyChange));
}
/*
*/
void
WidstateForm::OnCustomListSelectedTitlePropertyChange (int column)
{
    sprintf (buff,"VDKCustomList::SelectedTitle changed: %d",column);
    statusbar0->Push (buff);
```

```
  }
/*
*/
void
WidstateForm::OnEntryTextPropertyChange (char* text)
{
    sprintf (buff,"VDKEntry::Text changed: \"%s\"",text);
    statusbar0->Push (buff);

  }
/*
*/
void
WidstateForm::OnCustomListSelectedPropertyChange (VDKPoint p)
{
    sprintf (buff,"VDKCustomList::Selected changed - row:%d - col:%d",
            p.x, p.y);
    statusbar0->Push (buff);
}
/*
*/
void
WidstateForm::OnNotebookActivePagePropertyChange (int page)
{
    sprintf (buff,"VDKNotebook:ActivePage changed: %d", page);
    statusbar0->Push (buff);
}
/*
*/
void
WidstateForm::OnToggledButtonCheckedPropertyChange (bool checked)
{

    sprintf (buff,"VDKCustomButton::Checked change:%s",
            checked ? "checked" : "uncheck");
    statusbar0->Push (buff);
  }
// Widstate APPLICATION  CLASS
/*
application constructor
*/
WidstateApp::WidstateApp(int* argc, char** argv):
VDKApplication(argc,argv)
{
}
/*
application destructor
*/
WidstateApp::~WidstateApp()
{ }
/*
application setup
*/
void
WidstateApp::Setup(void)
{
MainForm = new WidstateForm(this,NULL);
MainForm->Setup();
MainForm->Visible = true;
}
// do not remove this mark: #!#
// end of file:widstate.cc
```

APPENDIX A
SIGNALS AND EVENTS

GENERAL

There are two strategies available to manage signals and events. The programmer can choose to implement either or both. The first one, used by VDK itself, must be established at compile−time and uses static signal tables. The second one uses dynamic signal tables and allows the programmer to connect and disconnect both signals and events at run−time and is very similar to that which Gtk+ uses. Here some considerations about the differences between the two methods:

- Signal management with static tables in VDK is quite different than Gtk+ since all signals are connected by default to all widgets supported so they will be emitted anyway. What you can control is answering signals, this "broadcast" strategy makes things much easier avoiding connect/disconnect stuff and makes a clear separation between interface and application, application does not care about managing signals but only to answer those signals it is interested in. However there remains a great flexibility to change or customize this behaviour like will be explained later.
- Signal management with dynamic tables is very similar to Gtk+, user has full control of connecting/disconnecting signals/events.
- Static tables manage signal/events connected by VDK at compile−time whereas dynamic tables manage signals connected by the programmer at run−time. Thus if the programmer wants to make use of signals or events already provided with VDKObjects then static tables must be used. If the programmer wants to connect more signals/events at execution time then he or she should use dynamic ones.
- Both strategies can be used concurrently.

SIGNALS AND EVENTS WITH STATIC TABLES

Signals and events are connected with class methods using two different tables: SIGNAL_MAP and EVENT_MAP. With this strategy callbacks are converted into form class methods that can directly answer to widgets signals and/or events.

To construct signal map user needs to make use of these macros:

- DECLARE_SIGNAL_MAP(class) inside form class declaration, this macro declares a general response function and the response table.
- DEFINE_SIGNAL_MAP(class,ancestor) outside class declaration, this macro defines a general response function and initializes response table.
- ON_SIGNAL(object, signal, response function), actually constructs table entries that bind <object> with <signal> and <response function>, this will be called when the object is interested in the <signal>. Response function will receive one arg: VDKObject* <sender>, the VDKObject that generated the signal. Signal types enum is declared in <vdktypes.h>.
- END_SIGNAL_MAP Closes and finishes the table.

To construct the event map the programmer must use the following macros:

- DECLARE_EVENT_MAP(class) inside form class declaration, this macro declares a general response function and the response table.
- DEFINE_EVENT_MAP(class,ancestor) outside class declaration, this macro defines a general response function and initialize response table.
- ON_EVENT (object, event, response function), actually constructs table entries that bind <object> with <event> and <response function>, this will be called when the object is interested in the <event>. Response function will receive two args: VDKObject* <sender> and GdkEvent* <event>. Event types enum is declared in <vdktypes.h>.
- END_EVENT_MAP Closes and finishes the table.

RESPONSE METHODS (WERE CALLBACKS IN GTK+)

- Response member functions named into signal table can be virtual
- Methods answering to a signal have this signature:

```
bool SomeFormClass::AnswerMethod(VDKObject* sender);
```

  so they receive the address of signal generating object.

- Method answering to events have this signature:

```
bool SomeFormClass:AnswerEvent(VDKObject* sender, GdkEvent* event);
```

  (Hint: make a cast from GdkEvent event to GdkEventXxxx* to use it effectively:

```
bool SomeFormClass::OnMouseMotion(VDKObject* sender, GdkEvent* ev)
{
GdkEventMotion* event = (GdkEventXxxx*) ev;
// .. use event
}
```

- Signals/events are considered **"treated"** if and only if the user−defined answering method returns boolean true, in this case any further signal/event propagation will be stopped.

MULTIPLE RESPONSE METHODS FOR SIGNAL AND EVENTS

It is possible to establish more than one response method for a single signal on a single object by just adding a line to signal table with ON_SIGNAL or ON_EVENT macro. Response method will be called in the same order they appear in signal/event table. Each single response method will be considered independently from previous ones, thus will be called even if the previous has already been "treated".

DETACHING SIGNALS/EVENTS

Some VDKObject member functions are provided to manage signal/event connecting/disconnecting:

- **int SignalDetach(object, signal)**
  Disconnect signal <signal> from <object>, the response method previously declared into SIGNAL_MAP will stop answering to <signal>. Method returns the signal "slot" number that was deactivated and that should be used in order to reconnect the signal later. Answering −1 means that the signal was not detached. Multiple response method will be detached in the same order they appear on signal table.

- **bool SignalAttach(int slot)**
  Reactivate the signal into slot <slot>. This number should be previously achieved with a SIGNAL_DETACH() call on a signal.

- **int EventDetach(object , event)**
  Same as SignalDetach()

- **bool EventAttach(int slot)**
  Same as SignalAttach()

  The following code:

```
static int slots[X+1];
// where X is the number of response methods on that signal/object
void MyForm::DetachAllButtonSignals()
{
int slot,t;
for( t= 0;(slot = SignalDetach(button ,clicked_signal)) != −1; t++)
      slots[t] = slot;
slot[t] = −1;
}
```

will detach all clicked signals on <button> and store slot numbers into slots[] array for a later use.
These functions are provided in the aim to maintain Gtk+ similarity but, as will be explained later, the broadcasting and hierarchy strategy that VDK adopts in dispatching signals should make it be rare that you need the above functions.

SIGNALS DISPATCHING

Signals are emitted by objects in a recursive "pattern visiting" dispatching algorithm. A first recursion is made into the object hierarchy (so called class level) , another is made into object parent hierarchy (so called parent level). Signal flow will be stopped as soon as signal is flagged as "treated", that is the dispatching algorithm finds a defined signal response table entry (so called slot) for that signal on that object that answered "**true**".

Here a typical signal path:

1. Signal emitted by an object
2. Signal goes to object class , if "treated" it stops, otherwise recursively goes up to object hierarchy until "treated" or it reaches the object root class.
3. If not yet "treated" and object has a parent goes to parent class, if "treated" it stops otherwise goes up to parent hierarchy until "treated" or it reaches parent root class. If parent has a parent step 3 is repeated until no parent is found.
   Tip: in parent hierarchy there is only one object that surely doesn't have a parent, this is the application's main form..
4. If not yet "treated" signal is lost.

Above and below considerations apply to events as well.

This mechanism permits a wide broadcasting strategy and allows to inherit response tables among objects hierarchy.
Signals can be emitted in two ways:

- autonomously by Gtk+ low level widgets.
- by user with **VDKObject::SignalEmit(signal)** class method. In both cases the signal flows like above.
  Events cannot be user emitted.

OWNERSHIP , PARENTHOOD AND SIGNAL FLOW LEVELS

The concepts explained below are intended mainly for signal flow only and should not be confused with same concepts regarding object organization and management specially in Gtk+ side.

Ownership

All objects have an owner, the owner is always a form where the object is contained in, owner takes care of objects displacement and memory freeing, does not interest signal flow.

Parenthood

A parent is an object, usually a form, a box, a table, a frame or others container but generally speaking it can be any object. Parent groups objects into a logical unit with respect to signal flow. Normally objects have a parent (owner form is their outmost parent by default), but user can set the object parenthood using **VDKObject::Parent(parent)** class method. Adding any object to a container sets by default the container as object parent so the user doesn't have to call Parent() explicitly.

Class level

This is the first level a signal goes through. In VDK stock hierarchy no class answers to signals at this level. The purpose is to allow the programmer to derive his or her own objects and to let them answer "internally" to signals before emitting the same or another signal to upper levels. In this case macros for established static response tables are slightly different:

- DECLARE_OBJECT_SIGNAL_MAP(class) inside object class declaration, this macro declare a general response function and the response table.
- DEFINE_OBJECT_SIGNAL_MAP(class,ancestor) outside object class declaration, this macro defines a general response function and initialize response table.
- ON_OBJECT_SIGNAL(signal, response function), actually constructs table entries that bind object with <signal> and <response function> that will be called when the object is interested in the <signal>. Response function will receive one arg: VDKObject* <sender>, thus the VDKObject that

generated the signal. Signal types enum is declared in <vdktypes.h>.

- END_OBJECT_SIGNAL_MAP Closes and finishes the table.
- The object class method answering to signal has this form:
  bool *SomeObjectClass::AnswerMethod(VDKObject* sender);*

Emitting a signal from a derived object can be made in two ways:

- Answering **false** in response method, this makes the signal proceed up the object hierarchy and then to the parent level.
- Using **SignalEmitParent (signal)** this forces the signal to skip class level and go directly to parent.
- There is a third possibility: calling Ancestor::ResponseMethod(sender) if it is known base class has it.

Parent level

If the signal is not responded to at the class level it goes to the next object parent and recursively into the parent hierarchy until it reaches the parent root class. This process is repeated until the outer most parent is reached. If the signal is not responded to, it can therefore reach the application main form that surely is "the mother af all parents".

Parent level is most used when you want to make a composite object that manages several "inner" objects catching their signal and sending a unified or modified answer to its parent. This gives the ability to make components completely abstracted from the application that is using them. The application can only answer to a signal coming from one of it's components and knows nothing about component internals.

PROVIDING FLEXIBILITY

The above topics discusses the signal dispatch system flexibility. Here is an example. Imagine you want to derive a specialized class of VDKEntry widget that intends to do some checks on end−user input before it forwards the "activate" signal to the parent. In this case we should define a:

```
ON_OBJECT_SIGNAL(activate_signal, HandleActivateSignal)
```

in response table and write in the handler something like this:

```
bool MyVDKEntry::HandleActivatedSignal(MyVDKEntry* sender)
{
if(some_checks_on_entry_are_ok())
      return false;
// signal proceeds to object ancestor (no treated) and then to parent.
else
      {
      prompts_to_the_user() ||
      makes_correcting_actions();
      }
return true; // signal is treated, stops here.
}
```

This tecnique allows you to derive a new object from VDK stock hierarchy implementing new features and mantaining inheritance of signal tables as well.

Another possibility is to directly address activate_signal to a parent level using **SignalEmitParent (signal).**

```
bool MyVDKEntry::HandleActivateSignal(MyVDKEntry* sender)
{
if(some_checks_on_entry_are_ok())
      SignalEmitParent (activated_signal);
else
      {
      prompts_to_the_user() ||
      makes_correcting_actions();
```

```
    }
return true; // signal is treated, stops here.
}
```

USER DEFINED SIGNAL

The programmer can define his or her own signals and dispatch them using **SignalEmit(signal)** or **SignalEmitParent (signal).** I strongly suggest to define your signal using:

```
#define MYSIGNAL user_signal + <somevalue>
```

to avoid possible conflicts with already defined signals.

COMPOSITE OBJECTS (COMPONENTS)

Signal dispatching lets the user construct composite objects that can manage signals of their inner objects and send their own signals to the outer world.
Let's assume an object made of 3 buttons packed into a box:

```
#define MYSIGNAL (user_signal+1)
class MyComposite: public VDKBox
{
protected:
 VDKCustomButton * button1;
 VDKCustomButton * button2;
public:
 VDKCustomButton * button0;  // could be a property
 int ButtonPressed;
 MyComposite(VDKForm * owner, int mode = v_box );
 ~MyComposite();
 bool SignalCallback(VDKObject *);
 int button;DECLARE_SIGNAL_MAP(MyComposite);
};

DEFINE_SIGNAL_MAP(MyComposite,VDKBox)
 ON_SIGNAL(button0,clicked_signal,SignalCallback),
 ON_SIGNAL(button1,clicked_signal,SignalCallback),
 ON_SIGNAL(button2,clicked_signal,SignalCallback)
END_SIGNAL_MAP

bool MyComposite::SignalCallback(VDKObject * sender)
{
if(sender == button0)
 {
 ButtonPressed = 0;
 return false;
 }
else if(sender == button1)
 ButtonPressed = 1;
else if(sender == button2)
 ButtonPressed = 2;
SignalEmitParent(MYSIGNAL);
return true;
}
```

SignalCallback() catches buttons signals and send an unified answer emitting his own MYSIGNAL to parent, since SignalCallback() return true button signals are trapped into MyComposite scope. (except <button0> whose signal isn't trapped, the reason will be explained soon)

ANSWERING SIGNAL AT ANY LEVEL OF PARENTHOOD

It is possible (even if not very useful in a OO point of view) to answer to a signal coming from some lower level. Following the above example you can answer at form level that contains MyComposite object to signals coming from <button0> provided that you can know address of the object whose signals you want to answer (that's the reason why <button0> was declared as public member). This piece of code shows how you can do it:

```
class MyForm: public VDKForm
{
protected:
      MyComposite *composite;
      VDKCustomButton *label_button;
public:
      MyForm(/*?.*/):VDKForm(/*?*/) {}
      /?
      void Setup()
      {
      composite = new MyComposite(/*?.*/);
      // here you store button0 address
      label_button = composite->button0;
      }
DECLARE_SIGNAL_MAP(MyForm);
};

DEFINE_SIGNAL_MAP(MyForm, VDKForm)
      ON_SIGNAL(label_button,clicked_signal, MyCallback)
END_SIGNAL_MAP

bool MyForm::MyCallback(VDKObject*)  { /*?..*/ return true; }
```

Since MyComposite::SignalCallback() answers false if sender is button0, the signal flows up the parent hierarchy and will be answered by signal table at MyForm level, so MyForm::MyCallback() will be called.

SIGNAL AND EVENTS WITH DYNAMIC TABLES

SIGNALS

To connect/disconnect with signal user must write these macros:

- DECLARE_SIGNAL_LIST(class) inside form class declaration, this macro declares a general response function and the dynamic response table.
- DEFINE_SIGNAL_LIST(class,ancestor) outside class declaration, this macro defines a general response function and initializes dynamic response table.

After that, the user makes use of these class methods:

- **int SignalConnect(VDKObject* object, char* signal,**
  bool (_owner_class::*Pmf)(VDKObject* sender) , bool gtk = true, bool after = false)

  Connect <object> of a method of <this> through <signal>. Return an integer that is the connected "slot" number.
  Usage: int slot = SignalConnect(button,"clicked",&MyForm::ButtonClicked)
  Tip: use this to connect a object contained in a form or another container with a container method.
  Tip: signal names are those provided by Gtk+
- **int SignalConnect(char* signal,**
  bool (_owner_class::*Pmf)(VDKObject* sender) , bool gtk = true, bool after = false)

  Connect <this> of a method of <this> through <signal>. Return an integer that is the connected "slot" number.
  Usage: int slot = SignalConnect("clicked",&MyButton::ButtonClicked);
  Tip: use this to connect an object with an object method.
- **bool SignalDisconnect(int slot)**
  Disconnect signal, <slot> must be retrieved from a SignalConnect () call.

TIP

- Argument <bool gtk> is a little useful trick that allow you to define and use your own signals without registering them.

If you set <gtk> to false no real connection will be made with Gtk+, will be handled if explicitely generated by a SignalEmit() call..

- Argument <after> if true connect signal with the after slot and is meaningful only if previous <gtk> arg is set to true.

Here is an example:

```
class MyForm: public VDKForm
{
VDKSomeObject *someobject;
//?
DECLARE_SIGNAL_LIST(MyForm);
};
DEFINE_SIGNAL_LIST(MyForm,VDKForm);

void MyForm::Setup()
{
Add(someobject  =  new VDKSomeObject(this));
//? Gtk+ would refuse to connect this since does not know "MySignal"..
// but gtk=false will override Gtk+ connecting stuff
SignalConnect(someobject,"MySignal",&MyForm::HandleMySignal,false);
// HandleMySignal will be called.
someobject->SignalEmit("MySignal");
}
```

EVENTS

To connect/disconnect with events user must use these macros:

- DECLARE_EVENT_LIST(class) inside form class declaration, this macro declares a general response function and the dynamic response table.
- DEFINE_EVENT_LIST(class,ancestor) outside class declaration, this macro defines a general response function and initializes dynamic response table.

After that the programmer makes use of these class methods:

- **int EventConnect(VDKObject* object, char* event,**
  bool (_owner_class::*Pmf)(VDKObject* sender, GdkEvent*), bool after = false))
  Connect <object> of a method of <this> through <event>. Return an integer that is the connected "slot" number.
  Usage: int slot = EventConnect(button,"button_press_event",&MyForm::ButtonPressed)
  Tip: use this to connect a object contained into a form or another container with a container method.
  Tip: event names are those provided by Gtk+
- **int EventConnect(char* event,**
  bool (_owner_class::*Pmf)(VDKObject* sender, GdkEvent*), bool after = false))
  Connect <this> of a method of <this> through <event>. Return an integer that is the connected "slot" number.
  Usage: int slot = EventConnect("button_press−event",&MyButton::ButtonPressed);
  Tip: use this to connect an object with an object method.
- **bool EventDisconnect(int slot)**
  Disconnect event, <slot> must be retrieved from a EventConnect () call.

RESPONSE METHODS (WERE CALLBACKS IN GTK+)

Same rules as static tables apply.

EMITTING SIGNALS

To emit signal using dynamic tables the programmer uses:
void SignalEmit(char* sig);
void SignalEmitParent(char* sig);
The same rules as static tables apply.

MULTIPLE RESPONSE METHODS FOR SIGNALS AND EVENTS

It is possible to establish more than one response method for a single signal/event on a single object by adding more SignalConnect() or EventConnect() calls. Response method will be called in the same order as connecting calls. Each single response method will be considered separately from previous ones, thus each will be called even if the previous is already been responded to.

SIGNALS DISPATCHING, OWNERSHIP , PARENTHOOD AND SIGNAL FLOW LEVELS

The same rules as static tables apply.

EXTENDED SYGNAL SYSTEM

Even if VDK sygnal system covers most of gtk+ signals and provides great flexibility it can handle only gtk+ signals whose callbacks have this signature: **void callback(GtkWidget*, gpointer)**. These types of signals are what most of the gtk+ signals are and it should be a rare case when the user needs to connect with others, however the extended signal system covers these few cases, also.

APPENDIX B

ABOUT PROPERTIES

Properties behave the same as in other modern RAD tools. You can set/get properties to obtain different behaviours from objects.

As an example let Visible be a form or object property, writing:

```
form->Visible = false; // hides form
form->Visible = true;  // shows form again.
```

It's also possible to use properties to evaluate an expression:

```
if(form->Visible)
      form->Visible = false;
else
      form->Visible = true;
```

or better:

```
      form->Visible = form->Visible ? false : true;
```

toggles form visibility.

Properties are implemented as templates on <class T> and have following method available:

- **virtual operator T()**
  Converts property to the property type.

- **void operator=(T value)**
  Assign property value.

- **VDKString Name()**
  Return property name.

Refer to each class for a better explanation how to use properties.

Properties have some limitations:

- cannot be copied/assigned, `form->Visible = form1->Visible` will be flagged as compilation error. If you want to copy a property to another object do this:
  ```
  bool visible = form->Visible;
  form1->Visible = visible;
  ```
  or
  ```
  form->Visible = bool(form1->Visible);
  ```
- cannot be used as args for those functions that not allow compiler to make an implicit conversion, for instance: `printf("\nvisible: %d",form->Visible);` will be flagged as error. In such case you must use an explicit cast: `printf("\nvisible: %d", (bool) form->Visible);`
- some properties are run−time read only, refer to "using properties" sections for more informations.

USER DEFINED PROPERTIES

VDK allows the programmer to define his or her own properties that can be used with forms and objects. In order to use properties effectively some internal things should be known.

- **PROPERTY DECLARATION**

  Properties are declared as template classes on owner class and type:

  VDKReadWriteValueProp< ownerClass, propertyType> where ownerClass is the class that owns the property and propertyType is the parameter type. For instance the **Visible** property of VDKForm class is declared into public part of VDKForm as:

```
VDKReadWriteValueProp<VDKForm,bool> Visible;
```

- **PROPERTY INSTANTIATION**

  A property must be instantiated in owner class constructor where we must pass to property constructor following argument:

  - property name
  - owner object
  - default value
  - <owner class member function> class member that is responsible to set the property value
  - <owner class member function> class member that is responsible to get property value.

  For instance **VDKForm::Visible** property is instantiated like this:

```
Visible("Visible",this,true,
                    &VDKForm::SetVisible,&VDKForm::GetVisible)
```

  Where

  - property name = "Visible" (is likely a convention to use the var name as property name)
  - owner object = this (obviously)
  - default value = true
  - set function = VDKForm::SetVisible()
  - get function = VDKForm::GetVisible()
  -

  In most cases the "getting" function is not necessary since a raw read of property value is sufficient, in other few cases also the "setting" is redundant, for this reason the property constructor is declared like this:

```
VDKReadWriteValueProp( char* name, T* object, S defValue,
   void (T::*write)(S) = NULL, S (T::*read)(void) = NULL)
```

  If setting/getting functions are set to NULL no setting/getting function will be called. In such cases writing: **property = something;** or **something = property;** leads into a raw write/read of property internal value. Otherwise if setting/getting function are set to a valid <owner class member function> besides the raw value write/read a function call will be made.

- **WRITING PROPERTY GETTING AND SETTING FUNCTIONS**

  - Property setting function must be declared as owner class member functions and with this signature:

    `void SetPropertyFunction(propertyType value);`

    Thus VDKForm::Visible property setting function is declared and defined like this:

```
void SetVisible(bool flag)
    {
        if(flag) Show();
        else Hide();
       }
```

    As you can see writing **myform−>Visible = true;** turns out in calling myform−>SetVisible(true) and than myform−>Show();

  - Property getting function must be declared as owner class member functions and with this signature:

    propertyType GetPropertyFunction(void);

    Thus VDKForm::Visible property getting function is declared and defined like this:

```
bool GetVisible() { return GTK_WIDGET_VISIBLE(window); };
```

    As you can see writing **bool visible = myform−>Visible;** turns out in calling myform−>GetVisible().

- **AN EXAMPLE OF USER DEFINED PROPERTY**

  There is also a convenience macro that helps to declare properties:

  – **__rwproperty(ownerClass, propertyType)** // declare a read/write property

  Let's show an example, assume an user form class like this:

```
class MyForm: public VDKForm
{
   public:
   _rwproperty(MyForm, int) MyProperty;
   MyForm(VDKApplication* app, gchar* title = "", int mode = v_box,
         GtkWindowType display = GTK_WINDOW_TOPLEVEL);
   ~MyForm() {}
   void SetMyProperty(int value)  { //?.. }
   int GetMyProperty() { //?.}
 };
```

```
// MyProperty initialization will be done into MyForm constructor
// like this:
MyForm ::MyForm(VDKApplication* app,
      gchar* title, int mode,GtkWindowType display ):
      VDKForm(app, title,mode,display),
      MyProperty("MyProperty",this,0,
      &MyForm::SetMyProperty,&MyForm::GetMyProperty)
      { // ..}
      That's all.
```

- **READ–ONLY PROPERTIES**

  Read only property is a subclass of write/read property with a restricted interface, any attempt to write a read–only property will be flagged as compilation error. A read only property is declared as:

  VDKReadOnlyValueProp< ownerClass, propertyType >

  Property constructor is declared as:
  VDKReadOnlyValueProp(char* name,T* object, S defValue,
  S (T::*read)(void) = NULL, (T::*write)(S) = NULL);

  where <write> argument will be always NULL.

  A convenience macro also exists:
  __rproperty(ownerClass, propertyType) to declare a read–only property.

  Property initialization is same as read/write properties recalling that <write> argument should be left always to NULL.

- **SETTING PROPERTY VALUES AVOIDING "SETTING" FUNCTION**

  It may sometimes be useful to directly set a property value without calling the setting function even if present, in this case you can use the "function" operator of a property.

  See this example:

```
myproperty = something; // set property value calling setting
function

myproperty(something);  // set property value without calling
setting function.
```

  Remember that last construct is a violation of "data hiding" concept and should be used with caution. To prevent potential errors this direct setting of a property should leave the property in the same state that a normal call would have.

Properties are convenient for several reasons:

- Encapsulate many details of low level operations, for instance setting the fore/back of an objects could be a long and detailed task, for the user is an easy job instead: **myform->Foreground = clNavyBlue;**
- Enhance code reusability and readability.
- Properties can be managed in a container such as lists or arrays, this is a first step toward construction of a property browser and thereafter a visual development environment.
- Properties can be applied to any class.

VDK library uses extensively the property concept and therefore its use is encouraged even if raw function call like a `myform->SetVisible()` is possible.

APPENDIX C
AUTOCONF/AUTOMAKE SUPPORT TO BUILD GNU PACKAGES

Supposing that you want to make a package that can be compiled without VDKBuilder you can use autoconf/automake feature. It prepares a complete package that can be used and compiled without VDKBuilder. This feature is particularly useful if you want to add NLS (Native Language Support) to your application. In order to have this capability you need:

- autoconf version >= 2.13
- automake (GNU automake) version >= 1.4

Here we want to make a GNU package for our "draw" application (refer to "Widgets for drawing section "), at this end we will use Autoconf/Automake support that builder provides.
The GNU package prepared can be built without having VDKBuilder installed, furthermore through the already famous sequence: **"./configure − make −make install"** you have a package that can be easily tailored to different systems.
Before proceeding let's briefly explain what the package contains:

- A configure script that fits compilation, linking and installing for the target system.
- A makefile that provides all most important operations including installing and uninstalling the package
- All necessary source and include files to build the application
- An "extra_dist.am" that exists for an list of additional files that aren't directly involved in the package build but could be necessary to properly run the application, e.g. Pixmap files, documentation, etc.
  "extra_dist.am" format is simple just lines containg file names that will be distributed and installed with the package. Be sure to always write a complete line with line−feed into the file.
- A file named "am_include.am" that exists for if there is a need to append to EXTRA_DIST, and the contents of this file are copied verbatim to the end of Makefile.am. "am_include.am" can be used for automake features not yet supported directly
- Package name will be made using project name as suffix followed with a versioning number that by default is 0.0.1. Both package name and version can be customized using Project−>Options menu and editing related fields in project options.
- Some extra files, like AUTHORS,COPYING,README and INSTALL.
  Writing these files is your job.

Now let's proceed:

- Open with builder "draw" project
- Write extra_dist.am and am_include.am if needed (shouldn't be the case)
- Select Autoconf/Automake−>Autogen and answer yes to the prompts
- VDKBuilder maker will show you the package preparing sequence.
- Repeat step above (due to a not yet investigated bug, the sequence does not complete at first try)
- Select Autoconf/Automake−>Make, you will see the compilation process
- Select Autoconf/Automake−>Make Dist, you will see that package will be prepared and named draw−0.0.1.tar.gz

To check if the package is complete and working close vdkbuilder and open a xterm:

```
$ cd <where−you−make−tutorial−exercises>/draw
$ mkdir tmp
$ cd tmp
$ cp ../draw/draw−0.0.1.tar.gz .
$ tar xvzf draw−0.0.1.tar.gz
$ cd draw−0.0.1
$ ./configure
$ make
$ ./draw&
```

- you should see draw3 program compiled and working without VDKBuilder.
- now you can delete tmp dir

```
$ cd <where−you−make−tutorial−exercises>
$ rm −fr tmp/
```

intentionally left blank page

APPENDIX D

NLS (How to make a Native Language Support application using VDKBuilder)

As of version 1.2.4 of VDKBuilder there has been NLS support. However support is implemented in a way that requires some additional code writing to support NLS in gui files side (those files that are overwritten by builder and out of user control). The remaining part of the NLS support is the user's responsibility. They must write their own code in a way to support NSL and take advantage of gettextize, edit configure.in, aclocal.m4, POTFILES.in, po.* files and other necessary activities. For this, the following notes should help the new user in constructing a nls application with VDKBuilder. Notes are written as a check−list and all actions should be made in the presented order. Where applicable I have put also explainatory notes together for the actions that must be taken. I have taken a simple hello program as a tutorial base, however the procedure is applicable to more complex examples. So let's assume the user will construct a simple hello program that shows a form with a label and two buttons, one labeled "Hello" and the other "Close". Clicking on Hello button label will show the famous phrase "Hello world" and clicking on "Close" button quits the application. So the phrases to translate are: **"Hello world", "Hello" and "Close".**

CONSTRUCTING NLS_HELLO PROGRAM WITH VDKBuilder

First action to take is to create a new project named "nlsHello", build a program following above guidelines and in "project options" be sure to check the NLS support checkbutton.
When you finished and checked that the program runs ok save the project.
Taking a look to hello_gui.cc you will see that all button captions are defined using a _("caption") notation, you will see also the defines for macro _(str) and N_(str), this define depends on some other defines such as ENABLE_LS and HAVE_GNOME. Those definitions drive gettext operations. VDKBuilder wrote for you these definitions and use the _("") or N_("") notation but other work is necessary bfore you can see hello program translated in your language. Below are the notes to guide you through the process.

USING AUTOCONF/AUTOMAKE

In order to localize your program (thus make it translatable in other languages) you have to use Autoconf/Automake VDKBuilder support. Once you are sure that your nlsHello program works well use Autoconf/Automake menu and run Autogen (you will notice that you must run it twice in order to have other menus enabled). This will prepare all necessary files to construct hello using automake features. Now an important point: from now on when you will use Autogen again remember to choose to DO NOT REGENERATE, so answer always "no" to the prompt that ask you if you want regenerate some files. The reason is simple: you are going to manually modify some files to prepare NLS support (configure.in, Makefile.am and acconfig.h) and you want avoid to overwriting those changes. However even if you make the mistake of answering yes, you can recover your modified files from a backup copy that is by builder for you. They are named configure.in.old, etc...

PREPARING YOUR PROGRAM TO BE USED WITH GETTEXT

The first thing to do is to "gettexttize" the directory where nlsHello project is contained. For this we will use gettextize program that will prepare two subdirs: intl and po that will contain files for nls operations.
So open an xterm and:

```
– $ cd <where nlsHello project is>
– $ gettextize −f .
```

answer yes to the program prompt, if any. gettextize will add to your dir file: ABOUT−NLS and will create ./po and ./intl subdirs with all necessary files.
Once gettextize has finished it will leave you with the following message:
"You should update your own 'aclocal.m4' by adding the necessary macro packages gettext.m4, lcmessage.m4 and progtest.m4 from the directory '/aclocal'"
aclocal.m4 is a file prepared by autogen and that now should be updated with above macro packages. You should know where aclocal packages are, generally they are either on /usr/share/aclocal or /usr/local/share/aclocal. Here we assume that are located into /usr/share/aclocal. So append to your aclocal.m4 macro packages contained into above suggested m4 files.

```
– $ cat /usr/share/aclocal/gettext.m4 > aclocal.m4
– $ cat /usr/share/aclocal/lcmessage.m4 > aclocal.m4
– $ cat /usr/share/aclocal/progtest.m4 > aclocal.m4
```

now update your project dir with aclocal program

– $ aclocal

EDITING AUTOCONF/AUTOMAKE FILES

Now it is necessary to update manually some autoconf/automake generated files in your project dir.

– **Editing ./configure.in**

Edit configure.in inserting the following lines below AC_CANONICAL_HOST

```
dnl -------------------------
dnl Gettext stuff
ALL_LINGUAS="en <your prefix>"
AM_GNU_GETTEXT
dnl -------------------------
```

substitute <your prefix> with your languages prefix, as an example could be:
ALL_LINGUAS="en it fr de es" thus meaning that program should be localized in en−english, it−italian, fr−french,de−german and es−spanish
now insert these lines below AC_SUBST(CXXFLAGS)

```
dnl ---------------------------------------
dnl Set the location of the locale messages to be used in main.cc
dnl This will substitute NLSHELLO_LOCALE_DIR in config.h
dnl (from acconfig.h).
if test "x$prefix" = "xNONE"; then
 NLSHELLO_LOCALE_DIR=$ac_default_prefix/share/locale
else
 NLSHELLO_LOCALE_DIR=$prefix/share/locale
fi
AC_DEFINE_UNQUOTED(NLSHELLO_LOCALE_DIR, "$NLSHELLO_LOCALE_DIR")
AC_DEFUN(FC_EXPAND_DIR, [
$1=$2
    $1=‘(
       test "x$prefix" = xNONE && prefix="$ac_default_prefix"
       test "x$exec_prefix" = xNONE && exec_prefix="${prefix}"
       eval echo \""[$]$1"\"
         )‘
        ])
dnl---------------------------------------
```

now change line:

```
AC_OUTPUT(Makefile)
```

to read:

```
AC_OUTPUT(Makefile po/Makefile.in intl/Makefile)
```

Save configure.in

– **Editing Makefile.am**

Edit Makefile.am and change line:

```
SUBDIRS = .
```

to read:

```
SUBDIRS = po intl
```

Again save Makefile.am

– **Editing acconfig.h**

Edit acconfig.h and add these lines:

```
#undef ENABLE_NLS
#undef HAVE_CATGETS
#undef HAVE_GETTEXT
#undef HAVE_LC_MESSAGES
#undef HAVE_STPCPY
#undef NLSHELLO_LOCALE_DIR
```

Again save acconfig.h

– **Creating po/POTFILES.in file**

POTFILES.in file is used to know which source files should be scanned to search translatable strings.

Create and save po/POTFILES.in putting a source file for each line:

```
nlsHello.cc
nlsHello_gui.cc
```

this file will be used by autogen to create the correct Makefile into po subdirectory.

RUNNING AUTOGEN AND MAKE

- Using Autoconf/Automake menu Make distclean to remove old files
- Autogen (recall DO NOT REGENERATE)
- Make

You should receive following messages:
Makefile.am:20: it in 'ALL_LINGUAS' but po/it.po does not exist
Makefile.am:20: en in 'ALL_LINGUAS' but po/en.po does not exist
make[2]: *** No rule to make target 'en.po', needed by 'en.gmo'. Stop.
that is normal since constructing above files will be the next step. If you look into po subdir you will see file "hello.pot" that is the template file obtained from source scanning.

TRANSLATING INTO LOCALE LANGUAGES

Copy hello.pot into en.po and whichever other prefix you defined into ALL_LINGUAS macro in configure.in. So with the xterm :

```
$ cd po
$ cp nlsHello.pot en.po
$ cp nlsHello.pot <your prefix>.po
```

Normally en.po does not need to be translated since all strings should be as in sources. What you translate are it.po and companions. Each po file appear in this format:

```
#: nlsHello_gui.cc:36
msgid "nlsHello Main Form"
msgstr ""
```

where the first line shows which source line were scanned "msgid" shows original string and "msgstr" what you have to translate.
Edit to read:

```
#: nlsHello_gui.cc:36
msgid "nlsHello Main Form"
msgstr "<your translation>"
```

and so on for all msgid/msgstr pairs in the file.
NOTE:
emacs editor recognizes po files and provides an easy way to edit this file with many nice features. If you use emacs just add to your .emacs file these lines:

```
(setq auto-mode-alist
      ( cons '("\\.pox?\\'" . po-mode) auto-mode-alist))
(autoload 'po-mode "po-mode")
```

PREPARING YOUR SOURCE CODE TO NLS

VDKBuilder supports nls on the gui files only, other sources are your responsibility, so you have to prepare them to support nls.

- **Editing nlsHello.cc**

  At file header insert:

```
/*
nlsHello Plain VDK Application
Main unit implementation file:hello.cc
*/
#if HAVE_CONFIG_H
#include <config.h>
#endif
#if !HAVE_GNOME
#if ENABLE_NLS
#include <libintl.h>
#define _(str) gettext(str)
#define N_(str) str
#else
#define _(str) str
#define N_(str) str
#endif
#endif
//--------------------------------
#include <nlsHello.h>
```

  (you will notice that same header is shown also in hello_gui.cc)
  now edit main() to read:

```
/*
main program
*/
int main (int argc, char *argv[])
```

```
  {
  #if ENABLE_NLS
    bindtextdomain(PACKAGE,NLSHELLO_LOCALE_DIR);
    textdomain(PACKAGE);
  #endif
  // makes application
  HelloApp app(&argc, argv);
  // runs application
  app.Run();
  return 0;
  }
```

– **Changing project options**
  Use "project −> options" menu and edit the "defines field" to read:
  `−DHAVE_CONFIG_H=1 −DHAVE_GNOME=1` (or 0 if gnome is not installed or incompatible)

BUILDING NSL AWARE APPLICATION

The first build should be made using an xterm and Makefile generated by automake:

– `$ make`
  this will make hello program and prepare translated files

– `# make install`
  this will install translated files to appropriate usr/share/locale directory so it can be correctly bound by bindtextdomain().

– `$ nlsHello`
  you should see that hello now shows button captions translated into your language.

ADDING MORE TRANSLATABLE STRINGS

If you click on hello button you will see still "hello world" in english not in your language as you expect.
This is because "hello world" was not marked for translation yet. As a general rule each string with _("a string") notation is marked for translation and will be parsed to be added to po files.
Most likely you have established a signal response function on hello button clicked:

```
//signal response method
bool
HelloForm::OnhelloButtonClick(VDKObject* sender)
{
  label−>Caption = "Hello world";
  return true;
}
```

Now change to read:

```
label0−>Caption = _("Hello world");
```

Adding a translatable string need to update po files and make−install again:

– `$ cd po`
– `$ make update−po`

Edit <your prefix>.po and translate "hello world"

– `# make install`
– `$ hello`
  should show "Hello world" translated if you click on hello button.

FINAL NOTES

As long as you do not add any translatable strings directly in your source code or indirectly by adding widgets with captions to gui, you can continue to use builder's normal project−making facilities. But if you want see your newly added translatable strings you have to repeat the steps:

```
$ cd po
$ make update−po
```

translate

```
# make install
```

For further reading on NLS you should look at GNU gettext manual available at http://www.gnu.org

APPENDIX E − vdkxdb library
USING VDKBUILDER WITH DATA−AWARE WIDGETS

Libvdkxdb is a wrapper on xbase[16], a light−weight but powerfull database library that uses the well known dbf and .ntx data formats. Since most of vdkxdb classes are derived from xbase classes you should have a knowledge of xbase or at least read the related documentation.
Xbase can be found at http://xdb.sourceforge.net
Libvdkxdb is a set of data−aware widgets that can be linked with underlying database as well as used as normal GUI widgets:

- **VDKXDataBase**
  Is a root class, can be thought as a table of data files
- **VDKXTable**
  A single relational data table made of fields (or columns) and records (or rows)
- **VDKXTableIndex**
  A single index based on a table field that becames the index key
- **VDKXEntry**
  Derived from VDKEntry, is a widget that can handle character based fields limited to 256 chars in length
- **VDKXCheckButton**
  Derived from VDKCheckButton, is a widget that can handle boolean fields
- **VDKXMemo**
  Derived from VDKTextView, can handle character based fields longer than 256 chars

VDKBuilder has a lot of features to support using this set of data−aware widgets.

How build libxdb
You need xbase library installed, at the time of this writing latest version is 2.0.0.

- download vdkxdb−2.0.0.tar.gz from builder site
- unpack

```
$ tar xvzf vdkxdb−2.0.0.tar.gz
```

- configure

```
$ ./configure <your options>
```

- build

```
$ make
```

- install (have root permission)

```
$ su
enter pwd
# make install
```

Prepare VDKBuilder for libvdkxdb
Now you need to rebuild VDKBuilder in order to support libvdxdb.

- `$ cd <where−builder−is>/vdkbuilder−2.0.0`
- `$ ./configure <your options> −−enable−vdkxdb`
  in order to let some parts of builder be compiled
- `$ make`
- `$ make install`

Running builder you will see that there have been some changes:
- tool palette shows a new page labeled “Xdb widgets”
- the Project menu has a new item labeled “XDB support”

In this section we will make an application named “yam” (**Y**et **A**nother **M**emorandum) to demonstrate how libvdkxdb works.

16 Initially this project, was a fork off “xbase” project and was named “xdb”, now, due to licensing problems it has returned to it’s original name. Here we will refer to it as xdb and xbase, interchangably.

THE YAM PROJECT

Our reader should be at this point able to construct a new project named “yam”, for now let’s be satisfied with an empty form with a fuzzy background just to see what builder has changed when generating source files to support xbase operations, here we will discuss those differences.
As a general rule differences with previously generated source files are contained in conditional compilation statements:

```
#ifdef VDKXDB_SUPPORT
// ...
#endif
```

the reason is is simple, if VDKXDB_SUPPORT is not defined all xdb support will be simply ignored, therefore we have to add such define to our project options.

- use Project−>Options menu
- add −DVDKXDB_SUPPORT to “defines” field
  From now on all statements between `#ifdef VDKXDB_SUPPORT/#endif` will be compiled. Taking a look at project options you can see another difference compared with default values filled by builder without xdb support: configuration script name has changed from “vdk−config−2” to “vdkxdb−config−2”.
  Trying executing in an xterm:

```
$ vdkxdb-config-2 --cflags
$ vdkxdb-config-2 --libs
```

  you will see that the former has added include paths for libvdkxdb and the latter linking flags for both libvdkxdb and xbase libraries.
- click on the button labeled “Close and Save”.

Now have step−by−step look at the generated source files and using bold face to highlight the changes.
yam.h

```
/*
yam Plain VDK Application
Main unit header file: yam.h
*/
#ifndef _yam_main_form_h_
#define _yam_main_form_h_
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif
// vdkxdb support
#ifdef VDKXDB_SUPPORT
#include <vdkxdb/vdkxdb_support.h>
      /* All tables and indexes files names are relative to XDB_DATA_PATH symbolic
      constant, change it if you need a path other than default , however all data and index files
      should be contained in a single directory
      */
#define XDB_DATA_PATH "./"
#endif

// vdk support
#include <vdk/vdk.h>
// Yam FORM  CLASS
class YamForm: public VDKForm
{
// gui object declarations
private:
// gui object declarations
      void GUISetup(void);

public:
      YamForm(VDKApplication* app, char* title);
      ~YamForm();
      void Setup(void);
/*
 gui setup include
 do not patch below here
*/
#include <yam_gui.h>
};
```

```
// Yam APPLICATION  CLASS
class YamApp: public VDKApplication
{
```

```
      /* xdb is a pointer to a database that you will use to manage overall xdb operations.
      xdb will be freed at program termination by app destructor.
      */
```

```
#ifdef VDKXDB_SUPPORT
      VDKXDatabase* xdb;
#endif
public:
YamApp(int* argc, char** argv);
~YamApp();
```

```
      /* By analogy application class has a setup function written by builder where it takes care
      of opening table and indexes, etc.. At any moment class xdb can be accessed via TheXdb()
      method. Since this method can violate data hiding concepts, hopefully, in the future it will
      converted into a read−only property.
      */
```

```
#ifdef VDKXDB_SUPPORT
      void XDBSetup(void);
      VDKXDatabase* TheXdb() { return xdb; }
#endif
void Setup(void);
};

#endif
// do not remove this mark: #!#
// end of file:yam.h
```

yam.cc

```
/*
yam Plain VDK Application
Main unit implementation file:yam.cc
*/
#include <yam.h>
/*
main program
*/
int main (int argc, char *argv[])
{
// makes application
YamApp app(&argc, argv);
// runs application
app.Run();
return 0;
}

// Yam MAIN FORM  CLASS
/*
main form constructor
*/
YamForm::YamForm(VDKApplication* app, char* title):
      VDKForm(app,title,v_box,DisplayType)
{
}

/*
main form destructor
*/
YamForm::~YamForm()
{
}
```

```
/*
main form setup
*/
void
YamForm::Setup(void)
{
      GUISetup();
```

```
// put your code below here

}

// Yam APPLICATION  CLASS
/*
application constructor
*/
YamApp::YamApp(int* argc, char** argv):
      VDKApplication(argc,argv)
{
      /* construct class database
      */
#ifdef VDKXDB_SUPPORT
  xdb = new VDKXDatabase;
#endif
}

/*
application destructor
*/
YamApp::~YamApp()
{
      /* delete class database
      */
#ifdef VDKXDB_SUPPORT
  if(xdb) delete xdb;
#endif
}

/*
application setup
*/

void
YamApp::Setup(void)
{

#ifdef VDKXDB_SUPPORT
      XDBSetup();
#endif
      MainForm = new YamForm(this,NULL);
      MainForm->Setup();
      MainForm->Show(YamForm::InitialPosition);
}

// do not remove this mark: #!#
// end of file:yam.cc
```

yam_gui.h

```
/*
yam gui header
*/
public:
/*
declaring signal and events
dynamics tables
*/
DECLARE_SIGNAL_LIST(YamForm);
DECLARE_EVENT_LIST(YamForm);
// declares two static used to initialize
// form display type and initial position
static GtkWindowType YamForm::DisplayType;
static GtkWindowPosition YamForm::InitialPosition;
// do not remove this mark: #!#

// end of file:yam_gui.h
```

yam_gui.cc

```
#include <yam.h>
//define static display type and initial form position
GtkWindowType YamForm::DisplayType = GTK_WINDOW_TOPLEVEL;
GtkWindowPosition YamForm::InitialPosition = GTK_WIN_POS_NONE;
/*
defining signal and events
dynamics tables
*/
DEFINE_SIGNAL_LIST(YamForm,VDKForm);
DEFINE_EVENT_LIST(YamForm,VDKForm);
/*
main form setup
*/
void
YamForm::GUISetup(void)
{
      SetSize(402,304);
      Title = "Yet Another Memorandum";
}
```

/* since at the moment we are working with an empty database set, setup method is void
*/

```
#ifdef VDKXDB_SUPPORT
void
YamApp::XDBSetup(void)
{
}
#endif

// do not remove this mark: #!#
// end of file:yam_gui.cc
```

PLANNING YAM TABLES

Constructing our database requires a little planning, what we should make is a simple database made of one table that on each row (or record) contains following fields (or columns):

- "FIRSTNAME", that contains up to 12 characters
- "LASTNAME", that contains up to 12 characters
- "BIRTHDATE", that contains a date
- "ADDRESS", that contains up to 32 characters
- "CITY", that contains up to 16 characters
- "INCOMING", that contains a numerical value with 2 decimals
- "BOOLEAN", that contains a boolean value.
- "MEMO", that contains a free form description

I guess field names are self−explanatory, agreed this table does not make a lot of sense, but is just for demonstration of all the available types of data that can be dealt with.

Furthermore we want index our table using both FIRSTNAME/LASTNAME and BIRTHDATE, so we plan to have:

- primary key on FISRTNAME+LASTNAME (with leading and trailing blanks removed if any), accordding to relational rules this key must be unique.
- a secondary key on BIRTHDATE

Xbase requires you to have an index file for each key (multiple key index files aren't supported yet).

So let's say that our database will be made of:

- a table (whose data file will be named "memorandum.dbf") that contains data
- two indexes (whose files will be named key1.ndx and key2.ndx).

VDKXdb uses same conventions as xbase to describe a record track, you have to construct a map, using an array of structures (of type VDKXRecordTemplate) like this:

```
VDKXRecordTemplate Memorandum[] =
{
//  field name    data type         length      decimal (if any)
// ------------------------------------------------------------
  { "FIRSTNAME",  XB_CHAR_FLD,      12,         0 },
  { "LASTNAME",   XB_CHAR_FLD,      12,         0 },
  { "BIRTHDATE",  XB_DATE_FLD,      8,          0 },
  { "ADDRESS",    XB_CHAR_FLD,      32,         0 },
  { "CITY",       XB_CHAR_FLD,      16,         0 },
  { "INCOMING",   XB_NUMERIC_FLD,   9,          2 },
  { "BOOLEAN",    XB_LOGICAL_FLD,   1,          0 },
  { "MEMO",       XB_MEMO_FLD,      10,         0 },
  { "GAPFILLER",  XB_CHAR_FLD,      28,         0 },
// null ended
  { "",0,0,0 }
};
```

structure fields are described above, let's just say that:

- field name lentgh is up to 10 characters
- date, logical and memo fields are fixed in length: 8,1 and 10 respectively. For memo fields contents there is no limit since a special file will be constructed to manage them.
- structure array must end with a structure with nulled fields as shown above.
- a good habit I suggest, is to use a gap filler to normalize record size to a multiple of 64 bytes. This should speed up read/write operations on hard disk.
- We will use this structure in order to construct data table and indexes at first run, all these informations will be written on the data file header, futher operations won't need them.

So lets write above map along with other informations in yam.cc:

```
/*
yam Plain VDK Application
Main unit implementation file:yam.cc
*/
#include <yam.h>
// database files names
#define DBFNAME "./memorandum.dbf"
#define KEY1    "./key.ndx"
#define KEY2    "./key2.ndx"

VDKXRecordTemplate Memorandum[] =
{
//  field name    data type         length      decimal (if any)
// ---------------------------------------------------------------
  { "FIRSTNAME",  XB_CHAR_FLD,      12,         0 },
  { "LASTNAME",   XB_CHAR_FLD,      12,         0 },
  { "BIRTHDATE",  XB_DATE_FLD,      8,          0 },
  { "ADDRESS",    XB_CHAR_FLD,      32,         0 },
  { "CITY",       XB_CHAR_FLD,      16,         0 },
  { "INCOMING",   XB_NUMERIC_FLD,   9,          2 },
  { "BOOLEAN",    XB_LOGICAL_FLD,   1,          0 },
  { "MEMO",       XB_MEMO_FLD,      10,         0 },
  { "GAPFILLER",  XB_CHAR_FLD,      28,         0 },
// null ended
  { "",0,0,0 }
};
/*
main program
*/
// ....
```

Creating database

As said before on the first run we will construct data and indexes file, so we need to do this before any other initialization, later we will see where is the best place to do this. Now let's code the method that does what we need:

```
void
YamApp::SetupDatabase(void)
{
      /* constructs a new table, that will be added to database
      */
  VDKXTable* table = new VDKXTable(xdb,DBFNAME);
  xbShort rc;
  VDKXTableIndex* index;
      /* physically creates table data file, if exits does nothing returning
      XB_FILE_EXISTS  return code.
      */
  rc = table->Create(memorandum);
      /* successful creation, opens table and creates indexes
      */
  if(rc != XB_NO_ERROR)
    table->Open();
  index = new VDKXTableIndex(table,KEY1);
      /* key expressions can be used in index key, in this case "FIRSTNAME+LASTNAME"
      means concatenate the two fields removing leading and/or trailing white spaces
      */
  index->Create("FIRSTNAME+LASTNAME");
  index = new VDKXTableIndex(table,KEY2);
  // last arg set to false means not unique key, so key clashes are
  // allowed
  index->Create("BIRTHDATE",false);
```

```
        /* close table, all associated indexes are closed as well
        */
    table->Close();
        /* remove temporary table from database
        */
    xdb->RemoveTable(DBFNAME);
}
```

Above method constructs a temporary table creating data file if does not exists and associated indexes, table will be removed from database at the end. Since this method most likely will be completely used on first program run to create data files and indexes we must be sure that will be executed before any other widgets or table initialization so the best place is as first line in YamApp::Setup().

Edit yam.cc to read:

```
/*
*/
void
YamApp::Setup(void)
{
      SetupDatabase ();
#ifdef VDKXDB_SUPPORT
      XDBSetup();
#endif
      MainForm = new YamForm(this,NULL);
      MainForm->Setup();
      MainForm->Show(YamForm::InitialPosition);
}
```

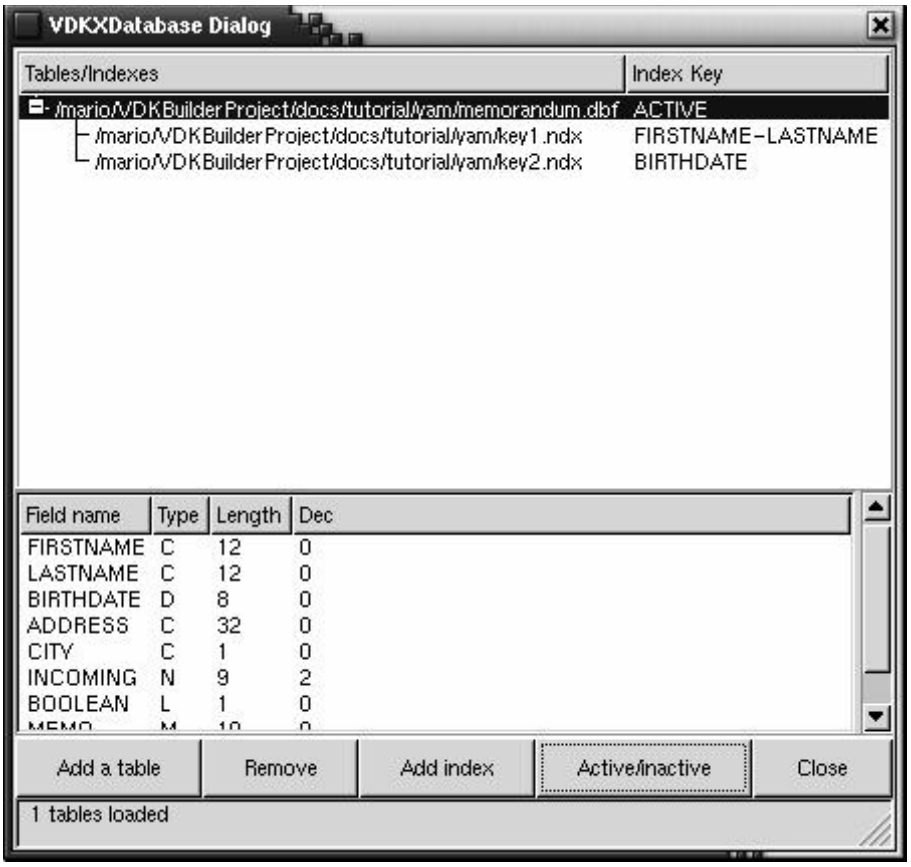
Building and running the program you can verify that has constructed following data−emptyfiles:

- *key1.ndx* 1024 bytes
  Primary key index.
- *key2.ndx* 1024 bytes
  Secondary key index.
- *memorandum.dbf* 321 bytes
  table data file.
- *memorandum.dbt* 512 bytes
  data file containing memo fields text.

Adding database to builder knowledge

Now we will let the builder know about our database so it will be able to use data−aware widgets at design time and generate the code to support vdxdb operations.

- use Project−>XDBSupport
  a dialog form will appear
  - Click on button labeled “Add table”
  - Choose in File dialog memorandum.dbf
  - Click on button labled “Open”
- Now memorandum.dbf will appear as the first node of the upper tree.
- Click on root node item, lower list will be filled with all table fields togheter with name, type, length and decimal if applicable.
- Click on button labeled “Add index”, choose in file dialog key.ndx and click on “Open” button.
- Repeat as above for key2.ndx
- Select again root node memorandum.dbf on upper tree and click on “Active/inactive” button. This will make the builder use database also during design time and generate the code for opening both table and indexes. Below should be the result.
- Click on “Close” button.



Just a word on data and index files location: by default they should stay in the project directory, however by changing XDB_DATA_PATH defined in yam.h we can have them in a different directories. However all data and index files must stay be in the same directory.

− build the application

A look to yam_gui.cc clarifies what is the role of YamApp::XDBSetup(void)

/* Initializes all tables and indexes contained in yam.prj.xdb and open tables if marked as active.

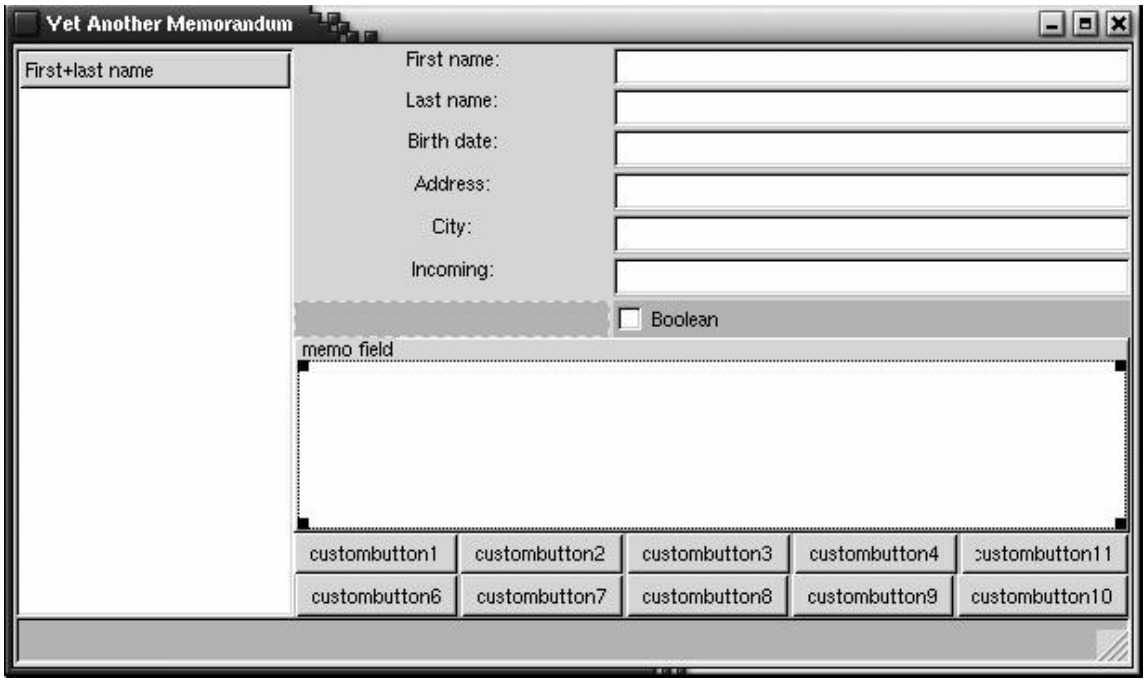
- all path to tables and indexes files are relative to XDB_DATA_PATH symbolic constant, change it in yam.h header file if you need a different path, all data files should be contained in a single directory however
- method will be empty if no tables are used

*/

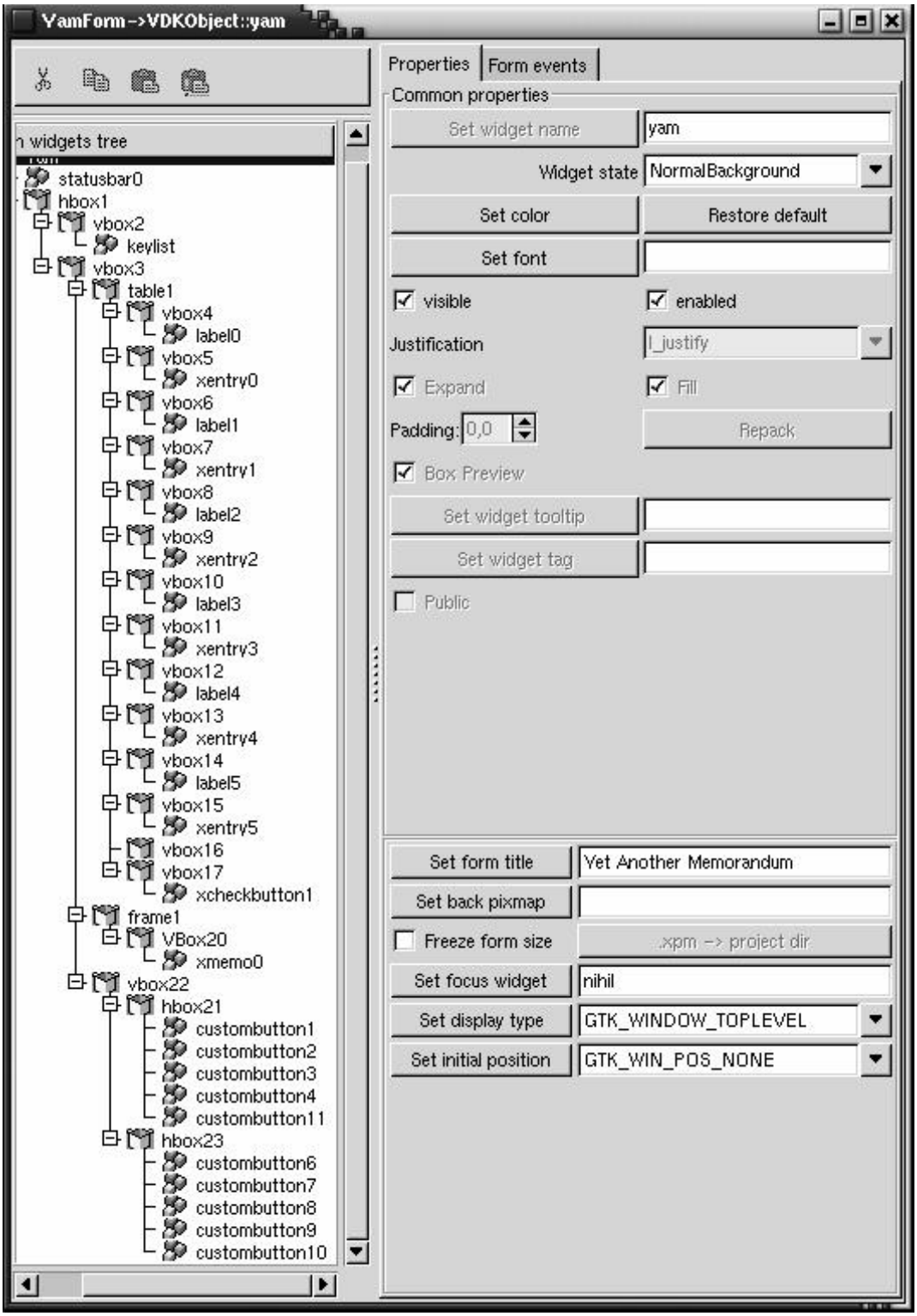
```
#ifdef VDKXDB_SUPPORT
void
YamApp::XDBSetup(void)
{
VDKXTable* table = NULL;
VDKXTableIndex* index = NULL;
VDKString dbname;
VDKString ndxname;
dbname = XDB_DATA_PATH;
dbname += "memorandum.dbf";
table = new VDKXTable(xdb,(char*) dbname);
if(table)
  table->Open();
ndxname = XDB_DATA_PATH;
ndxname += "key1.ndx";
index = new VDKXTableIndex(table,(char*) ndxname);
if(index)
  index->Open();
ndxname = XDB_DATA_PATH;
ndxname += "key2.ndx";
index = new VDKXTableIndex(table,(char*) ndxname);
if(index)
  index->Open();
if(index && table && table->IsOpen())
```

```
    table->Order = 0;
if(table && table->IsOpen())
    table->First();
}
#endif
```

Designing the interface
We can now design the interface as usual, leaving the database considerations as last ones.
Let's show how the application main form should look like in the early stages:



The related widgets tree that should help reader in constructing the interface:



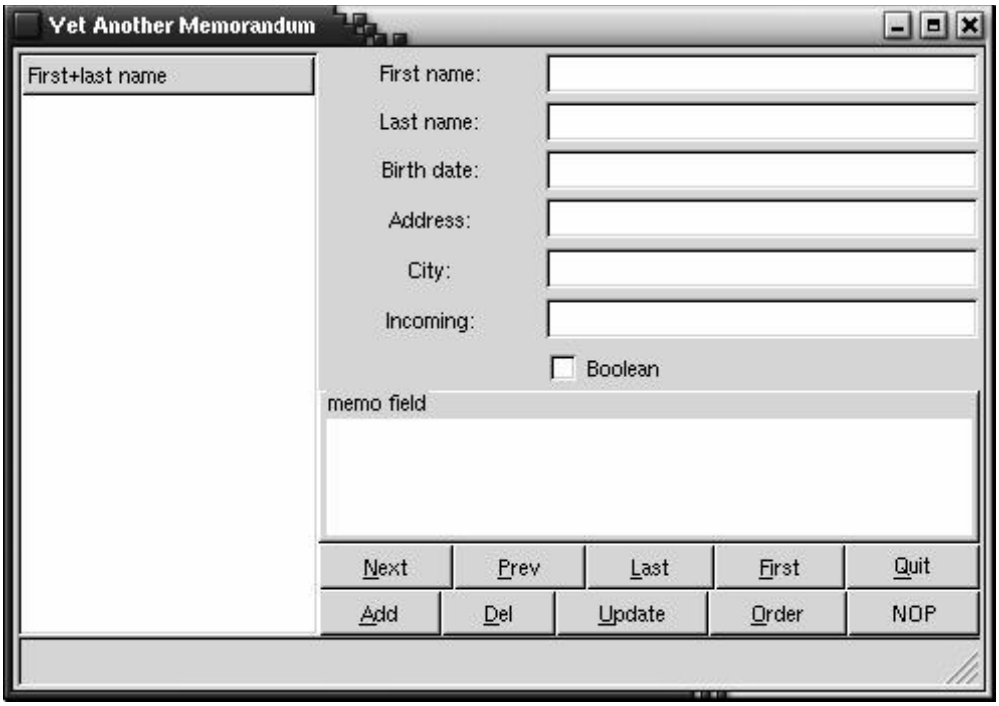
Refining the interface

Now let's refine interface widgets:

- Begin by making widget names more meaningful:
  - xcustomlist0 −> keylist
  - xentry0 −> firstname
  - xentry1 −> lastname
  - xentry2 −> birthdate
  - xentry3 −> address
  - xentry4 −> city
  - xentry5 −> incoming

  - xcheckbutton0 −> boolean
  - xmemo0 −> memo
  - statusbar0 −>statusbar
  - custombutton1 −> nextBtn
  - custombutton2 −> prevBtn
  - custombutton3 −> lastBtn
  - custombutton4 −> firstBtn
  - custombutton11 −> quitBtn
  - custombutton6 −> addBtn
  - custombutton7 −> delBtn
  - custombutton8 −> updateBtn
  - custombutton9 −> orderBtn
  - custombutton10 −> nopBtn
- set some captions on labels:
  - label0 caption = “First name:”
  - label1 caption = “Last name:”
  - label2 caption = “Birth date:”
  - label3 caption = “Address:”
  - label4 caption = “City:”
  - label5 caption = “Incoming:”
  - boolean caption = “Boolean”
- set some captions on buttons:
  - nextBtn caption = “_Next”
  - prevBtn caption = “_Prev”
  - lastBtn caption = “_Last”
  - firstBtn caption = “_First”
  - quitBtn caption = “_Quit”
  - addBtn caption = “_Add”
  - delBtn caption = “_Del”
  - updateBtn caption = “_Update”
  - orderBtn caption = “_Order”
  - nopBtn caption = “NOP”
- Set keylist column header title to “First+last name” and border shadow to shadow_in
- Set frame 1abel to “memo field”

Just a note, may be you do not have same auto−numbered widgets name, however previous picture should help you in constructing the GUI without pain.
Last action is to shrink the form to a satisfactory size allowing all widgets to be easily visible.
Build and run the application, next picture shows the interface when it’s finished.



Linking data−aware widgets to database

Now it's time to connect data aware widgets to underlying database table:

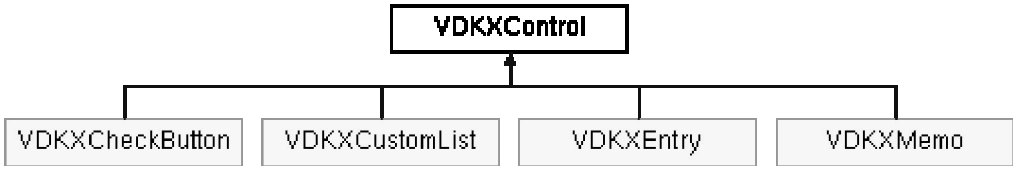
- select firstname widget
- WI will show you a table with memorandum.dbf and all its fields
- select list item named "FIRSTNAME" and click on "Assign to table field" button.
  This means that FIRSTNAME field was assigned to firstname widget, from now on all read/write operations will be done using this widget. The first list item "nihil" if assigned to the widgets will unlink the widgets from database table.
- Now repeat above actions assigning:
  - LASTNAME to lastname
  - BIRTHDATE to birthdate
  - ADDRESS to address
  - CITY to city
  - INCOMING to incoming
  - BOOLEAN to boolean
  - MEMO to memo

Also keylist could be linked with database, assigning a field to each list column, but we will do it later on, most likely changing the widget.

Data−aware widgets mass operations

Just a few words about data−aware widget operations: they can both Read() and Write() from/to data file, meaning that the field contents is transferred from data buffer to the widget and the widget content is transferred to data buffer.

Taking a look to VDKXdb hierarchy you will notice that all data widgets are subclasses of VDKXControl class as described in the picture below



VDKXControl implements Read() and Write() virtual methods that are overridden in subclasses. The purpose is clear, since each widget can handle data field of different type we need different behaviours. For instance a VDKXEntry can handle date, numeric, float and ASCII data types, VDKXCheckButton

can handle only boolean values so different behaviour is needed. Furthemore data widgets are sensitive to field data type, if a widget handles date, it will accept date format and will write date format into data buffer, taking care of all necessary conversions. For now what's interesting to us is that we can think of all data widgets as if they were VDKXControls, we can insert them in a data structure like a list or an array and invoke Read() or Write() on each widget, virtuality will assure that the correct method will be invoked depending on data widget type. In this way we can create a sort of "mass operation" that can read or /write data in one shot.

For this, we have an array of VDKXControls* as YamForm member, modify Yam.h to read:

```
class YamForm: public VDKForm
{
//...
// gui object declarations
private:
// gui object declarations
  void GUISetup(void);
  VDKArray<VDKXControl*> AllFields;
  void Read();
  void Write();
  VDKXTable* memorandum; // memorandum table address
public:
//...
```

Now let's modify YamForm::Setup() in order to load array with data widgets addresses:

```
/*
main form setup
*/
void
YamForm::Setup(void)
{
  const int DATA_WID_NUMBER = 9;
  GUISetup();
  // load AllFields array
  AllFields.resize (DATA_WID_NUMBER);
  AllFields[0] = firstname;
  AllFields[1] = lastname;
  AllFields[2] = birthdate;
  AllFields[3] = address;
  AllFields[4] = city;
  AllFields[5] = incoming;
  AllFields[6] = firstname;
  AllFields[7] = boolean;
  AllFields[8] = memo;
  // since data file is opened, make a mass reading
  // to load all data widgets
  Read();
}
```

Now lets code YamForm::Read() and YamForm::Write() mass operations

```
/*
  mass reading operation
*/
void
YamForm::Read ()
{
  for (int t = 0; t < AllFields.size (); t++)
    AllFields[t]->Read ();
  }
/*
  mass writing operation
*/
void
YamForm::Write ()
{
  for (int t = 0; t < AllFields.size (); t++)
    AllFields[t]->Write ();
  }
```

At this point we are able to implement the first action on the database: adding a new record.

First of all we store into <VDKXTable* memorandum> the address of database table for quicker operations, at this end we use YamForm constructor:

```
/*
main form constructor
*/
YamForm::YamForm(VDKApplication* app, char* title):
      VDKForm(app,title,v_box,DisplayType)
{
  // get table pointer
  YamApp* theApp = dynamic_cast<YamApp*>(app);
  memorandum = theApp ? (*theApp->TheXdb())[DBFNAME] : NULL;
}
```

and finally write the code to add records to our memorandum data file:

- select addBtn on GUI designer
- connect it with clicked_signal
- modify response method to read:

```
//signal response method
static char buff[256];
bool
YamForm::OnaddBtnClick(VDKObject* sender)
{
  xbShort rc;
  if(!memorandum)
    return true;
  // prepares a blank data buffer
  memorandum->BlankRecord();
  // mass writing from widgets to data buffer
  Write();
  // append data buffer (a record) to data file
  rc = memorandum->AppendRecord();
  // report status on status bar
  sprintf(buff,"Adding a record - %s",VDKXError(rc));
  statusbar->Push(buff);
  return true;
 }
```

Building and running the program you will be able to add some records to data file after having filled all necessary entry widgets. Furthermore, on next builder run you will see your data widgets filled with first data record fields values also into GUI designer.

Going on with Yam application

Now let's code some other actions like moving aroung in the data file, updating a record and so on.
We show here how to customize response methods name

- select nextBtn and switch on signal page
- edit "Class response method field" to read: `OnMovingBetweeenRecords`
- hit return key
- click on "Click" button
  now signal map will show:
  `ON_SIGNAL(nextBtn,clicked_signal,OnMovingBetweenRecords)`
- reuse this response method also for prevBtn,firstBtn and lastBtn

here the code for the method:

```
//signal response method
bool
YamForm::OnMovingBetweenRecords(VDKObject* sender)
{
  VDKCustomButton *button = dynamic_cast <VDKCustomButton*>(sender);
  xbShort rc;
  if (!button || !memorandum)
    return true;
  if (button == nextBtn)
    rc = memorandum->Next ();
  else if (button == prevBtn)
    rc = memorandum->Prev ();
  else if (button == firstBtn)
    rc = memorandum->First ();
  else
    rc = memorandum->Last ();
  if(rc == XB_NO_ERROR)
     Read();
  sprintf(buff,"Moving - %s",VDKXError(rc) );
  statusbar->Push(buff);
  return true;
 }
```

- select updateBtn and connect with clicked signal

Here the code for response method:

```
//signal response method
bool
YamForm::OnupdateBtnClick(VDKObject* sender)
{
  xbShort rc;
  if(!memorandum)
    return true;
  Write();
  rc = memorandum->PutRecord();
  sprintf(buff,"Updating record - %s",VDKXError(rc));
  statusbar->Push(buff);
  return true;
 }
```

- select delBtn and connect with clicked signal

Here the code for response method:

```
bool
YamForm::OndelBtnClick(VDKObject* sender)
{
  xbShort rc;
  if(!memorandum)
    return true;
/* lock the files for our exclusive use */
  memorandum->ExclusiveLock( F_SETLKW );
  rc = memorandum->DeleteRecord();
  sprintf(buff,"Deleting record - %s",VDKXError(rc));
  statusbar->Push(buff);
  // make a move into data file
  if ( (rc = memorandum->Next () ==   XB_NO_ERROR) ||
    (rc = memorandum->Prev () ==   XB_NO_ERROR))
    Read ();
  return true;
 }
```

Remeber that deleting a record does not means a physical removing, but rather the record is marked for deletion and is still present, vdkxdb routines simply ignore it unless you explicitely set `VDKXTable::ShowDeleted` property to true. To phisically remove all records marked for deletion you have to use `VDKXTable::PackDatabase()` inherited from xbDbf xbase class.

- select orderBtn and connect with clicked signal
  the button labled "Order" should switch the index that actually drives datafile beween key1 (FIRSTNAME+LASTNAME) and key2 (BIRTHDATE).

```
//signal response method
bool
YamForm::OnorderBTnClick(VDKObject* sender)
{
   if(!memorandum)
        return true;
    memorandum->Order = memorandum->Order == 0 ? 1 : 0;
    Read();
    VDKXTableIndex* index = memorandum->Index(memorandum->Order);
    if(index)
    {
        sprintf(buff,"Index order on: - %s",(char*) index->Key());
        statusbar->Push(buff);
    }
   return true;
 }
```

Index order is numbered from 0 to <n−1> where <n> is the number of opened indexes, order number is assigned as soon as the index is opened, first one has order 0, second one has order 1 and so on..

Now return to our keylist left unfinished before; we want to remove it and substitute it with a three−columns data aware list.

- select keylist and remove it
- substitute it with a three columns vvdxdblist
- restore its name to keylist
- resize to enlarge a bit
- set autoresize to true
- click on column headers and assign them respectively to:
  - FIRSTNAME,LASTNAME and BIRTHDATE fields
  - change column name to: "First name", "Last name" and "Birth date" respectively.

VDKXdbCustomList needs a few words of explanation:

- is a read−only data widget, you cannot Write() on it, it's use is typically to easily browse a database.
- reading operations are typically Θ(n) where n is database table size in rows, it's use should be carefully planned, it should be loaded, invoking Read(), at the beginning and whenever something changes on database such as adding/deleting/updating record operations or changing database ordering.

So what we have to do is to slightly modify our code to have keylist loaded at the beginning and whenever there is a change (such as adding a record). (left as exercise to the reader).

More interesting is to connect keylist with user selection so he/she will be able to see all records data without use Next/Prev buttons. This is a typical user operation: browsing on keys he/she can quickly access the other data for that key.

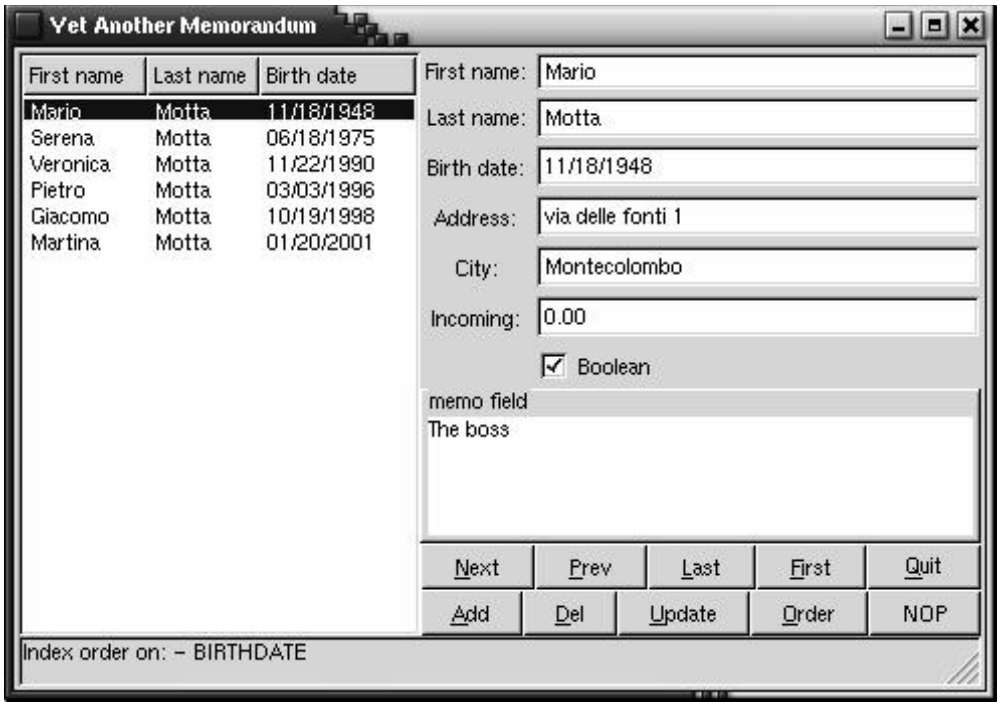
- select keylist and connect with "Select row" signal

here the code for response method:

```
//signal response method
bool
YamForm::OnkeylistSelectRow(VDKObject* sender)
{
  VDKString firstname,lastname,birthdate,key;
  Tuple tuple;
  int ndx = keylist->Selected.Row();
  // get table
  if( (ndx < 0) || (! memorandum))
        return true;
  // get selected tuple
  tuple =  keylist->Tuples[ndx];
```

```
  // load keys from tuple pending on index order
  if(memorandum->Order == 0)
  {
    firstname = tuple[0];
    lastname = tuple[1];
  }
  else if(memorandum->Order == 1)
    birthdate = tuple[2];
  // make key (either firstname or birthdate must be null)
  if(!firstname.isNull())
    key = firstname+lastname;
  else if(!birthdate.isNull())
  {
    calendardate d = (char*) birthdate;
    key = d.AsString();
  }
  // find key
  if(!key.isNull())
  {
    VDKXTableIndex* index = memorandum->Index(memorandum->Order);
    if(index)
    {
      xbShort rc = index->FindKey((char*) key);
      if(rc == XB_FOUND)
        Read();
    }
  }
 return true;
 }
```

Here what could be a final result:



APPENDIX F vdksdl library
USING VDKBUILDER WITH VDKSDL LIBRARY

VDKSdl is a wrapper on SDL (Simple Direct media Layer) a powerful library mainly used for constructing games, it handles very well both graphics and sounds. You can visit http://www.libsdl.org/ to learn more about SDL. VDKSdl provides two subsystemS (video and sound) and a VDKSdlCanvas widget is used for drawing operations.

How to build VDKSdl library
You need both SDL and SDL_image libraries installed, you can download them at http://www.libsdl.org/ and http://www.libsdl.org/libraries.html respectively, at the time of this writing latest version of SDL is 1.2.0 and SDL_image is 1.2.0.

- download vdk−sdl−0.1.0.tar.gz from builder site (or newer version if available)
- unpack:
  - `$ tar xvzf vdksdl−0.1.0.tar.gz`
- `$ cd vdksdl−0.1.0`
- `$ . /configure [your options]`
- `$ make`
- `# make install`

It's helpful to take a look to /vdksdl−0.0.1/tests/testvdksdl program.

Some general informations
Using SDL with VDK requires an hack in setting some environment variable, this leads into a drawback: you can have just one VDKSdlCanvas operating for each task . For this reason using VDKSdlCanvas should be limited to those applications such as games, demos, presentations or whatever that require only a single VDKSdlCanvas. On the other hand VDKSdlCanvas is easy to use, powerful, flexible and an order of magnitude faster than other drawing widgets.
Using VDKSdl library with builder requires:
- to use a place holder widget in GUI designer
- to edit some fields in project options in order to have the correct include paths and linking flags for both VDKSdl, SDL and SDL_image libraries.

The Sdl project
Let's begin with our project, we will take many ideas from the Draw project seen before using a placeholder in place of VDKCanvas.
- make a new project naming it "sdl"
- insert a vbox into the form. (vbox1)
- insert into vbox1 a vertical box (vbox2) and an horizontal box (hbox1)[17]
- Set vbox2 border to 5 pixel
- Select hBox1 and uncheck both "fill" and "expand" check boxes, click on "Repack" button. Notice that selecting a geometry property to a container will automatically apply also to contained widgets if any.
- Insert into hbox1 4 buttons
- Select from "Misc" tool palette a placeholder and drop it into vbox1
- Select placeholder and name it simply canvas.
  - Select "NormalBackground" to white
  - Edit "Def constructor" field to read : `VDKSdlCanvas(this)`
  - Click on Def constructor" button.

Now lets use Project−>Options menu editing:
- change "Include Paths" field to read:
  `−I./ −I<prefix>/include/vdksdl −I<prefix>/include/SDL`,
  where <prefix> is where you installed VDKSdl, SDL and SDL_image libraries (most likely /usr or /usr/local)
- change "Shared libs" field to read:
  `'vdk−config−2 −−libs' −lvdksdl −lSDL −lSDL_image`
  to keep the compiler and preprocessor informed of where to search for vdksdl include files and the dynamic loader where libvdksdl, libSDL and libSDL_image are.
  Also a bit of coding is necessary, just add to sdl.h a line including vdk_sdlcanvas.h:

17 Since object name are auto−numbered by default may be names won't match, isn't a problem however.

```
// vdk support
#include <vdk/vdk.h>
#include <vdk_sdlcanvas.h>
// Sdl FORM  CLASS
//..
```

Building and running sdl program should look like this:

Initializing VDKSdlCanvas

SDL canvas initialization is a bit more complicated than a normal widget, as said before we implemented a SDL surface in a GTK+ widget, this required a little hacking to avoid the two toolkits fighting over which would be drawing on the same surface. SDL provides a video subsystem that lets you to draw on a "surface", in VDKSdlCanvas <Video> is a public class member which has Surface() as a public method allowing you to access the underlying video subsystem and associated surface. However it should be rare for you to need such low level access even if a knowledge of how SDL works could be of help. The first thing to take into account is that initializing signals are handled by VDK on its own thus you have to connect with them using dynamic signal system and specify also that native gtk+ system must be ignored. VDKSdlCanvas emits these signals:

- "*sdl_initialized*"
  Emitted when SDL Video subsystem has been initialized and a valid surface constructed.
- "*sdl_first_update*"
  Emitted soon after above signal, at this time VDKSdlCanvas is shown for the first time.
- "*sdl_update*"
  Emitted whenever VDKSdlCanvas needs to be redrawn (ie: windows was resized by user)
  those signals are the ones that you have to deal with when initializing your canvas and making any necessary updates.

A bit of design here: at initializing we will provide a logo that runs from left to right until reaches the center of the canvas. We get the logo from <where−you−put−builder>/vdkbuilder−2.0.0/example/hello/vdklogo.png and copy it into sdl project directory. In order to mantain a copy of logo image in memory we declare a SDLForm class member.

So edit sdl.h to read:

```
// Sdl FORM  CLASS
class SdlForm: public VDKForm
{
// gui object declarations
private:
  // gui object declarations
  void GUISetup(void);
  SDL_surface* logo_image;
public:
//....
```

we plan to load image into memory as soon as we receive sdl_initialized signal and destroy it when form gets destroyed. We connect with sdl_initialized signal during form setup.

So edit sdl.h to read:

```
class SdlForm: public VDKForm
{
// gui object declarations
private:
  // gui object declarations
  void GUISetup(void);
  SDL_Surface* logo_image;
  bool OnSDLInit (VDKObject* sender);
public:
//...
```

and sdl.cc to read:

```
/*
main form destructor
*/
SdlForm::~SdlForm()
{
  if (logo_image)
    {
    printf ("\nfreeing logo_image");
    SDL_FreeSurface(logo_image);
    }
}
/*
main form setup
*/
void
SdlForm::Setup(void)
{
  GUISetup();
```

```
    /* note the last argument set to false, this tells vdk to override gtk+ native signal, since
    this signal will be handled by vdk.
    */
```

```
  SignalConnect(canvas,"sdl_initialized",&SdlForm::OnSDLInit,false);
}
/*
 answers to sdl initialization message
*/
bool
SdlForm::OnSDLInit (VDKObject* sender)
{
  printf("\nSDL initialized");
  // get logo image into logo_image buffer
  // (NULL on failure)
  logo_image = canvas->GetImage("vdklogo.png");
  if (logo_image)
    printf("\nlogo_image got in memory");
  fflush(stdout);
  return true;
  }
```

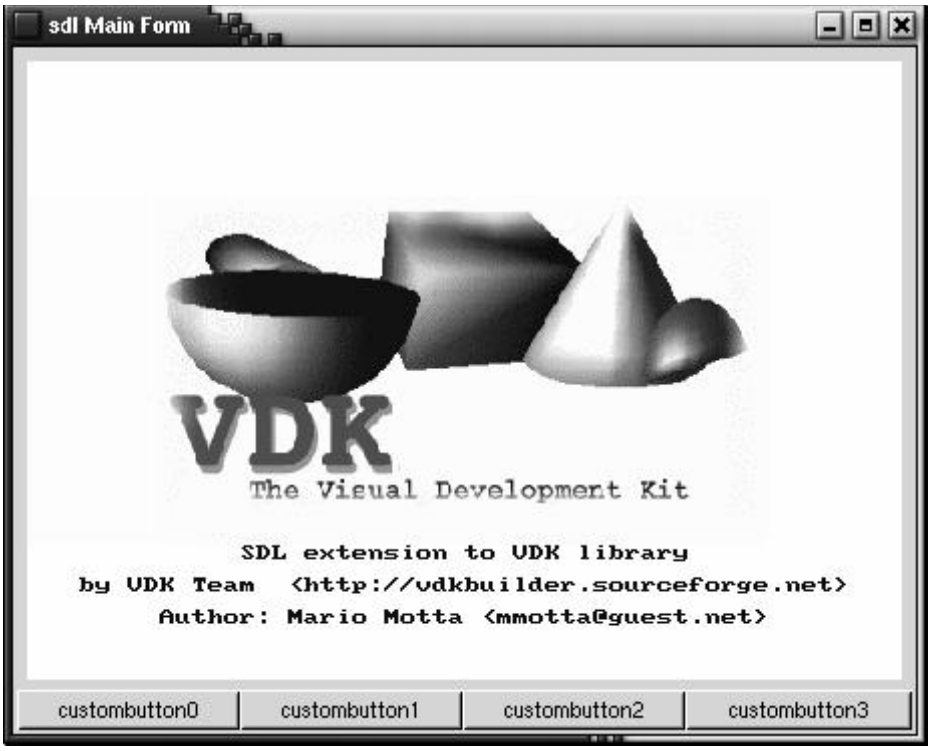
Now let's first concentrate on the form's initial display, where vdk logo will be displayed with an animation. Let's first connect with update signal:

```
void
SdlForm::Setup(void)
{
 //...
 gtk_window_set_resizable (GTK_WINDOW (WrappedWidget ()), false);
 SignalConnect(canvas,"sdl_initialized",&SdlForm::OnSDLInit,false);
 SignalConnect(canvas,"sdl_first_update",
            &SdlForm::OnSDLFirstUpdate,false);
}
```

and write the code for the response method:

```
/*
  Receives first updating message when window is showed
 */
#define OFFSET 3
#define FONT_WIDTH 8
#define FONT_HEIGHT 8
static  char* text[] =  {
"SDL extension to VDK library",
"by VDK Team  <http://vdkbuilder.sourceforge.net>",
"Author: Mario Motta <mmotta@guest.net>",   NULL
};

bool
SdlForm::OnSDLFirstUpdate(VDKObject* sender)
{
  printf("\nVideo subsystem first update");
  fflush(stdout);
  canvas->DrawBackground(clWhite);
  SDL_Delay(100);
      /* writes text,  TIP: always draw string first, otherwise you can experience some weird
      effects
      */
  for(int z = 0; text[z]; z++)
    {
      int len = strlen(text[z]) * FONT_WIDTH;
      int t_center = (canvas->Surface()->w - len)/2;
      canvas->DrawString(text[z],
       t_center,
       canvas->Surface()->h/2+(logo_image->h/2) + (z * FONT_HEIGHT*2),
       VDKRgb("navy blue"));
    }
  if(logo_image)
    {
   // transfers logo into screen moving right wise until center of screen
      int center = (canvas->Surface()->w - logo_image->w)/2;
      int y = (canvas->Surface()->h - logo_image->h)/2;
      for(int j = 0; j <= center; j+=OFFSET)
      {
        int x = j;
        canvas->BlitSurface(logo_image,x,y);
        {
       // erase image tail (OFFSET wide)
        SDL_Rect shadow;
        shadow.x = x; shadow.y = y;
        shadow.w = OFFSET; shadow.h = logo_image->h;
        canvas->DrawRect(x,y,OFFSET,logo_image->h,clWhite);
        canvas->UpdateRects(1,&shadow);
        }
      }
    }
  return true;
}
```

Don't forget to declare: `bool OnSDLFirstUpdate(VDKObject* sender);` in sdl.h
Building and running the project should show something like this (at the end of animation).



As we said before we set the form in such way that can't be resized using the native GTK+ call `gtk_window_set_resizable(GTK_WINDOW (WrappedWidget()),false);` this was for the sake of simplicity, otherwise in order to discuss an analogy with the draw project we would have had to connect also with "sdl_update" signal and correctly handle a form resize.

Go on with sdl application

let's continue with sdl application:

- select custombutton0 and rename it as sinBtn and its caption to "_Sin"
- connect it with clicked signal

now let's code some utility functions such as computing radians −> degrees function and having a double value point class (as said before we took many ideas from "draw" project seen before).
Edit sdl.h to read:

```
#include <vdk_sdlcanvas.h>
class DPoint
{
  public:
  double x,  y;
  DPoint (double x = 0.0,  double y = 0.0):x (x), y (y) {}
  ~DPoint () {}
  bool operator == (DPoint& d) { return x ==  d.x; }
  bool operator < (DPoint& d) { return x <  d.x; }
};
typedef VDKArray <DPoint> PointArray;
// Sdl FORM  CLASS
// ..
```

and sdl.cc to read:

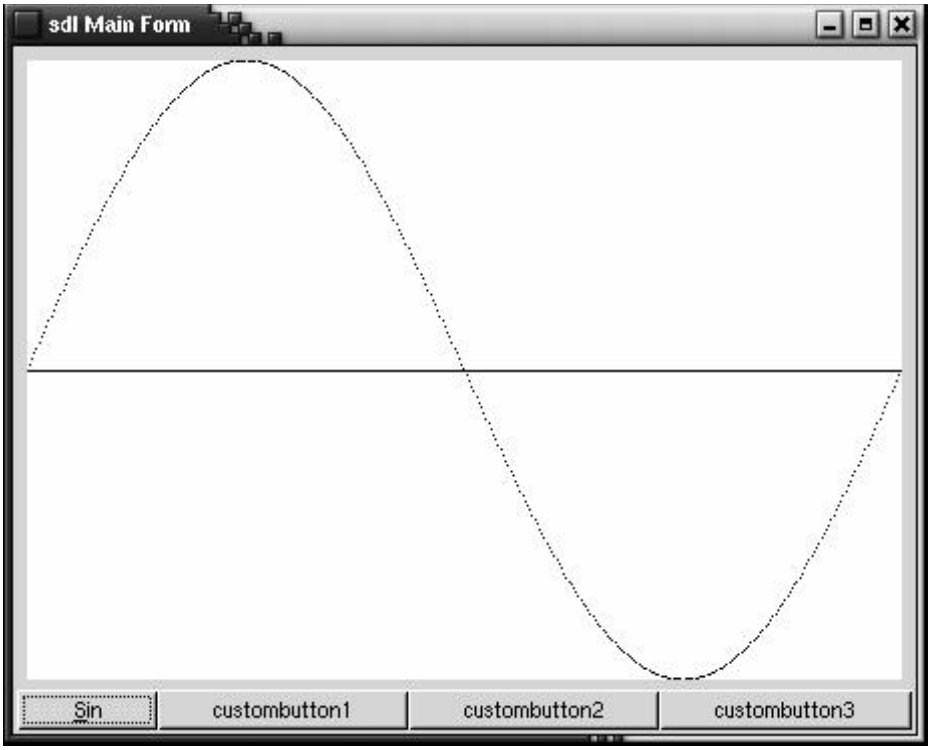
```
#include <sdl.h>
#include <math.h>
#ifndef PI
#define PI 3.14159265358979323846
#endif
inline double deg2rad(double d) {  return d*PI/180.0; }
```

Here the code for sinBtn clicked signal response:

```
bool
SdlForm::OnsinBtnClick(VDKObject* sender)
{
 DPoint size(canvas->Surface()->w,canvas->Surface()->h);
```

```
// draw all surface (clears it)
canvas->DrawBackground(clWhite);
// draw x axis
canvas->DrawLine(0,    // origin x
     (int) size.y/2,   // origin y
     (int) size.x,     // end x
     (int) size.y/2,   // end y
     clBlack);         // color
// compute x step to bound 0 - 360°
double xstep =  size.x/360.0;
// fill array with scaled values
PointArray pArray(360);
int z;
for ( z = 0; z < 360; z++)
  {
    double x = z*xstep; // x values are degrees
    pArray[z] = DPoint (x, size.y/2-((size.y/2)*sin (deg2rad (z))));
  }
 // plots all computed points
for ( z = 0; z < pArray.size (); z++)
   canvas->PutPixel ((int) pArray[z].x, (int) pArray[z].y, clBlack);
// update all surface to see the result
canvas->UpdateRect ();
return true;
}
```

and here the result after have built and run sdl application:



Last step left we are done is playing a .wav file using VDKSdl.

- Select custombutton1, name it palyBtn and set its captio to "_Play"
- connect it with clicked signal.

Before showing the responde method code let's construct and initialize a sound subsystem using VDKSdl, so modify sdl.cc to read:

```
//..
inline double deg2rad(double d)
{     return d*PI/180.0; }
static VDKSdlAudio* audio; // audio subsystem address
//..
/*
```

```
main form constructor
*/
SdlForm::SdlForm(VDKApplication* app, char* title):
      VDKForm(app,title,v_box,DisplayType)
{
  audio = new VDKSdlAudio(this);
}
/* main form destructor */
SdlForm::~SdlForm()
{
  if (logo_image)
    SDL_FreeSurface(logo_image);
  if(audio)
    delete audio;
}
```

here we use a VDKThread to play a .wav file, copying it from <where−builder−is>/vdkbuilder−2.0.0/example/hello/wav.wav to sdl project directory. Using a thread let us play wav file asyncronously without stopping main thread. To avoid intereferences we deny overlapped playings.

```
//..
//signal response method
static VDKThread *play_thread;
static void* Play (void);
static bool can_play = true;
bool
SdlForm::OnplayBtnClick(VDKObject* sender)
{
  // play sound into a thread,  plotting can be made  asincronously
  if (can_play)
    {
    play_thread = new VDKThread;
    play_thread->Start( (void*) Play);
    can_play = false;
    }
  return true;
 }
/*   Asincronously play a sound */
void * Play(void)
{
  if(audio)
    {
     audio->PlayWAV("wav.wav");
     can_play = true;
     if(play_thread)
       delete play_thread;
    }
  return NULL;
}
```

intentionally left blank page

Appendix G

VDK Input Tutorial

Jonathan Hudson

jonathan@daria.co.uk

VDKInputChannel is a wrapper for gdk_input_add(). This document provides tutorial material on using VDKInputChannel.

1. Introduction

VDKInputChannel allows you to monitor input from one or more file descriptors within a VDK application and define member function(s) to be called when some activity is detected on that file descriptor. It is particularly useful for pipes, sockets or serial devices.

> **Note:** In this tutorial we talk about Unix signals (man signal) and VDK Signals. Using the same name *signal* for these two different entities may be confusing, take care!

This version expands on the example distributed with VDK 2.0 under

vdk−2.0.0/example/iotut

1.1. Copyright Information

1.2. Disclaimer

No liability for the contents of this documents can be accepted. Use the concepts, examples and other content at your own risk. As this is a new edition of this document, there may be errors and inaccuracies, that may of course be damaging to your system. Proceed with caution, and although this is highly unlikely, the author(s) do not take any responsibility for that.

2. What you (might) need to know

Although VDKInputChannel is primarily concerned with enabling VDK programs to react to input on a Unix file descriptor as part of the normal VDK event loop, it is necessary to be aware of some possibly more advanced Unix programming topics. These topics include:

- Child processes, creation and management;
- Unix signals;
- Inter−process communications (pipes, sockets).

This tutorial is not primarily concerned with how these techniques are used, but it does attempt to point out any pitfalls in using these techniques in VDK programs.

3. Class Definitions

The class defines a constructor, a destructor, and two public methods. It is necessary to include the vdkinchannel.h header file before using this class.

VDKInputChannel may be used with either static or dynamic VDK Signals. The constructors wrap gdk_input_add(). The destructor wraps `gdk_input_remove()`.

3.1. Constructor

```
VDKInputChannel (VDKForm* obj, int fd, GdkInputCondition
condition=GDK_INPUT_READ );
```

obj: is the parent object
fd: is the file descriptor to be monitored
condition: is a GdkInputCondition (default = GDK_INPUT_READ)

3.2. Destructor

```
virtual ~VDKInputChannel (void);
```

3.3. Member Functions

The following member functions are defined.

```
int getfd (void);
```

Returns the file descriptor associated with the object.

```
int getcondition (void);
```

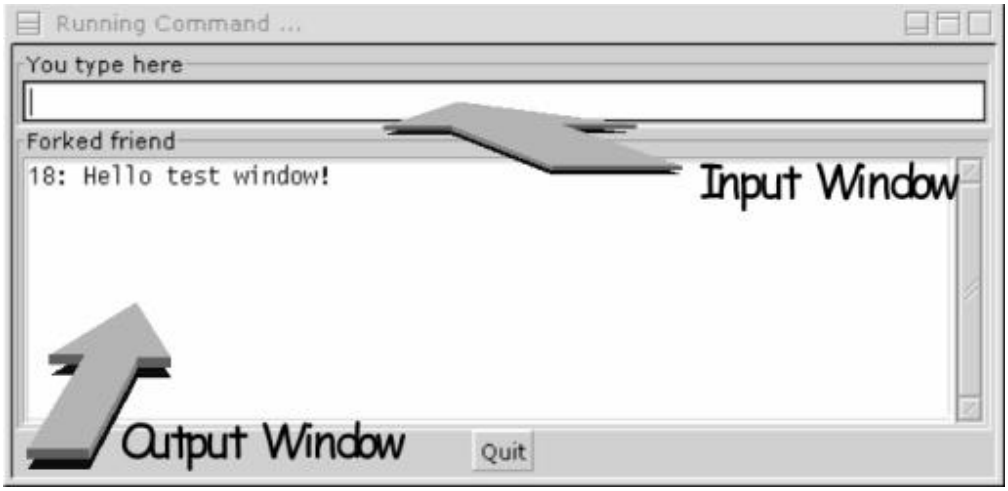
Returns the condition that caused the VDK Signal to be fired.

> **Note:** VDKInputChannel defines its own VDK Signal, INPUT_SIGNAL, in vdkinchannel.h, as (user_signal−1024); it also uses a named signal, "input_signal".

4. VDKInputChannel Tutuorial

In order to demonstrate the use of VDKInputChannel, as well as some techniques for using child processes with VDK, and catching Unix signals in C++, a number of example programs are provided.

All the examples have a single theme; there is a window with two fields (and a Quit button).



In every case, we have a demo (parent) program, into which you can enter some text, and a helper (child) program that returns the text to you.

A line typed in the *Input* window is sent to the *helper* process, which will just send it back to the parent program, where it is displayed in the *Output* window.

VDKInputChannel is used to notify our application when the child sends data back to you. It uses a VDK Signal, so the VDK event loop manages the invocation of the input handler.

> **Note:** In all examples, it is assumed that the test programs are on your path, if they are not, you will have to give some path specification:

```
e.g. $ ./iodemo0 ./helper0
```

4.1. Walk through iodemo0

The complete suite of examples is described in Section 4.2

Before we get to deeply into the example, you should be aware that the author is a somewhat old 'C' programmer who doesn't really like or understand C++. If this looks more like "*C with a few classes*" rather than real C++, then you know why.

The iodemo0.cc example includes the iodemo0.h header file. This, in turn, includes vdkinchannel.h. Once VDKInputChannel becomes an integrated VDK class, then you'll need <vdk/vdkinchannel.h>.

Of particular note in iodemo0.h is the section in our MyApp class:

```
class MyApp: public VDKApplication
{
  protected:
    static void reaper(int);   (1)
    static void plumber(int);  (1)
    static MyApp *me;          (2)
```

(1) We define two static member functions that are used as Unix signal handlers. How we define and use these functions is described in the MyApp::Setup , MyApp::reaper and MyApp::plumber sections.

(2) This will be used for a pointer to the MyApp class, in order that we can use member functions in Unix signal handlers.

In order to be able to handle the Unix signals that we need to catch while using a child process, and have access to our classes in the signal handlers, it is necessary to declare the signal handlers as static member functions and to have a static pointer to our application class.

At this stage we should also remind ourselves that the signal handling in this example is naïve, in a real application we might want to handle signal differently. We are (here) mainly interested in SIGCHLD to tell us when the child process has terminated.

Have a look at function MyApp::Setup() where we set up the signal handlers.

```
MyApp::me = this;       // (1)
struct sigaction sac;
sigemptyset(&(sac.sa_mask));
sac.sa_flags=0;
sac.sa_handler = MyApp::reaper; // (2)
sigaction(SIGCHLD, &sac, NULL); // (3)
for(int i =  0; i < NSIG; i++)
{
    // (4)
    if(i != SIGQUIT && i != SIGKILL && i != SIGTERM  &&
            i != SIGCHLD &&
       i != SIGSTOP && i != SIGCONT && i != SIGTSTP)
    {
        sigemptyset(&(sac.sa_mask));
        sac.sa_flags=0;
        sac.sa_handler = MyApp::plumber; // (5)
```

```
            sigaction(i, &sac, NULL);    // (6)
        }
    }
```

(1) Here we set up the pointer to our application class. This will allow us to access member functions in the signal handlers.

(2) We defined the static member function MyApp:reaper to catch child termination. We examine what this function must do in *ccurldlfflthe MyApp::reaper description.

(3) We use the function sigaction to connect the reception of SIGCHLD to our signal handler.

(4) The function MyApp::plumber (see Note 1) will be used for many other signals; however there are some we use the default action.

(5) We defined the static member function MyApp:plumber to catch these miscellaneous signals. We examine what this function must do in MyApp::plumber description.

(6) We use the function sigaction to connect the reception of the signals to our signal handler.

Now let's look at the VDK Signal definitions we need to support our application:

```
    DEFINE_SIGNAL_MAP(MyForm,VDKForm)
      ON_SIGNAL(inp1,INPUT_SIGNAL,DoIO),        // (1)
      ON_SIGNAL(text, realize_signal,runcmd),   // (2)
      ON_SIGNAL(entry, activate_signal,sendstuff), // (3)
      ON_SIGNAL(timer,timer_tick_signal,Quit),   // (4)
      ON_SIGNAL(quit,clicked_signal,Quit)        // (5)
    END_SIGNAL_MAP
```

(1) We use the default VDKInputChannel signal INPUT_SIGNAL to call the myForm::DOIO (the inp1 object is created in the MyForm::runcmd function. We'll look at this function in some detail *ccurldlfflater on .

(2) When our text widget is realised, it starts up the child process, via the MyForm::runcmd. See below for more detail.

(3) When we enter a text string in the input window, this VDK Signal definition will invoke the MyForm::sendstuff to send the text to the child process.

(4) We use a VDK Timer, so we need a VDK Signal definition.

(5) This is a standard VDK button Signal.

The only thing of note in MyForm::Setup is setting a user defined VDK Signal. This is largely just to demonstrate user defined VDK Signals, rather than of necessity.

The setup of the windows is a trivial VDL application, the real work starts when the text window is realised and the MyForm::runcmd function is invoked.

```
bool MyForm::runcmd(VDKObject *obj)
{
  MyApp *a = static_cast<MyApp *>(Application()); // (1)
  int inpipe[2], outpipe[2];

  if(access (a->args[0], X_OK) == 0)
                    (2)
  {
    pipe(inpipe);   // (3)
    pipe(outpipe);  // (3)
    pid = fork();   // (4)
    switch(pid)
    {
        // Child
      case 0:
        dup2(inpipe[0], 0);  // (5)
        dup2(outpipe[1], 1); // (5)
```

```
        dup2(outpipe[1], 2); // (5)
        execvp(a->args[0], a->args);  // (6)
        break;
      case -1:
        break;                // (7)
      default:
        // Parent
        close(outpipe[1]);// Important! Close my end here(8)
        close(inpipe[0]); // Important! Close my end here(8)
        xmit = inpipe[1]; // (9)
        inp1 = new VDKInputChannel(this,outpipe[0]); // (10)
        text->Clear();
        break;
    }
  }
  else
  {
    char x[256];
    sprintf(x, "Can't execute your program \"%s\"",
            (a->args[0]) ? a->args[0] : "????");
    text->TextInsert(x); //(11)
    SignalEmit("KillSignal"); //(11)
  }
  return true;
}
```

(1) The helper program name was supplied as an argument to the demonstration program, we need MyApp to retrieve this information.

(2) We also test that we can run the helper program!

(3) We need a pair of Unix pipes to communicate with the helper program.

(4) We call fork to create a child process. The return value is evaluated in both the parent (where it is a positive pid (process ID), and in the child, where it is 0.

(5) In the child, we duplicate the pipe file descriptors as standard input, output and error streams.

(6) Having set up the file descriptors, we can now exec the child program.

(7) If the fork failed, we catch it here.

(8) In the parent, we *must* close the end of the pipe that we're not using.

(9) This is the file descriptor that we will write on (i.e. the text typed in the input window), we have parent inpipe[1] -> child inpipe[0] (stdin).

(10) We create an instance the the VDKInputChannel. The parameter is the channel (file descriptor) we listen on. Here we have child outpipe[1] -> parent outpipe[0]. Note the symmetry with normal Unix usage, 0 is input, 1 is output.

(11) If the fork fails, then we display some error text and issue a VDK Signal.

Once the text has been entered in the input window, it gets sent, via a pipe to the helper process from the MyForm::sendstuff. This is pretty trivial stuff.

The child process is just going to echo the text back at us, and when this happens, the MyForm::DoIO will be invoked.

```
bool MyForm::DoIO (VDKObject *obj)
{
  VDKInputChannel *ip = static_cast<VDKInputChannel*>(obj);(1)
  int res;
  char buf[1024];
  res = read (ip->getfd(), buf, sizeof(buf)); // (2)
  if(res > 0)
  {
    text->TextInsert(buf, res); // (3)
  }
  else if(res == 0)
```

```
  {
    close(ip->getfd()); //(4)
    ShowInfanticide(); // (4)
    ip->Destroy(); // (4)
  }
  return true;
}
```

(1) We have to cast the callback object into a VDKInputChannel object.

(2) We can do a standard Unix read on the channel (which we get from a member function). We must check the return value.

(3) If the result is positive, then we got data. As it will not be NUL terminated, we supply the length to text->TextInsert().

(4) If the read indicates EOF, we *must* close the channel and destroy the class. If we don't, the MyForm::DOIO function will be called repeatedly.

The Unix signal handler MyApp::reaper will be invoked when the child process exits.

```
void MyApp::reaper(int) // (1)
{
  int sts;
  int pid;
  pid = waitpid(-1, &sts, WNOHANG); //(2)
  if(pid == ((MyForm*)MyApp::me->MainForm)->Pid()) // (3)
  {
    cerr << "-- child dies, pid was " << pid << ", status " <<
      sts << endl; // (4)
    ((MyForm*)MyApp::me->MainForm)->Pid(0); //(5)
  }
}
```

(1) The parameter (which we don't use) is the signal number.

(2) We must call wait or waitpid after the child exits. We also should *NOT do work (or I/O) in a signal handler*, so this is a bad example!

(3) If this is the pid that we established in MyForm::runcmd, then we know it's our child and we clean up.

(4) Display the pid of the affected child process.

(5) And set the pid member variable to zero to indicate there is no longer a child process.

The Unix signal handler MyApp::plumber will be invoked when the Unix signals other than SIGCHLD are caught.

```
void MyApp::plumber(int sig) // (1)
{
  cerr << "Otherwise unhandled Unix Signal " << sig << endl;(2)
}
```

(1) Here we use the signal number parameter.

(2) And merely display it.

4.2. The examples

4.2.1. helper0

helper0 is a simple 'C' program that receives data on STDIN and returns the data on STDOUT, preceded by a character count.

For example. if you type **hello** (and **LF**) on STDIN, it will echo "5: hello" on STDOUT.

If you send it **quit** it will echo "4: quit" and then terminate.

4.2.2. iodemo0

iodemo0 requires a single parameter, the name of its helper program, which in this example is **helper0**.

```
usage: $ iodemo0 helper0
```

The program creates anonymous pipes and then forks and execs the helper program.

Any lines you type are piped to the helper program, which then pipes a character count and the line back again.

If you send 'quit', then **helper0** will exit, and you will see the termination caught by the SIGCHLD (unix) signal handler, and EOF on the pipe.

The program will wait 30 seconds and then exit (unless you press the quit button first).

4.2.3. iodemo1

```
usage: $ iodemo1 helper0
```

iodemo1 is pretty much the same as **iodemo0**, however, rather than using pipes, we use anonymous sockets, using the libc socketpair(2) function. This is slightly neater then using pipe(2), as we don't end up with 'redundant' pipes. In the this example, we use SOCK_STREAM (reliable byte stream) sockets. This example behaves like example **iodemo0**.

4.2.4. iodemo1a

```
usage: $ iodemo1a helper0
```

We change iodemo1 to use SOCK_DGRAM (datagram) sockets. This behaves differently! The EOF on is not seen on the input side when the client closes (reasons ?).

We also use a non−default static VDK Signal number.

Goto the line:

```
    if(res <= 0 /*|| strncasecmp(buf, "4: quit\n", 5) == 0*/)
```

and remove the comment. Recompile ... use like **iodemo0** again.

4.2.5. iodemo1b

We change iodemo1a.cc to catch the signal SIGUSR1 and arrange for the new helper program **helper1** (from helper1.c) to send its parent SIGUSR1 on exit. This causes **iodemo1b** to go into its tidy up mode.

We also illustrate using a dynamic Signal for VDKInputChannel. This illustrates how flexible the VDK Signal handling is; we can now create VDKInputChannels as we require, without having to have created a static Signal table entry before hand.

4.2.6. iodemo2

iodemo2 works in a different way, in that it uses a new Socket class and a 'named' socket. **iodemo2** does not start the **helper** program (as the **helper** program may now reside on a different machine).

Warning

Socket:: is a very trivial class, don't use this for anything serious!

You should also run these examples within the security constraints of your environment.

```
usage: $ iodemo2 tcp|udp host:port|filename
```

The program can use either tcp or udp type sockets and either internet AF_INET or unix AF_UNIX domain sockets. Please refer to a good reference (GNU libc 'Info' or Stevens "Unix Network Programming") if you need more information on socket communications.

Where AF_INET sockets are used, the second parameter is a hostname and a port number or name, for example:

```
iodemo2 tcp trespassersw.daria.co.uk:7
iodemo2 udp localhost:echo
iodemo2 udp kanga.daria.co.uk:iodemo
```

At my home, the first two examples would use the echo server on the same machine, one using tcp and the other udp; I could run the first command from any other machine on my LAN. The echo service (port 7) is a standard inetd /etc/services port and provides just the 'echo' service we need. You should check that /etc/services defines the following lines:

```
echo            7/tcp
echo            7/udp
```

and `inetd.conf`:

```
echo    stream  tcp     nowait  root    internal
echo    dgram   udp     wait    root    internal
```

In the final case **`udp kanga.daria.co.uk:iodemo`**, we have defined a bespoke service, and we will need an echo type server listening on it.

You could define the iodemo service in /etc/services, using a spare port number, for example:

```
iodemo          12345/tcp
iodemo          12345/udp
```

To try this, I've included the perl iodemo−inet.pl program, run this as:

```
iodemo−inet.pl
```

iodemo−inet.pl assumes a symbolic port name of iodemo unless you give it a port number (or name) on the command line.

and start as many iodemo2 programs as we like:

```
iodemo2 tcp hostname:iodemo
```

where hostname is the machine running iodemo−inet.pl'.

or

```
iodemo−inet.pl 24680
```

with

```
iodemo2 tcp hostname:24680
```

To use an Unix domain server, we again need to write one first, I've included a very simple example as iodemo−server.pl. For example:

```
iodemo−unix.pl /tmp/.iodemo
iodemo2 tcp /tmp/.iodemo
```

The iodemo−unix.pl server only understands tcp sockets! It can also handle multiple connections.

In gnome−terminal 1

```
iodemo−unix.pl /tmp/.iodemo
```

In gnome−terminal 2

```
iodemo2   tcp /tmp/.iodemo
```

In gnome−terminal 3

```
iodemo2   tcp /tmp/.iodemo
```

In gnome−terminal n

```
iodemo2   tcp /tmp/.iodemo
```

4.3. Other techniques

This is not an exhaustive set of examples; it does not consider named pipes mkfifo(2), which might be used on a single machine in the manner of **iodemo2**, or SysV IPC msg functions.

Hope this helps someone.

5. (Un)frequently asked questions

If anyone were to ask any questions, they might conceiveably be answered here.

Q: Why do I end up with zombie [defunct] child processes ?

A: You are not waiting for your child process(es) to finish.

Q: How can I handle asynchronous input from muliple sources.

A: The VDK Signal system is not only one−to−one but one−to−many and many−to−one, in other word you can connect different objects to the same signal−function slot or viceversa or different object to different signal−function slot, <sender> lets you discover which object is interested in the call.

Q: Sometimes I run a very short child process and it has completed before I've even stored it's pid away. The Unix signal handler then doesn't work properly because it doesn't recognise the terminating program as my child. My application then becomes very confused.

A: This can be a problem, particularly with the Linux 2.4.x kernels with the 'child runs first' fork() policy. You can protect your application from this behaviour using sigprocmask with SIGBLOCK and SIG_UNBLOCK modifiers to delay the reception of SIGCHLD until you know your application is in a state to handle it.

Q: Why isn't there a VDKChildTask class ?

A: One day there may be. However, there are many combinations of how signals should be handled and how channels (file descriptors) should be redirected or inherited inthe child that make this a non−trivial task.

Q: Why don't I get a SIGCHLD when my child process terminates?

A: I don't know! One common cause is failing to close fd 0 (close(0)) when the child does not use stdin.

6. Acknowledgements

My wife, Daria Hudson, kindly lets me use her domain name. She also lets me spend an inordinate amount of time playing with computers.

Mario Motta <mmotta@guest.net> not only wrote *VDK* (http://vdkbuilder.sourceforge.net), he also persuaded me to write this tutorial.

Notes

1. so called because it was used to catch SIGPIPE

intentionally left blank

APPENDIX H
GOING ON WITH JONATHAN HUDSON WORK

J.Hudson did a very good job implementing VDKInputChannel discussed in previous appendix. Starting from his work I have made a new class: **VDKChildTask** that encapsulates the work needed when you want to invoke an external program within a VDK application. Child tasks can be either simple or duplex, meaning that external program sends data (simplex) or sends and receives data to/from its parent task (duplex).

**** WARNING ****

VDKChildTask is a simplified wrapper of fork() and signal(), often it's a more complex work and you need full control over the child environment (fd's and signals particularly). So do not rely completely on this wrapper for critical applications but prefere your own code when using fork() and managing unix signals.

Create and run a child task from a VDK application is straightforward:

- create a VDKChildTask
- connect with child task to receive signals when child sends data or child has finished its work
- run the task passing program arguments to child task
- manage all received data
- eventually send data to child task
- eventually manage child end.

MyGrep project[18]

So let's go on with a project that demonstrates how to use above class.

- Make a new project naming it MyGrep

Once the new MyGrep dir is created copy into this directory following files:

- helper0.c from vdk−2.0.0/example/iotut/
- vdkchildtask.cc from vdk−2.0.0/example/childtask
- vdkchildtask.h from vdk−2.0.0/example/childtask

Make helper0 program:

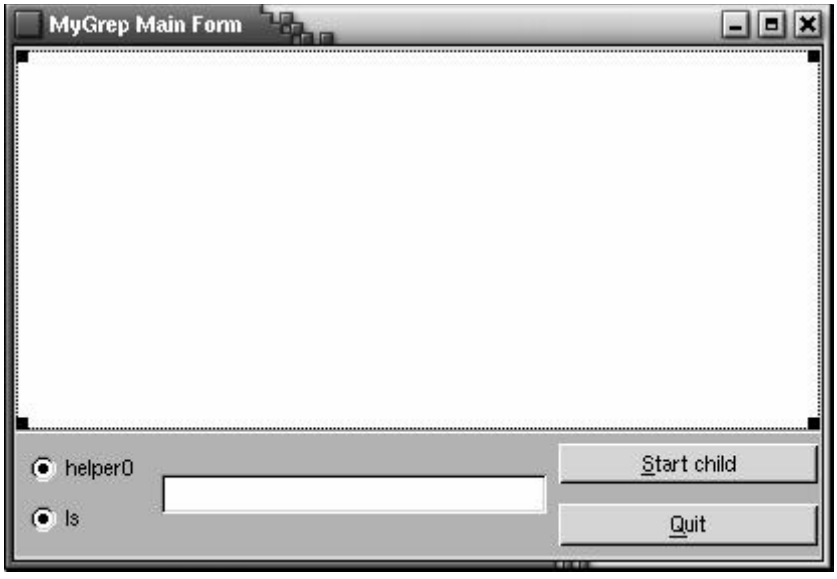
```
$ gcc -o helper0 helper0.c
```

(helper0 program will be used as a duplex child task)

Add vdkchildtask to your project, as you see VDKChildTask is not a published class since is not completely reliable for critical applications as discussed in above warning, i have tested this class under severe conditions and in several applications, under some circumstances it failed.

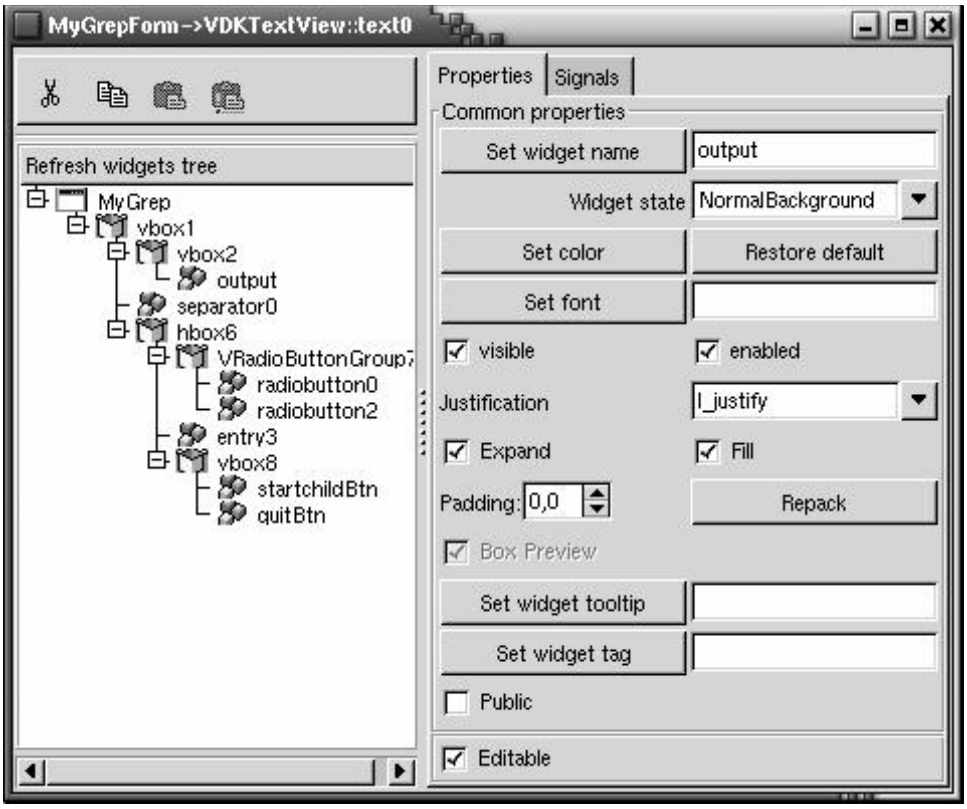
GUI design

Use VDKBuilder GUI editor to have a picture like this:



18 MyGrep name has nothing to do with the program content, just a random name.

Also the widgets tree could be of help:



Once you are done with constructing the interface, try a build and run of the program.

Coding

Now it's time to begin to write our code, first of all let's modify MyGrep.h to declare a child task and a couple of response functions:

```
//...
// vdk support
#include <vdk/vdk.h>
#include "vdkchildtask.h"
// MyGrep FORM  CLASS
class MyGrepForm: public VDKForm
{
// gui object declarations
private:
// gui object declarations
  void GUISetup(void);
  bool OnDataSignal(VDKObject* sender);
  bool OnChildDie(VDKObject* sender);
  VDKChildTask* task;
public:
//...
```

above we declare a VDKChildTask pointer and two response functions to be triggered when child task sends data to our application and/or it dies.

The great part of code goes now into MyGrep.cc, here is how to modify form Setup() in order to make a child task and connect to child signals:

```
/*
main form setup
*/
void
MyGrepForm::Setup(void)
{
      GUISetup();
// put your code below here
```

```
  task = new VDKChildTask (this);
  SignalConnect (task, "child_task_data",
            &MyGrepForm::OnDataSignal, false);
  SignalConnect (task, "child_task_died",
            &MyGrepForm::OnChildDie, false);
}
```

Note that signal connection last arg is false since these signals are handled internally by VDK and gtk+ signal system isn't interested.

And now response function code to handle datareceived from child task:

```
/*
*/
bool
MyGrepForm::OnDataSignal(VDKObject* sender)
{
  VDKChildTask* task = dynamic_cast <VDKChildTask*>(sender);
  if (task)
   {
   output->TextInsert (task->GetData ());
   entry3->Text = "";
   }
  return true;
}
```

Once signal is received we use char* VDKChildTask::GetData() to get data and display them on output board.

We are interested also in receiving a signal when child task ends, here the code that shows a prompt on output board:

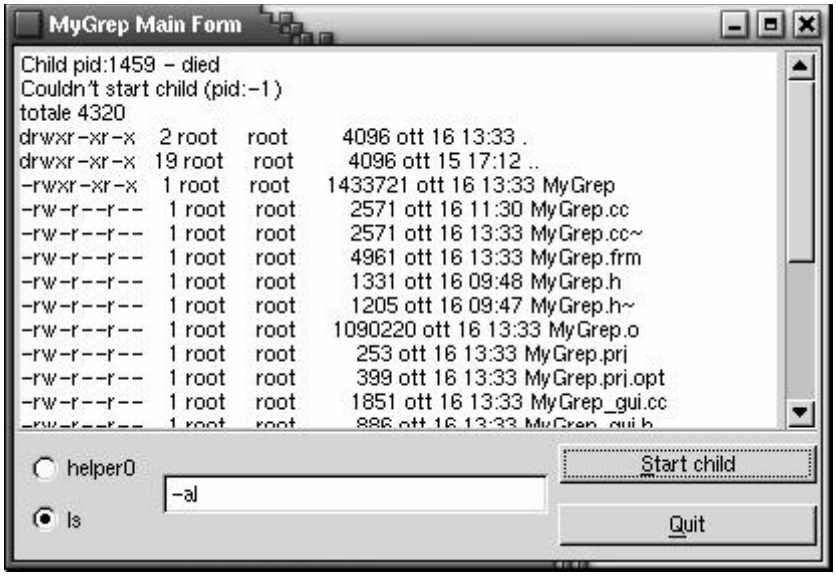
```
/*
*/
bool
MyGrepForm::OnChildDie(VDKObject* sender)
{
  char prompt[64];
  VDKChildTask* t = dynamic_cast <VDKChildTask*>(sender);
  if (t)
    {
    sprintf (prompt, "Child pid:%d - died",t->Pid ());
    output->TextInsert (prompt);
    }
  return true;
  }
```

Now we write the code to activate a simplex child like /bin/ls:

- select startchildBtn and connect it with "clicked" signal
- here the code of response function limited to one of the two radio buttons option:

```
//signal response method
bool
MyGrepForm::OnstartchildBtnClick(VDKObject* sender)
{
  int selected = VRadioButtonGroup7->Selected;
  switch (selected)
    {
      case 0:
        break;
      case 1: // runs ls
        {
          char prompt[64];
          char* match = entry3->Text;
          char** args = new char*[3];
          output->Clear ();
          args[0] = "/bin/ls";
          args[1] = *match ? match : NULL;
          args[2] = NULL;
          pid_t pid = task->StartChild (args);
          if (pid >=  0)
             sprintf (prompt, "Child pid:%d started\n", pid);
          else
             sprintf (prompt, "Couldn't start child (pid:%d)\n", pid);
          output->TextInsert (prompt);
          delete [] args;
        }
        break;
    }
 return true;
 }
```

VDKChildTask::StartChild() method is used to start a child task, it receives <args> as an array of strings, the first one is the task itself, others are task arguments if any, array must be NULL terminated. The method returns a child task id or a negative number if it fails. When ls sends output data to parent MyGrepForm::OnDataSignal() is triggered and ls output is shown on output board.
Below an example using −al as ls argument:



Similarly we can run a child task in duplex mode using helper0 program, we modify above response function writing the code to manage also helper0 selection on radio button group. Helper0 is a simple program that eats a string and echoes it with it's length in characters, entering "quit" makes helper0 terminate. We need also to connect with "activate" signal of entry widget in order to send data to helper0.

```
bool
MyGrepForm::OnstartchildBtnClick(VDKObject* sender)
{
  int selected = VRadioButtonGroup7->Selected;
  switch (selected)
    {
      case 0: // runs helper0
        {
          char prompt[128];
          char *args[] = {"./helper0",NULL};
          output->Clear ();
          pid_t pid = task->StartChild (args, true);
          if (pid >=  0)
              sprintf (prompt,
          "helper0 with pid:%d started\nenter a word in entry widget
and hit enter\n", pid);
          else
             sprintf (prompt, "Couldn't start child (pid:%d)\n", pid);
          output->TextInsert (prompt);
        }
        break;

      case 1: // runs ls
      //...
```

– select entry3 widget and connect it with "activate" signal, this signal is triggered whenever user hits "Enter" on an entry widget.

```
bool
MyGrepForm::Onentry3Activate(VDKObject* sender)
{
 char* text = entry3->Text;
 char* buffer = new char[strlen (text)+2];
 if (*text)
    {
    strcpy (buffer, text);
    char* t = buffer + strlen (buffer);
    *t++ = '\n'; *t = '\0';
    bool result = task->SendData (buffer);
    if (!result)
       output->TextInsert("\nCan't sent data");
    }
 delete[] buffer;
 return true;
 }
```

Here we use VDKChildTask::SendData() method to send entry3 contents to our helper0 program.
To complete the program connect quitBtn to "clicked" signal in order to nicely close the application (left as an exercise to the reader).

Below an example run:
– run the program
– hit Start child button
– enter "querty" into entry3 widget
– hit Enter
– enter "quit" into entry3 widget
– you will see helper0 finish.
– Click on Quit button.

On next page a picture of the example run

VDKChildTask can be a useful class that greatly simplifies fork and pipe stuff, just remember to carefully test your application under several circustances before considering it reliable.