JBS 4.x

Installation & User's Guide

Author: Jean-François GOBBERS (JFGO)

Revision: 1.3

Date: 27-04-2005

© Barco N.V. 2005

Table of Contents

1. Document History

Rev.	Date	Author(s)	Comment
1.0	06-10-2004	JFGO	Initial draft release.
1.1	22-11-2004	JFGO	Synchronized with 4.2.
1.2	26-01-2005	JFGO	First public release.
1.3	27-04-2005	JFGO	Added: - case sensitivity - auto-completion mechanism.

© Barco N.V. 2005

2. Introduction

JBS (*JFG's Building System*) is a command-line oriented, Makefile based, non-intrusive, cross-platform software tool designed to help software developers perform most of the common tasks associated with the development and the maintenance of inter-dependent C/C++ software packages ("projects").

Let's split this definition and explain each part separately.

- the intended users of **JBS** are software developers. Although **JBS** could be used by others to automate some specific tasks, it's not the intended use of **JBS**.
- those developers should be developing inter-dependent C/C++ packages, for example some executables and accompanying dlls. Note that support for additional languages exist or is currently under development (Java, VHDL). Contact the author to get further information (see Annex).
- the core engine of **JBS** is implemented as a set of Makefiles, which are to be interpreted by GNU Make; this application is a command line tool, invoked from a shell (i.e. in a Cygwin window or in a Linux terminal).
- the tasks performed by **JBS** are essentially:
  - compile and link executables and libraries
  - automatically generate documentation of the API
  - keep track of the location of all your projects, and of the dependencies existing between those projects
  - generate archives of those projects.
- **JBS** is cross-platform. It runs at least on Linux and on Windows (with the help of the Cygwin environment in this later case), but should also run on any POSIX-compliant platform.
- **JBS** is non-intrusive, meaning that it doesn't require a long and painful installation process, with many files created everywhere on your disk, or with configuration settings put in random places in the registry (Windows) or in hidden files in your home directory (Linux). See Installation below for more information.

Note that **JBS** is copyrighted software, although it is distributed under the LGPL (see copying.lgpl file, provided with the installation package).

© Barco N.V. 2005

3. Installation

3.1 Where to install ?

JBS works on a per-development tree base, where a development tree is simply a folder that contains all your projects (using the folder hierarchy you want). The common ancestor folder of this development tree is called the "root" folder, and its path will be referred to as the `<JBS-ROOT>` path.

Some notes regarding this root folder:

- a **JBS** root folder is simply a folder containing an empty file named "`.jbs.root`" (note the dots). This file is automatically created when you install JBS in some folder.
- **JBS** can handle more than one development tree at a time (more than one "root" folder). In fact, as has been said earlier, **JBS** is non-intrusive, and it doesn't use any global settings stored in exotic places. Everything **JBS** needs is kept inside the root folder.

Example (for a Windows environment)

Imagine you have a "`D:`" disk where you put all your user data files. Create a folder called "`Barco`" at the root of this disk, and a subfolder of this folder, called "`Src`", where you will create all your projects. In this case, `<JBS-ROOT>` would be "`D:/Barco/Src`". We will use this example in the rest of this document.

3.2 Installation process

Once you have decided where you want to store all or some of your projects (i.e. you have decided where your `<JBS-ROOT>` should be), you can proceed with the installation of **JBS**.

*You should start with an empty root folder, although the chance that **JBS** overrides existing files is rather small. If you want to start with a non-empty root folder, please backup you folder first!*

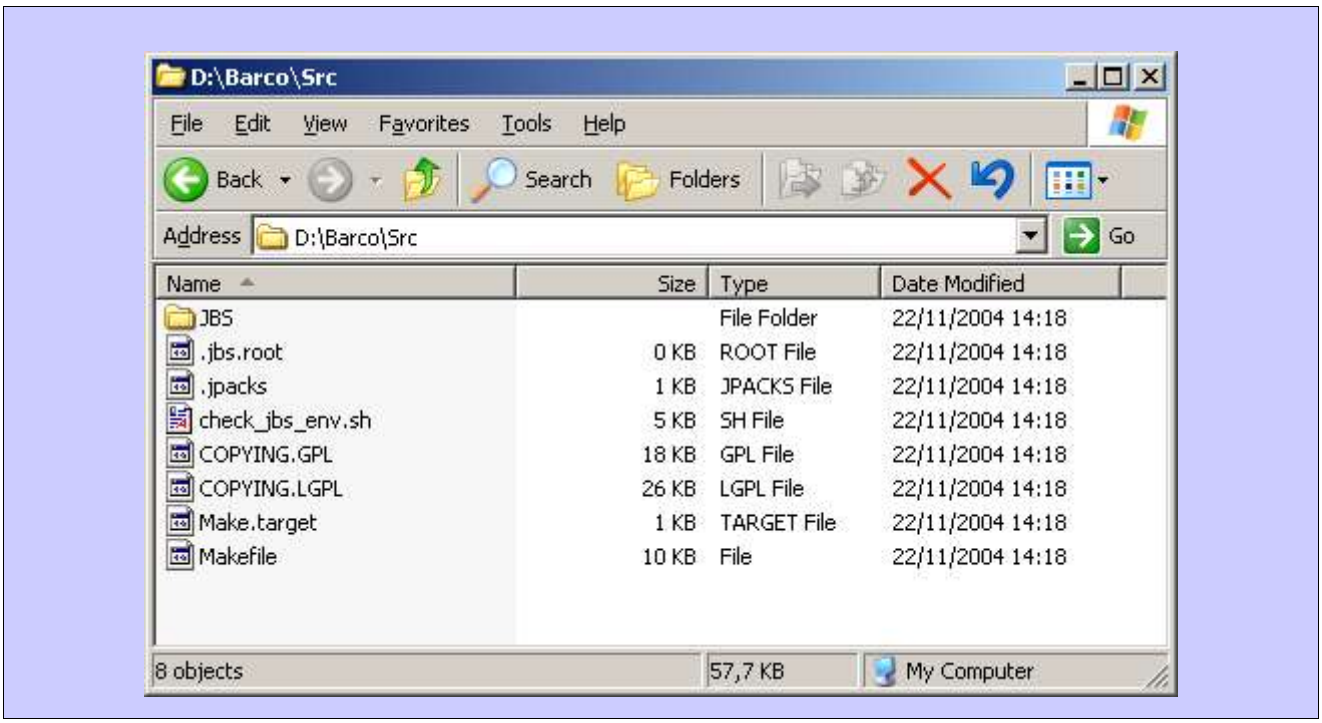
The installation process is straightforward:

- launch the `VMPPacksInstaller` found on the central **VMP** web server (http://vmp.barco.com/)
- select your `<JBS-ROOT>` as installation folder
- connect to the **VMP** packages server (also at vmp.barco.com, TCP port 1960)
- select and install the package called "`JBS`" (type and platform independent, version should be at least 4.2).

That's it!

Now, your `<JBS-ROOT>` should contain a number of new files (see screenshot of a Windows explorer below).

© Barco N.V. 2005



Those files are described below.

- `.jbs.root`: this empty file is used by **JBS** to determine the root of the current development tree. NEVER delete this file, as **JBS** would sometimes enter infinite loops!!!
- `.jpacks`: this file is managed by the packages installer, and keeps track of all the packages you installed in this folder and subfolders.
- `check_jbs_env.sh`: [obsolete! Not found in the latest releases!]
- `COPYING.*`: those files contain the text of the GNU licenses used for some packages that you will sooner or later install on your PC.
- `Make.target` & `Makefile`: see chapter on usage below.

Last but not least, the folder called "`JBS`" contains the building system itself, and a number of other files, which will be described later.

© Barco N.V. 2005

4. Usage

4.1 Introduction

JBS is case sensitive! In all the examples below, ALWAYS use the correct case when you have to manually enter a file or folder name.

Invoking **JBS** to perform some action is always done by launching Make[1] using the command "make" or "gmake" (or some variation around this – read the manual) in a shell environment (either in a Cygwin window, or using some terminal on Linux).

Make will try to find, read, and interpret a file named "`Makefile`" in the current directory. Failing to do so will result in an error being displayed, and "make" exiting.

*Those `Makefile` files are automatically generated by **JBS** in each and every folder of your development tree, starting at the `<JBS-ROOT>` folder.*
*Modifying such a `Makefile` will simply result in your modified version being irremediably replaced the next time **JBS** will enter your directory.*

JBS can perform a lot of different tasks in your folders, but the principle is always the same:

- open a Cygwin window, or open a terminal under Linux
- type "`cd <folder_name>`" to go to the folder named <folder_name>, which must contain a valid **JBS** Makefile,
- type "`make [<args>]`", where the optional arguments <args> tell **JBS** what to do.

JBS will then start to perform the requested task in the current folder, BUT ALSO recursively in all child folders.

Interacting with JBS via a command-line interface can sometimes be painful, for example when you have to enter very long file or folder names. The terminals we suggest to use (Cygwin window, or Linux eterm for example) all provide a so-called "auto-completion" mechanism, which helps you enter such long names easily. Simply type in the first letter(s) of a command, file, or folder name, and type 1 or 2 times on the <TAB> key to get a list of possible completions. From the manual page of bash:

```
complete (TAB)

       Attempt to perform completion on the text before point. Bash attempts
       completion treating the text as a variable (if the text begins with
       $), username (if the text begins with ~), hostname (if the text
       begins with @), or command (including aliases and functions) in turn.
       If none of these produces a match, filename completion is attempted.
```

1 "Make" or "GNU Make" is the GNU Make program (see http://www.gnu.org/software/make/make.html), while "make" is the name of the executable file.

© Barco N.V. 2005

*You don't have to always "`cd`" to a folder before using **JBS**. You can also type*

```
make -C <folder_name> [<args>]
```

from your current directory. See the Make documentation for more information on all the possible ways of invoking Make.

The recursive traversal of your folders is governed by the following rules:

- if a subfolder of the current folder contains a file called "`.jbs.noenter`" (note the leading dot), then this subfolder will never be entered, and its content will never be modified in any way.
- otherwise, if this subfolder doesn't already contain a `Makefile`, a new `Makefile` is generated in this subfolder, by copying the `Makefile` found in the current folder.
- otherwise, if this subfolder already contains a `Makefile`, but it's not a copy of the `Makefile` in the current folder, then this "invalid" `Makefile` is renamed to "`Makefile.old`", and a fresh copy of the `Makefile` in the current folder is placed in this subfolder.

Let's continue our example started in the previous chapter. Now, let's open a shell, go to the `<JBS-ROOT>` folder, and type "`make`". Here is a transcript of this session.

```
[ ~ ]: cd D:/Barco/Src
[ /cygdrive/d/Barco/Src ]: ls
COPYING.GPL  COPYING.LGPL  JBS  Make.target  Makefile  check_jbs_env.sh
[ /cygdrive/d/Barco/Src ]: make
make[1]: Entering directory `/cygdrive/d/Barco/Src/JBS'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `/cygdrive/d/Barco/Src/JBS'
[ /cygdrive/d/Barco/Src ]:
```

The first command ("`cd D:/Barco/Src`") enters our `<JBS-ROOT>` folder. Be sure you use the correct case when you type this command yourself. Use the auto-completion mechanism to help you! It won't let you use the wrong case! Then, the "`ls`" command shows the content (without the hidden files – those having a leading '.') of the current folder; this is of course optional!

The "`make`" command (without any argument) invokes **JBS**, and asks to perform the default action in this folder.

The default action is called "all". So, typing "`make`" is equivalent to typing "`make all`".

The 3 lines of output of Make shows that **JBS** doesn't have anything to do in the current folder. This is because the installation process has created an "empty" project in our `<JBS-ROOT>` file. Using other kinds or projects is described in the rest of this chapter.

4.2 "Makefile" & "Make.target"

The 2 most important files in any **JBS** folder are the "`Makefile`" and the "`Make.target`" files. Both have to exist to be able to use **JBS** in any folder.

As has been explained in the previous section, the `Makefile` is automatically generated by **JBS** when needed. This file should never be modified by hand! Any change would be lost the next time **JBS** is invoked in an upper folder!

© Barco N.V. 2005

An “empty” `Make.target` is also generated by **JBS** when no such file is found in a folder that **JBS** should enter. This default `Make.target` has then to be edited by the user. Once a `Make.target` exists, it'll never be modified by **JBS**.

The `Makefile` is used by **JBS** to do the following:

- figure out the platform being used (Windows, Linux, ...)
- adapt the behavior of **JBS** to this particular environment
- call the core engine of **JBS**, which resides in the “`JBS`” folder of the `<JBS-ROOT>` folder, to recursively perform the requested task.

One of the first thing the core engine will do upon invocation is to try to read the “`Make.target`” file in the current directory. This file should contain a number of definitions, telling **JBS** what to do in the current directory.

For example, a folder containing the source code of a DLL will contain a `Make.target` describing this DLL (location of the source code, compilation flags, additional dependencies, ...).

4.3 JBS variables

All “`Make.target`” files contain comment lines (those starting with a '#') and definitions of variables. Those definitions use the Make syntax, and are of the form:

```
<variable_name>          :=     <variable_value>
```

Here is a list of all variables recognized by **JBS**. Please consult the next section to see which variables are actually used for each type of project.

Variable	*Description*
target-type	This is the most important variable to be defined in any `Make.target` file, as it tells **JBS** the type of project the current folder contains. Examples of value are: “none”, “exe”, “plugin”, ...
target-name	All projects should have a unique name, which distinguishes them from each other. A project name MAY NOT contain any ' ' (space) or '-' (dash) character. If this variable is left empty, and a project name is needed, JBS will provide a default one, traditionally 'NoName', or something like that.
target-ver-maj **target-ver-med** **target-ver-min**	Sometimes, **JBS** needs to create a version number for a project. Those variables are then used to form a **JPF** compliant version number, of the form: `version == “<target-ver-maj>.<target-ver-med>.<target-ver-min>”` If any of these variables is undefined, the default value '0' is used.
target-debug	For target types for which this is meaningful, especially for executables and libraries, this variable can be used to force **JBS** to compile the current project with debugging symbols added. You should use the value “yes” to do this. Any other value will be interpreted as “no”.
target-stripped	For executables under Linux, this variable can be used to force **JBS** to strip the executable, thus considerably reducing its size. Use “yes” to do so. Any other value will be interpreted as “no”.

© Barco N.V. 2005

Variable	*Description*
target-deps	A project can depend on other projects to get rebuilt correctly. For example, an executable can use additional static or dynamic libraries, which should be compiled before the executable is actually linked. This variable should contain a space separated list of all the other **JBS** projects contained in the current development tree the current project depends on.
target-sources	If the current project involves compiling some source files, the file names should be provided here as a space separated list. Note that wildcards can be used! See examples below!
target-mocs	If the current project uses the Qt toolkit, this variable can be used to tell **JBS** to use the MOC compiler on the specified space separated list of .h files, which should contain Qt signals or slots.
target-guis	If the current project uses the Qt toolkit, this variable can be used to tell **JBS** to use the UIC compiler on the specified space separated list of .ui files, which should have been generated using the QtDesigner GUI builder.
target-cxxdefs	Projects that involve compilation of C++ files can use this variable to pass additional defines when compiling. All those defines should be given as a space separated list of definitions, each definition being given without compiler specific prefix (i.e. no "-D" or "/D" prefix). Example: target-cxxdefs := DEBUG ALGO=6 HAVE_CONFIG_H
target-incldirs	If the current project uses header files (*.h) coming with a 3rd party software package, the path to those file can then be provided here (space separated list). Example: target-incldirs := D:/Libs/include
target-libs	If the current project uses 3rd party libraries, their name can be provided here (space separated list), without any linker specific flag. Example: target-libs := jpeg would pass "-ljpeg" to the Linux linker, or "jpeg.lib" to the Windows linker.
target-libdirs	The path to those 3rd party libraries can be given here. Example: target-libdirs := /usr/local/lib/ffmpeg
target-toclean	This variable can be used to specify additional files that should get deleted when the user does a "make clean" (more on that later).
target-docdir	See section 6.1 on page 23.
target-doxydir	See section 6.1 on page 23.
target-doxyimg	See section 6.1 on page 23.

© Barco N.V. 2005

4.4 Project types

The following table describes all the available project types **JBS** can handle (see the "target-type" variable description above).

<target-type>	*Description*
none	The "null" target. **JBS** will consider that the current folder doesn't contain any project, but will still try to recursively traverse all subfolders found in the current folder, with the restrictions described above.
lib	This type is used to build static[2] libraries ("`*.lib`" under Windows, "`*.a`" under Linux).
dll	This type is used to build dynamic libraries ("`*.dll`" under Windows, "`*.so`" under Linux).
exe	Executables ("`*.exe`" under Windows, no extension under Linux).
rte	Special type used for dynamic libraries coming without source code or API.
sdk	Special type used for dynamic libraries coming with their API, but without any source code.
plugin	This type is used for dynamic libraries that should be loaded at run-time. Those libraries cannot be used as dependencies by other projects.
user	User-defined type. Not documented.

4.5 Global definitions repository

This repository is used to keep track of the exact location of your projects, and various other things. This enables you to move your projects around in your folder hierarchy, without having to edit thousands of .dsp files for example to update the links between projects.

4.6 Common tasks

Command	*Description*
all	Rebuilds the target (DLL, executable, ...).
clean	Reverses the effects of doing "all" (i.e. clean up everything)

2 Discussing the differences between static and dynamic is outside the scope of the present document.

© Barco N.V. 2005

Command	*Description*
defs	Updates the global definitions repository. The definitions of the project in this folder, and all subfolders, are concatenated, and push upwards to the global definitions repository. This means that the definitions of projects lying in parallel folders won't get updated! The intermediary files and the global definitions repository are all called "`.jbs.defs.<platform>`". Doing a "make defs" in `<JBS-ROOT>` is still the best way of updating the global repository! Don't do it in subfolders unless you're a **JBS** expert!
depend	Generates a local list of dependencies (in the C sense) between your source and header files. This file is called "`.jbs.deps.<platform>`" and is placed in the project folder.
doc	Generates the documentation of your source code.
packs	Generates a number of **JPF** files of your project in the "`<JBS-ROOT>/Packs`" folder.
tarbz2	Generates a compressed (using bzip2) archive (using tar) or the local folder, and places the generated file one folder above.
project	Interactively generates a new project in the current folder.
main	Interactively generates a new "main" C++ file.
class	Interactively generates a new source + header pair of files implementing a new C++ class.
dumpvars	For debugging purposes: displays some of the internal variables **JBS** would use if asked to perform something in the current directory.

© Barco N.V. 2005

5. Complete example

5.1 Introduction

In this chapter, we will build a complete development tree containing 2 libraries and 1 executable, ... We will also automatically generate the API documentation, and prepare **JPF** packages to distribute.

More precisely, we would like to build the following elements:

- a static library, called LowLevelLib, containing only one C++ class, called LowLevelClass.
- A dynamic library, called HighLevelLib, containing only one C++ class, called HighLevelClass. The HighLevelClass will use the LowLevelClass (dependencies!).
- An executable called MainApp, using the services provided by the HighLevelClass.

5.2 Creating the development tree

We will reuse the `<JBS-ROOT>` folder used in the first example, and suppose **JBS** has been successfully installed in this folder using the Packages Installer.

We verify the content of this folder now. It should look like this:

```
[ ~ ]: cd D:/Barco/Src
[ /cygdrive/d/Barco/Src ]: ls
COPYING.GPL  COPYING.LGPL  JBS  Make.target  Makefile
[ /cygdrive/d/Barco/Src ]:
```

5.3 Creating LowLevelLib

Let's start by creating our low level static library.

To do so, we use the instantiator coming with **JBS**! At the time this document was written (**JBS** version 4.2.6 is the latest version), this instantiator only asks for a project name, which is used to create a new folder of that name, and to create a `Makefile` and a `Make.target` in this new folder.

After the project has been created, we enter its folder.

```
[ /cygdrive/d/Barco/Src ]: make project
Project name                                    ? LowLevelLib
[ /cygdrive/d/Barco/Src ]: ls
COPYING.GPL  COPYING.LGPL  JBS  LowLevelLib  Make.target  Makefile
[ /cygdrive/d/Barco/Src ]: cd LowLevelLib
[ /cygdrive/d/Barco/Src/LowLevelLib ]: ls
Make.target  Makefile
[ /cygdrive/d/Barco/Src/LowLevelLib ]:
```

Now, we will create a new empty C++ class, called “LowLevelClass”. Let's use the C++ class instantiator coming with **JBS** for this purpose.

© Barco N.V. 2005

```
[ /cygdrive/d/Barco/Src/LowLevelLib ]: make class
Class name                                              ? LowLevelClass
Copyright owner ['j' - JFGO, 'b' - Barco, '' - none] ? b
License type    ['l' - LGPL, 'g' - GPL, '' - none  ] ?
Namespace(s)    ['j' - JFC, 'v' - VMP, '' to stop  ] ?
In DLL project                                          ?
[ /cygdrive/d/Barco/Src/LowLevelLib ]: ls
LowLevelClass.cpp  LowLevelClass.h  Make.target  Makefile
[ /cygdrive/d/Barco/Src/LowLevelLib ]:
```

The instantiator has now created 2 files named "`LowLevelClass.h`" and "`LowLevelClass.cpp`" (first answer above - "`LowLevelClass`"), each of which contains a copyright mention at the top (second answer above - "`b`").

Use your favorite editor to edit those files now. You will see that those files provide the declaration (header file) and the definition (source file) of a class called "LowLevelClass", and additional stuff. Use your editor to modify those files, so that they look like this (empty lines and some comments have been removed to spare space):

`LowLevelClass.h`

```
// ============================================================================
#ifndef _LowLevelClass_H_
#define _LowLevelClass_H_
// ============================================================================
class LowLevelClass {
public :
    LowLevelClass() {}
    virtual ~LowLevelClass() {}
    void doIt();
protected :
private :
    // others...
    LowLevelClass(
          const  LowLevelClass&
    );
    LowLevelClass& operator = (
          const  LowLevelClass&
    );
};
// ============================================================================
#endif // _LowLevelClass_H_
// ============================================================================
```

`LowLevelClass.cpp`

```
// ============================================================================
#include <iostream>
#include "LowLevelClass.h"
// ============================================================================
void LowLevelClass::doIt() {
    std::cerr << "LowLevelClass::doIt()" << std::endl;
}
// ============================================================================
```

Now, we have to configure this project. Edit the "`Make.target`" file, so that it looks like this (the only thing to change is the definition of "target-type"):

© Barco N.V. 2005

```
target-type      :=    lib
target-debug     :=    no
target-stripped  :=    yes
target-name      :=    LowLevelLib
target-ver-maj   :=    1
target-ver-med   :=    0
target-ver-min   :=    0
target-sources   :=    *.cpp *.h
target-mocs      :=
target-guis      :=
target-deps      :=    # JFC_Common
target-incldirs  :=
target-libs      :=
target-libdirs   :=
target-docdir    :=    # Doc         # Base folder for documentation.
target-doxydir   :=    # Classes     # Auto-gen. files go to ./Doc/Classes
target-doxyimg   :=    # Images      # User images go to ./Doc/Images
```

OK! Your project is ready for building! Let's do it now!

```
[ /cygdrive/d/Barco/Src/LowLevelLib ]: ls -l
total 15
-rwxr-xr-x    ...      668 Nov 23 14:42 LowLevelClass.cpp
-rwxr-xr-x    ...      943 Nov 23 14:42 LowLevelClass.h
-rwxr-xr-x    ...     2873 Nov 23 14:44 Make.target
-rwxr-xr-x    ...     9736 Nov 23 14:10 Makefile
[ /cygdrive/d/Barco/Src/LowLevelLib ]: make
===> [COMPILE] LowLevelClass.cpp...
LowLevelClass.cpp
===> [LNK LIB] LowLevelLib.lib...
[ /cygdrive/d/Barco/Src/LowLevelLib ]: ls -l
total 53
-rwxr-xr-x    ...      668 Nov 23 14:42 LowLevelClass.cpp
-rwxr-xr-x    ...      943 Nov 23 14:42 LowLevelClass.h
-rwxr-xr-x    ...     1735 Nov 23 14:46 LowLevelClass.obj
-rwxr-xr-x    ...     2080 Nov 23 14:46 LowLevelLib.lib
-rwxr-xr-x    ...     2873 Nov 23 14:44 Make.target
-rwxr-xr-x    ...     9736 Nov 23 14:10 Makefile
-rwxr-xr-x    ...    33792 Nov 23 14:46 vc60.idb
[ /cygdrive/d/Barco/Src/LowLevelLib ]:
```

The first command above dumps the content of the current folder using a longer format than above. We see that our folder contains our class files, and the **JBS** files `Make.target` and `Makefile`.

The second command starts the building process. **JBS** first compiles the class, then generates a static library.

The third command shows that 2 new files appeared: the object file for our class ("`LowLevelClass.obj`") and the static library file ("`LowLevelLib.lib`"). The "`vc60.idb`" is a residue of VisualStudio, and is not present when using Linux.

Perfect!

Try to do another "`make`". **JBS** doesn't generate the library a second time because the library is already up to date.

Now, modify the source (.cpp) file, for example by inserting some spaces somewhere (no new code!). Then, do a "`make`" again. You see the difference ? This time, **JBS** has regenerated the library, because one of its sources changed.

© Barco N.V. 2005

Now, modify the header, and do a "`make`". Nothing happens! This is because **JBS** doesn't know that your library depends on this header file too (see the section on dependencies!).

To help **JBS**, we have to generate the dependencies now! This is done by typing "`make depend`". After that, we do a new "`make`", and see that **JBS** regenerates our library this time by looking at the output of the "`ls -la`" command, which dumps the content of the current folder, without omitting the hidden files this time!

```
[ /cygdrive/d/Barco/Src/LowLevelLib ]: make depend
===> [GEN DEP] LowLevelClass.cpp...
===> [GEN DEP] LowLevelClass.cpp...
[ /cygdrive/d/Barco/Src/LowLevelLib ]: make
===> [COMPILE] LowLevelClass.cpp...
LowLevelClass.cpp
===> [LNK LIB] LowLevelLib.lib...
[ /cygdrive/d/Barco/Src/LowLevelLib ]: ls -la
total 54
drwxr-xr-x+   ...        0 Nov 23 15:12 .
drwxr-xr-x+   ...        0 Nov 23 14:09 ..
-rw-r--r--    ...       53 Nov 23 15:12 .jbs.deps.WIN32
-rwxr-xr-x    ...      669 Nov 23 15:06 LowLevelClass.cpp
-rwxr-xr-x    ...      944 Nov 23 15:07 LowLevelClass.h
-rwxr-xr-x    ...     1735 Nov 23 15:12 LowLevelClass.obj
-rwxr-xr-x    ...     2080 Nov 23 15:12 LowLevelLib.lib
-rwxr-xr-x    ...     2873 Nov 23 14:44 Make.target
-rwxr-xr-x    ...     9736 Nov 23 14:10 Makefile
-rwxr-xr-x    ...    33792 Nov 23 15:12 vc60.idb
[ /cygdrive/d/Barco/Src/LowLevelLib ]:
```

We see that **JBS** has created a file called "`.jbs.deps.WIN32`" here. If you take a rapid look at the content of this file, you see that it contains a rule stating that the `LowLevelClass.obj` object file depends on the source file (but **JBS** knew that already), but also on the header file (this is new). **JBS** will read this file every time is has to get your library up to date.

5.4 Creating HighLevelLib

Let's now create our HighLevelLib dynamic library (a DLL in this case – this example is built on a Windows platform).

The first steps are to create a folder next to the LowLevelLib (but we could put it anywhere else, provided it's somewhere below the same `<JBS-ROOT>` folder), and to create header and source files.

Note the values we provide to the class instantiator. This time, we also have to say that our class will be part of a DLL. Although this is no problem for Linux, Windows – as always - uses a very cumbersome mechanism (you must prefix your declarations with "__declspec(dllimport)" when including some header of a DLL, and "__declspec(dllexport)" when you recompile this DLL, which is not easy/clear). **JBS** works around this problem by using a set of defines that get passed to the compiler, and that get used in the headers of your DLLs. See below.

```
[ /cygdrive/d/Barco/Src/LowLevelLib ]: cd ..
[ /cygdrive/d/Barco/Src ]: make project
Project name                                    ? HighLevelLib
```

© Barco N.V. 2005

```
[ /cygdrive/d/Barco/Src ]: cd HighLevelLib
[ /cygdrive/d/Barco/Src/HighLevelLib ]: make class
Class name                                      ? HighLevelClass
Copyright owner ['j' - JFGO, 'b' - Barco, '' - none] ? b
License type    ['l' - LGPL, 'g' - GPL, '' - none  ] ?
Namespace(s)    ['j' - JFC, 'v' - VMP, '' to stop  ] ?
In DLL project                                  ? HighLevelLib
[ /cygdrive/d/Barco/Src/HighLevelLib ]:
```

Now, edit the files, so they look like this:

`HighLevelClass.h`

```
#ifndef _HighLevelClass_H_
#define _HighLevelClass_H_

#if defined(PLATFORM_WIN32)
    #define NO_DLL
    #define DLLEXPORT __declspec(dllexport)
    #define DLLIMPORT __declspec(dllimport)
    #pragma warning ( disable : 4251 )
#else
    #define NO_DLL
    #define DLLEXPORT
    #define DLLIMPORT
#endif

#include "LowLevelClass.h"

class HighLevelLib_DLL HighLevelClass {
public :
    void doThat();
protected :
    LowLevelClass mbr;
};

#endif // _HighLevelClass_H_
```

`HighLevelClass.cpp`

```
#include <iostream>
#include "HighLevelClass.h"

void HighLevelClass::doThat() {
    std::cerr << "HighLevelClass::doThat(): --->" << std::endl;
    mbr.doIt();
    std::cerr << "HighLevelClass::doThat(): <---" << std::endl;
}
```

`Make.target`

```
target-type      :=    dll
target-debug     :=    no
target-stripped  :=    yes
target-name      :=    HighLevelLib
target-ver-maj   :=    1
target-ver-med   :=    0
target-ver-min   :=    0
target-sources   :=    *.cpp *.h
target-mocs      :=
target-guis      :=
target-deps      :=    # JFC_Common
target-incldirs  :=
target-libs      :=
target-libdirs   :=
```

© Barco N.V. 2005

```
target-docdir    :=    # Doc        # Base folder for documentation.
target-doxydir   :=    # Classes    # Auto-gen. files go to ./Doc/Classes
target-doxyimg   :=    # Images     # User images go to ./Doc/Images
```

When compiling this DLL under Windows, **JBS** will automatically define the following flags (to be used by the compiler):

HighLevelLib_DLL=DLLEXPORT

PLATFORM_WIN32

Additionally, **JBS** will define the following flags when compiling some other project that depends on your DLL:

HighLevelLib_DLL=DLLIMPORT

PLATFORM_WIN32

Try to compile it now. Do a "`make`"...

```
[ /cygdrive/d/Barco/Src/HighLevelLib ]: make
===> [COMPILE] HighLevelClass.cpp...
HighLevelClass.cpp
HighLevelClass.h(15) : fatal error C1083: Cannot open include file:
'LowLevelClass.h': No such file or directory
make: *** [HighLevelClass.obj] Error 2
[ /cygdrive/d/Barco/Src/HighLevelLib ]:
```

ERROR! The compiler failed because it didn't find the included header file, coming from the static library LowLevelLib. This is because we didn't tell **JBS** that HighLevelLib depends on LowLevelLib! Edit your HighLevelLib `Make.target` file:

```
...
target-guis      :=
target-deps      :=    LowLevelLib
target-incldirs  :=
...
```

Now, rebuild the global definitions repository:

```
[ /cygdrive/d/Barco/Src/HighLevelLib ]: make -C .. defs
make: Entering directory `/cygdrive/d/Barco/Src'
make[1]: Entering directory `/cygdrive/d/Barco/Src/HighLevelLib'
===> [GEN DEF] HighLevelLib...
make[1]: Leaving directory `/cygdrive/d/Barco/Src/HighLevelLib'
make[1]: Entering directory `/cygdrive/d/Barco/Src/JBS'
===> [GEN DEF] JBS...
make[1]: Leaving directory `/cygdrive/d/Barco/Src/JBS'
make[1]: Entering directory `/cygdrive/d/Barco/Src/LowLevelLib'
===> [GEN DEF] LowLevelLib...
make[1]: Leaving directory `/cygdrive/d/Barco/Src/LowLevelLib'
===> [GEN DEF] ....
make: Leaving directory `/cygdrive/d/Barco/Src'
[ /cygdrive/d/Barco/Src/HighLevelLib ]:
```

Take a look at the "`.jbs.defs.<your_platform>`" file that just got created in the `<JBS-ROOT>` folder...

Now, try again to build you DLL. Type "`make`":

```
[ /cygdrive/d/Barco/Src/HighLevelLib ]: make
make[1]: Entering directory `/cygdrive/d/Barco/Src/LowLevelLib'
make[1]: Nothing to be done for `local-target'.
```

© Barco N.V. 2005

```
make[1]: Leaving directory `/cygdrive/d/Barco/Src/LowLevelLib'
===> [COMPILE] HighLevelClass.cpp...
HighLevelClass.cpp
===> [LNK LIB] HighLevelLib-1.0.0.dll...
   Creating library HighLevelLib.lib and object HighLevelLib.exp
[ /cygdrive/d/Barco/Src/HighLevelLib ]:
```

It works! Note **JBS** also first made sure that all the dependencies were up to date before generating your dynamic library! You can verify this by slightly modifying some source file in the LowLevelLib folder, and then doing a "`make`" again in the HighLevelLib folder:

```
[ /cygdrive/d/Barco/Src/HighLevelLib ]: touch ../LowLevelLib/LowLevelClass.h
[ /cygdrive/d/Barco/Src/HighLevelLib ]: make
make[1]: Entering directory `/cygdrive/d/Barco/Src/LowLevelLib'
===> [COMPILE] LowLevelClass.cpp...
LowLevelClass.cpp
===> [LNK LIB] LowLevelLib.lib...
LINK : warning LNK4044: unrecognized option "STACK:10000000"; ignored
make[1]: Leaving directory `/cygdrive/d/Barco/Src/LowLevelLib'
[ /cygdrive/d/Barco/Src/HighLevelLib ]:
```

Rem: the "`touch`" command simply set the access and modification times of the file to the current time, faking that the file has been edited.

You see in the session above that **JBS** has correctly rebuilt the static library (the dependency), but that is HAS NOT rebuilt the dynamic library! Guess why! The reason is simple: missing dependencies!

Do the following:

```
[ /cygdrive/d/Barco/Src/HighLevelLib ]: make depend
===> [GEN DEP] HighLevelClass.cpp...
===> [GEN DEP] HighLevelClass.cpp...
[ /cygdrive/d/Barco/Src/HighLevelLib ]: make
make[1]: Entering directory `/cygdrive/d/Barco/Src/LowLevelLib'
make[1]: Nothing to be done for `local-target'.
make[1]: Leaving directory `/cygdrive/d/Barco/Src/LowLevelLib'
===> [COMPILE] HighLevelClass.cpp...
HighLevelClass.cpp
===> [LNK LIB] HighLevelLib-1.0.0.dll...
   Creating library HighLevelLib.lib and object HighLevelLib.exp
[ /cygdrive/d/Barco/Src/HighLevelLib ]: make
make[1]: Entering directory `/cygdrive/d/Barco/Src/LowLevelLib'
make[1]: Nothing to be done for `local-target'.
make[1]: Leaving directory `/cygdrive/d/Barco/Src/LowLevelLib'
[ /cygdrive/d/Barco/Src/HighLevelLib ]:
```

This time, **JBS** has done it correctly! The second "`make`" shows that everything is up to date now.

As a last test, touch again the header file of the LowLevelLib, and do a "`make`":

```
[ /cygdrive/d/Barco/Src/HighLevelLib ]: touch ../LowLevelLib/LowLevelClass.h
[ /cygdrive/d/Barco/Src/HighLevelLib ]: make
make[1]: Entering directory `/cygdrive/d/Barco/Src/LowLevelLib'
===> [COMPILE] LowLevelClass.cpp...
LowLevelClass.cpp
===> [LNK LIB] LowLevelLib.lib...
LINK : warning LNK4044: unrecognized option "STACK:10000000"; ignored
make[1]: Leaving directory `/cygdrive/d/Barco/Src/LowLevelLib'
===> [COMPILE] HighLevelClass.cpp...
HighLevelClass.cpp
===> [LNK LIB] HighLevelLib-1.0.0.dll...
   Creating library HighLevelLib.lib and object HighLevelLib.exp
[ /cygdrive/d/Barco/Src/HighLevelLib ]:
```

© Barco N.V. 2005

Everything works fine now!

To remember:

- always regenerate definitions when new projects are created
- always regenerate dependencies when new files are added, or when the "#include" in your files changed.

5.5 Creating MainApp

The following session transcript should be no surprise:

```
[ /cygdrive/d/Barco/Src/HighLevelLib ]: cd ..
[ /cygdrive/d/Barco/Src ]: make project
Project name                                    ? MainApp
[ /cygdrive/d/Barco/Src ]: cd MainApp
[ /cygdrive/d/Barco/Src/MainApp ]: make main
Making main .cpp file from template.
File name (ex: TestCons):
MainApp
[ /cygdrive/d/Barco/Src/MainApp ]: ls
MainApp.cpp  Make.target  Makefile
[ /cygdrive/d/Barco/Src/MainApp ]:
```

What's new here is the "`make main`" command, which created a new "`MainApp.cpp`" file, based on some **JBS** template.

Now, let's edit the config file, and the source file of our application.

`Make.target`

```
target-type      :=    exe
target-debug     :=    no
target-stripped  :=    yes
target-name      :=    MainApp
target-ver-maj   :=    1
target-ver-med   :=    0
target-ver-min   :=    0
target-sources   :=    *.cpp *.h
target-mocs      :=
target-guis      :=
target-deps      :=    LowLevelLib HighLevelLib
target-incldirs  :=
...
```

`MainApp.cpp`

```
#include <iostream>
#include "HighLevelClass.h"

int main() {

    HighLevelClass c;
    c.doThat();

    return 0;

}
```

© Barco N.V. 2005

Note the “target-deps” of the MainApp contains the direct dependencies (HighLevelLib), but also the indirect dependencies (LowLevelLib, used by HighLevelLib)!

Now, we generate the “`.jbs.deps.<platform>`” file, and we build the executable:

```
[ /cygdrive/d/Barco/Src/MainApp ]: make depend
===> [GEN DEP] MainApp.cpp...
===> [GEN DEP] MainApp.cpp...
[ /cygdrive/d/Barco/Src/MainApp ]: make
make[1]: Entering directory `/cygdrive/d/Barco/Src/LowLevelLib'
make[1]: Nothing to be done for `local-target'.
make[1]: Leaving directory `/cygdrive/d/Barco/Src/LowLevelLib'
make[1]: Entering directory `/cygdrive/d/Barco/Src/HighLevelLib'
make[2]: Entering directory `/cygdrive/d/Barco/Src/LowLevelLib'
make[2]: Nothing to be done for `local-target'.
make[2]: Leaving directory `/cygdrive/d/Barco/Src/LowLevelLib'
make[1]: Leaving directory `/cygdrive/d/Barco/Src/HighLevelLib'
===> [COMPILE] MainApp.cpp...
MainApp.cpp
===> [LNK EXE] MainApp-1.0.0.exe...
[ /cygdrive/d/Barco/Src/MainApp ]:
```

Once again, JBS first made sure that all dependencies are up to date before building the local target, and this also recursively!

Let's take a look at the content of the current folder:

```
[ /cygdrive/d/Barco/Src/MainApp ]: ls -al
total 68
drwxr-xr-x+   2 ...        0 Nov 24 09:20 .
drwxr-xr-x+   6 ...        0 Nov 24 09:02 ..
-rw-r--r--    ...      135 Nov 24 09:20 .jbs.deps.WIN32
-rwxr-xr-x    ...      144 Nov 24 09:20 MainApp-1.0.0.bat
-rwxr-xr-x    ...    16384 Nov 24 09:20 MainApp-1.0.0.exe
-rwxr-xr-x    ...      220 Nov 24 09:20 MainApp-1.0.0.sh
-rwxr-xr-x    ...      112 Nov 24 09:12 MainApp.cpp
-rwxr-xr-x    ...     1317 Nov 24 09:20 MainApp.obj
-rwxr-xr-x    ...     2881 Nov 24 09:20 Make.target
-rwxr-xr-x    ...     9736 Nov 24 09:02 Makefile
-rwxr-xr-x    ...    33792 Nov 24 09:20 vc60.idb
[ /cygdrive/d/Barco/Src/MainApp ]:
```

We see that **JBS** generated more than one file here!

The “`MainApp-1.0.0.exe`” is the executable file, as expected. Note the version number has been automatically appended. Let's try to run this application! As soon as we type the following:

```
[ /cygdrive/d/Barco/Src/MainApp ]: ./MainApp-1.0.0.exe
[ /cygdrive/d/Barco/Src/MainApp ]:
```

(where we prefixed the executable name with a “`./`” to make sure we try to run the executable found in the current folder), we get a warning from Windows, stating that:

© Barco N.V. 2005

The reason this failed is because our application needs the HighLevelLib DLL, which is not somewhere in our `<PATH>`. To overcome this, **JBS** provides 2 wrappers ("`<filename>.bat`" and "`<filename>.sh`") which can be used to launch the application. Those wrappers are simple scripts that first set the environment so that the `<PATH>` contains all the needed components (to our dependencies), and then start the application.

The .bat file can be used outside of any Cygwin environment, for example if you double-click its icon in a browser, while the shell script (.sh extension) has to be interpreted inside a Cygwin window.

Let's start the shell script now:

```
[ /cygdrive/d/Barco/Src/MainApp ]: ./MainApp-1.0.0.sh
HighLevelClass::doThat(): --->
LowLevelClass::doIt()
HighLevelClass::doThat(): <---
[ /cygdrive/d/Barco/Src/MainApp ]:
```

It works! Note our application didn't write a "Hello, world!" message. This is a known bug ;-) Fixing the source code is left as an exercise...

© Barco N.V. 2005

6. Advanced example

In this chapter, we will progressively discover advanced features of **JBS**, such as automatic documentation generation, or packages creation.

The reader is supposed to already have accomplished all the steps presented in the previous chapter.

6.1 Documentation generation

One of the nice features of **JBS** is its ability to use the doxygen application to automatically generate documentation of your source code.

Before trying to use this feature, make sure you have a working copy of the doxygen application installed on your PC (see Annex/Links/Doxygen).

Now, go to the root of your development tree, and type "`make doc`":

```
[ /cygdrive/d/Barco/Src/MainApp ]: cd ..
[ /cygdrive/d/Barco/Src ]: make doc
make[1]: Entering directory `/cygdrive/d/Barco/Src/HighLevelLib'
make[1]: Leaving directory `/cygdrive/d/Barco/Src/HighLevelLib'
make[1]: Entering directory `/cygdrive/d/Barco/Src/JBS'
make[1]: Leaving directory `/cygdrive/d/Barco/Src/JBS'
make[1]: Entering directory `/cygdrive/d/Barco/Src/LowLevelLib'
make[1]: Leaving directory `/cygdrive/d/Barco/Src/LowLevelLib'
make[1]: Entering directory `/cygdrive/d/Barco/Src/MainApp'
make[1]: Leaving directory `/cygdrive/d/Barco/Src/MainApp'
[ /cygdrive/d/Barco/Src ]:
```

As usual, **JBS** processed the current folder recursively, but nothing magic happened...

For **JBS** to generate documentation, you first have to tell it where to place the automatically generated files. This is done using the following 3 **JBS** variables:

- **target-docdir**: this is the folder where you will put all your documentation, should it be automatically generated or not. This folder, if given, is also the only one to get copied when generating a "doc" package.
- **target-doxydir**: this should be the name of a subfolder of "target-docdir", where **JBS** will place its automatically generated files.
- **target-doxyimg**: this should be the name of a subfolder of "target-docdir", where you should place all the images referenced by your code documentation. Doxygen will use the "`<target-docdir>/<target-doxyimg>`" folder as its "IMAGE_PATH".

Now, create "Doc" folders in each project folder, and prevent **JBS** from entering them:

```
[ /cygdrive/d/Barco/Src ]: mkdir LowLevelLib/Doc
[ /cygdrive/d/Barco/Src ]: mkdir HighLevelLib/Doc
[ /cygdrive/d/Barco/Src ]: mkdir MainApp/Doc
[ /cygdrive/d/Barco/Src ]: touch LowLevelLib/Doc/.jbs.noenter
[ /cygdrive/d/Barco/Src ]: touch HighLevelLib/Doc/.jbs.noenter
[ /cygdrive/d/Barco/Src ]: touch MainApp/Doc/.jbs.noenter
[ /cygdrive/d/Barco/Src ]:
```

It's time to edit the "`Make.target`" file in each folder (3 files)! Each should contain at least the following 2 lines:

© Barco N.V. 2005

```
...
target-docdir    :=    Doc          # Base folder for documentation.
target-doxydir   :=    Classes      # Auto-gen. files go to ./Doc/Classes
...
```

You can now try to generate the doc again.

```
[ /cygdrive/d/Barco/Src ]: make doc

make[1]: Entering directory `/cygdrive/d/Barco/Src/HighLevelLib'

.... output skipped!

[ /cygdrive/d/Barco/Src ]:
```

This time it worked! Now, read the output of the previous command carefully. You should note that, as usual, JBS tried to update the dependencies first. This is also valid for the documentation!

Also, note that documentation was generated only for the libraries, and not for the main application. This is because documentation should only be generated for reusable components, and those should are naturally placed in libraries.

Now, point your favorite HTML browser to “HighLevelLib/Doc/Classes/html/index.html” and enjoy...

6.2 JPF packages generation

JBS is tightly coupled with the **JPF** (*JFG's Package Format*), and can be used to generate such packages, to be distributed using the `VMPPacksServer` and installed using the `VMPPacksInstaller`.

*You need to have installed the “`jpf_create`” utility on your computer before trying to generate **JPF** packages. This utility is available on the **VMP** central web server (http://vmp.barco.com/).*

Simply type “`make packs`”, and packages for the local project, and projects in all subfolders recursively, will be generated and moved to the “`<JBS-ROOT>/Packs`” folder. Do it now:

```
[ /cygdrive/d/Barco/Src ]: make packs
make[1]: Entering directory `/cygdrive/d/Barco/Src/HighLevelLib'
make[2]: Entering directory `/cygdrive/d/Barco/Src/HighLevelLib/Doc'
make[2]: Leaving directory `/cygdrive/d/Barco/Src/HighLevelLib/Doc'
===> Creating SRC package for project 'HighLevelLib'!...
===> ... creating temp directory...
===> ... copying tree...
===> ... archiving and compressing package!...
make[2]: Entering directory `/cygdrive/d/Barco/Src/LowLevelLib'
make[3]: Entering directory `/cygdrive/d/Barco/Src/LowLevelLib/Doc'
make[3]: Leaving directory `/cygdrive/d/Barco/Src/LowLevelLib/Doc'
make[2]: Leaving directory `/cygdrive/d/Barco/Src/LowLevelLib'
===> Generating Doxygen configuration file for HighLevelLib...
===> Generating Doxygen documentation for HighLevelLib...
Searching for include files...
Searching for example files...
Searching for images...
...
```

© Barco N.V. 2005

```
[ /cygdrive/d/Barco/Src ]: ls -l Packs
total 701
-rwxr-xr-x    ...   107630 Nov 24 11:59 HighLevelLib-1.0.0-doc-all.jpf
-rwxr-xr-x    ...    20884 Nov 24 12:00 HighLevelLib-1.0.0-rte-win32.jpf
-rwxr-xr-x    ...    24836 Nov 24 11:59 HighLevelLib-1.0.0-sdk-win32.jpf
-rwxr-xr-x    ...    26288 Nov 24 11:59 HighLevelLib-1.0.0-src-all.jpf
-rwxr-xr-x    ...   372677 Nov 24 12:00 JBS-4.2.5-all-all.jpf
-rwxr-xr-x    ...    92144 Nov 24 12:00 LowLevelLib-1.0.0-doc-all.jpf
-rwxr-xr-x    ...    25865 Nov 24 12:00 LowLevelLib-1.0.0-src-all.jpf
-rwxr-xr-x    ...    17025 Nov 24 12:00 MainApp-1.0.0-bin-win32.jpf
-rwxr-xr-x    ...    25640 Nov 24 12:00 MainApp-1.0.0-src-all.jpf
[ /cygdrive/d/Barco/Src ]:
```

You see that **JBS** has been repackaged too!

What's interesting is that all possible distribution formats are automatically generated: source code, SDK with accompanying documentation, RTE, ...

Note that static libraries are only available in source format! This should be improved in a future release of **JBS**.

© Barco N.V. 2005

7. Adding new target types

N/A

© Barco N.V. 2005

8. FAQs

N/A

© Barco N.V. 2005

9. Annex

9.1 Contact

Please contact "jeanfrancois.gobbers@barco.com" (JFGO) for any additional information.

9.2 Links

9.2.1 Cygwin (www.cygwin.com)

Cygwin is a GPL-ed Linux-like environment for Windows. It consists of two parts:

- A DLL (cygwin1.dll) which acts as a Linux emulation layer providing substantial Linux API functionality.
- A collection of tools, which provide Linux look and feel.

Cygwin is available in source form (protected by the GPL) using CVS.

9.2.2 OpenOffice (www.openoffice.org)

This document (text and some figures) has been created using the OpenOffice.org free of charge office productivity suite. OpenOffice.org contains the following elements:

- Writer: a word processor for creating professional documents, reports, ... Writer can read/write Microsoft Word documents (to some extent – see limitations on the website)... Writer can create PDF files without any additional tool.
- Calc: a spreadsheet with some advanced functions. Calc can also read/write Microsoft Exell docs (same remark).
- Impress: a PowerPoint-like multimedia presentations creator. Guess what it does with Microsoft PowerPoint files...
- Draw: produce everything from simple diagrams to dynamic 3D illustrations and special effects (was used for some figures in this document).
- and more...

9.2.3 Qt (www.trolltech.com)

Trolltech's graphical toolkit Qt is at the core of VMP. Qt does come in many different versions, and with various licensing schemes.

<explain further – GPL / why buy books & other commercial things>

9.2.4 VisualParadigm (www.visual-paradigm.com)

Visual Paradigm for the Unified Modeling Language (VP-UML) is a UML CASE tool available free of charge in its most stripped down version (Community Edition).

© Barco N.V. 2005

This tool has been used to model parts of VMP. Files produced with VP-UMP use a ".vpp" filename extension.

9.2.5 Doxygen (http://www.stack.nl/~dimitri/doxygen/)

Copied verbatim from the website:

> Doxygen is a documentation system for C++, C, Java, Objective-C, IDL (Corba and Microsoft flavors) and to some extent PHP, C# and D.
>
> It can help you in three ways:
>
> 1. It can generate an on-line documentation browser (in HTML) and/or an off-line reference manual (in LaTeX) from a set of documented source files. There is also support for generating output in RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and Unix man pages. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code.
> 2. You can configure doxygen to extract the code structure from undocumented source files. This is very useful to quickly find your way in large source distributions. You can also visualize the relations between the various elements by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically.

All platforms supported by JBS provide some pre-packaged version of the latest release of Doxygen:

- Cygwin: use the setup.exe installer of Cygwin to locate a package called "doxygen-???" in the list of available packages, and install it.
- Linux: you should easily find a .rpm, a .deb, or some other package for your distribution. Consult your documentation.

Thank you for using **JBS**!

JFGO

© Barco N.V. 2005