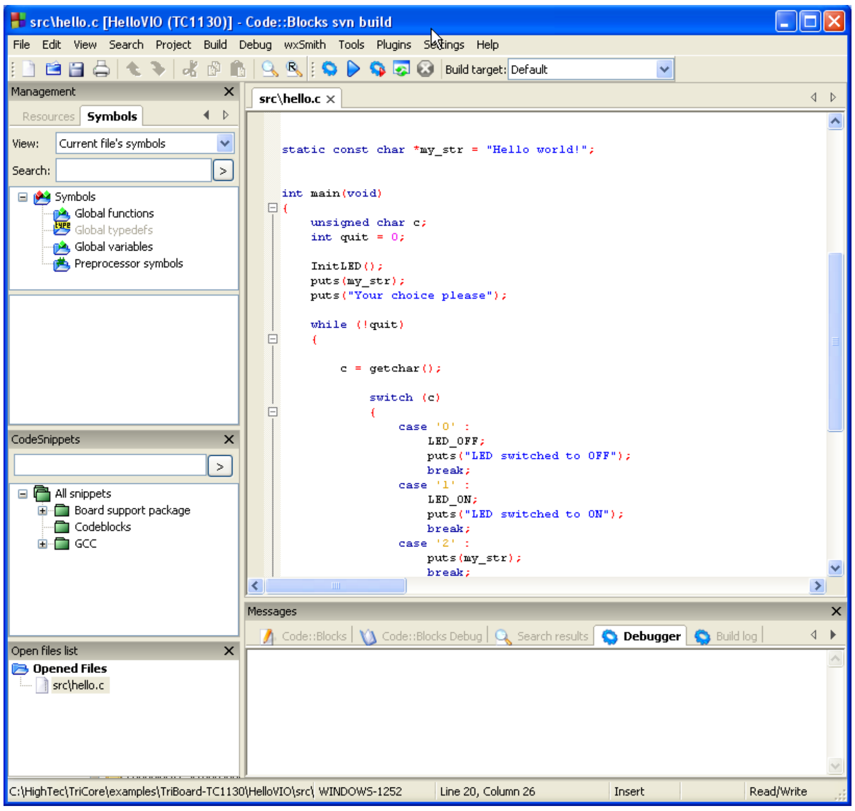
1 Code::Blocks Project Management

The instructions for chapter 3 on page 23 and chapter 4 on page 31 are official documentations of the Code::Blocks Wiki site and available in english only.

The below illustration shows the design of the Code::Blocks user interface.



```c
static const char *my_str = "Hello world!";

int main(void)
{
    unsigned char c;
    int quit = 0;

    InitLED();
    puts(my_str);
    puts("Your choice please");

    while (!quit)
    {

        c = getchar();

            switch (c)
            {
                case '0' :
                    LED_OFF;
                    puts("LED switched to OFF");
                    break;
                case '1' :
                    LED_ON;
                    puts("LED switched to ON");
                    break;
                case '2' :
                    puts(my_str);
                    break;
```

Figure 1.1: IDE Code::Blocks

Management This window contains the interface 'Projects' which will in the following text be referred to as the project view. This view show all the projects opened in Code::Blocks at a certain time. The 'Symbols' tab of the Management window shows symbols, variables etc..

Editor In the above illustration, a source named `hello.c` is opened with syntax highlighting in the editor.

Open files list shows a list of all files opened in the editor, in this example: `hello.c`.

CodeSnippets can be displayed via the menu 'View' →'CodeSnippets' . Here you can manage text modules and links to files.

Messages . This window is used for outputting search results, log messages of a compiler etc..

Code::Blocks offers a very flexible and comprehensive project management. The following text will address only some of the features of the project management.

1.1 Project View

In Code::Blocks, the sources and the settings for the build process are stored in a project file `<name>.cbp`. C/C++ sources and the corresponding header files are the typical components of a project. The easiest way to create a new project is executing the command 'File' →'Project' and selecting a wizard. Then you can add files to the project via the context menu 'Add files' in the Management window. Code::Blocks governs the project files in categories according to their file extensions. These are the preset categories:

Sources includes source files with the extensions `*.c;*.cpp;`.

Headers includes, among others, files with the extension `*.h;`.

Resources includes files for layout descriptions for xxWidgets windows with the extensions `*.res;*.xrc;`. These file types are shown in the 'Resources' tab of the Manangement window.

The settings for types and categories of files can be adjusted via the context menu 'Project tree' →'Edit file types & categories' . Here you can also define custom categories for file extensions of your own. For example, if you wish to list linker scripts with the `*.ld` extension in a category called `Linkerscript`, you only have to create the new category.

Note:

If you deactivate 'Project tree' →'Categorize by file types' in the context menu, the category display will be switched off, and the files will be displayed as they are stored in the file system.

1.2 Notes for Projects

In Code::Blocks, so-called notes can be stored for a project. These notes should contain short descriptions or hints for the corresponding project. By displaying this information during the opening of a project, other users are provided with a quick survey of the project. The display of notes can be switched on or off in the Notes tab of the Properties of a project.

1.3 Project Templates

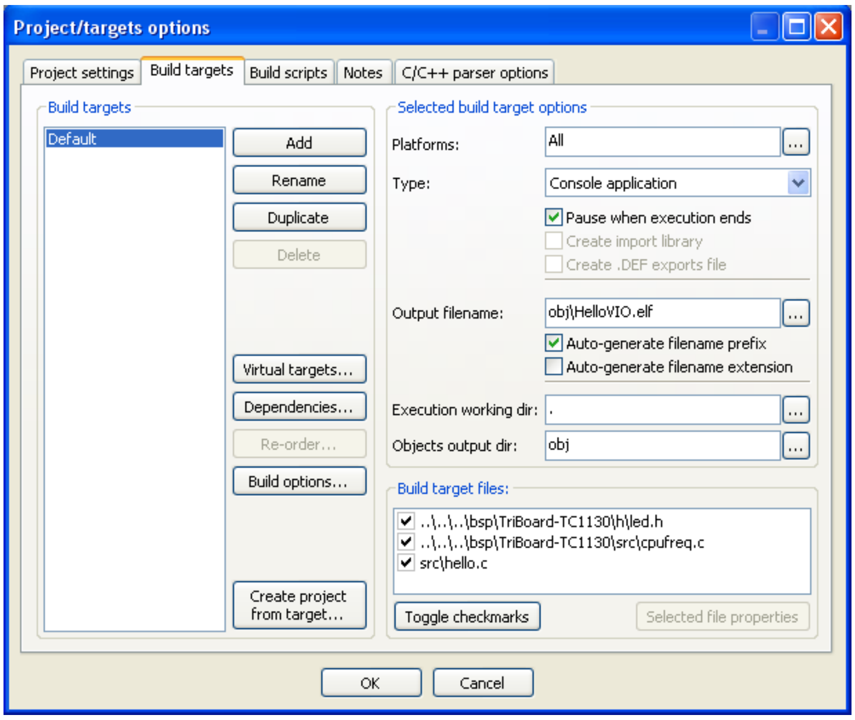
Code::Blocks is supplied with a variety of project templates which are displayed when creating a new project. However, it is also possible to store custom templates for collecting your own specifications for compiler switches, the optimisation to be used, machine-specific switches etc. in templates. These templates will be stored in the `Documents and Settings\<user>\Application Data\codeblocks\UserTemplates` directory. If the templates are to be open to all users, they have to be copied to a corresponding directory of the Code::Blocks installation. These templates will then be displayed at the next startup of Code::Blocks under New Project User templates.

Note:

The available templates in the Project Wizard can be edited by selection via right-click.

1.4 Create Projects from Build Targets

In projects it is necessary to have different variants of the project available. Variants are called Build Targets. They differ with respect to their compiler options, debug information and/or choice of files. A Build Target can also be outsourced to a separate project. To do so, click 'Project' →'Properties' , select the variant from the tab 'Build Targets' and click the 'Create project from target' button (see Figure 1.2 on page 3).



Default

Platforms: All

Type: Console application

☑ Pause when execution ends
☐ Create import library
☐ Create .DEF exports file

Output filename: obj\HelloVIO.elf

☑ Auto-generate filename prefix
☐ Auto-generate filename extension

Execution working dir: .

Objects output dir: obj

☑ ..\..\..\bsp\TriBoard-TC1130\h\led.h
☑ ..\..\..\bsp\TriBoard-TC1130\src\cpufreq.c
☑ src\hello.c

Figure 1.2: Build Targets

1.5 Virtual Targets

Projects can be further structured in Code::Blocks by so-called Virtual Targets. A frequently used project structure consists of two Build Targets, one 'Debug' Target which contains debug information and one 'Release' Target without this information. By adding Virtual Targets via 'Project' →'Properties' →'Build Targets' individual Build Targets can be combined. For example, a Virtual Target 'All' can create the Targets Debug and Release simultaneously. Virtual Targets are shown in the symbol bar of the compiler under Build Targets.

1.6 Pre- and Postbuild steps

Code::Blocks makes it possible to perform additional operations before or after compiling a project. These operations are called Prebuilt or Postbuilt Steps. Typical Postbuilt Steps are:

- Creating an Intel Hexformat from a finished object
- Manipulating objects by `objcopy`
- Generating dump files by `objdump`

Example

Creating a Disassembly from an object under Windows. Piping to a file requires calling `cmd` with the `/c` option.

```
cmd /c objdump -D name.elf > name.dis
```

Archiving a project can be another example for a Postbuilt Step. For this purpose, create a Build Target 'Archive' and include the following instruction in the Postbuilt Step:

```
zip -j9 $(PROJECT_NAME)_$(TODAY).zip src h obj $(PROJECT_NAME).cbp
```

With this command, the active project and its sources, header and objects will be packed as a zip file. In doing so, the Built-in variables $(PROJECT_NAME) and $(TODAY), the project name and the current date will be extracted (see section 3.2 on page 24). After the execution of the Target 'Archive', the packed file will be stored in the project directory.

1.7 Adding Scripts in Build Targets

Code::Blocks offers the possibility of using menu actions in scripts. The script represents another degree of freedom vor controlling the generation of your project.

Note:
A script can also be included at a Build Target.

1.8 Workspace and Project Dependencies

In Code::Blocks, multiple projects can be open. By saving open projects in a so-called Workspace via 'File' →'Save workspace' you can collect the projects in a single workspace under `<name>.workspace`. If you open `<name>.workspace` during the next startup of von Code::Blocks, all projects will show up again. Complex software systems consist of components which are managed in different Code::Blocks projects. Furthermore, with the generation of such software systems, there are often dependencies between these projects.

Example

A project A contains central functions which are made available to other projects in the form of a library. Now, if the sources of the central function are changed, then the library has to be recreated. To maintain consistency between a project B which uses the functions and project A which implements the functions, project B has to depend on project A. The necessary information on the dependencies of projects is stored in the relevant workspace, so that each project can be created separately. The usage of dependencies makes it also possible to control the order in which the projects will be generated. The dependencies for projects can be set via the selecting the menu 'Project' →'Properties' and then clicking the 'Project's dependencies' button.

1.9 Including Assembler files

In the Management window of the Project View, Assembler files are shown in the `Others` directory. Right-clicking one of the listed Assembler files will open a context menu. Select 'Properties' to open a new window. Now select the 'Build' tab and activate the two fields 'Compile file' and 'Link file'. Then select the 'Advanced' tab and execute the following steps:

1. Set 'Compiler variable' to CC
2. Select the compiler under 'For this compiler'
3. Select 'Use custom command to build this file'
4. In the window, enter:

   ```
   $compiler $options $includes <asopts> -c $file -o $object
   ```

The Code::Blocks variables are marked by `$` (see section 3.4 on page 26). They are set automatically so that you only have to replace the Assembler option <asopt> by your own settings.

1.10 Editor and Tools

1.10.1 Default Code

The company's Coding Rules require source files to have a standard design. Code::Blocks makes it possible to include a predefined content at the beginning of a file automatically when creating new C/C++ sources and headers. This predefined content is called default code. This setting can be selected under 'Stettings' →'Editor' Default Code. A new file can be created via the menu 'File' →'New' →'File' .

Example

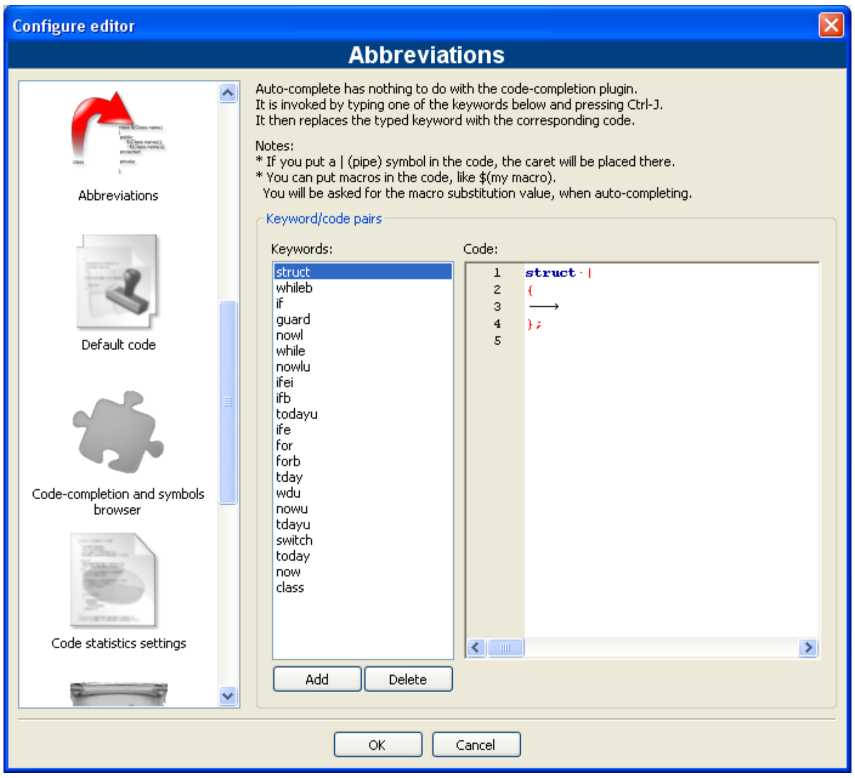
```
/*************************************************************************
 *  Project:
 *  Function:
 *************************************************************************
 *  $Author: mario $
 *  $Name:  $
 *************************************************************************
 *
 *  Copyright 2007 by company name
 *
 ************************************************************************/
```

1.10.2 Abbreviation

A lot of typing can be saved in Code::Blocks by defining abbreviation. This is done by selecting 'Settings' →'Editor' and defining the abbreviations under the name <name>, which can then be called by the keyboard shortcut Ctrl+J. Parametrisation is also possible by including variables `$(NAME)` in the abbreviations.

```
#ifndef $(Guard token)
#define $(Guard token)
#endif // $(Guard token)
```

When performing the abbreviation <name> in the source text and performing Ctrl+J, the content of the variable is requested and included.



Auto-complete has nothing to do with the code-completion plugin.
It is invoked by typing one of the keywords below and pressing Ctrl-J.
It then replaces the typed keyword with the corresponding code.

Notes:
* If you put a | (pipe) symbol in the code, the caret will be placed there.
* You can put macros in the code, like $(my macro).
You will be asked for the macro substitution value, when auto-completing.

Keywords: struct, whileb, if, guard, nowl, while, nowlu, ifei, ifb, todayu, ife, for, forb, tday, wdu, nowu, tdayu, switch, today, now, class

```
struct |
{

};
```

Figure 1.3: Defining abbreviations

1.10.3 Configuration Files

The Code::Blocks settings are stored in the `default.conf` profile in the `codeblocks` directory of your Application Data. When using personalities (see ?? on page ??), the configuration details will be stored in the `<personality>.conf` file.

The tool `cb_share_conf`, which can be found in the Code::Blocks installation directory, is used for managing and storing these settings.

If you wish to define standard settings for several users of a computer, the configuration file `default.conf` has to be stored in the directory `\Documents and Settings\Default User\Application Data\codeblocks`. During the first startup, Code::Blocks will copy the presettings from 'Default User' to the application data of the current users.

To create a portable version of Code::Blocks on a USB stick, proceed as follows. Copy the Code::Blocks installation to a USB stick and store the configuration file `default.conf` in this directory. The configuration will be used as a global setting. Please take care that the file is writeable, otherwise changes of the configuration cannot be stored.

1.10.4 Personalities

Code::Blocks settings are saved as application data in a file called `<user>.conf` in the `codeblocks` directory. This configuration file contains information such as the last opened projects, settings for the editor, display of symbol bars etc.. By default, the 'default' personality is set so that the configuration is stored in the file `default.conf`. If Code::Blocks is called from the command line with the parameter `--personality=myuser`, the settings will be stored in the file `myuser.conf`. If the profile does not exist already, it will automatically be created. This procedure makes it possible to create the corresponding profiles for different work steps. If you start Code::Blocks from the command line with the additional parameter`--personality=ask`, a selection box will be displayed for all the available profiles.

1.10.5 Navigate and Search

In Code::Blocks there are different ways of quick navigation between files and functions. Setting bookmarks is a typical procedure. Via the shortcut Ctrl+B a bookmark is set or deleted in the source file. Via Alt+PgUp you can jump to the previous bookmark, and via Alt+PgDn you can jump to the next bookmark.

Code::Blocks bietet verschiedene Möglichkeiten für die Suche in einer Datei oder in Verzeichnissen. In der Projektansicht können Sie durch Auswählen eines Projektes über das Kontextmenü 'Find file' in einem Dialog einen Dateinamen angeben. Dieser wird anschließend in der Projektansicht markiert und durch Eingabe mit Return im Editor geöffnet. Mit dem 'Search' →'Find' (Ctrl+F) oder 'Find in Files' (Ctrl+Shift+F) öffnet sich der Dialog für die Suche.

Code::Blocks offeres several ways of searching within a file or directory. In the project view, you can select a project via the context menu 'Find file' , and then enter a file name in the dialogue box. The corresponding file will be selected in the project view and can then be opened in the editor by pressing the Return key. The dialogue box for searching is opened via 'Search' →'Find' (Ctrl+F) or 'Find in Files' (Ctrl+Shift+F).

Ctrl-Alt-G is another useful function. The dialogue which will open on using this shortcut, lets you select functions and then jumps to the implementation of the selected function (see Figure 1.4 on page 8).

Note:

With the Ctrl+PgUp shortcut you can jump to the previous function, and via Ctrl+PgDn you can jump to the next function.

In the editor, you can switch between the tabs with the open files via Ctrl+Tab. Alternatively you can set 'Use Smart Tab-switching scheme' in 'Settings' →'Notebook appear-

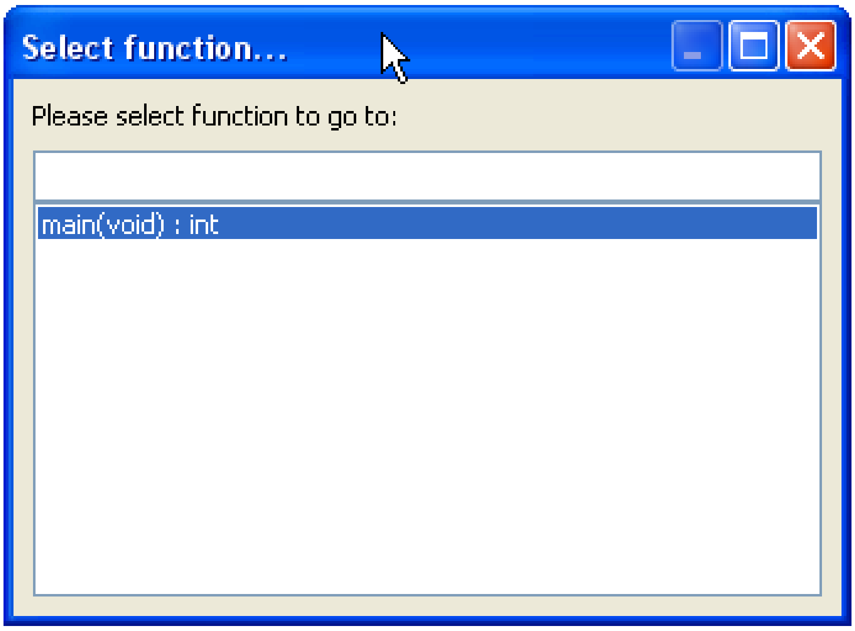


Figure 1.4: Search for functions

ance' , then Ctrl+Tab will bring up an Open Tabs window in which all the open files will be listed which can then be selected by mouse-click.

A common procedure when developing software is to struggle with a set of functions which are implemented in different files. The BrowseTracks plugin will help you solve this problem by showing you the order in which the files were selected. You can then comfortably navigate the function calls.

The display of line numbers in Code::Blocks can be activated via 'Settings' →'General Settings' in the field 'Show line numbers'. The shortcut Ctrl+G or the menu command 'Search' →'Goto line' will help you jump to the desired line.

1.10.6 Symbol view

The Code::Blocks Management window offers a tree view for symbols of C/C++ sources for navigating via functions or variables. As the scope of this view, you can set the current file or project, or the whole workspace. The following categories exist for the symbols:

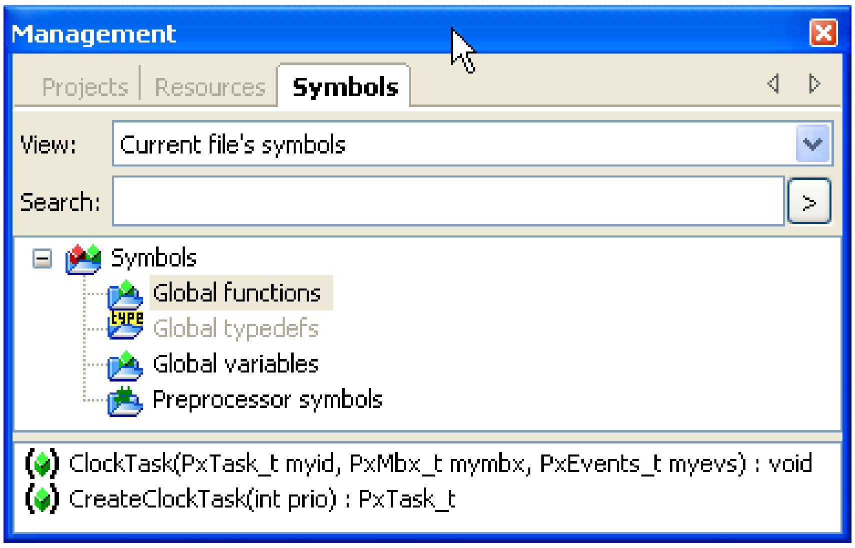
Global functions Lists the implementation of globale functions.

Global typedefs Lists the use of **typedef** definitions.

Global variables Displays the symbols of globale variables.

Preprocessor symbols Lists the pre-processor directives created by **#define**.

Structures and classes are displayed below the pre-processor symbols. If a category is selected by mouse-click, the found symbols will be displayed in the lower part of the window (see Figure 1.5 on page 9). Double-clicking the symbol will open the file in which the symbol is defined or the function implemented, and jumps to the corresponding line.



ClockTask(PxTask_t myid, PxMbx_t mymbx, PxEvents_t myevs) : void
CreateClockTask(int prio) : PxTask_t

Figure 1.5: Symbol view

Note:

In the editor, a list of the classes can be displayed via the context menus 'Insert Class method declaration implementation' or 'All class methods without implementation' .

1.10.7 Including external help files

The Code::Blocks development environment supports the inclusion of external help files via the menu 'Settings' →'Environment' . Include the manual of your choice in the chm format in 'Help Files' select 'this is the default help file'. Now you can select a function in an opened source file in Code::Blocks by mouse-click, and the corresponding documentation will appear.

If you have included multiple help files, you can select a term in the editor and choose a help file from the context menu 'Locate in' for Code::Blocks to search in.

1.10.8 Including external tools

Including external tools is possible in Code::Blocks via 'Tools' →'Configure Tools' →'Add' . Built-in variables (see section 3.2 on page 24) can also be accessed for tool parameters. Furthermore there are several kinds of launching options for starting external applications. Depending on the option, the externally started applications are stopped when Code::Blocks is quit. If the applications are to remain open after quitting Code::Blocks, the option 'Launch tool visible detached' must be set.

1.11 Tips for working with Code::Blocks

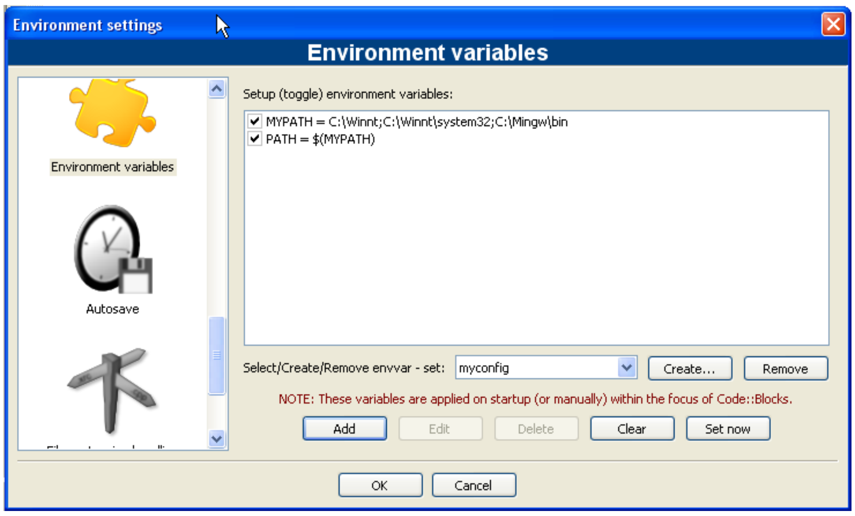
In this chapter we will present some useful settings in Code::Blocks.

1.11.1 Configuring environmental variables

The configuration for an operating system is specified by so-called environmental variables. The environmental variable PATH for example contains the path to an installed compiler. The operating system will process this environmental variable from beginning to end, i.e. the entries at the end will be searched last. If different versions of a compiler or other applications are installed, the following situations can occur:

- An incorrect version of a software is called
- Installed software packages call each other

So it might be the case that different versions of a compilers or other tools are mandatory for different projects. One possibility in such a case is to change the environmental variables in the system control for every project. However, this procedure is error-prone and not flexible. For this requirement, Code::Blocks offers an elegant solution. Different configurations of environmental variables can be created which are used only internally in Code::Blocks. Additionally, you can switch between these configurations. The Figure 1.6 on page 10 shows the dialogue which you can open via 'Environment Varibales' under 'Settings' →'Environment' . A configuration is created via the 'Create' button.



[✓] MYPATH = C:\Winnt;C:\Winnt\system32;C:\Mingw\bin
[✓] PATH = $(MYPATH)

Select/Create/Remove envvar - set: myconfig

NOTE: These variables are applied on startup (or manually) within the focus of Code::Blocks.

Figure 1.6: Environmental variables

Access and scope of the environmental variables created here, is limited to Code::Blocks. You can expand these environmental variables just like other Code::Blocks variables via $(NAME).

Note:
A configuration for the environmental variable for each project can be selected in the context menu 'Properties' of the 'EnvVars options' tab.

Example

You can write the used environment into a postbuild Step (see section 1.6 on page 4) in a file `<project>.env` and archive it within your project.

```
cmd /c echo %PATH% > project.env
```

or under Linux

```
echo $PATH > project.env
```

1.11.2 Switching between projects

If several projects or files are opened at the same time, the user needs a way to switch quickly between the projects or files. Code::Blocks has a number of shortcuts for such situations.

Alt-F5 Activates the previous project from the project view.

Alt-F6 Activates the next project from the project view.

F11 Switches within the editor between a source file `<name>.cpp` and the corresponding header file `<name>.h`

1.11.3 Extended settings for compilers

During the build process of a project, the compiler messages are displayed in the Messages window in the Build Log tab. If you wish to receive detailed information, the display can be extended. For this purpose click 'Settings' →'Compiler and Debugger' and select 'Other Settings' in the drop-down field.

Take care that the correct compiler is selected. The 'Full command line' setting in the Compiler Logging field outputs the complete information in the Build Log. In addition, this output can be logged in a HTML file. For this purpose select 'Save build log to HTML file when finished'. Furthermore, Code::Blocks offers a progress bar for the build process in the Build Log window which can be activated via the 'Display build progress bar' setting.

1.11.4 Zooming within the editor

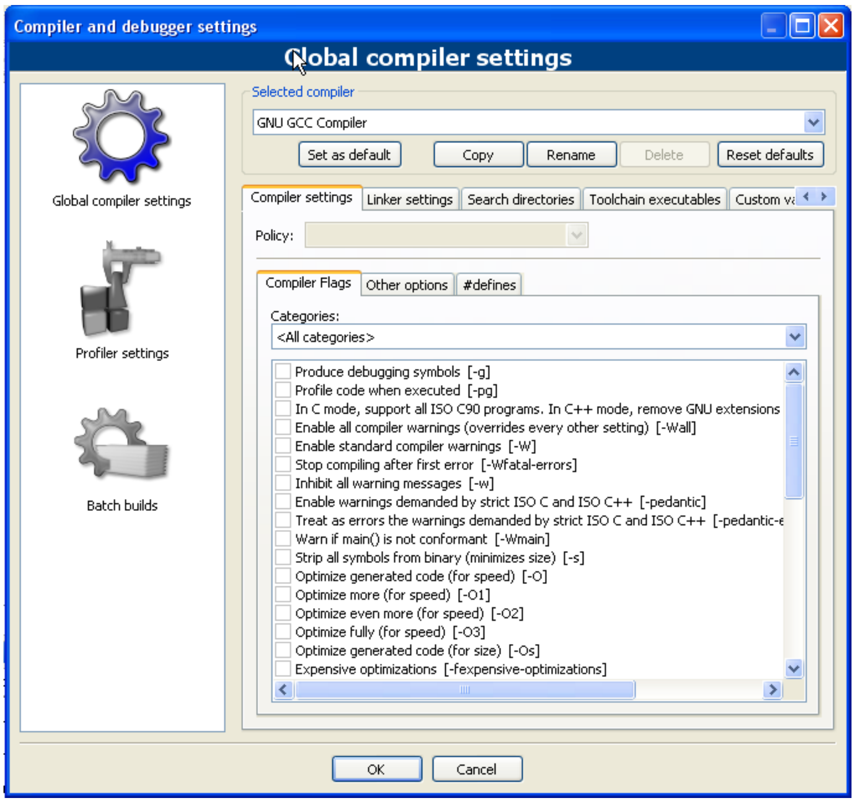
Code::Blocks offers a very efficient editor. This editor allows you to change the size in which the opened text is displayed. If you use a mouse with a jog dial, you only need to press the Ctrl key and scroll via the jog dial to zoom in and out of the text.

1.11.5 Including libraries

In the build options of a project, you can add the used libraries via the 'Add' button in the 'Link libraries' entry of the 'Linker Settings'. In doing so, you can either use the absolute path to the library or just give the name without the `lib` prefix and file extension.

Example

For a library called `<path>\libs\lib<name>.a`, just write `<name>`. The linker with the corresponding search paths will then include the libraries correctly.



GNU GCC Compiler

<All categories>

Produce debugging symbols [-g]
Profile code when executed [-pg]
In C mode, support all ISO C90 programs. In C++ mode, remove GNU extensions
Enable all compiler warnings (overrides every other setting) [-Wall]
Enable standard compiler warnings [-W]
Stop compiling after first error [-Wfatal-errors]
Inhibit all warning messages [-w]
Enable warnings demanded by strict ISO C and ISO C++ [-pedantic]
Treat as errors the warnings demanded by strict ISO C and ISO C++ [-pedantic-e
Warn if main() is not conformant [-Wmain]
Strip all symbols from binary (minimizes size) [-s]
Optimize generated code (for speed) [-O]
Optimize more (for speed) [-O1]
Optimize even more (for speed) [-O2]
Optimize fully (for speed) [-O3]
Optimize generated code (for size) [-Os]
Expensive optimizations [-fexpensive-optimizations]

Figure 1.7: Setting detail information

1.11.6 Object linking order

During compiling, objects `name.o` are created from the sources `name.c/cpp`. The linker then binds the individual objects into an application `name.exe` or for the embedded systems `name.elf`. In some cases, it might be desirable to predefine the order in which the objects will be linked. In Code::Blocks, this can be achieved by assigning priorities. In the context menu 'Properties' , you can define the priorities of a file in the Build tab. A low priority will cause the file to be linked earlier.

1.11.7 Autosave

Code::Blocks offers ways of automatically storing projects and source files, or of creating backup copies. This feature can be activated in the menu 'Settings' →'Environment' →'Autosave' . In doing so, 'Save to .save file' should be specified as the method for creating the backup copy.

1.11.8 Settings for file extensions

In Code::Blocks, you can choose between several ways of treating file extensions. The settings dialogue can be opened via 'Settings' →'Files extension handling' . You can either use the applications assigned by Windows for each file extension (open it with the associated application), or change the setting for each extensions in such a way that either a user-defined program will start (launch an external program), or the file will be opened in the Code::Blocks editor (open it inside Code::Blocks editor).

Note:

If a user-defined program is assigned to a certain file extension, the setting 'Disable Code::Blocks while the external program is running' should be deactivated because otherwise Code::Blocks will be closed whenever a file with this extension is opened.

1.12 Code::Blocks at the command line

IDE Code::Blocks can be executed from the command line without a graphic interface. In such a case, there are several switches available for controlling the build process of a project. Since Code::Blocks is thus scriptable, the creation of executables can be integrated into your own work processes.

```
codeblocks.exe /na /nd --no-splash-screen --built <name>.cbp --target='Release'
```

`<filename>`	Specifies the project `*.cbp` filename or workspace `*.workspace` filename. For instance, <filename> may be `project.cbp`. Place this argument at the end of the command line, just before the output redirection if there is any.
`/h, --help`	Shows a help message regarding the command line arguments.
`/na, --no-check-associations`	Don't perform any file association checks (Windows only).
`/nd, --no-dde`	Don't start a DDE server (Windows only).
`/ns, --no-splash-screen`	Hides the splash screen while the application is loading.
`/d, --debug-log`	Display the debug log of the application.
`--prefix=<str>`	Sets the shared data directory prefix.
`/p, --personality=<str>, --profile=<str>`	Sets the personality to use. You can use ask as the parameter to list all available personalities.
`--rebuild`	Clean and build the project or workspace.
`--build`	Build the project or workspace.
`--target=<str>`	Sets target for batch build. For example `--target='Release'`.
`--no-batch-window-close`	Keeps the batch log window visible after the batch build is completed.
`--batch-build-notify`	Shows a message after the batch build is completed.

`> <build log file>`
Placed in the very last position of the command line, this may be used to redirect standard output to log file. This is not a codeblock option as such, but just a standard DOS/*nix shell output redirection.

1.13 Shortcuts

Even if an IDE such as Code::Blocks is mainly handled by mouse, keyboard shortcuts are nevertheless a very helpful way of speeding up and simplifying work processes. In the below table, we have collected some of the available keyboard shortcuts.

1.13.1 Editor

Function	Shortcut Key
Undo last action	Ctrl+Z
Redo last action	Ctrl+Shift+Z
Swap header / source	F11
Comment highlighted code	Ctrl+Shift+C
Uncomment highlighted code	Ctrl+Shift+X
Auto-complete / Abbreviations	Ctrl+Space/Ctrl+J
Toggle bookmark	Ctrl+B
Goto previous bookmark	Alt+PgUp
Goto next bookmark	Alt+PgDown

This is a list of shortcuts provided by the Code::Blocks editor component. These shortcuts cannot be rebound.

Create or delete a bookmark	Ctrl+F2
Go to next bookmark	F2
Select to next bookmark	Alt+F2
Find selection.	Ctrl+F3
Find selection backwards.	Ctrl+Shift+F3
Find matching preprocessor conditional, skipping nested ones.	Ctrl+K

1.13.2 Files

Function	Shortcut Key
New file or project	Ctrl+N
Open existing file or project	Ctrl+O
Save current file	Ctrl+S
Save all files	Ctrl+Shift+S
Close current file	Ctrl+F4/Ctrl+W
Close all files	Ctrl+Shift+F4/Ctrl+Shift+W

1.13.3 View

Function	Shortcut Key
Show / hide Messages pane	F2
Show / hide Management pane	Shift+F2
Activate prior (in Project tree)	Alt+F5
Activate next (in Project tree)	Alt+F6

1.13.4 Search

Function	Shortcut Key
Find	Ctrl+F
Find next	F3
Find previous	Shift+F3
Find in files	Crtl+Shift+F
Replace	Ctrl+R
Replace in files	Ctrl+Shift+R
Goto line	Ctrl+G
Goto file	Alt+G
Goto function	Ctrl+Alt+G

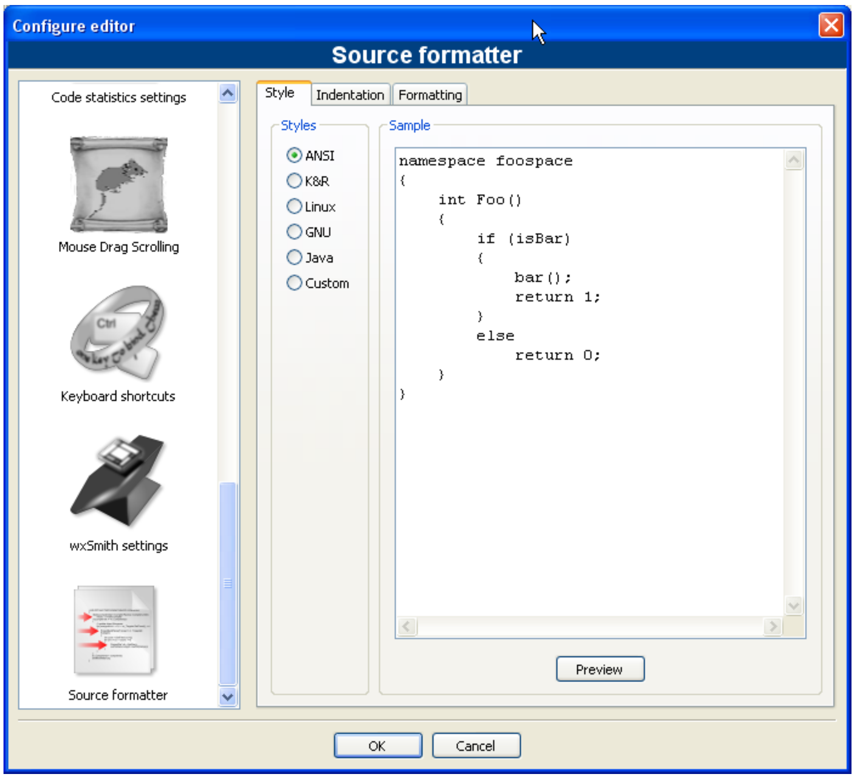
1.13.5 Build

Function	Shortcut Key
Build	Ctrl+F9
Compile current file	Ctrl+Shift+F9
Run	Ctrl+F10
Build and Run	F9
Rebuild	Ctrl+F11

2 Plugins

2.1 Astyle

Artistic Style is a source code indenter, source code formatter, and source code beautifier for the C, C++, C# programming languages. It can be used to select different styles of coding rules within Code::Blocks.



Styles: [✓ ANSI] K&R Linux GNU Java Custom

```
namespace foospace
{
    int Foo()
    {
        if (isBar)
        {
            bar();
            return 1;
        }
        else
            return 0;
    }
}
```

Figure 2.1: Formating your source code

When indenting source code, we as programmers have a tendency to use both spaces and tab characters to create the wanted indentation. Moreover, some editors by default insert spaces instead of tabs when pressing the tab key, and other editors have the ability to prettify lines by automatically setting up the white space before the code on the line, possibly inserting spaces in a code that up to now used only tabs for indentation.

Since the number of space characters shown on screen for each tab character in the source code changes between editors, one of the standard problems programmers are facing when moving from one editor to another is that code containing both spaces and tabs that was up to now perfectly indented, suddenly becomes a mess to look at when changing to another editor. Even if you as a programmer take care to ONLY use spaces or tabs, looking at other people's source code can still be problematic.

To address this problem, Artistic Style was created - a filter written in C++ that automatically re-indents and re-formats C / C++ / C# source files.

Note:

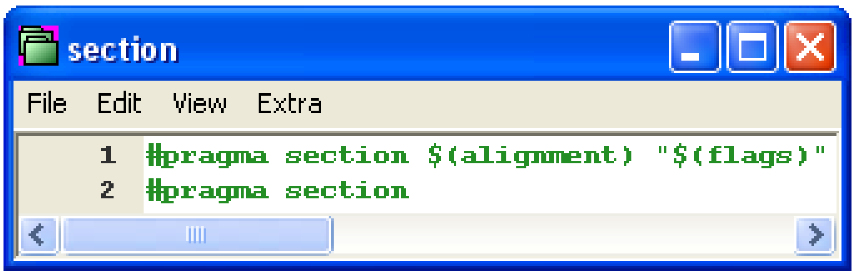
When copying code, for example from the internet or a manual, this code will automatically be adapted to the coding rules in Code::Blocks.

2.2 CodeSnippets

The CodeSnippets plug-in makes it possible to structure text modules and links to files according to categories in a tree view. The modules are used for storing often used files and constructs in text modules and managing them in a central place. Imagine the following situation: A number of frequently used source files are stored in different directories of the file system. The CodeSnippets window provides the opportunity to create categories, and below the categories, links to the required files. With these features, you can control the access to the files independently from where they are stored within the file system, and you can navigate quickly between the files without the need to search the whole system.

The list of text modules and links can be stored in the CodeSnippets window by right-clicking and selecting 'Save Index' from the context menu. The file `codesnippets.xml` which will be created by this procedure, can then be found in the `codeblocks` subdirectory of your `Documents and Settings\Application data` directory. Under Linux, this information is stored in the `.codeblocks` subdirectory of your HOME directory. The Code::Blocks configuration files will be loaded during the next start-up. If you wish to save the content of CodeSnippets at a different location, select the 'Save Index As' entry. To load this file, select 'Load Index File' during the next start-up of Code::Blocks or include the directory in the 'Settings' context menu under 'Snippet Folder'.

For including a category, use the 'Add SubCategory' menu. A category can contain Snippets (text modules) or File Links. A text module is created via the 'Add Snippet' command in the context menu. The content is integrated into the text module by selecting the text passage in the Code::Blocks editor and dragging and dropping it onto the module. Double-clicking the newly included entry or selecting 'Edit Text' will open an editor for the content.



```
#pragma section $(alignment) "$(flags)"
#pragma section
```

Figure 2.2: Editing a text module

Output of a text module is handled in Code::Blocks via the context menu command 'Apply' or by dragging and dropping into the editor. Under Windows, the contents of a Snippet can also be dragged and dropped into other applications.

Beyond this, text modules can be parametrised by <name> variables which can be accessed via `$(name)` (see Figure 2.2 on page 18). The values of the variables can be retrieved in an entry field if the text module is called via the context menu command 'Apply'.

Besides the text modules, links to files can also be created. If, after having created a text module, you click the context menu command 'Properties', then you can select the link target by clicking the 'Link target' button. This procedure will automatically convert the text module into a link to a file. In CodeSnippets, all text modules will be marked by a T symbol, links to a file by an F symbol. If you prefer to process this file with an external editor instead of with the integrated Code::Blocks editor, just drag the link entry from the CodeSnippets view into the 'Open files list' window. After editing and saving the file, drag the corresponding entry from 'Open files list' back to where the link entry is located within the CodeSnippets view.

Note:

If you have chosen the setting 'open it with the associated application' under 'Settings' →'Environment' for file extension handling, the application assigned by Windows for the file extension will be used (see subsection 1.11.8 on page 12).

Example: With this setting, if you drag a link to a pdf file from the codesnippets view into the 'Open files list' window, a pdf viewer will be started automatically. This method makes it possible for a user to access files which are spread over the whole network, such as wiring plans, documentations etc., with the common applications, simply via the link. The content of the codesnippets is stored in the file `codesnippets.xml`, the configuration is stored in the file `codesnippets.ini` in your `application data` directory. This ini file will, for example, contain the path of the file `codesnippets.xml`.

Code::Blocks supports the usage of different profiles. These profiles are called personalities. A profile is created by starting Code::Blocks with the command line option `--personality=<profile>`. Then the settings will not be stored in the file `default.conf`, but in `<personality>.conf` in your `application data` directory instead. The Codesnippets plugin will then store its settings in the file `<personality>.codesnippets.ini`. Now, if you load a new content `<name.xml>` in the Codesnippets settings via 'Load Index File', this content will be stored in the corresponding ini file. The advantage of this method lies in the fact that in case of different profiles, different configurations for text modules and links can be managed.

The plug-in offers an additional search function for navigating between the categories and Snippets. The scope for searching Snippets, categories or Snippets and categories can be adjusted. By entering the required search expression, the corresponding entry is automatically selected in the view. Figure 2.3 on page 20 shows a typical display in the CodeSnippets window.

Note:

When using voluminous text modules, the content of these modules should be saved in files via 'Convert to File Link' in order to reduce memory usage within the system.

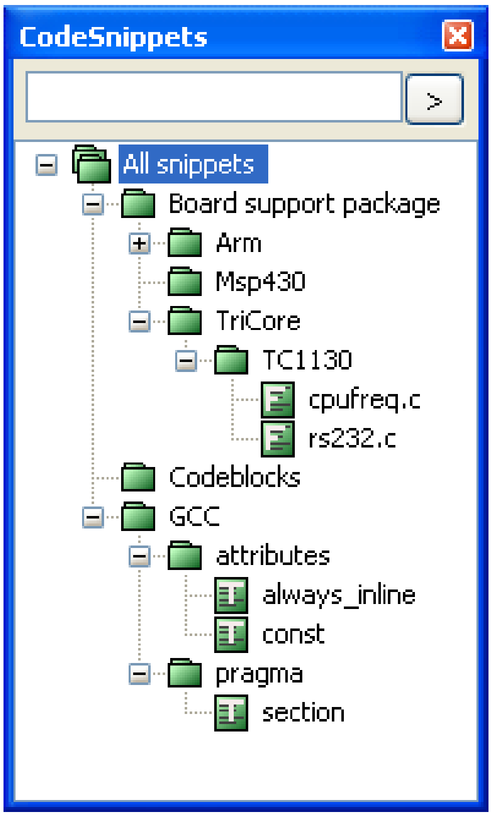
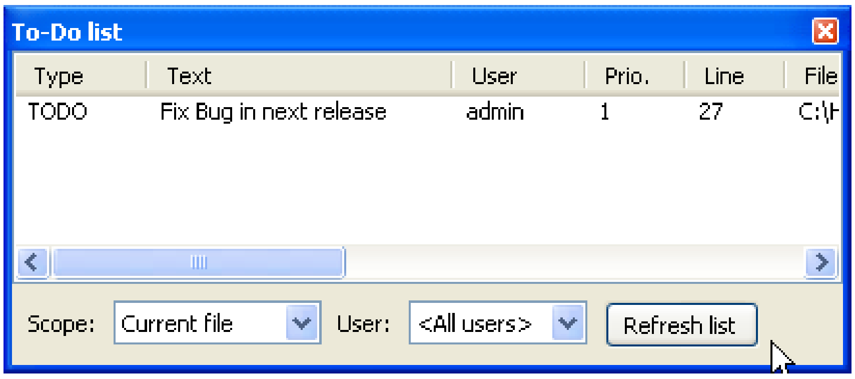


Figure 2.3: CodeSnippets View

2.3 ToDo List

In complex software projects, where different users are involved, there is often the requirement of different tasks to be performed by different users. For this purpose, Code::Blocks offers a Todo List. This list can be opened via 'View' →'ToDo list' , and contains the tasks to be performed, together with their priorities, types and the responsible users. The list can be filtered for tasks, users and/or source files.



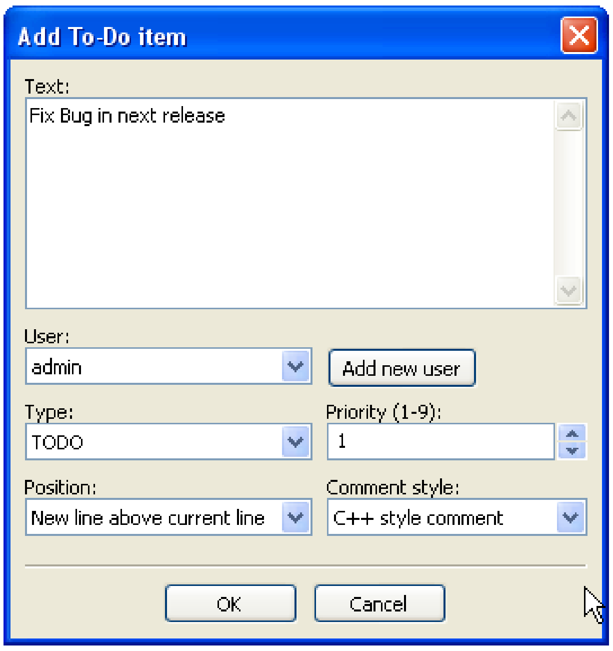
Type	Text	User	Prio.	Line	File
TODO	Fix Bug in next release	admin	1	27	C:\H

Figure 2.4: Displaying the ToDo List

If the sources are opened in Code::Blocks, a Todo can be added to the list via the context menu command 'Add To-Do item'. A comment will be added in the selected line of the source code.

```
// TODO (user#1#): add new dialog for next release
```

When adding a To-Do, a dialogue box will open where the following settings can be made (see Figure 2.5 on page 21).



Text: Fix Bug in next release
User: admin
Type: TODO
Priority (1-9): 1
Position: New line above current line
Comment style: C++ style comment

Figure 2.5: Dialogue for adding a ToDo

User User name <user> in the operating system. Tasks for other users can also be created here. In doing so, the corresponding user name has to be created by Add new. The assignment of a Todo is then made via the selection of entries listed for the User.

> **Note:**
>
> Note that the Users have nothing to do with the Personalities used in Code::Blocks.

Type By default, type is set to Todo.

Priority The importance of tasks can be expressed by priorities (1 - 9) in Code::Blocks.

Position This setting specifies whether the comment is to be included before, after or at the exact position of the cursor.

Comment Style A selection of formats for comments (e.g. doxygen).

2.4 Source Code Exporter

The necessity occurs frequently of transferring source code to other applications or to e-mails. If the text is simply copied, formatting will be lost, thus rendering the text very unclear. The Code::Blocks Export function serves as a remedy for such situations. The required format for the export file can be selected via 'File' →'Export' . The program will then adopt the file name and target directory from the opened source file and propose these for saving the export file. The appropriate file extension in each case will be determined by the export format. The following formats are available.

html A text-based format which can be displayed in a web browser or in word processing applications.

rtf The Rich Text format is a text-based format which can be opened in word processing applications such as Word or OpenOffice.

odt Open Document Text format is a standardised format which was specified by Sun and O'Reilly. This format can be processed by Word, OpenOffice and other word processing applications.

pdf The Portable Document Format can be opened by applications such as the Acrobat Reader.

2.5 Thread Search

Via the 'Search' →'Thread Search' menu, the appropriate plug-in can be shown or hidden as a tab in the Messages Console. In Code::Blocks, a preview can be displayed for the occurrence of a character string in a file, workspace or directory. In doing so, the list of search results will be displayed on the right-hand side of the ThreadSearch Console. By clicking an entry in the list, a preview is displayed on the left-hand side. By double-clicking in the list, the selected file is opened in the Code::Blocks editor.

Note:

Note that the scope of file extensions to be included in the search, is preset and might have to be adjusted.

2.6 File Browser

2.7 Interpreter Languages

2.8 BrowseTracks

3 Variable Expansion

Code::Blocks differentiates between several types of variables. These types serve the purpose of configuring the environment for creating a program, and at the same of improving the maintainability and portability. Access to the Code::Blocks variables is achieved via `$<name>`.

Envrionment Variable are set during the startup of Code::Blocks. They can modify system environment variables such as PATH. This can be useful in cases where a defined environment is necessary for the creation of projects. The settings for environment variables in Code::Blocks are made at 'Settings' →'Environment' →'Environment Variables' .

Builtin Variables are predefined in Code::Blocks, and can be accessed via their names (see section 3.2 on page 24 for details).

Command Macros . This type of variables is used for controlling the build process. For further information please refer to section 3.4 on page 26.

Custom Variables are user-defined variables which can be specified in the build options of a project. Here you can, for example define your derivative as a variable MCU and assign a corresponding value to it. Then set the compiler option `-mcpu=$(MCU)`, and Code::Blocks will automatically replace the content. By this method, the settings for a project can be further parametrised.

Global Variables are mainly used for creating Code::Blocks from the sources or developments of wxWidgets applications. These variables have a very special meaning. In contrast to all others if you setup such a variables and share your project file with others that have *not* setup this GV Code::Blocks will ask the user to setup the variable. This is a very easy way to ensure the 'other developer' knows what to setup easily. Code::Blocks will ask for all path's usually necessary.

3.1 Syntax

Code::Blocks treats the following functionally identical character sequences inside pre-build, post-build, or build steps as variables:

- $VARIABLE
- $(VARIABLE)
- ${VARIABLE}
- %VARIABLE%

Variable names must consist of alphanumeric characters and are not case-sensitive. Variables starting with a single hash sign (#) are interpreted as global user variables (see section 3.7 on page 27 for details). The names listed below are interpreted as built-in types.

Variables which are neither global user variables nor built-in types, will be replaced with a value provided in the project file, or with an environment variable if the latter should fail.

Note:

Per-target definitions have precedence over per-project definitions.

3.2 List of available built-ins

The variables listed here are built-in variables of Code::Blocks. They cannot be used within source files.

3.2.1 Files and directories

`$(PROJECT_FILENAME), $(PROJECT_FILE), $(PROJECTFILE)`
The filename of the currently compiled project.

`$(PROJECT_NAME)`
The name of the currently compiled project.

`$(PROJECT_DIR), $(PROJECTDIR), $(PROJECT_DIRECTORY)`
The common top-level directory of the currently compiled project.

`$(ACTIVE_EDITOR_FILENAME)`
The filename of the file opened in the currently active editor.

`$(ACTIVE_EDITOR_DIRNAME)`
the directory containing the currently active file (relative to the common top level path).

`$(ACTIVE_EDITOR_STEM)`
The base name (without extension) of the currently active file.

`$(ACTIVE_EDITOR_EXT)`
The extension of the currently active file.

`$(ALL_PROJECT_FILES)`
A string containing the names of all files in the current project.

`$(MAKEFILE)` The filename of the makefile.

`$(CODEBLOCKS), $(APP_PATH), $(APPPATH), $(APP-PATH)`
The path to the currently running instance of Code::Blocks.

`$(DATAPATH), $(DATA_PATH), $(DATA-PATH)`
The 'shared' directory of the currently running instance of Code::Blocks.

`$(PLUGINS)` The `plugins` directory of the currently running instance of Code::Blocks.

3.2.2 Build targets

`$(FOOBAR_OUTPUT_FILE)`
The output file of a specific target.

`$(FOOBAR_OUTPUT_DIR)`
The output directory of a specific target.

`$(FOOBAR_OUTPUT_BASENAME)`
The output file's base name (no path, no extension) of a specific target.

`$(TARGET_OUTPUT_DIR)`
The output directory of the current target.

`$(TARGET_OBJECT_DIR)`
The object directory of the current target.

`$(TARGET_NAME)`
The name of the current target.

`$(TARGET_OUTPUT_FILE)`
The output file of the current target.

`$(TARGET_OUTPUT_BASENAME)`
The output file's base name (no path, no extension) of the current target.

`$(TARGET_CC)`, `$(TARGET_CPP)`, `$(TARGET_LD)`, `$(TARGET_LIB)`
The build tool executable (compiler, linker, etc) of the current target.

3.2.3 Language and encoding

`$(LANGUAGE)`	The system language in plain language.
`$(ENCODING)`	The character encoding in plain language.

3.2.4 Time and date

`$(TDAY)`	Current date in the form YYYYMMDD (for example 20051228)
`$(TODAY)`	Current date in the form YYYY-MM-DD (for example 2005-12-28)
`$(NOW)`	Timestamp in the form YYYY-MM-DD-hh.mm (for example 2005-12-28-07.15)
`$(NOW_L)`	] Timestamp in the form YYYY-MM-DD-hh.mm.ss (for example 2005-12-28-07.15.45)
`$(WEEKDAY)`	Plain language day of the week (for example 'Wednesday')

`$(TDAY_UTC)`, `$(TODAY_UTC)`, `$(NOW_UTC)`, `$(NOW_L_UTC)`, `$(WEEKDAY_UTC)`
These are identical to the preceding types, but are expressed relative to UTC.

3.2.5 Random values

`$(COIN)`	This variable tosses a virtual coin (once per invocation) and returns 0 or 1.
`$(RANDOM)`	A 16-bit positive random number (0-65535)

3.3 Script expansion

For maximum flexibility, you can embed scripts using the [[ ]] operator as a special case of variable expansion. Embedded scripts have access to all standard functionalities available to scrips and work pretty much like bash backticks (except for having access to Code::Blocks namespace). As such, scripts are not limited to producing text output, but can also manipulate Code::Blocks state (projects, targets, etc.).

Note:

Manipulating Code::Blocks state should be implemented rather with a pre-build script than with a script.

Example with Backticks

```
objdump -D `find . -name *.elf` > name.dis
```

The expression in backticks returns a list of all executables `*.elf` in any subdirectories. The result of this expression can be used directly by `objdump`. Finally the output is piped to a file named `name.dis`. Thus, processes can be automatted in a simple way without having to program any loops.

Example using Script

The script text is replaced by any output generated by your script, or discarded in case of a syntax error.

Since conditional evaluation runs prior to expanding scripts, conditional evaluation can be used for preprocessor functionalities. Built-in variables (and user variables) are expanded after scripts, so it is possible to reference variables in the output of a script.

```
[[ print(GetProjectManager().GetActiveProject().GetTitle()); ]]
```

inserts the title of the active project into the command line.

3.4 Command Macros

`$compiler`	Access to name of the compiler executable.
`$linker`	Access to name of the linker executable.
`$options`	Compiler flags
`$link_options`	Linker flags

`$includes`	Compiler include paths
`$libdirs`	Linker include paths
`$libs`	Linker libraries
`$file`	Source file
`$object`	Object file
`$exe_output`	Executable output file
`$objects_output_dir`	Object Output Directory

3.5 Compile single file

```
$compiler $options $includes -c $file -o $object
```

3.6 Link object files to executable

```
$linker $libdirs -o $exe_output $link_objects $link_resobjects $link_options $libs
```

3.7 Global compiler variables

3.8 Synopsis

Working as a developer on a project which relies on 3rd party libraries involves a lot of unnecessary repetitive tasks, such as setting up build variables according to the local file system layout. In the case of project files, care must be taken to avoid accidentially committing a locally modified copy. If one does not pay attention, this can happen easily for example after changing a build flag to make a release build.

The concept of global compiler variables is a unique new solution for Code::Blocks which addresses this problem. Global compiler variables allow you to set up a project once, with any number of developers using any number of different file system layouts being able to compile and develop this project. No local layout information ever needs to be changed more than once.

3.9 Names and Members

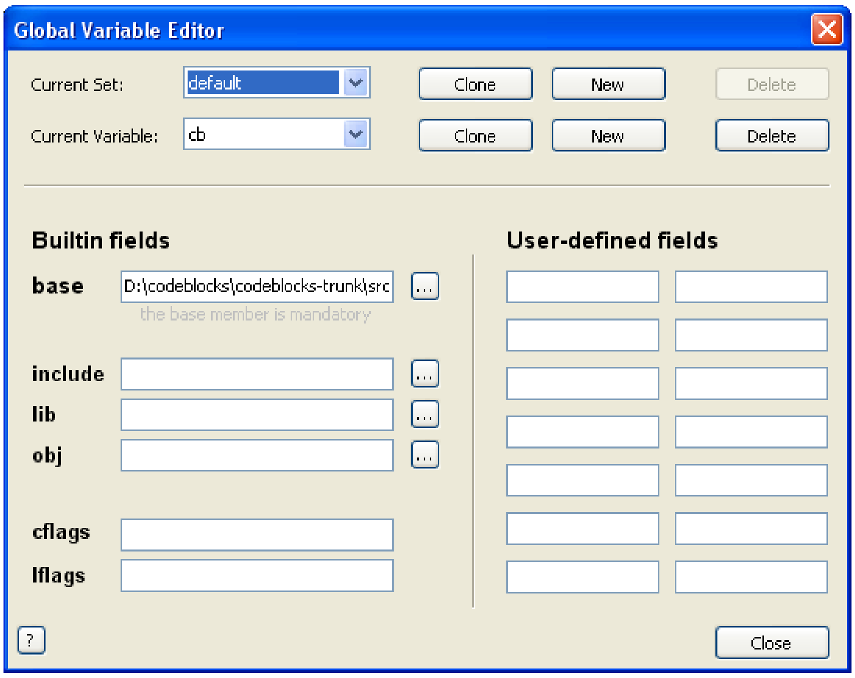
Global compiler variables in Code::Blocks are discriminated from per-project variables by a leading hash sign. Global compiler variables are structured; every variable consists of a name and an optional member. Names are freely definable, while some of the members are built into the IDE. Although you can choose anything for a variable name in principle, it is advisable to pick a known identifier for common packages. Thus the amount of information that the user needs to provide is minimised. The Code::Blocks team provides a list of recommended variables for known packages.

The member base resolves to the same value as the variable name uses without a member (alias).

The members include and lib are by default aliases for base/include and base/lib, respectively. However, a user can redefine them if another setup is desired.

It is generally recommended to use the syntax $(#variable.include) instead of $(#variable)/include, as it provides additional flexibility and is otherwise exactly identical in functionality (see subsection 3.12.1 on page 30 and Figure 3.1 on page 28 for details).

The members cflags and lflags are empty by default and can be used to provide the ability to feed the same consistent set of compiler/linker flags to all builds on one machine. Code::Blocks allows you to define custom variable members in addition to the built-in ones.



Current Set: default

Current Variable: cb

base: D:\codeblocks\codeblocks-trunk\src

the base member is mandatory

Figure 3.1: Global Variable Environment

3.10 Constraints

- Both set and global compiler variable names may not be empty, they must not contain white space, must start with a letter and must consist of alphanumeric characters. Cyrillic or Chinese letters are not alphanumeric characters. If Code::Blocks is given invalid character sequences as names, it might replace them without asking.
- Every variable requires its base to be defined. Everything else is optional, but the base is absolutely mandatory. If you don't define a the base of a variable, it will not be saved (no matter what other fields you have defined).
- You may not define a custom member that has the same name as a built-in member. Currently, the custom member will overwrite the built-in member, but in general, the behaviour for this case is undefined.

- Variable and member values may contain arbitrary character sequences, subject to the following three constraints:
  - You may not define a variable by a value that references the same variable or any of its members
  - You may not define a member by a value that references the same member
  - You may not define a member or variable by a value that references the same variable or member through a cyclic dependency.

Code::Blocks will detect the most obvious cases of recursive definitions (which may happen by accident), but it will not perform an in-depth analysis of every possible abuse. If you enter crap, then crap is what you will get; you are warned now.

Examples

Defining wx.include as $(#wx)/include is redundant, but perfectly legal Defining wx.include as $(#wx.include) is illegal and will be detected by Code::Blocks Defining wx.include as $(#cb.lib) which again is defined as $(#wx.include) will create an infinite loop

3.11 Using Global Compiler Variables

All you need to do for using global compiler variables is to put them in your project! Yes, it's that easy.

When the IDE detects the presence of an unknown global variable, it will prompt you to enter its value. The value will be saved in your settings, so you never need to enter the information twice.

If you need to modify or delete a variable at a later time, you can do so from the settings menu.

Example

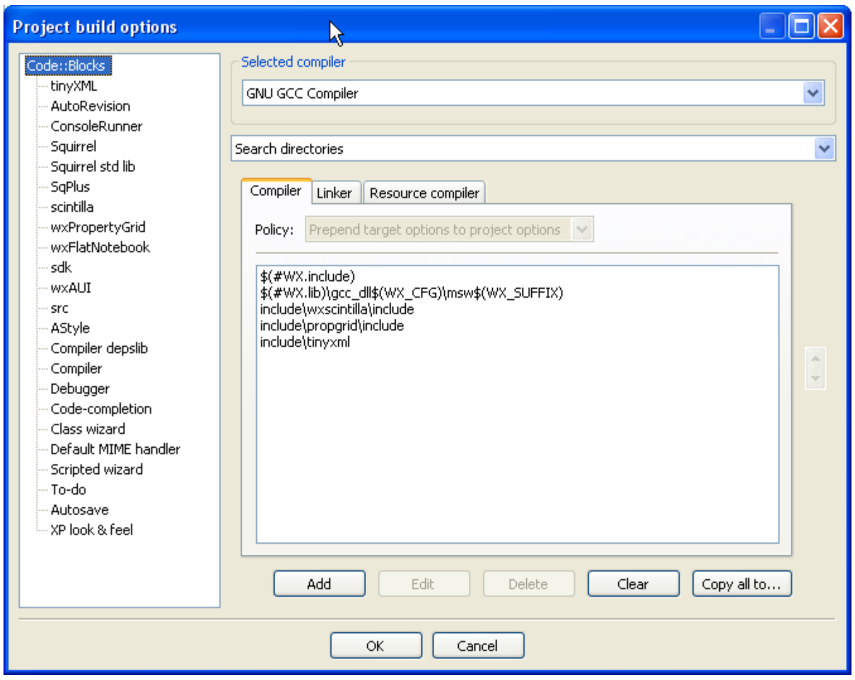
The above image shows both per-project and global variables. WX_SUFFIX is defined in the project, but WX is a global user variable.

3.12 Variable Sets

Sometimes, you want to use different versions of the same library, or you develop two branches of the same program. Although it is possible to get along with a global compiler variable, this can become tedious. For such a purpose, Code::Blocks supports variable sets. A variable set is an independent collection of variables identified by a name (set names have the same constraints as variable names).

If you wish to switch to a different set of variables, you simply select a different set from the menu. Different sets are not required to have the same variables, and identical variables in different sets are not required to have the same values, or even the same custom members.

Another positive thing about sets is that if you have a dozen variables and you want to have a new set with one of these variables pointing to a different location, you are not



Selected compiler: GNU GCC Compiler

Search directories

Policy: Prepend target options to project options

```
$(#WX.include)
$(#WX.lib)\gcc_dll$(WX_CFG)\msw$(WX_SUFFIX)
include\wxscintilla\include
include\propgrid\include
include\tinyxml
```

Figure 3.2: Global Variables

required to re-enter all the data again. You can simply create a clone of your current set, which will then duplicate all of your variables.

Deleting a set also deletes all variables in that set (but not in another set). The `default` set is always present and cannot be deleted.

3.12.1 Custom Members Mini-Tutorial

As stated above, writing `$(#var.include)` and `$(#var)/include` is exactly the same thing by default. So why would you want to write something as unintuitive as `$(#var.include)`?

Let's take a standard Boost installation under Windows for an example. Generally, you would expect a fictional package ACME to have its include files under ACME/include and its libraries under ACME/lib. Optionally, it might place its headers into yet another subfolder called acme. So after adding the correct paths to the compiler and linker options, you would expect to `#include <acme/acme.h>` and link to `libacme.a` (or whatever it happens to be).

4 Building Code::Blocks from sources

4.1 Introduction

This article will describe the process used in creating the nightly builds, and can be used as a guideline if you want to build Code::Blocks yourself. It is described as a sequence of actions.

In order to perform our build tasks, we will need several tools. Let's create an ingredient list for our cooking experiments

- a compiler
- an initial build system
- the Code::Blocks sources
- zip program
- svn (version control system)
- wxWidgets

4.1.1 WIN32

Since the Code::Blocks developers build Code::Blocks using GCC, we might as well use that one under windows. The easiest and cleanest port is MinGW. This is the compiler distributed with Code::Blocks when you download the official package. We will stick to version 3.4.4, which works nicely.

First, a brief explanation of MinGW components:

gcc-core the core of the GCC suite

gcc-g++ the c++ compiler

mingw Runtime implementation of the run time libraries

mingw utils several utilities (implementation of smaller programs that GCC itself uses)

win32Api the APIs for creating Windows programs

binutils several utilities used in build environments

make the Gnu make program, so you can build from make files

GDB the Gnu debugger

I would suggest extracting (and installing for the GDB) everything in the `C:\MinGW` directory. The remainder of this article will assume that this is where you have put it. If you already have an installation of Code::Blocks that came bundled with MinGW, I still advise you to install MinGW as described here. A compiler does not belong under the directory tree of an IDE; they are two separate things. Code::Blocks just brings it along in the official versions so that the average user does not need to bother with this process.

You may need to add the `bin` directory of your MinGW installation to your path. An easy way to do this is with the following command at the command prompt:

```
set path=%PATH%;C:\MinGW\bin;C:\MinGW\mingw32\bin;
```

4.1.2 Initial Build System

For Code::Blocks a project description `CodeBlocks.cbp` is available. If you load this project file in Code::Blocks then you are able to build Code::Blocks from sources. All we need to do is get hold of a pre-built version Code::Blocks.

First, download a nightly build. You can make your selection from here. The nightly builds are unicode versions, containing the core and contributed plug-ins.

Next, unpack the 7-zip file to any directory you like. If you don't have 7-zip, you can download it for free from the 7-zip website.

Now, Code::Blocks needs one more `dll` to work correctly: the WxWidgets dll. You can also download it at the nightly builds forum. Just unzip it into the same directory that you unpacked the Code::Blocks nightly build. It also needs the `mingwm10.dll`. It's in the bin directory of our MinGW installation. So, it's important to make sure the bin directory of your MinGW installation is in your path variable.

Finally, start up this new nightly build of Code::Blocks. It should discover the MinGW compiler we just installed.

4.1.3 Version Control System

In order to be able to retrieve the latest and greatest Code::Blocks sources, we need to install a Version Control System.

The Code::Blocks developers provide their sources through the SVN version control system. So, we need a client to access their svn repository of sources. A nice, easy client for Windows is TortoiseSVN, which is freely available. Download and install it, keeping all suggested settings.

Now, go create a directory wherever you like, for example `D:\projects\CodeBlocks`. Right click on that directory and choose from the pop-up menu: svn-checkout. In the dialog that pops up, fill in the following information for Url of Repository:

`svn://svn.berlios.de/codeblocks/trunk`

and leave all other settings as they are.

Now be patient while TortoiseSVN retrieves the most recent source files from the Code::Blocks repository into our local directory. Yes; all those Code::Blocks sources are coming your way!

For more info on SVN settings, see info on SVN settings. If you don't like an Explorer integration or look for a cross-plattform client you might want to have a look at RapidSVN..

4.1.4 wxWidgets

WxWidgets is a platform abstraction that provides an API to support many things such as GUI, sockets, files, registry functionality. By using this API, you can create a platform independent program.

Code::Blocks is a wxWidgets (here after: wx) application, that means if you want to run Code::Blocks you needed the wx functionality. This can be provided in a couple of ways. It could be a `.dll` or a static library. Code::Blocks uses wx as a dll and this dll can also be downloaded from the nightly build section of the forum.

However, if we want to build a wx application, we need to include the headers of the wx sources. They tell the compiler about the functionality of wx. In addition to those header files, our application needs to link to the wx import libraries. Well, let's take it step by step.

Wx is provided as a zip file of it's sources, so we need to build that ourselves. We already shopped for the MinGW compiler, so we have all the tools we need at hand.

Next, let's unzip the wx sources into `C:\Projects` so we will end up with a wx root directory like this: `C:\Projects\wxWidgets-2.8.3`. Next unzip the patch into the same directory letting it overwrite files. Note that we are going to refer to the wx root directory from now on as <wxDir>

Now, we are going to build the wxWidgets. This is how we do it:

First, make sure `C:\MingGW\bin` is in your path, during the build some programs will be called that reside in the the MinGW\bin directory. Also, Make has to be version 3.80 or above.

Now it is time to compile wxWidgets. Open the command prompt and change to the wxWidgets directory:

```
cd <wxDir>\build\msw
```

We are now in the right place. We are first going to clean up the source:

```
mingw32-make -f makefile.gcc SHARED=1 MONOLITHIC=1 BUILD=release UNICODE=1 clean
```

Once everything is clean, we can compile wxWidgets:

```
mingw32-make -f makefile.gcc SHARED=1 MONOLITHIC=1 BUILD=release UNICODE=1
```

This is going to take some time.

For making the debug build, follow these steps:

- Clean any previous compilation with

  ```
  mingw32-make -f makefile.gcc SHARED=1 MONOLITHIC=1 BUILD=debug UNICODE=1 clean
  ```

- Compile with

```
        mingw32-make -f makefile.gcc SHARED=1 MONOLITHIC=1 BUILD=debug UNICODE=1
```

Well have a little look in the directory (`<wxDir>\lib\gcc_dll`) now. The import libraries and the dll have shown up and there should also a `mswu\wx` subdirectory at that position containing `setup.h`.

Congratulations! You have just built wxWidgets!

Let's do some more preliminary tasks before we get to the real deal of compiling Code::Blocks.

4.1.5 Zip

During the build of Code::Blocks, several resources are going to be zipped in zip files. Therefore, the build process should have access to a zip.exe. We have to download that zip.exe and put it somewhere in our path. A good place is: `MingW\bin`.

You can download zip.exe for free from this site and this is a direct link(32bit) to the most recent version at the time of this writing.

Once downloaded, simply extract zip.exe to the appropriate location.

4.1.6 Building Codeblocks

win32 Starting `update_revision.bat`

linux Starting `update_revision.sh`

With this function the SUN revision of the Nightly Builts is updated in the sources. The file can be found in the main directory of the Code::Blocks sources.

4.1.7 WIN32

Now, open the project `CodeBlocks.cbp` in Code::Blocks. Generate Code::Blocks by starting the build process. After the creation of Code::Blocks, the generated files with the debug information can be found in the `devel` subdirectory. By calling the batch file `update.bat` from the source directory, the files are copied to the `output` subdirectory and the debug information is stripped.

4.1.8 LINUX

When generating under Linux, the following steps are necessary. In this example we assume that you are in the Code::Blocks source directory. Under Linux, the environment variable PKG_CONFIG_PATH must be set. The <prefix> directory has to contain the `codeblocks.pc` file.

```
PKG_CONFIG_PATH=$PKG_CONFIG_PATH:<prefix>

sh update_revsion.sh
./bootstrap
./configure --enable-contrib --prefix=<install-dir>
make
make install (root)
```

4.1.9 Generate a plugins

Afterwards, configure the global variables via 'Settings' →'Global Variables' .

Variable cb

For the `cb` variable, set the `base` entry to the source directory of Code::Blocks.

```
<prefix>/codeblocks/src
```

Variable wx

For the `wx` variable, set the `base` entry to the source directory of wx (e.g.

```
C:\Programme\wxWidgets-2.8.3
```

In the Code::Blocks project, the project variable `WX_SUFFIX` is set to `u`. This means that, when generating Code::Blocks linking will be carried out against the `*u_gcc_custom.dll` library. The official nightly Builts of Code::Blocks will be linked against `gcc_cb.dll`. In doing so, the layout is as follows.

```
gcc_<VENDOR>.dll
```

The <VENDOR> variable is set in the configuration file `compiler.gcc`. To ensure, that a distinction is possible between the officially generated Code::Blocks and those generated by yourself, the default setting `VENDOR=custom` should never be changed.

Afterwards create the workspace `ContribPlugins.cbp` via 'Project' →'Build workspace' . Then execute `update.bat` once more.

4.1.10 Linux

Variable `wx` bei globalen Variablen konfigurieren. Configure the `wx` variable with the global variables.

base /usr

include /usr/include/wx-2.8

lib /usr/lib

debug Code::Blocks. Start Code::Blocks in the output directory and load `CodeBlocks.cbp` as the project. Then set the breakpoint and start with 'Debug and Run' (f8).

4.1.11 Contributed Plugins

This brings us to the last preliminary task. The Code::Blocks code can be divided into 2 major parts: the core with internal plug-ins, and the contributed plug-ins. You always need to build the core/internal parts before building the contrib part.

To build the internal part, you can use the Code::Blocks project file which you can find at: `<cbDir>\src\CodeBlocks.cbp`. Our Code::Blocks master directory is from now one mentioned as <cbDir>, by the way. A workspace is something that groups several projects together. To build the contrib plug-ins, they can be found at

```
<cbDir>\src\ContribPlugins.workspace
```

But, let's create a worksapce containing everything. Let's put that workspace in the master directory <cbDir>. Just use a regular text editor and create a file with the name `CbProjects.workspace` and give it the following content :

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<CodeBlocks_workspace_file>
 <Workspace title="Workspace">
   <Project filename="src\CodeBlocks.cbp" active="1" />
   <Project filename="src\plugins\contrib\codestat\codestat.cbp" />
   <Project filename="src\plugins\contrib\copystrings\copystrings.cbp" />
   <Project filename="src\plugins\contrib\dragscroll\dragscroll.cbp" />
   <Project filename="src\plugins\contrib\devpak_plugin\DevPakPlugin.cbp" />
   <Project filename="src\plugins\contrib\help_plugin\help-plugin.cbp" />
   <Project filename="src\plugins\contrib\keybinder\keybinder.cbp" />
   <Project filename="src\plugins\contrib\profiler\cbprofiler.cbp" />
   <Project filename="src\plugins\contrib\source_exporter\Exporter.cbp" />
   <Project filename="src\plugins\contrib\wxSmith\wxSmith.cbp" />
 </Workspace>
</CodeBlocks_workspace_file>
```

We will use this workspace to build all of Code::Blocks.

4.1.12 Building Code::Blocks

Finally we have arrived at the final step; our final goal. Run the Code::Blocks executable from your nightly build download. Choose Open from the File menu and browse for our above created workspace, and open it up. Be a little patient while Code::Blocks is parsing everything, and Code::Blocks will ask us for 2 global variables, these global variables will tell the nightly Code::Blocks where it can find wxWidgets (remember : header files and import libraries) and where it can find .... Code::Blocks, this is needed for the contrib plug-ins, they need to know (as for any user created plug-in) where the sdk (Code::Blocks header files) are. These are the values in our case :

wx <wxDir> base directory of wxWidgets.

cb `<cbDir>/src` Code::Blocks directory contaning the sources.

Now go to the Project Menu and choose (re)build workspace, and off you go. Watch how Code::Blocks is building Code::Blocks.

Once the build is complete, open up a console in `<cbDir>/src` and run the command `update.bat`. This will transfer all built deliverables from `<cbDir>/src/devel` to `<cbDir>/src/output`. In addition, it will strip out all debugging symbols. This step is very important - never ever forget it.

Now you can copy the wx dll in both that output and the devel directory.

Then you can close Code::Blocks. That was the downloaded nightly remember?

Time to test it. In the output directory, start up the CodeBlocks.exe. If everything went well, you'll have your very own home-built Code::Blocks running.