Code::Blocks

Manual

Version 1.0

HighTec EDV-Systeme GmbH
Stammsitz & Entwicklung
Feldmannstraße 98
D-66119 Saarbrücken
Tel.: 0681/92613-16 Fax: -26
mailto:htc@hightec-rt.com

HighTec EDV-Systeme GmbH
Niederlassung Leonberg
Heidenheimer Straße 14
D-71229 Leonberg
Tel.: 07152/35413-0 Fax: -77
mailto:info@hightec-rt.com

www.hightec-rt.com

Inhaltsverzeichnis

1 Code::Blocks Projektverwaltung

Die Dokumentation für Kapitel 3 auf Seite 25 und Kapitel 4 auf Seite 33 sind offizielle Dokumentationen der Code::Blocks Wiki-Seite und nur in englischer Sprache verfügbar.

Die nachfolgende Abbildung zeigt den Aufbau der Code::Blocks Oberfläche.

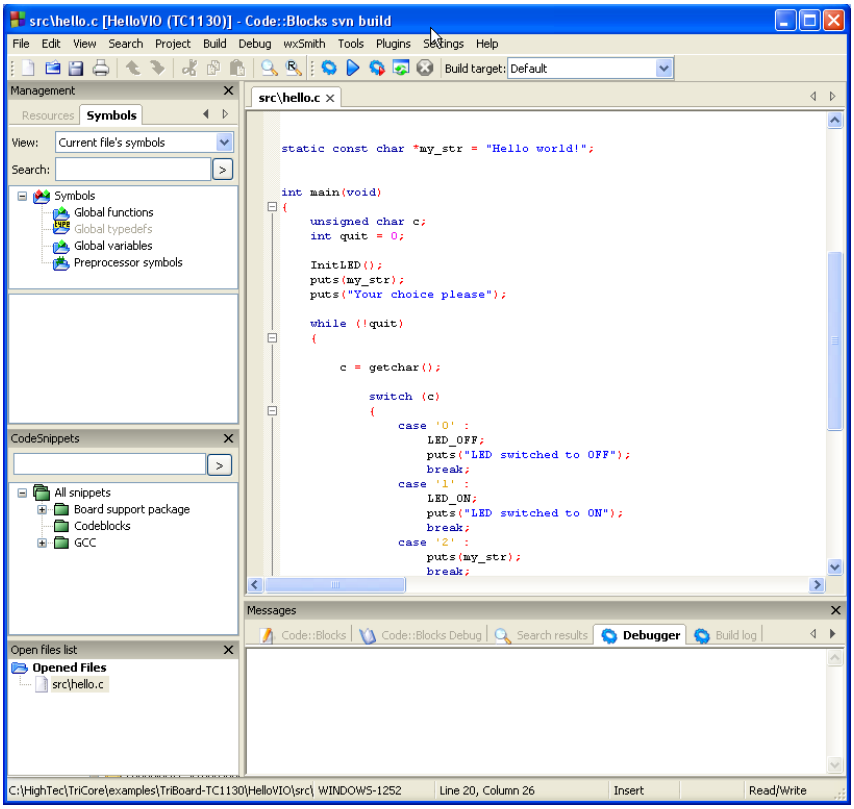

Abbildung 1.1: IDE Code::Blocks

Management Diese Fenster enthält die Ansicht 'Projects' , im nachfolgenden als Projectansicht bezeichnet. In dieser werden die in Code::Blocks aktuell geöffneten Projekte angezeigt. In dem Management Fenster erhält man im Reiter 'Symbols' die Anzeige von Symbolen, Variablen etc.

Editor In der obigen Abbildung ist eine Quelle `hello.c` mit Syntaxhighlighting im Editor geöffnet.

Open files list Zeigt die Liste der im Editor geöffneten Dateien an, hier `hello.c`.

CodeSnippets Lässt sich über das Menü 'View' →'CodeSnippets' anzeigen. Hier können Textbausteine und Verknüpfungen auf Dateien verwaltet werden.

Messages Fenster zur Ausgabe von Suchergebnisse, Logmeldung eines Compilers etc.

Code::Blocks bietet eine sehr flexible und umfassende Projektverwaltung. Der folgende Text geht nur auf einige Besonderheiten der Projektverwaltung ein.

1.1 Projektansicht

In Code::Blocks werden Quellen und die Einstellungen für den Builtprozess in einer Projektdatei <name>.cbp gespeichert. Ein Projekt besteht typischerweise aus C/C++ Quellen und zugehörige Header Dateien. Ein neues Projekt legen Sie am einfachsten an, indem Sie das Menü 'File' →'Project' ausführen und einen Wizard auswählen. Anschließend können Sie im Management Fenster über das Kontextmenü 'Add files' Dateien zum Projekt hinzufügen. In Code::Blocks werden die Projektdateien abhängig von ihrer Dateiendung in Kategorien verwalten. Die voreingestellen Kategorien sind für

Sources Unter der Kategorie `Sources` werden Quellen z.B. mit den Endungen `*.c;*.cpp;` aufgelistet.

Headers Unter der Kategorie `Headers` werden Dateien z.B. mit den Endungen `*.h;` angezeigt.

Resources Unter der Kategorie `Resources` werden z.B. Dateien `*.res;*.xrc;` für die Beschreibung von Layout von wxWidgets Fenster gelistet. Für die Anzeigen dieser Dateitypen dient im Manangement Fenster der Reiter 'Resources'.

Die Einstellungen für Typen und Kategorien von Dateien können über das Kontextmenü 'Project tree' →'Edit file types & categories' angepasst werden. Dabei können auch eigene Kategorien für Dateiendungen angelegt werden. Wenn Sie z.B. Linkerskripte mit der Endung `*.ld` unter der Kategorie `Linkerscript` anzeigen möchten, legen Sie einfach eine neue Kategorie an.

> **Hinweis:**
>
> Wenn Sie im Kontextmenü 'Project tree' →'Categorize by file types' deaktivieren, wird die Anzeige in Kategorien aufgehoben und die Dateien erscheinen wie sie im Dateisystem abgelegt sind.

1.2 Notizen für Projekte

In Code::Blocks können zu Projekten sogenannte Notes hinterlegt werden. Diese sollten eine Kurzbeschreibung oder Hinweise für das jeweilige Projekt enthalten. Durch Anzeige dieser Information beim Öffnen des Projektes bekommen andere Bearbeiter einen schnellen Überblick. Die Anzeige von Notes kann bei den Properties eines Projektes im Reiter Notes aktiviert bzw. deaktiviert werden.

1.3 Projektvorlagen

Code::Blocks wird mit einer Vielzahl von Projektvorlagen ausgeliefert, die beim Anlegen eines neuen Projektes angezeigt werden. Es ist aber auch möglich, eigene Vorlagen zu speichern und somit eigene Vorgaben für Compilerschalter, wie zu verwendete Optimierung, maschinenspezifische Schalter etc. in Vorlagen zusammenzufassen. Diese werden im Verzeichnis `Dokumente und Einstellungen\<user>\Anwendungsdaten\codeblocks\UserTemplates` abgelegt. Wenn die Vorlagen für alle Benutzer zugänglich sein sollen, müssen die Vorlagen

in zugehöriges Verzeichnis der Code::Blocks Installation kopiert werden. Diese Vorlagen erscheinen dann beim nächsten Start von Code::Blocks unter New Project User templates.

> **Hinweis:**
>
> Die verfügbaren Vorlagen im Project Wizard können durch Auswahl mit der rechten Maustaste bearbeitet werden.

1.4 Projekte aus Build Targets erstellen

In Projekten ist es notwendig unterschiedliche Varianten eines Projektes vorzuhalten. Varianten werden als Build Target bezeichnet. Diese unterscheiden sich in der Regel durch unterschiedliche Compileroptionen, Debug-Information und Auswahl von Dateien. Ein Build Target kann auch in ein eigenständiges Projekt ausgelagert werden, dafür selektieren Sie in 'Project' →'Properties' und dann den Reiter 'Build Targets' die Variante und wählen Sie Schaltfläche 'Create project from target' (siehe Abbildung 1.2 auf Seite 3).

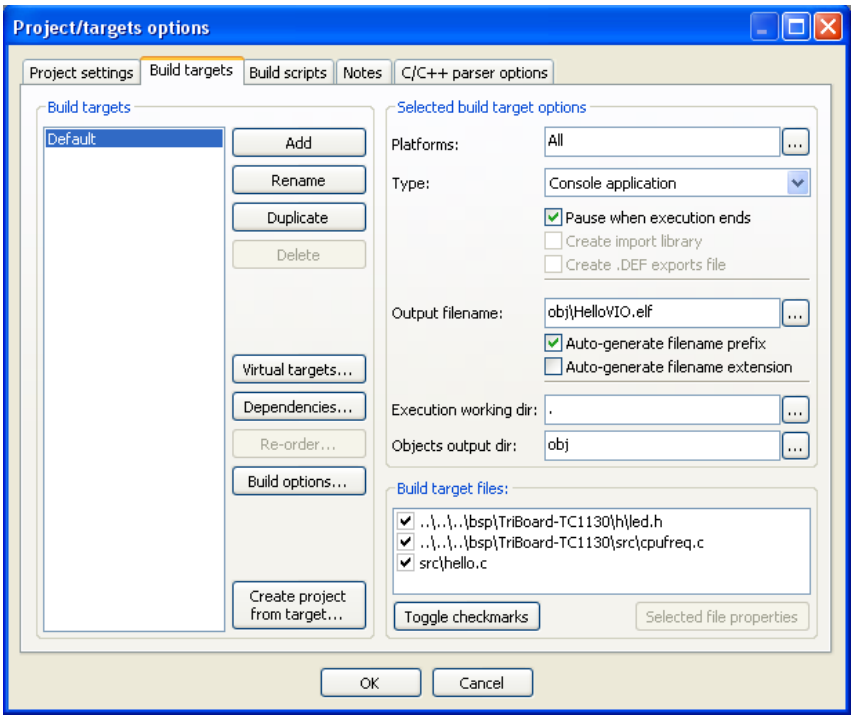

Abbildung 1.2: Build Targets

1.5 Virtual Targets

Mit sogenannten Virtual Targets können Projekte in Code::Blocks weiter strukturiert werden. Eine häufige Projektstruktur besteht aus zwei Build Targets. Einem Target 'Debug' mit Debuginformation und einem anderen Target 'Release' ohne diese Information. Durch Hinzufügen von Virtual Targets unter 'Project' →'Properties' →'Build Targets' können einzelne Build Targets zusammengefasst werden. So kann zum Beispiel ein Virtual Target 'All' die Targets Debug und Release gleichzeitig erzeugen. Die Virtual Targets werden auch in der Symbolleiste des Compilers unter Build Targets angezeigt.

1.6 Pre- und Postbuild Schritte

Code::Blocks ermöglicht es, weitere Arbeitschritte vor oder nach der Compilierung eines Projektes durchzuführen. Die Arbeitsschritte werden als Prebuilt bzw. Postbuilt Step bezeichnet. Typische Postbuilt Steps sind:

- Erzeugung eines Intel Hexformats aus einem fertigen Objekt
- Manipulation von Objekten mit `objcopy`
- Generierung von Dumpdateien mit `objdump`

Beispiel

Erzeugung einer Disassembly aus einem Objekt unter Windows. Die Umlenkung in eine Datei erfordert den Aufruf der `cmd` mit der Option `/c`.

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

Ein weiteres Beispiel für ein Postbuilt Step kann die Archivierung eines Projektes sein. Hierzu erstellen Sie ein Build Target 'Archive' und tragen im Postbuilt Step folgende Anweisung ein

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

Mit diesem Befehl werden das aktive Projekt und seine Quellen, Header und Objekte als Zip-Datei gepackt. Dabei werden über die Built-in Variablen $(PROJECT_NAME) und $(TODAY), der Projektname und das aktuelle Datum extrahiert (siehe Abschnitt 3.2 auf Seite 26). Im Verzeichnis des Projektes liegt dann nach Ausführen des Targets 'Archive' die gepackte Datei.

1.7 Hinzufügen von Scripts in Build Targets

Code::Blocks bieten die Möglichkeit, Aktionen die vom Benutzer in Menüs ausgeführt werden, auch in Skripten zu verwenden. Mit dem Skript entsteht somit ein zusätzlicher Freiheitsgrad um die Generierung Ihres Projektes zu steuern.

> **Hinweis:**
> Ein Skript kann auch bei einem Build Target angegeben werden.

1.8 Workspace und Project Dependencies

In Code::Blocks können Sie mehrere Projekte geöffnet halten. Durch speichern der geöffneten Projekte in einem sogenannten Workspace über 'File' →'Save workspace' können Sie die Projekte in einem Arbeitsbereich unter <`name`>`.workspace` zusammenfassen. Wenn Sie beim nächsten Start von Code::Blocks den Arbeitsbereich <`name`>`.workspace` öffnen erscheinen wieder alle Projekte. Komplexe Softwaresysteme bestehen aus Komponenten,

die in unterschiedlichen Code::Blocks Projekten verwaltet werden. Des weiteren existieren bei der Generierung von solchen Softwaresystemen oftmals Abhängigkeiten zwischen diesen Projekten.

Beispiel

Ein Projekt A enthält zentrale Funktionen die auch anderen Projekten in Form einer Bibliothek zugänglich gemacht werden. Wenn nun die Quellen der zentralen Funktionen geändert werden, muss die Bibliothek neu erzeugt werden. Damit die Konsistenz zwischen einem Projekt B, das die Funktionen verwendet und dem Projekt A, das die Funktionen implementiert, gewahrt bleibt, muss Projekt B von Projekt A abhängen. Die Information für die Abhängigkeit von Projekten wird im jeweiligen Workspace gespeichert, damit jedes Projekt weiterhin einzeln erzeugt werden kann. Durch die Verwendung von Abhängigkeiten kann auch die Reihenfolge bei der Generierung von Projekten gesteuert werden. Die Abhängigkeiten für Projekte werden über den Menüeintrag 'Project' →'Properties' und Auswahl der Schaltfläche 'Project's dependencies' gesetzt.

1.9 Einbinden von Assembler Dateien

In der Projektansicht (Project View) im Fenter Management werden Assembler Dateien im Ordner `Others` aufgeführt. Durch einen Rechtsklick einer der gelisteten Assembler Dateien erhält man ein Kontextmenü. Darin öffnet der Befehl 'Properties' ein neues Fenster. Klicken Sie darin auf den Reiter 'Build' und aktivieren Sie die beiden Felder 'Compile file' und 'Link file'. Wechseln Sie nun auf den Reiter 'Advanced' und führen Sie folgende Schritte durch:

1. 'Compiler variable' auf CC setzen
2. Den Compiler unter 'For this compiler' auswählen
3. 'Use custom command to build this file' anwählen
4. Inhalt im Fenster eingeben:

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

Dabei sind die Code::Blocks Variablen durch `$` gekennzeichnet (siehe Abschnitt 3.4 auf Seite 28). Diese werden automatisch ersetzt, so dass Sie lediglich die Assembleroption <asopt> durch Ihre Einstellungen ersetzen brauchen.

1.10 Editor und Hilfsmittel

1.10.1 Default Code

Durch vorgegebene Coding Rules im Unternehmen müssen Quelldateien einen einheitlichen Aufbau vorweisen. Code::Blocks bietet die Möglichkeit, beim Anlegen von neuen C/C++ Quellen und Header einen vorgegebenen Inhalt am Anfang einer Datei automatisiert einzufügen. Die vorgebene Inhalt wird als Default Code bezeichnet. Die Einstellung

hierfür kann unter 'Stettings' →'Editor' Default Code vorgenommen werden. Eine neue Datei erzeugen Sie über das Menü 'File' →'New' →'File' .

Beispiel

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

1.10.2 Abbreviation

Durch Definition von Abkürzung in Code::Blocks kann einiges an Schreibarbeit und Zeit gespart werden. Hierzu werden in 'Settings' →'Editor' sogenannte Abbreviations unter dem Namen <name> angelegt, die über das Tastenkürzel Ctrl+J aufgerufen werden. Durch Einfügen von Variablen `$(NAME)` in den Abkürzungen ist auch eine Parametrisierung möglich.

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

Bei Aufruf der Abkürzung <name> im Quelltext und Ausführen von Ctrl+J, wird der Inhalt der Variablen abgefragt und eingefügt.

1.10.3 Konfigurationsdateien

Die Einstellungen für Code::Blocks werden im Profil `default.conf` im Ordner `codeblocks` in Ihren Anwendungsdaten gespeichert. Bei Verwendung von personalities (siehe Unterabschnitt 1.10.4 auf Seite 7 werden die Konfiguration in der Datei `<personality>.conf` abgelegt.

Mit dem Werkzeug `cb_share_conf`, aus dem Code::Blocks Installationsverzeichnis, können diese Einstellungen verwaltet und gesichert werden.

Falls Sie Standardeinstellung für mehrere Benutzer eines PCs vorgeben möchten, muss die Konfigurationsdatei `default.conf` im Ordner `\Dokumente und Einstellungen\Default User\Anwendungsdaten\codeblocks` abgelegt sein. Beim ersten Start von Code::Blocks werden die Voreinstellungen aus 'Default User' in die Anwendungsdaten der aktuellen Benutzers kopiert.

Zur Erzeugung einer portablen Version von Code::Blocks auf einem USB-Stick gehen Sie wie folgt vor. Kopieren Sie die Code::Blocks Installation auf einen USB-Stick und legen Sie die Konfigurationsdatei `default.conf` in dieses Verzeichnis. Die Konfiguration wird als globale Einstellung verwendet. Bitte achten Sie darauf, dass die Datei schreibbar sein muss, damit Änderungen in der Konfiguration auch gespeichert werden können.

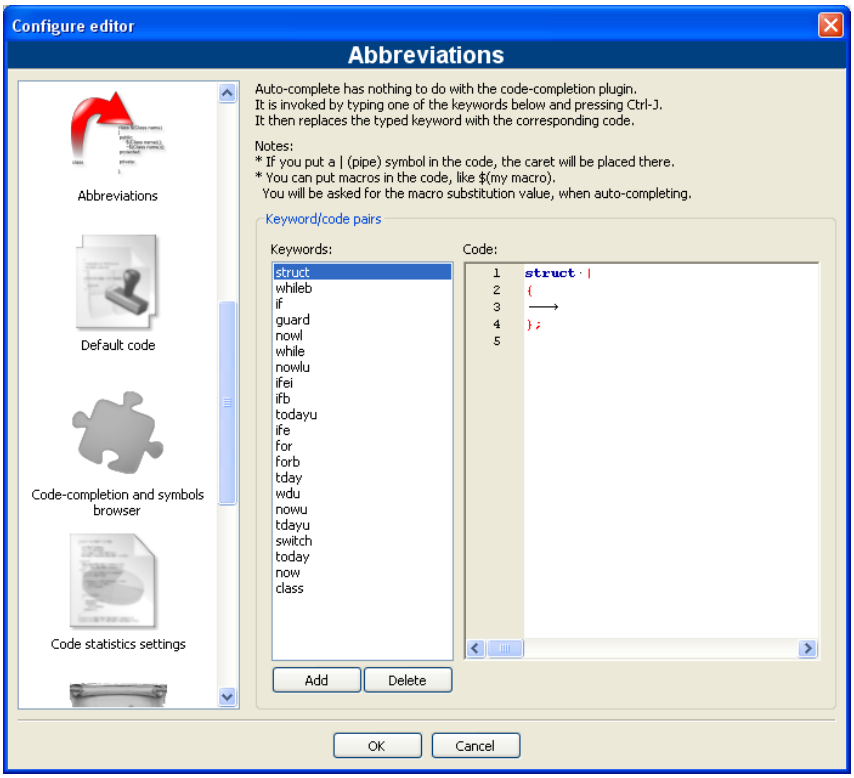

Abbildung 1.3: Definition von Abkürzungen

1.10.4 Personalities

Code::Blocks Einstellungen werden als Anwendungsdaten im Verzeichnis `codeblocks` in einer Datei `<user>.conf` gespeichert. Diese Konfigurationsdatei enthält Informationen wie beispielsweise zuletzt geöffnete Projekte, Einstellungen für Editor, Anzeige von Symbolleisten etc. Standardmäßig ist die Personality 'default' eingestellt, so dass die Konfiguration in der Datei `default.conf` abgelegt ist. Wenn Code::Blocks mit dem Parameter `--personality=myuser` in der Kommandozeile aufgerufen wird, werden die Einstellungen in der Datei `myuser.conf` gespeichert. Falls das Profil nicht bereits existiert, wird es automatisch angelegt. Durch diese Vorgehensweise können für unterschiedliche durchzuführende Arbeitsschritte auch zugehörige Profile gespeichert werden. Wenn Sie Code::Blocks mit dem zusätzlichen Parameter `--personality=ask` starten erscheint ein Auswahldialog für die verfügbaren Profile.

1.10.5 Navigieren und Suchen

In Code::Blocks existieren unterschiedliche Möglichkeiten zum schnellen Navigieren zwischen Dateien und Funktionen. Eine typische Vorgehensweise ist das Setzen von Lesezeichen (Bookmarks). Durch Betätigen des Tastenkürzel (Ctrl+B) wird ein Lesezeichen in einer Quelldatei gesetzt bzw. gelöscht. Mit (Alt+PgUp) wird zum vorherigen Lesezeichen gesprungen und mit (Alt+PgDn) zum nächsten gewechselt.

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.

Eine weitere komfortable Funktion bietet das Tastenkürzel (Ctrl+Alt+G). Der sich öffnende Dialog erlaubt die Auswahl von Funktionen und springt anschließend an die Implementierung der Funktion (siehe Abbildung 1.4 auf Seite 8).

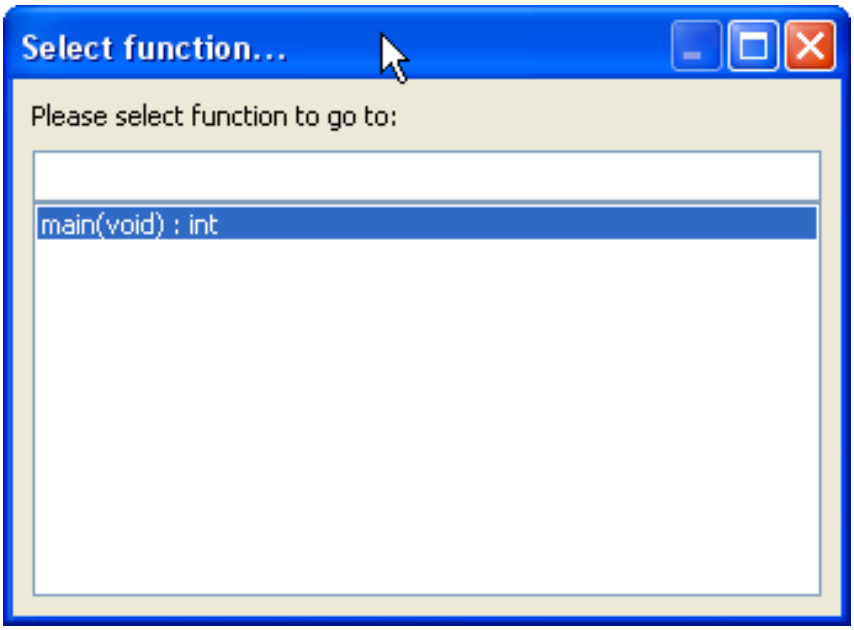

Abbildung 1.4: Suche nach Funktionen

Hinweis:

Mit dem Tastenkürzel Ctrl+PAGEUP springen Sie an die vorherige Funktion und mit Ctrl+PAGEDOWN zur nächsten Funktion.

Wenn Sie sich im Editor Fenster befinden, können Sie mit Ctrl+Tab zwischen den Reiter von geöffneten Dateien springen. Durch setzen der Einstellung 'Use Smart Tab-switching scheme' in 'Settings' →'Notebook appearance' erhalten Sie nun über Ctrl+Tab ein zusätzliches Open Tabs Fenster im Editor. Dabei wird die Liste in der Reihenfolge der geöffneten Dateien gezeigt.

Eine häufige Arbeitsweise bei der Entwicklung von Software ist jedoch, dass man sich durch ein Satz von Funktion hangelt, die in unterschiedlichen Dateien implementiert sind. Durch das Plugin BrowseTracks zeigt die Open Tabs eine Liste in der Reihenfolge wie Dateien selektiert wurden. Somit können Sie komfortabel zwischen den Aufrufen navigieren.

In Code::Blocks aktivieren Sie die Anzeige von Zeilennummern in 'Settings' →'General Settings' im Feld 'Show line numbers'. Mit dem Tastenkürzel Ctrl+G oder über das Menü 'Search' →'Goto line' springen Sie an die gewünschte Zeile.

1.10.6 Symbolansicht

Für das Navigieren über Funktionen oder Variablen bietet das Management Fenster in Code::Blocks eine Baumansicht für Symbole von C/C++ Quellen. Dabei lässt sicht der Gültigkeitsbereich (Scope) der Ansicht auf die aktuelle Datei oder Projekt oder den gesamten Arbeitsbereich einstellen. Für die Kategorien der Symbole existieren folgende Kategorien.

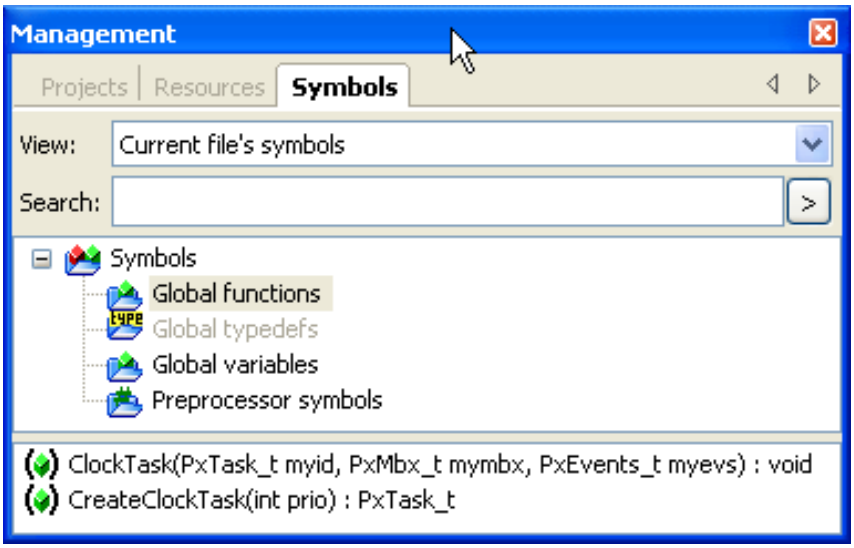

Abbildung 1.5: Symbolansicht

Global functions Listet die Implementierung von globalen Funktionen.

Global typedefs Listet die Verwendung von **typedef** Definitionen.

Global variables Zeigt die Symbole von globalen Variablen an.

Preprocessor symbols Auflistung der mit **#define** erzeugten Präprozessor Direktiven.

Strukturen und Klassen werden unterhalb von Preprocessor symbols angezeigt. Wenn eine Kategorie mit der Maus angewählt wird, erscheinen die gefundenen Symbole in dem unteren Teil des Fensters (siehe Abbildung 1.5 auf Seite 9). Ein Doppelklick auf das Symbol öffnet die Datei, wo das Symbol definiert bzw. die Funktion implementiert ist und springt an die zugehörige Zeile.

> **Hinweis:**
>
> Im Editor können Sie über das Kontextmenü 'Insert Class method declaration implementation' bzw. 'All class methods without implementation' sich auch die Liste der Klassen anzeigen lassen.

1.10.7 Einbinden von externen Hilfen

Die Entwicklungsumgebung Code::Blocks unterstützt das Einbinden von externen Hilfen über das Menü 'Settings' →'Environment' . Fügen Sie ein Manual Ihrer Wahl im chm Format in 'Help Files' hinzu und wählen Sie die Einstellung 'this is the default help

file'. Nun können Sie in Code::Blocks in einer geöffneten Quelldatei eine Funktion mit der Maus durch Doppelklick markieren und anschließend die Hilfe mit F1 aufrufen und erhalten die zugehörige Dokumentation.

Wenn Sie mehrere Hilfedateien einbinden, können Sie im Editor einen Begriff markieren und anschließend über das Kontextmenü 'Locate in' die Hilfedatei auswählen, in der Code::Blocks suchen soll.

1.10.8 Einbinden von externen Werkzeugen

Die Einbindung von externen Tools ist in Code::Blocks unter dem Menüeintrag 'Tools' →'Configure Tools' →'Add' vorgesehen. Für die Übergabeparameter der Tools kann auch auf Built-in Variables (see Abschnitt 3.2 auf Seite 26) zugegriffen werden. Des weiteren existieren für das Starten von externen Anwendungen unterschiedliche Arten (Launching options). Je nach Option werden die extern gestarteten Anwendung beim Beenden von Code::Blocks gestoppt. Falls die Anwendungen auch beim Beenden von Code::Blocks geöffnet bleiben sollen, ist die Option 'Launch tool visible detached' einzustellen.

1.11 Tips zum Arbeiten mit Code::Blocks

In diesem Kapitel werden Ihnen einige nützliche Einstellungen in Code::Blocks vorgestellt.

1.11.1 Konfiguration von Umgebungsvariablen

Die Konfiguration für ein Betriebssystem wird durch sogenannte Umgebungsvariablen festgelegt. Zum Beispiel enthält die Umgebungsvariablen PATH den Pfad auf einen installierten Compiler. Das Betriebssystem geht diese Umgebungsvariable von vorne nach hinten durch, d.h. die Einträge am Ende werden als letztes durchsucht. Wenn nun unterschiedliche Versionen eines Compilers oder anderer Anwendungen installiert sind, können nun folgende Situationen auftreten:

- Die falsche Version einer Software wird aufgrufen
- Installierte Softwarepakete stören sich gegenseitig

Es könnte zum Beispiel notwendig sein, dass für unterschiedliche Projekte unterschiedliche Versionen eines Compilers oder anderer Werkzeugen vorgeschrieben sind. Eine Möglichkeit ist die Umgebungsvariablen in der Systemsteuerung jeweils für ein Projekt zu ändern. Diese Vorgehensweise ist jedoch fehleranfällig und nicht flexibel. Für diese Anforderung bietet Code::Blocks eine elegante Lösung. Es lassen sich hier unterschiedliche Konfigurationen von Umgebungsvariablen erstellen, die nur intern in Code::Blocks verwendet werden. Zusätzlich kann zwischen diesen Konfiguration umgeschaltet werden. Die Abbildung 1.6 auf Seite 11 zeigt den Eingabedialog, den Sie über das Menü 'Settings' →'Environment' und Auswahl von 'Environment Varibales' erhalten. Eine Konfiguration wird über die Schaltfläche 'Create' erzeugt. Die Übernahme der hinzugefügten Umgebungsvariablen erfolgt durch Bestätigen des OK Knopfes. Das Aktivieren einer Konfiguration erfolgt über den Knopf Set Now.

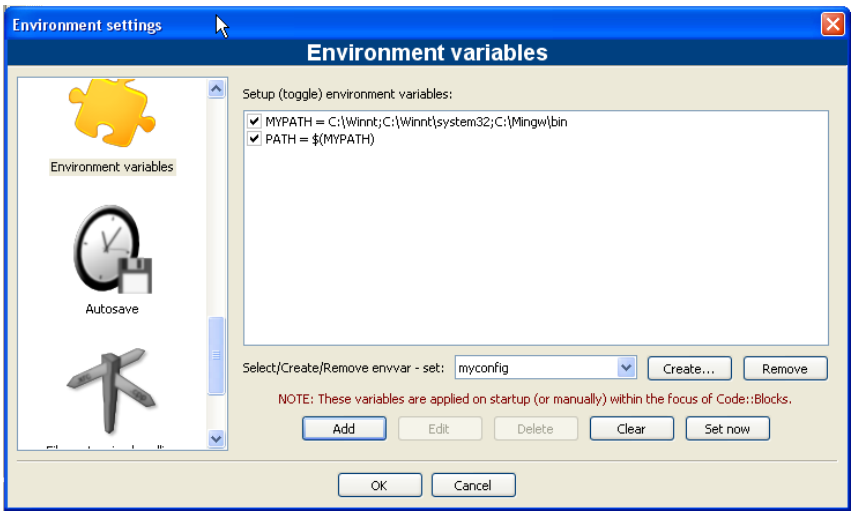

Abbildung 1.6: Umgebungsvariablen

Der Zugriff und der Gültigkeitkeitbereich auf die hier erstellten Umgebungsvariablen ist auf Code::Blocks begrenzt. Sie können diese Umgebungsvariablen wie auch andere Code::Blocks Variablen über `$(NAME)` expandieren.

> **Hinweis:**
>
> Eine Konfiguration von Umgebungsvariable lässt sich pro Projekt im Kontextmenü 'Properties' im Reiter 'EnvVars options' selektieren.

Beispiel

Sie können die verwendete Umgebung in einem postbuild Step (siehe Abschnitt 1.6 auf Seite 4) in einer Datei `<project>.env` schreiben und zu Ihrem Projekt archivieren.

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

oder unter Linux

`echo $PATH > project.env`

1.11.2 Umschalten zwischen Projekten

Wenn mehrere Projekte oder Dateien gleichzeitig geöffnet sind, so will der Benutzer häufig zwischen den Projekten und Dateien schnell wechseln können. Code::Blocks stellt hierfür eine Reihe an Shortcuts zur Verfügung.

Alt-F5 Aktiviert vorheriges Projekt aus der Projektansicht.

Alt-F6 Aktiviert nachfolgendes Projekt aus der Projektansicht.

F11 Wechselt im Editor zwischen einer Quelldatei `<name>.cpp` und der zugehörigen Header Datei `<name>.h`

1.11.3 Erweitere Einstellung für Compiler

Beim Buildprozess eines Projektes werden die Ausgaben des Compilers in Fenster Messages im Reiter Build Log ausgegeben. Wenn Sie an detaillierten Information interessiert sind, kann die Ausgabe erweitert werden. Dazu wählen Sie unter 'Settings' →'Compiler and Debugger' im Reiter 'Other Settings'.

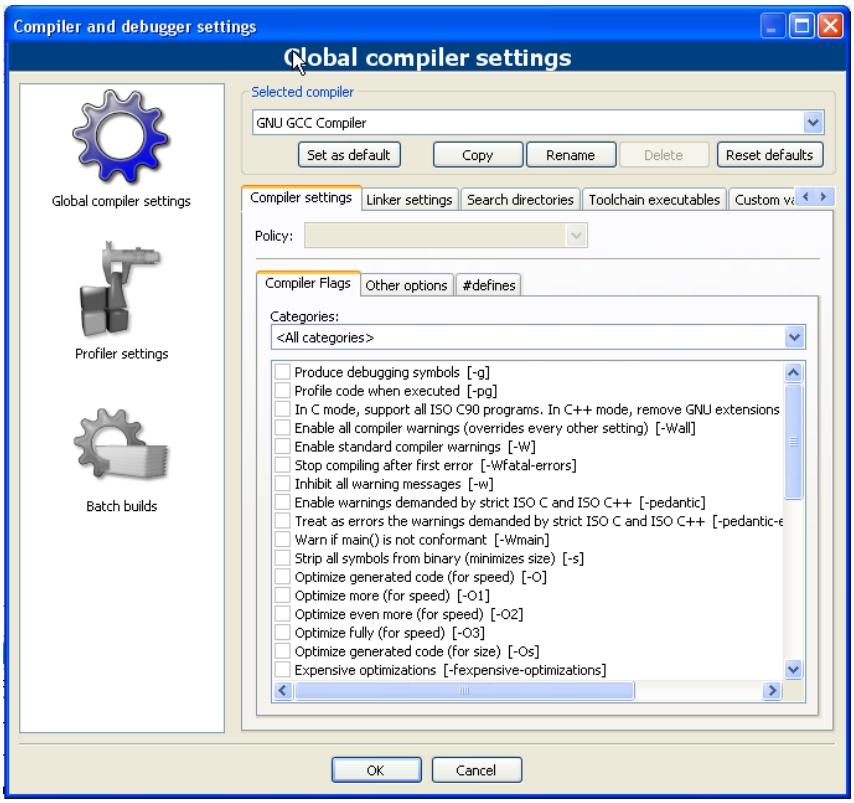

Abbildung 1.7: Einstellung von Detailinformationen

Achten Sie darauf, dass beim Eintrag Selected Compiler der gewünschte Compiler eingestellt ist. Die Einstellung 'Full command line' im Feld Compiler Logging gibt die vollständige Information im Build Log aus. Zusätzlich kann diese Ausgabe in eine HTML-Datei geloggt werden. Hierzu ist die Einstellung 'Save build log to HTML file when finished' erforderlich. Des weiteren bietet Code::Blocks eine Fortschrittsanzeige des Buildprozesses im Fenster Build Log. Diese aktivieren Sie mit dem Einstellung 'Display build progress bar'.

1.11.4 Zoom im Editor

Code::Blocks bietet einen sehr leistungsfähigen Editor. Eine Besonderheit ist, dass Sie innerhalb einer geöffneten Datei die Darstellung vergrößern und verkleinern können. Wenn Sie eine Maus mit einem Scrollrad haben, halten Sie einfach die Ctrl-Taste gedrückt und scrollen im Editor über das Rad nach vorne oder hinten.

1.11.5 Einbinden von Bibliotheken

In den Builtoption eines Projektes können Sie unter 'Linker Settings' im Eintrag 'Link libraries' über die Schaltfläche 'Add' verwendete Bibliotheken hinzufügen. Dabei können Sie entweder den absoluten Pfad zur Bibliothek durchsuchen oder nur den Namen ohne den Prefix `lib` und die Dateiendung angeben.

Beispiel

Für eine Bibliothek <`path`>\`libs`\`lib`<`name`>`.a` geben Sie einfach <`name`> an. Der Linker mit den jeweiligen Suchpfaden für die Bibliotheken bindet diese dann korrekt ein.

1.11.6 Linkreihenfolge von Objekten

Beim Compilierung werden aus Quellen `name.c/cpp` werden Objekte `name.o` erzeugt. Der Linker bindet die einzelnen Objekten zu einer Anwendung `name.exe` oder für den Embedded Bereiche `name.elf`. In einigen Fällen ist es wünschenswert die Reihenfolge für das Binden von Objekten vorzugeben. In Code::Blocks kann dies durch die Vergabe von sogenannten Prioritäten erzielt werden. Stellen Sie für eine Datei über das Kontextmenü 'Properties' im Reiter Build die Priorität ein. Dabei führt eine geringe Priorität des Objekts dazu, dass es zu erst gebunden wird.

1.11.7 Autosave

Code::Blocks bietet die Möglichkeit Projekte und Quelldateien automatisch zu speichern bzw. eine Sicherungskopie anzulegen. Diese Funktionalität wird im Menü 'Settings' →'Environment' →'Autosave' eingestellt. Dabei sollte als 'Save to .save file' als Methode für das Erstellen einer Sicherungskopie eingestellt werden.

1.11.8 Einstellen für Dateizuordnungen

In Code::Blocks können Sie zwischen verschiedenen Arten der Behandlung von Dateiendungen wählen. Die Einstellungen erhalten Sie über 'Settings' →'Files extension handling' . Sie können entweder die von Windows zugeordneten Anwendungen (open it with the associated application) für entsprechende Dateiendungen verwenden oder für jede Dateiendungen die Einstellungen so ändern, dass entweder ein benutzerdefiniertes Programm (launch an external program) gestartet wird oder die Datei in Editor von Code::Blocks geöffnet wird (open it inside Code::Blocks editor).

> **Hinweis:**
>
> Wenn ein benutzerdefiniertes Programm für eine Dateiendung gewählt wird, sollte die Einstellung 'Disable Code::Blocks while the external programm is running' deaktiviert werden, da sonst beim Öffnen dieser Dateien Code::Blocks beendet wird.

1.12 Code::Blocks in der Kommandozeile

Die IDE Code::Blocks kann auch ohne grafische Oberfläche in der Kommandozeile ausgeführt werden. Dabei stehen unterschiedliche Schalter zur Verfügung um den Buildprozess

eines Projektes zu steuern. Da Code::Blocks somit skriptfähig ist, kann die Erzeugung von Exectutables in eigene Arbeitsabläufe integriert werden.

```
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

Auch wenn man eine IDE wie Code::Blocks überwiegend mit der Maus bedient, erweisen sich dennoch Tastenkombinationen immer wieder als hilfreich, um die Arbeit zu vereinfachen und zu beschleunigen. In nachstehender Tabelle sind einige verfügbare Tastenkombinationen zusammengefasst.

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.

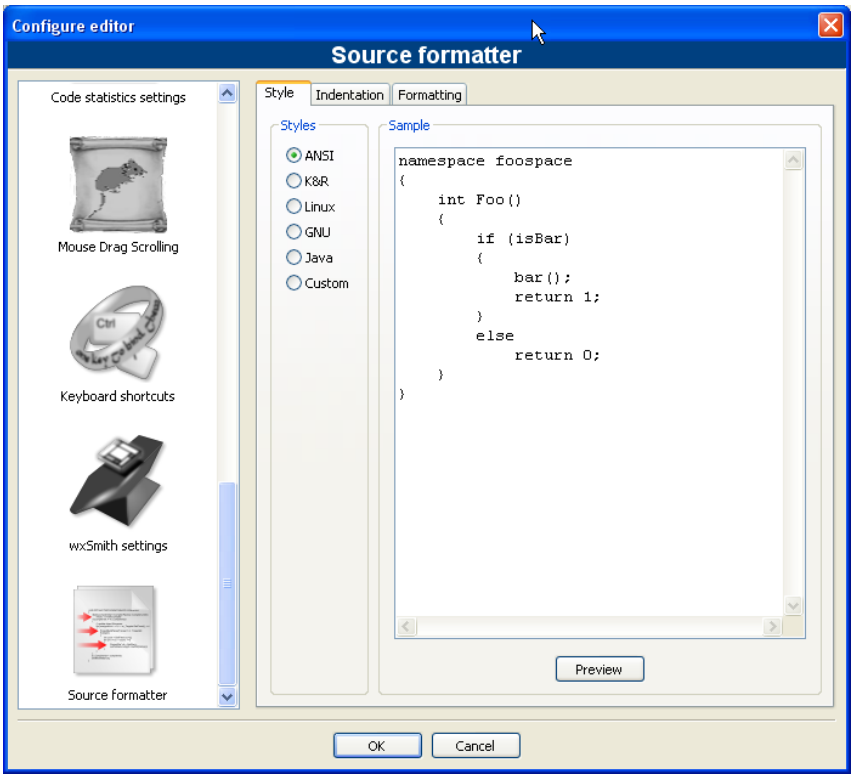

Abbildung 2.1: Formatierung für Quellcode

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.

Hinweis:

Durch Kopieren von Code z.B aus dem Internet oder aus einem Manual wird in Code::Blocks der Code automatisch an die ausgewählten Coding-Rules angepasst, indem Sie den Text markieren und das Plugin über das Menü 'Plugins' →'Source code formatter' ausführen.

2.2 CodeSnippets

Das Plugin CodeSnippets ermöglicht es Textbausteine und Verknüpfungen auf Dateien in einer Baumansicht nach Kategorien zu strukturieren. Die Bausteine dienen dazu, häufig verwendete Dateien oder Konstrukte in Textbausteine abzulegen und zentral zu verwalten. Stellen Sie sich vor eine Reihe von häufig verwendeten Quelldateien sind im Dateiesystem in unterschiedlichen Ordnern abgelegt. Im Fenster CodeSnippets können Sie nun Kategorien und darunter Verknüpfungen auf die gewünschten Dateien erstellen. Damit können Sie den Zugriff auf die Dateien unabhängig von der Ablage im Dateisystem verwalten und ohne das Dateisystem zu durchsuchen schnell zwischen diesen Dateien navigieren.

Die Liste der Textbausteine und Verknüpfungen können im CodeSnippets Fenster mit der rechten Maustaste über das Kontextmenü 'Save Index' gespeichert werden. Die dabei erzeugte Datei `codesnippets.xml` befindet sich anschließend in Ihren `Dokumente und Einstellungen\Anwendungsdaten` im Ordner `codeblocks`. Unter Linux wird diese Information im HOME-Verzeichnis im Ordner `.codeblocks` abgelegt. Die Konfigurationsdateien von Code::Blocks werden beim nächsten Start geladen. Falls Sie den Inhalt von CodeSnippets an einen anderen Ort speichern möchten, selektieren Sie den Eintrag 'Save Index As'. Zum Laden dieser Datei wählen Sie beim nächsten Start von Code::Blocks'Load Index File' oder stellen das Verzeichnis in dem Kontextmenü 'Settings' unter 'Snippet Folder' ein. Diese Einstellungen werden in der zugehörigen Datei `codesnippets.ini` in den Anwendungsdaten hinterlegt.

Das Einfügen einer Kategorie geschieht über das Menü 'Add SubCategory'. In einer Kategorie können Snippets (Textbausteine) oder File Links (Verknüpfungen) liegen. Ein Textbaustein wird mit dem Kontextmenü über 'Add Snippet' angelegt. Indem Sie einen Text im Code::Blocks Editor markieren und anschließend bei gedrückter linker Maustaste per Drag and Drop auf den Textbaustein ziehen, wird der Inhalt in den Textbaustein eingefügt. Durch einen Doppelklick auf den neu eingefügten Eintrag oder durch Auswahl von 'Edit Text' öffnet sich ein eigenständiger Editor zum Bearbeiten des Inhaltes.

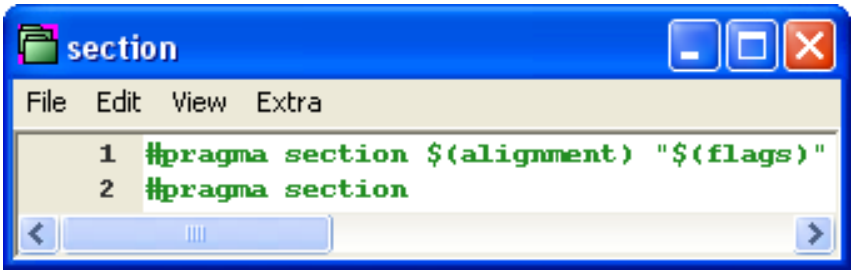

Abbildung 2.2: Bearbeiten eines Textbausteins

Die Ausgabe eines Textbausteines in Code::Blocks erfolgt über das Kontextmenü 'Apply'

oder durch Drag und Drop in den Editor. Die Inhalte eines Snippets können auch in andere Anwendungen gezogen werden.

Textbausteine sind darüberhinaus auch über Variablen <name>, die über $(name) zugegriffen werden, parametrisierbar (siehe Abbildung 2.2 auf Seite 18). Die Abfrage für die Werte der Variablen erfolgt über ein Eingabefeld, wenn der Textbaustein mit dem Kontextmenü 'Apply' aufgerufen wird.

Neben den Textbausteinen können auch Verknüpfungen auf Dateien angelegt werden. Wenn Sie zuvor einen Textbaustein angelegt haben und anschließend das Kontextmenü 'Properties' auswählen, selektieren Sie mit der Schaltfläche 'Link target' das Ziel der Verknüpfung. Eine Verknüpfung kann auch über das Kontextmenü 'Convert to FileLink' erzeugt werden. Dieser Schritt wandelt den Textbaustein automatisch in eine Verknüpfung auf eine Datei um. In CodeSnippets werden Textbausteine mit einem T-Symbol und Verknüpfungen auf eine Datei mit einen F-Symbol gekennzeichnet. Falls Sie diese Datei statt mit einem externen Editor lieber mit dem integrierten Code::Blocks Editor bearbeiten wollen, so ziehen Sie einfach den Eintrag für eine Verknüpfung von der CodeSnippets Ansicht in das Fenster 'Open files list'. Nach dem Bearbeiten und Speichern der Datei ziehen Sie den zugehörigen Eintrag von der 'Open files list' wieder an die Stelle wo der Eintrag der Verknüpfung innerhalb der CodeSnippets Ansicht liegt.

Hinweis:

Wenn Sie für die Dateizuordnung (File extension handling) die Einstellung 'open it with the associated application' unter 'Settings' →'Environment' vorgenommen haben, wir die von Windows zugeordnete Anwendungen für Dateiendungen verwendet (siehe Unterabschnitt 1.11.8 auf Seite 13).

Falls Sie diese Einstellung vorgenommen haben, dann wird wenn Sie z.B. einen Verknüpfung auf eine pdf-Datei aus der Codesnippets Ansicht in das Fenster 'Open files list' ziehen automatisch ein pdf-Viewer estartet. Dieses Vorgehen ermöglicht dem Benutzer Dateien, die über das Netzwerk verteilt liegen, wie z.B. Schaltpläne, Dokumentation etc. als Verknüpfung einfach über die gewohnten Anwendungen zuzugreifen. Der Inhalt der Codesnippets wird in der Datei `codesnippets.xml` und die Konfiguration in der Datei `codesnippets.ini` in Ihren Anwendungsdaten gespeichert. In dieser ini Datei wird z.B. der Ablageort der Datei `codesnippets.xml` hinterlegt.

Code::Blocks unterstützt die Verwendung von unterschiedlichen Profilen. Diese werden als personalities bezeichnet. Das Anlegen eines Profils geschieht indem Code::Blocks mit der Kommandozeilen Option `--personality=<profile>` gestartet wird. Die Einstellungen werden dann statt in `default.conf` in der Datei `<personality>.conf` in den Anwendungsdaten gespeichert. Das Plugin Codesnippets speichert seine Einstellungen dann in der Datei `<personality>.codesnippets.ini`. Wenn nun Sie in den Settings von Codesnippets über 'Load Index File' einen neuen Inhalt `<name.xml>` laden, wird dies in der zugehörigen ini Datei hinterlegt. Der Vorteil von dieser Vorgehensweise ist, dass Sie zu unterschiedlichen Profilen auch unterschiedliche Konfigurationen für Textbausteine und Verknüpfungen verwalten können.

Für das Navigieren zwischen den Kategorien und Snippets bietet das Plugin eine zusätzliche Suchfunktion. Hierbei lässt sich auch der Gültigkeitsbereich (Scope) für die Suche auf Snippets, Categories oder Snippets and Categories einstellen. Durch Eingabe des gewünschten Suchbegriffes wird automatisch der zugehörige Eintrag in der Ansicht ausgewählt. Die Abbildung 2.3 auf Seite 20 zeigt eine typische Ansicht im CodeSnippets Fenster.

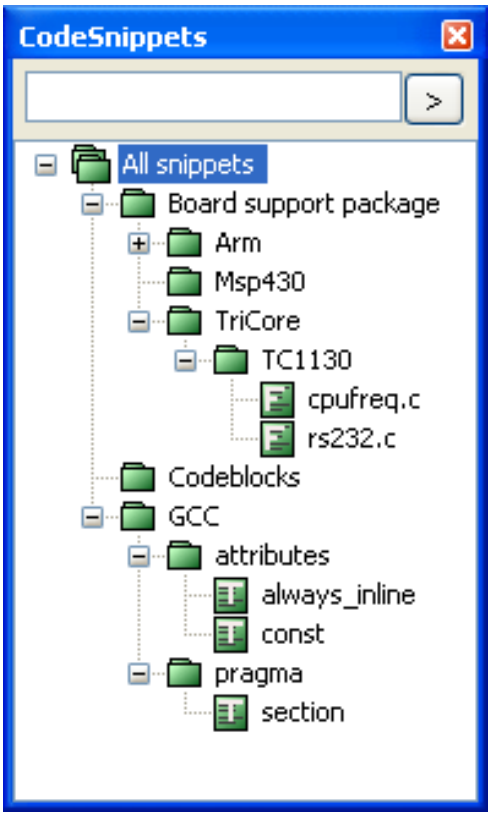

Abbildung 2.3: Ansicht von CodeSnippets

Hinweis:

Bei Verwendung von umfangreichen Textbausteine sollte deren Inhalt über 'Convert to File Link' in Dateien ausgelagert werden, um die Speicherauslastung im System zu reduzieren.

2.3 ToDo List

Für komplexe Software-Projekte, an denen unterschiedliche Benutzer arbeiten, hat man häufig die Anforderung, dass zu erledigende Arbeiten von unterschiedlichen Usern umzusetzen sind. Für dieses Problem bietet Code::Blocks eine Todo List. Diese Liste, zu öffnen unter 'View' →'ToDo list' , enthält die zu erledigenden Aufgaben mit Prioritäten, Typ und zuständige User. Dabei kann die Ansicht nach zu erledigenden Aufgaben nach Benutzer und/oder Quelldatei gefiltert werden.

Ein Todo lässt sich bei geöffneten Quellen in Code::Blocks über die rechte Maustaste

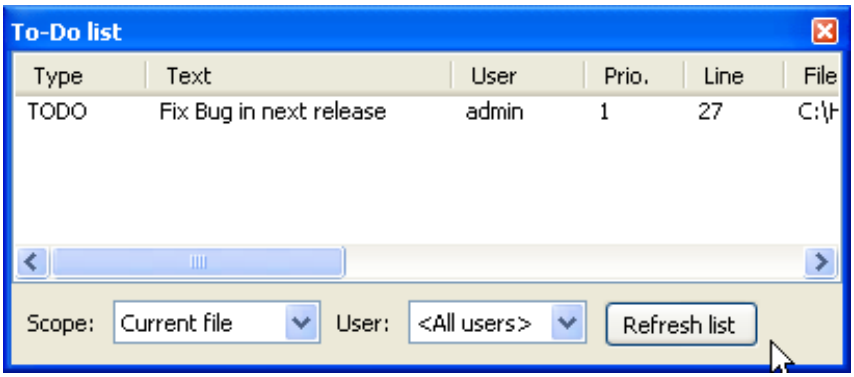

Abbildung 2.4: Anzeige der ToDo List

'Add To-Do item' hinzufügen. Im Quellcode wird ein entsprechender Kommentar an der ausgewählten Quellzeile eingefügt.

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

Beim Hinzufügen eines To-Do erhalten Sie einen Eingabedialog mit folgenden Einstellungen (siehe Abbildung 2.5 auf Seite 21).

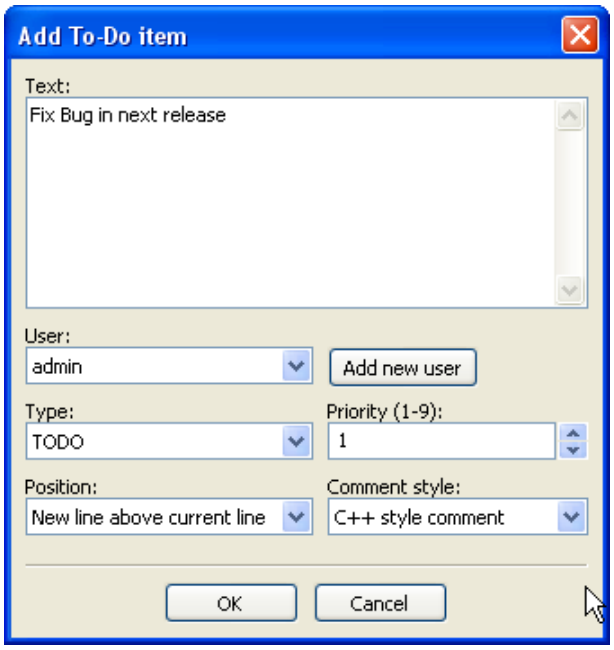

Abbildung 2.5: Dialog für Eingabe von ToDo

User Username <user> im Betriebssystem. Hierbei können auch Aufgaben für andere Benutzer angelegt werden. Dabei muss der zugehörige Benutzername über Add new user hinzugefügt werden. Die Zuordnung eines Todo geschieht dann über die Auswahl der unter User aufgelisteten Einträge.

> **Hinweis:**
>
> Beachten Sie, dass die User nichts mit den in Code::Blocks verwendeten Personalities zu tun haben.

Type Standardmäßig ist der Typ auf Todo eingestellt.

Priority Die Wichtigkeit von Aufgaben können in Code::Blocks durch Prioritäten (Wertebereich: 1 - 9) gewichtet werden.

Position Einstellung ob der Kommentar vor, nach oder exakt an der Stelle des aktuell befindlichen Cursor eingefügt werden soll.

Comment Style Auswahl der Formatierung für Kommentare (zum Beispiel doxygen).

2.4 Source Code Exporter

Oft ergibt sich die Notwendigkeit, den Quelltext in andere Anwendungen oder in Emails zu übernehmen. Beim schlichten Kopieren des Textes geht jedoch die Formatierung verloren, was den Text sehr unübersichtlich macht. Die Export Funktion in Code::Blocks schafft hier Abhilfe. Über 'File' →'Export' kann ein gewünschtes Dateiformat für die Exportdatei ausgewählt werden. Danach übernimmt das Programm den Dateinamen und das Zielverzeichnis der geöffneten Quelldatei und schlägt diesen als Name zum speichern vor. Die jeweilige Dateiendung wird durch das Exportformat bestimmt. Es stehen folgende Formate zur Verfügung.

html Ein textbasiertes Format, das in einem Web-Browser oder Anwendungen zur Textverarbeitung angezeigt werden kann.

rtf Das Rich Text Format ist ein textbasiertes Format, das sich in Programmen zur Textverarbeitung wie Word oder OpenOffice öffnen lässt.

odt Open Document Text Format ist ein standardisiertes Format, dass von Sun und O'Reilly festgelegt wurde. Dieses Format kann von Word, OpenOffice und anderen Textverarbeitungsprogrammen eingelesen werden.

pdf Das Portable Document Format kann mit Anwendungen wie Acrobat Reader geöffnet werden.

2.5 Thread Search

Über das Menu 'Search' →'Thread Search' lässt sich das entsprechende Plugin als Tab in der Messages Console ein- und ausblenden. In Code::Blocks kann mit diesem Plugin eine Vorschau für das Auftreten einer Zeichenkette in einer Datei, Workspace oder Verzeichnis angezeigt werden. Dabei wird die Liste der Suchergebnisse in der rechten Seite der ThreadSearch Console angezeigt. Durch Anklicken eines Eintrages in der Liste wird auf der linken Seite eine Vorschau angezeigt. Durch einen Doppelklick in der Liste wird die ausgewählte Datei im Code::Blocks Editor geöffnet.

> **Hinweis:**
>
> Beachten Sie, dass die Einstellung von zu durchsuchenden Dateiendungen voreingestellt ist und eventuell angepasst werden muss.

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 Abschnitt 3.2 auf Seite 26 for details).

Command Macros . This type of variables is used for controlling the build process. For further information please refer to Abschnitt 3.4 auf Seite 28.

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

Abschnitt 3.7 auf Seite 29 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.

Hinweis:

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.).

> **Hinweis:**
>
> 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 Unterabschnitt 3.12.1 auf Seite 32 and Abbildung 3.1 auf Seite 30 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.

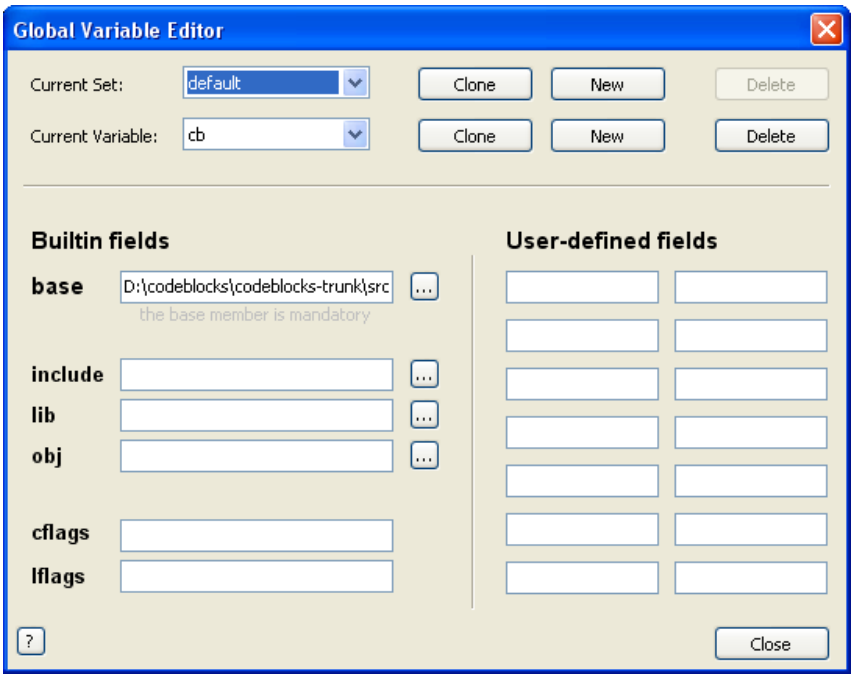

Abbildung 3.1: Global Variable Environment

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

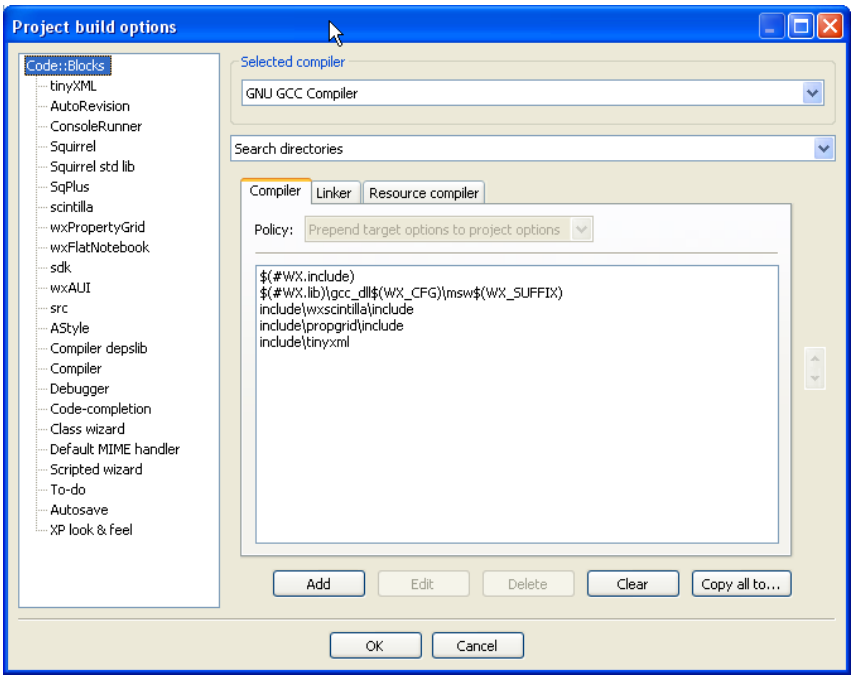

Abbildung 3.2: Global Variables

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.