HP InfoTech

# CodeVisionAVR

VERSION 3.03

# User Manual

CodeVisionAVR V3.03 User Manual
Revision 38/03.2013

## Table of Contents



# 1. Introduction

CodeVisionAVR is a C cross-compiler, Integrated Development Environment and Automatic Program Generator designed for the Atmel AVR family of microcontrollers.
It is designed to run under the XP, Vista, Windows 7 and Windows 8 32bit and 64bit operating systems.

The C cross-compiler implements all the elements of the ANSI C language, as allowed by the AVR architecture, with some features added to take advantage of specificity of the AVR architecture and the embedded system needs.
The compiled COFF object files can be C source level debugged, with variable watching, using the Atmel Studio and AVR Studio debuggers.

The Integrated Development Environment (IDE) has built-in AVR Chip In-System Programmer software that enables the automatic transfer of the program to the microcontroller chip after successful compilation/assembly. The In-System Programmer software is designed to work in conjunction with the Atmel STK500, STK600, AVRISP, AVRISP MkII, AVR Dragon, JTAGICE MkII, JTAGICE 3, AVRProg (AVR910 application note), Kanda Systems STK200+, STK300, Dontronics DT006, Vogel Elektronik VTEC-ISP, Futurlec JRAVR and MicroTronics' ATCPU, Mega2000 development boards.
For debugging embedded systems, which employ serial communication, the IDE has a built-in Terminal.

CodeVisionAVR can be also used as an extension in Atmel Studio 6.1 or later, allowing seamless editing, compiling and debugging projects in this IDE.

Besides the standard C libraries, the CodeVisionAVR C compiler has dedicated libraries for:

- Alphanumeric and Graphic LCD modules
- Philips I$^2$C bus
- National Semiconductor LM75 Temperature Sensor
- Philips PCF8563, PCF8583, Maxim/Dallas Semiconductor DS1302 and DS1307 Real Time Clocks
- Maxim/Dallas Semiconductor 1 Wire protocol
- Maxim/Dallas Semiconductor DS1820, DS18S20 and DS18B20 Temperature Sensors
- Maxim/Dallas Semiconductor DS1621 Thermometer/Thermostat
- Maxim/Dallas Semiconductor DS2430 and DS2433 EEPROMs
- SPI
- TWI for both XMEGA and non-XMEGA chips
- Power management
- Delays
- Gray code conversion
- MMC/SD/SD HC FLASH memory cards low level access
- FAT acces on MMC/SD/SD HC FLASH memory cards.

CodeVisionAVR also contains the CodeWizardAVR Automatic Program Generator, that allows you to write, in a matter of minutes, all the code needed for implementing the following functions:

- External memory access setup
- Chip reset source identification
- Input/Output Port initialization
- External Interrupts initialization
- Timers/Counters initialization
- Watchdog Timer initialization
- UART (USART) initialization and interrupt driven buffered serial communication
- Analog Comparator initialization
- ADC and DAC initialization
- SPI Interface initialization

- Two Wire Interface initialization
- CAN Interface initialization
- $I^2C$ Bus, LM75 Temperature Sensor, DS1621 Thermometer/Thermostat and PCF8563, PCF8583, DS1302, DS1307 Real Time Clocks initialization
- 1 Wire Bus and DS1820/DS18S20 Temperature Sensors initialization
- Alphanumeric and graphic display module initialization.



## 1.1 Credits

The HP InfoTech team wishes to thank:

- Mr. Jack Tidwell for his great help in the implementation of the floating point routines
- Mr. Yuri G. Salov for his excellent work in improving the Mathematical Functions Library
- Mr. Olivier Wuillemin and Mr. Franc Marx for their help in beta testing
- Mr. Lee H. Theusch for his support in improving the compiler
- Mr. ChaN from Electronic Lives Mfg. http://elm-chan.org for the open source **MMC/SD/SD HC FLASH Memory Card Driver** and **FAT File System** functions.

# 2. Using the CodeVisionAVR Extension for Atmel Studio

The CodeVisionAVR extension allows editing, building and program debugging of the C source files grouped in projects, from within the Atmel Studio 6.1 or later IDE.

The following chapters cover only the specific aspects of CodeVisionAVR integrated in Atmel Studio. For more details about using Atmel Studio, please refer to its documentation.

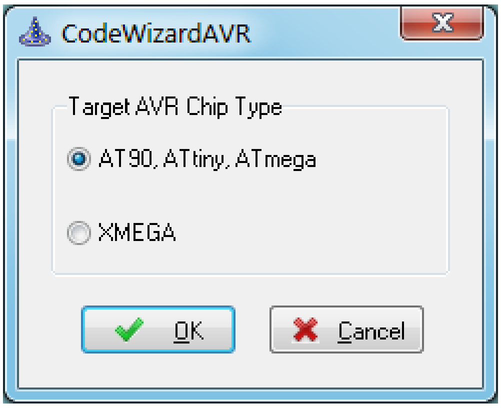
## 2.1 Working with Projects and Solutions

The Project groups the source file(s) and compiler settings that you use for building a particular program.
Project files have the **.cproj** extension.
In Atmel Studio several projects can be grouped in a Solution.
Solution files have the **.atsln** extension.

### 2.1.1 Creating a New Project using the CodeWizardAVR

New CodeVisionAVR projects can be created in Atmel Studio by invoking the CodeWizardAVR automatic program generator, using the **File|New|Project using CodeWizardAVR...** menu command.

The following dialog box will open:




allowing to select between the AVR chip families for which automatic code generation will be performed.

After the chip configuration was specified, as outlined in the chapters:

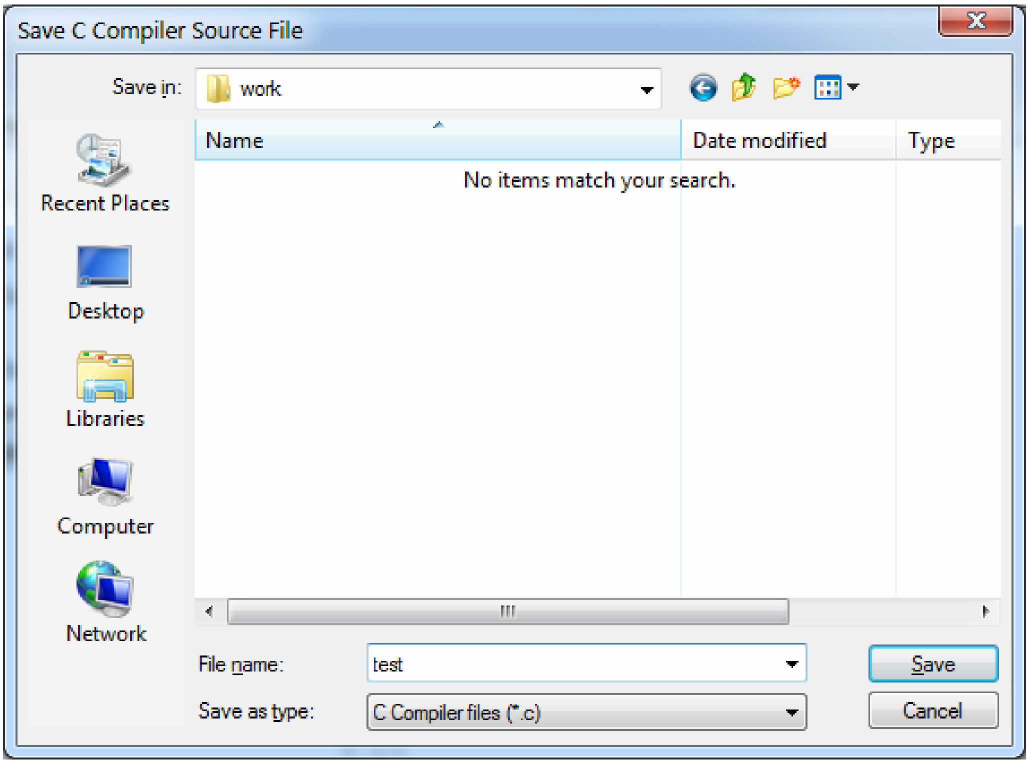
**6. CodeWizardAVR Automatic Program Generator**
**7. CodeWizardAVR Automatic Program Generator for the XMEGA Chips**

the **Program|Generate, Save and Exit** menu option must be selected or the  toolbar button must be clicked in CodeWizardAVR.

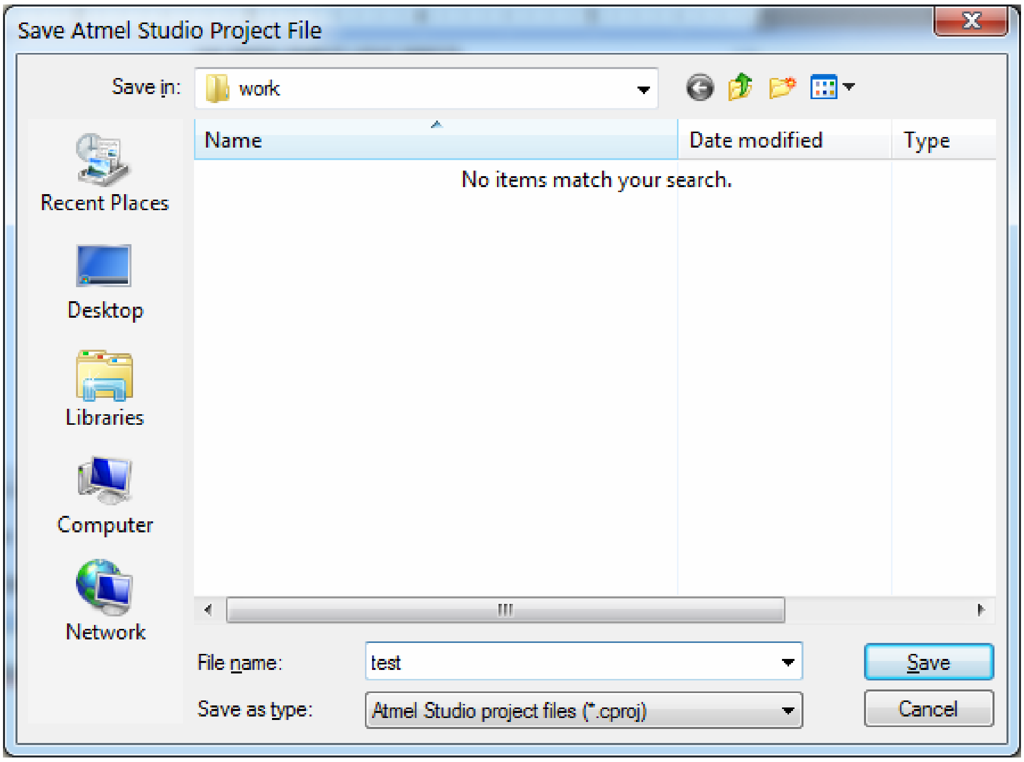
It will generate the main **.c** source, Atmel Studio project **.cproj** and it's own **.cwp** (for non-XMEGA chips) or **.cwx** (for XMEGA chips) project files.

Eventual peripheral configuration conflicts will be prompted to the user, allowing him to correct the errors.

In the course of program generation the user will be prompted for the name of the first **.c** source file of the project:

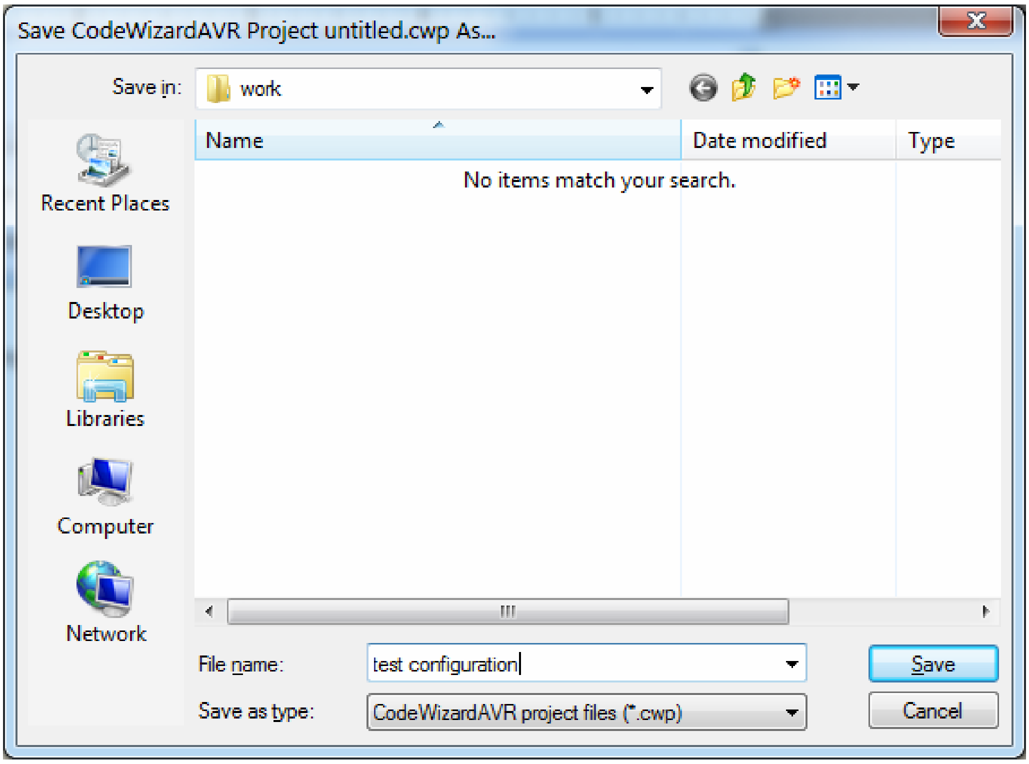



and for the name of the CodeVisionAVR project for Atmel Studio file:

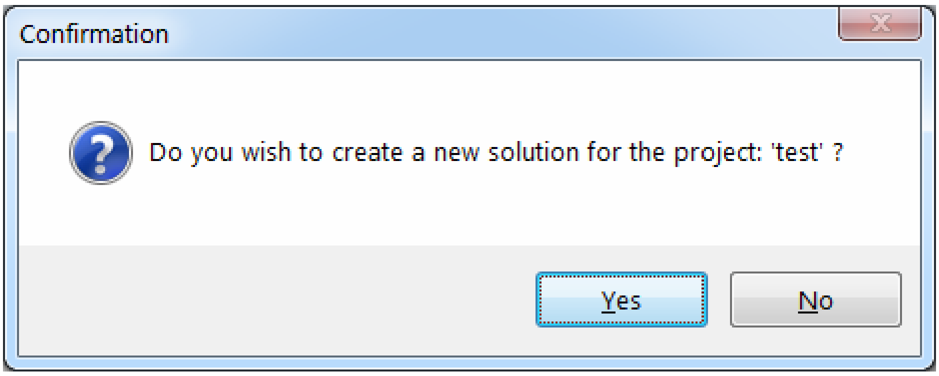


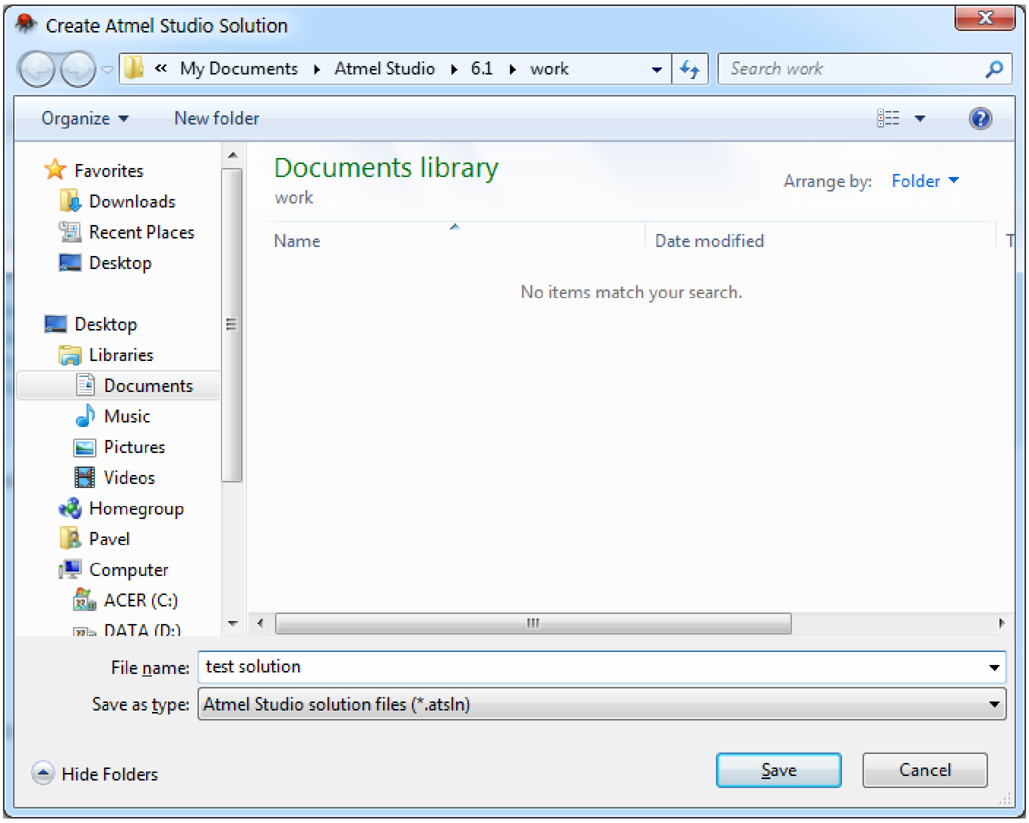
The user will be also prompted to save the current chip configuration in a CodeWizardAVR **.cwp** or **.cwx** project file:




Once all these files were created, the user will be prompted for creating an Atmel Studio solution which will incorporate the new CodeVisionAVR project:



If the answer is affirmative, the user will be prompted for the Atmel Studio solution name:

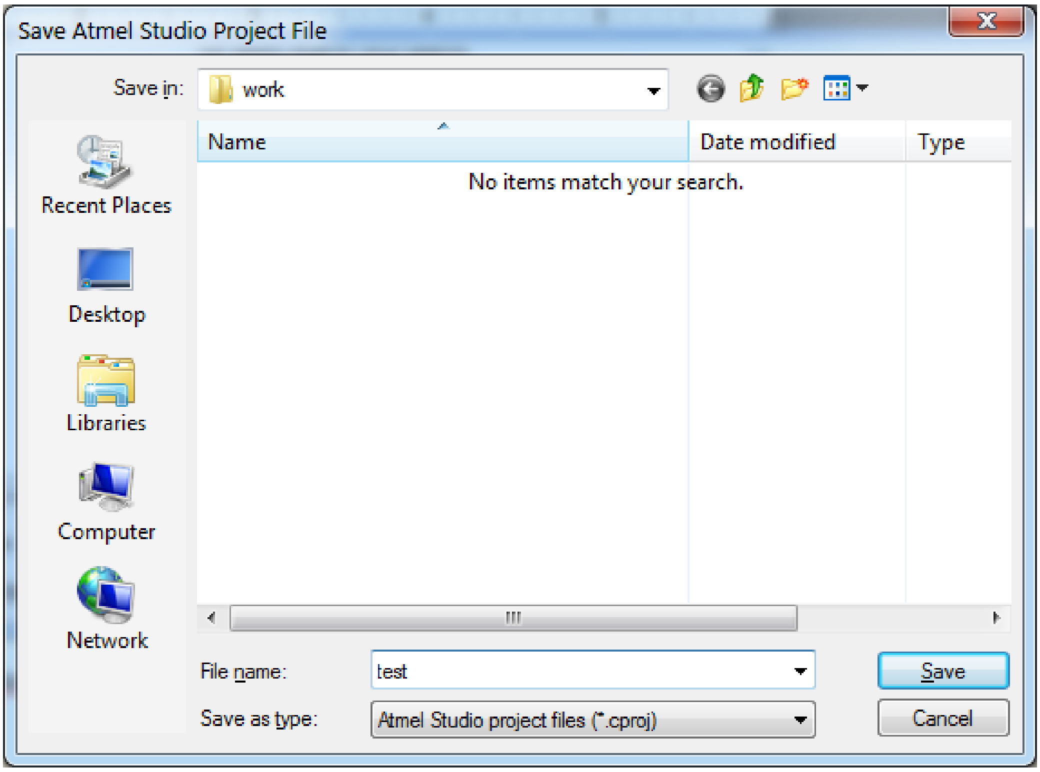



If a solution is already opened, the user will be prompted for the inclusion of the new project in that solution, without creating a new one.

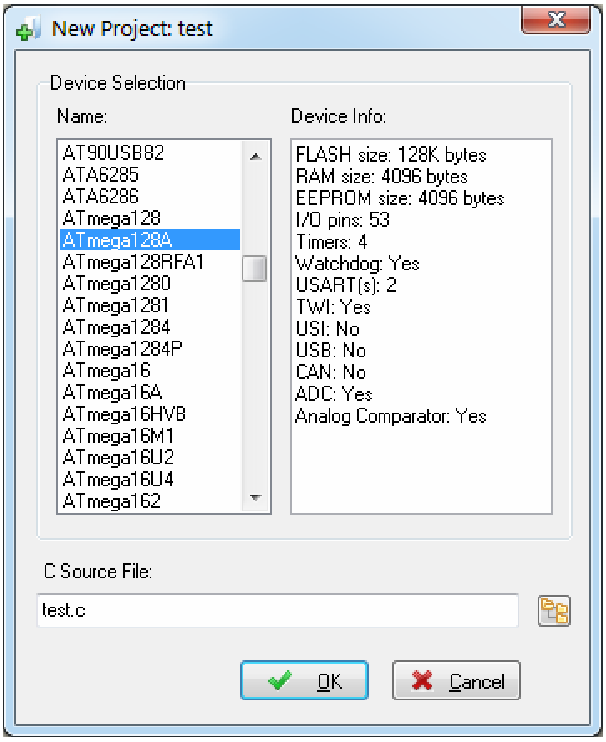
**Note:** When an Atmel Studio **.cproj** project is created, a corresponding **.prj** project file for the CodeVisionAVR IDE will be created too.
This allows editing/compiling the same project in both Atmel Studio and CodeVisionAVR IDE.

## 2.1.2 Creating a New Project without using the CodeWizardAVR

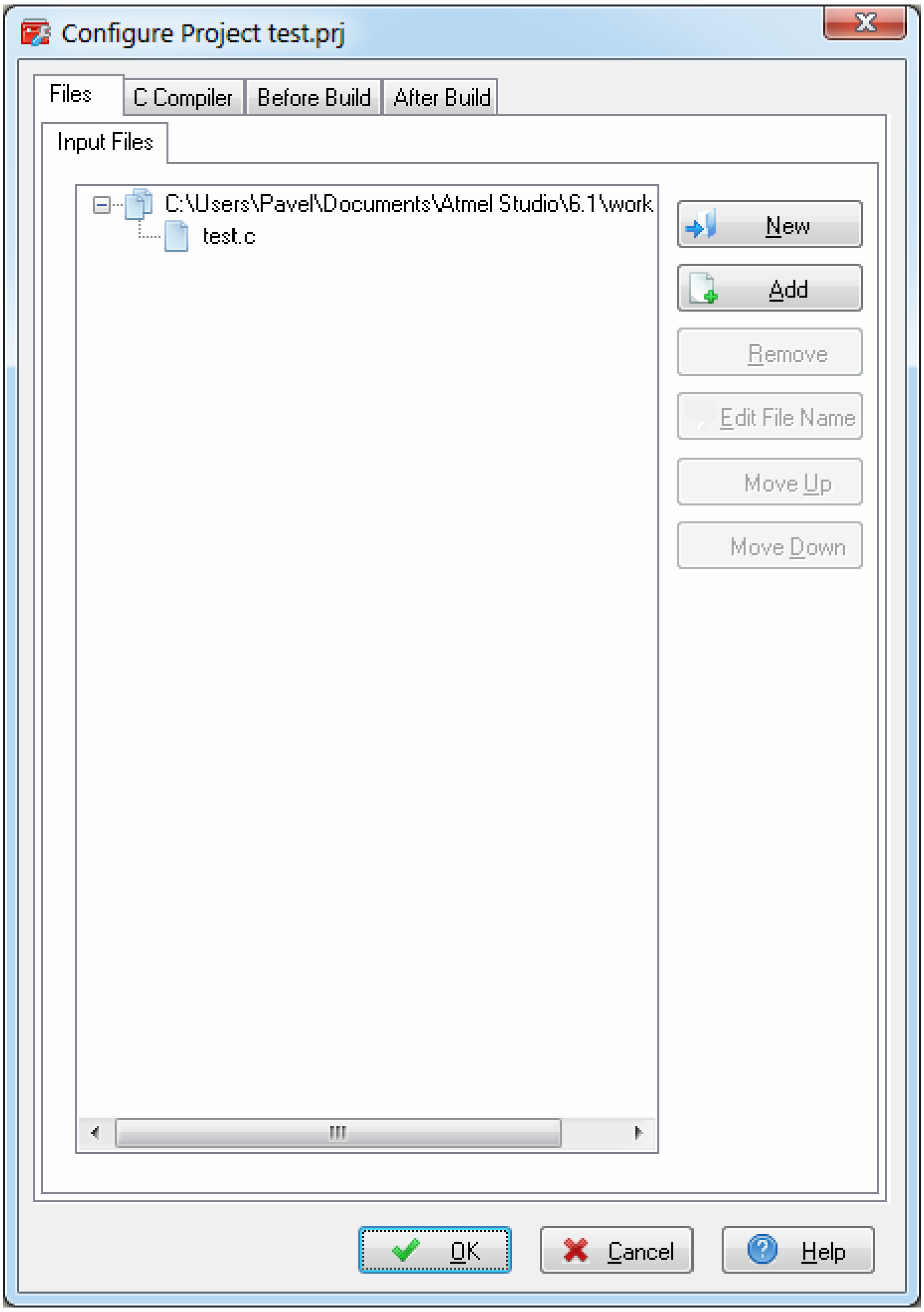
New CodeVisionAVR projects can be created in Atmel Studio using the **File|New|Project for CodeVisionAVR...** menu command.
The user will be prompted for the project file name:




After the project file name was specified, the user will be prompted to select the target chip type and the name of the first **.c** source file of the project:

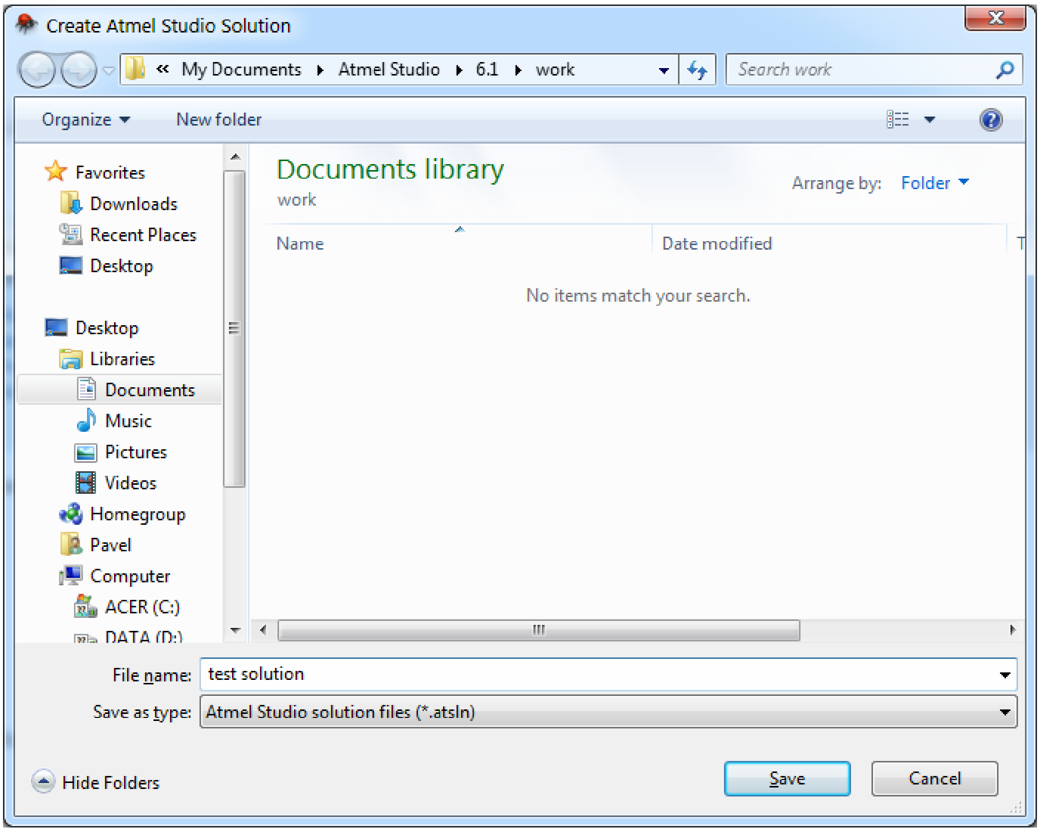


Once these are specified, a configuration window for the newly created project will be displayed:




More details can be found in the chapter: **3.3.6 Configuring the Project**.

After the project configuration was specified, the user will be prompted for the name of the Atmel Studio solution that will hold the new project:

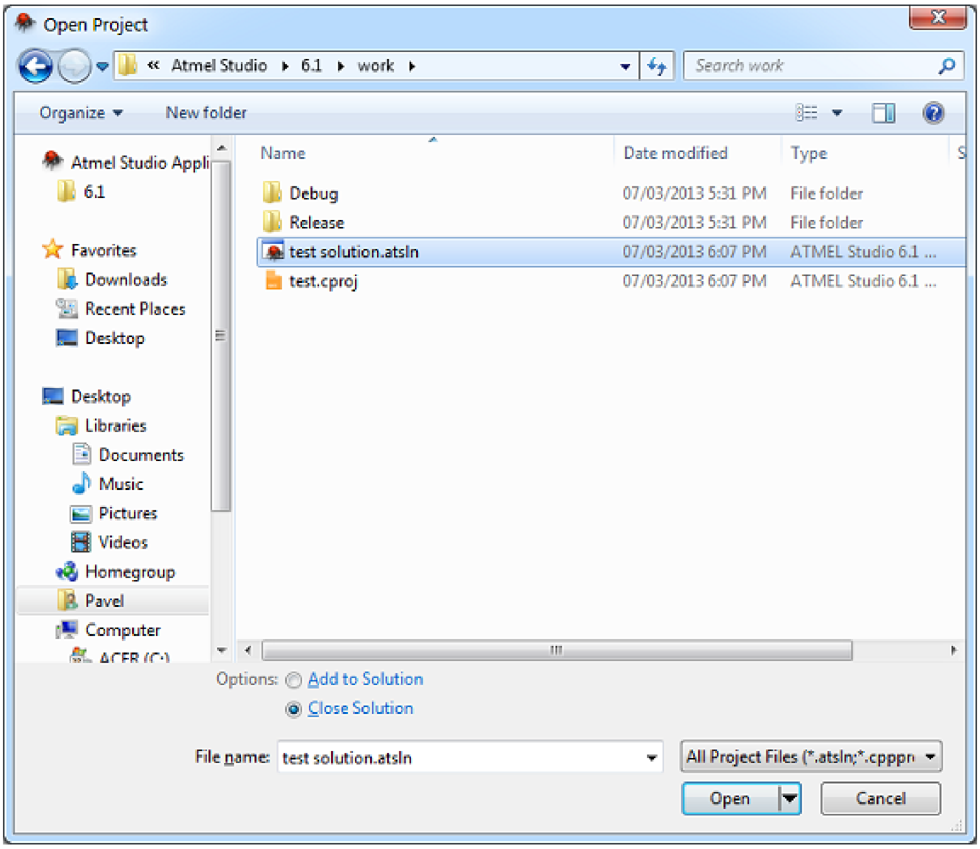



If a solution is already opened, the user will be prompted for the inclusion of the new project in that solution, without creating a new one.

**Note:** When an Atmel Studio **.cproj** project is created, a corresponding **.prj** project file for the CodeVisionAVR IDE will be created too.
This allows editing/compiling the same project in both Atmel Studio and CodeVisionAVR IDE.

### 2.1.3 Opening an Existing Project or Solution

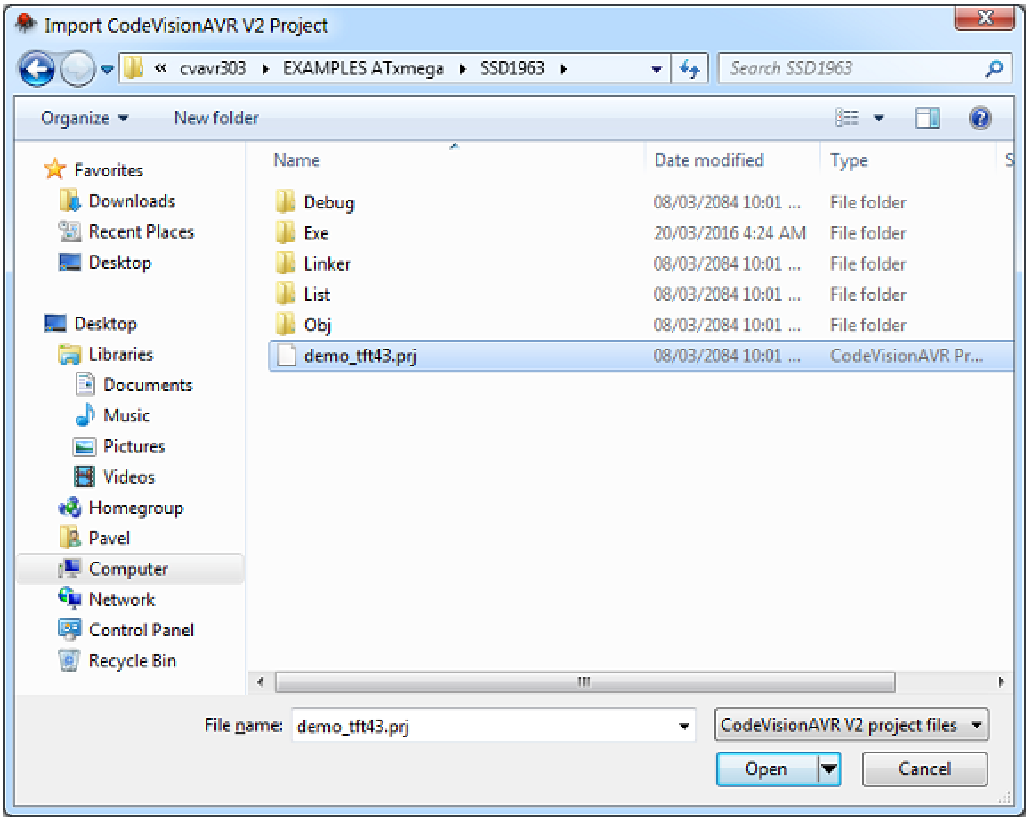
An existing project or solution can be opened in Atmel Studio using the **File|Open|Project/Solution** menu command.
The following dialog window will be displayed:



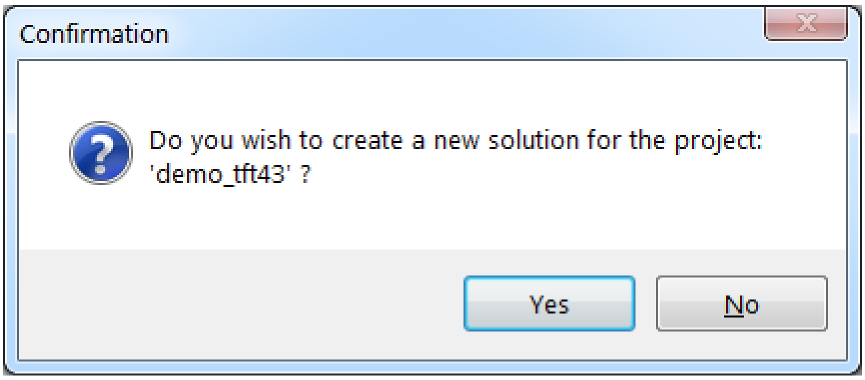

allowing the user to select the project or solution he wishes to open.

## 2.1.4 Importing a CodeVisionAVR V2 Project

A project **.prj** file created for the CodeVisionAVR V2.xx compiler can be imported in Atmel Studio using the **File|Open|Import CodeVisionAVR V2 Project...** menu command.
A dialog window will de displayed, allowing the user to select the project to be imported:

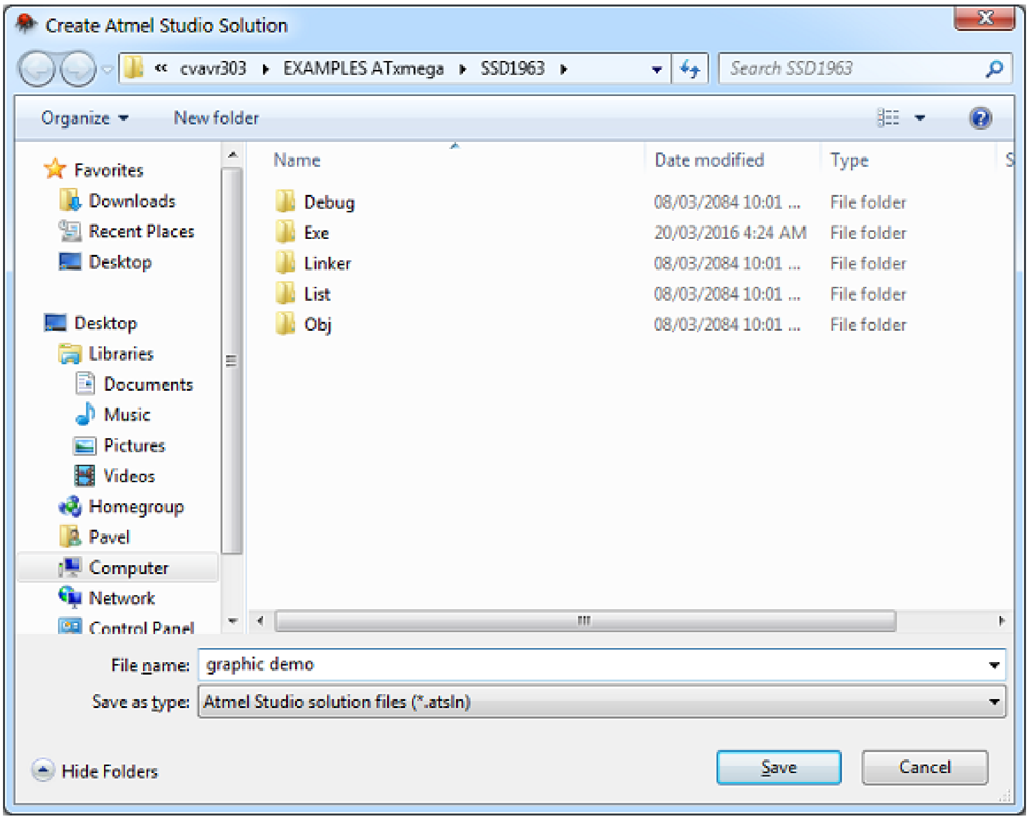



A corresponding Atmel Studio **.cproj** project file will be created.
If no solution is currently opened, the user will be prompted to create a new one:



If the answer is affirmative, the user will be prompted to specify the name of the new solution that will hold the imported project:




If a solution is already opened, the user will be prompted for the inclusion of the imported project in that solution, without creating a new one.

## 2.1.5 Configuring the Project

The Project can be configured using the **Project|Configure** *project name* menu command.
The available options are detailed in the chapters:

**3.3.6.1 Adding or Removing a File from the Project**
**3.3.6.2 Setting the Project Output Directories**
**3.3.6.3 Settings the C Compiler Options**
**3.3.6.4 Setting the 1 Wire Library Options**
**3.3.6.5 Setting the I2C Library Options**
**3.3.6.6 Setting the MMC/SD/SD HC Card Library Options**
**3.3.6.7 Setting the Alphanumeric LCD Library Options**
**3.3.6.8 Setting the Graphic Display Library Options**
**3.3.6.9 Executing an User Specified Program before Build**
**3.3.6.10 Transferring the Compiled Program to the AVR Chip**
**3.3.6.11 Executing an User Specified Program after Build**.

## 2.1.6 Obtaining an Executable Program

Obtaining an executable program requires the following steps:

1. Compiling the Project's C program modules, using the CodeVisionAVR C Compiler, and obtaining object files needed by the linker
2. Linking the object files files created during compilation and obtaining a single assembler source file
3. Assembling the assembler source file, using the Atmel AVR assembler AVRASM2.

The resulting **.rom**, **.hex** and **.eep** files will be placed in the **Executable Files** directory.
The object files, including the **.cof** COFF object file used for debugging, will be placed in the **Object Files** directory.
The **.asm**, **.lst** and **.map** files will be placed in the **List Files** directory.
Various files created by the linker during the **Build** process will be placed in the **Linker Files** directory.

The **Executable Files**, **Object Files,** **List Files** and **Linker Files** directories are specified in the **Project|Configure|Files|Output** menu.

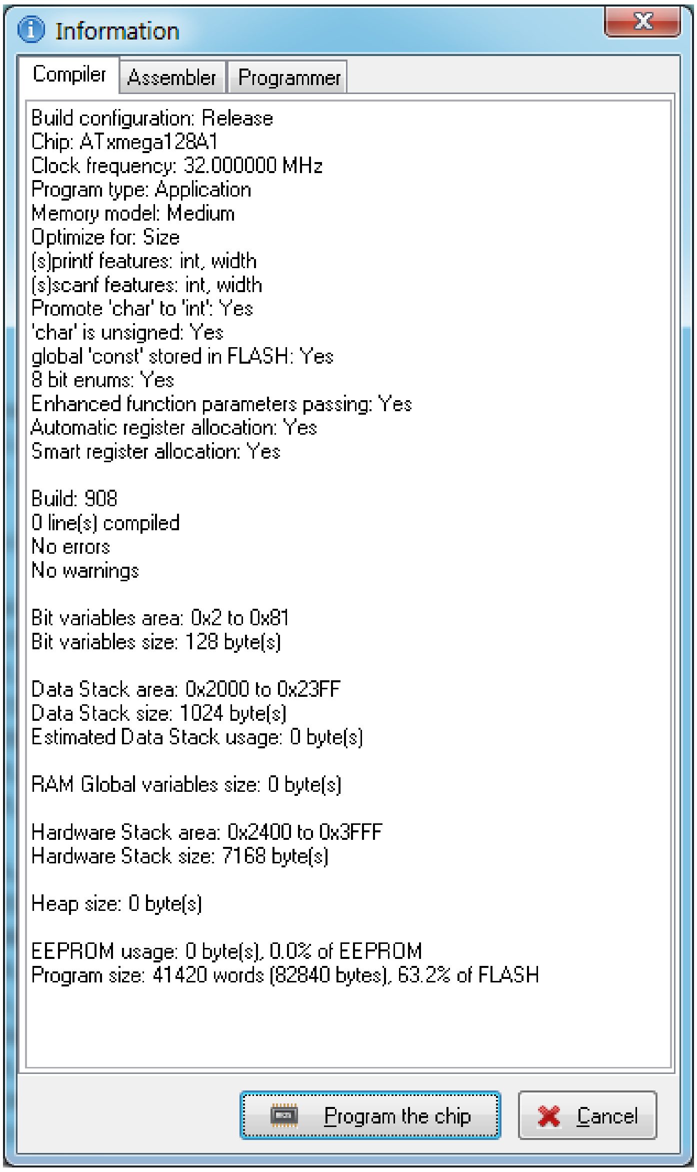
## 2.1.6.1 Building the Project

To build the Project you must use the **Build|Build** *project name* menu command or the button of the toolbar.
The CodeVisionAVR C Compiler will be executed, producing the object files needed by the linker.
Compilation will be performed only for the program modules that were modified since the previous similar process.
If the complete recompilation of all the program modules is needed, then the **Build|Rebuild** *project name* menu command must be used.

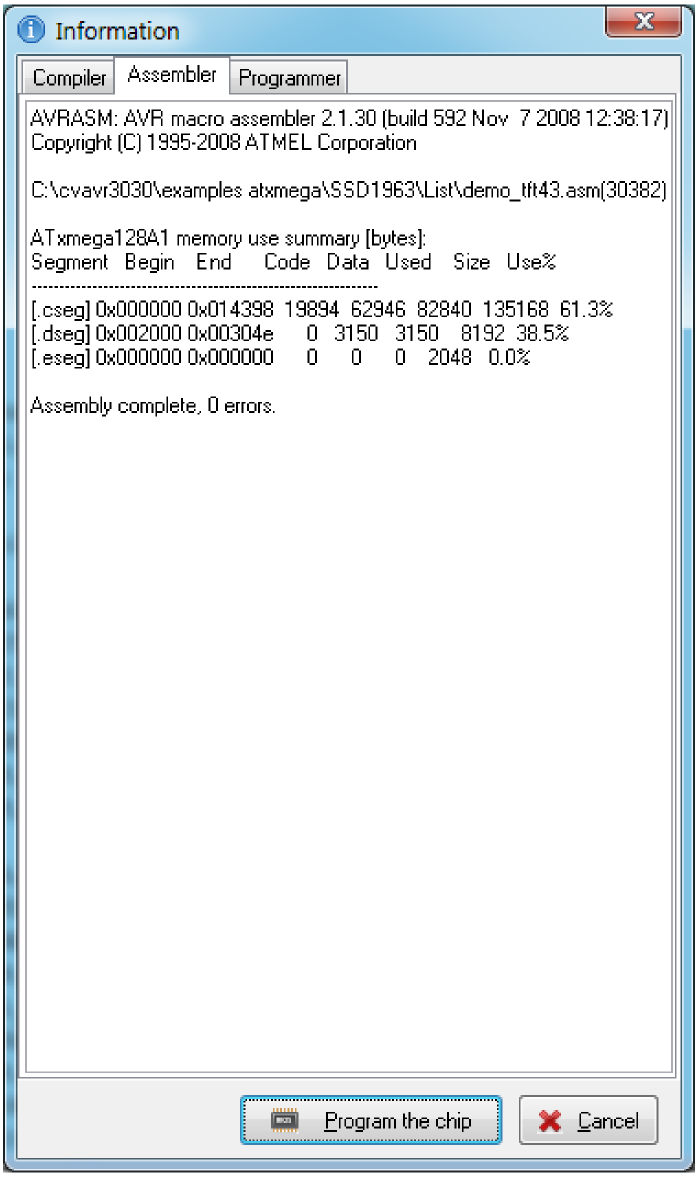
After successful compilation the object files will be linked and an assembly .asm file will be produced.
If no compilation or linking errors were encountered, then the Atmel AVR assembler AVRASM2 will be executed, obtaining the output file types specified in **Project|Configure|C Compiler|Code generation |File Output Formats**.

Eventual compilation errors and/or warnings will be listed in Atmel Studio's **Output** window.

After the build process is completed, an Information window will open showing the build results.
Pressing the **Compiler** tab will display compilation results.

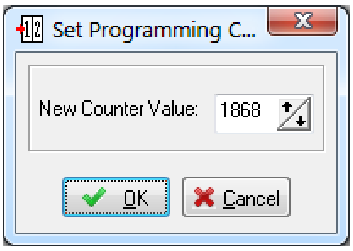


Pressing the **Assembler** tab will display assembly results.



Pressing the **Programmer** tab will display the **Chip Programming Counter**, which shows how many times was the AVR chip programmed so far.


Pressing the **Set Counter** button will open the **Set Programming Counter** window:




This dialog window allows setting the new **Chip Programming Counter** value.
Pressing the **Program the chip** button allows automatic programming of the AVR chip after successful build. Pressing **Cancel** will disable automatic programming.

The Information window for the last build, can be always displayed using the **Project|Build Information for** *project name* menu.

### 2.1.6.2 Cleaning Up the Project Output Directories

The various files created during the **Build** process can be deleted using the **Build|Clean** menu.
The following **Project Output Directories** will be cleaned:

- **Object Files** directory - all files will be deleted, except the **.cof** COFF object file
- **List Files** directory - all files will be deleted, except the **.asm** and **.vec** assembly source files
- **Linker Files** directory – all files will be deleted.

## 2.1.7 Debugging the Executable Program

Once the program was successfully built, it can be debugged in source level form, using the Atmel Studio's **Debug|Start Debugging and Break** menu command, the toolbar button or by pressing the **Alt+F5** keys.
If the source files were modified since the last **Build**, a **Rebuild** will be automatically performed, before the debugging session will be started.

Atmel Studio uses the **.cof** object file produced by CodeVisionAVR for debugging.
Therefore it is important that this file is created, by selecting in the **Project|Configure|C Compiler|Code generation|File Output Formats** list box the following formats for the files generated by the compiler: **COFF, ROM, Intel HEX and EEP.**

The following commands can be used when debugging:

- **Debug|Step Into**, F11 key or toolbar button to execute one instruction
- **Debug|Step Over**, F10 key or toolbar button to execute one instruction.
  If the instruction contains a function call, the function is executed as well.
- **Debug|Step Out**, Shift+F11 keys or toolbar button to continue execution until the current function has completed
- **Debug|Run To Cursor**, Ctrl+F10 keys or toolbar button to continue execution until the current cursor position in the source file or disassembly view is reached
- **Debug|Reset**, Shift+F5 keys or toolbar button to restart program execution from the beginning
- **Debug|Restart** or toolbar button to restart the debugger and reload the debugged program
- **Debug|Toggle Breakpoint** or F9 key to set a breakpoint at the current cursor position in the C source file or disassembly view
- **Debug|New Breakpoint|Break at Function** to set a breakpoint at the beginning of a particular function
- **Debug|Delete All Breakpoints** or Ctrl+Shift+F9 keys to delete all the breakpoints that were set
- **Debug|Disable All Breakpoints** to temporarily disable all the breakpoints that were set
- **Debug|Enable All Breakpoints** to re-enable all the breakpoints that were set
- **Debug|Continue**, F5 key or toolbar button to continue execution after a breakpoint
- **Debug|Break All**, Ctrl+F5 keys or toolbar button to stop program execution
- **Debug|Windows** allow displaying specific windows for watching variables, processor registers, I/O and peripheral registers, memory contents, code disassembly, etc.

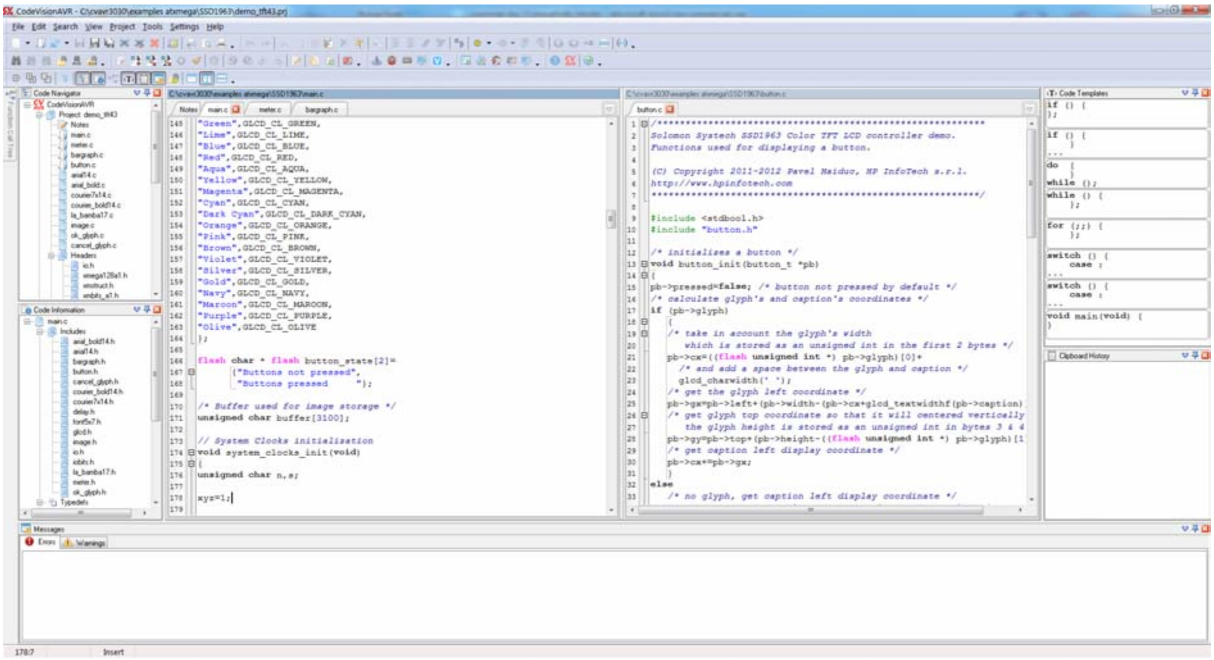
To obtain more information about using the debugger, please consult the Atmel Studio Help.

**Note:** The compiler applies some optimization techniques that may prevent correct debugging of the executable program.
Therefore it is recommended to select the **Project|Configure|C Compiler|Code generation|Optimize for: Speed** option for code debugging.
If the program fits in the chip's FLASH, this option must be left enabled for **Release** too, as the program will execute faster this way.

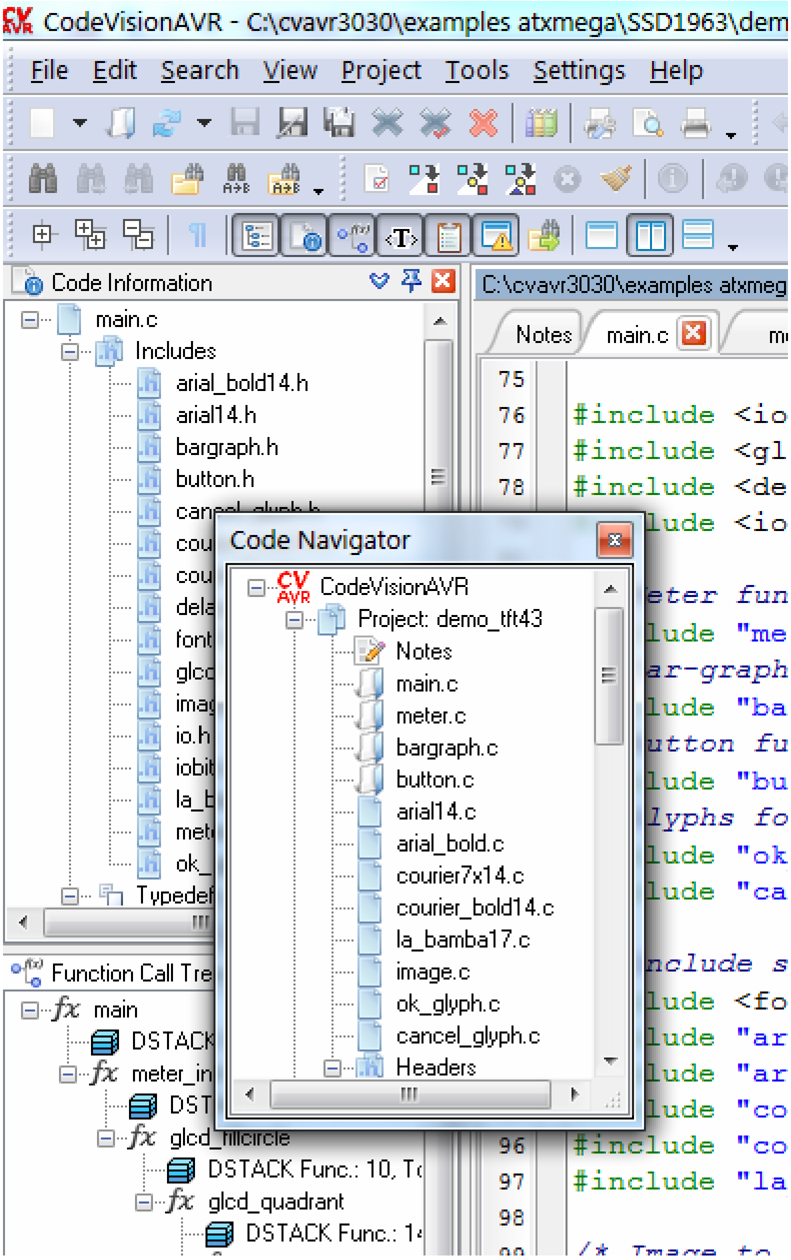
# 3. The CodeVisionAVR Integrated Development Environment

## 3.1 Using the Integrated Development Environment Workspace

The CodeVisionAVR IDE workspace consist from several windows that may be docked to the main application window or left floating on the desktop to suit the user's preferences.

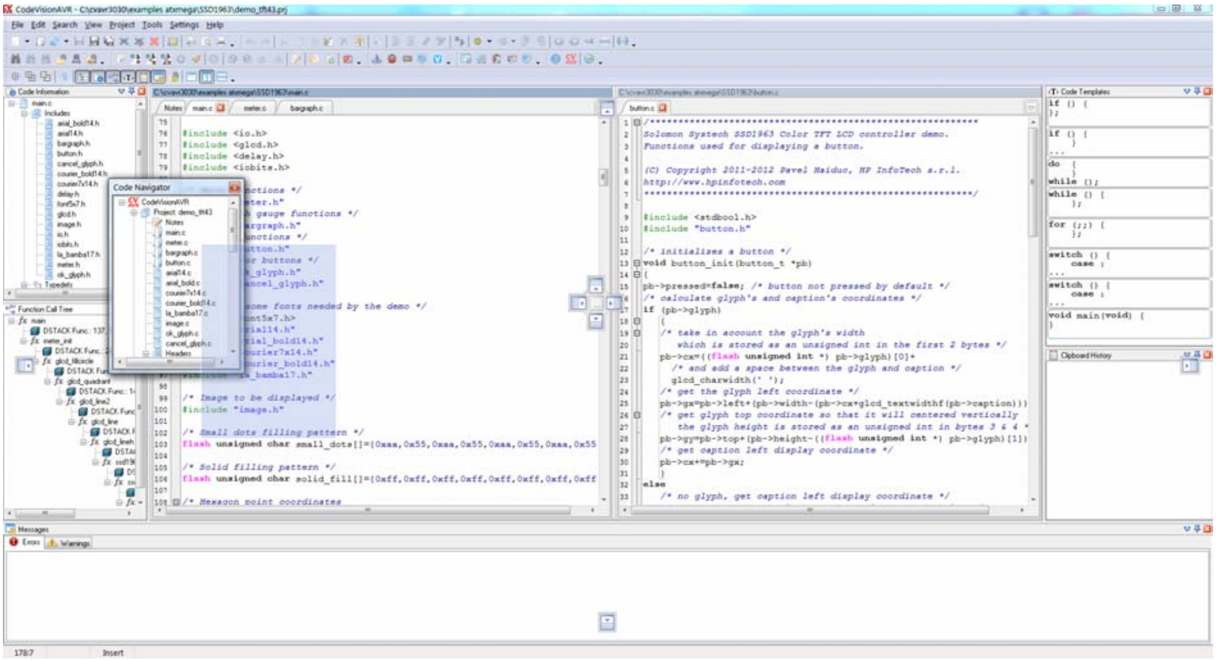


In order to undock a window, its top bar must be clicked with the left mouse button and keeping the button pressed, dragged to any suitable position on the desktop.

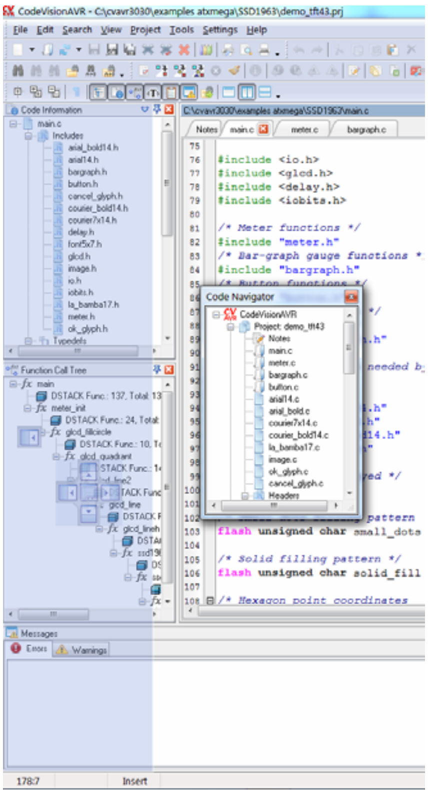


The window can be resized by dragging its margins and corners with the left mouse button pressed.

An undocked window can be docked to any position in the main application window or even to another docked window.
In order to dock the window, its top bar must be dragged, keeping the left mouse button pressed.
The possible dock locations of the window are outlined with special docking markers like in the picture below:

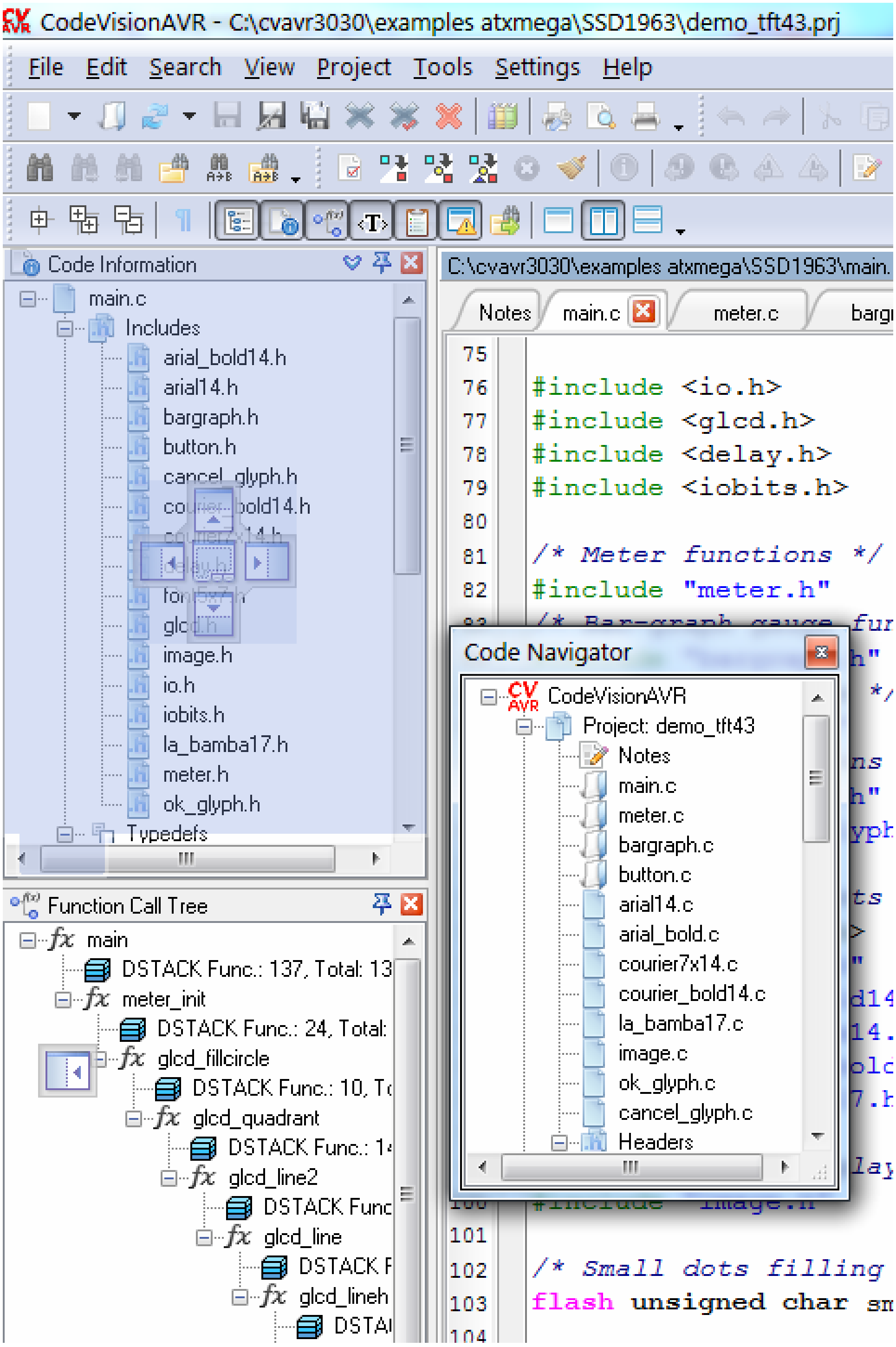


When the mouse cursor arrives on one of the docking markers, the future docking position will be outlined:

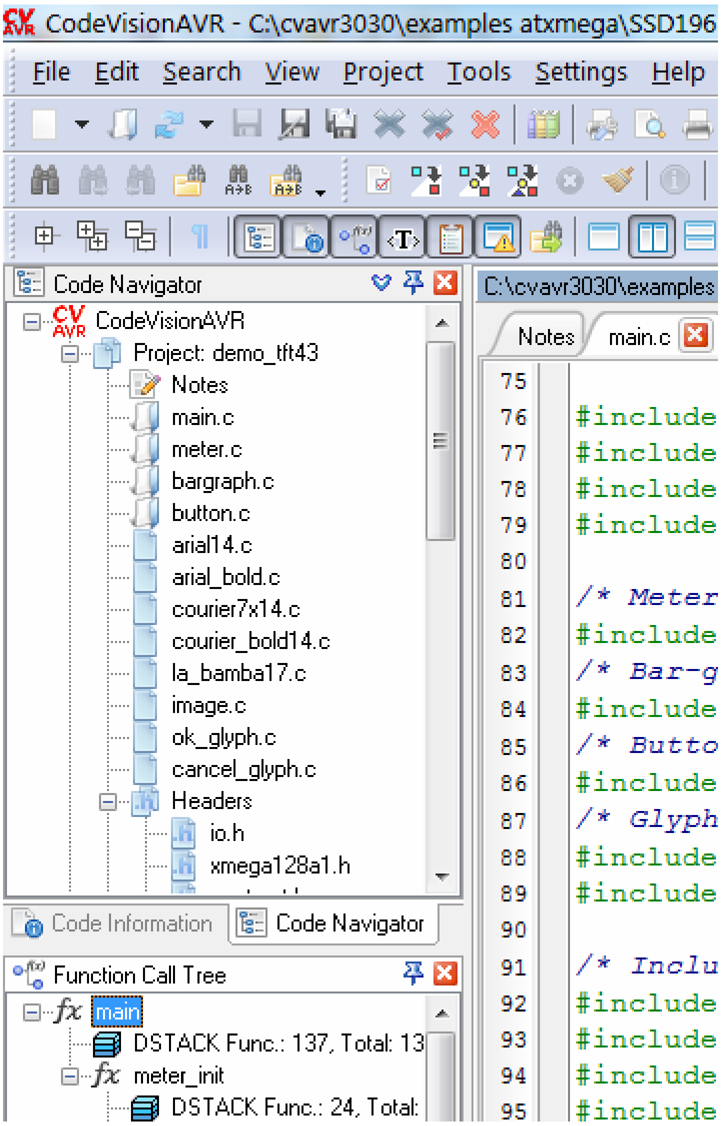


After the mouse button will be released, the window will become docked.

If the window is desired to be docked to another docked window, the future position of the window will be that of a tabbed page, like in the picture below:



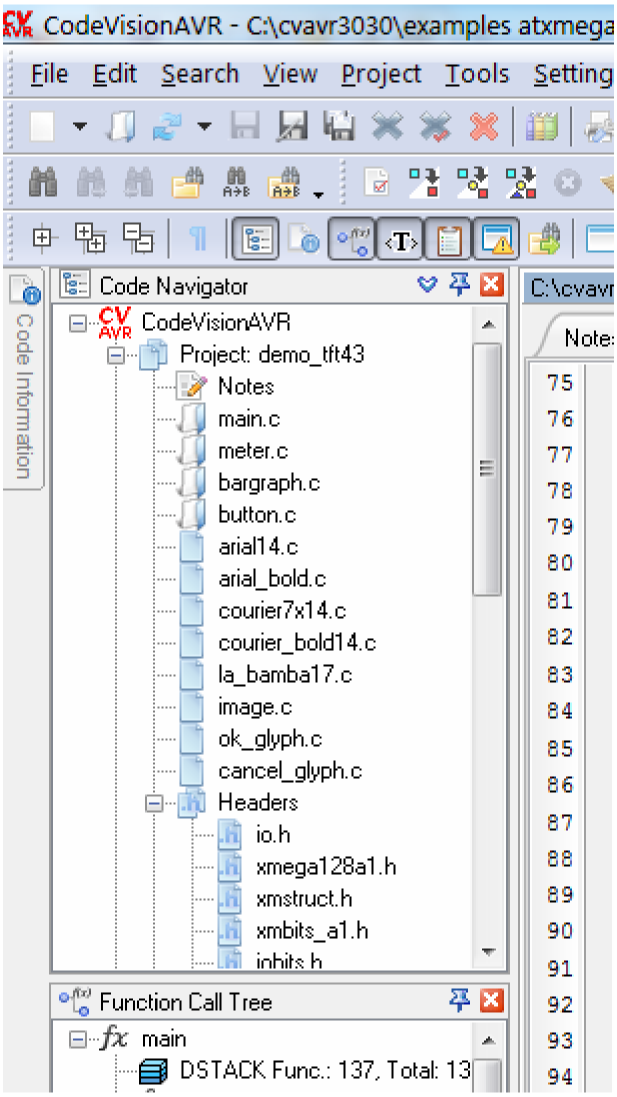
Once docked, the window will become a tabbed page:



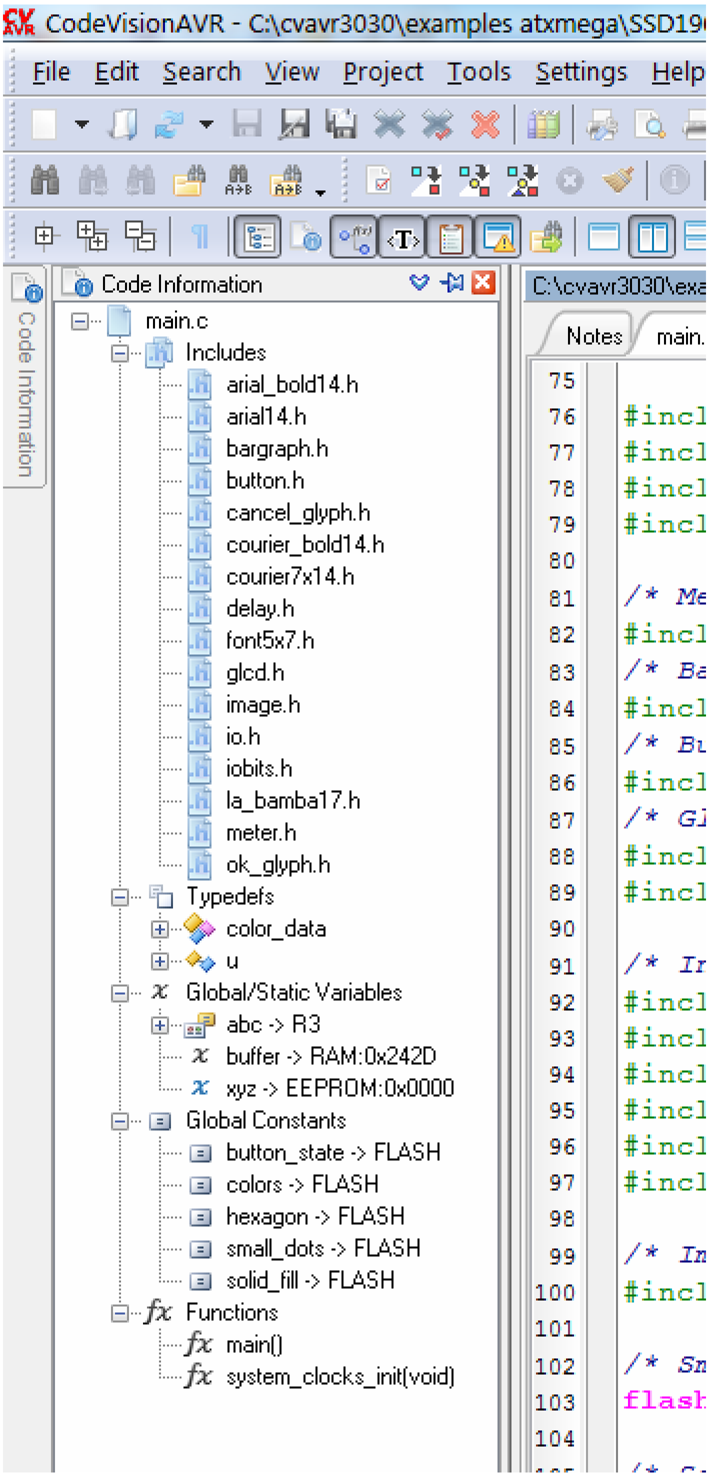

To undock a single tabbed page, the bottom tab must be dragged with the mouse.

A workspace window can be hidden by left clicking on its icon, by pressing its corresponding button on the **View** toolbar or by using the **View** menu.
A window's corresponding button on the **View** toolbar must be pressed or the **View** menu must be used in order to make a hidden window visible again.

Clicking on the icon will make the docked window temporarily hidden, its position will be displayed by a vertical bar located on the left or right of the docking site:

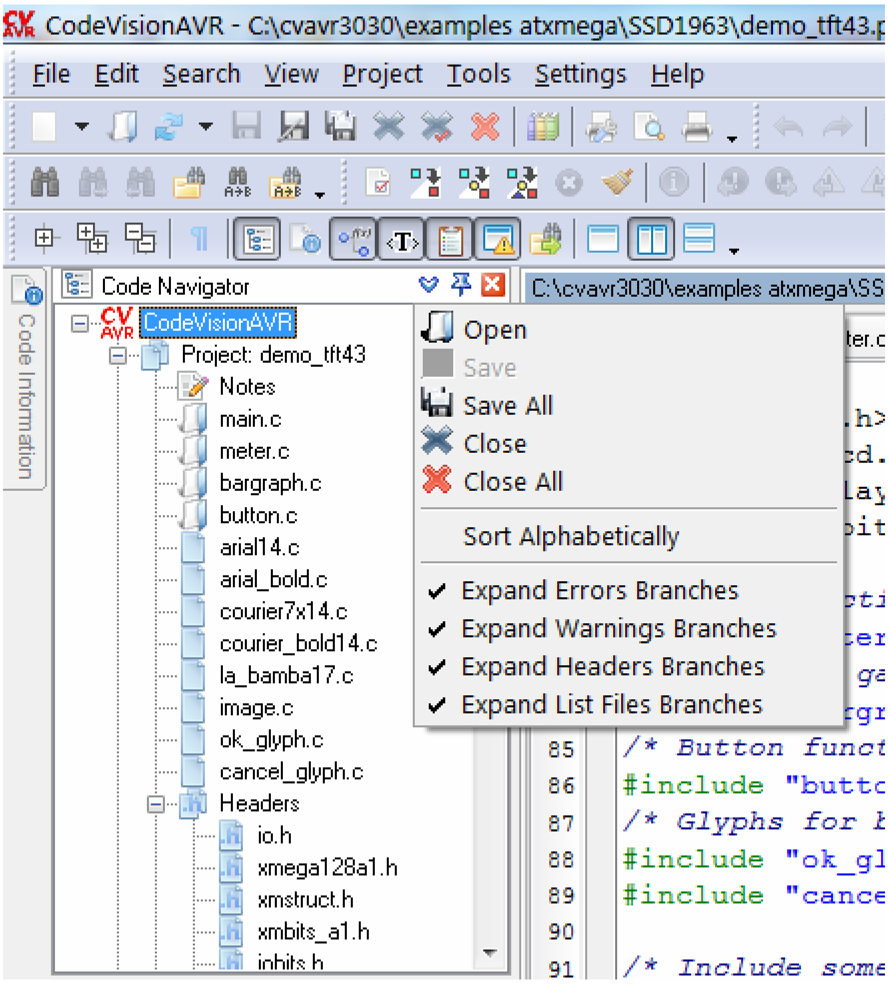


If the user will place the mouse cursor on the vertical bar, the hidden window will be displayed for a short amount of time



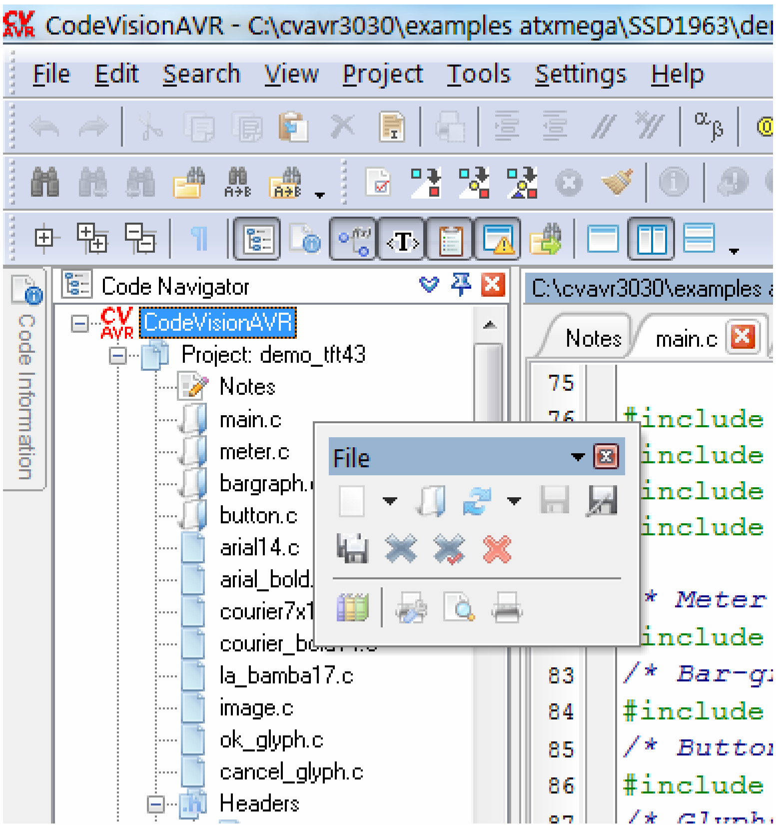
and then will become hidden again.
In order to lock the temporarily displayed window in position, the user must click on the icon.

Clicking with the mouse on the window icon will open a specific drop down menu:



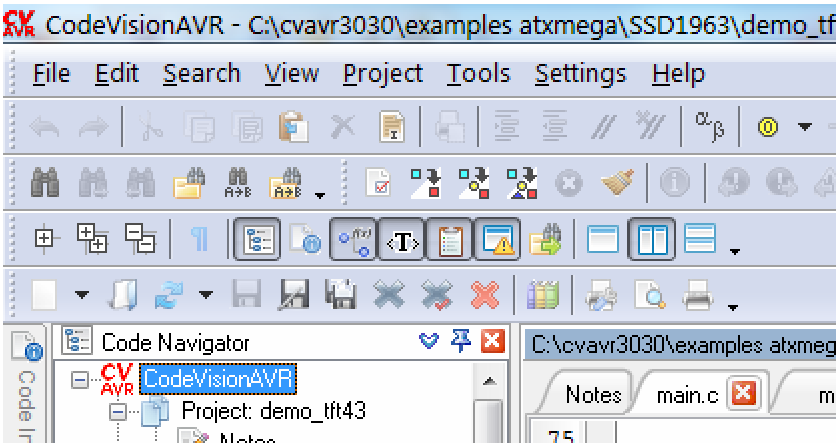

Alternatively this menu can be also invoked by right clicking with the mouse inside the window.

The menu toolbars can be placed to any position, by clicking with the left mouse button on the  handle and dragging it, while keeping the button pressed.
If the toolbar is moved outside the menu, it will become floating, like in the following picture:



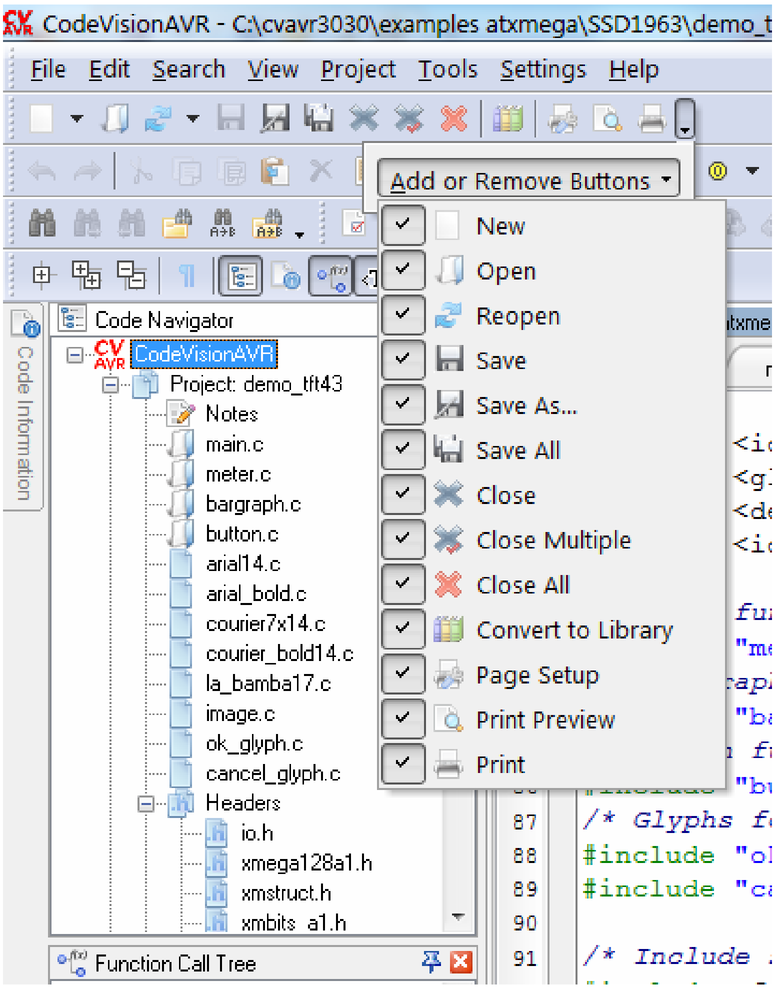

and can be placed anywhere on the desktop.

The toolbar can be docked to a new position:

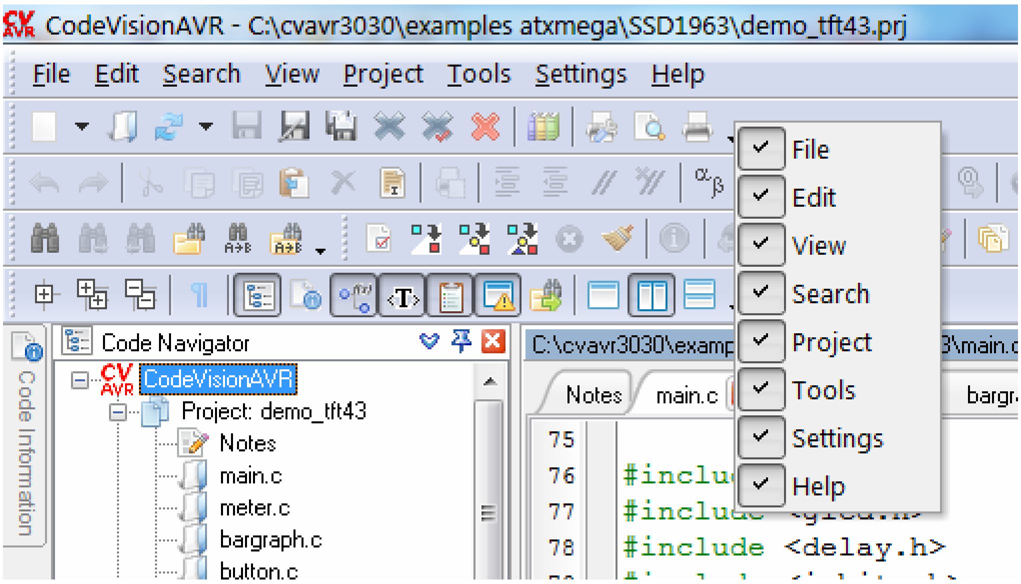



An undocked toolbar can be hidden by clicking on its icon.
Alternatively the toolbars' visible state can be changed by using the **View|Toolbars** menu.

The buttons on a toolbar can be individually enabled or disabled by left clicking with the mouse on the button.
A drop-down menu will open for this purpose:



The visibility state of the toolbars can be also individually modified by right clicking with the mouse on the button.
A drop-down menu will open for this purpose:




All the workspace layout will be automatically saved at program exit and restored back on the next launch.

The Editor uses a tabbed multiple window interface.
The following key shortcuts are available:

- Ctrl+TAB - switch to the next editor tabbed window
- Ctrl+Shift+TAB - switch to the previous editor tabbed window
- Ctrl+W - close the current editor tabbed window.

The current editor tabbed window can be also closed by clicking on the icon located on the top right of the tabbed control.

The Editor can display the edited files in one of the following modes:

- Single Editor Pane
- Dual Vertical Editor Pane
- Dual Horizontal Editor Pane.

Switching between the above mentioned modes is performed using the **View|File Panes** menu or the buttons of the **View** toolbar.

In dual editor pane mode, the active pane is selected by left-clicking with the mouse on the pane's top caption bar:


All **File**, **Edit**, **Search** menu operations will be performed on the active editor pane.

The editor panes can be resized by dragging the pane splitter with the mouse.

An editor pane can be maximized by double-clicking with the mouse left button on its top caption bar. By double-clicking once again, the pane is restored to its original size.

## 3.2 Working with Files

Using the CodeVisionAVR IDE you can view and edit any text file used or produced by the C compiler or assembler.

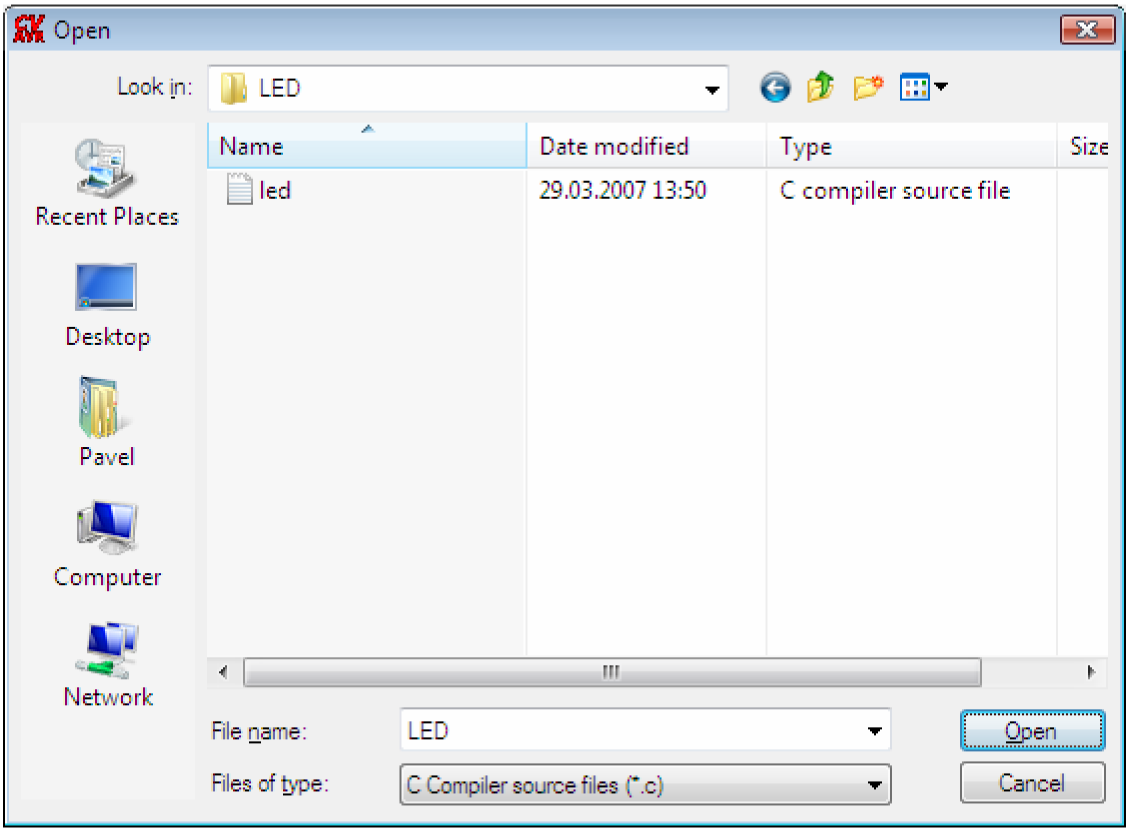
### 3.2.1 Creating a New File

You can create a new source file using the **File|New|Source File** menu command, by pressing the **Ctrl+N** keys or the and buttons on the toolbar.

A new editor window appears for the newly created file.
The new file has the name **untitled.c**. You can save this file under a new name using the **File|Save As** menu command or the toolbar button.

## 3.2.2 Opening an Existing File

You can open an existing file using the **File|Open** menu command, by pressing the **Ctrl+O** keys or the button on the toolbar.
An **Open** dialog window appears.




You must select the name and type of file you wish to open.
By pressing the **Open** button you will open the file in a new editor window.

## 3.2.3 Files History

The CodeVisionAVR IDE keeps a history of the opened files.
The most recent eight files that where used can be reopened using the **File|Reopen** menu command or the toolbar button.

## 3.2.4 Editing a File

A previously opened or a newly created file can be edited in the editor window by using the **Tab**, **Arrows**, **Backspace** and **Delete** keys.

Pressing the **Home** key moves the cursor to the start of the current text line.
Pressing the **End** key moves the cursor to the end of the current text line.
Pressing the **Ctrl+Home** keys moves the cursor to the start of the file.
Pressing the **Ctrl+End** keys moves the cursor to the end of the file.

Portions of text can be selected by dragging with the mouse.

You can copy the selected text to the clipboard by using the **Edit|Copy** menu command, by pressing the **Ctrl+C** keys or by pressing the button on the toolbar.

By using the **Edit|Cut** menu command, by pressing the **Ctrl+X** keys or by pressing the button on the toolbar, you can copy the selected text to the clipboard and then delete it from the file.

Text previously saved in the clipboard can be placed at the current cursor position by using the **Edit|Paste** menu command, by pressing the **Ctrl+V** keys or pressing the button on the toolbar.

Clicking in the left margin of the editor window allows selection of a whole line of text.
Selected text can be deleted using the **Edit|Delete** menu command, by pressing the **Ctrl+Delete** keys or the toolbar button.

Dragging and dropping with the mouse can move portions of text.
Pressing the **Ctrl+Y** keys deletes the text line where the cursor is currently positioned.

Changes in the edited text can be undone, respectively redone, by using the **Edit|Undo**, respectively **Edit|Redo**, menu commands, by pressing the **Ctrl+Z**, respectively **Shift+Ctrl+Z** keys, or by pressing the , respectively buttons on the toolbar.

Clicking with the mouse right button in the Editor window, opens a pop-up menu that gives access to the above mentioned functions.

### 3.2.4.1 Searching/Replacing Text

You can find, respectively replace, portions of text in the edited file by using the **Search|Find**, respectively **Search|Replace**, menu commands, by pressing the **Ctrl+F**, respectively **Ctrl+R** keys, or by pressing the , respectively  buttons on the toolbar.
The **Search|Find Next**, respectively **Search|Find Previous**, functions can be used to find the next, respectively previous, occurrences of the search text.
The same can be achieved using the **F3**, respectively **Ctrl+F3** keys or the , respectively the , toolbar buttons.
Searching, respectively replacing, portions of text in files can be performed using the **Search|Find in Files**, respectively **Search|Replace in Files**, menu commands, by pressing the **Ctrl+Shift+F**, respectively **Ctrl+Shift+H** keys, or by pressing the , respectively  buttons on the toolbar.
These functions are also available in the pop-up menu, invoked by mouse right clicking in the Editor window.

### 3.2.4.2 Setting Bookmarks

Bookmarks can be inserted or removed, at the line where the cursor is positioned, by using the **Edit|Toggle Bookmark** menu command, by pressing the **Shift+Ctrl+0...9** keys or the  toolbar button.
The **Edit|Jump to Bookmark** menu command, the **Ctrl+0...9** keys or the  toolbar button will position the cursor at the start of the corresponding bookmarked text line.
Jumping to the next bookmark can be achieved by using the **Edit|Jump to Next Bookmark** menu command, by pressing the **F2** key or by using the  toolbar button.
Jumping to the previous bookmark can be achieved by using the **Edit|Jump to Previous Bookmark** menu command, by pressing the **Shift+F2** keys or by using the  toolbar button.

After a jump to a bookmark was performed, the **Edit|Go Back** menu command or the  toolbar button allow to return to the previous position in the file.
The **Edit|Go Forward** menu command or the  toolbar button allow to return to the file position before the **Edit|Go Back** menu command or the  toolbar button were used.

These functions are also available in the pop-up menu, invoked by mouse right clicking in the Editor window.

### 3.2.4.3 Jumping to a Symbol Definition or Declaration

When the editor cursor is located on a symbol name and the **Edit|Go to Definition/Declaration** menu command is performed, the **F12** key or the  toolbar button are pressed, a jump will be performed to the symbol definition or declaration located in any of the project's source files.

After a jump to the definition or declaration was performed, the **Edit|Go Back** menu command or the  toolbar button allow to return to the previous position in the edited file.
The **Edit|Go Forward** menu command or the  toolbar button allow to return to the file position before the **Edit|Go Back** menu command or the  toolbar button were used.

These functions are also available in the pop-up menu, invoked by mouse right clicking in the Editor window.

### 3.2.4.4 Jumping to a Specific Line Number in the Edited File

You can go to a specific line number in the edited file, by using the **Edit|Go to Line** menu command, by pressing the **Ctrl+G** keys or the toolbar button.

After a jump to a specific line was performed, the **Edit|Go Back** menu command or the toolbar button allow to return to the previous position in the edited file.
The **Edit|Go Forward** menu command or the toolbar button allow to return to the file position before the **Edit|Go Back** menu command or the toolbar button were used.

These functions are also available in the pop-up menu, invoked by mouse right clicking in the Editor window.

### 3.2.4.5 Printing a Text Selection

Portions of text can be selected by dragging with the mouse.
The **Edit|Print Selection** menu command or the toolbar button allows the printing of the selected text.
This function is also available in the pop-up menu, invoked by mouse right clicking in the Editor window.

### 3.2.4.6 Indenting/Unindenting a Text Selection

Portions of text can be selected by dragging with the mouse.
Selected portions of text can be indented, respectively unindented, using the **Edit|Indent Selection**, respectively **Edit|Unindent Selection**, menu commands, by pressing the **Ctrl+I**, respectively **Ctrl+U** keys or the , respectively , toolbar buttons.
These functions are also available in the pop-up menu, invoked by mouse right clicking in the Editor window.

### 3.2.4.7 Commenting/Uncommenting a Text Selection

Portions of text can be selected by dragging with the mouse.
Selected portions of text can be commented, respectively uncommented, using the **Edit|Comment Selection**, respectively **Edit|Unindent Selection**, menu commands, by pressing the **Ctrl+[**, respectively **Ctrl+]** keys or the , respectively , toolbar buttons.
These functions are also available in the pop-up menu, invoked by mouse right clicking in the Editor window.

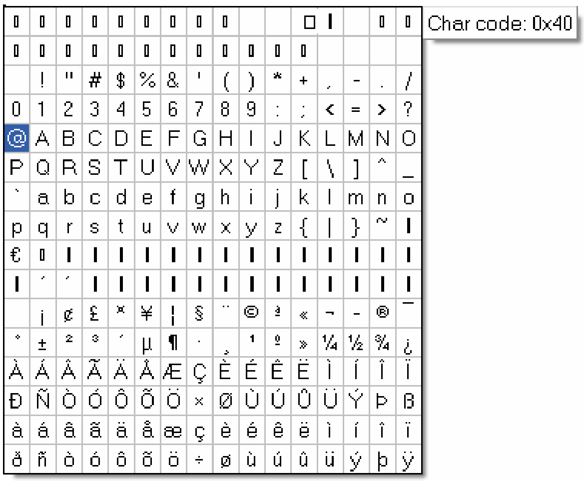
### 3.2.4.8 Match Braces

If the cursor is positioned before an opening, respectively after a closing, brace then selecting the **Edit|Match Braces** menu command, pressing the **Ctrl+M** keys or the toolbar button will position the cursor after, respectively before, the corresponding matching closing, respectively opening brace.
This function is also available in the pop-up menu, invoked by mouse right clicking in the Editor window.

## 3.2.4.9 Inserting Special Characters in the Text

Special characters can be inserted in the edited text, at the cursor is position, by using the **Edit|Insert Special Characters** menu command, by pressing the **Ctrl+.** keys or the αβ toolbar button.

A pop-up window containing a character map grid will be displayed, allowing the user to select the appropriate character to be inserted:




This function is also available in the pop-up menu, invoked by mouse right clicking in the Editor window.

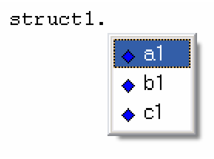
### 3.2.4.10 Using the Auto Complete Functions

The CodeVisionAVR Editor has the possibility to display pop-up hint windows for function parameters and structure or union members.
These functions can be enabled and configured using the **Settings|Editor|Auto Complete** menu.

Function parameter auto complete is automatically invoked when the user types the name of a function defined in the currently edited file, followed by a '**(**' auto completion triggering character.
A pop-up hint with parameter list will show like in the example below:


The parameter to be specified is highlighted with bold text.

Structure or union members auto complete is invoked after the user writes the name of a structure/union or pointer to structure/union followed by the '**.**' or '**->**' auto completion triggering characters.
A pop-up hint with the members list will show like in the example below:




The user can select the member to be inserted in the text at the cursor position, by using the arrow keys, respectively the mouse, and then pressing Enter, respectively the left mouse button.
The structure or union members auto completion works only for global structures/unions defined in the currently edited source file and after a **Project|Compile** or **Project|Build** was performed.

### 3.2.4.11 Using Code Folding

The CodeVisionAVR Editor has the possibility of displaying staples on the left side of code blocks delimited by the **{ }** characters.
For each code block there will be also displayed collapse or expansion marks on the gutter located on the left side of the Editor window. Clicking on these marks allow to individually fold or unfold blocks of code.

The **View|Toggle Fold** menu and the toolbar button allow to collapse/expand the block of code where the cursor is located.
The **View|Expand All Folds** menu and the toolbar button allow to expand all folded blocks of code.
The **View|Collapse All Folds** menu and the toolbar button allow to collapse all blocks of code delimited by the **{ }** characters.
These commands are also available in the pop-up menu that is invoked by right clicking with the mouse in the Editor window.

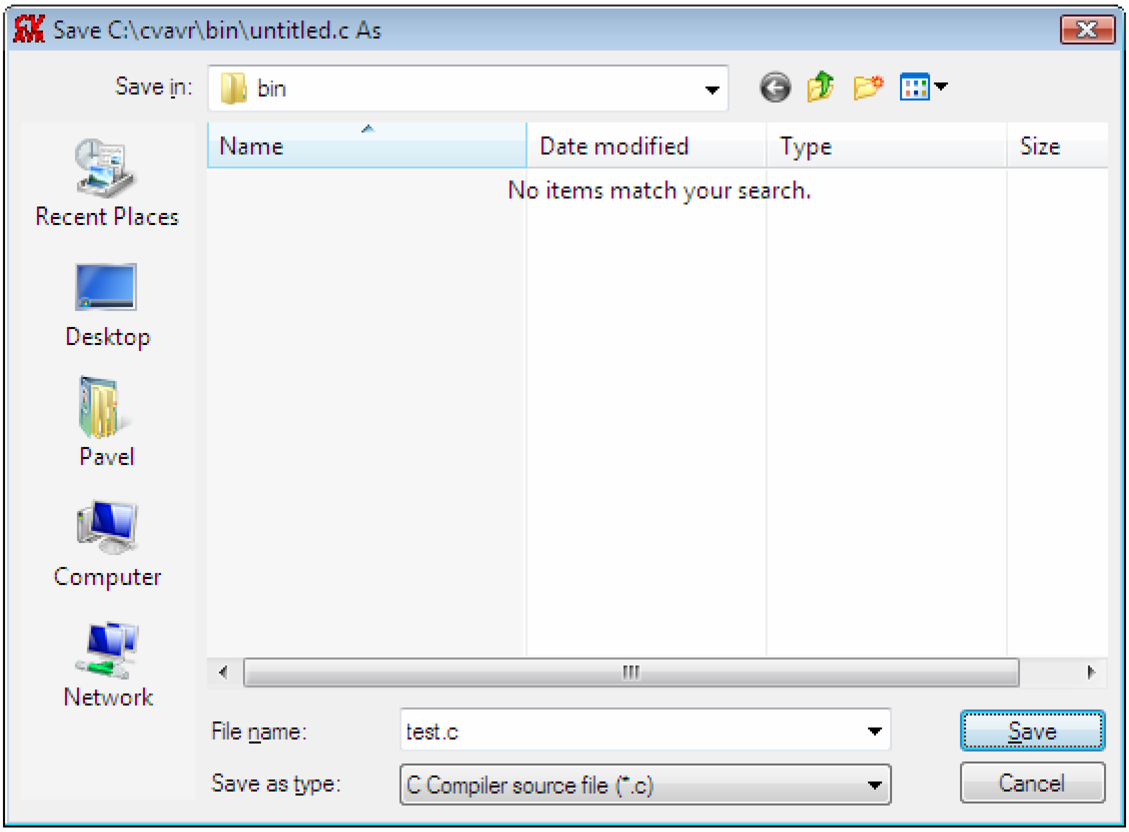
If the **Settings|Editor|General|Visual Aids|Save Folded Lines** option is enabled, the folded/unfolded state of the code blocks is saved when the file is closed and it will be restored back, when the file is opened again.

## 3.2.5 Saving a File

The currently edited file can be saved by using the **File|Save** menu command, by pressing the **Ctrl+S** keys or by pressing the button on the toolbar.
When saving, the Editor will create a backup file with a **~** character appended to the extension.
All currently opened files can be saved using the **File|Save All** menu command, by pressing the **Ctrl+Shift+S** keys or the toolbar button.

## 3.2.6 Renaming a File

The currently edited file can be saved under a new name by using the **File|Save As** menu command or the toolbar button.
A **Save As** dialog window will open.

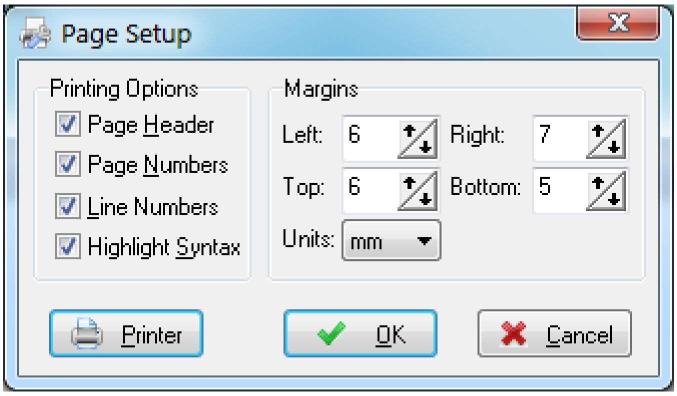



You will have the possibility to specify the new name and type of the file, and eventually its new location.

## 3.2.7 Printing a File

You can print the current file using the **File|Print** menu command or by pressing the button on the toolbar.
The contents of the file will be printed to the Windows default printer.

The paper margins used when printing can be set using the **File|Page Setup** menu command or the toolbar button, which opens the **Page Setup** dialog window.

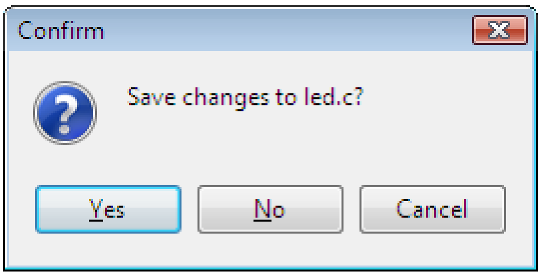



The units used when setting the paper margins are specified using the **Units** list box.
The printer can be configured by pressing the **Printer** button in this dialog window.
Changes can be saved, respectively canceled, using the **OK**, respectively **Cancel** buttons.

The print result can be previewed using the **File|Print Preview** menu command or by pressing the toolbar button.

## 3.2.8 Closing a File

You can quit editing the current file by using the **File|Close** menu command, the **Ctrl+F4** shortcut or the toolbar button.
The current editor tabbed window can be also closed by clicking on the icon located on the top right of the tabbed control.
If the file was modified, and wasn't saved yet, you will be prompted if you want to do that.

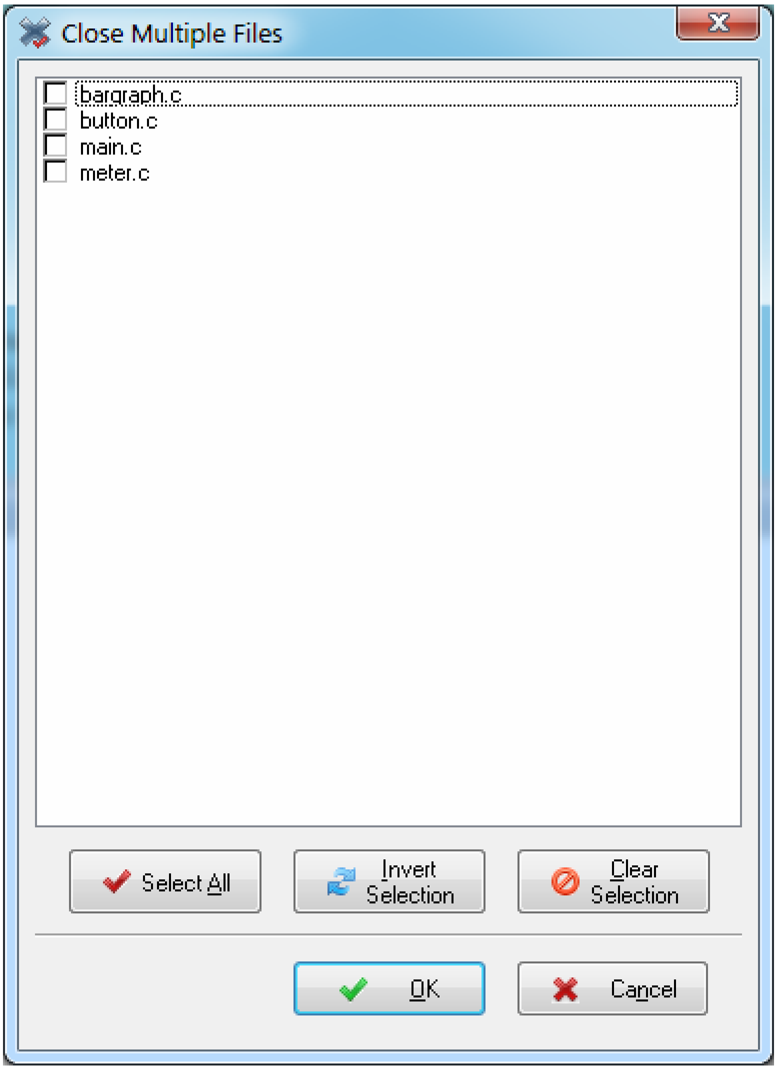



Pressing **Yes** will save changes and close the file.
Pressing **No** will close the file without saving the changes.
Pressing **Cancel** will disable the file closing process.

## 3.2.9 Closing Multiple Files

Closing several files can be performed using the **File|Close Multiple** menu command or the  toolbar button.
A dialog window, which lists all the opened files, will open for this purpose:

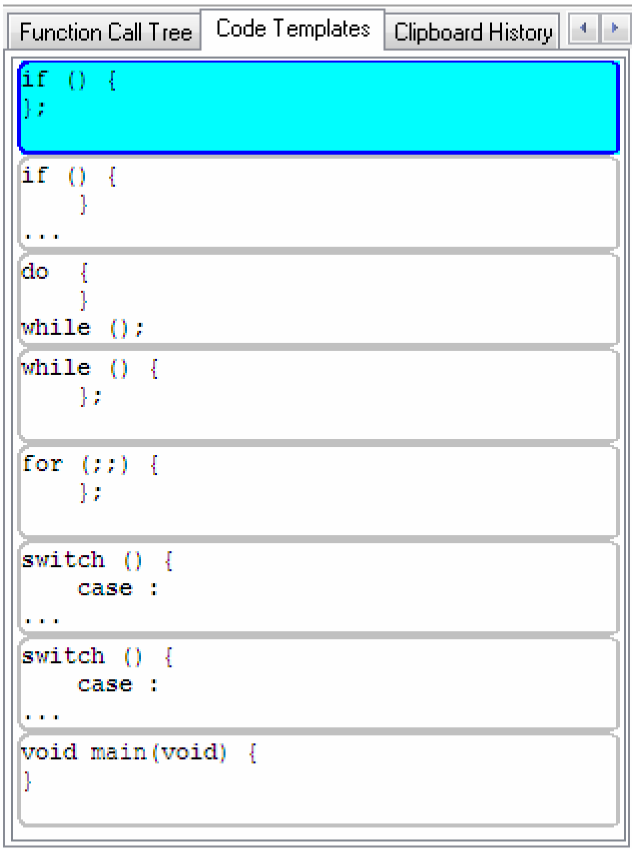



Files to be closed can be selected by checking the appropriate check boxes.
All the listed files can be selected using the **Select All** button.
The state of the check boxes can be reversed using the **Invert Selection** button.
The **Clear Selection** button can be used to un-check all the check boxes.

Pressing the **OK** button will close all the selected files from the list.
Pressing the **Cancel** button will close the dialog window, without closing any file.

## 3.2.10 Using the Code Templates

The **Code Templates** window allows easy adding most often used code sequences to the currently edited file.




This is achieved by clicking on the desired code sequence in the **Code Templates** window and then dragging and dropping it to the appropriate position in the Editor window.

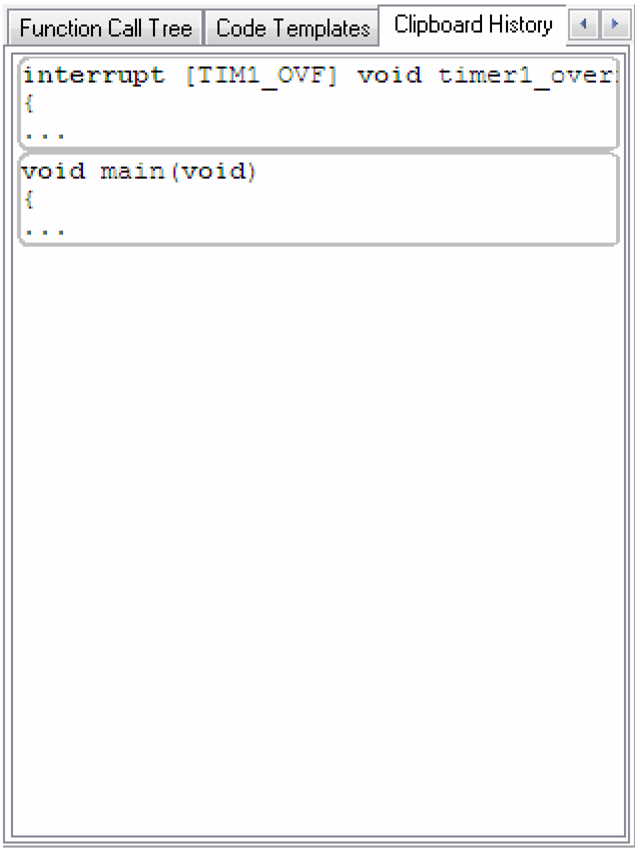
New code templates can be added to the list by dragging and dropping a text selection from the Editor window to the **Code Templates** window.

By right clicking in the **Code Templates** window you can open a pop-up menu with the following choices:

- **Copy to the Edit Window** the currently selected code template
- **Paste** a text fragment from the clipboard to the **Code Templates** window
- **Move Up** in the list the currently selected code template
- **Move Down** in the list the currently selected code template
- **Delete** the currently selected code template from the list.

## 3.2.11 Using the Clipboard History

The **Clipboard History** window allows viewing and accessing text fragments that were recently copied to the clipboard.




By right clicking in the **Clipboard History** window you can open a pop-up menu with the following choices:

- **Copy to the Edit Window** the currently selected text fragment from the **Clipboard History** window
- **Delete** the currently selected text fragment from the list
- **Delete All** the text fragments from the list.

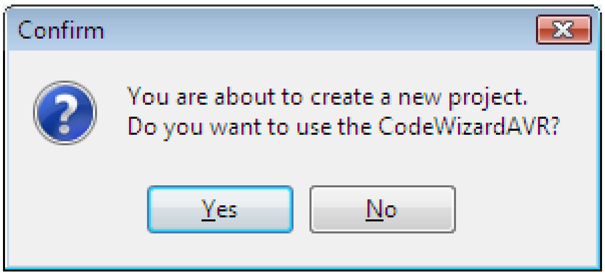
## 3.3 Working with Projects

The Project groups the source file(s) and compiler settings that you use for building a particular program.

### 3.3.1 Creating a New Project

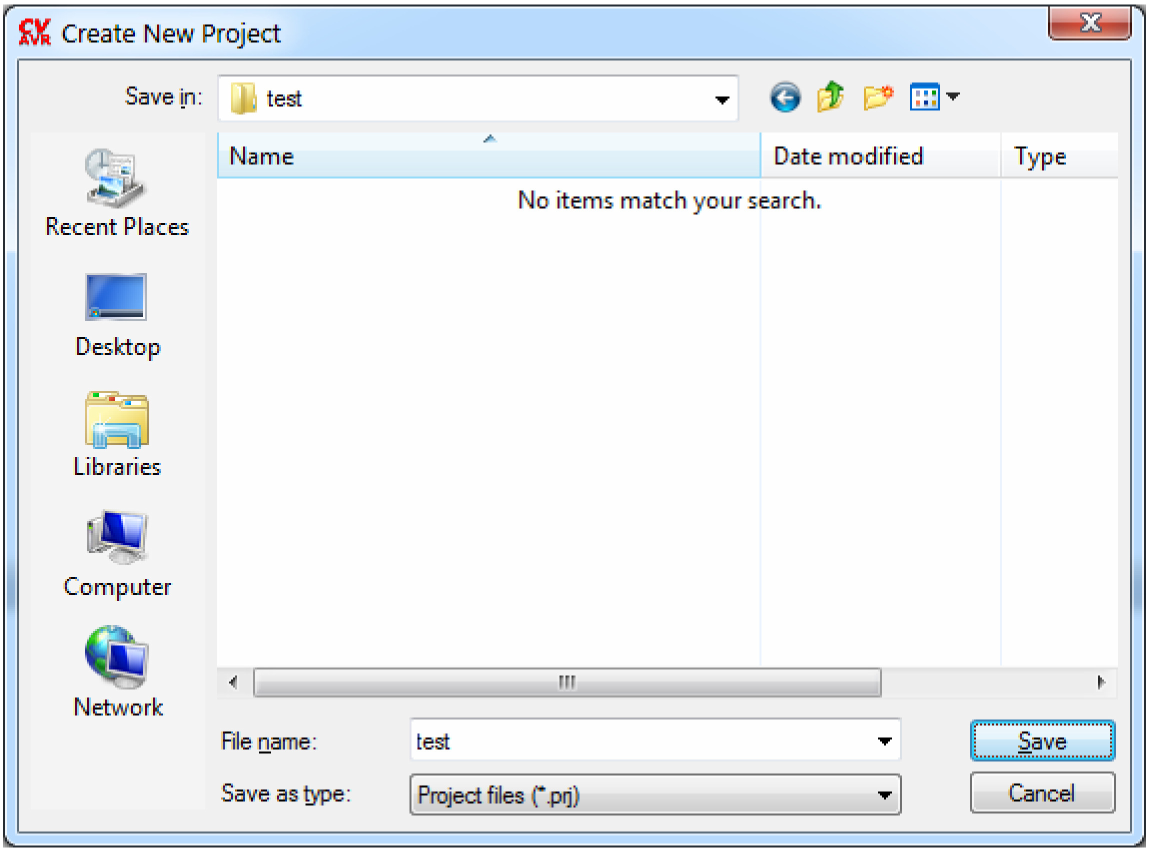
You can create a new project file using the **File|New|Project** menu command, by pressing the and buttons on the toolbar.

A dialog will open asking you to confirm if you would like to use the CodeWizardAVR to create the new project.



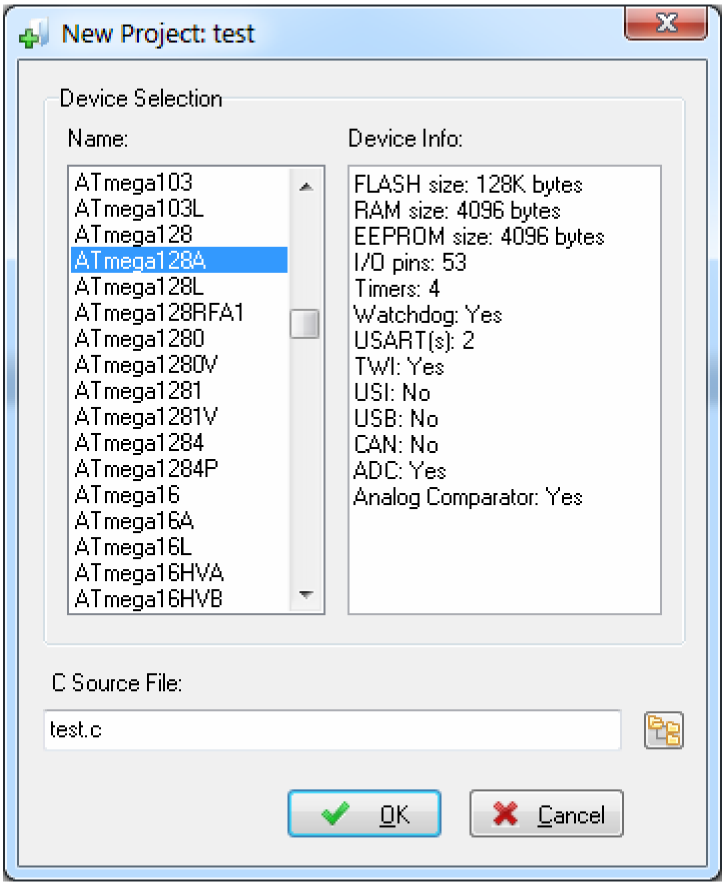

If you select **No** then the **Create New Project** dialog window will open.

You must specify the new Project file name and its location.



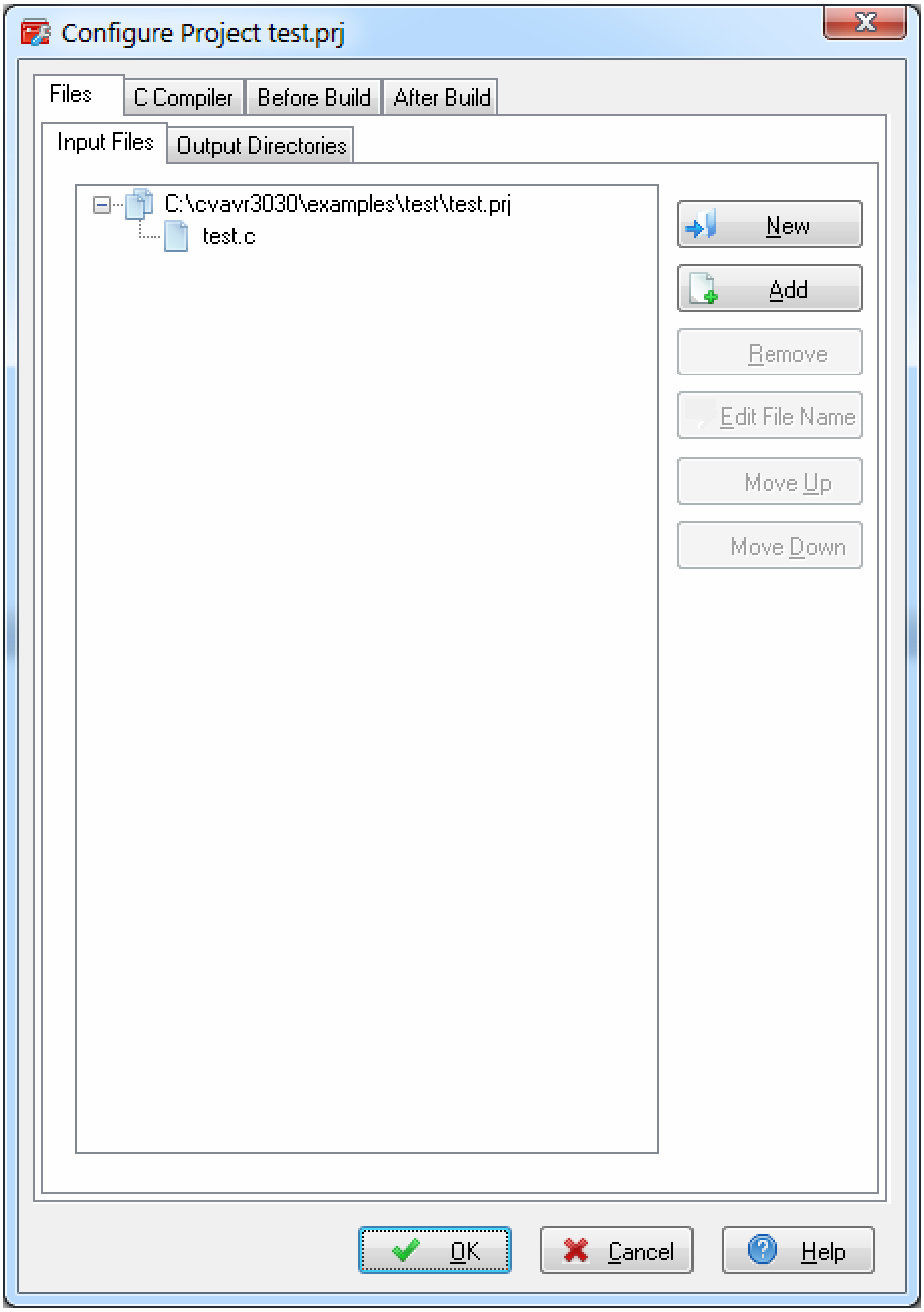

The Project file will have the .prj extension.

After the name and location of the project file was specified, a device selection dialog will open:




The dialog also allows to specify the name of the first C source file of the project.

Once the OK button is pressed the project and the C source file are created and the project configuration window is displayed:

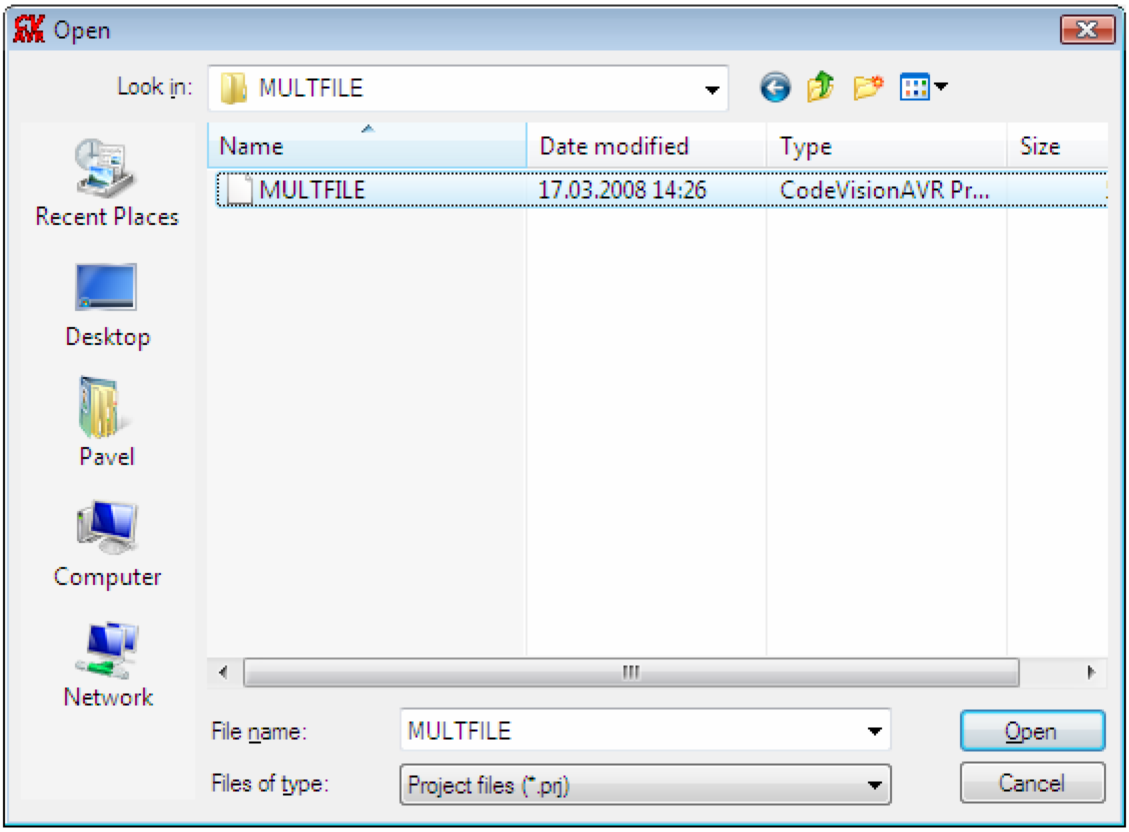



The Project configuration can be later modified by using the **Project|Configure** menu command or by pressing the  toolbar button.

**Note:** When a **.prj** project for the CodeVisionAVR IDE is created, a corresponding **.cproj** project file for Atmel Studio will be created too.
This allows editing/compiling the same project in both Atmel Studio and CodeVisionAVR IDE.

## 3.3.2 Opening an Existing Project

An existing Project file can be opened using the **File|Open** menu command or by pressing the button on the toolbar.
An **Open** dialog window appears.

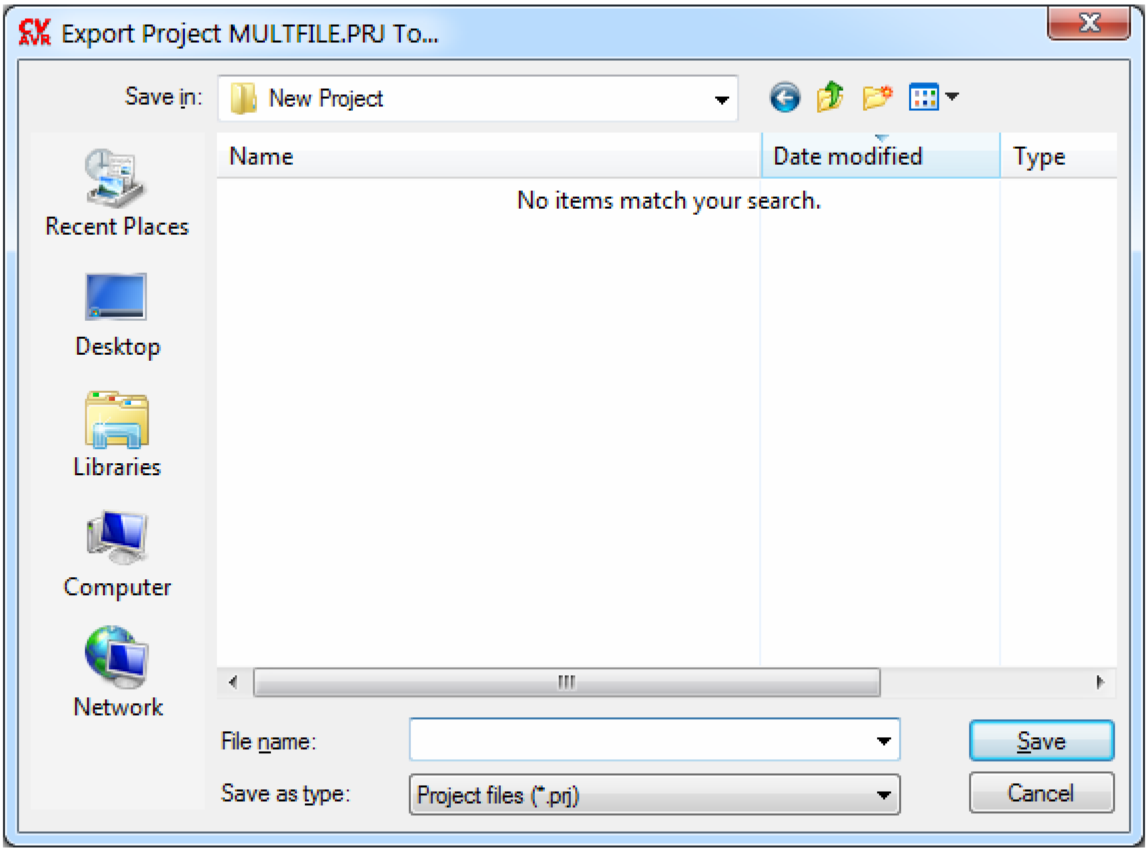



You must select the file name of the Project you wish to open.
By pressing the **Open** button you will open the Project file and its source file(s).
You can later configure the Project by using the **Project|Configure** menu command or by pressing the toolbar button.

## 3.3.3 Exporting a Project

The settings of the currently opened project can be exported to a new one by using the **Project|Export to a New CodeVisionAVR Project** menu command or the toolbar button.

Upon execution of this command a Project Export dialog window will open:

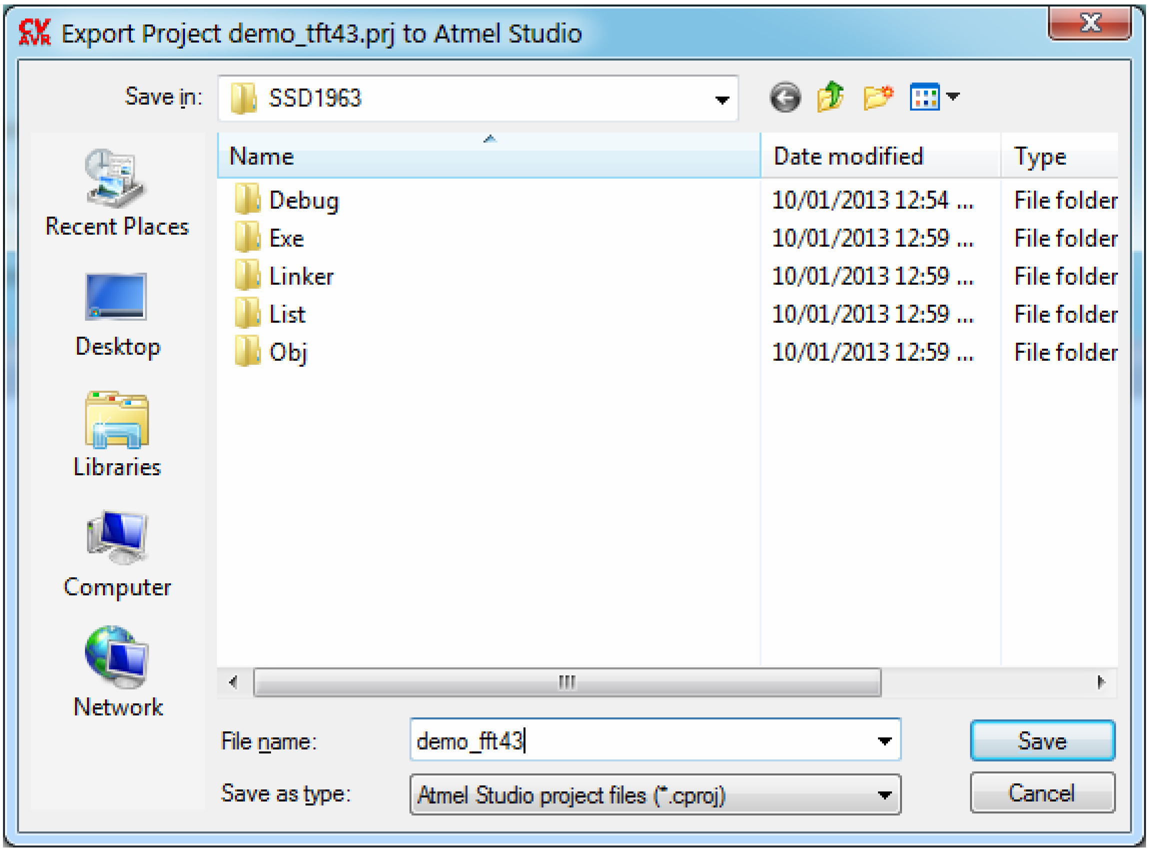



allowing to specify the name of the new project to which all the settings of the current project will be exported.

## 3.3.4 Exporting a Project to the CodeVisionAVR Extension for Atmel Studio

The currently opened project can be exported to a project file for Atmel Studio 6.1 or later, by using the **Project|Export to a CodeVisionAVR Extension for Atmel Studio** menu command or the toolbar button.

Upon execution of this command a Project Export dialog window will open:

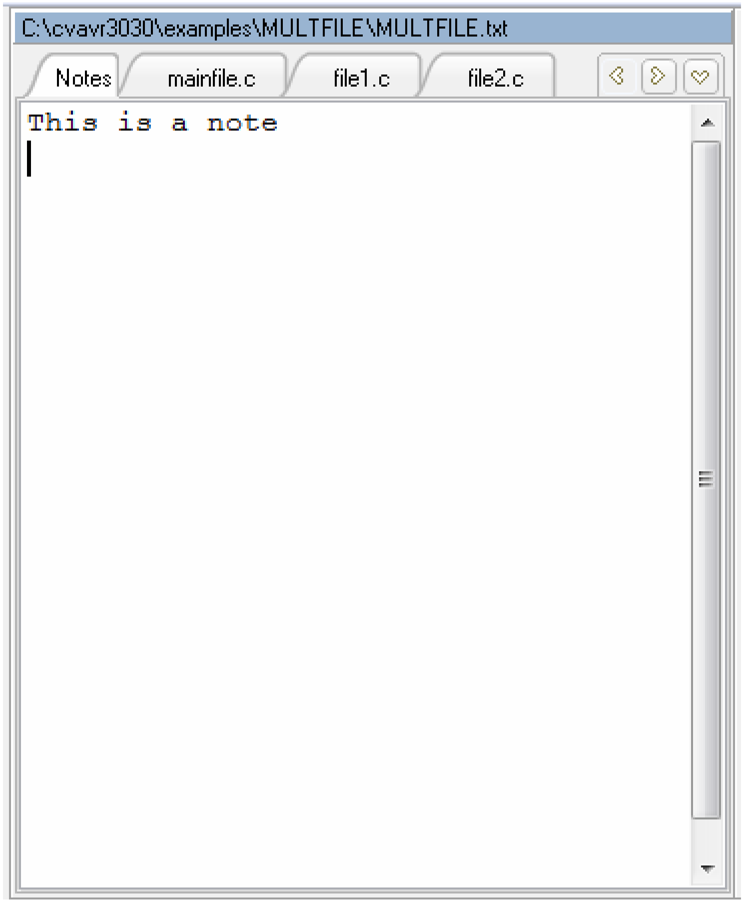



It will allow the creation of an Atmel Studio *.cproj* project file associated with the CodeVisionAVR project.
Once this file is opened in Atmel Studio, it will allow the project to be built using the CodeVisionAVR extension.

## 3.3.5 Adding Notes or Comments to the Project

With every Project the CodeVisionAVR IDE creates an associated text file where you can place notes and comments.
You can access this file using the **Project|Notes** menu command or the toolbar button.



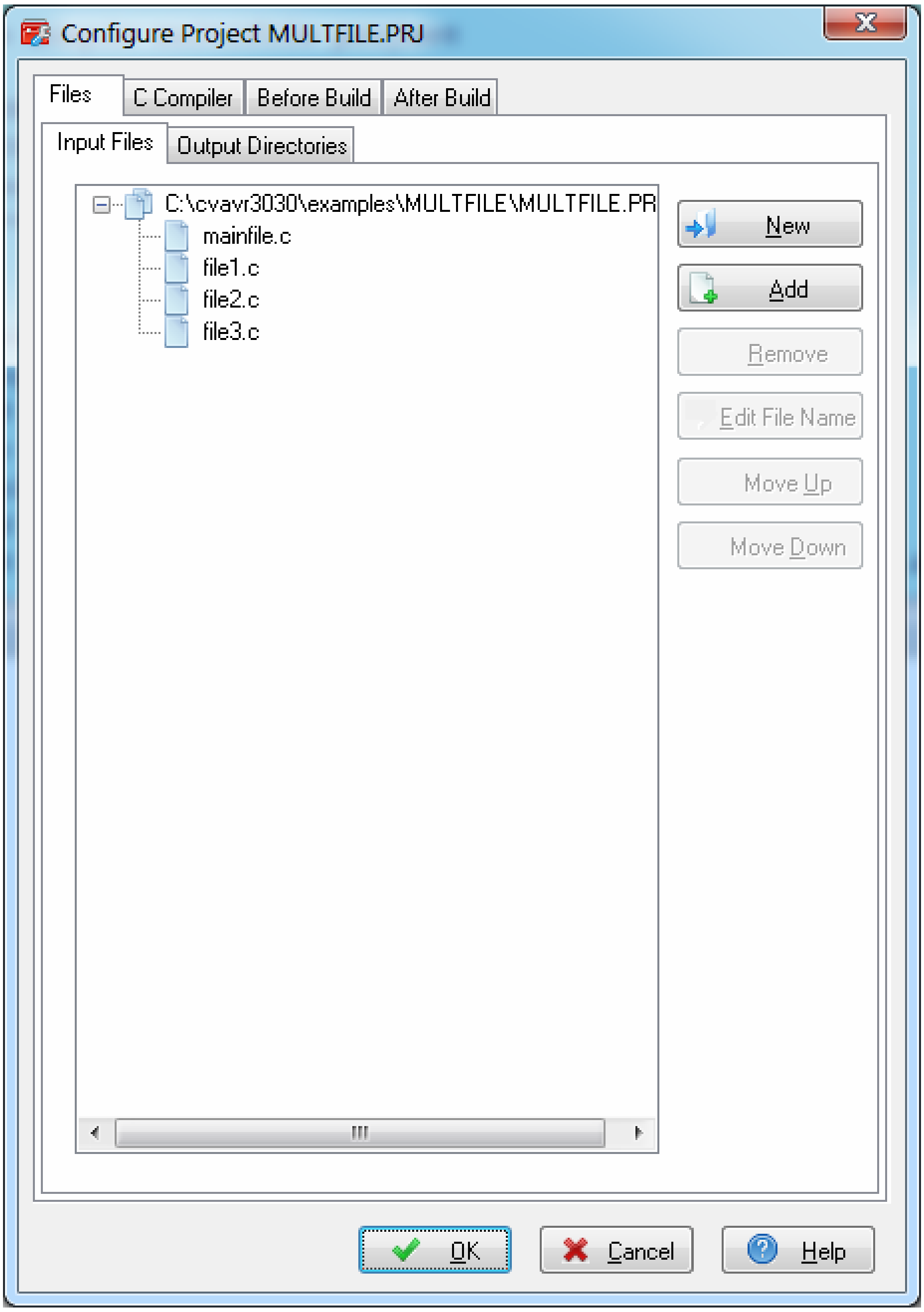

This file can be edited using the standard Editor commands.
The file is automatically saved when you **Close** the Project or **Quit** the CodeVisionAVR program.

## 3.3.6 Configuring the Project

The Project can be configured using the **Project|Configure** menu command or the toolbar button.

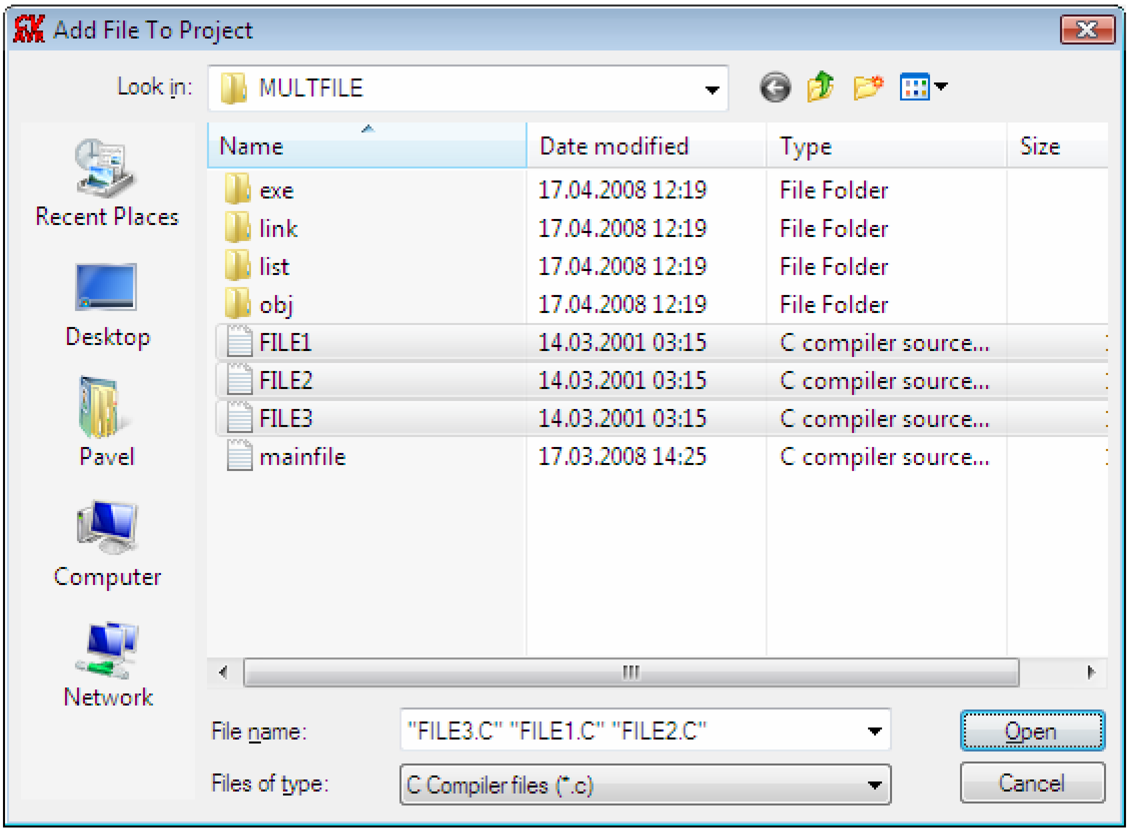
### 3.3.6.1 Adding or Removing a File from the Project

To add or remove a file from the currently opened project you must use the **Project|Configure** menu command or the toolbar button.
A **Configure Project** tabbed dialog window will open.
You must select the **Files** and **Input Files** tabs.




By pressing the **New** button you can create a new .c source file and add it to the project.
The **Add** button allows adding an existing source file to the project.

Multiple files can be added by holding the Ctrl key when selecting in the **Add File to Project** dialog.




When the project is **Open**-ed all project files will be opened in the editor.
By clicking on a file, and then pressing the **Remove** button, you will remove this file from the project.

The project's file compilation order can be changed by clicking on a file and moving it up, respectively down, using the **Move Up**, respectively **Move Down**, buttons.

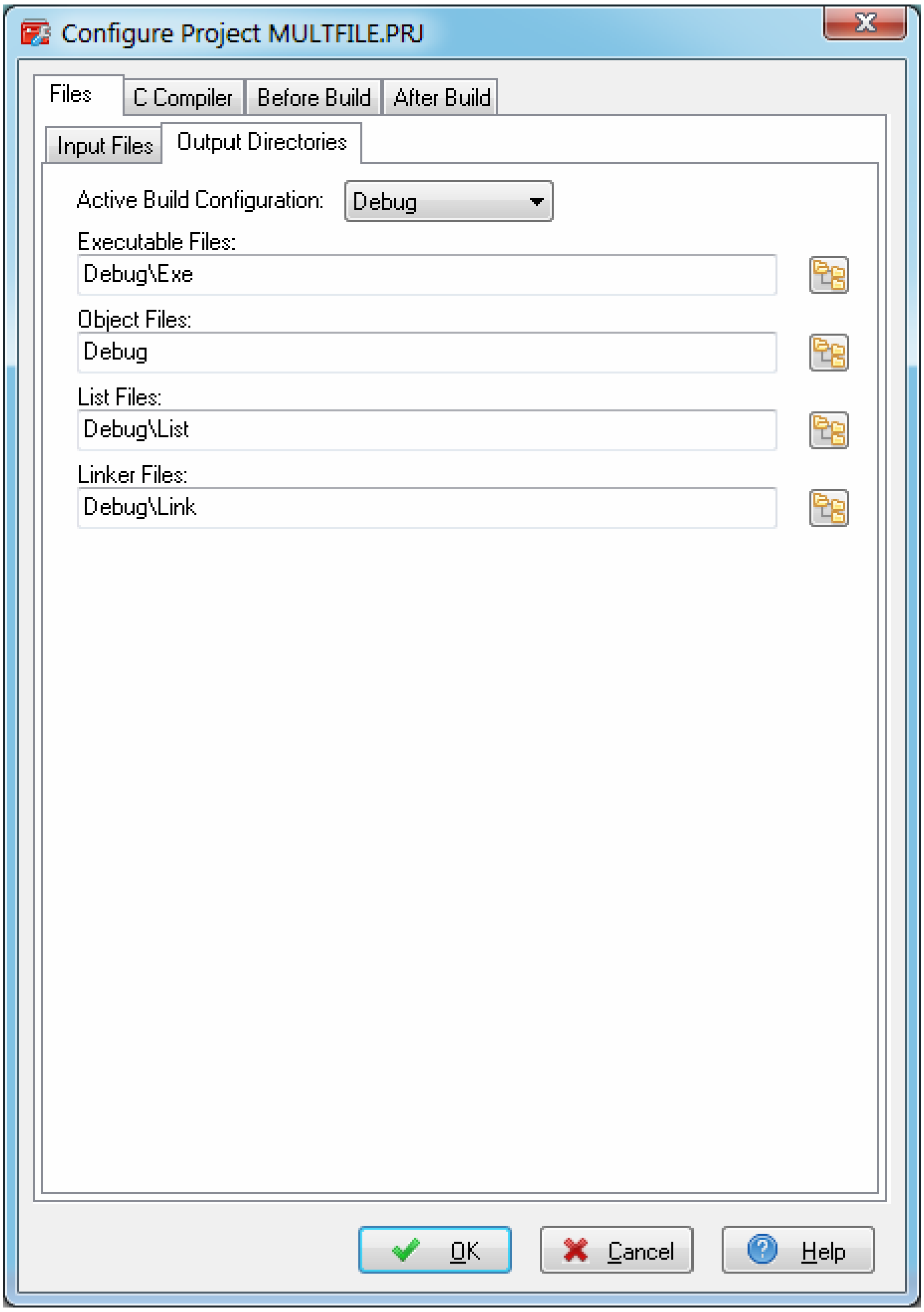
Changes can be saved, respectively canceled, using the **OK**, respectively **Cancel** buttons.

When creating a project with multiple files the following rules must be preserved:

- only .C files must be added to the project's Files list
- there's no need to #include the .C files from the Files list as they will be automatically linked
- data type definitions and function declarations must be placed in header .H files, that will be #include -d as necessary in the .C files
- global variables declarations must be placed in the .C files where necessary
- there's no need to declare global variables, that are not static, in header .H files, because if these files will be #include -d more than once, the compiler will issue errors about variable redeclarations.

## 3.3.6.2 Setting the Project Output Directories

Selecting the **Output Directories** tab allows the user to specify distinct directories where will be placed the files resulted after the compilation and linking.

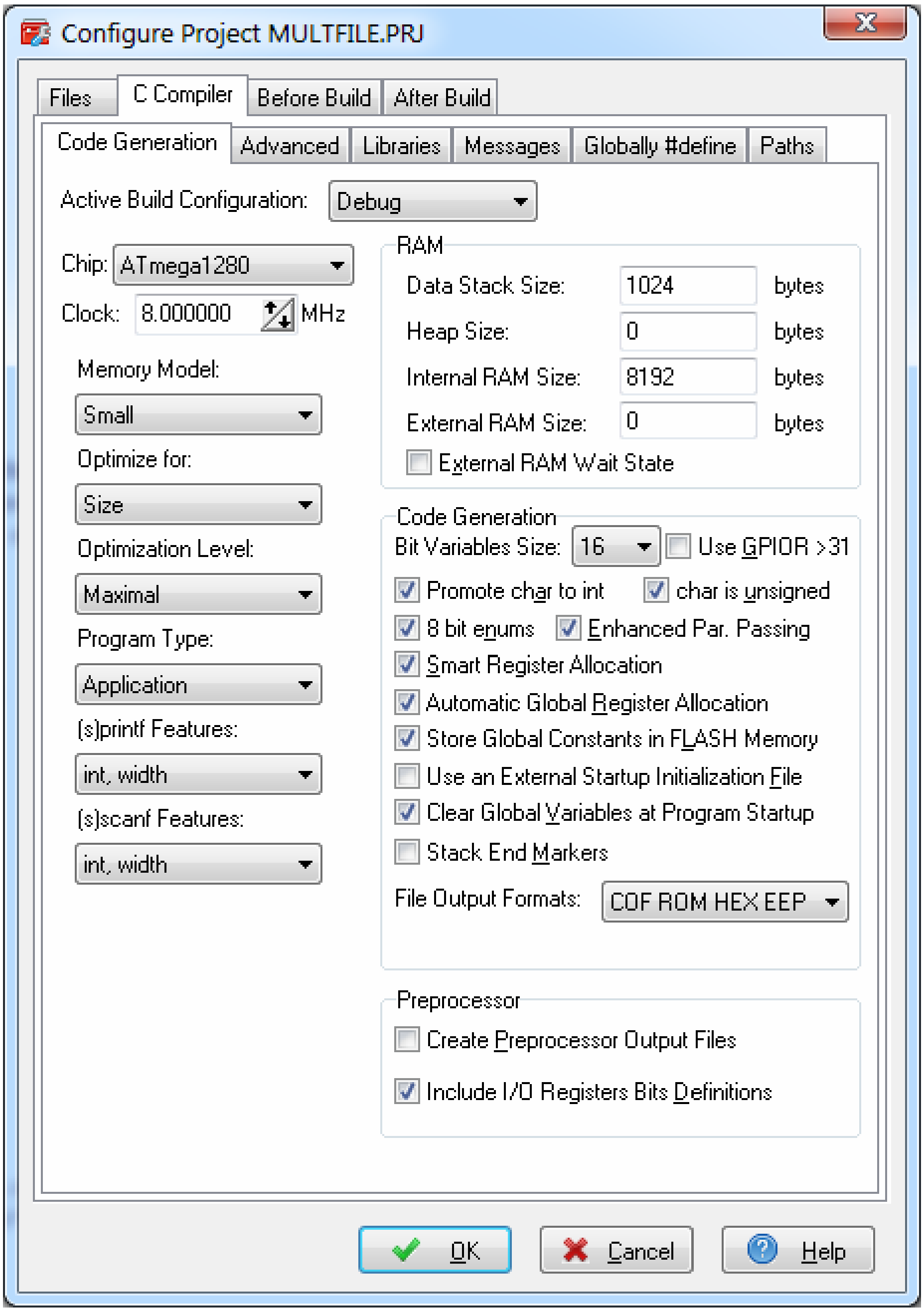



Pressing the button allows to select an existing directory.

The **.rom**, **.hex** and **.eep** files resulted after the **Build** process will be placed in the **Executable Files** directory.
The object files resulted after the **Compile** process will be placed in the **Object Files** directory.
The **.cof** COFF object file that results after the **Build** process will be also placed in the **Object Files** directory.
The **.asm**, **.lst** and **.map** files created during the **Build** process will be placed in the **List Files** directory.
Various files created by the linker during the **Build** process will be placed in the **Linker Files** directory.

### 3.3.6.3 Setting the C Compiler Options

To set the C compiler options for the currently opened project you must use the **Project|Configure** menu command or the toolbar button.
A **Configure Project** tabbed dialog window will open. You must select the **C Compiler** and **Code Generation** tabs.
CodeVisionAVR allows to specify two separate Build Configurations: **Debug** and **Release**, with different **Code Generation**, **Warning Messages** and **Globaly #define** options.
The **Active Build Configuration** selects which one of the above options will be used when a **Project|Build** or **Project|Build All** will be performed.




You can select the target AVR microcontroller chip by using the **Chip** combo box.
You must also specify the CPU **Clock** frequency in MHz, which is needed by the Delay Functions, 1 Wire Protocol Functions and Maxim/Dallas Semiconductor DS1820/DS18S20/DS18B20 Temperature Sensors Functions.

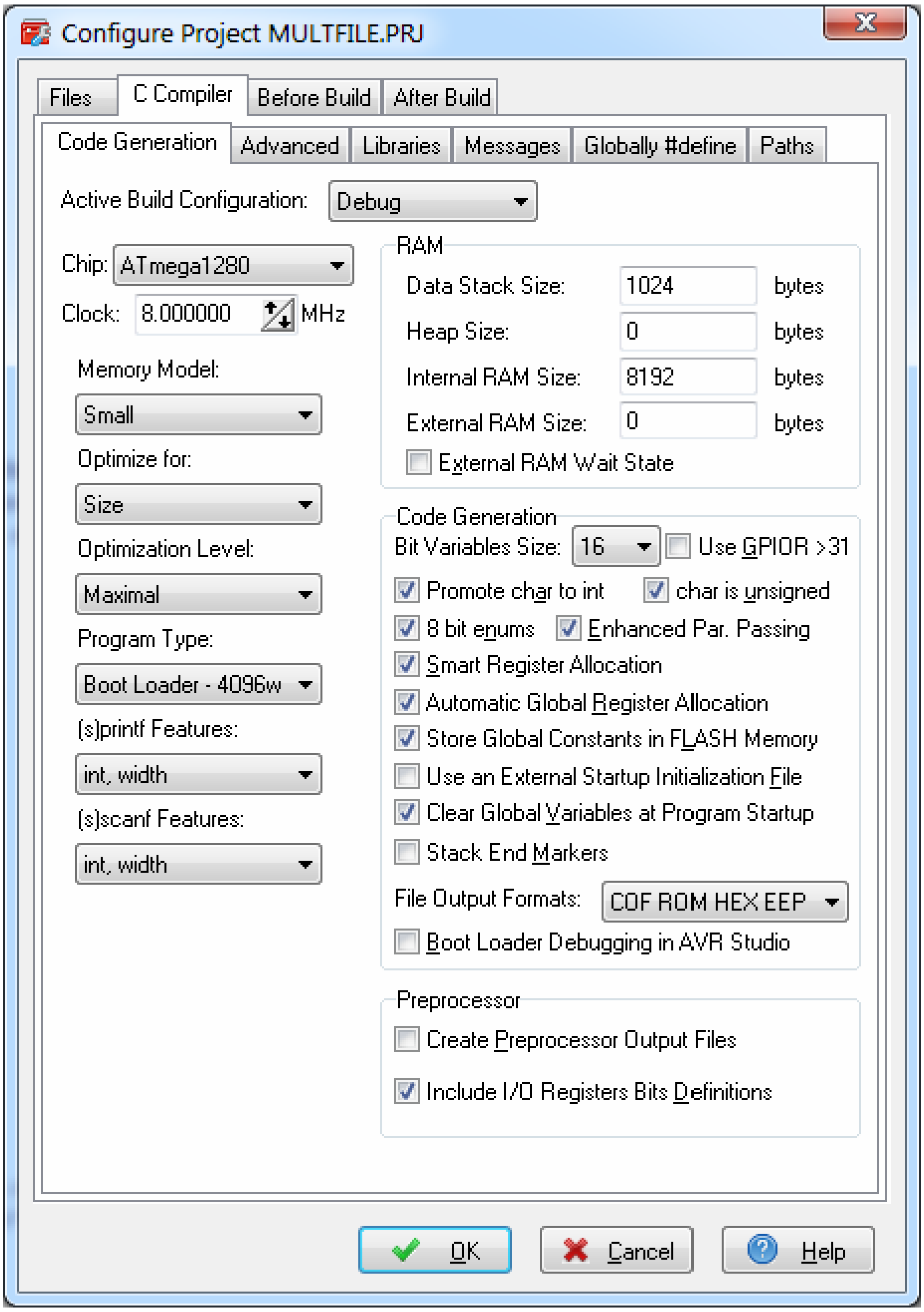
The required memory model can be selected by using the **Memory Model** list box.

The compiled program can be optimized for minimum size, respectively maximum execution speed, using the **Optimize for|Size**, respectively **Optimize for|Speed**, settings.

The amount of code optimization can be specified using the **Optimization Level** setting.
The *Maximal* optimization level may make difficult the code debugging with AVR Studio.

For devices that allow self-programming the **Program Type** can be selected as:
- **Application**
- **Boot Loader**

If the **Boot Loader** program type was selected, a supplementary **Boot Loader Debugging in AVR Studio** option is available.



If this option is enabled, the compiler will generate supplementary code that allows the Boot Loader to be source level debugged in the AVR Studio simulator/emulator.

<u>When programming the chip with the final Boot Loader code, the Boot Loader Debugging option must be disabled.</u>

For reduced core chips like ATtiny10, there is an additional option: **Enable auto Var. Watch in AVR Studio**.
If this option is enabled, the compiler will generate additional code that allows local automatic variables, saved in the Data Stack, to be watched in AVR Studio 4.18 SP2 or later.
After finishing debugging the program, this option should be disabled and the project rebuild.
This will allow to reduce the size of the program and increase its execution speed.

The **(s)printf features** option allows to select which versions of the printf and sprintf **Standard C Input/Oputput Functions** will be linked in your project:

- **int** - the following conversion type characters are supported: 'c', 's', 'p', 'i', 'd', 'u', 'x', 'X', '%', no width or precision specifiers are supported, only the '+' and ' ' flags are supported, no input size modifiers are supported
- **int, width** - the following conversion type characters are supported: 'c', 's', 'p', 'i', 'd', 'u', 'x', 'X', '%', the width specifier is supported, the precision specifier is not supported, only the '+', '-', '0' and ' ' flags are supported, no input size modifiers are supported
- **long, width** - the following conversion type characters are supported: 'c', 's', 'p', 'i', 'd', 'u', 'x', 'X', '%' the width specifier is supported, the precision specifier is not supported, only the '+', '-', '0' and ' ' flags are supported, only the 'l' input size modifier is supported
- **long, width, precision** - the following conversion type characters are supported: 'c', 's', 'p', 'i', 'd', 'u', 'x', 'X', '%', the width and precision specifiers are supported, only the '+', '-', '0' and ' ' flags are supported, only the 'l' input size modifier is supported
- **float, width, precision** - the following conversion type characters are supported: 'c', 's', 'p', 'i', 'd', 'u', 'e', 'E', 'f', 'x', 'X', '%', the width and precision specifiers are supported, only the '+', '-', '0' and ' ' flags are supported, only the 'l' input size modifier is supported.

The more features are selected, the larger is the code size generated for the printf and sprintf functions.

The **(s)scanf features** option allows to select which versions of the scanf and sscanf **Standard C Input/Oputput Functions** will be linked in your project:

- **int, width** - the following conversion type characters are supported: 'c', 's', 'i', 'd', 'u', 'x', '%', the width specifier is supported, no input size modifiers are supported
- **long, width** - the following conversion type characters are supported: 'c', 's', 'i', 'd', 'u', 'x', '%' the width specifier is supported, only the 'l' input size modifier is supported.

The more features are selected, the larger is the code size generated for the scanf and sscanf functions.

The **Data Stack Size** must be also specified.

If the dynamic memory allocation functions from the Standard Library are to be used, the **Heap Size** must be also specified.
It can be calculated using the following formulae:

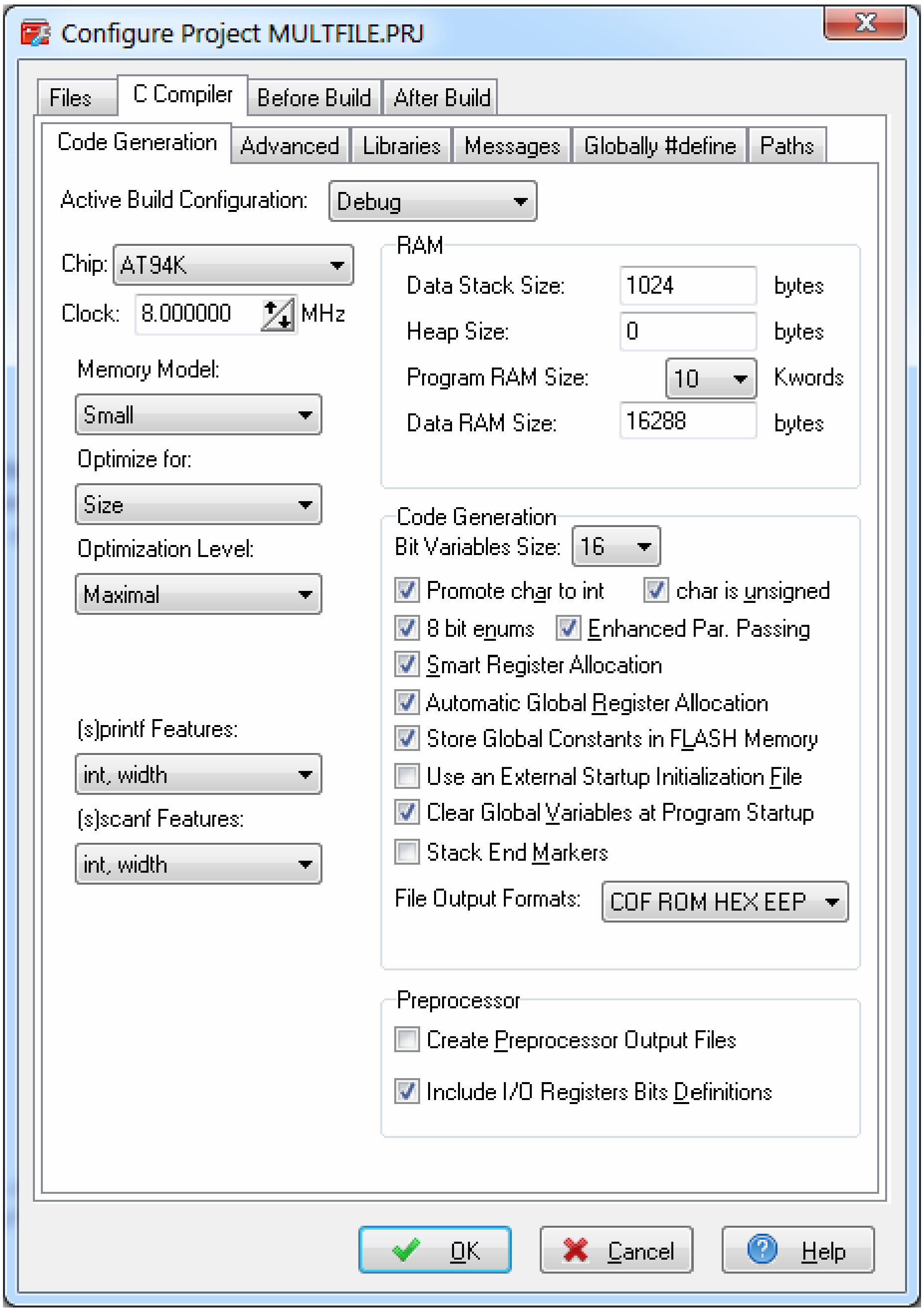
$$heap\_size = (n+1)\cdot 4 + \sum_{i=1}^{n} block\_size_i$$

where: $n$ is the number of memory blocks that will be allocated in the heap
$block\_size_i$ is the size of the memory block $i$

If the memory allocation functions will not be used, then the **Heap Size** must be specified as zero.
Eventually you may also specify the **External RAM Size** (in case the microcontroller have external SRAM memory connected).

The **External RAM Wait State** option enables the insertion of wait states during access to the external RAM. This is useful when using slow memory devices.

If an Atmel AT94K05, AT94K10, AT94K20 or AT94K40 FPSLIC device will be used, than there will be the possibility to specify the **Program RAM Size** in Kwords.




The maximum size of the global bit variables, which are placed in the GPIOR (if present) and registers R2 to R14, can be specified using the **Bit Variables Size** list box.

The **Use GPIOR >31** option, when checked, allows using GPIOR located at addresses above 31 for global bit variables.
Note that bit variables located in GPIOR above address 31 are accessed using the IN, OUT, OR , AND instructions, which leads to larger and slower code than for bit variables located in GPIOR with the address range 0…31, which use the SBI, CBI instructions. Also the access to bit variables located in GPIOR above address 31 is not atomic.

Therefore it is recommended to leave the **Use GPIOR >31** option not checked if the number of global bit variables is small enough and no additional registers are needed for their storage.

Checking the **Promote char to int** check box enables the ANSI promotion of **char** operands to **int**.
This option can also be specified using the **#pragma promotechar** compiler directive.
Promoting **char** to **int** leads to increases code size and lowers speed for an 8 bit chip microcontroller like the AVR.
In order to assure code compatibility with other C compilers, the **Promote char to int** option is enabled by default for newly created projects.

If the **char is unsigned** check box is checked, the compiler treats by default the **char** data type as an unsigned 8 bit in the range 0…255.
If the check box is not checked the **char** data type is by default a signed 8 bit in the range –128…127.
This option can also be specified using the **#pragma uchar** compiler directive.
Treating **char** as unsigned leads to better code size and speed.

If the **8 bit enums** check box is checked, the compiler treats the enumerations as being of 8 bit **char** data type, leading to improved code size and execution speed of the compiled program. If the check box is not checked the enumerations are considered as 16 bit **int** data type as required by ANSI.

The **Enhanced Par. Passing** check box allows enabling or disabling the generation of specific code for passing the last function parameter, with a size of up to 4 bytes, in the registers R26, R27, R24 and R25. Enabling this option allows a code size reduction of 3-4% for large projects. It can be disabled in case of incompatibility with functions written fully in assembler for older version of the compiler.

The **Smart Register Allocation** check box enables allocation of registers R2 to R14 (not used for bit variables) and R16 to R21 in such a way that 16bit variables will be preferably located in even register pairs, thus favouring the usage of the enhanced core MOVW instruction for their access. This option is effective only if the **Enhanced Instructions** check box is also checked.
If **Smart Register Allocation** is not enabled, the registers will be allocated in the order of variable declaration.
The **Smart Register Allocation** option should be disabled if the program was developed using CodeVisionAVR prior to V1.25.3 and it contains inline assembly code that accesses the variables located in registers R2 to R14 and R16 to R21.

The registers in the range R2 to R14, not used for bit variables, can be automatically allocated to **char** and **int** global variables and global pointers by checking the **Automatic Global Register Allocation** check box.

If the **Store Global Constants in FLASH Memory** check box is checked, the compiler will treat the **const** type qualifier as equivalent to the **flash** memory attribute and will place the constants in FLASH memory. If the option is not checked, constants marked with the **const** type qualifier will be stored in RAM memory and the ones marked with the **flash** memory attribute will be stored in FLASH memory.
The **Store Global Constants in FLASH Memory** option is, by default, not enabled for newly created projects.
In order to maintain compatibility with V1.xx projects, the **Store Global Constants in FLASH Memory** option must be checked.

An external startup.asm file can be used by checking the **Compilation|Use an External Startup File** check box.

The **Clear Global Variables at Program Startup** check box allows enabling or disabling the initialization with zero of global variables located in RAM and registers R2 to R14 at program startup after a chip reset. If an external startup.asm file is used, this option must signal to the compiler if the variable initialization with zero is performed in this file or not.

For debugging purposes you have the option **Stack End Markers**. If you select it, the compiler will place the strings **DSTACKEND**, respectively **HSTACKEND,** at the end of the **Data Stack**, respectively **Hardware Stack** areas.

When you debug the program with the AVR Studio debugger you may see if these strings are overwritten, and consequently modify the **Data Stack Size**.

When your program runs correctly you may disable the placement of these strings in order to reduce code size.

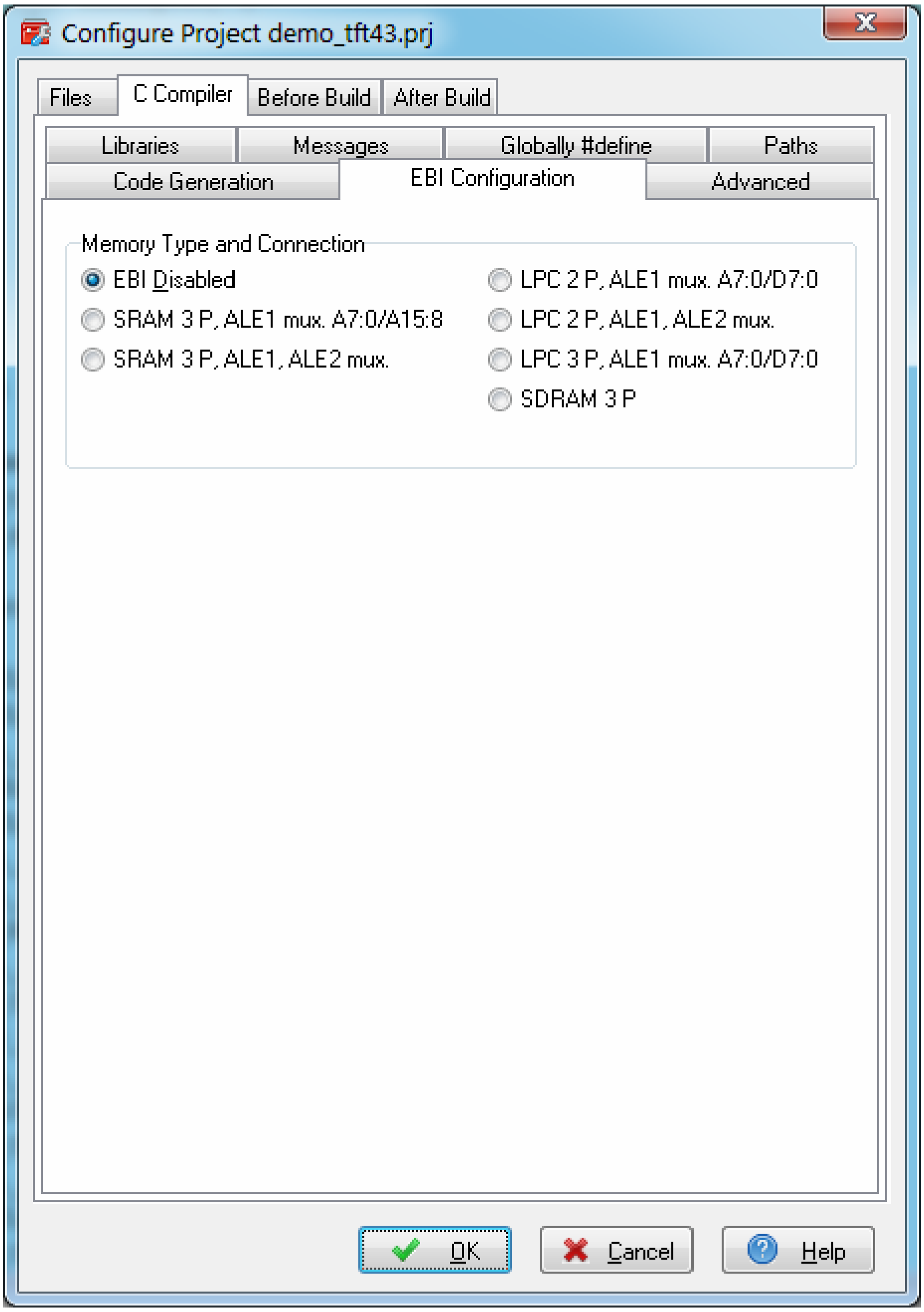
Using the **File Output Formats** list box you can select the following formats for the files generated by the compiler:

- COFF (required by the Atmel AVR Studio debugger), ROM, Intel HEX and EEP (required by the In-System Programmer) ;
- Atmel generic OBJ, ROM, Intel HEX and EEP (required by the In-System Programmer).

The following **Preprocessor** options can be set:

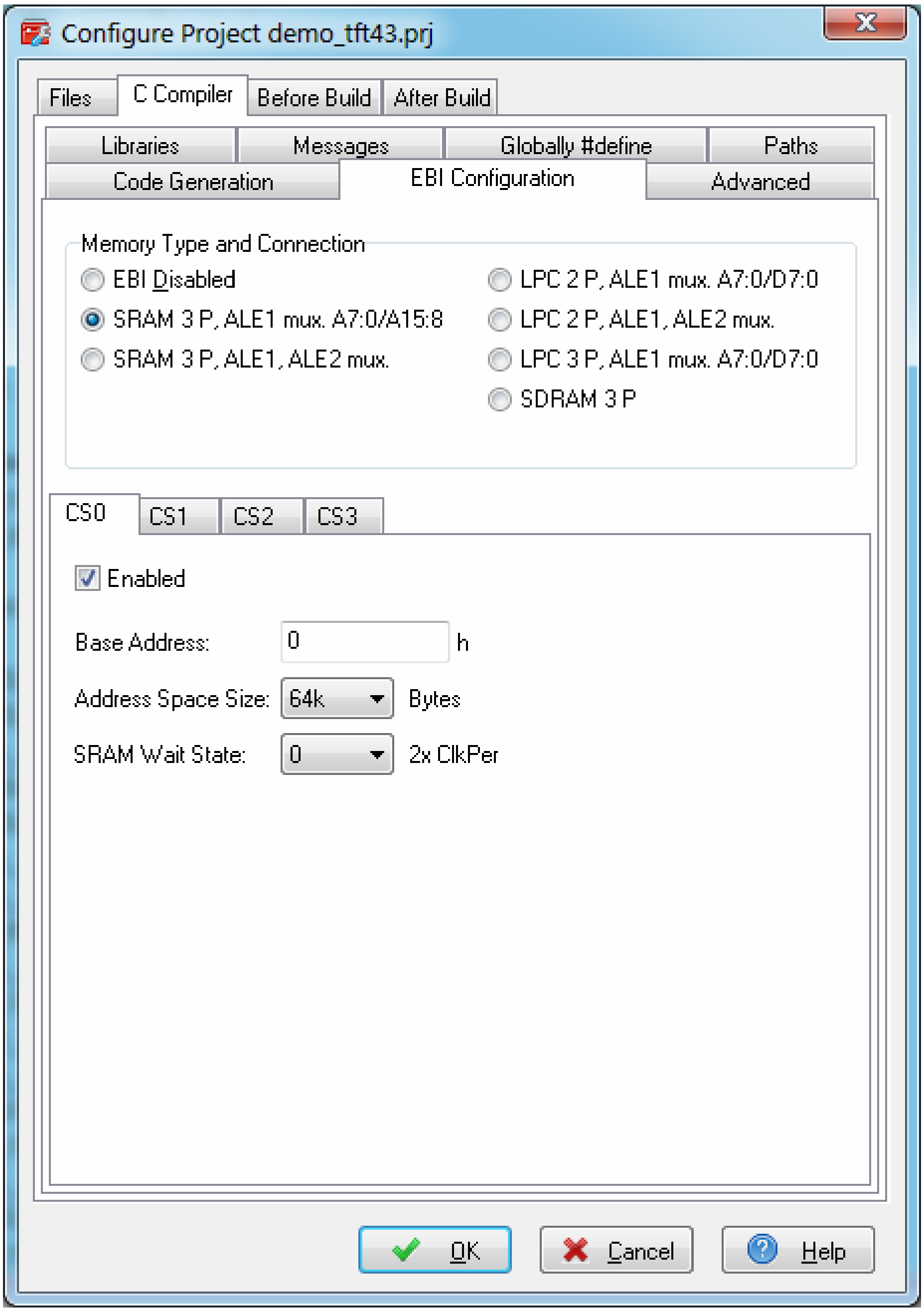
- **Create Preprocessor Output Files** - when enabled, an additional file with the **.i** extension will be created for each compiled source file. The preprocessor output files will contain the source files text will all the preprocessor macros expanded. Enabling this option will slow down the compilation process.
- **Include I/O Registers Bits Definitions** - will enable the I/O register bits definitions in the device header files.

For the XMEGA chips that feature an **External Bus Interface** (EBI) an additional **EBI Configuration** tab is present:



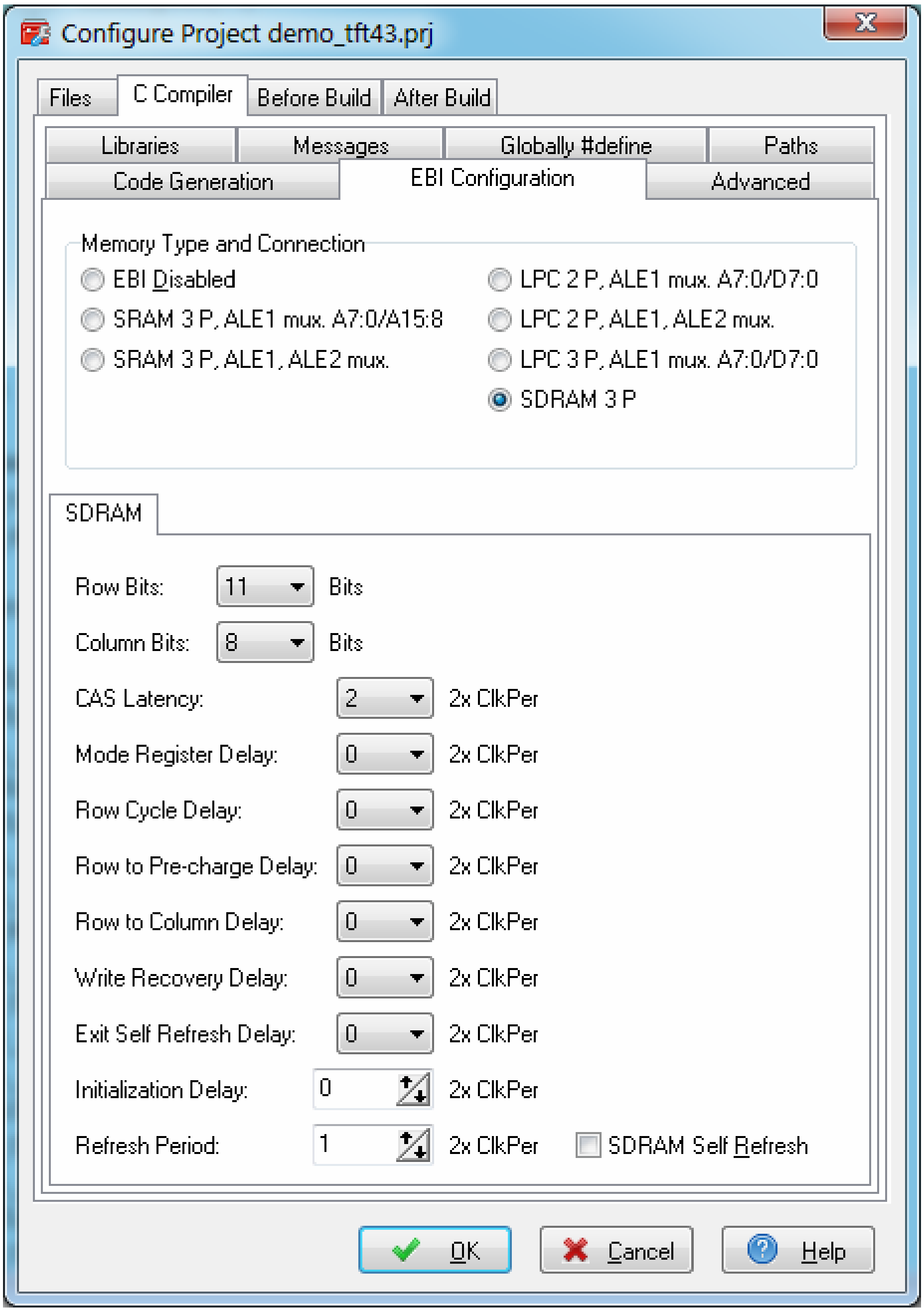

The check boxes from the **Memory Type and Connection** group allow to specify the **EBI** operating mode and kind of external RAM connected to the chip.

Depending on the **EBI** operating mode, additional tabs are displayed for the configuration of the **CS0...CS3** chip select signals:




The **Enable** check box activates the usage of the corresponding **CSn** chip select signal.
The **Base Address** represents the starting address, in hexadecimal, of the **Address Space** for which the chip select signal becomes active.
The **Address Space Size** list box allows to specify the address range size for which the chip select signal is active.
The **SRAM Wait State** list box allows inserting additional wait states when accessing slow external memory.

Specific options can be set if **SDRAM** chips are connected to the XMEGA chip:




These options are described in detail in Atmel's **XMEGA A Manual** in the **EBI - External Bus Interface** chapter.

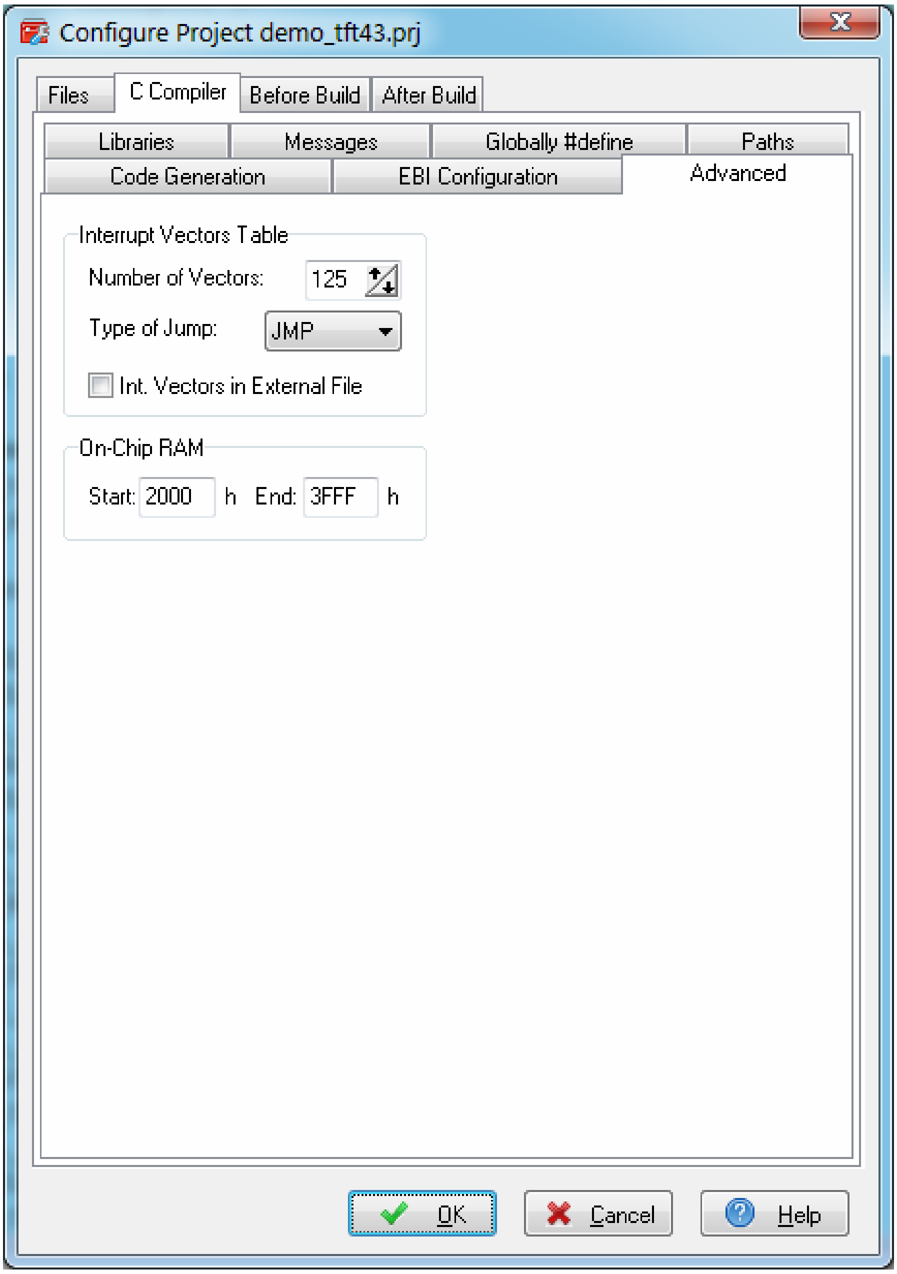
**Note:** All the necessary code for **EBI** setup will be automatically added by the compiler in the startup initialization that is executed immediately after the chip reset. There is no need for the programmer to write his own code for this purpose.
When **SDRAM** is used as external memory and a different clock source is used instead of the internal 2MHz oscillator, it is necessary to execute the function that configures the system clocks **before** the **EBI** setup sequence, which will ensure that correct timing is used for later **SDRAM** access by the startup code.

This can be achieved by using the **__reset** attribute applied to the clock initialization function:

```
__reset void system_clocks_init(void)
{
// Initialization code ...

}
```

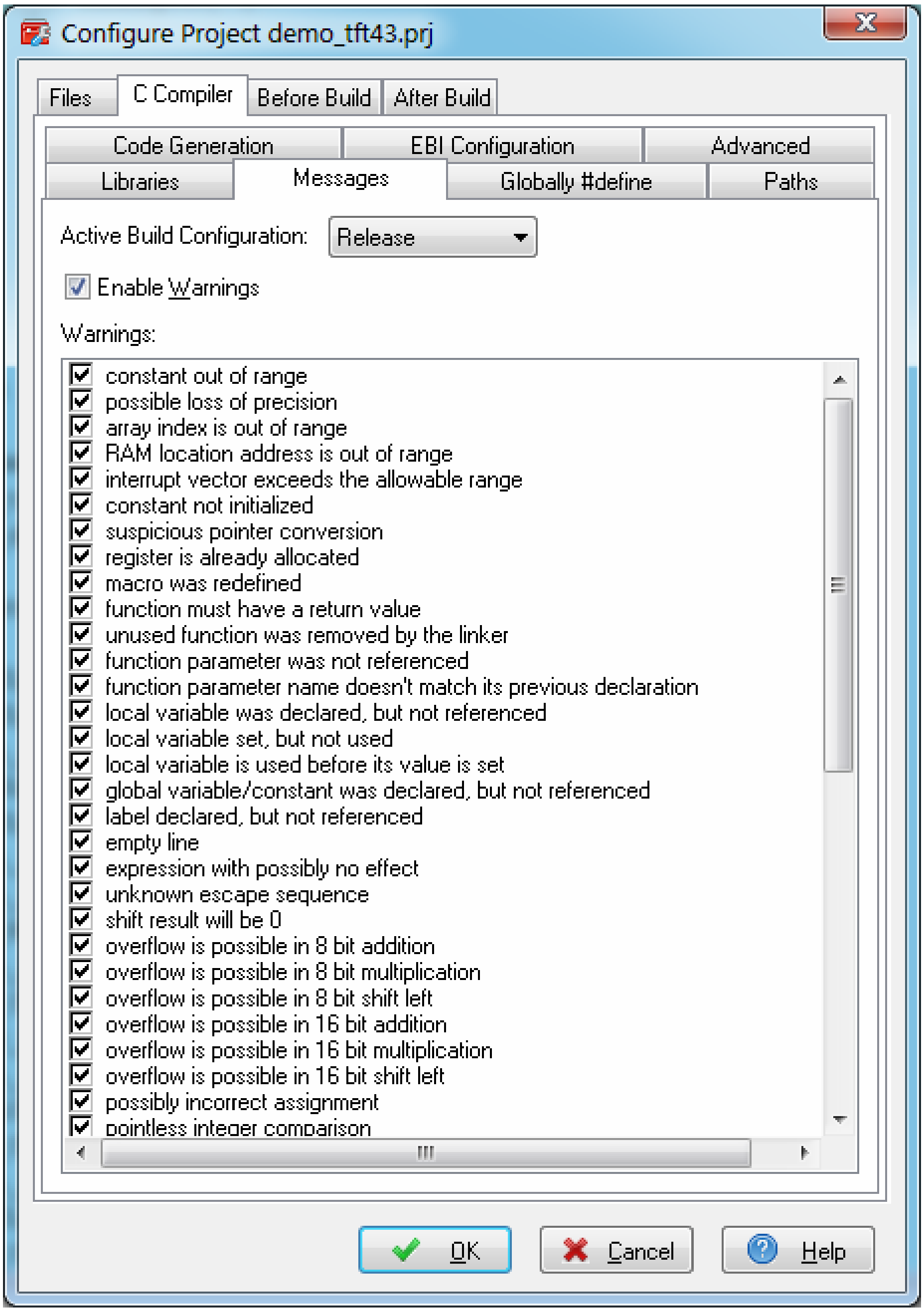
The code generated by the CodeWizardAVR for XMEGA chips, automatically handles such situations.

The **Advanced** tab, which is present only in the Advanced and Professional versions of the compiler, enables more detailed custom configuration like the number and jump type of the interrupt vectors and memory usage:



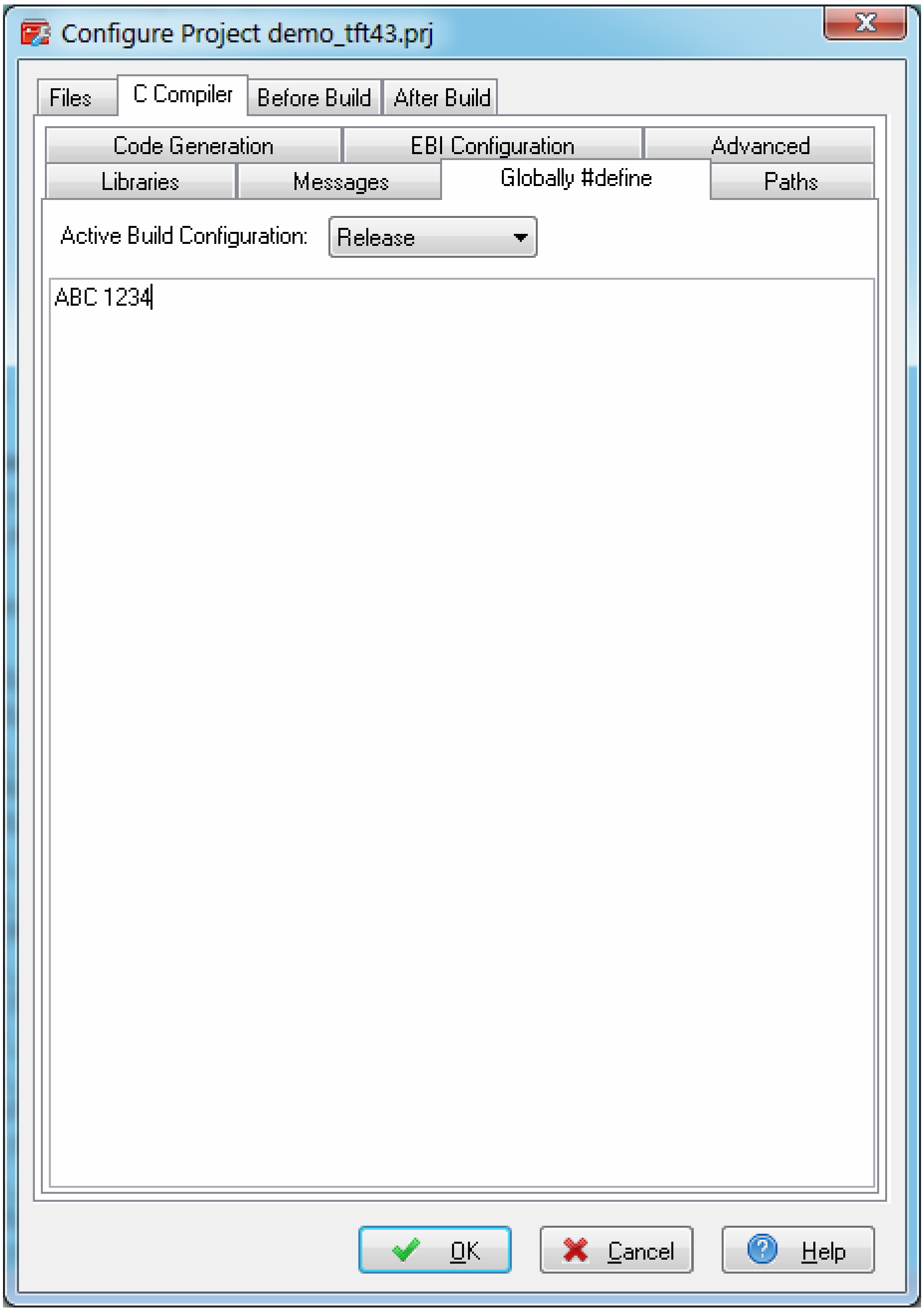

The **Int. Vectors in External File** option enables or disables placing the interrupt vectors in an external vectors.asm file created by the user. If this option is enabled the compiler will not generate any interrupt vectors by itself as the vectors will be present in the vectors.asm file.

The **Messages** tab allows to individually enable or disable various compiler and linker warnings:




The generation of warning messages during compilation can be globally enabled or disabled by using the **Enable Warnings** check box.

The **Globally #define** tab allows to #define macros that will be visible in all the project files.
For example:

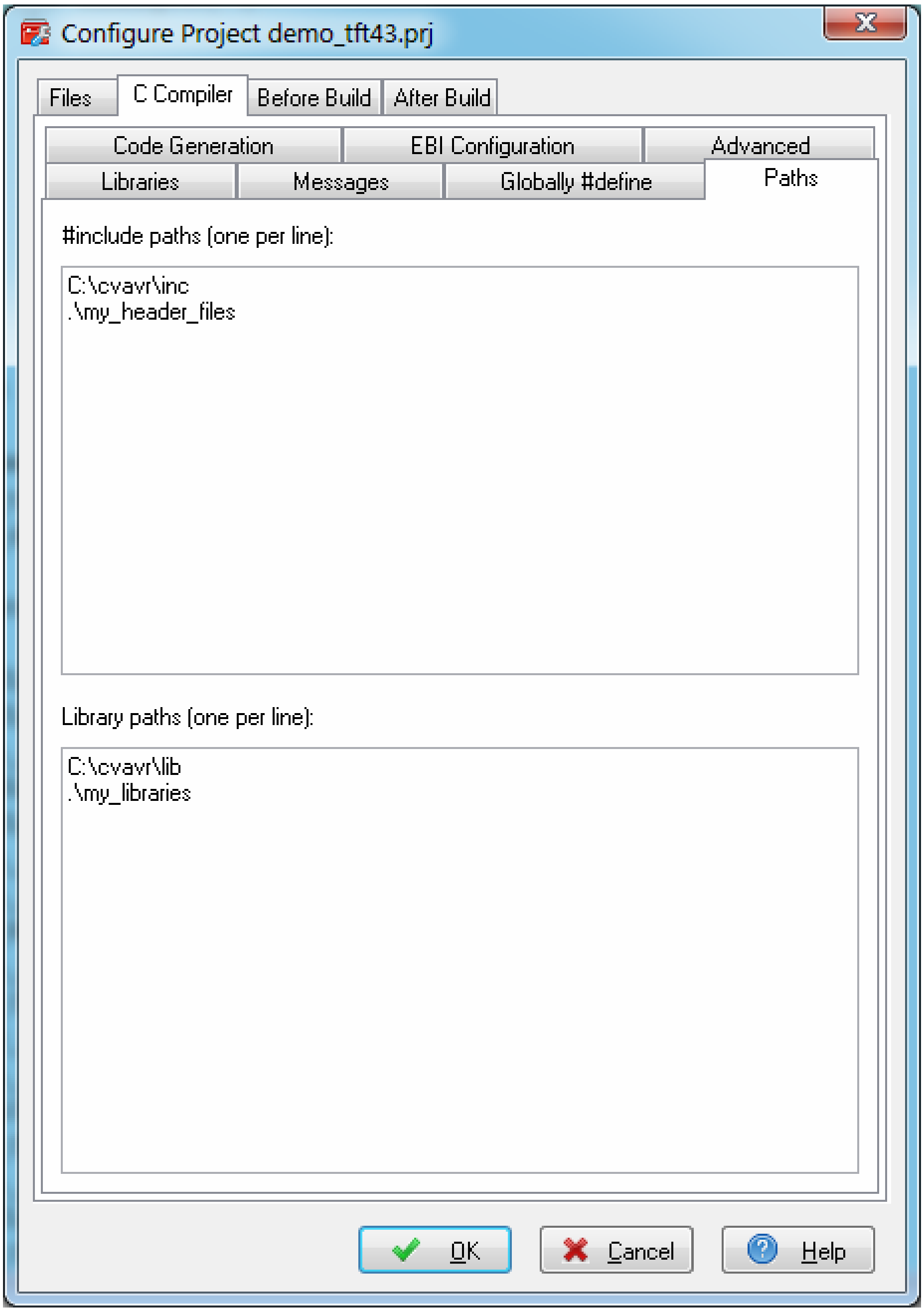



will be equivalent with placing the macro definition:

```
#define ABC 1234
```

in each project's program module.

The **Paths** tabs allows to specify additional paths for #include and library files.
These paths must be entered one per line in the appropriate edit controls.

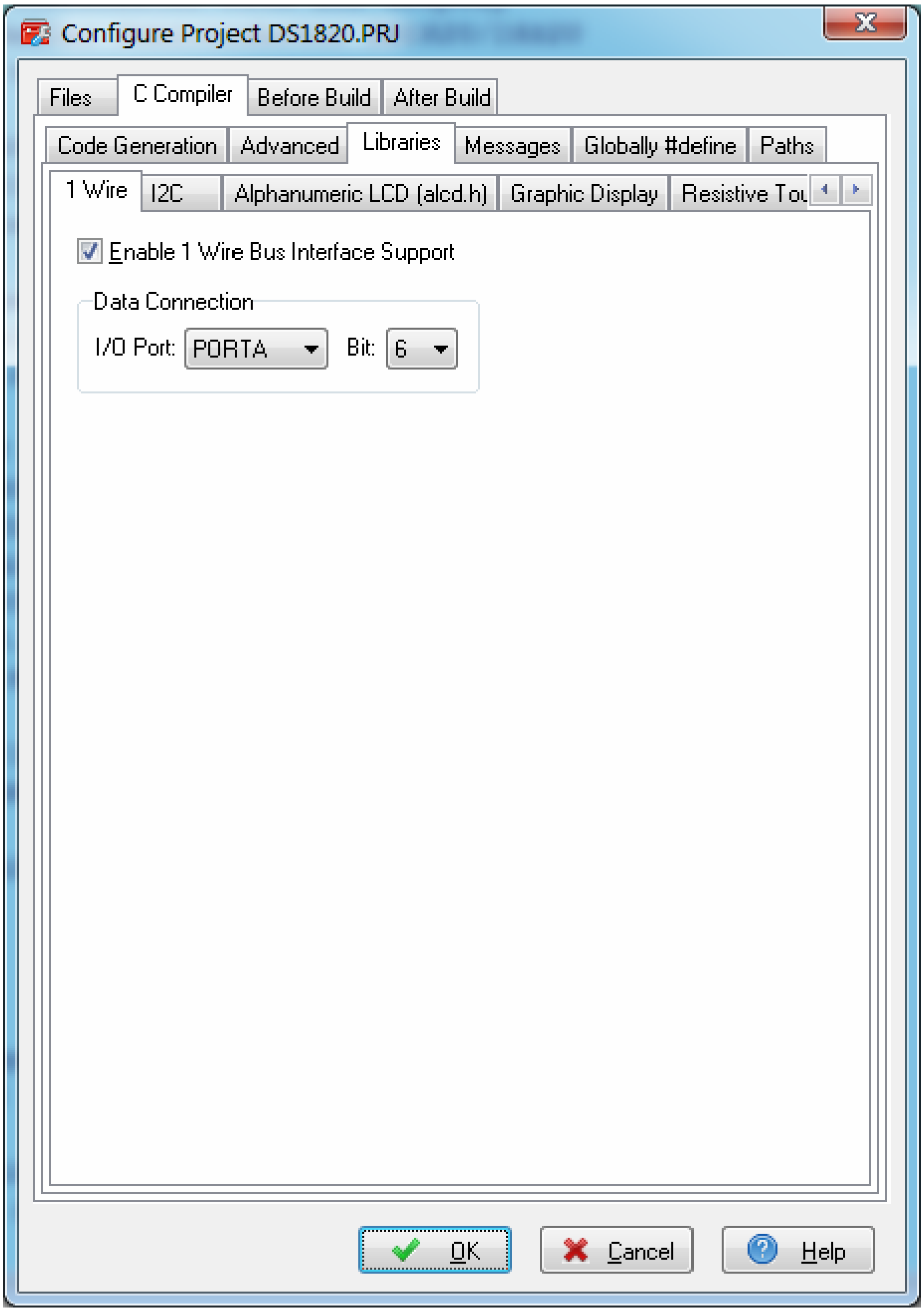



Changes can be saved, respectively canceled, using the **OK**, respectively **Cancel** buttons.

### 3.3.6.4 Setting the 1 Wire Library Options

The **Libraries** tab is used for configuring specific driver libraries used by the compiler.

The **1 Wire** tab is used for configuring the I/O port allocation for the 1 Wire Protocol Functions.



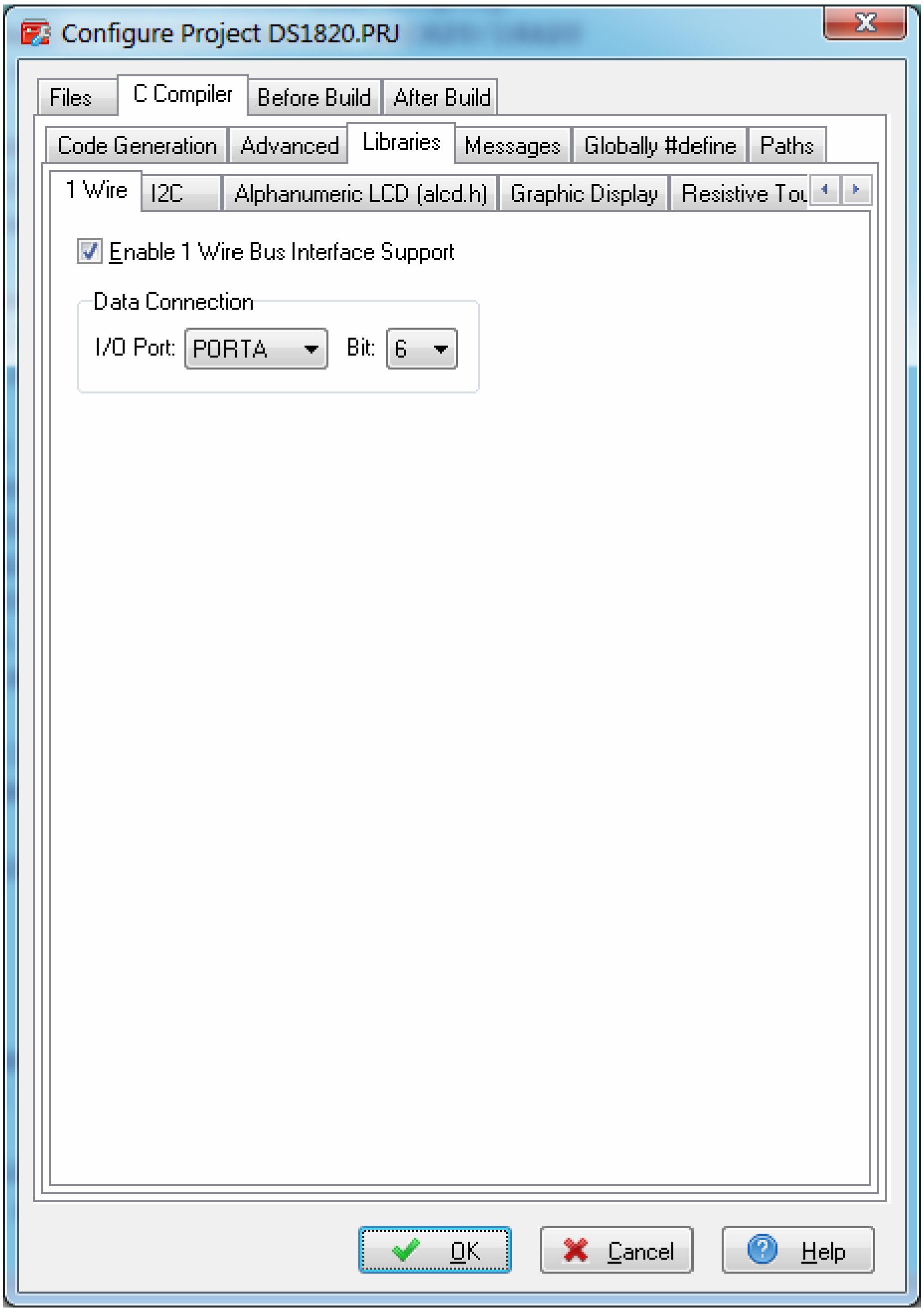

The following settings are available:

- **Enable 1 Wire Bus Interface Support** allows the activation of the 1 Wire Protocol Functions
- **I/O Port** and **Bit** specify in **Data Connection**, the port and bit used for 1 Wire bus communication.

## 3.3.6.5 Setting the I$^2$C Library Options

The **Libraries** tab is used for configuring specific driver libraries used by the compiler.

The **I2C** tab is used for configuring the I/O port allocation and bit rate of the software bit-banged I$^2$C Bus Functions.



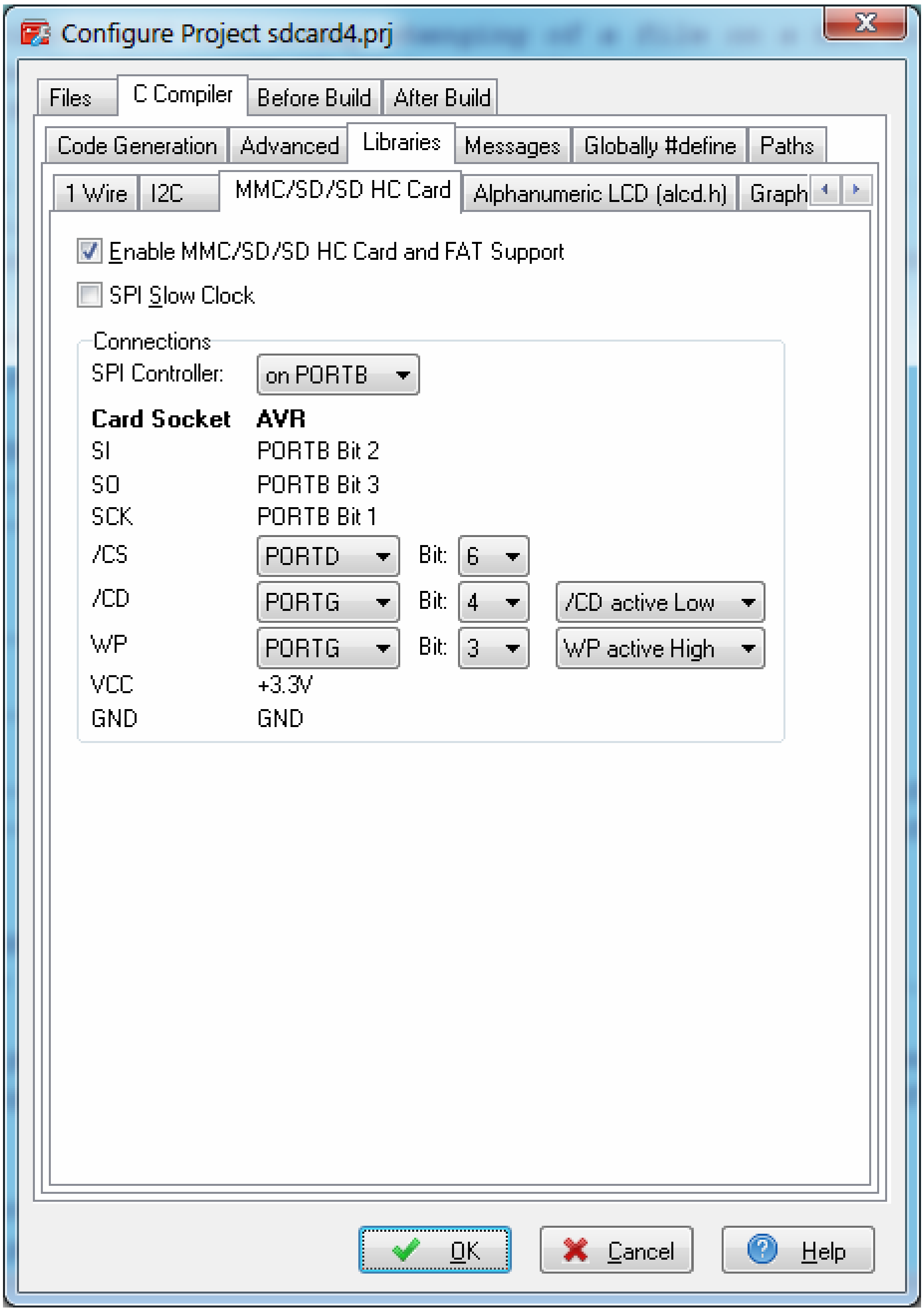

The following settings are available:

- **Enable Bit-Banged I2C Support** allows the activation of the **I$^2$C Bus Functions** library.
- **I/O Port, SDA** and **SCL** specify in **Data Connection**, the port and bits used for I$^2$C bus communication
- **Bit Rate** specifies the frequency of the clock pulses on the **SCL** line.

### 3.3.6.6 Setting the MMC/SD/SD HC Card Library Options

The **Libraries** tab is used for configuring specific driver libraries used by the compiler.

The **MMC/SD/SD HC Card** tab is used for configuring the I/O port allocation for the MMC/SD/SD HC FLASH Memory Card Driver Functions.




The **Enable MMC/SD/SD HC Card and FAT Support** check box activates the appropriate **MMC/SD/SD HC FLASH Memory Card Driver** and **FAT Access Functions** libraries.

The **SPI Slow Clock** options allows to use a two times slower data rate when communicating with the MMC/SD/SD HC Card in order to provide better compatibility with some hardware designs.

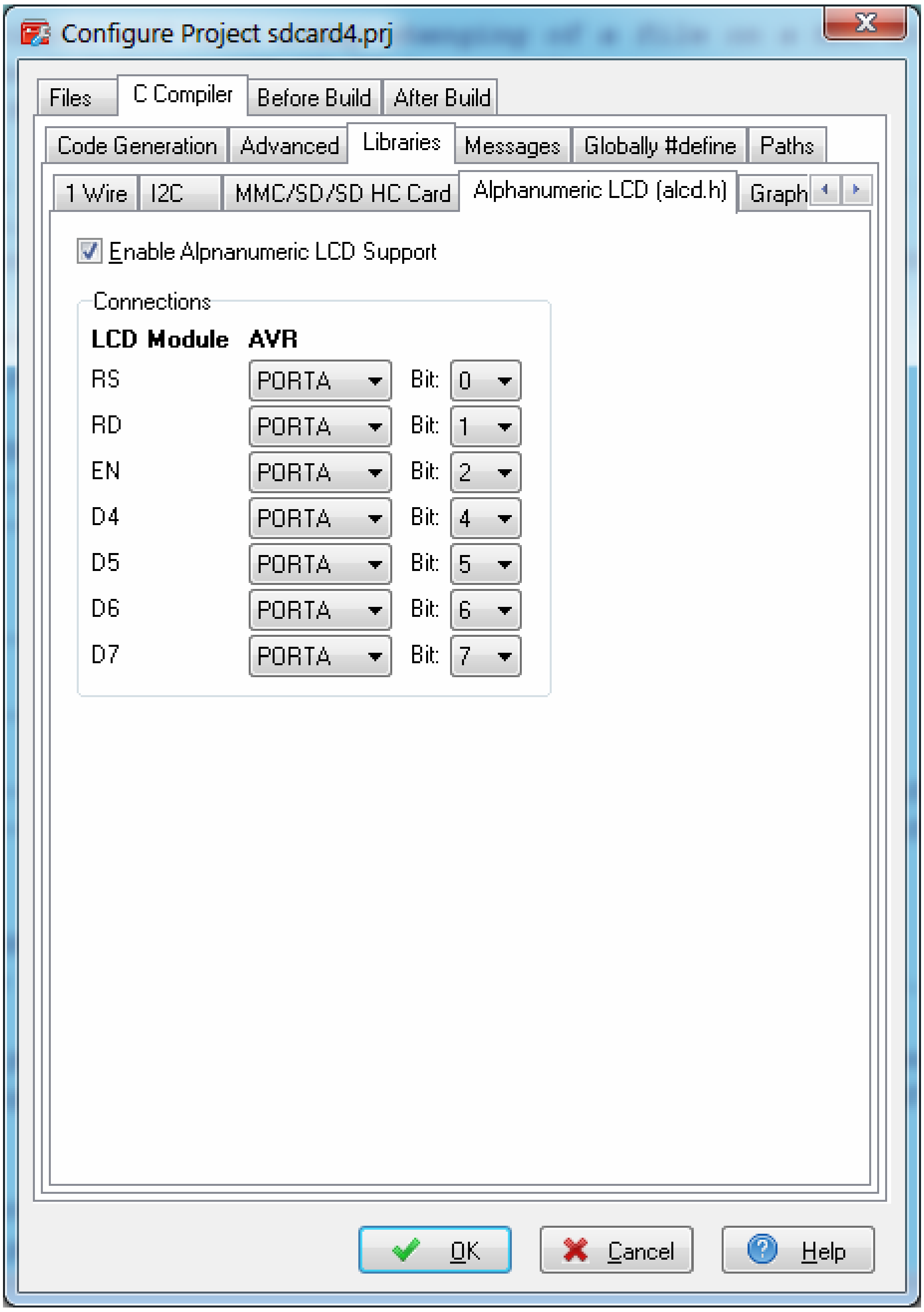
The user has the possibility to specify the polarity of the **CD (Card Detect)** signal as active **Low** or **High**, and even to disable it's usage.
In this situation, no I/O port signal is allocated for this purpose and the presence of the card must be detected by calling the **sdcard_present** function from the **MMC/SD/SD HC FLASH Memory Card Driver** library.

The polarity of the **WP (Write Protect)** signal can also be specified as active **Low** or **High**, or it's usage can be totally disabled.
In this later case, no I/O port signal will be allocated for it.

## 3.3.6.7 Setting the Alphanumeric LCD Library Options

The **Libraries** tab is used for configuring specific driver libraries used by the compiler.

The **Alphanumeric LCD** tab is used for configuring the I/O port allocation for the **LCD Functions for displays with up to 2x40 characters**.

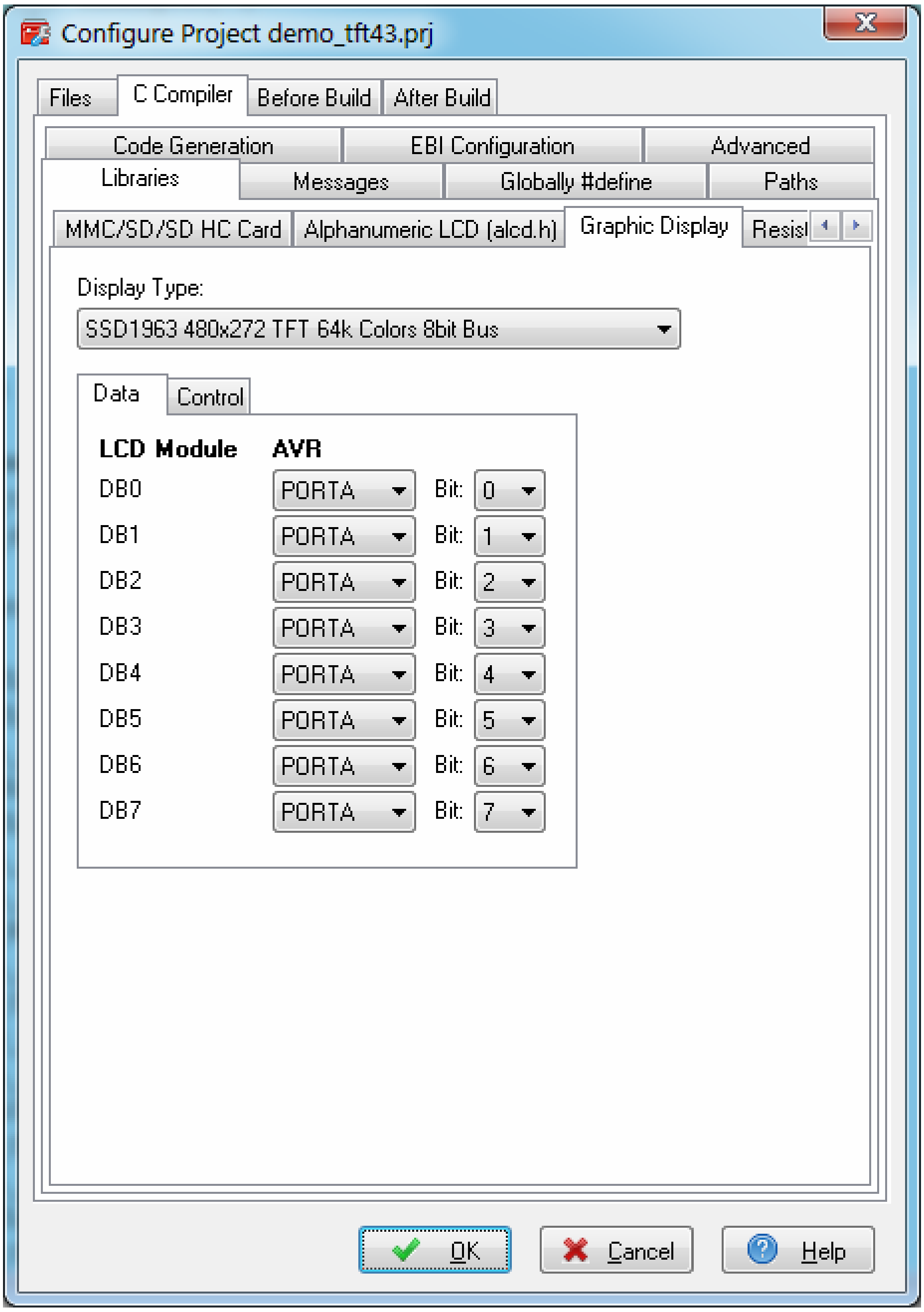



The **Enable Alhanumeric LCD Support** check box activates the configuration specified for the *alcd.h* library functions.
The connections between the LCD module and the AVR I/O ports can be specified individually for each signal in the **Connections** group box.

## 3.3.6.8 Setting the Graphic Display Library Options

The **Libraries** tab is used for configuring specific driver libraries used by the compiler.

The **Graphic Display** tab is used for configuring the I/O port allocation for the **Graphic Display Functions**.

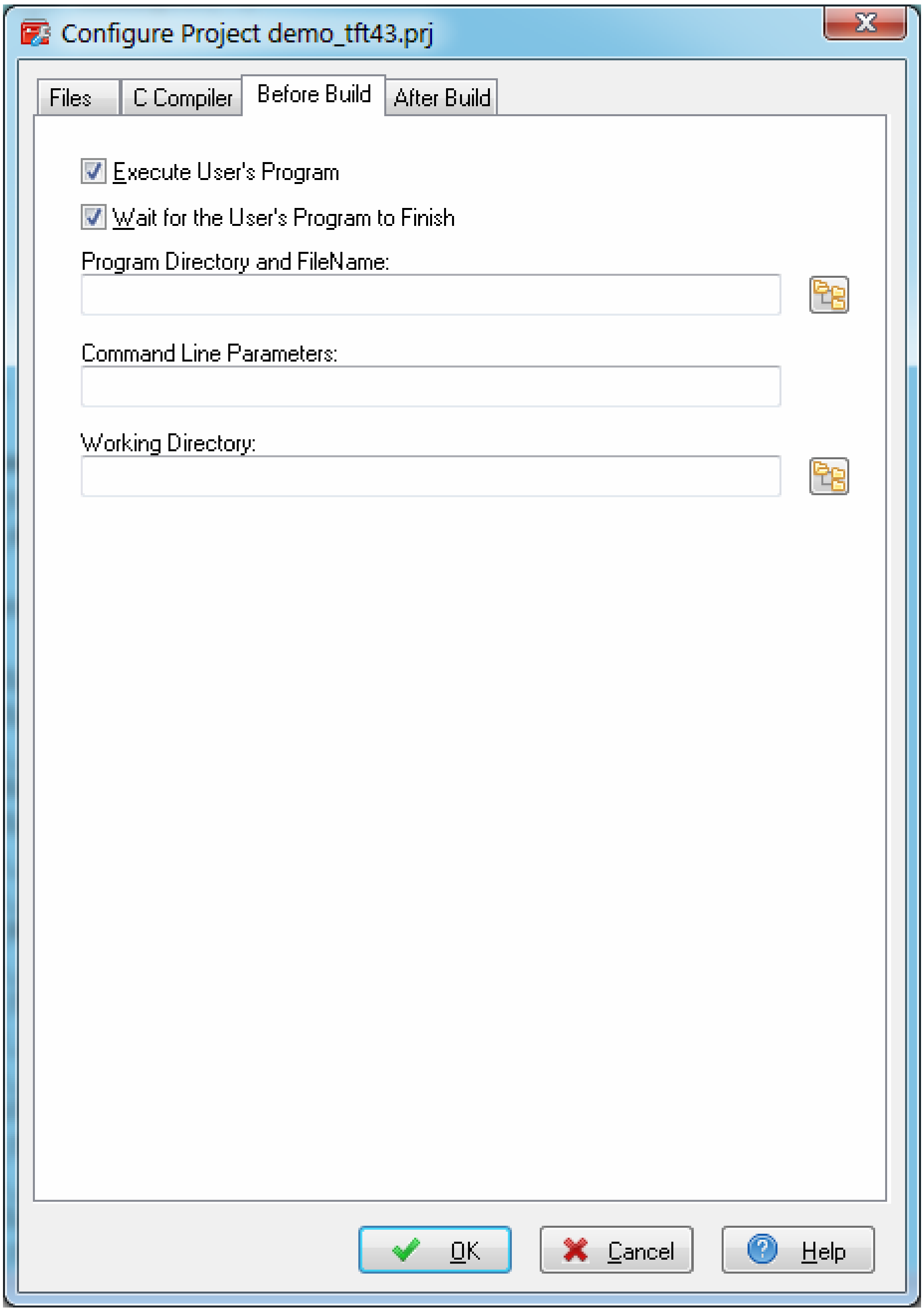



The **Display Type** list box allows to select the graphic controller type and LCD resolution.

The connections between the graphic display module and the AVR I/O ports can be specified individually for each signal in the **Connections** group box.
**Note:** In order to obtain maximum performance, it is advised to set the display controller's **data bus** bits to match the bits with the same numbers of the same AVR I/O port.

## 3.3.6.9 Executing an User Specified Program before Build

This option is available if you select the **Before Build** tab in the Project Configure window.
If you check the **Execute User's Program** option, then a program, that you have previously specified, will be executed before the compilation/assembly process.




The following parameters can be specified for the program to be executed:

- Program Directory and File Name
- Program Command Line Parameters
- Program Working Directory.

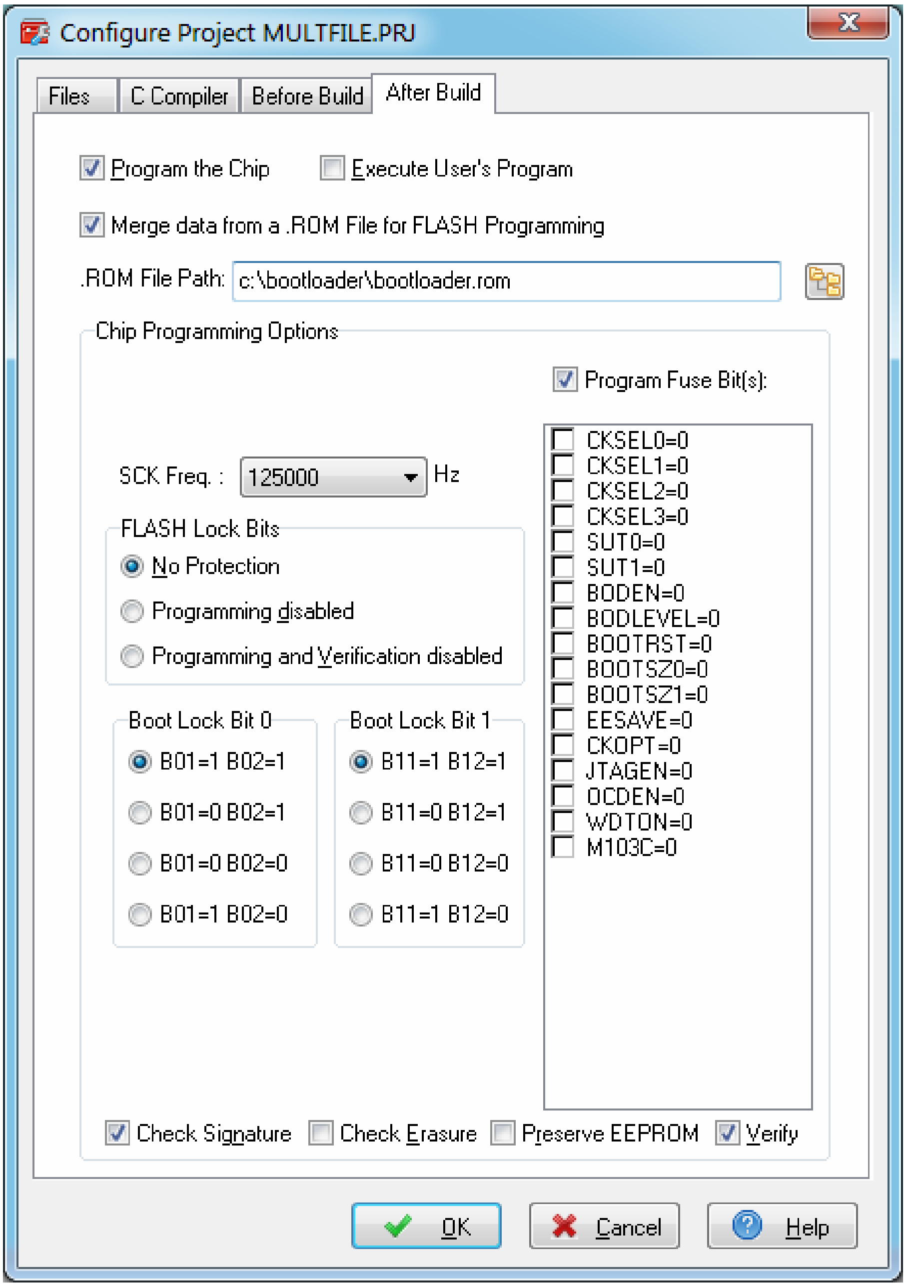
There is also the option to wait for the user's program to finish it's execution before staring the Build process.
Pressing the  button allows to select a directory and file.

The command line can accept the following parameters:

- %bc substitutes the Active Build Configuration: DEBUG or RELEASE
- %P substitutes the full project path
- %p substitutes the project name without path
- %h substitutes the name of the .hex file created by the compiler
- %e substitutes the name of the .eep file created by the compiler
- %f<project_file_number> substitutes the project's source file name without path
- %F<project_file_number> substitutes the project's source file name with full path.

## 3.3.6.10 Transferring the Compiled Program to the AVR Chip after Build

This option is available if you select the **After Build** tab in the Project Configure window.



If you check the **Program the Chip** option, then after successful compilation/assembly your program will be automatically transferred to the AVR chip using the built-in Programmer software.

The following steps are executed automatically:
- Chip erasure
- FLASH and EEPROM blank check
- FLASH programming and verification
- EEPROM programming and verification
- Fuse and Lock Bits programming

The **Merge data from a .ROM File for FLASH Programming** option, if checked, will merge in the FLASH programming buffer the contents of the .ROM file, created by the compiler after Make, with the data from the **.ROM** file specified in **.ROM File Path**.
This is useful, for example, when adding a boot loader executable compiled in another project, to an application program that will be programmed in the FLASH memory.

You can select the type of the chip you wish to program using the **Chip** combo box.

The SCK clock frequency used for In-System Programming with the STK500, AVRISP or AVRISP MkII can be specified using the **SCK Freq.** listbox. This frequency must not exceed ¼ of the chip's clock frequency.

If the chip you have selected has Fuse Bit(s) that may be programmed, then a supplementary **Program Fuse Bit(s)** check box will appear.
If it is checked, than the chip's Fuse Bit(s) will be programmed after **Build**.

The Fuse Bit(s) can set various chip options, which are described in the Atmel data sheets.
If a Fuse Bit(s) check box <u>is checked</u>, then the corresponding fuse bit <u>will be set to 0</u>, the fuse being considered as programmed (as per the convention from the Atmel data sheets).
If a Fuse Bits(s) check box <u>is not checked</u>, then the corresponding fuse bit <u>will be set to 1</u>, the fuse being considered as not programmed.

If you wish to protect your program from copying, you must select the corresponding option using the **FLASH Lock Bits** radio box.

If you wish to check the chip's signature before programming you must use the **Check Signature** option.

To speed up the programming process you can uncheck the **Check Erasure** check box.
In this case there will be no verification of the correctness of the FLASH erasure.

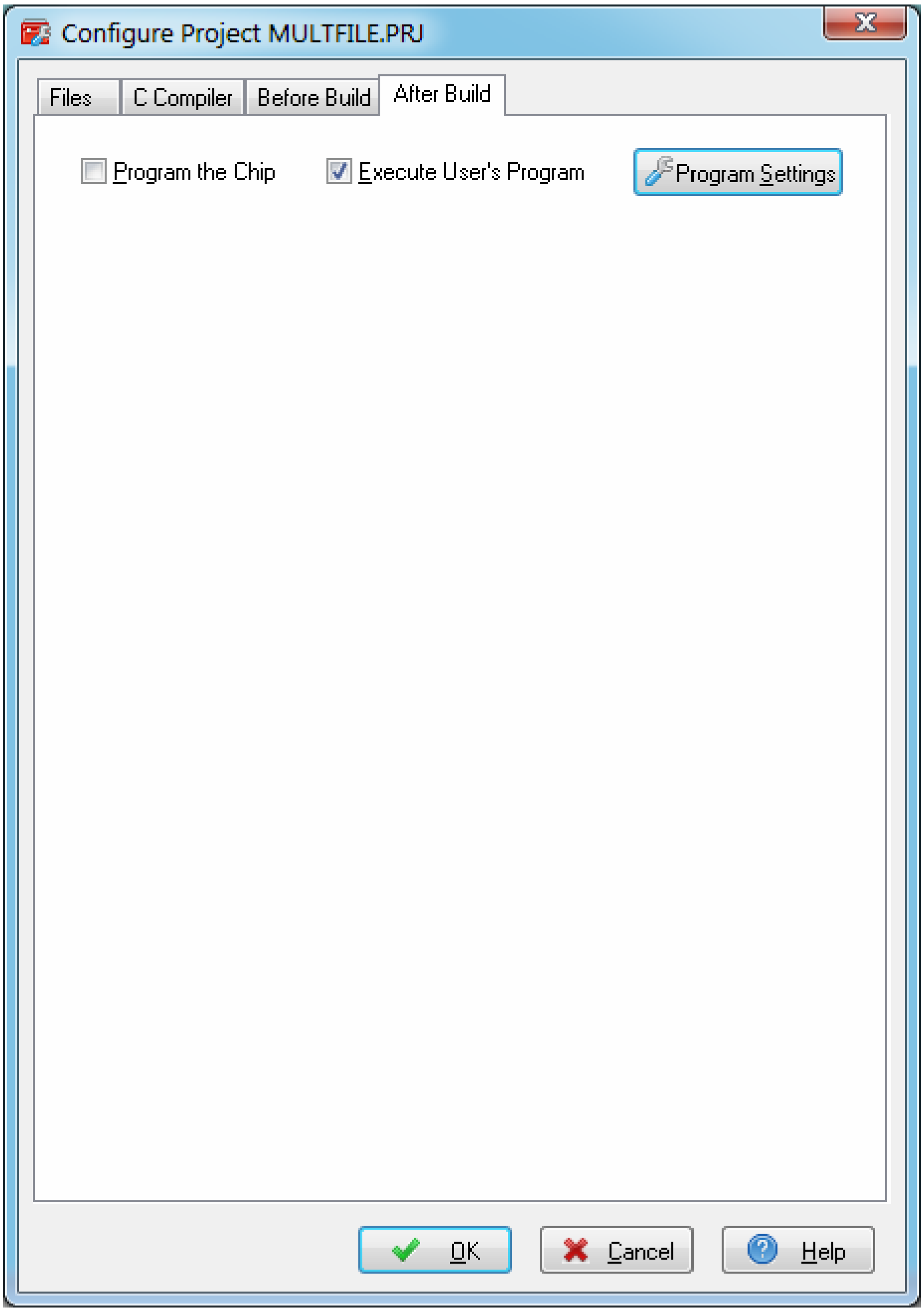
The **Preserve EEPROM** checkbox allows preserving the contents of the EEPROM during chip erasure.

To speed up the programming process you can uncheck the **Verify** check box.
In this case there will be no verification of the correctness of the FLASH and EEPROM programming.

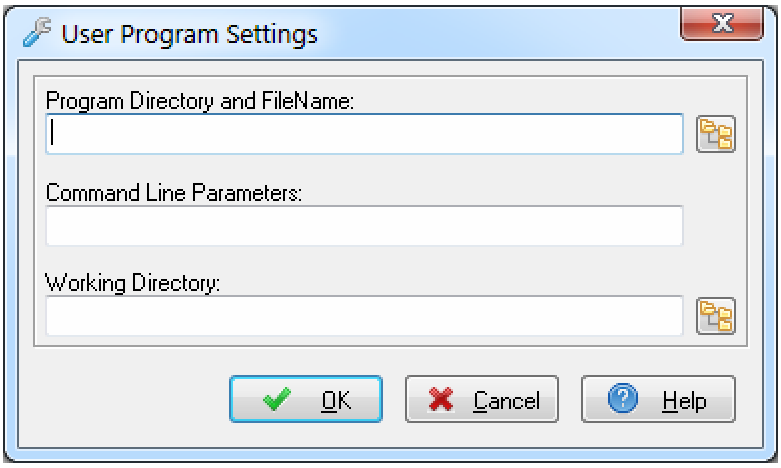
Changes can be saved, respectively canceled, using the **OK**, respectively **Cancel** buttons.

## 3.3.6.11 Executing an User Specified Program after Build

This option is available if you select the **After Build** tab in the Project Configure window.
If you check the **Execute User's Program** option, then a program, that you have previously specified, will be executed after the compilation/assembly process.



Using the **Program Settings** button you can modify the:
- Program Directory and File Name
- Program Command Line Parameters
- Program Working Directory




Pressing the button allows to select a directory and file.
Changes can be saved, respectively canceled, using the **OK**, respectively **Cancel** buttons.

The command line can accept the following parameters:

- %bc substitutes the Active Build Configuration: DEBUG or RELEASE
- %P substitutes the full project path
- %p substitutes the project name without path
- %h substitutes the name of the .hex file created by the compiler
- %e substitutes the name of the .eep file created by the compiler
- %f<project_file_number> substitutes the project's source file name without path
- %F<project_file_number> substitutes the project's source file name with full path.

## 3.3.7 Obtaining an Executable Program

Obtaining an executable program requires the following steps:

1. Compiling the Project's C program modules, using the CodeVisionAVR C Compiler, and obtaining object files needed by the linker
2. Linking the object files files created during compilation and obtaining a single assembler source file
3. Assembling the assembler source file, using the Atmel AVR assembler AVRASM2.

**Compiling**, executes step 1.
**Building**, executes step 1, 2 and 3.

Compilation is performed only for the program modules that were modified since the previous similar process.
This leads to significant project build reduction times, compared with the old CodeVisionAVR V1.xx, where all the program modules were compiled even if they were not changed.

The resulting **.rom**, **.hex** and **.eep** files will be placed in the **Executable Files** directory.
The object files, including the **.cof** COFF object file used for debugging, will be placed in the **Object Files** directory.
The **.asm**, **.lst** and **.map** files will be placed in the **List Files** directory.
Various files created by the linker during the **Build** process will be placed in the **Linker Files** directory.

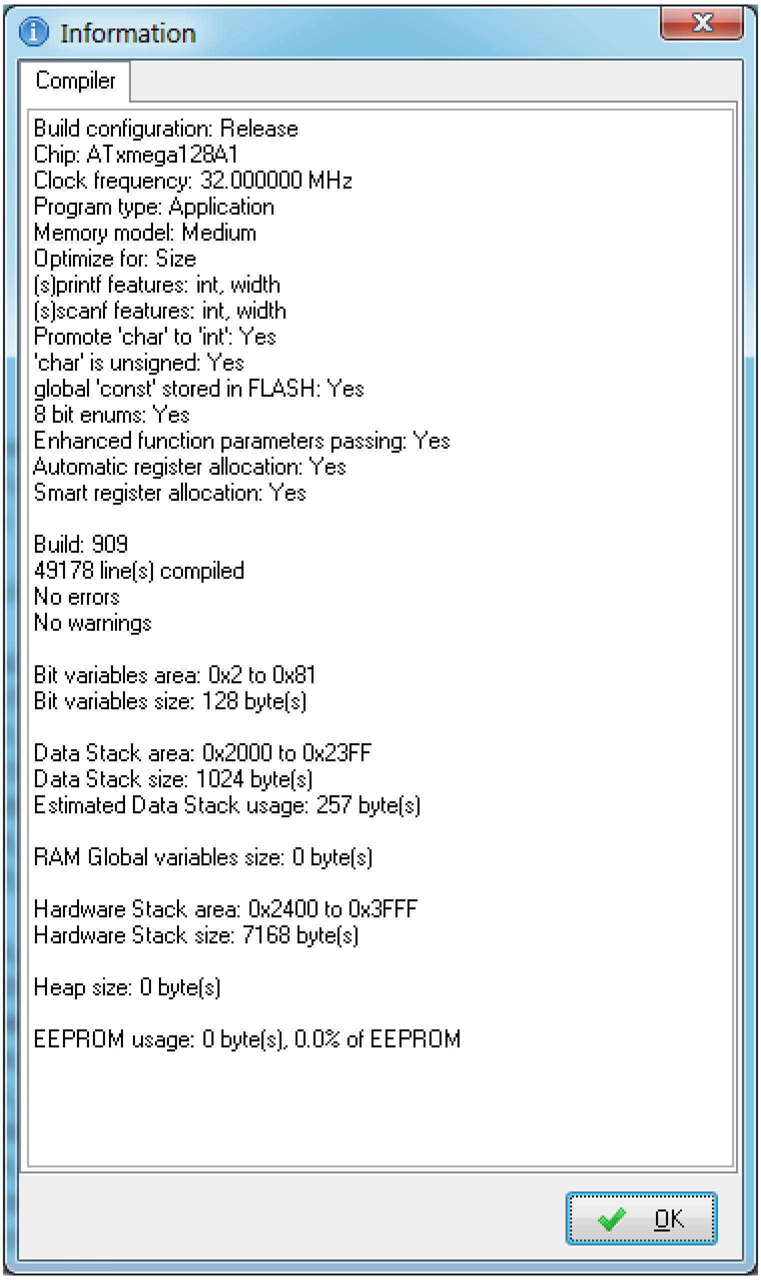
The **Executable Files**, **Object Files, List Files** and **Linker Files** directories are specified in the **Project|Configure|Files|Output** menu.

### 3.3.7.1 Checking Syntax

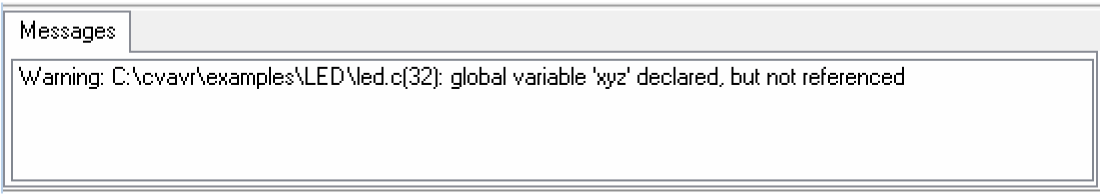
Checking the currently edited source file for syntax errors can be performed by using the **Project|Check Syntax** menu or by pressing the toolbar button.
This function is useful because it's faster than **Project|Compile** menu command, which compiles all the modified files in a project.
It can also be executed by selecting **Check Syntax** in the pop-up menu, which is invoked by right clicking with the mouse in the editor window.

## 3.3.7.2 Compiling the Project

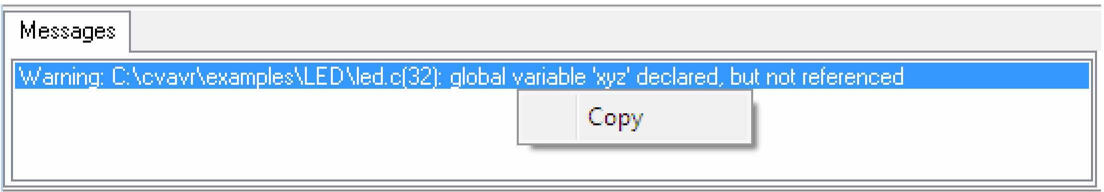
To compile the Project you must use the **Project|Compile** menu command, press the **F9** key or the button of the toolbar. The CodeVisionAVR C Compiler will be executed, producing the object files needed by the linker.
Compilation will be performed only for the program modules that were modified since the previous similar process.
The compilation process can be stopped using the **Project|Stop Compilation** menu command or by pressing the button on the toolbar.
After the compilation is complete, an **Information** window will open, showing the compilation results.



Eventual compilation errors and/or warnings will be listed in the **Message** window located under the **Editor** window, or in the **Code Navigator** window.



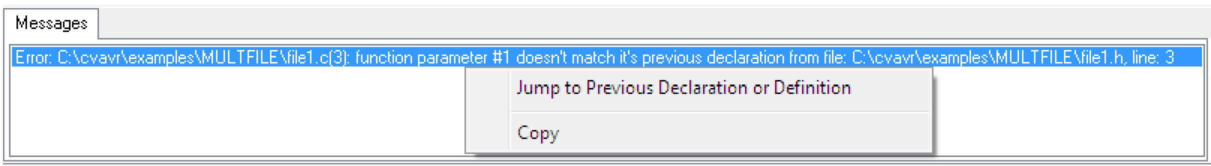

By left clicking with the mouse on the error or warning message, the source line with the problem will be highlighted.
Right clicking with mouse opens a pop-up menu that contains the option to **Copy** the error message to the clipboard:




The **Project|Go to Next Error**, respectively **Project|Go to Previous Error** menu commands, the **F8**, respectively **Ctrl+F8** keys or the , respectively toolbar buttons, allow moving to the next, respectively previous error message.

The **Project|Go to Next Warning**, respectively **Project|Go to Previous Warning** menu commands, the **F4**, respectively **Ctrl+F4** keys or the , respectively toolbar buttons, allow moving to the next, respectively previous warning message.

If the message refers also to a previous declaration or definition from a file that is different than the one where the error was signaled, right clicking with the mouse opens a pop-up menu with the **Jump to Previous Declaration or Definition** option:




Selecting this option will highlight the source line where the previous declaration or definition was made.

The size of the **Message** window can be modified using the horizontal slider bar placed between it and the **Editor** window.

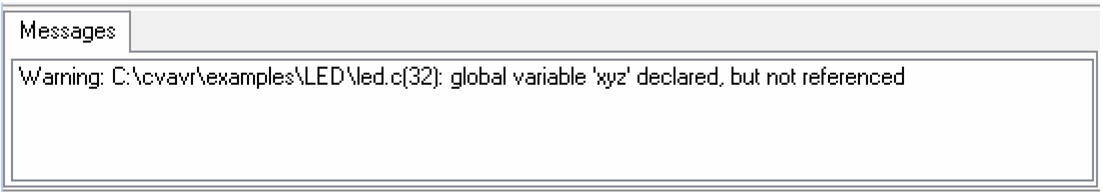
## 3.3.7.3 Building the Project

To build the Project you must use the **Project|Build** menu command, press the **Shift+F9** keys or the button of the toolbar. The CodeVisionAVR C Compiler will be executed, producing the object files needed by the linker.
Compilation will be performed only for the program modules that were modified since the previous similar process.
If the complete recompilation of all the program modules is needed, then the **Project|Build All** menu command or the button of the toolbar must be used.

After successful compilation the object files will be linked and an assembly **.asm** file will be produced.
If no compilation or linking errors were encountered, then the Atmel AVR assembler AVRASM2 will be executed, obtaining the output file types specified in **Project|Configure|C Compiler|Code Generation**.

The build process can be stopped using the **Project|Stop Compilation** menu command or by pressing the button on the toolbar.

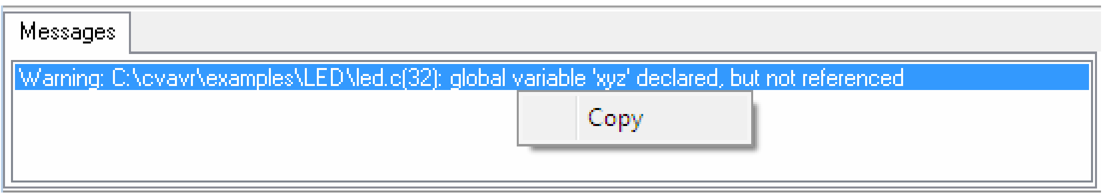
Eventual compilation errors and/or warnings will be listed in the **Message** window located under the **Editor** window, or in the **Code Navigator** window.



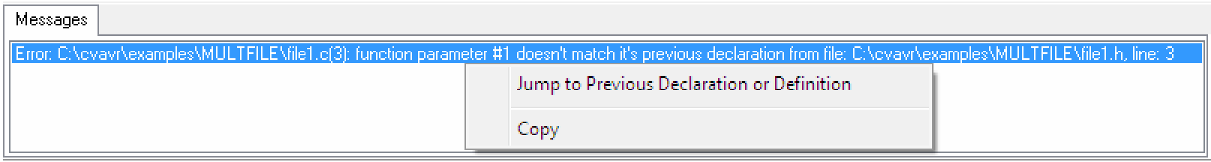

The **Project|Go to Next Error**, respectively **Project|Go to Previous Error** menu commands, the **F8**, respectively **Ctrl+F8** keys or the , respectively toolbar buttons, allow moving to the next, respectively previous error message.

The **Project|Go to Next Warning**, respectively **Project|Go to Previous Warning** menu commands, the **F4**, respectively **Ctrl+F4** keys or the , respectively toolbar buttons, allow moving to the next, respectively previous warning message.

By left clicking with the mouse on the error or warning message, the source line with the problem will be highlighted.
Right clicking with mouse opens a pop-up menu that contains the option to **Copy** the error message to the clipboard:

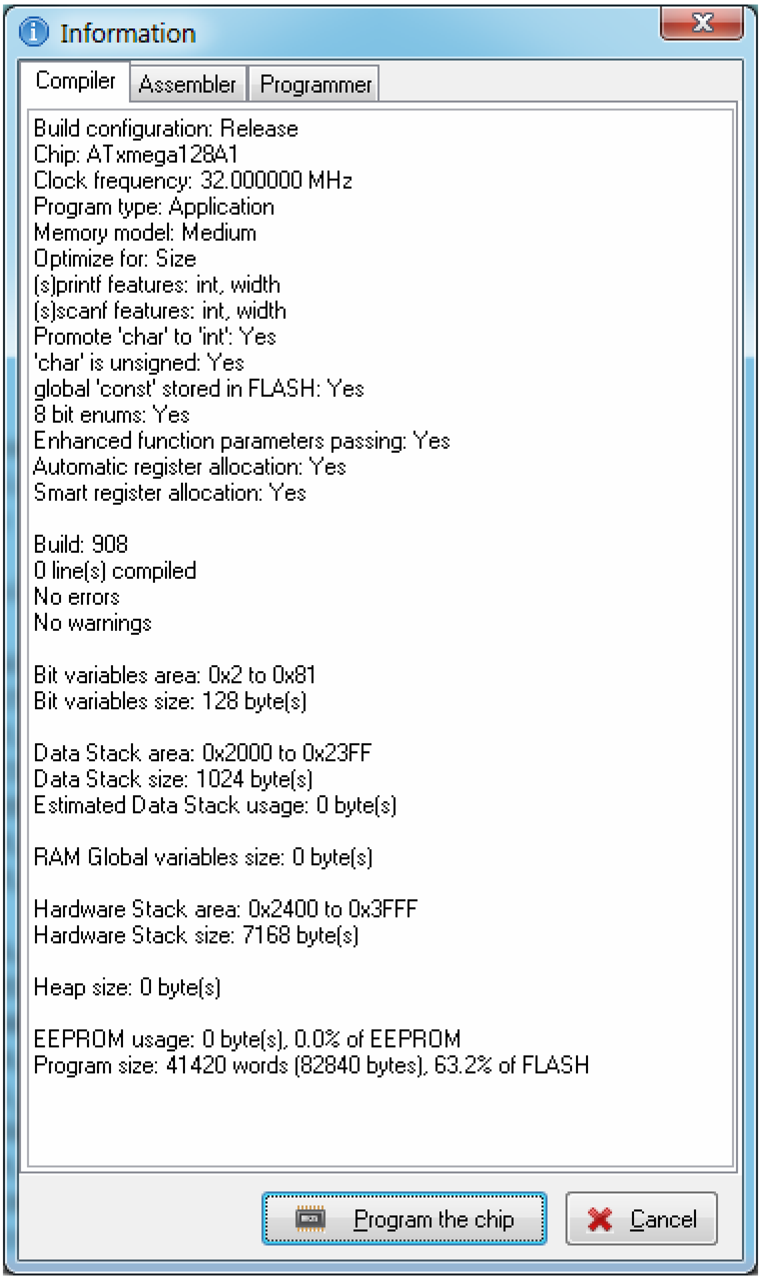


If the message refers also to a previous declaration or definition from a file that is different than the one where the error was signaled, right clicking with the mouse opens a pop-up menu with the **Jump to Previous Declaration or Definition** option:

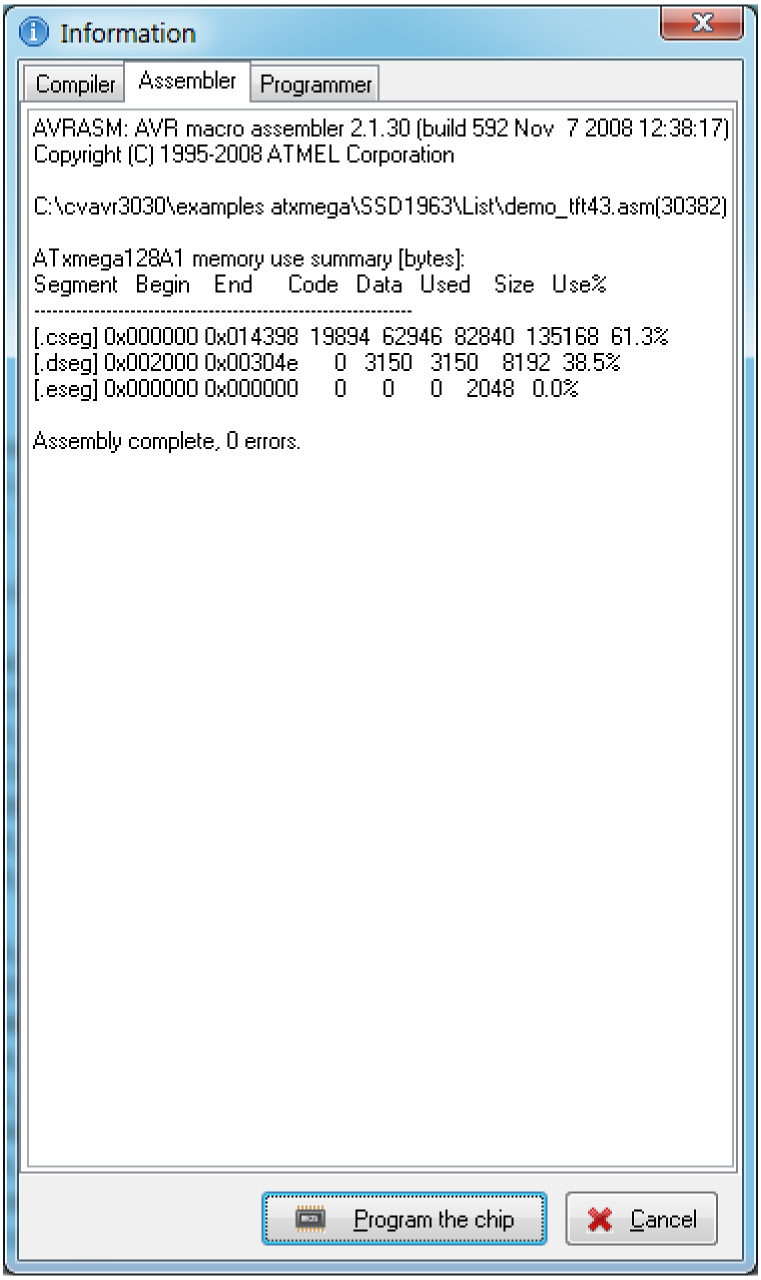



Selecting this option will highlight the source line where the previous declaration or definition was made.

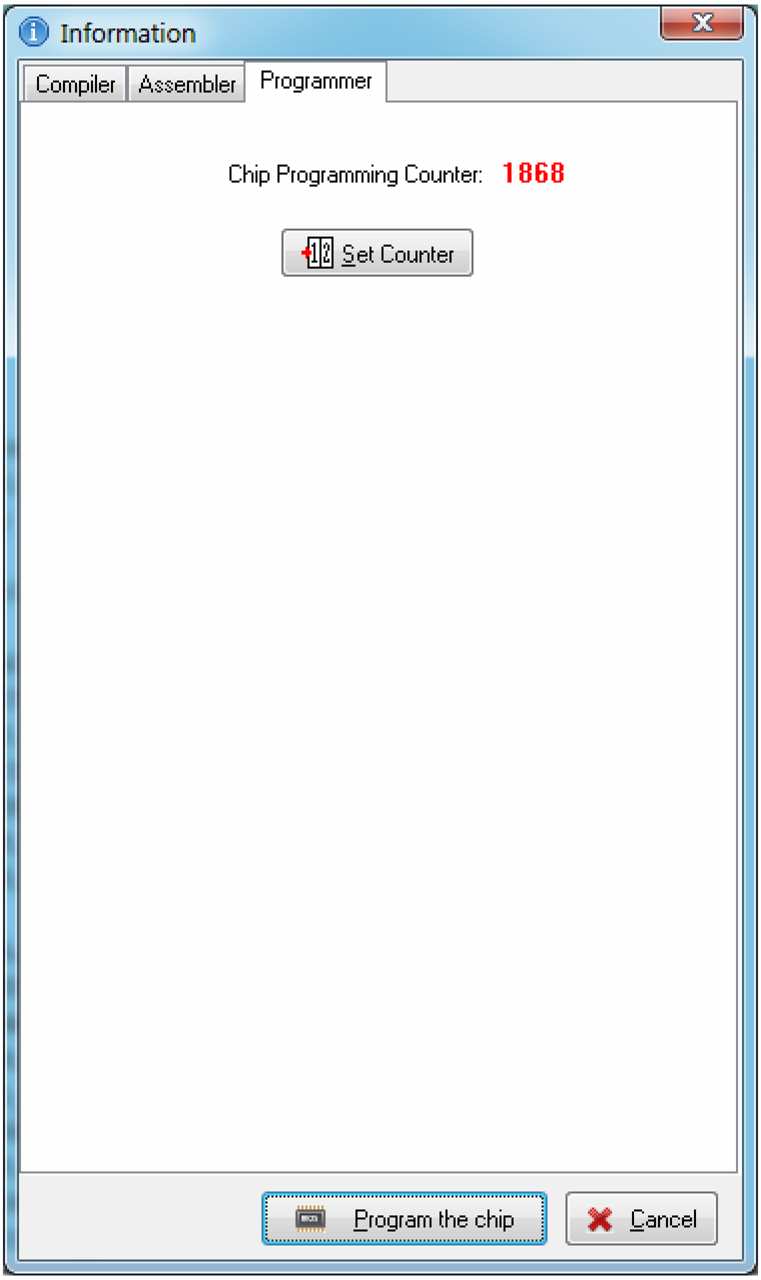
After the build process is completed, an **Information** window will open showing the build results. Pressing the **Compiler** tab will display compilation results.



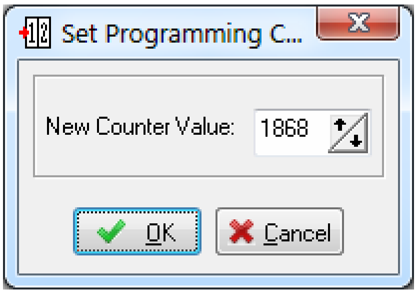
Pressing the **Assembler** tab will display assembly results.



Pressing the **Programmer** tab will display the **Chip Programming Counter**, which shows how many times was the AVR chip programmed so far.



Pressing the **Set Counter** button will open the **Set Programming Counter** window:




This dialog window allows setting the new **Chip Programming Counter** value.

Pressing the **Program the chip** button allows automatic programming of the AVR chip after successful build. Pressing **Cancel** will disable automatic programming.

### 3.3.7.4 Cleaning Up the Project Output Directories

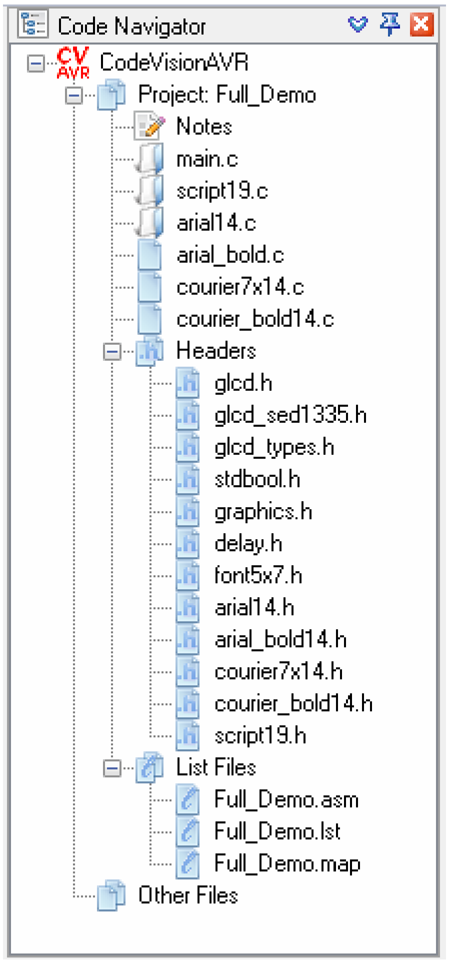
The various files created during the **Project Build** process can be deleted using the **Project|Clean Up** menu or by pressing the button on the toolbar.

The following **Project Output Directories** will be cleaned:

- **Object Files** directory - all files will be deleted, except the **.cof** COFF object file
- **List Files** directory - all files will be deleted, except the **.asm** and **.vec** assembly source files
- **Linker Files** directory – all files will be deleted.

### 3.3.7.5 Using the Code Navigator

The **Code Navigator** window allows displaying or opening of the project source files, along with errors or warnings that occured during the compile or build processes.

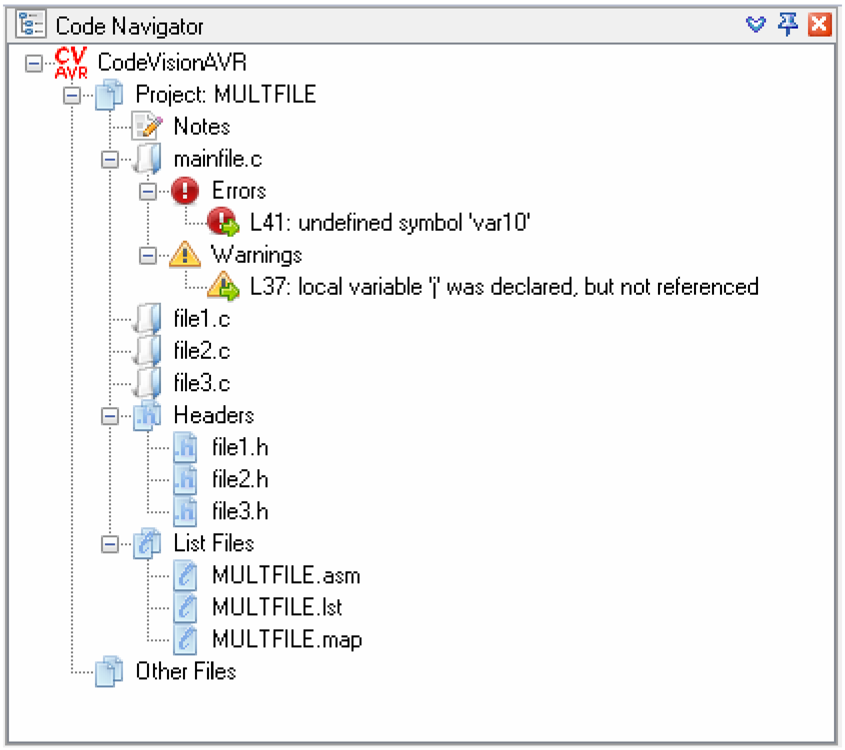



The project's program modules are listed as children of the **Project** node.
Other opened files, that are not part of the project, are listed as children of the **Other Files** node.
By clicking on a closed file node, the appropriate file is opened in the editor.
If the file is already opened, clicking on its node, will make it active in the editor.

After a **Compile** or **Build** process there is also displayed a list of header .h files that were #included in the project's program modules during this process.
The headers files are available as children of the **Headers** node.
By clicking on a closed header file node, the appropriate file is opened in the editor.
If the header file is already opened, clicking on its node, will make it active in the editor.

The **List Files** node contains the assembly, list and map files generated by the compiler after the **Compile** or **Build** process.
By clicking on a closed list file node, the appropriate file is opened in the editor.
If the list file is already opened, clicking on its node, will make it active in the editor.

If during compilation there are errors or warnings, these are also displayed in the **Code Navigator** window.




By clicking on the error  or warning  node, the corresponding source line is highlighted in the appropriate file.

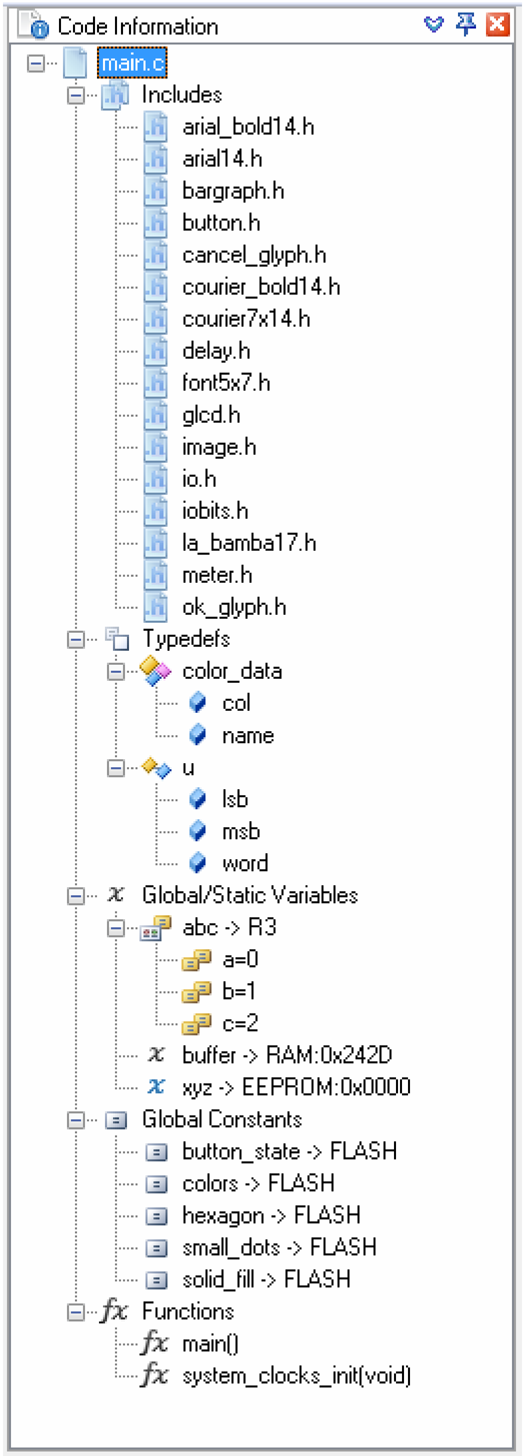
The **Code Navigator** tree branches can be expanded, respectively collapsed, by clicking on the **+**, respectively **-**, buttons.

By right clicking in the **Code Navigator** window you can open a pop-up menu with the following choices:

- **Open** a file
- **Save** the currently edited file
- **Save All** opened files
- **Close** currently edited file
- **Close All** opened files
- Toggle on or off alphabetically sorting the files in the **Code Navigator**
- Toggle on or off expanding the **Errors** and **Warnings** branches after a **Compile** or **Build** process
- Toggle on or off expanding the header file branches.

## 3.3.7.6 Using the Code Information

The **Code Information** window allows for easy access to declarations and definitions made in the currently edited source file.




The **Code Information** window is accessed using the tab with the same name and appears after the first **Compile** or **Build** process of the currently opened project.

The information is displayed in the form of a tree with several types of nodes:

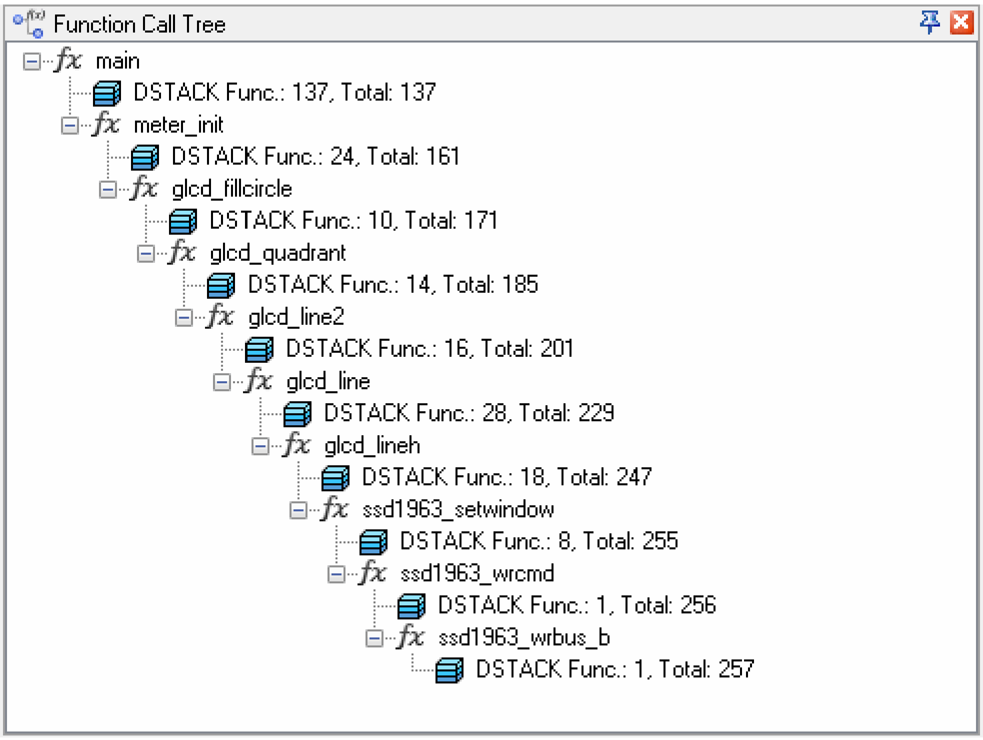
- **Includes** node which displays all the header .h files #included in the currently edited source file. Clicking on a header node moves the cursor to the corresponding *#include* directive in the edited source file.
- **Macros** node which displays all the preprocessor macros defined in the currently edited source file. Clicking on a macro node moves the cursor to the corresponding *#define* directive in the edited source file.
- **Typedefs** node which displays all the data types defined in the currently edited source file. Clicking on a type definition node moves the cursor to the corresponding data type definition in the edited source file. If the defined data type is a structure, union or enumeration, then its members/items are displayed as additional or nodes.
- **Global/Static Variables** node which displays all the global and static variables declared in the currently edited source file. Clicking on a RAM variable node or EEPROM variable node moves the cursor to the corresponding declaration in the edited source file.
- **Global Constants** node which displays all the global constants declared in the currently edited source file. Clicking on a constant node moves the cursor to the corresponding declaration in the edited source file.
- **Functions** node which displays all the functions that were defined in the currently edited source file. Clicking on a function node moves the cursor to the corresponding definition in the edited source file.

The **Code Information** tree branches can be expanded, respectively collapsed, by clicking on the **+**, respectively **-**, buttons.
By right clicking in the **Code Information** window you can open a pop-up menu with the following choices:

- Toggle on or off alphabetically sorting the items in the **Code Information** tree
- Toggle on or off expanding the **Code Information** tree branches.

## 3.3.7.7 Using the Function Call Tree

The **Function Call Tree** window displays the function call sequence that uses the largest amount of Data Stack during program execution.




The **Function Call Tree** window is accessed using the tab with the same name and appears after the first **Compile** or **Build** process of the currently opened project.

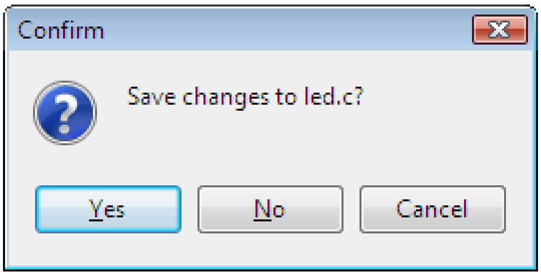
The Data Stack usage information is represented in the form of a tree with two types of nodes:

- Function *fx* nodes. Clicking on a function name moves the cursor to the corresponding definition in the source file.
- DSTACK nodes display the data stack used by the parent function and the total level of the Data Stack when the program is executed inside the function.

### 3.3.8 Closing a Project

You can quit working with the current Project by using the **File|Close All** menu command or the toolbar button.

If the Project files were modified, and were not saved yet, you will be asked if you want to do that.




Pressing **Yes** will save changes and close the project.
Pressing **No** will close the project without saving the changes.
Pressing **Cancel** will disable the project closing process.

When saving, the IDE will create a backup file with a **.prj~** extension.

## 3.4 Tools

Using the **Tools** menu you can execute other programs without exiting the CodeVisionAVR IDE.

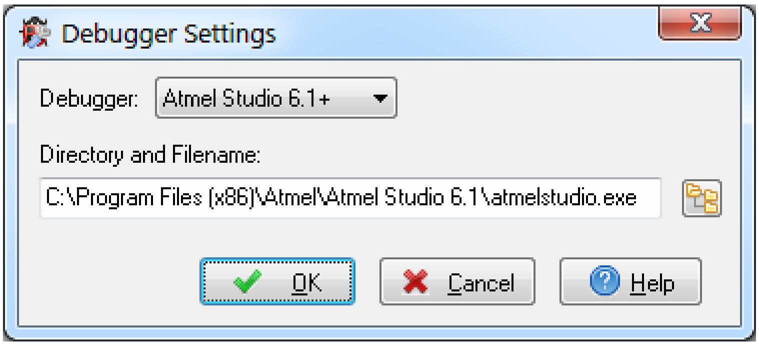
### 3.4.1 The AVR Debugger

The CodeVisionAVR C Compiler is designed to work along with the following debuggers from Atmel:

- AVR Studio 4.19
- Atmel Studio 6.1 or later version.

The compiler will generate a .COF object file that can be opened with the above mentioned programs, allowing C source and assembly debbuging.

Before you can invoke the debugger, you must first specify its location and file name using the **Settings|Debugger** menu command.




The **Debugger** list box allows to select one of the three versions of debuggers compatible with CodeVisionAVR.
After selecting the debugger, the IDE will detect automatically it's installation path and display it in the **Directory and Filename** edit box.
This path can be also manually edited and eventually other location can be selected by pressing the button.

Changes can be saved, respectively canceled, using the **OK**, respectively **Cancel** buttons.

The debugger is executed by selecting the **Tools|Debugger** menu command or by pressing the button on the toolbar.

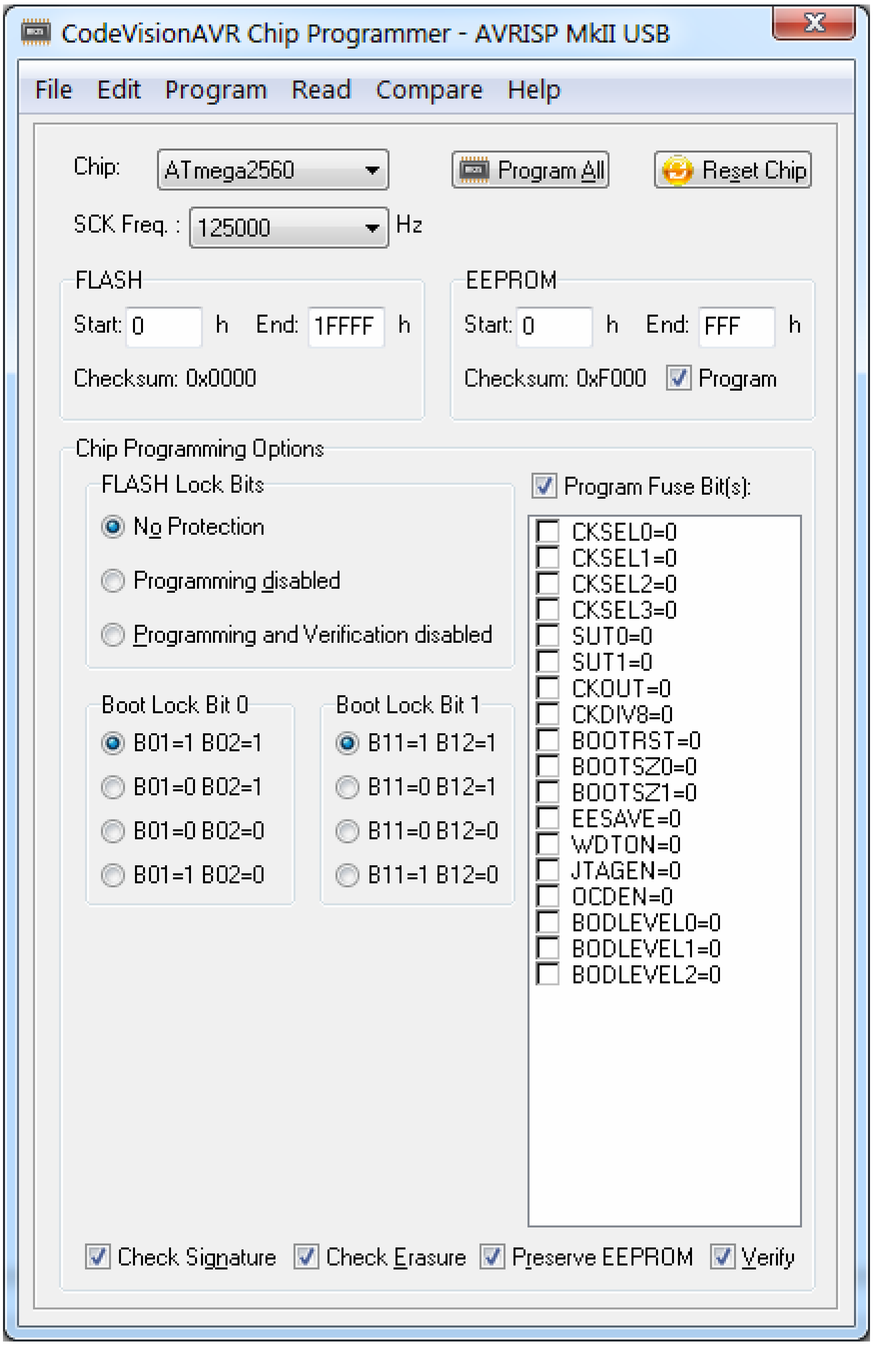
Details about using the debuggers with CodeVisionAVR can be found in the following chapters:

**2. Using the CodeVisionAVR Extension for Atmel Studio**
**4.21 Using the AVR Studio 4.19 Debugger**.

## 3.4.2 The AVR Chip Programmer

The CodeVisionAVR IDE has a built-**in In-System AVR Chip Programmer** that lets you easily transfer your compiled program to the microcontroller for testing.
The Programmer is designed to work with the Atmel STK500, AVRISP, AVRISP MkII, AVR Dragon, JTAGICE MkII, JTAGICE 3, AVRProg (AVR910 application note), Kanda Systems STK200+, STK300, Dontronics DT006, Vogel Elektronik VTEC-ISP, Futurlec JRAVR or the MicroTronics ATCPU, Mega2000 development boards.
The type of the used programmer and the printer port can be selected by using the **Settings|Programmer** menu command.

The Programmer is executed by selecting the **Tools|Chip Programmer** menu command or by pressing the button on the toolbar.



You can select the type of the chip you wish to program using the **Chip** combo box.

The SCK clock frequency used for In-System Programming with the STK500, AVRISP or AVRISP MkII can be specified using the **SCK Freq.** listbox. This frequency must not exceed ¼ of the chip's clock frequency.

The **EEPROM|Program** check box allows to enable or disable EEPROM programming when the **Program|All** menu command is executed or when the **Program All** button is pressed.

If the chip you have selected has Fuse Bit(s) that may be programmed, then a supplementary **Program Fuse Bit(s)** check box will appear.
If it is checked, than the chip's Fuse Bit(s) will be programmed when the **Program|All** menu command is executed or when the **Program All** button is pressed.

The Fuse Bit(s) can set various chip options, which are described in the Atmel data sheets.
If a Fuse Bit(s) check box is checked, then the corresponding fuse bit will be set to 0, the fuse being considered as programmed (as per the convention from the Atmel data sheets).
If a Fuse Bits(s) check box is not checked, then the corresponding fuse bit will be set to 1, the fuse being considered as not programmed.

If you wish to protect your program from copying, you must select the corresponding option using the **FLASH Lock Bits** radio box.
The Programmer has two memory buffers:

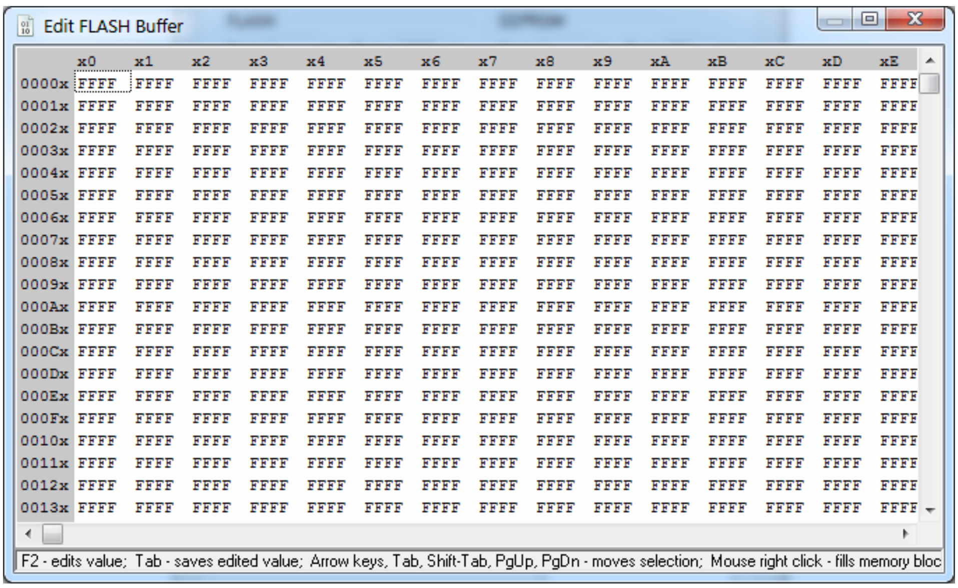
- The FLASH memory buffer
- The EEPROM memory buffer.

You can Load or Save the contents of these buffers using the **File** menu.
Supported file formats are:

- Atmel .rom and .eep
- Intel HEX
- Binary .bin

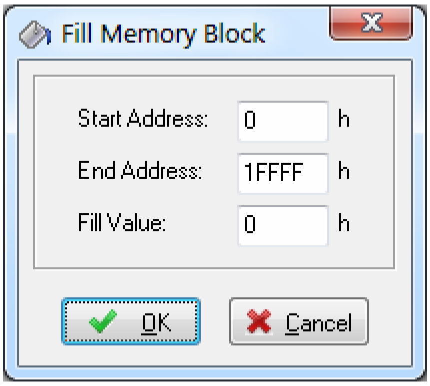
After loading a file in the corresponding buffer, the **Start** and **End** addresses are updated accordingly.
You may also edit these addresses if you wish.

The contents of the FLASH, respectively EEPROM, buffers can be displayed and edited using the **Edit|FLASH** , respectively **Edit|EEPROM** menu commands.
When one of these commands is invoked, an Edit window displaying the corresponding buffer contents will open:



The buffer's contents, at the highlighted address, can be edited by pressing the **F2** key and typing in the new value. The edited value is saved by pressing the **Tab** or **arrow** keys.
The highlighted address can be modified using the **arrow**, **Tab**, **Shift+Tab**, **PageUp** or **PageDown** keys.

The **Fill Memory Block** window can be opened by right clicking in the Edit window:




This window lets you specify the **Start Address**, **End Address** and **Fill Value** of the memory area to be filled.

If you wish to check the chip's signature before any operation you must use the **Check Signature** option.

To speed up the programming process you can uncheck the **Check Erasure** check box.
In this case there will be no verification of the correctness of the FLASH erasure.

The **Preserve EEPROM** checkbox allows preserving the contents of the EEPROM during chip erasure.

To speed up the programming process you also can uncheck the **Verify** check box.
In this case there will be no verification of the correctness of the FLASH and EEPROM programming.

For erasing a chip's FLASH and EEPROM you must select the **Program|Erase** menu command.
After erasure the chip's FLASH and EEPROM are automatically blank checked.
For simple blank checking you must use the **Program|Blank Check** menu command.
If you wish to program the FLASH with the contents of the FLASH buffer you must use the **Program|FLASH** menu command.
For programming the EEPROM you must use the **Program|EEPROM** menu command.
After programming the FLASH and EEPROM are automatically verified.

To program the Lock, respectively the Fuse Bit(s) you must use the **Program|Fuse Bit(s)**, respectively **Program|Lock Bits** menu commands.

The **Program|All** menu command allows to automatically:
- Erase the chip
- FLASH and EEPROM blank check
- Program and verify the FLASH
- Program and verify the EEPROM
- Program the Fuse and Lock Bits.

If you wish to read the contents of the chip's FLASH, respectively EEPROM, you must use the **Read|FLASH**, respectively **Read|EEPROM** menu commands.
For reading the chip's signature you must use the **Read|Chip Signature** menu command.
To read the Lock, respectively the Fuse Bits you must use the **Read|Lock Bits**, respectively **Read|Fuse Bits** menu commands.

For some devices there's also the **Read|Calibration Byte(s)** option available.
It allows reading the value of the calibration bytes of the chip's internal RC oscillator.

If the programmer is an Atmel STK500, AVRISP, AVRISP MkII or AVRProg (AVR910 application note), then an additional menu command is present: **Read|Programmer's Firmware Version**. It allows reading the major and minor versions of the above mentioned programmers' firmware.

For comparing the contents of the chip's FLASH, respectively EEPROM, with the corresponding memory buffer, you must use the **Compare|FLASH**, respectively **Compare|EEPROM** menu commands.

For exiting the Programmer and returning to the CodeVisionAVR IDE you must use the **File|Close** menu command.

### 3.4.3 The Serial Communication Terminal

The **Terminal** is intended for debugging embedded systems, which employ serial communication (RS232, RS422, RS485).
The Terminal is invoked using the **Tools|Terminal** menu command or the button on the toolbar.

The characters can be displayed in ASCII or hexadecimal format. The display mode can be toggled using the **Hex/ASCII** button.
The received characters can be saved to a file using the **Rx File** button.

Any characters typed in the Terminal window will be transmitted through the PC serial port.
The entered characters can be deleted using the **Backspace** key.
By pressing the **Send** button, the Terminal will transmit a character whose hexadecimal ASCII code value is specified in the Hex Code edit box.
By pressing the **Tx File** button, the contents of a file can be transmitted through the serial port.

By pressing the **Reset** button, the AVR chip on the STK200+/300, VTEC-ISP, DT006, ATCPU or Mega2000 development board is reseted.

At the bottom of the Terminal window there is a status bar in which are displayed the:
- computer's communication port;
- communication parameters;
- handshaking mode;
- received characters display mode;
- type of emulated terminal;
- the state of the transmitted characters echo setting.

## 3.4.4 The LCD Vision Font and Image Editor/Converter

LCD Vision is an application designed for creating, editing font and image data, and exporting it in form of C source code, compatible with the CodeVisionAVR **Graphic LCD Functions**.
Fonts can be created from scratch or imported from the installed system fonts.
Images can be also created from scratch or imported from popular graphic formats like: BMP, JPG, GIF, PNG, ICO, WMF, EMF.

LCD Vision is invoked using the **Tools|LCD Vision** menu command or the button on the toolbar.

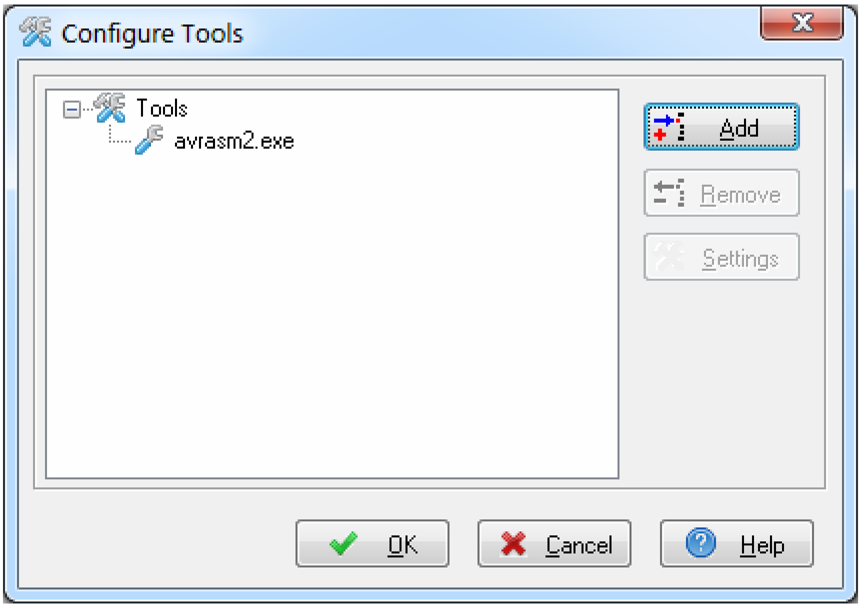
**Note:** The LCD Vision editor/converter can be used only with an Advanced CodeVisionAVR license.

## 3.4.5 Executing User Programs

User programs are executed by selecting the corresponding command from the **Tools** menu.
You must previously add the Program's name to the menu.

## 3.4.6 Configuring the Tools Menu

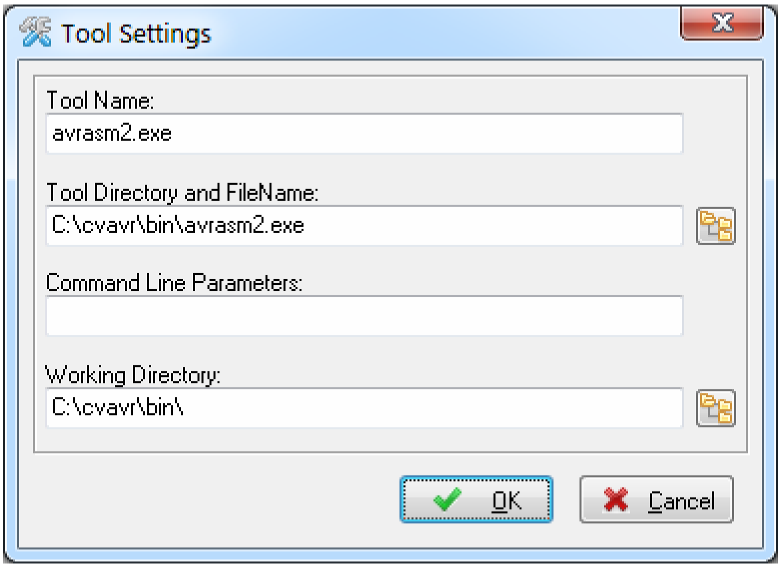
You can add or remove User Programs from the **Tools** menu by using the **Tools|Configure** menu command.
A **Configure Tools** dialog window, with a list of User Programs, will open.




Using the **Add** button you can add a Program to the **Tools** menu.
Using the **Remove** button you can remove a Program from the **Tools** menu.

Using the **Settings** button you can modify the:

- Tool Menu Name
- Tool Directory and File Name
- Command Line Parameters
- Working Directory of a selected Program from the list




Changes can be saved, respectively canceled, using the **OK**, respectively **Cancel** buttons.

The command line can accept the following parameters:

- %bc substitutes the Active Build Configuration: DEBUG or RELEASE
- %P substitutes the full project path
- %p substitutes the project name without path
- %h substitutes the name of the .hex file created by the compiler
- %e substitutes the name of the .eep file created by the compiler
- %f<project_file_number> substitutes the project's source file name without path
- %F<project_file_number> substitutes the project's source file name with full path.

## 3.5 IDE Settings

The CodeVisionAVR IDE is configured using the **View** and **Settings** menus.

### 3.5.1 The View Menu

The following settings can be configured using the **View** menu command:

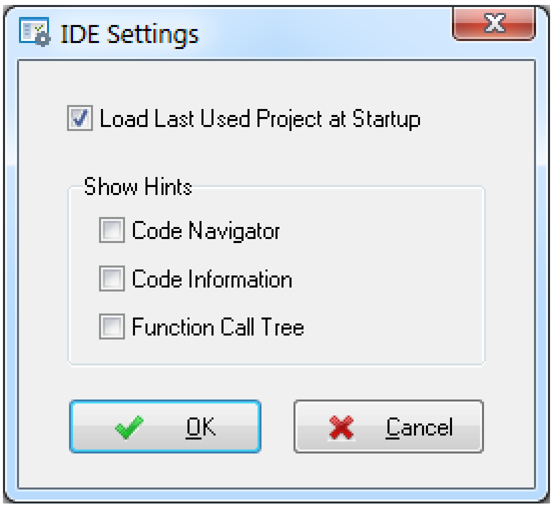
- The **View|Visible Non-Printable Characters** option allows to turn on or off the displaying of non-printable characters in the Editor window. The toolbar button can be also used for this purpose.
- The **View|Toolbar** option allows to turn on or off the displaying of the various toolbars containing the IDE command buttons;
- The **View|Code Navigator/Code Information/Code Templates/Clipboard History** option allows to turn on or off the displaying of the **Navigator, Code Templates** and **Clipboard History** window at the left of the **Editor** window. The toolbar button can be also used for this purpose;
- The **View|Messages** option allows to turn on or off the displaying of the **Message** window located under the **Editor** window. The toolbar button can be also used for this purpose;
- The **View|Information Window after Compile/Build** option allows to turn on or off the displaying of the **Information** window after the **Compile** or **Build** processes.

### 3.5.2 General IDE Settings

Some general IDE settings can be specified using the **Settings|IDE** menu or the toolbar button. These settings are:

- Load Last Used Project at Startup
- Show Hint for the **Code Navigator** window
- Show Hint for the **Code Information** window
- Show Hint for the **Function Call Tree** window.

The settings can be enabled or disabled by checking or unchecking the appropriate check boxes:




Changes can be saved, respectively canceled, using the **OK**, respectively **Cancel** buttons.

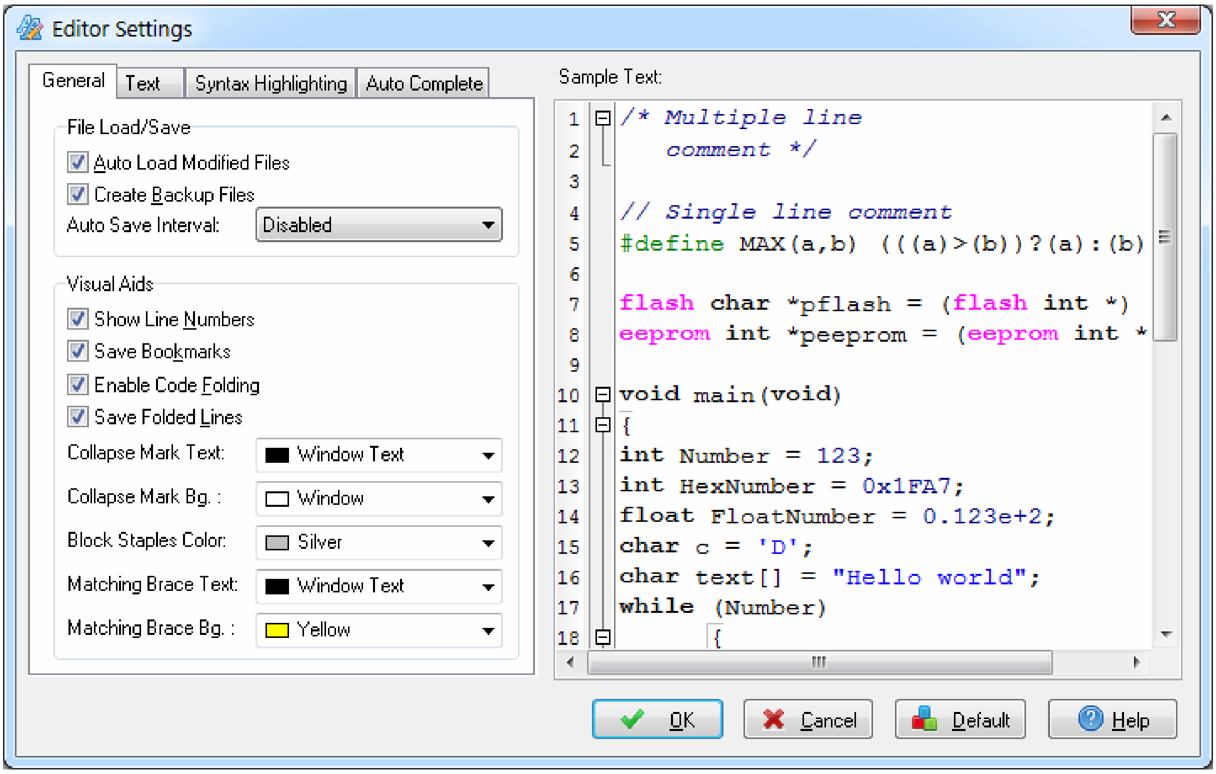
## 3.5.3 Configuring the Editor

The Editor can be configured using the **Settings|Editor** menu command.

The Editor configuration changes can be saved, respectively canceled, using the **OK**, respectively **Cancel** buttons.
By pressing the **Default** button the default Editor settings are restored.

### 3.5.3.1 General Editor Settings

The following groups of Editor settings can be established by clicking on the **General** tab:




- **File Load/Save** settings;
- **Visual Aids** settings.

The **File Load/Save** settings allow for the following options to be set:

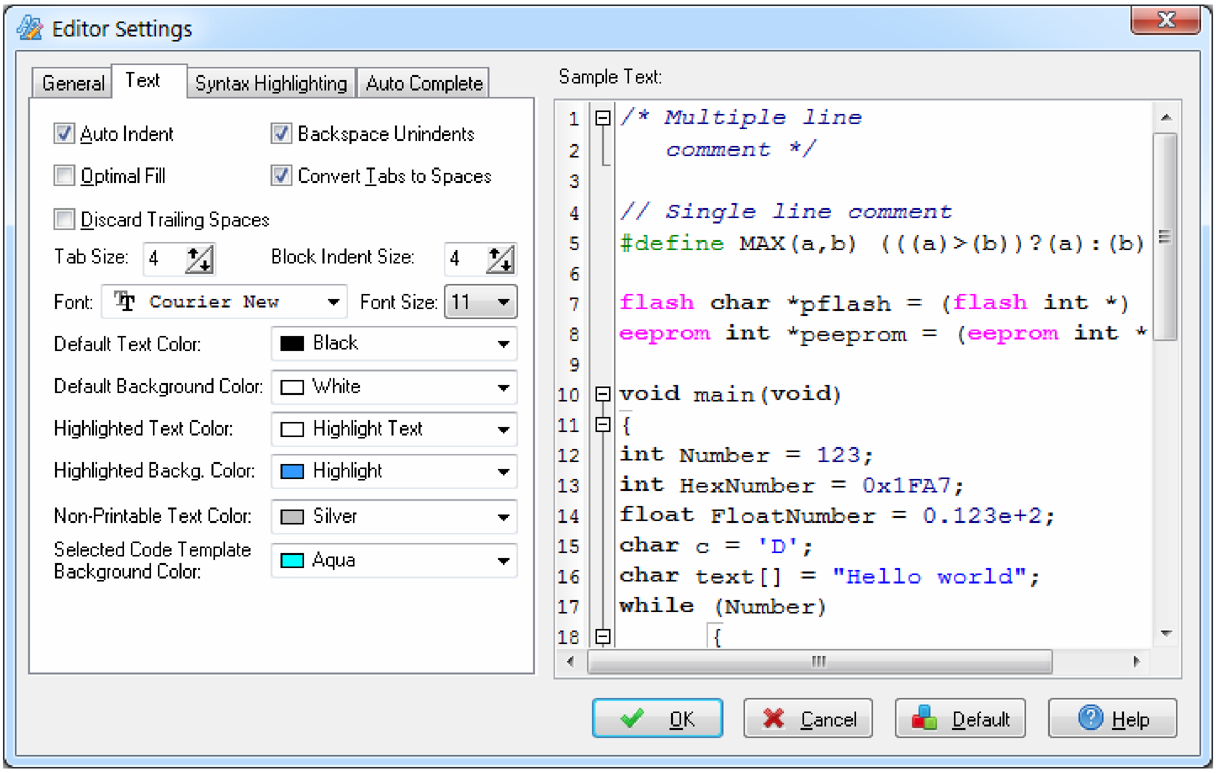
- **Auto Load Modified Files** enables or disables the automatic reloading, in the CodeVisionAVR Editor, of source files that were externally modified by some other program (another editor for example). If this option is disabled, the user will be prompted before the modified file will be reloaded in the Editor.
- **Create Backup Files** enables or disables the creation of backup copies of the files modified in the Editor. Backup copies will have the ~ character appended to their extension.
- **Auto Save Interval** specifies at which time interval all the modified source files will be automatically saved by the Editor.

The **Visual Aids** settings allow for the following options to be set:

- **Show Line Numbers** enables or disables the displaying of line numbers on the gutter located on the left side of the Editor windows;
- **Save Bookmarks** enables or disables saving the bookmarks set in each edited source file;
- **Enable Code Folding** enables or disables displaying of staples on the left side of code blocks delimited by the { } characters. If this option is enabled, block collapse/expansion marks will be also displayed on the gutter located on the left side of the Editor window.
- **Save Folded Lines** enables or disables saving the state of the folded blocks of lines for each edited source file;
- **Collapse Mark Text** specifies the text foreground color of the collapse marks;
- **Collapse Mark Bg.** specifies the text background color of the collapse marks;
- **Block Staples Color** specifies the foreground color of the folding block staples. The background color of the staples will be the same as the **Default Background Color** of the Editor window.
- **Matching Brace Text** specifies the text foreground color of the matching braces, which are automatically highlighted by the Editor when the user places the cursor before them;
- **Matching Brace Bg.** specifies the text background color of the highlighted matching braces.

## 3.5.3.2 Editor Text Settings

The following Editor settings can be established by clicking on the **Text** tab:

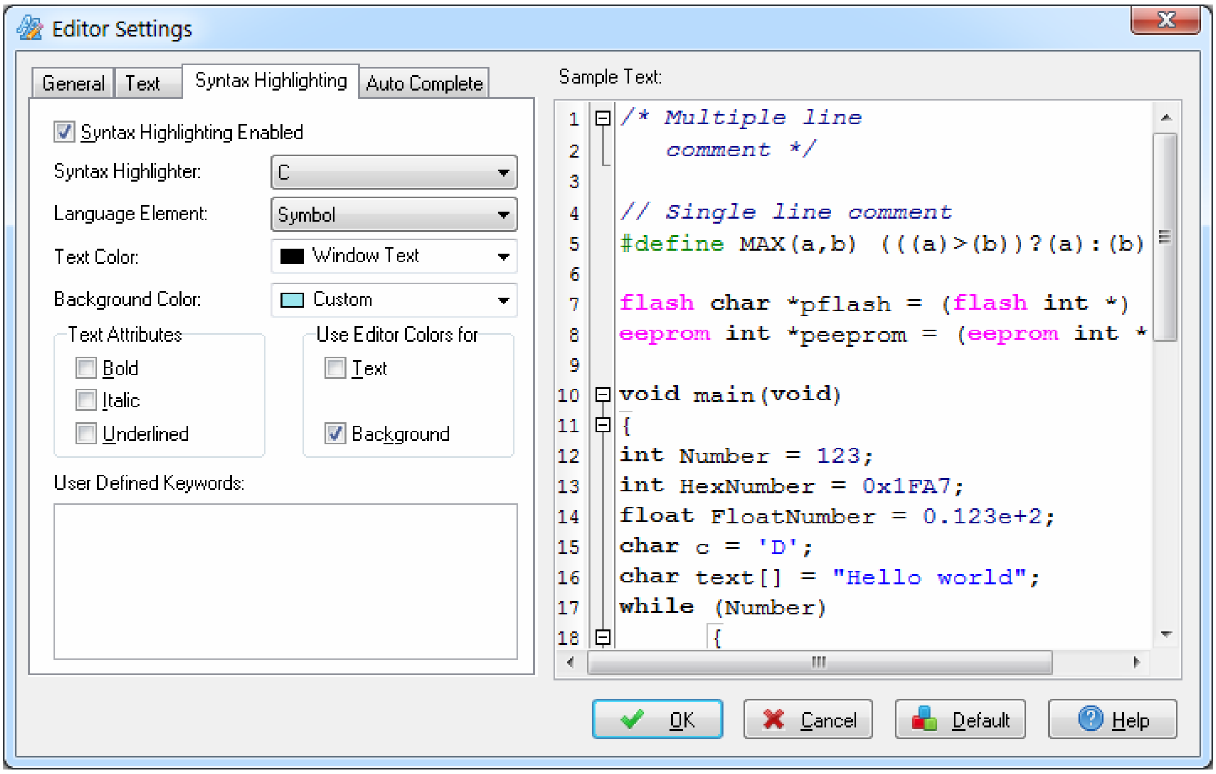



- **Auto Indent** enables or disables text auto indenting during file editing;
- **Backspace Unindents** when enabled, sets the Editor to align the insertion point to the previous indentation level (outdents it) when the user presses the Backspace key, if the cursor is on the first nonblank character of a line. If this option is disabled, pressing the Backspace key just deletes the character located on the left of the cursor.
- **Optimal Fill** enables or disables the beginning of every auto indented line with the minimum number of characters possible, using tabs and spaces as necessary;
- **Convert Tabs to Spaces** enables or disables the automatic replacement, while typing, of tab characters with the appropriate number of spaces, as specified by the **Tab Size** option;
- **Discard Trailing Spaces** enables or disables the automatic deletion from the end of each line, of spaces that are not followed by text,
- **Tab Size** specifies the number of spaces the Editor cursor is moved when the user presses the Tab key;
- **Block Indent Size** specifies the number of spaces the Editor indents a marked block of text;
- **Font** specifies the font type used by the Editorl;
- **Font Size** specifies the font size used by the Editorl;
- **Default Text Color** specifies the foreground color of the default (normal) text in the Editor and Terminal windows;
- **Default Background Color** specifies the background color of the default (normal) text in the Editor and Terminal windows;
- **Highlighted Text Color** specifies the foreground color of the text highlighted by the user in the Editor window;
- **Highlighted Background Color** specifies the background color of the text highlighted by the user in the Editor window;

- **Non-Printable Text Color** specifies the foreground color of the non-printable character markers displayed in the Editor window when the **View|Visible Non-Printable Characters** menu option is checked. The background color of the non-printable character markers will be the same as the **Default Background Color** of the Editor window.

### 3.5.3.3 Syntax Highlighting Settings

The following Editor settings can be established by clicking on the **Syntax Highlighting** tab:




- **Syntax Highlighting Enabled** enables or disables source file syntax highlighting;
- **Syntax Highlighter** list box selects the programming language for which the syntax highlighting settings will be applied. The CodeVisionAVR Editor supports syntax highlighting for the C and Atmel AVR Assembler programming languages.
- **Language Element** list box selects the element for which the text colors and attributes will be set;
- **Text Color** specifies the text foreground color for the above selected **Language Element**;
- **Background Color** specifies the text background color for the above selected **Language Element**;
- **Text Attributes** specifies how the text is displayed for the above selected **Language Element**. Text attributes can be combined by appropriately checking the **Bold**, **Italic** and **Underlined** check boxes. The displayed font will be the one selected in the **Text|Font** settings.

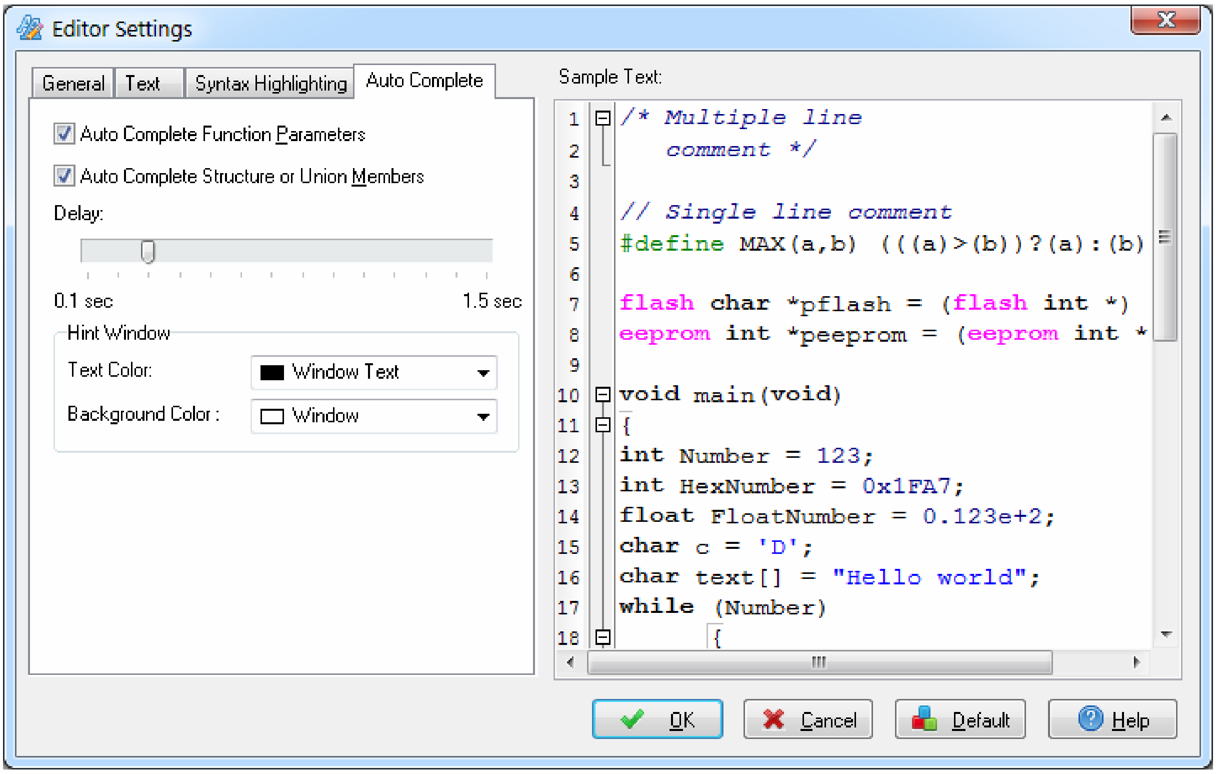
The **Text**, respectively **Background**, check boxes from the **Use Editor Colors** group box, when checked will set the foreground, respectively background, text colors for the selected **Language Element** to the default ones specified in the **Text|Default Text Color**, respectively **Text|Default Background Color** settings.

The **User Defined Keywords** list can contain additional keywords for which syntax highlighting is required. Their text colors and attributes can be specified when selecting the **Language Element** as *User defined keyword*.

The results of the applied syntax highlighting settings can be viewed in the **Sample Text** portion of the window.

### 3.5.3.4 Auto Complete Settings

The following Editor settings can be established by clicking on the **Auto Complete** tab:




- **Auto Complete Function Parameters** enables or disables displaying a pop-up hint window with the function parameters declaration, after the user writes the function name followed by a '(' auto completion triggering character. The function parameter auto completing works only for the functions defined in the currently edited source file.
- **Auto Complete Structure or Union Members** enables or disables displaying a pop-up hint window with the structure/union members list, after the user writes the structure/union or pointer to structure/union name followed by the '**.**' or '**->**' auto completion triggering characters. The structure or union members auto completion works only for global structures/unions defined in the currently edited source file and after a **Project|Compile** or **Project|Build** was performed.

The **Delay** slider specifies the time delay that must elapse between entering the auto completion triggering characters and the displaying of the pop-up hint window.  If the user writes any other character before this time delay, no pop-up hint window will show.

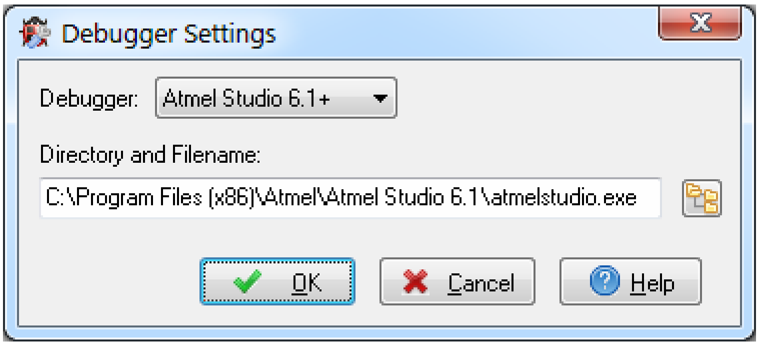
The **Hint Window** group box allows setting the **Text** and **Background Colors** of the auto complete pop-up hint window.

These colors will be also applied to the character grid pop-up hint window that is invoked using the **Edit|Insert Special Characters** menu, the **Insert Special Characters** right-click pop-up menu or by pressing the **Ctrl+.** keys.

## 3.5.4 Setting the Debugger Path

The CodeVisionAVR C Compiler is designed to work in conjunction with the  AVR Studio 4.19 and Atmel Studio 6.1 or later debuggers.

Before you can invoke the debugger, you must first specify its location and file name using the **Settings|Debugger** menu command.




Pressing the  button opens a dialog window that allows selecting the debugger's directory and filename.

Changes can be saved, respectively canceled, using the **OK**, respectively **Cancel** buttons.

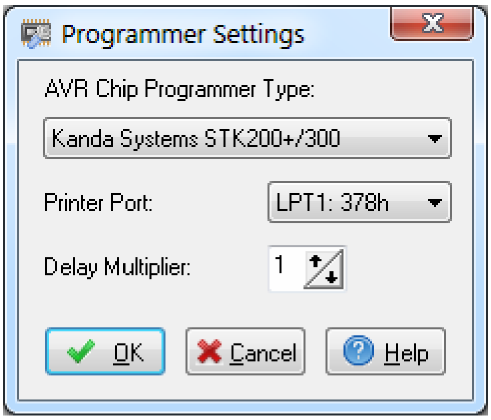
## 3.5.5 AVR Chip Programmer Setup

Using the **Settings|Programmer** menu command, you can select the type of the in-system programmer that is used, and the computer's port to which the programmer is connected.
The current version of CodeVisionAVR supports the following in-system programmers:

- Kanda Systems STK200+ and STK300
- Atmel STK500 and AVRISP (serial connection)
- Atmel AVRISP MkII (USB connection)
- Atmel AVR Dragon (USB connection)
- Atmel JTAGICE MkII (USB connection)
- Atmel JTAGICE 3 (USB connection)
- Atmel AVRProg (AVR910 application note)
- Dontronics DT006
- Vogel Elektronik VTEC-ISP
- Futurlec JRAVR
- MicroTronics ATCPU and Mega2000

The STK200+, STK300, DT006, VTEC-ISP, JRAVR, ATCPU and Mega2000 in-system programmers use the parallel printer port.
The following choices are available through the **Printer Port** radio group box:

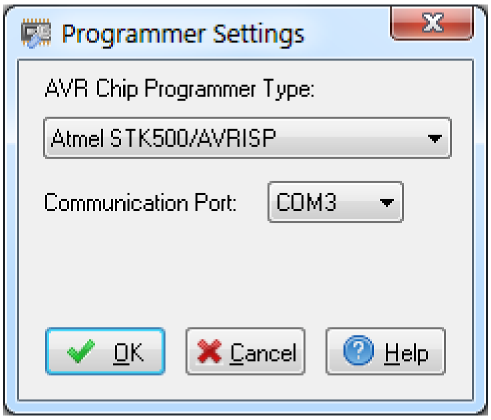
- LPT1, at base address 378h;
- LPT2, at base address 278h;
- LPT3, at base address 3BCh.




The **Delay Multiplier** value can be increased in case of programming problems on very fast machines. Of course this will increase overall programming time.

The **Atmega169 CKDIV8 Fuse Warning** check box, if checked, will enable the generation of a warning that further low voltage serial programming will be impossible for the *Atmega169 Engineering Samples*, if the CKDIV8 fuse will be programmed to 0.
For usual Atmega169 chips this check box must be left unchecked.

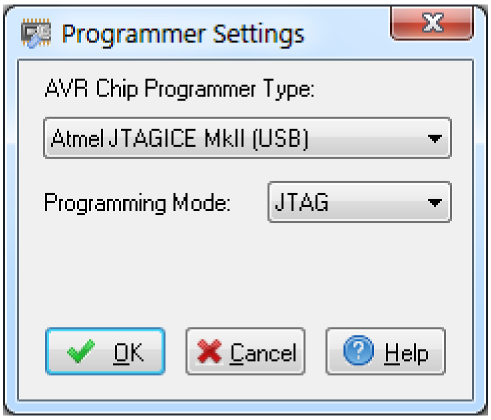
The STK500, AVRISP and AVRProg programmers use the RS232C serial communication port, which can be specified using the **Communication Port** list box.




The Atmel AVRISP MkII, AVR Dragon, JTAGICE MkII and JTAGICE 3 use the USB connection for communication with the PC.
Usage of these programmers requires the Atmel's AVR Studio 5.1 or later software to be installed on the PC.
The Atmel AVR Dragon, JTAGICE MkII and JTAGICE 3 can use two programming modes:

- JTAG
- ISP

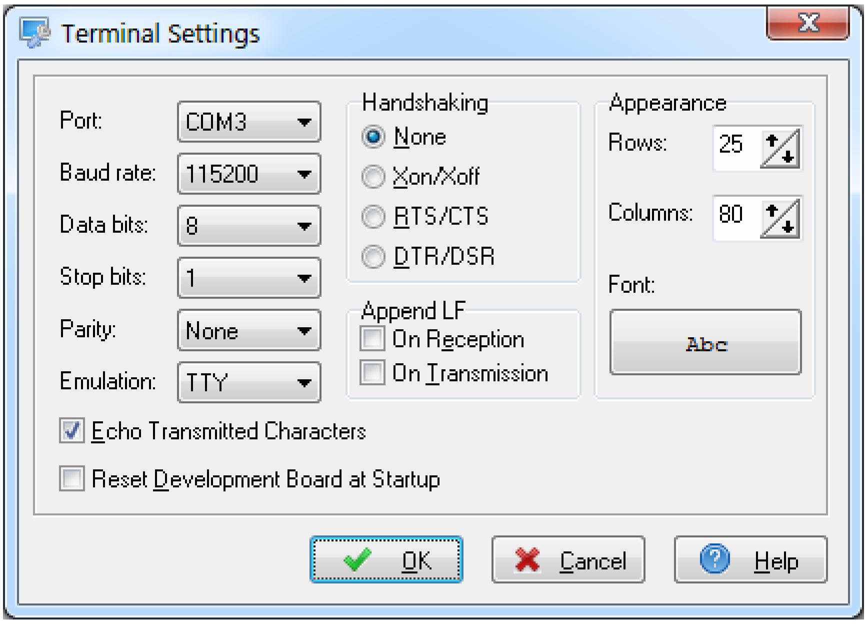
These can be selected using the **Programming Mode** list box:




Changes can be saved, respectively canceled, using the **OK**, respectively **Cancel** buttons.

## 3.5.6 Serial Communication Terminal Setup

The serial communication **Terminal** is configured using the **Settings|Terminal** menu command.




In the **Terminal Settings** window you can select the:

- computer's communication port used by the Terminal: COM1 to COM6;
- Baud rate used for communication: 110 to 115200;
- number of data bits used in reception and transmission: 5 to 8;
- number of stop bits used in reception and transmission: 1, 1.5 or 2;
- parity used in reception and transmission: None, Odd, Even, Mark or Space;
- type of emulated terminal: TTY, VT52 or VT100;
- type of handshaking used in communication: None, Hardware (CTS or DTR) or Software (XON/XOFF);
- possibility to append LF characters after CR characters on reception and transmission;
- enabling or disabling the echoing of the transmitted characters
- number of character **Rows** and **Columns** in the Terminal window
- **Font** type used for displaying characters in the Terminal window.

The **Reset Development Board at Startup** option, if enabled, allows to issue a chip reset when the Terminal is started, if a chip programmer is connected to the AVR microcontroller.

Changes can be saved, respectively canceled, using the **OK**, respectively **Cancel** buttons.

## 3.6 Accessing the Help

The CodeVisionAVR help system is accessed by invoking the **Help|Help** menu command or by pressing the toolbar button.

## 3.7 Connecting to HP InfoTech's Web Site

The **Help|HP InfoTech on the Web** menu command or the toolbar button opens the default web browser and connects to HP InfoTech's web site http://www.hpinfotech.com

## 3.8 Quitting the CodeVisionAVR IDE

To quit working with the CodeVisionAVR IDE you must select the **File|Exit** menu command.
If some source files were modified and were not saved yet, you will be prompted if you want to do that.

# 4. CodeVisionAVR C Compiler Reference

This section describes the general syntax rules for the CodeVisionAVR C compiler.
Only specific aspects regarding the implementation of the C language by this compiler are exposed.
This help is not intended to teach you the C language; you can use any good programming book to do that.
You must also consult the appropriate AVR data sheets from Atmel.

## 4.1 The C Preprocessor

The C Preprocessor directives allows you to:

- include text from other files, such as header files containing library and user function prototypes
- define macros that reduce programming effort and improve the legibility of the source code
- set up conditional compilation for debugging purposes and to improve program portability
- issue compiler specific directives

The Preprocessor output is saved in a text file with the same name as the source, but with the **.i** extension.

The **#include** directive may be used to include another file in your source.
You may nest as many as 300 **#include** files.
Example:

```
/* File will be looked for in the /inc directory of the compiler. */
#include <file_name>
```

or

```
/* File will be looked for in the current project directory.
   If it's not located there, then it will be included from
   the /inc directory of the compiler. */
#include "file_name"
```

The **#define** directive may be used to define a macro.
Example:

```
#define ALFA 0xff
```

This statement defines the symbol 'ALFA' to the value 0xff.
The C preprocessor will replace 'ALFA' with 0xff in the source text before compiling.

Macros can also have parameters. The preprocessor will replace the macro with its expansion and the formal parameters with the real ones.
Example:

```
#define SUM(a,b) a+b
/* the following code sequence will be replaced with int i=2+3; */
int i=SUM(2,3);
```

When defining macros you can use the **#** operator to convert the macro parameter to a character string.
Example:

```
#define PRINT_MESSAGE(t)  printf(#t)

/* ...... */
/* the following code sequence will be replaced with printf("Hello"); */
PRINT_MESSAGE(Hello);
```

Two parameters can be concatenated using the **##** operator.
Example:

```
#define ALFA(a,b) a ## b

/* the following code sequence will be replaced with char xy=1; */
char ALFA(x,y)=1;
```

A macro definition can be extended to a new line by using **** .
Example:

```
#define MESSAGE "This is a very \
long text..."
```

A macro can be undefined using the **#undef** directive.
Example:

```
#undef ALFA
```

The **#ifdef**, **#ifndef**, **#else** and **#endif** directives may be used for conditional compilation.
The syntax is:

```
#ifdef macro_name
[set of statements 1]
#else
[set of statements 2]
#endif
```

If 'alfa' is a defined macro name, then the **#ifdef** expression evaluates to true and the set of statements 1 will be compiled.
Otherwise the set of statements 2 will be compiled.
The **#else** and set of statements 2 are optional.
If 'alfa' is not defined, the **#ifndef** expression evaluates to true.
The rest of the syntax is the same as that for **#ifdef**.

The **#if**, **#elif**, **#else** and **#endif** directives may be also used for conditional compilation.

```
#if expression1
[set of statements 1]
#elif expression2
[set of statements 2]
#else
[set of statements 3]
#endif
```

If **expression1** evaluates to true, the set of statements 1 will be compiled.
If **expression2** evaluates to true, the set of statements 2 will be compiled.
Otherwise the set of statements 3 will be compiled.
The **#else** and set of statements 3 are optional.

There are the following predefined macros:

**__CODEVISIONAVR__** the version and revision of the compiler represented as an integer, example for V2.05.2 this will be 2052
**__STDC__** equals to 1
**__LINE__** the current line number of the compiled file
**__FILE__** the current compiled file
**__TIME__** the current time in *hh:mm:ss* format
**__UNIX_TIME__** unsigned long that represents the number of seconds elapsed since midnight UTC of 1 January 1970, not counting leap seconds
**__DATE__** the current date in *mmm dd yyyy* format
**__BUILD__** the build number
**_CHIP_ATXXXXX_** where ATXXXXX is the chip type, in uppercase letters, specified in the **Project|Configure|C Compiler|Code Generation|Chip** option
**_MCU_CLOCK_FREQUENCY_** the AVR clock frequency specified in the **Project|Configure|C Compiler|Code Generation|Clock** option, expressed as an unsigned long integer in Hz
**_MODEL_TINY_** if the program is compiled using the TINY memory model
**_MODEL_SMALL_** if the program is compiled using the SMALL memory model
**_MODEL_MEDIUM_** if the program is compiled using the MEDIUM memory model
**_MODEL_LARGE_** if the program is compiled using the LARGE memory model
**_OPTIMIZE_SIZE_** if the program is compiled with optimization for size (**Project|Configure|C Compiler|Code Generation|Optimize for: Size** option or **#pragma optsize+**)
**_OPTIMIZE_SPEED_** if the program is compiled with optimization for speed (**Project|Configure|C Compiler|Code Generation|Optimize for: Speed** option or **#pragma optsize-**)
**_WARNINGS_ON_** if the warnings are enabled by the **Project|Configure|C Compiler|Messages|Enable Warnings** option or **#pragma warn+**
**_WARNINGS_OFF_** if the warnings are disabled by the **Project|Configure|C Compiler|Messages|Enable Warnings** option or **#pragma warn-**
**_PROMOTE_CHAR_TO_INT_ON_** if the automatic ANSI char to int type promotion is enabled by the **Project|Configure|C Compiler|Code Generation|Promote char to int** option or **#pragma promotechar+**
**_PROMOTE_CHAR_TO_INT_OFF_** if the automatic ANSI char to int type promotion is disabled by the **Project|Configure|C Compiler|Code Generation|Promote char to int** option or **#pragma promotechar-**
**_AVR8L_CORE_** signals that the program is compiled using the reduced core instruction set, used in chips like ATtiny10, ATtiny20, ATtiny40. No ADIW, SBIW, LDD and STD instructions are generated in this case.
**_ENHANCED_CORE_** if the program is compiled using the enhanced core instructions available in the new ATmega chips
**_ENHANCED_FUNC_PAR_PASSING_** if the program is compiled with the **Project|Configure|C Compiler|Code Generation|Enhanced Function Parameter Passing** option enabled
**_ATXMEGA_DEVICE_** signals that the program is compiled for an XMEGA chip type
**_EXTERNAL_STARTUP_** signals that the **Project|Configure|C Compiler|Code Generation|Use an External Startup Initialization File** option is enabled
**_IO_BITS_DEFINITIONS_** if the **Project|Configure|C Compiler|Code Generation|Preprocessor|Include I/O Registers Bits Definitions** option is enbaled
**_SRAM_START_** the start address of on-chip SRAM
**_SRAM_END_** the end address of the SRAM accessible to the compiled program, including the eventual external memory
**_DSTACK_START_** the data stack starting address
**_DSTACK_END_** the last address of SRAM allocated for the data stack
**_DSTACK_SIZE_** the data stack size specified in the **Project|Configure|C Compiler|Code Generation|Data Stack Size** option
**_HEAP_START_** the heap starting address
**_HEAP_SIZE_** the heap size specified in the **Project|Configure|C Compiler|Code Generation|Heap Size** option

**_UNSIGNED_CHAR_** if the **Project|Configure|C Compiler|Code Generation|char is unsigned** compiler option is enabled or **#pragma uchar+** is used
**_8BIT_ENUMS_** if the **Project|Configure|C Compiler|Code Generation|8 bit enums** compiler option is enabled or **#pragma 8bit_enums+** is used
**_ATXMEGA_USART_** specifies which XMEGA chip USART is used by the **getchar** and **putchar** Standard C Input/Output Functions
**_ATXMEGA_SPI_** specifies which XMEGA chip SPI controller is used by the SPI Functions
**_ATXMEGA_SPI_PORT_** specifies which XMEGA chip I/O port is used by the SPI controller.

The **#line** directive can be used to modify the predefined **__LINE__** and **__FILE__** macros.
The syntax is:

```
#line integer_constant ["file_name"]
```

Example:

```
/* This will set __LINE__ to 50 and
   __FILE__ to "file2.c" */
#line 50 "file2.c"

/* This will set __LINE__ to 100 */
#line 100
```

The **#error** directive can be used to stop compilation and display an error message.
The syntax is:

```
#error error_message
```

Example:

```
#error This is an error!
```

The **#warning** directive can be used to display a warning message.
The syntax is:

```
#warning warning_message
```

Example:

```
#warning This is a warning!
```

The **#message** directive can be used to display a message dialog window in the CodeVisionAVR IDE.
The syntax is:

```
#message general_message
```

Example:

```
#message Hello world
```

## 4.2 Comments

The character string **"/*"** marks the beginning of a comment.
The end of the comment is marked with **"*/"**.
Example:

```
/* This is a comment  */
/* This is a
    multiple line comment */
```

One-line comments may be also defined by using the string **"//"**.
Example:

```
// This is also a comment
```

Nested comments are not allowed.

## 4.3 Reserved Keywords

Following is a list of keywords reserved by the compiler.
These can not be used as identifier names.

```
__eeprom
__flash
__interrupt
__task
_Bool
_Bit
break
bit
bool
case
char
const
continue
default
defined
do
double
eeprom
else
enum
extern
flash
float
for
goto
if
inline
int
interrupt
long
register
return
short
signed
sizeof
sfrb
sfrw
static
struct
switch
typedef
union
unsigned
void
volatile
while
```

## 4.4 Identifiers

An identifier is the name you give to a variable, function, label or other object.
An identifier can contain letters (A...Z, a...z) and digits (0...9), as well as the underscore character (_).
However an identifier can only start with a letter or an underscore.
Case is significant; i.e. **variable1** is not the same as **Variable1**.
Identifiers can have up to 64 characters.

## 4.5 Data Types

The following table lists all the data types supported by the CodeVisionAVR C compiler, their range of possible values and their size:

| Type | Size (Bits) | Range |
|---|---|---|
| bit, _Bit | 1 | 0 , 1 |
| bool, _Bool | 8 | 0 , 1 |
| char | 8 | -128 to 127 |
| unsigned char | 8 | 0 to 255 |
| signed char | 8 | -128 to 127 |
| int | 16 | -32768 to 32767 |
| short int | 16 | -32768 to 32767 |
| unsigned int | 16 | 0 to 65535 |
| signed int | 16 | -32768 to 32767 |
| long int | 32 | -2147483648 to 2147483647 |
| unsigned long int | 32 | 0 to 4294967295 |
| signed long int | 32 | -2147483648 to 2147483647 |
| float | 32 | ±1.175e-38 to ±3.402e38 |
| double | 32 | ±1.175e-38 to ±3.402e38 |

The **bit** or **_Bit** data types are not allowed as the type of an array element, structure/union member, function parameter or return value.

In order to use the **bool** data type, the **stdbool.h** header file must be #included in the source files where this data type is referenced.

If the **Project|Configure|C Compiler|Code Generation|char is unsigned** option is checked or **#pragma uchar+** is used, then **char** has by default the range 0..255.

## 4.6 Constants

Integer or long integer constants may be written in decimal form (e.g. 1234), in binary form with **0b** prefix (e.g. 0b101001), in hexadecimal form with **0x** prefix (e.g. 0xff) or in octal form with **0**-prefix (e.g. 0777).
Unsigned integer constants may have the suffix **U** (e.g. 10000U).
Long integer constants may have the suffix **L** (e.g. 99L).
Unsigned long integer constants may have the suffix **UL** (e.g. 99UL).
Floating point constants may have the suffix **F** (e.g. 1.234F).
Character constants must be enclosed in single quotation marks. E.g. 'a'.
Literal string constants must be enclosed in double quotation marks. E.g. "Hello world".

Constant expressions are automatically evaluated during compilation.

Program constants can be declared as global (accessible to all the functions in the program) or local (accessible only inside the function they are declared).
The constant declarations syntax is similar to that of variables, but preceded by the **const** keyword:

```
const <type definition> <identifier> = constant expression;
```

Example:

```
/* Global constants declaration */
const char char_constant='a';
const int b=1234+5;
const long long_int_constant1=99L;
const long long_int_constant2=0x10000000;
const float pi=3.14;

void main(void) {
/* Local constants declaration */
const long f=22222222;
const float x=1.5;

}
```

Constants can be grouped in arrays, which can have up to 64 dimensions.
The first element of an array has always the index 0.
Example:

```
const char string_constant2[]="This is a string constant";
const int abc[3]={1,2,3};
/* The first two elements will be 1 and 2,
   the rest will be 0 */
const int  integer_array2[10]={1,2};
/* multidimensional array */
const int  multidim_array[2][3]={{1,2,3},{4,5,6}};
```

If the **Project|Configure|C Compiler|Code Generation|Store Global Constants in FLASH Memory** option is enabled, global constants that were declared using the **const** keyword will be placed by the compiler in FLASH memory.
If the above option is not enabled, global constants declared using the **const** keyword will be located in RAM memory.
Local constants will be always placed in RAM memory.

The **flash** or **__flash** keywords can be used to specify that a constant must be placed in FLASH memory, no matter what is the state of the **Store Global Constants in FLASH Memory** option:

```
flash <type definition> <identifier> = constant expression;
__flash <type definition> <identifier> = constant expression;
```

Example:

```
flash int  integer_constant=1234+5;
flash char char_constant='a';
flash long long_int_constant1=99L;
flash long long_int_constant2=0x10000000;
flash int  integer_array1[]={1,2,3};
flash char string_constant1[]="This is a string constant located in FLASH";
```

The constant literal char strings, enclosed in double quotation marks, that are passed as function arguments, are stored in the memory type pointed by the pointer used as function parameter.
Example:

```
/* This function displays a string located in RAM. */
void display_ram(char *s) {
/* .......  */

}

/* This function displays a string located in FLASH. */
void display_flash(flash char *s) {
/* .......  */

}

/* This function displays a string located in EEPROM. */
void display_eeprom(eeprom char *s) {
/* .......  */

}

void main(void) {
/* The literal string "Hello world" will be placed
   by the compiler in FLASH memory and copied at program
   startup to RAM, so it can be accessed by the pointer
   to RAM used as function parameter.
   The code efficiency is low, because both FLASH and
   RAM memories are used for the string storage. */
display_ram("Hello world");

/* The literal string "Hello world" will be placed
   by the compiler in FLASH memory only, good code
   efficiency beeing achieved. */
display_flash("Hello world");

/* The literal string "Hello world" will be placed
   by the compiler in EEPROM memory only.
   The code efficiency is very good because no
   FLASH memory will be allocated for the string. */
display_eeprom("Hello world");

while(1);
}
```

## 4.7 Variables

Program variables can be global (accessible to all the functions in the program) or local (accessible only inside the function they are declared).
If not specifically initialized, the global variables are automatically set to 0 at program startup.
The local variables are not automatically initialized on function call.
The syntax is:

```
[<memory attribute>] [<storage modifier>] <type definition> <identifier>
                                                    [= constant expression];
```

Example:

```
/* Global variables declaration */
char a;
int  b;
/* and initialization */
long c=1111111;

void main(void) {
/* Local variables declaration */
char d;
int  e;
/* and initialization */
long f=22222222;
}
```

Variables can be grouped in arrays, which can have up to 64 dimensions.
The first element of an array has always the index 0.
If not specifically initialized, the elements of global variable arrays are automatically set to 0 at program startup.
Example:

```
/* All the elements of the array will be 0 */
int  global_array1[32];

/* Array is automatically initialized */
int  global_array2[]={1,2,3};
int  global_array3[4]={1,2,3,4};
char global_array4[]="This is a string";

/* Only the first 3 elements of the array are
   initialized, the rest 29 will be 0 */
int  global_array5[32]={1,2,3};

/* Multidimensional array */
int multidim_array[2][3]={{1,2,3},{4,5,6}};

void main(void) {
/* local array declaration */
int  local_array1[10];

/* local array declaration and initialization */
int  local_array2[3]={11,22,33};
char local_array3[7]="Hello";
}
```

Local variables that must conserve their values during different calls to a function must be declared as **static**. Example:

```
int alfa(void) {
/* declare and initialize the static variable */
static int n=1;
return n++;
}

void main(void) {
int i;

/* the function will return the value 1 */
i=alfa();

/* the function will return the value 2 */
i=alfa();
}
```

If not specifically initialized, **static** variables are automatically set to 0 at program startup.

Variables that are declared in other files must be preceded by the **extern** keyword.
Example:

```
extern int xyz;

/* now include the file which contains
   the variable xyz definition */
#include <file_xyz.h>
```

To instruct the compiler to allocate a variable to registers, the **register** modifier must be used.
Example:

```
register int abc;
```

The compiler may automatically allocate a variable to registers, even if this modifier is not used.

The **volatile** modifier must be used to warn the compiler that it may be subject to outside change during evaluation.
Example:

```
volatile int abc;
```

Variables declared as volatile will not be allocated to registers.

All the global variables, not allocated to registers, are stored in the **Global Variables** area of RAM.
All the local variables, not allocated to registers, are stored in dynamically allocated space in the **Data Stack** area of RAM.

If a global variable declaration is preceded by the **eeprom** or **__eeprom** memory attribute, the variable will be located in EEPROM.
Example:

```
eeprom float xyz=12.9;
__eeprom int w[5]={1,2,3,4,5};
```

## 4.7.1 Specifying the RAM and EEPROM Storage Address for Global Variables

Global variables can be stored at specific RAM and EEPROM locations at design-time using the @ operator.
Example:

```
/* the integer variable "a" is stored
   in RAM at address 80h */
int a @0x80;

/* the structure "alfa" is stored
   in RAM at address 90h */
struct s1 {
    int a;
    char c;
    } alfa @0x90;

/* the float variable "b" is stored
   in EEPROM at address 10h */
eeprom float b @0x10;

/* the structure "beta" is stored
   in EEPROM at address 20h */
eeprom struct s2 {
    int i;
    long j;
    } beta @0x20;
```

The following procedure must be used if a global variable, placed at a specific address using the @ operator, must be initialized during declaration:

```
/* the variable will be stored in RAM at address 0x182 */
float pi @0x182;
/* and it will be initialized with the value 3.14 */
float pi=3.14;

/* the variable will be stored in EEPROM at address 0x10 */
eeprom int abc @0x10;
/* and it will be initialized with the value 123 */
eeprom int abc=123;
```

## 4.7.2 Bit Variables

The global bit variables located in the GPIOR register(s) and R2 to R14 memory space.
These variables are declared using the **bit** or **_Bit** keywords.
The syntax is:

```
bit <identifier>;
```

Example:

```
/* declaration and initialization for an ATtiny2313 chip
   which has GPIOR0, GPIOR1 and GPIOR2 registers */
bit alfa=1; /* bit0 of GPIOR0 */
bit beta; /* bit1 of GPIOR0 */

void main(void)
{
if (alfa) beta=!beta;

/* ........ */
}
```

Memory allocation for the global bit variables is done, in the order of declaration, starting with bit 0 of GPIOR0, then bit 1 of GPIOR0 and so on, in ascending order.
After all the GPIOR registers are allocated, further bit variables are allocated in R2 up to R14.
If the chip does not have GPIOR registers, the allocation begins directly from register R2.

The size of the global bit variables allocated to the program can be specified in the **Project|Configure|C Compiler|Code Generation|Bit Variables Size** list box.
This size should be as low as possible, in order to free registers for allocation to other global variables.
If not specifically initialized, the global bit variables are automatically set to 0 at program startup.

The compiler allows also to declare up to 8 local bit variables which will be allocated in register R15.
Example:

```
void main(void)
{
bit alfa; /* bit 0 of R15 */
bit beta; /* bit 1 of R15 */
/* ........ */
}
```

In expression evaluation bit variables are automatically promoted to **unsigned char**.

As there is no support for the bit data type in the COFF object file format, the CodeVisionAVR compiler generates debugging information for the whole register where a bit variable is located.
Therefore when watching bit variables in the AVR Studio debugger, the value of the register is displayed instead of a single bit from it.
However it is quite simple to establish the value of the bit variable based on the register bit number allocated for it, which is displayed in the **Code Information** tab of the CodeVisionAVR IDE, and the register value displayed in hexadecimal in AVR Studio's **Watch** window.

## 4.7.3 Allocation of Variables to Registers

In order to fully take advantage of the AVR architecture and instruction set, the compiler allocates some of the program variables to chip registers.
The registers from R2 up to R14 can be allocated for global **bit** variables.
The register R15 can be allocated to local **bit** variables.

You may specify how many registers in the R2 to R14 range are allocated for global **bit** variables using the **Project|Configure|C Compiler|Code Generation|Bit Variables Size** list box. This value must be as low as required by the program.
If the **Project|Configure|C Compiler|Code Generation|Automatic Global Register Allocation** option is checked or the **#pragma regalloc+** compiler directive is used, the rest of registers in the R2 to R14 range, that aren't used for global **bit** variables, are allocated to **char** and **int** global variables and global pointers.
If the **Project|Configure|C Compiler|Code Generation|Smart Register Allocation** option is checked, the allocation of registers R2 to R14 (not used for bit variables) is performed in such a way that 16bit variables will be preferably located in even register pairs, thus favouring the usage of the enhanced core MOVW instruction for their access.
Otherwise the allocation is performed in order of variable declaration until the R14 register is allocated.

If the automatic register allocation is disabled, you can use the **register** keyword to specify which global variable to be allocated to registers.
Example:

```
/* disable automatic register allocation */
#pragma regalloc-
/* allocate the variable 'alfa' to a register */
register int alfa;
/* allocate the variable 'beta' to the register pair R10, R11 */
register int beta @10;
```

Local **char**, **int** and **pointer** local variables are allocated to registers R16 to R21.
If the **Project|Configure|C Compiler|Code Generation|Smart Register Allocation** option is checked, the allocation of these registers for local variables is performed in such a way that 16bit variables will be preferably located in even register pairs, thus favouring the usage of the enhanced core MOVW instruction for their access.
Otherwise the local variables are automatically allocated to registers in the order of declaration.

The **Project|Configure|C Compiler|Code Generation|Smart Register Allocation** option should be disabled if the program was developed using CodeVisionAVR prior to V1.25.3 and it contains inline assembly code that accesses the variables located in registers R2 to R14 and R16 to R21.

## 4.7.4 Structures

Structures are user-defined collections of named members.
The structure members can be any of the supported data types, arrays of these data types or pointers to them.
Structures are defined using the **struct** reserved keyword.
The syntax is:

```
[<memory attribute>] struct [<structure tag-name>] {
    [<type> <variable-name>[,<variable-name>, ...]];
    [<type> [<bitfield-id>]:<width>[,[<bitfield-id>]:<width>, ...]];
    ...
    } [<structure variables>];
```

Example:

```
/* Global structure located in RAM */
struct ram_structure {
             char a,b;
             int  c;
             char d[30],e[10];
             char *pp;
             } sr;

/* Global constant structure located in FLASH */
flash struct flash_structure {
             int  a;
             char b[30], c[10];
             } sf;

/* Global structure located in EEPROM */
eeprom struct eeprom_structure {
             char a;
             int  b;
             char c[15];
             } se;

void main(void) {
/* Local structure */
struct local_structure {
             char a;
             int  b;
             long c;
             } sl;

/* ............ */

}
```

The space allocated to the structure in memory is equal to sum of the sizes of all the members.

The same generic structure type can be declared in any memory type: RAM, FLASH or EEPROM:

```
/* Generic structure type */
struct my_structure {
             char a,b;
             int  c;
             char d[30],e[10];
             char *pp;
             };

/* Global structure located in RAM */
struct my_structure sr;

/* Global pointer located in RAM to the RAM located structure */
struct my_structure *ptrsr = &sr;

/* Global pointer located in FLASH to the RAM located structure */
struct my_structure * flash ptrfsr = &sr;

/* Global pointer located in EEPROM to the RAM located structure */
struct my_structure * eeprom ptresr = &sr;

/* Global constant structure located in FLASH */
flash struct my_structure sf = {0,0,0,{0},{0},0};

/* Global pointer located in RAM to the FLASH located structure */
flash struct my_structure *ptrsf = &sf;

/* Global pointer located in FLASH to the FLASH located structure */
flash struct my_structure * flash ptrfsf = &sf;

/* Global pointer located in EEPROM to the FLASH located structure */
flash struct my_structure * eeprom ptresf = &sf;

/* Global constant structure located in EEPROM */
eeprom struct my_structure se;

/* Global pointer located in RAM to the EEPROM located structure */
eeprom struct my_structure *ptrse = &se;

/* Global pointer located in FLASH to the EEPROM located structure */
eeprom struct my_structure * flash ptrfse = &se;

/* Global pointer located in EEPROM to the EEPROM located structure */
eeprom struct my_structure * eeprom ptrese = &se;

void main(void) {
/* Local structure */
struct my_structure sl;
/* Local pointer to the RAM located global structure */
struct my_structure *ptrlsr = &sr;
/* Local pointer to the FLASH located global structure */
flash struct my_structure *ptrlsf = &sf;
/* Local pointer to the EEPROM located global structure */
eeprom struct my_structure *ptrlse = &se;

/* ............. */

}
```

Structures can be grouped in arrays.
Example how to initialize and access an global structure array stored in EEPROM:

```
/* Global structure array located in EEPROM */
eeprom struct eeprom_structure {
             char a;
             int  b;
             char c[15];
             } se[2]={{'a',25,"Hello"},
                     {'b',50,"world"}};

void main(void) {
char k1,k2,k3,k4;
int i1, i2;

/* define a pointer to the structure */
struct eeprom_structure eeprom *ep;

/* direct access to structure members */
k1=se[0].a;
i1=se[0].b;
k2=se[0].c[2];
k3=se[1].a;
i2=se[1].b;
k4=se[1].c[2];

/* same access to structure members using a pointer */
ep=&se; /* initialize the pointer with the structure address */
k1=ep->a;
i1=ep->b;
k2=ep->c[2];
++ep;   /* increment the pointer */
k3=ep->a;
i2=ep->b;
k4=ep->c[2];
}
```

Because some AVR devices have a small amount of RAM, in order to keep the size of the **Data Stack** small, it is recommended not to pass structures as function parameters and use pointers for this purpose.
Example:

```
struct alpha {
             int a,b, c;
             } s={2,3};
/* define the function */
struct alpha *sum_struct(struct alpha *sp) {
/* member c=member a + member b */
sp->c=sp->a + sp->b;
/* return a pointer to the structure */
return sp;
}
void main(void) {
int i;
/* s->c=s->a + s->b */
/* i=s->c */
i=sum_struct(&s)->c;
}
```

Structure members can be also declared as bit fields, having a width from 1 to 32.
Bit fields are allocated in the order of declaration starting from the least significant bit.
Example:

```
/* this structure will occupy 1 byte in RAM
   as the bit field data type is unsigned char */
struct alpha1 {
              unsigned char a:1; /* bit 0 */
              unsigned char b:4; /* bits 1..4 */
              unsigned char c:3; /* bits 5..7 */
              };

/* this structure will occupy 2 bytes in RAM
   as the bit field data type is unsigned int */
struct alpha2 {
              unsigned int a:2; /* bits 0..1 */
              unsigned int b:8; /* bits 2..9 */
              unsigned int c:4; /* bits 10..13 */
                                /* bits 14..15 are not used */
              };

/* this structure will occupy 4 bytes in RAM
   as the bit field data type is unsigned long */
struct alpha3 {
              unsigned long a:10; /* bits 0..9 */
              unsigned long b:8;  /* bits 10..17 */
              unsigned long c:6;  /* bits 18..23 */
                                  /* bits 24..31 are not used */
              };
```

### 4.7.5 Unions

Unions are user-defined collections of named members that share the same memory space.
The union members can be any of the supported data types, arrays of these data types or pointers to them.
Unions are defined using the **union** reserved keyword.
The syntax is:

```
[<memory attribute>] [<storage modifier>] union [<union tag-name>] {
    [<type> <variable-name>[,<variable-name>, ...]];
    [<type> <bitfield-id>:<width>[,<bitfield-id>:<width>, ...]];
    ...
    } [<union variables>];
```

The space allocated to the union in memory is equal to the size of the largest member.
Union members can be accessed in the same way as structure members. Example:

```
/* union declaration */
union alpha {
            unsigned char lsb;
            unsigned int  word;
            } data;


void main(void) {
unsigned char k;

/* define a pointer to the union */
union alpha *dp;

/* direct access to union members */
data.word=0x1234;
k=data.lsb; /* get the LSB of 0x1234 */

/* same access to union members using a pointer */
dp=&data;  /* initialize the pointer with the union address */
dp->word=0x1234;
k=dp->lsb; /* get the LSB of 0x1234 */
}
```

Because some AVR devices have a small amount of RAM, in order to keep the size of the **Data Stack** small, it is recommended not to pass unions as function parameters and use pointers for this purpose.
Example:

```
#include <stdio.h> /* printf */
union alpha {
            unsigned char lsb;
            unsigned int  word;
            } data;
/* define the function */
unsigned char low(union alpha *up) {
/* return the LSB of word */
return up->lsb;
}
void main(void) {
data.word=0x1234;
printf("the LSB of %x is %2x",data.word,low(&data));
}
```

Union members can be also declared as bit fields, having a width from 1 to 32.
Bit fields are allocated in the order of declaration starting from the least significant bit.
Example:

```
/* this union will occupy 1 byte in RAM
   as the bit field data type is unsigned char */
union alpha1 {
             unsigned char a:1; /* bit 0 */
             unsigned char b:4; /* bits 0..3 */
             unsigned char c:3; /* bits 0..2 */
             };

/* this union will occupy 2 bytes in RAM
   as the bit field data type is unsigned int */
union alpha2 {
             unsigned int a:2; /* bits 0..1 */
             unsigned int b:8; /* bits 0..7 */
             unsigned int c:4; /* bits 0..3 */
                               /* bits 8..15 are not used */
             };

/* this union will occupy 4 bytes in RAM
   as the bit field data type is unsigned long */
union alpha3 {
             unsigned long a:10; /* bits 0..9 */
             unsigned long b:8;  /* bits 0..7 */
             unsigned long c:6;  /* bits 0..5 */
                                 /* bits 10..31 are not used */
             };
```

## 4.7.6 Enumerations

The enumeration data type can be used in order to provide mnemonic identifiers for a set of **char** or **int** values.
The **enum** keyword is used for this purpose.
The syntax is:

```
[<memory attribute>] [<storage modifier>] enum [<enum tag-name>] {
     [<constant-name[[=constant-initializer], constant-name, ...]>]}
     [<enum variables>];
```

Example:

```
/* The enumeration constants will be initialized as follows:
   sunday=0 , monday=1 , tuesday=2 ,..., saturday=6 */
enum days {
          sunday, monday, tuesday, wednesday,
          thursday, friday, saturday} days_of_week;

/* The enumeration constants will be initialized as follows:
   january=1 , february=2 , march=3 ,..., december=12 */
enum months {
            january=1, february, march, april, may, june,
            july, august, september, october, november, december}
            months_of_year;

void main {
/* the variable days_of_week is initialized with
   the integer value 6 */
days_of_week=saturday;
}
```

Enumerations can be stored in RAM, EEPROM or FLASH.
The **eeprom** or **__eeprom** memory attributes must be used to specify enumeration storage in EEPROM.
Example:

```
eeprom enum days {
                 sunday, monday, tuesday, wednesday,
                 thursday, friday, saturday} days_of_week;
```

The **flash** or **__flash** memory attributes must be used to specify enumeration storage in FLASH memory.
Example:

```
flash enum months {
                  january, february, march, april, may, june,
                  july, august, september, october, november,
                  december}
                  months_of_year;
```

It is recommended to treat enumerations as having 8 bit **char** data type, by checking the **8 bit enums** check box in **Project|Configure|CompilerCode Generation**. This will improve the size and execution speed of the compiled program.

## 4.8 Defining Data Types

User defined data types are declared using the **typedef** reserved keyword.
The syntax is:

```
typedef <type definition> <identifier>;
```

The symbol name <identifier> is assigned to <type definition>.
Examples:

```
/* type definitions */
typedef unsigned char byte;
typedef struct {
               int a;
               char b[5];
               } struct_type;

/* variable declarations */
byte alfa;

/* structure stored in RAM */
struct_type struct1;

/* structure stored in FLASH */
flash struct_type struct2;

/* structure stored in EEPROM */
eeprom struct_type struct3;
```

## 4.9 Type Conversions

In an expression, if the two operands of a binary operator are of different types, then the compiler will convert one of the operands into the type of the other.
The compiler uses the following rules:

If either of the operands is of type **float** then the other operand is converted to the same type.

If either of the operands is of type **long int** or **unsigned long int** then the other operand is converted to the same type.

Otherwise, if either of the operands is of type **int** or **unsigned int** then the other operand is converted to the same type.

Thus **char** type or **unsigned char** type gets the lowest priority.

Using casting you can change these rules.
Example:

```
void main(void) {
int  a, c;
long b;
/* The long integer variable b will be treated here as an integer */
c=a+(int) b;
}
```

It is important to note that if the **Project|Configure|C Compiler|Code Generation|Promote char to int** option isn't checked or the **#pragma promotechar+** isn't used, the **char**, respectively **unsigned char**, type operands are not automatically promoted to **int** , respectively **unsigned int**, as in compilers targeted for 16 or 32 bit CPUs.
This helps writing more size and speed efficient code for an 8 bit CPU like the AVR.
To prevent overflow on 8 bit addition or multiplication, casting may be required.
The compiler issues warnings in these situations.
Example:

```
void main(void) {
unsigned char a=30;
unsigned char b=128;
unsigned int c;

/* This will generate an incorrect result, because the multiplication
   is done on 8 bits producing an 8 bit result, which overflows.
   Only after the multiplication, the 8 bit result is promoted to
   unsigned int */
c=a*b;

/* Here casting forces the multiplication to be done on 16 bits,
   producing an 16 bit result, without overflow */
c=(unsigned int) a*b;
}
```

The compiler behaves differently for the following operators:

```
+=
-=
*=
/=
%=
&=
|=
^=
<<=
>>=
```

For these operators, the result is to be written back onto the left-hand side operand (which must be a variable). So the compiler will always convert the right hand side operand into the type of left-hand side operand.

## 4.10 Operators

The compiler supports the following operators:

```
  +         -
  *         /
  %         ++
  --        =
  ==        ~
  !         !=
  <         >
  <=        >=
  &         &&
  |         ||
  ^         ? :
  <<        >>
  -=        +=
  /=        %=
  &=        *=
  ^=        |=
  >>=       <<=
sizeof
```

## 4.11 Functions

You may use function prototypes to declare a function.
These declarations include information about the function parameters.
Example:

```
int alfa(char par1, int par2, long par3);
```

The actual function definition may be written somewhere else as:

```
int alfa(char par1, int par2, long par3) {
/* Write some statements here */

}
```

The old Kernighan & Ritchie style of writing function definitions is not supported.
Function parameters are passed through the **Data Stack**.
Function values are returned in registers R30, R31, R22 and R23 (from LSB to MSB).

The special **__reset** attribute can be applied to a function that must be executed immediately after the chip reset before that startup initialization sequence.
This may be useful for XMEGA chips when **SDRAM** is used as external memory and a different clock source is used instead of the internal 2MHz oscillator, which will ensure that correct timing is used for later **SDRAM** access by the startup code.
Example:

```
__reset void system_clocks_init(void)
{
// Initialization code ...

}
```

## 4.12 Pointers

Due to the Harvard architecture of the AVR microcontroller, with separate address spaces for data (RAM), program (FLASH) and EEPROM memory, the compiler implements three types of pointers.
The syntax for pointer declaration is:

```
[<memory attribute>] type * [<memory attribute>]
                                [* [<memory attribute>] ...] pointer_name;
```

or

```
type [<memory attribute>] * [<memory attribute>]
                                [* [<memory attribute>] ...] pointer_name;
```

where type can be any data type.

Variables placed in RAM are accessed using normal pointers.
For accessing constants placed in FLASH memory, the **flash** or **__flash** memory attributes are used.
For accessing variables placed in EEPROM, the **eeprom** or **__eeprom** memory attributes are used.

Although the pointers may point to different memory areas, they are by default stored in RAM.
Example:

```
/* Pointer to a char string placed in RAM */
char *ptr_to_ram="This string is placed in RAM";

/* Pointer to a char string placed in FLASH */
flash char *ptr_to_flash1="This string is placed in FLASH";
char flash *ptr_to_flash2="This string is also placed in FLASH";

/* Pointer to a char string placed in EEPROM */
eeprom char *ptr_to_eeprom1="This string is placed in EEPROM";
char eeprom *ptr_to_eeprom2="This string is also placed in EEPROM";
```

In order to store the pointer itself in other memory areas, like FLASH or EEPROM, the **flash (__flash)** or **eeprom** (**__eeprom**) pointer storage memory attributes must be used as in the examples below:

```
/* Pointer stored in FLASH to a char string placed in RAM */
char * flash flash_ptr_to_ram="This string is placed in RAM";

/* Pointer stored in FLASH to a char string placed in FLASH */
flash char * flash flash_ptr_to_flash="This string is placed in FLASH";

/* Pointer stored in FLASH to a char string placed in EEPROM */
eeprom char * flash eeprom_ptr_to_eeprom="This string is placed in EEPROM";

/* Pointer stored in EEPROM to a char string placed in RAM */
char * eeprom eeprom_ptr_to_ram="This string is placed in RAM";

/* Pointer stored in EEPROM to a char string placed in FLASH */
flash char * eeprom eeprom_ptr_to_flash="This string is placed in FLASH";

/* Pointer stored in EEPROM to a char string placed in EEPROM */
eeprom char * eeprom eeprom_ptr_to_eeprom="This string is placed in
EEPROM";
```

In order to improve the code efficiency several memory models are implemented.

The **TINY** memory model uses 8 bits for storing pointers to the variables placed in RAM. In this memory model you can only have access to the first 256 bytes of RAM.

The **SMALL** memory model uses 16 bits for storing pointers the variables placed in RAM. In this memory model you can have access to 65536 bytes of RAM.

In both **TINY** and **SMALL** memory models pointers to the FLASH memory area use 16 bits.
Because in these memory models pointers to the FLASH memory are 16 bits wide, the total size of the constant arrays and literal char strings is limited to 64K.
However the total size of the program can be the full amount of FLASH.

In order to remove the above mentioned limitation, there are available two additional memory models: **MEDIUM** and **LARGE**.
The **MEDIUM** memory model is similar to the **SMALL** memory model, except it uses pointers to constants in FLASH that are 32 bits wide. The pointers to functions are however 16 bit wide because they hold the *word* address of the function, so 16 bits are enough to address a function located in all 128kbytes of FLASH.
The **MEDIUM** memory model can be used only for chips with 128kbytes of FLASH.

The **LARGE** memory model is similar to the **SMALL** memory model, except it uses pointers to the FLASH memory area that are 32 bits wide.
The **LARGE** memory model can be used for chips with 256kbytes or more of FLASH.

In all memory models pointers to the EEPROM memory area are 16 bit wide.

Pointers can be grouped in arrays, which can have up to 8 dimensions.
Example:

```
/* Declare and initialize a global array of pointers to strings
   placed in RAM */
char *strings[3]={"One","Two","Three"};

/* Declare and initialize a global array of pointers to strings
   placed in FLASH
   The pointer array itself is also stored in FLASH */
flash char * flash messages[3]={"Message 1","Message 2","Message 3"};

/* Declare some strings in EEPROM */
eeprom char m1[]="aaaa";
eeprom char m2[]="bbbb";

void main(void) {
/* Declare a local array of pointers to the strings placed in EEPROM
   You must note that although the strings are located in EEPROM,
   the pointer array itself is located in RAM */
char eeprom *pp[2];

/* and initialize the array */
pp[0]=m1;
pp[1]=m2;
}
```

Pointers to functions always access the FLASH memory area. There is no need to use the **flash** or **__flash** memory attributes for these types of pointers.
Example:

```
/* Declare a function */
int sum(int a, int b) {
return a+b;
}

/* Declare and initialize a global pointer to the function sum */
int (*sum_ptr) (int a, int b)=sum;

void main(void) {
int i;

/* Call the function sum using the pointer */
i=(*sum_ptr) (1,2);
}
```

## 4.13 Compiler Directives

Compiler specific directives are specified using the **#pragma** command.
You can use the **#pragma warn** directive to enable or disable compiler warnings.
Example:

```
/* Warnings are disabled */
#pragma warn-
/* Write some code here */

/* Warnings are enabled */
#pragma warn+
```

The compiler's code optimizer can be turned on or off using the **#pragma opt** directive. This directive must be placed at the start of the source file.
The default is optimization turned on.
Example:

```
/* Turn optimization off, for testing purposes */
#pragma opt-
```

or

```
/* Turn optimization on */
#pragma opt+
```

If the code optimization is enabled, you can optimize some portions or all the program for size or speed using the **#pragma optsize** directive.
The default state is determined by the **Project|Configure|C Compiler|Code Generation|Optimization** menu setting.

Example:

```
/* The program will be optimized for minimum size */
#pragma optsize+

/* Place your program functions here */

/* Now the program will be optimized for maximum execution speed */
#pragma optsize-

/* Place your program functions here */
```

The default optimization for **Size** or **Speed** specified the **Project|Configure|C Compiler|Code Generation|Optimization** menu setting can be restored using the **#pragma optsize_default** directive.
Example:

```
/* The program will be optimized for maximum speed */
#pragma optsize-

/* Place your program functions here */

/* Now the program will be optimized for the setting
   specified in the project configuration */
#pragma optsize_default

/* Place your program functions here */
```

The automatic saving and restoring of registers affected by the interrupt handler, can be turned on or off using the **#pragma savereg** directive.
Example:

```
/* Turn registers saving off */
#pragma savereg-

/* interrupt handler */
interrupt [1] void my_irq(void) {
/* now save only the registers that are affected by the routines in the
   interrupt handler, for example R30, R31 and SREG */
#asm
    push r30
    push r31
    in   r30,SREG
    push r30
#endasm

/* place the C code here */
/* .... */
/* now restore SREG, R31 and R30 */
#asm
    pop r30
    out SREG,r30
    pop r31
    pop r30
#endasm
}
/* re-enable register saving for the other interrupts */
#pragma savereg+
```

The default state is automatic saving of registers during interrupts.
The **#pragma savereg** directive is maintained only for compatibility with versions of the compiler prior to V1.24.1. <u>This directive is not recommended for new projects</u>.

The automatic allocation of global variables to registers can be turned on or off using the **#pragma regalloc** directive.
The default state is determined by the **Project|Configure|C Compiler|Code Generation|Automatic Global Register Allocation** check box.

Example:

```
/* the following global variable will be automatically
allocated to a register */
#pragma regalloc+
unsigned char alfa;

/* the following global variable will not be automatically
allocated to a register and will be placed in normal RAM */
#pragma regalloc-
unsigned char beta;
```

The ANSI **char** to **int** operands promotion can be turned on or off using the **#pragma promotechar** directive.
Example:

```
/* turn on the ANSI char to int promotion */
#pragma promotechar+

/* turn off the ANSI char to int promotion */
#pragma promotechar-
```

This option can also be specified in the **Project|Configure|C Compiler|Code Generation|Promote char to int** menu.

Treating **char** by default as an unsigned 8 bit can be turned on or off using the **#pragma uchar** directive.
Example:

```
/* char will be unsigned by default */
#pragma uchar+

/* char will be signed by default */
#pragma uchar-
```

This option can also be specified in the **Project|Configure|C Compiler|Code Generation|char is unsigned** menu.

The **#pragma library** directive is used for specifying the necessity to compile/link a specific library file.
Example:

```
#pragma library mylib.lib
```

The **#pragma glbdef+** directive is used for compatibility with projects, created with versions of CodeVisionAVR prior to V1.0.2.2, where the **Project|Configure|C Compiler|Global #define** option was enabled.
It signals the compiler that macros are globally visible in all the program modules of a project.
This directive must be placed in beginning of the first source file of the project.
By default this directive is not active, so macros are visible only in the program module where they are defined.

The **#pragma vector** directive is used for specifying that the next declared function is an interrupt service routine.

Example:

```
/* Vector numbers are for the AT90S8515 */
/* Specify the vector number using the #pragma vector directive */
#pragma vector=8

/* Called automatically on TIMER0 overflow */
__interrupt void timer0_overflow(void) {
/* Place your code here */

}
```

The **#pragma vector** directive and the **__interrupt** keyword are used for compatibility with other C compilers for the Atmel AVR.

The **#pragma keep+** directive forces a function, global variable or global constant to be linked even if it wasn't used anywhere in the program.
Example:

```
/* force the next function to be linked even if it's not used */
#pragma keep+

int func1(int a, int b)
{
return a+b;
}

/* the next function will not be linked if it's not used
#pragma keep-

int func2(int a, int b)
{
return a-b;
}
```

The **#pragma data_alignment=value** directive is used to align variables located in RAM at addresses which are multiples of **value**.
Example:

```
#pragma data_alignment=2

unsigned char alfa; /* alfa will be located at an even RAM address */
```

## 4.14 Accessing the I/O Registers

The compiler uses the **sfrb** and **sfrw** keywords to access the AVR microcontroller's I/O registers, located in the 0..3Fh address range, using the IN and OUT assembly instructions.
Example:

```
/* Define the SFRs */
sfrb PINA=0x19;  /* 8 bit access to the SFR */
sfrw TCNT1=0x2c; /* 16 bit access to the SFR */

void main(void)
{
unsigned char a;

a=PINA;        /* Read PORTA input pins */
TCNT1=0x1111; /* Write to TCNT1L & TCNT1H registers */
}
```

The **io.h** header file, located in the .\INC subdirectory, contains the definitions of the I/O registers for all the chips supported by the compiler.
The definitions are selected based on the AVR chip setting, specified by the **Project|Configure|C Compiler|Code Generation|Chip** option.
This header must be **#include** -d at the beginning of the .C source file.

For XMEGA chips the following syntax must be used for accessing I/O registers:

```
/* I/O register definitions for the XMEGA128A1 chip */
#include <io.h>

void main(void)
{
unsigned char a;

/* Set all PORTA pins as inputs */
PORTA.DIR=0x00;

/* Read PORTA input pins */
a=PORTA.IN;

/* Set all PORTB pins as outputs */
PORTB.DIR=0xFF;

/* Write data to PORTB outputs */
PORTB.OUT=0x11;

/* Set PORTB pin 2 to 1 */
PORTB.OUTSET=1 << 2;

/* Set PORTB pin 4 to 0 */
PORTB.OUTCLR=1 << 4;

/* Toggle PORTB pin 0 */
PORTB.OUTTGL=1 << 0;
}
```

The XMEGA I/O ports can be also accessed using the Virtual Ports:

```
/* I/O register definitions for the XMEGA128A1 chip */
#include <io.h>

void main(void)
{
unsigned char a;

/* Map PORTA to virtual port VPORT0 and
   PORTB to virtual port VPORT1  */
PORTCFG.VPCTRLA=PORTCFG_VP1MAP_PORTB_gc | PORTCFG_VP0MAP_PORTA_gc;

/* Set all VPORT0 (PORTA) pins as inputs */
VPORT0_DIR=0x00;

/* Read VPORT0 (PORTA) input pins */
a=VPORT0_IN;

/* Set all VPORT1 (PORTB) pins as outputs */
VPORT1_DIR=0xFF;

/* Write data to VPORT1 (PORTB) outputs */
VPORT1_OUT=0x11;

/* Set VPORT1 (PORTB) pin 2 to 1 */
VPORT1_OUT|=1 << 2;

/* Set VPORT1 (PORTB) pin 4 to 0 */
VPORT1.OUT&=1 << 4;

/* Toggle VPORT1 (PORTB) pin 0 */
VPORT1_OUT^=1 << 0;
}
```

More details about accessing I/O ports for the XMEGA chips can be found in the following Atmel documents:

- AVR1000: Getting Started Writing C-code for XMEGA
- XMEGA A Manual
- XMEGA AU Manual
- XMEGA D Manual.

## 4.14.1 Bit level access to the I/O Registers

Bit level access to the I/O registers can be performed by using the special macros that are defined in the *iobits.h* header file, located in the .\INC subdirectory.
The following macros are available:

**SETBIT(port,b)**

sets bit **b** of **port** to logic 1 state.

Example:

```
/* set bit 5 of I/O Port A output to logic 1 for non-XMEGA chips */
SETBIT(PORTA,5);

/* set bit 5 of I/O Port A output to logic 1 for XMEGA chips */
SETBIT(PORTA.OUT,5);
```

**CLRBIT(port,b)**

sets bit **b** of **port** to logic 0 state.

Example:

```
/* set bit 5 of I/O Port A output to logic 0 for non-XMEGA chips */
CLRBIT(PORTA,5);

/* set bit 5 of I/O Port A output to logic 0 for XMEGA chips */
CLRBIT(PORTA.OUT,5);
```

**TGLBIT(port,b)**

toggles (inverts) the logic state of bit **b** of **port**.

Example:

```
/* toggles bit 5 of I/O Port A output for non-XMEGA chips */
TGLBIT(PORTA,5);

/* toggles bit 5 of I/O Port A output for XMEGA chips */
TGLBIT(PORTA.OUT,5);
```

**EQUBIT(port,b,value)**

assigns a **value** to bit **b** of **port**.
If the assigned value is different from 0, then the bit is set to logic 1 state.
If the assigned value is 0, then the bit is set to logic 0 state.

Example:

```
/* sets bit 5 of I/O Port A output to the logic state of variable i
for non-XMEGA chips */
EQUBIT(PORTA,5,i);

/* sets bit 5 of I/O Port A output to the logic state of variable i
for XMEGA chips */
EQUBIT(PORTA.OUT,5,i);
```

**Note:** The **SETBIT**, **CLRBIT**, **TGLBIT** and **EQUBIT** macros always perform atomic I/O port bit access for the XMEGA chips.
For non-XMEGA chips, the atomic I/O port bit access using the **SETBIT**, **CLRBIT** and **EQUBIT** macros can be performed only for I/O ports with addresses located in I/O register space in the 0 to 1Fh range.

**TSTBIT(port,b)**

returns the logic state of bit **b** of **port**.

Example:

```
/* tests bit 5 of PINA (I/O Port A input) for non-XMEGA chips */
if (TSTBIT(PINA,5))
   {
   /* bit 5 of I/O Port A input is logic 1, do something... */

   }

/* tests bit 5 of PORTA.IN (I/O Port A input) for XMEGA chips */
if (TSTBIT(PORTA.IN,5))
   {
   /* bit 5 of I/O Port A input is logic 1, do something... */

   }
```

The bit level access to the I/O registers can be also accomplished by using bit selectors appended after the name of the I/O register.
Because bit level access to I/O registers is done using the CBI, SBI, SBIC and SBIS instructions, the register address must be in the 0 to 1Fh range for **sfrb** and in the 0 to 1Eh range for **sfrw**.

Example:

```
sfrb PORTA=0x1b;
sfrb DDRA=0x18;
sfrb PINA=0x19;

void main(void) {
/* set bit 0 of Port A as output */
DDRA.0=1;

/* set bit 1 of Port A as input */
DDRA.1=0;

/* set bit 0 of Port A output to logic 1 */
PORTA.0=1;
```

```
/* test bit 1 input of Port A */
if (PINA.1) { /* place some code here */ };

/* ....... */

}
```

The same program for XMEGA chips using the Virtual Port VPORT0:

```
/* I/O register definitions for the ATxmega128A1 chip */
#include <xmega128a1.h>

void main(void) {
/* Map PORTA to virtual port VPORT0 and
   PORTB to virtual port VPORT1  */
PORTCFG.VPCTRLA=PORTCFG_VP1MAP_PORTB_gc | PORTCFG_VP0MAP_PORTA_gc;

/* set bit 0 of Port A as output */
VPORT0_DIR.0=1;

/* set bit 1 of Port A as input */
VPORT0_DIR.1=0;

/* set bit 0 of Port A output to logic 1 */
VPORT0_OUT.0=1;

/* test bit 1 input of Port A */
if (VPORT0_IN.1) { /* place some code here */ };

/* ....... */

}
```

To improve the readability of the program you may wish to **#define** symbolic names to the bits in I/O registers:

```
sfrb PINA=0x19;
#define alarm_input PINA.2
void main(void)
{
/* test bit 2 input of Port A */
if (alarm_input) { /* place some code here */ };
/* ....... */
}
```

**Note:** Bit selector access to I/O registers located in internal RAM above address 5Fh (like PORTF for the ATmega128 for example) will not work, because the CBI, SBI, SBIC and SBIS instructions can't be used for RAM access.

## 4.15 Accessing the EEPROM

Accessing the AVR internal EEPROM is accomplished using global variables, preceded by the **eeprom** or **__eeprom** memory attributes.
Example:

```
/* The value 1 is stored in the EEPROM during chip programming */
eeprom int alfa=1;

eeprom char beta;
eeprom long array1[5];

/* The string is stored in the EEPROM during chip programming */
eeprom char string[]="Hello";

void main(void) {
int i;

/* Pointer to EEPROM */
int eeprom *ptr_to_eeprom;

/* Write directly the value 0x55 to the EEPROM */
alfa=0x55;
/* or indirectly by using a pointer */
ptr_to_eeprom=&alfa;
*ptr_to_eeprom=0x55;

/* Read directly the value from the EEPROM */
i=alfa;
/* or indirectly by using a pointer */
i=*ptr_to_eeprom;
}
```

Pointers to the EEPROM always occupy 16 bits in memory.

## 4.16 Using Interrupts

The access to the AVR interrupt system is implemented with the **interrupt** keyword.
Example:

```
/* Vector numbers are for the AT90S8515 */

/* Called automatically on external interrupt */
interrupt [2] void external_int0(void) {
/* Place your code here */

}

/* Called automatically on TIMER0 overflow */
interrupt [8] void timer0_overflow(void) {
/* Place your code here */

}
```

Interrupt vector numbers start with 1.
The compiler will automatically save the affected registers when calling the interrupt functions and restore them back on exit.
A RETI assembly instruction is placed at the end of the interrupt function.
Interrupt functions can't return a value nor have parameters.
You must also set the corresponding bits in the peripheral control registers to configure the interrupt system and enable the interrupts.

Another possibility to declare an interrupt service routine is by using the **#pragma vector** preprocessor directive and the **__interrupt** keyword.
**#pragma vector** is used for specifying that the next declared function is an interrupt service routine.
Example:

```
/* Vector numbers are for the AT90S8515 */

/* Specify the vector number using the #pragma vector directive */
#pragma vector=2

/* Called automatically on external interrupt */
__interrupt void external_int0(void) {
/* Place your code here */

}

/* Specify the vector number using the #pragma vector directive */
#pragma vector=8

/* Called automatically on TIMER0 overflow */
__interrupt void timer0_overflow(void) {
/* Place your code here */

}
```

The **#pragma vector** preprocessor directive and the **__interrupt** keyword are used for compatibility with other C compilers for the Atmel AVR.

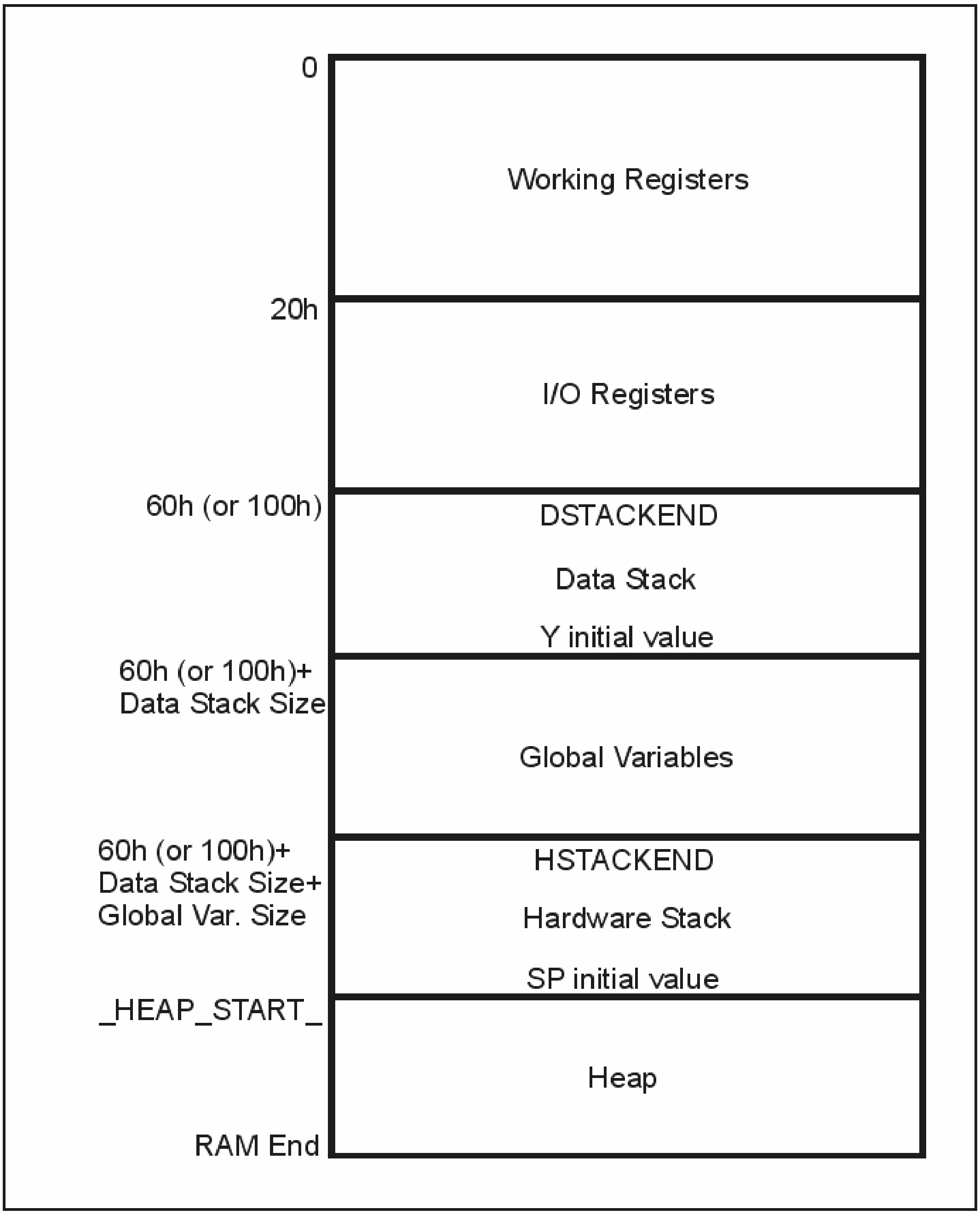
The automatic saving and restoring of registers affected by the interrupt handler, can be turned on or off using the **#pragma savereg** directive.
Example:

```
/* Turn registers saving off */
#pragma savereg-

/* interrupt handler */
interrupt [1] void my_irq(void) {
/* now save only the registers that are affected by the routines in the
   interrupt handler, for example R30, R31 and SREG */
#asm
    push r30
    push r31
    in   r30,SREG
    push r30
#endasm

/* place the C code here */
/* .... */
/* now restore SREG, R31 and R30 */
#asm
    pop r30
    out SREG,r30
    pop r31
    pop r30
#endasm
}
/* re-enable register saving for the other interrupts */
#pragma savereg+
```

The default state is automatic saving of registers during interrupts.
The **#pragma savereg** directive is maintained only for compatibility with versions of the compiler prior to V1.24.1. This directive is not recommended for new projects.

## 4.17 RAM Memory Organization and Register Allocation

A compiled program has the following memory map:




The **Working Registers** area contains 32x8 bit general purpose working registers.

The register usage depends on the type of the AVR core for which code is generated:

- **standard core**: the compiler uses the following registers: R0, R1, R15, R22, R23, R24, R25, R26, R27, R28, R29, R30 and R31.
  Also some of the registers from R2 to R15 may be allocated by the compiler for global and local bit variables. The rest of unused registers, in this range, are allocated for global char and int variables and pointers. Registers R16 to R21 are allocated for local char and int variables.
- **reduced core** (ATtiny10): the compiler uses the following registers: R16, R17, R22, R23, R24, R25, R26, R27, R28, R29, R30 and R31.
  No registers may be allocated by the compiler for global and local bit variables.
  Registers R18 to R21 are allocated for local char and int variables.

The **I/O Registers** area contains 64 addresses for the CPU peripheral functions as Port Control Registers, Timer/Counters and other I/O functions. You may freely use these registers in your assembly programs.

The **Data Stack** area is used to dynamically store local variables, passing function parameters and saving registers during interrupt routine servicing:

- **standard core**: R0, R1, R15, R22, R23, R24, R25, R26, R27, R30, R31 and SREG
- **reduced core**: R16, R17, R22, R23, R24, R25, R26, R27, R30, R31 and SREG.

The **Data Stack Pointer** is implemented using the Y register.
At start-up the **Data Stack Pointer** is initialized with the value 5Fh (or FFh for some chips)+Data Stack Size.
When saving a value in the **Data Stack,** the **Data Stack Pointer** is decremented.
When the value is retrieved, the **Data Stack Pointer** is incremented back.
When configuring the compiler, in the **Project|Configure|C Compiler|Code Generation** menu, you must specify a sufficient **Data Stack Size**, so it will not overlap the **I/O Register** area during program execution.

The **Global Variables** area is used to statically store the global variables during program execution.
The size of this area can be computed by summing the size of all the declared global variables.

The **Hardware Stack** area is used for storing the functions return addresses.
The SP register is used as a stack pointer and is initialized at start-up with value of the **_HEAP_START_** -1 address.
During the program execution the **Hardware Stack** grows downwards to the **Global Variables** area.

When configuring the compiler you have the option to place the strings **DSTACKEND**, respectively **HSTACKEND,** at the end of the **Data Stack**, respectively **Hardware Stack** areas.

When you debug the program with AVR Studio you may see if these strings are overwritten, and consequently modify the **Data Stack Size** using the **Project|Configure|C Compiler|Code Generation** menu command.
When your program runs correctly, you may disable the placement of the strings in order to reduce code size.

The **Heap** is a memory area located between the **Hardware Stack** and the **RAM end**.
It is used by the memory allocation functions from the Standard Library: malloc, calloc, realloc and free.

The **Heap** size must be specified in the **Project|Configure|C Compiler|Code Generation** menu.
It can be calculated using the following formulae:

$$heap\_size = (n+1) \cdot 4 + \sum_{i=1}^{n} block\_size_i$$

where: $n$ is the number of memory blocks that will be allocated in the **Heap**
$block\_size_i$ is the size of the memory block $i$

If the memory allocation functions will not be used, then the **Heap** size must be specified as zero.

## 4.18 Using an External Startup Assembly File

In every program the CodeVisionAVR C compiler automatically generates a code sequence to make the following initializations immediately after the AVR chip reset:
1. interrupt vector jump table
2. global interrupt disable
3. EEPROM access disable
4. Watchdog Timer disable
5. external RAM access and wait state enable if necessary
6. clear registers R2 … R14 (for AVR8 standard core chips)
7. clear the RAM
8. initialize the global variables located in RAM
9. initialize the **Data Stack Pointer** register Y
10. initialize the **Stack Pointer** register SP
11. initialize the UBRR register if necessary

The automatic generation of code sequences 2 to 8 can be disabled by checking the **Use an External Startup Initialization File** check box in the **Project|Configure|C Compiler|Code Generation** dialog window. The C compiler will then include, in the generated .asm file, the code sequences from an external file that must be named **STARTUP.ASM** . This file must be located in the directory where your main C source file resides.
You can write your own **STARTUP.ASM** file to customize or add some features to your program. The code sequences from this file will be immediately executed after the chip reset.
A basic **STARTUP.ASM** file is supplied with the compiler distribution and is located in the .\BIN directory.
Here's the content of this file:

```
;CodeVisionAVR C Compiler
;(C) 1998-2008 Pavel Haiduc, HP InfoTech s.r.l.

   .EQU __CLEAR_START=0X60 ;START ADDRESS OF RAM AREA TO CLEAR
                 ;SET THIS ADDRESS TO 0X100 FOR THE
                 ;ATmega128 OR ATmega64 CHIPS
   .EQU __CLEAR_SIZE=256   ;SIZE OF RAM AREA TO CLEAR IN BYTES

   CLI           ;DISABLE INTERRUPTS
   CLR  R30
   OUT  EECR,R30  ;DISABLE EEPROM ACCESS

;DISABLE THE WATCHDOG
   LDI  R31,0x18
   OUT  WDTCR,R31
   OUT  WDTCR,R30

   OUT  MCUCR,R30 ;MCUCR=0, NO EXTERNAL RAM ACCESS

;CLEAR R2-R14
   LDI  R24,13
   LDI  R26,2
   CLR  R27
__CLEAR_REG:
   ST   X+,R30
   DEC  R24
   BRNE __CLEAR_REG
```

```
;CLEAR RAM
   LDI  R24,LOW(__CLEAR_SIZE)
   LDI  R25,HIGH(__CLEAR_SIZE)
   LDI  R26,LOW(__CLEAR_START)
   LDI  R27,HIGH(__CLEAR_START)
__CLEAR_RAM:
   ST   X+,R30
   SBIW R24,1
   BRNE __CLEAR_RAM

;GLOBAL VARIABLES INITIALIZATION
   LDI  R30,LOW(__GLOBAL_INI_TBL*2)
   LDI  R31,HIGH(__GLOBAL_INI_TBL*2)
__GLOBAL_INI_NEXT:
   LPM
   ADIW R30,1
   MOV  R24,R0
   LPM
   ADIW R30,1
   MOV  R25,R0
   SBIW R24,0
   BREQ __GLOBAL_INI_END
   LPM
   ADIW R30,1
   MOV  R26,R0
   LPM
   ADIW R30,1
   MOV  R27,R0
   LPM
   ADIW R30,1
   MOV  R1,R0
   LPM
   ADIW R30,1
   MOV  R22,R30
   MOV  R23,R31
   MOV  R31,R0
   MOV  R30,R1
__GLOBAL_INI_LOOP:
   LPM
   ADIW R30,1
   ST   X+,R0
   SBIW R24,1
   BRNE __GLOBAL_INI_LOOP
   MOV  R30,R22
   MOV  R31,R23
   RJMP __GLOBAL_INI_NEXT
__GLOBAL_INI_END:
```

The **__CLEAR_START** and **__CLEAR_SIZE** constants can be changed to specify which area of RAM to clear at program initialization.
The **__GLOBAL_INI_TBL** label must be located at the start of a table containing the information necessary to initialize the global variables located in RAM. This table is automatically generated by the compiler.

## 4.19 Including Assembly Language in Your Program

You can include assembly language anywhere in your program using the **#asm** and **#endasm** directives.
Example:

```
void delay(unsigned char i) {
while (i--) {
      /* Assembly language code sequence */
      #asm
          nop
          nop
      #endasm
      };
}
```

Inline assembly may also be used.
Example:

```
#asm("sei") /* enable interrupts */
```

The registers R0, R1, R22, R23, R24, R25, R26, R27, R30 and R31 can be freely used in assembly routines.
However when using them in an interrupt service routine the programmer must save, respectively restore, them on entry, respectively on exit, of this routine.

## 4.19.1 Calling Assembly Functions from C

The following example shows how to access functions written in assembly language from a C program:

```
// function in assembler declaration
// this function will return a+b+c
#pragma warn- // this will prevent warnings
int sum_abc(int a, int b, unsigned char c) {
#asm
    ldd   r30,y+3 ;R30=LSB a
    ldd   r31,y+4 ;R31=MSB a
    ldd   r26,y+1 ;R26=LSB b
    ldd   r27,y+2 ;R27=MSB b
    add   r30,r26 ;(R31,R30)=a+b
    adc   r31,r27
    ld    r26,y   ;R26=c
    clr   r27     ;promote unsigned char c to int
    add   r30,r26 ;(R31,R30)=(R31,R30)+c
    adc   r31,r27
#endasm
}
#pragma warn+ // enable warnings

void main(void) {
int r;
// now we call the function and store the result in r
r=sum_abc(2,4,6);
}
```

The compiler passes function parameters using the **Data Stack**.
First it pushes the integer parameter a, then b, and finally the unsigned char parameter c.
On every push the Y register pair decrements by the size of the parameter (4 for long int, 2 for int, 1 for char).
For multiple byte parameters the MSB is pushed first.
As it is seen the **Data Stack** grows downward.
After all the functions parameters were pushed on the **Data Stack**, the Y register points to the last parameter c, so the function can read its value in R26 using the instruction: ld r26,y.
The b parameter was pushed before c, so it is at a higher address in the **Data Stack**.
The function will read it using: ldd r27,y+2 (MSB) and ldd r26,y+1 (LSB).
The MSB was pushed first, so it is at a higher address.
The a parameter was pushed before b, so it is at a higher address in the **Data Stack**.
The function will read it using: ldd r31,y+4 (MSB) and ldd r30,y+3 (LSB).

The functions return their values in the registers (from LSB to MSB):

- R30 for char and unsigned char
- R30, R31 for int and unsigned int
- R30, R31, R22, R23 for long and unsigned long.

So our function must return its result in the R30, R31 registers.

After the return from the function the compiler automatically generates code to reclaim the **Data Stack** space used by the function parameters.

The **#pragma warn-** compiler directive will prevent the compiler from generating a warning that the function does not return a value.
This is needed because the compiler does not know what it is done in the assembler portion of the function.

## 4.20 Creating Libraries

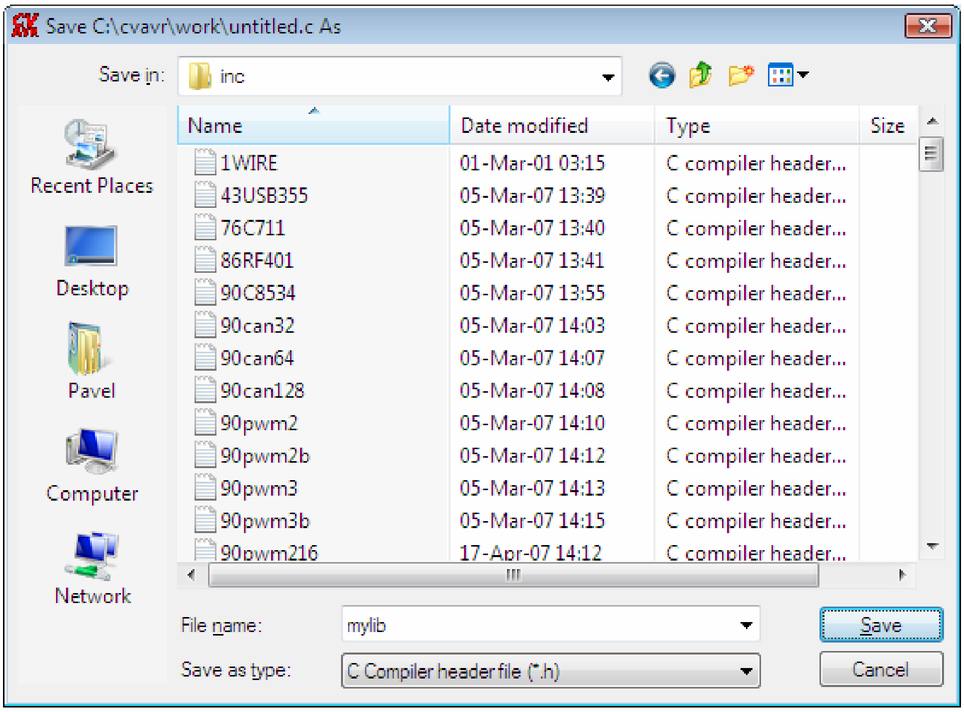
In order to create your own libraries, the following steps must be followed:

**1.** Create a header .h file with the prototypes of the library functions using the **File|New|Source File** menu command, by pressing the **Ctrl+N** keys or the and buttons on the toolbar.

A new editor window will be opened for the **untitled.c** source file.
Type in the prototype for your function. Example:

```
/* this #pragma directive will prevent the compiler from generating a
   warning that the function was declared, but not used in the program */
#pragma used+
/* library function prototypes */
int sum(int a, int b);
int mul(int a, int b);
#pragma used-
/* this #pragma directive will tell the compiler to compile/link the
   functions from the mylib.lib library */
#pragma library mylib.lib
```

Save the file, using the **File|Save As** menu command or the toolbar button, in the .**INC** directory using the **File|Save As** menu command, for example **mylib.h** :

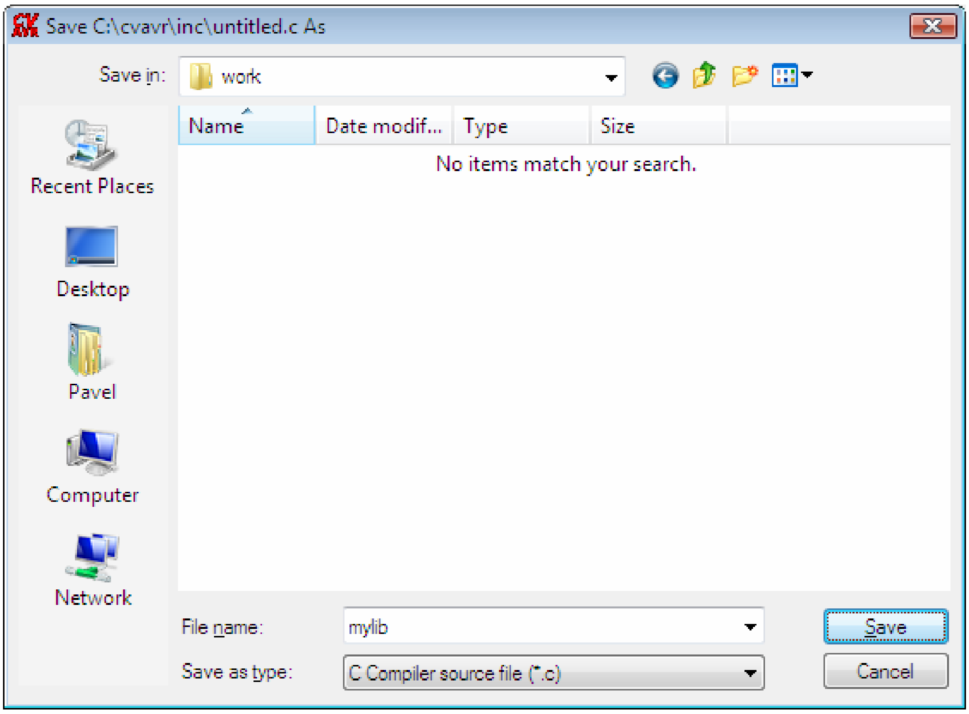


Create the library file using the **File|New|Source File** menu command, by pressing the **Ctrl+N** keys or the and buttons on the toolbar.

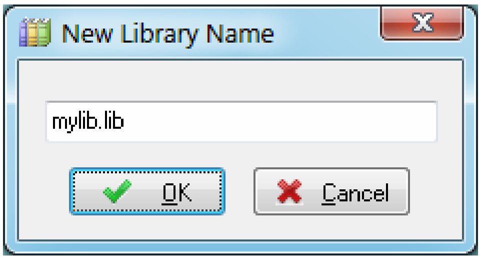
A new editor window will be opened for the **untitled.c** source file.
Type in the definitions for your functions.
Example:

```
int sum(int a, int b) {
return a+b;
}

int mul(int a, int b) {
return a*b;
}
```

Save the file, under a new name, for example **mylib.c** , in any directory using the **File|Save As** menu command or the toolbar button:



Finally use the **File|Convert to Library** menu command or the toolbar button, to save the currently opened .c file under the name **mylib.lib** in the **.\LIB** directory:




In order to use the newly created **mylib.lib** library, just **#include** the **mylib.h** header file in the beginning of your program.
Example:

```
#include <mylib.h>
```

Library files usually reside in the **.\LIB** directory, but paths to additional directories can be added in the **Project|Configure|C Compiler|Paths|Library paths** menu.

## 4.21 Using the AVR Studio 4.19 Debugger

CodeVisionAVR is designed to work in conjunction with the Atmel AVR Studio 4.19 debugger.

In order to be able to do C source level debugging using AVR Studio, you must select the COFF **Output File Format** in the **Project|Configure|C Compiler|Code Generation** menu option.

**Important Note:** It is highly recommended to set the **Optimize for: Speed** option in the **Project|Configure|C Compiler|Code Generation** menu, which will allow correct debugging of the program. Once debugging was finished, this option can be also set to **Optimize for: Size**.

The debugger is invoked using the **Tools|Debugger** menu command or the  toolbar button.
In order to be able to do this, the debugger version and it's installation path must be first specified using the **Settings|Debugger** menu.

After AVR Studio is launched, the user must first select **File|Open File** (**Ctr+O** keys) in order to load the COFF file to be debugged.
After the COFF file is loaded, and no AVR Studio project file exists for this COFF file, the debugger will open a **Select device and debug platform** dialog window.
In this window the user must specify the Debug Platform: ICE or AVR Simulator and the AVR Device type.
Pressing the **Finish** button will create a new AVR Studio project associated with the COFF file.
If an AVR Studio project associated with the COFF file already exists, the user will be asked if the debugger may load it.

Once the program is loaded, it can be launched in execution using the **Debug|Run** menu command, by pressing the **F5** key or by pressing the **Run** toolbar button.
Program execution can be stopped at any time using the **Debug|Break** menu command, by pressing **Ctrl+F5** keys or by pressing the **Break** toolbar button.

To single step the program, the **Debug|Step Into** (**F11** key), **Debug|Step Over** (**F10** key), **Debug|Step Out** (**Shift+F11** keys) menu commands or the corresponding toolbar buttons should be used.

In order to stop the program execution at a specific source line, the **Debug|Toggle Breakpoint** menu command, the **F9** key or the corresponding toolbar button should be used.

In order to watch program variables, the user must select **Debug|Quickwatch** (**Shift+F9** keys) menu command or press the **Quickwatch** toolbar button, and specify the name of the variable in the **QuickWatch** window in the **Name** column.

The AVR chip registers can be viewed using the **View|Register** menu command or by pressing the **Alt+0** keys. The registers can be also viewed in the **Workspace|I/O** window in the **Register 0-15** and **Register 16-31** tree branches.
The AVR chip PC, SP, X, Y, Z registers and status flags can be viewed in the **Workspace|I/O** window in the **Processor** tree branch.
The contents of the FLASH, RAM and EEPROM memories can be viewed using the **View|Memory** menu command or by pressing the **Alt+4** keys.
The I/O registers can be viewed in the **Workspace|I/O** window in the **I/O** branch.

To obtain more information about using AVR Studio 4.19, please consult its Help system.

## 4.22 Using the Command Line Compiler

The CodeVisionAVR C Compiler can be also executed from the command line, allowing for the automation of program build tasks.

The following syntax is used when invoking the compiler from the command line:

**cvavrcl <project_file> <command> [options] [arguments]**

where **project_file** is the full path of the project **.prj** file created/modified using the CodeVisionAVR IDE. The path must be enclosed between **" "** if it contains long file names.

The following commands are accepted:

| | |
|---|---|
| **-b [options] [-m <message_out_file>]** | *Build* project - only the modified source files are re-compiled |
| **-ba [options] [-m <message_out_file>]** | *Build All* project files – all source files are re-compiled |
| **-s <source_file> [-m <message_out_file>]** | Check *Syntax* for **source_file** |
| **-cl** | *Cleanup* project directory. |

If the **–m** argument is used, the output of the command line compiler is written to the **message_out_file**, otherwise it is directed to the console.

The following options are available:

| | |
|---|---|
| **-dbg** | Forces build using the *Debug* project configuration |
| **-rel** | Forces build using the *Release* project configuration |
| **-p {STK500\|STK600\|JTAGICE2\|JTAGICE3\|AVRDRAGON\|AVRISP2\|AVR910}** | Select the *Programmer* type used for **After Build** chip programming |
| **-com <n>** | Specifies the *COM* port number used for communication with the STK500, AVR910 programmers |
| **-pi {ISP_PDI\|JTAG}** | Specifies the *Programming Interface* used by the programmer |
| **-sck <n>** | Specifies the *SCK* frequency [Hz] used for ISP programming |
| **-rc** | Specifies that the build results or messages will be directed to the *console* |
| **-rw** | Specifies that the build results or messages will be displayed in a *window* in the GUI. |

Examples:

**cvavrcl "c:\cvavr303\examples\ds1820\ds1820.prj" -ba –rc**

Re-compiles all source files in the **ds1820.prj** project and outputs the results to the console.

**cvavrcl "c:\cvavr303\examples\ds1820\ds1820.prj" -ba -rw -p AVRISP2 -pi ISP_PDI -sck 115200 -m messages.txt**

Re-compiles all source files in the **ds1820.prj** project, outputs the build results to the GUI and automatically programs the chip using the AVRISP MKII programmer with a SCK frequency of 115200 Hz.
All messages will be also written to the **messages.txt** file, that will be located in the directory where the **ds1820.prj** project file is located.

**cvavrcl "c:\cvavr303\examples\ds1820\ds1820.prj" -s ds1820.c -m messages.txt**

Checks the syntax for the **ds1820.c** file from the **ds1820.prj** project and writes the results to the **messages.txt** file.

## 4.23 Compiling the Sample Code of the XMEGA Application Notes from Atmel

In order to support the new XMEGA chips, Atmel has released a number of Application Notes that are available for download from www.atmel.com.

The sample code for these Application Notes can be easily compiled with CodeVisionAVR.
For this purpose the header file **avr_compiler.h** supplied for each Application Note must be replaced with the same file located in the ..\INC directory of the CodeVisionAVR installation.

## 4.24 Hints

In order to decrease code size and improve the execution speed, you must apply the following rules:

- If possible use **unsigned** variables;
- Use the smallest data type possible, i.e. **bit** and **unsigned char**;
- The size of the bit variables, allocated to the program in the **Project|Configure|C Compiler|Code Generation|Bit Variables Size** list box, should be as low as possible, in order to free registers for allocation to other global variables;
- If possible use the smallest possible memory model (TINY or SMALL);
- Always store constant strings in FLASH by using the **flash** or **__flash** memory models;
- After finishing debugging your program, compile it again with the **Stack End Markers** option disabled.

## 4.25 Limitations

The current version of the CodeVisionAVR C compiler has the following limitations:

- the **long long**, **double**, **_Complex** and **_Imaginary** data types are not yet supported
- the **bit** data type is not supported for the reduced core, used in chips like ATtiny10
- functions with variable number of parameters are not supported for reduced core chips
- signal handling (signal.h) is not implemented yet
- date and time functions (time.h) are not implemented yet
- extended multibyte/wide character utilities (wchar.h) are not implemented
- wide character classification and mapping utilities (wctype.h) are not implemented
- the **printf**, **sprintf**, **snprintf**, **vprintf**, **vsprintf** and **vsnprintf** Standard C Input/Output Functions can't output strings longer than 255 characters for the **%s** format specifier
- the size of the compiled code is limited for the Evaluation version
- the libraries for Philips PCF8563, Philips PCF8583, Maxim/Dallas Semiconductor DS1302, DS1307, 4x40 character LCD, XMEGA TWI functions and EBI support are not available in the Evaluation version
- the libraries for color graphic LCDs (ILI9325, SSD1289, SSD1963, SSD2119 and XG7100) are available and can be used only with Advanced or Professional licenses.

# 5. Library Functions Reference

You must **#include** the appropriate header files for the library functions that you use in your program.
Example:

```
/* Header files are included before using the functions */
#include <stdlib.h>  // for abs
#include <stdio.h> // for putsf

void main(void) {
int a,b;
a=-99;
/* Here you actually use the functions */
b=abs(a);
putsf("Hello world");
}
```

## 5.1 Character Type Functions

The prototypes for these functions are placed in the file **ctype.h**, located in the .\INC subdirectory. This file must be **#include -d** before using the functions.

**unsigned char isalnum(char c)**

returns 1 if c is alphanumeric.

**unsigned char isalpha(char c)**

returns 1 if c is alphabetic.

**unsigned char isascii(char c)**

returns 1 if c is an ASCII character (0..127).

**unsigned char iscntrl(char c)**

returns 1 if c is a control character (0..31 or 127).

**unsigned char isdigit(char c)**

returns 1 if c is a decimal digit.

**unsigned char islower(char c)**

returns 1 if c is a lower case alphabetic character.

**unsigned char isprint(char c)**

returns 1 if c is a printable character (32..127).

**unsigned char ispunct(char c)**

returns 1 if c is a punctuation character (all but control and alphanumeric).

**unsigned char isspace(char c)**

returns 1 c is a white-space character (space, CR, HT).

**unsigned char isupper(char c)**

returns 1 if c is an upper-case alphabetic character.

**unsigned char isxdigit(char c)**

returns 1 if c is a hexadecimal digit.

**char toascii(char c)**

returns the ASCII equivalent of character c.

**unsigned char toint(char c)**

interprets c as a hexadecimal digit and returns an unsigned char from 0 to 15.

**char tolower(char c)**

returns the lower case of c if c is an upper case character, else c.

**char toupper(char c)**

returns the upper case of c if c is a lower case character, else c.

## 5.2 Standard C Input/Output Functions

The prototypes for these functions are placed in the file **stdio.h**, located in the .\INC subdirectory. This file must be **#include -d** before using the functions.

The standard C language I/O functions were adapted to work on embedded microcontrollers with limited resources.

The lowest level Input/Output functions are:

**char getchar(void)**

returns a character received by the UART, using polling.

**void putchar(char c)**

transmits the character c using the UART, using polling.

Prior to using these functions you must:
- initialize the UART's Baud rate
- enable the UART transmitter
- enable the UART receiver.

Example:

```
#include <mega8515.h>
#include <stdio.h>

/* quartz crystal frequency [Hz] */
#define xtal 4000000L

/* Baud rate */
#define baud 9600

void main(void) {
char k;

/* initialize the USART control register
   TX and RX enabled, no interrupts, 8 data bits */
UCSRA=0x00;
UCSRB=0x18;
UCSRC=0x86;

/* initialize the USART's baud rate */
UBRRH=(xtal/16/baud-1) >> 8;
UBRRL=(xtal/16/baud-1) & 0xFF;
```

```
while (1) {
        /* receive the character */
        k=getchar();
        /* and echo it back */
        putchar(k);
        };
}
```

If you intend to use other peripherals for Input/Output, you must modify accordingly the **getchar** and **putchar** functions like in the example below:

```
#include <stdio.h>

/* inform the compiler that an alternate version
   of the getchar function will be used */
#define _ALTERNATE_GETCHAR_

/* now define the new getchar function */
char getchar(void) {
/* write your code here */

}

/* inform the compiler that an alternate version
   of the putchar function will be used */
#define _ALTERNATE_PUTCHAR_

/* now define the new putchar function */
void putchar(char c) {
/* write your code here */

}
```

For the XMEGA chips the **getchar** and **putchar** functions use by default the **USARTC0**.
If you wish to use another USART, you must define the **_ATXMEGA_USART_** preprocessor macro prior to #include the **stdio.h** header file, like in the example below:

```
/* use the ATxmega128A1 USARTD0 for getchar and putchar functions */
#define _ATXMEGA_USART_ USARTD0

/* use the Standard C I/O functions */
#include <stdio.h>
```

The **_ATXMEGA_USART_** macro needs to be defined only once in the whole program, as the compiler will treat it like it is globally defined.

All the high level Input/Output functions use **getchar** and **putchar**.

**void puts(char *str)**

outputs, using **putchar**, the null terminated character string **str**, located in RAM, followed by a new line character.

**void putsf(char flash *str)**

outputs, using **putchar**, the null terminated character string **str**, located in FLASH, followed by a new line character.

**int printf(char flash *fmtstr [ , arg1, arg2, ...])**

outputs formatted text, using **putchar**, according to the format specifiers in the **fmtstr** string.
The format specifier string **fmtstr** is constant and must be located in FLASH memory.
The implementation of **printf** is a reduced version of the standard C function.
This was necessary due to the specific needs of an embedded system and because the full implementation would require a large amount of FLASH memory space.
The function returns the number of outputed characters.

The format specifier string has the following structure:

```
%[flags][width][.precision][l]type_char
```

The optional **flags** characters are:
'-' left-justifies the result, padding on the right with spaces. If it's not present, the result will be right-justified, padded on the left with zeros or spaces;
'+' signed conversion results will always begin with a '+' or '-' sign;
' ' if the value isn't negative, the conversion result will begin with a space. If the value is negative then it will begin with a '-' sign.

The optional **width** specifier sets the minimal width of an output value. If the result of the conversion is wider than the field width, the field will be expanded to accommodate the result, so not to cause field truncation.

The following width specifiers are supported:
n - at least n characters are outputted. If the result has less than n characters, then its field will be padded with spaces. If the '-' flag is used, the result field will be padded on the right, otherwise it will be padded on the left;
0n - at least n characters are outputted. If the result has less than n characters, it is padded on the left with zeros.

The optional **precision** specifier sets the maximal number of characters or minimal number of integer digits that may be outputted.
For the 'e', 'E' and 'f' conversion type characters the precision specifier sets the number of digits that will be outputted to the right of the decimal point.
The precision specifier always begins with a '.' character in order to separate it from the width specifier.
The following precision specifiers are supported:
none - the precision is set to 1 for the 'i', 'd', 'u', 'x', 'X' conversion type characters. For the 's' and 'p' conversion type characters, the char string will be outputted up to the first null character;
.0 - the precision is set to 1 for the 'i', 'd', 'u', 'x', 'X' type characters;
.n - n characters or n decimal places are outputted.
For the 'i', 'd', 'u', 'x', 'X' conversion type characters, if the value has less than n digits, then it will be padded on the left with zeros. If it has more than n digits then it will not be truncated.
For the 's' and 'p' conversion type characters, no more than n characters from the char string will be outputted.
For the 'e', 'E' and 'f' conversion type characters, n digits will be outputted to the right of the decimal point.
The precision specifier has no effect on the 'c' conversion type character.

The optional '**l**' input size modifier specifies that the function argument must be treated as a long int for the 'i', 'd', 'u', 'x', 'X' conversion type characters.

The **type_char** conversion type character is used to specify the way the function argument will be treated.
The following conversion type characters are supported:
'i' - the function argument is a signed decimal integer;
'd' - the function argument is a signed decimal integer;
'u' - the function argument is an unsigned decimal integer;
'e' - the function argument is a float, that will be outputted using the [-]*d.dddddd* e[±]*dd* format
'E' - the function argument is a float, that will be outputted using the [-]*d.dddddd* E[±]*dd* format
'f' - the function argument is a float, that will be outputted using the [-]*ddd.dddddd* format
'x' - the function argument is an unsigned hexadecimal integer, that will be outputted with lowercase characters;
'X' - the function argument is an unsigned hexadecimal integer, that will be outputted with with uppercase characters;
'c' - the function argument is a single character;
's' - the function argument is a pointer to a null terminated char string located in RAM;
'p' - the function argument is a pointer to a null terminated char string located in FLASH;
'%' - the '%' character will be outputted.

**int sprintf(char *str, char flash *fmtstr [ , arg1, arg2, ...])**

this function is identical to **printf** except that the formatted text is placed in the null terminated character string **str**.
The function returns the number of outputed characters.

**int snprintf(char *str, unsigned char size, char flash *fmtstr [ , arg1, arg2, ...])**
*for the TINY memory model.*

**int snprintf(char *str, unsigned int size, char flash *fmtstr [ , arg1, arg2, ...])**
*for the other memory models.*

this function is identical to **sprintf** except that at most **size** (including the null terminator) characters are placed in the character string **str**.
The function returns the number of outputed characters.

In order to reduce program code size, there is the **Project|Configure|C Compiler|Code Generation|(s)printf Features** option.
It allows linking different versions of the printf and sprintf functions, with only the features that are really required by the program.

The following **(s)printf features** are available:
- **int** - the following conversion type characters are supported: 'c', 's', 'p', 'i', 'd', 'u', 'x', 'X', '%', no width or precision specifiers are supported, only the '+' and ' ' flags are supported, no input size modifiers are supported
- **int, width** - the following conversion type characters are supported: 'c', 's', 'p', 'i', 'd', 'u', 'x', 'X', '%', the width specifier is supported, the precision specifier is not supported, only the '+', '-', '0' and ' ' flags are supported, no input size modifiers are supported
- **long, width** - the following conversion type characters are supported: 'c', 's', 'p', 'i', 'd', 'u', 'x', 'X', '%' the width specifier is supported, the precision specifier is not supported, only the '+', '-', '0' and ' ' flags are supported, only the 'l' input size modifier is supported
- **long, width, precision** - the following conversion type characters are supported: 'c', 's', 'p', 'i', 'd', 'u', 'x', 'X', '%', the width and precision specifiers are supported, only the '+', '-', '0' and ' ' flags are supported, only the 'l' input size modifier is supported
- **float, width, precision** - the following conversion type characters are supported: 'c', 's', 'p', 'i', 'd', 'u', 'e', 'E', 'f', 'x', 'X', '%', the width and precision specifiers are supported, only the '+', '-', '0' and ' ' flags are supported, only the 'l' input size modifier is supported.

The more features are selected, the larger is the code size generated for the printf and sprintf functions.

**int vprintf(char flash *fmtstr, va_list argptr)**

this function is identical to **printf** except that the **argptr** pointer, of **va_list** type, points to the variable list of arguments. The **va_list** type is defined in the stdarg.h header file.
The function returns the number of outputed characters.

**int vsprintf(char *str, char flash *fmtstr, va_list argptr)**

this function is identical to **sprintf** except that the **argptr** pointer, of **va_list** type, points to the variable list of arguments. The **va_list** type is defined in the stdarg.h header file.
The function returns the number of outputed characters.

**int vsnprintf(char *str, unsigned char size, char flash *fmtstr, va_list argptr)**
*for the TINY memory model.*

**int vsnprintf(char *str, unsigned int size, char flash *fmtstr, va_list argptr)**
*for the other memory models.*

this function is identical to **vsprintf** except that at most **size** (including the null terminator) characters are placed in the character string **str**.
The function returns the number of outputed characters.

**char *gets(char *str, unsigned char len)**

inputs, using **getchar**, the character string **str** terminated by the new line character.
The new line character will be replaced with 0.
The maximum length of the string is **len**. If **len** characters were read without encountering the new line character, then the string is terminated with 0 and the function ends.
The function returns a pointer to **str**.

**signed char scanf(char flash *fmtstr [ , arg1 address, arg2 address, ...])**

formatted text input by scanning, using **getchar**, a series of input fields according to the format specifiers in the **fmtstr** string.
The format specifier string **fmtstr** is constant and must be located in FLASH memory.
The implementation of **scanf** is a reduced version of the standard C function.
This was necessary due to the specific needs of an embedded system and because the full implementation would require a large amount of FLASH memory space.
The format specifier string has the following structure:

```
%[width][l]type_char
```

The optional **width** specifier sets the maximal number of characters to read. If the function encounters a whitespace character or one that cannot be converted, then it will continue with the next input field, if present.

The optional '**l**' input size modifier specifies that the function argument must be treated as a long int for the 'i', 'd', 'u', 'x' conversion type characters.

The **type_char** conversion type character is used to specify the way the input field will be processed.

The following conversion type characters are supported:
'd' - inputs a signed decimal integer in a pointer to int argument;
'i' - inputs a signed decimal integer in a pointer to int argument;
'u' - inputs an unsigned decimal integer in a pointer to unsigned int argument;
'x' - inputs an unsigned hexadecimal integer in a pointer to unsigned int argument;
'c' - inputs an ASCII character in a pointer to char argument;
's' - inputs an ASCII character string in a pointer to char argument;
'%' - no input is done, a '%' is stored.

The function returns the number of successful entries, or -1 on error.

**signed char sscanf(char *str, char flash *fmtstr [ , arg1 address, arg2 address, ...])**

this function is identical to **scanf** except that the formatted text is inputted from the null terminated character string **str**, located in RAM.

In order to reduce program code size, there is the **Project|Configure|C Compiler|Code Generation|(s)scanf Features** option.
It allows linking different versions of the scanf and sscanf functions, with only the features that are really required by the program.

The following **(s)scanf features** are available:

- **int, width** - the following conversion type characters are supported: 'c', 's', 'i', 'd', 'u', 'x', '%', the width specifier is supported, no input size modifiers are supported
- **long, width** - the following conversion type characters are supported: 'c', 's', 'i', 'd', 'u', 'x', '%' the width specifier is supported, only the 'l' input size modifier is supported.

The more features are selected, the larger is the code size generated for the scanf and sscanf functions.

The following Standard C Input/Output functions are used for file access on MMC/SD/SD HC FLASH memory cards.
Before using any of these functions, the logical drive that needs to be accessed must be mounted using the **f_mount** function and the appropriate file must be opened using the **f_open** function.

**int feof(FIL* fp)**

establishes if the end of file was reached during the last file access.
The **fp** pointer must point to a **FIL** type structure, defined in the **ff.h** header file, that was previously initialized using the **f_open** function.

If the end of file was reached, a non-zero value (1) will be returned.
Otherwise, the function will return 0.

**int ferror(FIL *fp)**

establishes if an error occurred during the last file access.
The **fp** pointer must point to a **FIL** type structure, defined in the **ff.h** header file, that was previously initialized using the **f_open** function.

If an error occurred, a non-zero value (1) will be returned.
Otherwise, the function will return 0.

**int fgetc(FIL *fp)**

reads a character from a file.
The **fp** pointer must point to a **FIL** type structure, defined in the **ff.h** header file, that was previously initialized using the **f_open** function.

On success the function returns a positive value: an 8bit character in the LSB, the MSB beeing 0.
In case of error or if the end of file was reached, the function returns the value of the predefined macro **EOF** (-1).
The **ferror** function must be used to establish if an error occured.
In order to check if the end of file was reached, the **feof** function must be called.

**char *fgets(char *str,unsigned int len,FIL *fp)**

reads from a file maximum **len**-1 characters to the char array pointed by **str.**
The read process stops if a new-line '\n' character is encountered or the end of file was reached. The new-line character is also saved in the char string.
The string is automatically NULL terminated after the read process is stopped.
The **fp** pointer must point to a **FIL** type structure, defined in the **ff.h** header file, that was previously initialized using the **f_open** function.

On success the function returns the same pointer as **str**.
If the end of file was encountered <u>and no characters were read</u>, a NULL pointer is returned.
The same happens in case of file access error.
The **ferror** function must be used to establish if an error occured.
In order to check if the end of file was reached, the **feof** function must be called.

**int fputc(char k,FIL* fp)**

writes the character **k** to a file.
The **fp** pointer must point to a **FIL** type structure, defined in the **ff.h** header file, that was previously initialized using the **f_open** function.

On success the function returns a positive value which represents the same character as **k**, the MSB beeing 0.
In case of error the function returns the value of the predefined macro **EOF** (-1).

**int fputs(char *str,FIL* fp)**

writes to a file the NULL terminated char string, stored in RAM, pointed by **str**.
The terminating NULL character is not written to the file.
The **fp** pointer must point to a **FIL** type structure, defined in the **ff.h** header file, that was previously initialized using the **f_open** function.

On success the function returns a positive value (1).
In case of error the function returns the value of the predefined macro **EOF** (-1).

**int fputsf(char flash *str,FIL* fp)**

writes to a file the NULL terminated char string, stored in FLASH memory, pointed by **str**.
The terminating NULL character is not written to the file.
The **fp** pointer must point to a **FIL** type structure, defined in the **ff.h** header file, that was previously initialized using the **f_open** function.

On success the function returns a positive value (1).
In case of error the function returns the value of the predefined macro **EOF** (-1).

**int fprintf(FIL *fp, char flash *fmtstr,...)**

this function is identical to **printf** except that the formatted text is written to a file.
The **fp** pointer must point to a **FIL** type structure, defined in the **ff.h** header file, that was previously initialized using the **f_open** function.

On success the function returns a positive value: the number of written characters.
In case of error the function returns the value of the predefined macro **EOF** (-1).

**int fscanf(FIL *fp, char flash *fmtstr,...)**

this function is identical to **scanf** except that the formatted text is read from a file.
The **fp** pointer must point to a **FIL** type structure, defined in the **ff.h** header file, that was previously initialized using the **f_open** function.

On success the function returns a positive value: the number of read entries.
In case of error the function returns the value of the predefined macro **EOF** (-1).

## 5.3 Standard Library Functions

The prototypes for these functions are placed in the file **stdlib.h**, located in the .\INC subdirectory. This file must be **#include -d** before using the functions.

**unsigned char cabs(signed char x)**

returns the absolute value of the byte x.

**unsigned int abs(int x)**

returns the absolute value of the integer x.

**unsigned long labs(long int x)**

returns the absolute value of the long integer x.

**float fabs(float x)**

returns the absolute value of the floating point number x.

**int atoi(char *str)**

converts the string str to integer.

**long int atol(char *str)**

converts the string str to long integer.

**void itoa(int n, char *str)**

converts the integer n to characters in string str.

**void ltoa(long int n, char *str)**

converts the long integer n to characters in string str.

**void ftoa(float n, unsigned char decimals, char *str)**

converts the floating point number n to characters in string str.
The number is represented with a specified number of decimals.

**void ftoe(float n, unsigned char decimals, char *str)**

converts the floating point number n to characters in string str.
The number is represented as a mantissa with a specified number of decimals and an integer power of 10 exponent (e.g. 12.35e-5).

**float atof(char *str)**

converts the characters from string str to floating point.

**int rand (void)**

generates a pseudo-random number between 0 and 32767.

**void srand(int seed)**

sets the starting value seed used by the pseudo-random number generator in the **rand** function.

**void *malloc(unsigned int size)**

allocates a memory block in the heap, with the length of size bytes.
On success the function returns a pointer to the start of the memory block, the block being filled with zeroes.
The allocated memory block occupies size+4 bytes in the heap.
This must be taken into account when specifying the **Heap size** in the **Project|Configure|C Compiler|Code Generation** menu.
If there wasn't enough contiguous free memory in the heap to allocate, the function returns a null pointer.

**void *calloc(unsigned int num, unsigned int size)**

allocates a memory block in the heap for an array of num elements, each element having the size length.
On success the function returns a pointer to the start of the memory block, the block being filled with zeroes.
If there wasn't enough contiguous free memory in the heap to allocate, the function returns a null pointer.

**void *realloc(void *ptr, unsigned int size)**

changes the size of a memory block allocated in the heap.
The ptr pointer must point to a block of memory previously allocated in the heap.
The size argument specifies the new size of the memory block.
On success the function returns a pointer to the start of the newly allocated memory block, the contents of the previously allocated block being copied to the newly allocated one.
If the newly allocated memory block is larger in size than the old one, the size difference is not filled with zeroes.
If there wasn't enough contiguous free memory in the heap to allocate, the function returns a null pointer.

**void free(void *ptr)**

frees a memory block allocated in the heap by the malloc, calloc or realloc functions and pointed by the ptr pointer.

After being freed, the memory block is available for new allocation.
If ptr is null then it is ignored.

## 5.4 Mathematical Functions

The prototypes for these functions are placed in the file **math.h**, located in the .\INC subdirectory. This file must be **#include -d** before using the functions.

**signed char cmax(signed char a, signed char b)**

returns the maximum value of bytes a and b.

**int max(int a, int b)**

returns the maximum value of integers a and b.

**long int lmax(long int a, long int b)**

returns the maximum value of long integers a and b.

**float fmax(float a, float b)**

returns the maximum value of floating point numbers a and b.

**signed char cmin(signed char a, signed char b)**

returns the minimum value of bytes a and b.

**int min(int a, int b)**

returns the minimum value of integers a and b.

**long int lmin(long int a, long int b)**

returns the minimum value of long integers a and b.

**float fmin(float a, float b)**

returns the minimum value of floating point numbers a and b.

**signed char csign(signed char x)**

returns -1, 0 or 1 if the byte x is negative, zero or positive.

**signed char sign(int x)**

returns -1, 0 or 1 if the integer x is negative, zero or positive

**signed char lsign(long int x)**

returns -1, 0 or 1 if the long integer x is negative, zero or positive.

**signed char fsign(float x)**

returns -1, 0 or 1 if the floating point number x is negative, zero or positive.

**unsigned char isqrt(unsigned int x)**

returns the square root of the unsigned integer x.

**unsigned int lsqrt(unsigned long x)**

returns the square root of the unsigned long integer x.

**float sqrt(float x)**

returns the square root of the positive floating point number x.

**float floor(float x)**

returns the smallest integer value of the floating point number x.

**float ceil(float x)**

returns the largest integer value of the floating point number x.

**float fmod(float x, float y)**

returns the remainder of x divided by y.

**float modf(float x, float *ipart)**

splits the floating point number x into integer and fractional components.
The fractional part of x is returned as a signed floating point number.
The integer part is stored as floating point number at ipart.

**float ldexp(float x, int expn)**

returns x * $2^{expn}$.

**float frexp(float x, int *expn)**

returns the mantissa and exponent of the floating point number x.

**float exp(float x)**

returns $e^x$ .

**float log(float x)**

returns the natural logarithm of the floating point number x.

**float log10(float x)**

returns the base 10 logarithm of the floating point number x.

**float pow(float x, float y)**

returns $x^y$ .

**float sin(float x)**

returns the sine of the floating point number x, where the angle is expressed in radians.

**float cos(float x)**

returns the cosine of the floating point number x, where the angle is expressed in radians.

**float tan(float x)**

returns the tangent of the floating point number x, where the angle is expressed in radians.

**float sinh(float x)**

returns the hyperbolic sine of the floating point number x, where the angle is expressed in radians.

**float cosh(float x)**

returns the hyperbolic cosine of the floating point number x, where the angle is expressed in radians.

**float tanh(float x)**

returns the hyperbolic tangent of the floating point number x, where the angle is expressed in radians.

**float asin(float x)**

returns the arc sine of the floating point number x (in the range -PI/2 to PI/2).
x must be in the range -1 to 1.

**float acos(float x)**

returns the arc cosine of the floating point number x (in the range 0 to PI).
x must be in the range -1 to 1.

**float atan(float x)**

returns the arc tangent of the floating point number x (in the range -PI/2 to PI/2).

**float atan2(float y, float x)**

returns the arc tangent of the floating point numbers y/x (in the range -PI to PI).

## 5.5 String Functions

The prototypes for these functions are placed in the file **string.h**, located in the .\INC subdirectory. This file must be **#include -d** before using the functions.

The string manipulation functions were extended to handle strings located both in RAM and FLASH memories.

**char *strcat(char *str1, char *str2)**

concatenate the string str2 to the end of the string str1.

**char *strcatf(char *str1, char flash *str2)**

concatenate the string str2, located in FLASH, to the end of the string str1.

**char *strncat(char *str1, char *str2, unsigned char n)**

concatenate maximum n characters of the string str2 to the end of the string str1.
Returns a pointer to the string str1.

**char *strncatf(char *str1, char flash *str2, unsigned char n)**

concatenate maximum n characters of the string str2, located in FLASH, to the end of the string str1.
Returns a pointer to the string str1.

**char *strchr(char *str, char c)**

returns a pointer to the first occurrence of the character c in the string str, else a NULL pointer.

**char *strrchr(char *str, char c)**

returns a pointer to the last occurrence of the character c in the string str, else a NULL pointer.

**signed char strpos(char *str, char c)**

returns the index to first occurrence of the character c in the string str, else -1.

**signed char strrpos(char *str, char c)**

returns the index to the last occurrence of the character c in the string str, else -1.

**signed char strcmp(char *str1, char *str2)**

compares the string str1 with the string str2.
Returns <0, 0, >0 according to str1<str2, str1=str2, str1>str2.

**signed char strcmpf(char *str1, char flash *str2)**

compares the string str1, located in RAM, with the string str2, located in FLASH.
Returns <0, 0, >0 according to str1<str2, str1=str2, str1>str2.

**signed char strncmp(char *str1, char *str2, unsigned char n)**

compares at most n characters of the string str1 with the string str2.
Returns <0, 0, >0 according to str1<str2, str1=str2, str1>str2.

**signed char strncmpf(char *str1, char flash *str2, unsigned char n)**

compares at most n characters of the string str1, located in RAM, with the string str2, located in FLASH.
Returns <0, 0, >0 according to str1<str2, str1=str2, str1>str2.

**char *strcpy(char *dest, char *src)**

copies the string src to the string dest.

**char *strcpyf(char *dest, char flash *src)**

copies the string src, located in FLASH, to the string dest, located in RAM.
Returns a pointer to the string dest.

**char *strncpy(char *dest, char *src, unsigned char n)**

copies at most n characters from the string src to the string dest.
Returns a pointer to the string dest.
If there is no null character among the first n characters of src, then dest will not be null terminated.
If n is less then the length of src, then the remainder of dest will be padded with nulls.

**char *strncpyf(char *dest, char flash *src, unsigned char n)**

copies at most n characters from the string src, located in FLASH, to the string dest, located in SRAM.
Returns a pointer to the string dest.
If there is no null character among the first n characters of src, then dest will not be null terminated.
If n is less then the length of src, then the remainder of dest will be padded with nulls.

**unsigned char strlcpy(char *dest, char *src, unsigned char n)**

copies at most n-1 characters from the string src to the string dest, ensuring that it will be always null terminated, even if the src string was truncated to fit.
If n is less then the length of src, then the remainder of dest after the null terminator, will not be padded with nulls.
The function returns the size of the src string.

**unsigned char strlcpyf(char *dest, char flash *src, unsigned char n)**

copies at most n-1 characters from the string src, located in FLASH, to the string dest located in SRAM, ensuring that it will be always null terminated, even if the src string was truncated to fit.
If n is less then the length of src, then the remainder of dest after the null terminator, will not be padded with nulls.
The function returns the size of the src string.

**unsigned char strspn(char *str, char *set)**

returns the index of the first character, from the string str, that doesn't match a character from the string set.
If all characters from set are in str returns the length of str.

**unsigned char strspnf(char *str, char flash *set)**

returns the index of the first character, from the string str, located in RAM, that doesn't match a character from the string set, located in FLASH.
If all characters from set are in str returns the length of str.

**unsigned char strcspn(char *str, char *set)**

searches the string str for the first occurrence of a character from the string set.
If there is a match returns, the index of the character in str.
If there are no matching characters, returns the length of str.

**unsigned char strcspnf(char *str, char flash *set)**

searches the string str for the first occurrence of a character from the string set, located in FLASH.
If there is a match, returns the index of the character in str.
If there are no matching characters, returns the length of str.

**char *strpbrk(char *str, char *set)**

searches the string str for the first occurrence of a char from the string set.
If there is a match, returns a pointer to the character in str.
If there are no matching characters, returns a NULL pointer.

**char *strpbrkf(char *str, char flash *set)**

searches the string str, located in RAM, for the first occurrence of a char from the string set, located in FLASH.
If there is a match, returns a pointer to the character in str.
If there are no matching characters, returns a NULL pointer.

**char *strrpbrk(char *str, char *set)**

searches the string str for the last occurrence of a character from the string set.
If there is a match, returns a pointer to the character in str.
If there are no matching characters, returns a NULL pointer.

**char *strrpbrkf(char *str, char flash *set)**

searches the string str, located in RAM, for the last occurrence of a character from the string set, located in FLASH.
If there is a match, returns a pointer to the character in str.
If there are no matching characters, returns a NULL pointer.

**char *strstr(char *str1, char *str2)**

searches the string str1 for the first occurrence of the string str2.
If there is a match, returns a pointer to the character in str1 where str2 begins.
If there is no match, returns a NULL pointer.

**char *strstrf(char *str1, char flash *str2)**

searches the string str1, located in RAM, for the first occurrence of the string str2, located in FLASH.
If there is a match, returns a pointer to the character in str1 where str2 begins.
If there is no match, returns a NULL pointer.

**char *strtok(char *str1, char flash *str2)**

scans the string str1, located in RAM, for the first token not contained in the string str2, located in FLASH.
The function considers the string str1 as consisting of a sequence of text tokens, separated by spans of one or more characters from the string str2.

The first call to strtok, with the pointer to str1 being different from NULL, returns a pointer to the first character of the first token in str1. Also a NULL character will be written in str1, immediately after the returned token.
Subsequent calls to strtok, with NULL as the first parameter, will work through the string str1 until no more tokens remain. When there are no more tokens, strtok will return a NULL pointer.

**unsigned char strlen(char *str)**

*for the TINY memory model.*
returns the length of the string str (in the range 0..255), excluding the null terminator.

**unsigned int strlen(char *str)**

*for the SMALL memory model.*
returns the length of the string str (in the range 0..65535 ), excluding the null terminator.

**unsigned int strlenf(char flash *str)**

returns the length of the string str located in FLASH, excluding the null terminator.

**void *memcpy(void *dest,void *src, unsigned char n)**

*for the TINY memory model.*

**void *memcpy(void *dest,void *src, unsigned int n)**

*for the SMALL memory model.*

Copies n bytes from src to dest. dest must not overlap src, else use **memmove**.
Returns a pointer to dest.

**void *memcpyf(void *dest,void flash *src, unsigned char n)**

*for the TINY memory model.*

**void *memcpyf(void *dest,void flash *src, unsigned int n)**

*for the SMALL memory model.*

Copies n bytes from src, located in FLASH, to dest. Returns a pointer to dest.

**void *memccpy(void *dest,void *src, char c, unsigned char n)**

*for the TINY memory model.*

**void *memccpy(void *dest,void *src, char c, unsigned int n)**

*for the SMALL memory model.*

Copies at most n bytes from src to dest, until the character c is copied.
dest must not overlap src.
Returns a NULL pointer if the last copied character was c or a pointer to dest+n+1.

**void *memmove(void *dest,void *src, unsigned char n)**

*for the TINY memory model.*

**void *memmove(void *dest,void *src, unsigned int n)**

*for the SMALL memory model.*

Copies n bytes from src to dest. dest may overlap src.
Returns a pointer to dest.

**void *memchr(void *buf, unsigned char c, unsigned char n)**

*for the TINY memory model.*

**void *memchr(void *buf, unsigned char c, unsigned int n)**

*for the SMALL memory model.*

Scans n bytes from buf for byte c.
Returns a pointer to c if found or a NULL pointer if not found.

**signed char memcmp(void *buf1,void *buf2, unsigned char n)**

*for the TINY memory model.*

**signed char memcmp(void *buf1,void *buf2, unsigned int n)**

*for the SMALL memory model.*

Compares at most n bytes of buf1 with buf2.
Returns <0, 0, >0 according to buf1<buf2, buf1=buf2, buf1>buf2.

**signed char memcmpf(void *buf1,void flash *buf2, unsigned char n)**

*for the TINY memory model.*

**signed char memcmpf(void *buf1,void flash *buf2, unsigned int n)**

*for the SMALL memory model.*

Compares at most n bytes of buf1, located in RAM, with buf2, located in FLASH.
Returns <0, 0, >0 according to buf1<buf2, buf1=buf2, buf1>buf2.

**void *memset(void *buf, unsigned char c, unsigned char n)**

*for the TINY memory model.*

**void *memset(void *buf, unsigned char c, unsigned int n)**

*for the SMALL memory model.*

Sets n bytes from buf with byte c. Returns a pointer to buf.

## 5.6 Variable Length Argument Lists Macros

These macros are defined in the file **stdarg.h**, located in the .\INC subdirectory. This file must be **#include -d** before using the macros.

**void va_start(argptr, previous_par)**

This macro, when used in a function with a variable length argument list, initializes the **argptr** pointer of **va_list** type, for subsequent use by the **va_arg** and **va_end** macros.
The **previous_par** argument must be the name of the function argument immediately preceding the optional arguments.
The **va_start** macro must be called prior to any access using the **va_arg** macro.

**type va_arg(argptr, type)**

This macro is used to extract successive arguments from the variable length argument list referenced by **argptr**.
**type** specifies the data type of the argument to extract.
The **va_arg** macro can be called only once for each argument. The order of the parameters in the argument list must be observed.
On the first call **va_arg** returns the first argument after the **previous_par** argument specified in the **va_start** macro. Subsequent calls to **va_arg** return the remaining arguments in succession.

**void va_end(argptr)**

This macro is used to terminate use of the variable length argument list pointer **argptr**, initialized using the **va_start** macro.

Example:

```
#include <stdarg.h>

/* declare a function with a variable number of arguments */
int sum_all(int nsum, ...)
{
va_list argptr;
int i, result=0;

/* initialize argptr */
va_start(argptr,nsum);

/* add all the function arguments after nsum */
for (i=1; i <= nsum; i++)
    /* add each argument */
    result+=va_arg(argptr,int);

/* terminate the use of argptr */
va_end(argptr);

return result;
}

void main(void)
{
int s;
/* calculate the sum of 5 arguments */
s=sum_all(5,10,20,30,40,50);
}
```

## 5.7 Non-local Jump Functions

These functions can execute a non-local goto.
They are usually used to pass control to an error recovery routine.
The prototypes for the non-local jump functions are placed in the file **setjmp.h**, located in the .\INC subdirectory. This file must be **#include -d** before using the functions.

**int setjmp(char *env)**

This function saves the current CPU state (Y, SP, SREG registers and the current instruction address) in the env variable.
The CPU state can then be restored by subsequently calling the **longjmp** function.
Execution is then resumed immediately after the **setjmp** function call.
The **setjmp** function will return 0 when the current CPU state is saved in the env variable.
If the function returns a value different from 0, it signals that a **longjmp** function was executed.
In this situation the returned value is the one that was passed as the retval argument to the **longjmp** function.
In order to preserve the local variables in the function where setjmp is used, these must be declared with the **volatile** attribute.

**void longjmp(char *env, int retval)**

This function restores the CPU state that was previously saved in the env variable by a call to **setjmp**.
The retval argument holds the integer non-zero value that will be returned by **setjmp** after the call to **longjmp**. If a 0 value is passed as the retval argument then it will be substituted with 1.

In order to facilitate the usage of these functions, the **setjmp.h** header file also contains the definition of the **jmp_buf** data type, which is used when declaring the env variables.

Example:

```
#include <mega8515.h>
#include <stdio.h>
#include <setjmp.h>

/* declare the variable used to hold the CPU state */
jmp_buf cpu_state;

void foo(void)
{
printf("Now we will make a long jump to main()\n\r");
longjmp(cpu_state,1);
}

/* ATmega8515 clock frequency [Hz] */
#define xtal 4000000L

/* Baud rate */
#define baud 9600

void main(void)
{
/* this local variable will be preserved after a longjmp */
volatile int i;
/* this local variable will not be preserved after a longjmp */
int j;
```

```
/* initialize the USART control register
   TX enabled, no interrupts, 8 data bits */
UCSRA=0x00;
UCSRB=0x08;
UCSRC=0x86;

/* initialize the USART's baud rate */
UBRRH=(xtal/16/baud-1) >> 8;
UBRRL=(xtal/16/baud-1) & 0xFF;

if (setjmp(cpu_state)==0)
   {
   printf("First call to setjmp\n\r");
   foo();
   }
else
   printf("We jumped here from foo()\n\r");
}
```

## 5.8 BCD Conversion Functions

The prototypes for these functions are placed in the file **bcd.h**, located in the .\INC subdirectory. This file must be **#include -d** before using the functions.

**unsigned char bcd2bin(unsigned char n)**

Converts the number n from BCD representation to its binary equivalent.

**unsigned char bin2bcd(unsigned char n)**

Converts the number n from binary representation to its BCD equivalent.
The number n values must be from 0 to 99.

## 5.9 Gray Code Conversion Functions

The prototypes for these functions are placed in the file **gray.h**, located in the .\INC subdirectory. This file must be **#include -d** before using the functions.

**unsigned char gray2binc(unsigned char n)**
**unsigned int gray2bin(unsigned int n)**
**unsigned long gray2binl(unsigned long n)**

Convert the number n from Gray code representation to its binary equivalent.

**unsigned char bin2grayc(unsigned char n)**
**unsigned int bin2gray(unsigned int n)**
**unsigned long bin2grayl(unsigned long n)**

Convert the number n from binary representation to its Gray code equivalent.

## 5.10 Memory Access Macros

The memory access macros are defined in the **mem.h** header file, located in the .\INC subdirectory.
This file must be **#include -d** before using these macros.

**pokeb(addr, data)**

this macro writes the **unsigned char** data to RAM at address addr.

**pokew(addr, data)**

this macro writes the **unsigned int** data to RAM at address addr.
The LSB is written at address addr and the MSB is written at address addr+1.

**peekb(unsigned int addr)**

this macro reads an **unsigned char** located in RAM at address addr.

**peekw (unsigned int addr)**

this macro reads an **unsigned int** located in RAM at address addr.
The LSB is read from address addr and the MSB is read from address addr+1.

## 5.11 Alphanumeric LCD Functions

## 5.11.1 LCD Functions for displays with up to 2x40 characters

The Alphanumeric LCD Functions are intended for easy interfacing between C programs and alphanumeric LCD modules built with the Hitachi HD44780 controller or equivalent.
The prototypes for these functions are placed in the file **alcd.h**, located in the .\INC subdirectory. This file must be **#include -d** before using the functions.
For LCD modules that use the Samsung KS0073 controller the **alcd_ks0073.h** header file must be used.

The LCD functions do support both the XMEGA and non-XMEGA chips.

The following LCD formats are supported in **alcd.h**: 1x8, 2x12, 3x12, 1x16, 2x16, 2x20, 4x20, 2x24 and 2x40 characters.

The allocation of LCD module signals to the I/O ports must be specified in the **Project|Configure|C Compiler|Libraries|Alphanumeric LCD** menu.
The LCD power supply and contrast control voltage must also be connected according to the module data sheet.

The low level LCD Functions are:

**void _lcd_write_data(unsigned char data)**

writes the byte data to the LCD instruction register.
This function may be used for modifying the LCD configuration.
Example:

```
/* enables the displaying of the cursor */
_lcd_write_data(0xe);
```

**void lcd_write_byte(unsigned char addr, unsigned char data);**

writes a byte to the LCD character generator or display RAM.

Example:

```
/* LCD user defined characters
   Chip: ATmega8515

   Use an 2x16 alphanumeric LCD connected
   to the STK600 PORTC header as follows:

   [LCD]   [STK600 PORTC HEADER]
    1 GND- 9  GND
    2 +5V- 10 VCC
    3 VLC- LCD HEADER Vo
    4 RS - 1  PC0
    5 RD - 2  PC1
    6 EN - 3  PC2
   11 D4 - 5  PC4
   12 D5 - 6  PC5
   13 D6 - 7  PC6
   14 D7 - 8  PC7

   The connections must be specified in the
   Project|Configure|C Compiler|Libraries|Alphanumeric LCD menu */
```

```
/* include the LCD driver routines */
#include <alcd.h>

typedef unsigned char byte;

/* table for the user defined character
   arrow that points to the top right corner */
flash byte char0[8]={
0b10000000,
0b10001111,
0b10000011,
0b10000101,
0b10001001,
0b10010000,
0b10100000,
0b11000000};

/* function used to define user characters */
void define_char(byte flash *pc,byte char_code)
{
byte i,a;
a=(char_code<<3) | 0x40;
for (i=0; i<8; i++) lcd_write_byte(a++,*pc++);
}

void main(void)
{
/* initialize the LCD for 2 lines & 16 columns */
lcd_init(16);

/* define user character 0 */
define_char(char0,0);

/* switch to writing in Display RAM */
lcd_gotoxy(0,0);
lcd_putsf("User char 0:");

/* display used defined char 0 */
lcd_putchar(0);

while (1); /* loop forever */
}
```

**unsigned char lcd_read_byte(unsigned char addr);**

reads a byte from the LCD character generator or display RAM.

The high level LCD Functions are:

**void lcd_init(unsigned char lcd_columns)**

initializes the LCD module, clears the display and sets the printing character position at row 0 and column 0. The numbers of columns of the LCD must be specified (e.g. 16). No cursor is displayed. This is the first function that must be called before using the other high level LCD Functions.

**void lcd_clear(void)**

clears the LCD and sets the printing character position at row 0 and column 0.

**void lcd_gotoxy(unsigned char x, unsigned char y)**

sets the current display position at column x and row y. The row and column numbering starts from 0.

**void lcd_putchar(char c)**

displays the character c at the current display position.

**void lcd_puts(char *str)**

displays at the current display position the string str, located in RAM.

**void lcd_putsf(char flash *str)**

displays at the current display position the string str, located in FLASH.

**void lcd_putse(char eeprom *str)**

displays at the current display position the string str, located in EEPROM.

## 5.11.2 LCD Functions for displays with 4x40 characters

The LCD Functions are intended for easy interfacing between C programs and alphanumeric LCD modules with 4x40 characters, built with the Hitachi HD44780 chip or equivalent.
The prototypes for these functions are placed in the file **lcd4x40.h**, located in the .\INC subdirectory. This file must be **#include -d** before using the functions.

The LCD functions do not yet support the XMEGA chips.

Prior to #include -ing the **lcd4x40.h** file, you must declare which microcontroller port is used for communication with the LCD module.
Example:

```
/* the LCD module is connected to PORTC */
#asm
    .equ __lcd_port=0x15
#endasm

/* now you can include the LCD Functions */
#include <lcd4x40.h>
```

The LCD module must be connected to the port bits as follows:

**[LCD]** **[AVR Port]**
RS (pin 11) --- bit 0
RD (pin 10) --- bit 1
EN1 (pin 9) ---- bit 2
EN2 (pin 15) -- bit 3
DB4 (pin 4) ---- bit 4
DB5 (pin 3) ---- bit 5
DB6 (pin 2) ---- bit 6
DB7 (pin 1) ---- bit 7

You must also connect the LCD power supply and contrast control voltage, according to the data sheet.

The low level LCD Functions are:

**void _lcd_ready(void)**

waits until the LCD module is ready to receive data.
This function must be called prior to writing data to the LCD with the **_lcd_write_data** function.

**void _lcd_write_data(unsigned char data)**

writes the byte data to the LCD instruction register.
This function may be used for modifying the LCD configuration.

Prior calling the low level functions **_lcd_ready** and **_lcd_write_data**, the global variable **_en1_msk** must be set to **LCD_EN1**, respectively **LCD_EN2**, to select the upper, respectively lower half, LCD controller.
Example:

```
/* enables the displaying of the cursor on the upper half
   of the LCD */
_en1_msk=LCD_EN1;
_lcd_ready();
_lcd_write_data(0xe);
```

**void lcd_write_byte(unsigned char addr, unsigned char data);**

writes a byte to the LCD character generator or display RAM.

**unsigned char lcd_read_byte(unsigned char addr);**

reads a byte from the LCD character generator or display RAM.

The high level LCD Functions are:

**unsigned char lcd_init(void)**

initializes the LCD module, clears the display and sets the printing character position at row 0 and column 0. No cursor is displayed.
The function returns 1 if the LCD module is detected and 0 if it is not.
This is the first function that must be called before using the other high level LCD Functions.

**void lcd_clear(void)**

clears the LCD and sets the printing character position at row 0 and column 0.

**void lcd_gotoxy(unsigned char x, unsigned char y)**

sets the current display position at column x and row y. The row and column numbering starts from 0.

**void lcd_putchar(char c)**

displays the character c at the current display position.

**void lcd_puts(char *str)**

displays at the current display position the string str, located in RAM.

**void lcd_putsf(char flash *str)**

displays at the current display position the string str, located in FLASH.

## 5.11.3 LCD Functions for displays connected in 8 bit memory mapped mode

These LCD Functions are intended for easy interfacing between C programs and alphanumeric LCD modules built with the Hitachi HD44780 chip or equivalent.
The LCD is connected to the AVR external data and address buses as an 8 bit peripheral.
This type of connection is used in the Kanda Systems STK200+ and STK300 development boards.
For the LCD connection, please consult the documentation that came with your development board.

The LCD functions do not yet support the XMEGA chips.

**These functions can be used only with AVR chips that allow using external memory devices.**

The prototypes for these functions are placed in the file **lcdstk.h**, located in the .\INC subdirectory.
This file must be **#include -d** before using the functions.
The following LCD formats are supported in **lcdstk.h**: 1x8, 2x12, 3x12, 1x16, 2x16, 2x20, 4x20, 2x24 and 2x40 characters.

The LCD Functions are:

**void _lcd_ready(void)**

waits until the LCD module is ready to receive data.
This function must be called prior to writing data to the LCD with the **_LCD_RS0** and **_LCD_RS1** macros.
Example:

```
/* enables the displaying of the cursor */
_lcd_ready();
_LCD_RS0=0xe;
```

The **_LCD_RS0**, respectively **_LCD_RS1**, macros are used for accessing the LCD Instruction Register with RS=0, respectively RS=1.

**void lcd_write_byte(unsigned char addr, unsigned char data);**

writes a byte to the LCD character generator or display RAM.

Example:

```
/* LCD user defined characters

   Chip: ATmegaS8515
   Memory Model: SMALL
   Data Stack Size: 128 bytes

   Use an 2x16 alphanumeric LCD connected
   to the STK200+ LCD connector */

/* include the LCD driver routines */
#include <lcdstk.h>

typedef unsigned char byte;
```

```
/* table for the user defined character
   arrow that points to the top right corner */
flash byte char0[8]={
0b10000000,
0b10001111,
0b10000011,
0b10000101,
0b10001001,
0b10010000,
0b10100000,
0b11000000};

/* function used to define user characters */
void define_char(byte flash *pc,byte char_code)
{
byte i,a;
a=(char_code<<3) | 0x40;
for (i=0; i<8; i++) lcd_write_byte(a++,*pc++);
}

void main(void)
{
/* initialize the LCD for 2 lines & 16 columns */
lcd_init(16);

/* define user character 0 */
define_char(char0,0);

/* switch to writing in Display RAM */
lcd_gotoxy(0,0);
lcd_putsf("User char 0:");

/* display used defined char 0 */
lcd_putchar(0);

while (1); /* loop forever */
}
```

**unsigned char lcd_read_byte(unsigned char addr);**

reads a byte from the LCD character generator or display RAM.

**unsigned char lcd_init(unsigned char lcd_columns)**

initializes the LCD module, clears the display and sets the printing character position at row 0 and column 0. The numbers of columns of the LCD must be specified (e.g. 16). No cursor is displayed.
The function returns 1 if the LCD module is detected and 0 if it is not.
This is the first function that must be called before using the other high level LCD Functions.

**void lcd_clear(void)**

clears the LCD and sets the printing character position at row 0 and column 0.

**void lcd_gotoxy(unsigned char x, unsigned char y)**

sets the current display position at column x and row y. The row and column numbering starts from 0.

**void lcd_putchar(char c)**

displays the character c at the current display position.

**void lcd_puts(char *str)**

displays at the current display position the string str, located in RAM.

**void lcd_putsf(char flash *str)**

displays at the current display position the string str, located in FLASH.

## 5.12 Graphic Display Functions

The Graphic Display Functions are intended for easy interfacing between C programs and graphic LCD, TFT and OLED modules built with a large variety of controllers.
The prototypes for these functions are placed in the file **glcd.h**, located in the .\INC subdirectory.
This file must be **#include -d** before using the functions.

Before using these functions, the type of the graphic display controller, the I/O port signals employed for communication with it and the display resolution must be specified in the **Project|Configure|C Compiler|Libraries|Graphic Display** menu.
The following graphic display controllers are supported:

- Ilitek ILI9325 (for color 240x320 TFT displays)
- Samsung KS0108 (equivalent: HD61202)
- Philips PCD8544
- Epson S1D13700
- S-MOS Systems SED1520 (equivalents: NJU6450, PT6520)
- Epson SED1335 (equivalent: RA8835)
- Epson SED1530
- Sunplus SPLC501C
- Sino Wealth SH1101A (for 132x64 OLED displays)
- Solomon Systech SSD1289 (for color 240x320 TFT displays)
- Solomon Systech SSD1303 (for 132x64 OLED displays)
- Solomon Systech SSD1963 (for color 320x240, 480x272 and 800x480 TFT displays)
- Solomon Systech SSD2119 (for color 320x240 TFT displays)
- Sitronix ST7565
- Sitronix ST7920
- Toshiba T6963C
- UltraChip UC1701
- Delcomp XG7100.

**Note:** The library functions for color TFT displays are supported only for the Advanced or Professional CodeVisionAVR licenses.

The Graphic Display functions do support both the XMEGA and non-XMEGA AVR chips.

The coordinate system employed by these functions has the origin (0,0) in the upper left corner of the display, with the x-coordinates increasing from left to right, and the y-coordinates increasing from top to bottom.
Coordinate clipping is performed, therefore no graphic data will be displayed for invalid coordinates.

The following helper data types are defined in the header file **glcd_types.h**:

- **GLCDX_t** type used for specifying the X horizontal display coordinate
- **GLCDY_t** type used for specifying the Y vertical display coordinate
- **GLCDDX_t** type used for specifying a horizontal displacement
- **GLCDDY_t** type used for specifying a vertical displacement
- **GLCDRAD_t** type used for specifying a circle radius
- **GLCDCOL_t** type used for specifying foreground and background display colors
- **GLCDMEMADDR_t** type used for specifying RAM, EEPROM, FLASH or external memory addresses for bitmap image storage

- **GLCDBLOCKMODE_t** enumeration type used for specifying the read/write modes for the **glcd_block** function:

```
typedef enum
        {
        GLCD_PUTCOPY, /* copy a bitmap from memory to display
                         overwriting previous display data */
        GLCD_PUTXOR,  /* copy a bitmap from memory to display
                         performing a XOR with previous display data */
        GLCD_PUTOR,   /* copy a bitmap from memory to display
                         performing an OR with previous display data */
        GLCD_PUTNOT,  /* copy a bitmap from memory to display
                         performing a bit negation */
        GLCD_PUTAND,  /* copy a bitmap from memory to display
                         performing an AND with previous display data */
        GLCD_PUTTP,   /* used for displaying image in tranparent mode */
        GLCD_GET      /* used for storing a block of data from
                         specified coordinates to memory */
        GLCD_PUTCHAR, /* used internally by the glcd_putchar function */
        GLCD_PUTFILL, /* used internally by the rectangular area fill
                         function */
        GLCD_CLEARBLOCK, /* used internally by the rectangular area
                            clear function */
        GLCD_SETBLOCK, /* used internally by the rectangular area set
                          function */
        } GLCDBLOCKMODE_t;
```

**Notes:**

- The **GLCD_PUTXOR**, **GLCD_PUTOR**, **GLCD_PUTNOT**, **GLCD_PUTAND** modes are supported only for displays with 2 colors (monochrome).
- The **GLCD_PUTTP** mode is supported only for displays with more than 2 colors.

- **GLCDMEMADDR_t** type used for specifying RAM, EEPROM, FLASH or external memory addresses for bitmap image storage

- **GLCDTEXT_t** structure type used for specifying the text displaying parameters:

```
typedef struct
        {
        flash unsigned char *font; /* current font */
        unsigned char horiz;    /* horizontal justification in pixels */
        unsigned char vert;     /* vertical justification in pixels */
        unsigned char transparent:1; /* enable transparent text
                                        display mode */
        } GLCDTEXT_t;
```

**Note:** The **transparent** structure member is defined only for displays with more than 2 colors.

- **GLCDLINE_t** structure type used for specifying the line drawing parameters:

```
typedef struct
        {
        unsigned char thick;   /* line thickness */
        unsigned char pattern; /* bit pattern */
        } GLCDLINE_t;
```

- **GLCDPOINT_t** structure type used for specifying point coordinates:

```
typedef struct
        {
        GLCDX_t x;
        GLCDY_t y;
        } GLCDPOINT_t;
```

- **GLCDFILL_t** structure type used for specifying the fill style parameters:

```
typedef struct
        {
        GLCDCOL_t color; /* fill color */

        /* bit pattern for filling a rectangular area */
        unsigned char pattern[_GLCD_FILL_PATTERN_WIDTH_];
        } GLCDFILL_t;
```

- **GLCDARCCOORDS_t** structure type used for specifying the arc coordinates:

```
typedef struct
        {
        GLCDX_t x;      /* arc x center coordinate */
        GLCDY_t y;      /* arc y center coordinate */
        GLCDX_t xstart; /* arc start x coordinate */
        GLCDY_t ystart; /* arc start y coordinate */
        GLCDX_t xend;   /* arc end x coordinate */
        GLCDY_t yend;   /* arc end y coordinate */
        } GLCDARCCOORDS_t;
```

- **GLCDINIT_t** structure type used for specifying various parameters used for initializing the display controller:

```
typedef struct
        {
        flash unsigned char *font; /* default font after initialization */

        /* pointer to the function used for reading a byte
           from external memory */
        unsigned char (*readxmem) (GLCDMEMADDR_t addr);

        /* pointer to the function used for writing a byte
           to external memory */
        void (*writexmem) (GLCDMEMADDR_t addr, unsigned char data);

        /* additional structure members which are specific
           to each display controller... */

        } GLCDINIT_t;
```

- **GLCDSTATE_t** structure type used for specifying the graphic display state:

```
typedef struct
        {
        GLCDCOL_t fgcolor;   /* foreground color */
        GLCDCOL_t bkcolor;   /* background color */
        GLCDCOL_t tpcolor;   /* transparency color */
        GLCDX_t cx;          /* current x horizontal coordinate */
        GLCDY_t cy;          /* current y vertical coordinate */
        GLCDTEXT_t text;     /* current text display settings */
        GLCDLINE_t line;     /* current line display settings */
        GLCDARCCOORDS_t arc; /* coordinates of last displayed arc */
        GLCDFILL_t fill;     /* current fill display settings */

        /* pointer to the function used for reading a byte
           from external memory */
        unsigned char (*readxmem) (GLCDMEMADDR_t addr);

        /* pointer to the function used for writing a byte
           to external memory */
        void (*writexmem) (GLCDMEMADDR_t addr, unsigned char data);
        } GLCDSTATE_t;
```

**Note:** The **tpcolor** transparency color member is available only for displays with more than 2 colors.

- **GLCDMEM_t** enumeration type used for specifying the kind of memory access:

```
typedef enum
        {
        GLCD_MEM_RAM,    /* RAM access */
        GLCD_MEM_FLASH,  /* FLASH access */
        GLCD_MEM_EEPROM, /* EEPROM access */
        GLCD_MEM_EXT     /* external memory accessed
                            using special functions */
        } GLCDMEM_t;
```

The following macros are defined in the header file **glcd_types.h** :

- **GLCD_LINE_SOLID** solid line pattern
- **GLCD_LINE_DOT_SMALL** small dots line pattern
- **GLCD_LINE_DOT_LARGE** large dots line pattern
- **_GLCD_FILL_PATTERN_WIDTH_** width of the fill pattern measured in pixels
- **_GLCD_MAXCOLOR_** highest color value for a pixel in ON state.

The following macros are predefined by the compiler based on the settings from the **Project|Configure|C Compiler|Libraries|Graphic Display** menu:

- **_GLCD_MAXX_** horizontal display resolution measured in pixels
- **_GLCD_MAXY_** vertical display resolution measured in pixels
- **_GLCD_BYTEY_** signals that the display controller uses vertical byte organization for accessing the display RAM
- **_GLCD_INTERNAL_FONT_WIDTH_** specifies the width of the internal character generator's font, measured in pixels
- **_GLCD_INTERNAL_FONT_HEIGHT_** specifies the height of the internal character generator's font, measured in pixels
- **_GLCD_CTRL_ILI9325_** signals that the display controller type is Ilitek ILI9325
- **_GLCD_CTRL_KS0108_** signals that the display controller type is Samsung KS0108
- **_GLCD_CTRL_SED1335_** signals that the display controller type is Epson SED1335
- **_GLCD_CTRL_SED1520_** signals that the display controller type is S-MOS Systems SED1520

- **_GLCD_CTRL_SED1530_** signals that the display controller type is Epson SED1520
- **_GLCD_CTRL_S1D13700_** signals that the display controller type is Epson S1D13700
- **_GLCD_CTRL_ST7565_** signals that the display controller type is Sitronix ST7565
- **_GLCD_CTRL_ST7920_** signals that the display controller type is Sitronix ST7920
- **_GLCD_CTRL_SPLC501_** signals that the display controller type is Sunplus SPLC501C
- **_GLCD_CTRL_SSD1289_** signals that the display controller type is Solomon Systech SSD1289
- **_GLCD_CTRL_SSD1303_** signals that the display controller type is Solomon Systech SSD1303 or the equivalent Sino Wealth SH1101A
- **_GLCD_CTRL_SSD1963_** signals that the display controller type is Solomon Systech SSD1963
- **_GLCD_CTRL_SSD2119_** signals that the display controller type is Solomon Systech SSD2119
- **_GLCD_CTRL_T6963_** signals that the display controller type is Toshiba T6963C
- **_GLCD_CTRL_UC1701_** signals that the display controller type is UltraChip UC1701
- **_GLCD_CTRL_PCD8544_** signals that the display controller type is Philips PCD8544
- **_GLCD_CTRL_XG7100_** signals that the display controller type is Delcomp XG7100.

**Note:** The header file **glcd_types.h** is automatically *#include*-d by **glcd.h**, so there is no need to *#include* it directly. The same applies to the header file **graphics.h** where the high level graphic display functions are declared.

The following high level graphic display functions are available:

**bool glcd_init(GLCDINIT_t *init_data)**

initializes the graphic display controller and performs the following initializations of the graphic system:

- clears the display
- sets the current plot coordinates to (0,0)
- sets the current font used for displaying text as specified by the **font** member of the structure pointed by **init_data**
- sets the current background color to 0
- sets the current foreground color to **_GLCD_MAXCOLOR_**
- sets the current transparency color to 0 (for displays with more than 2 colors)
- sets the current text horizontal justification to 1 pixel
- sets the current text vertical justification to 1 pixel
- sets the current line width to 1 pixel
- sets the transparent text mode display to OFF for LCDs with more than 2 colors
- sets the current line pattern to solid line
- sets the current line color to **_GLCD_MAXCOLOR_**
- sets the current fill pattern to solid
- sets the current fill color to **_GLCD_MAXCOLOR_**
- sets the pointers to the external memory read and write functions.

*Parameter:*

**init_data** points to a **GLCDINIT_t** structure that specifies various parameters used for initializing the display controller, including the font used for displaying text.
If the **font** member of the structure pointed by **init_data** is NULL, then the internal character generator (if present) of the display controller will be used for displaying text.
In this situation the horizontal and vertical text justification settings will have no effect, the text will be aligned to character cell boundaries specific to the controller.
If **init_data** is NULL, then the default settings for the specific display controller will be used, including the internal character generator, if present.

*Return values:*

**true** on success, **false** in case of error.

**Note:** A 5x7 pixel font is supplied as standard with CodeVisionAVR.

In order to use this font, the **font5x7.h** header file must be *#include*-d and the **GLCDINIT_t** structure member **font** must be initialized with the **font5x7** array address, when **glcd_init** is called.
Additional fonts can be created using the LCD Vision font editor, supplied with the Advanced version of CodeVisionAVR.

Example:

```
/* Include the graphic display driver functions.
   The display controller type and connections must be specified in
   the Project|Configure|C Compiler|Libraries|Graphic Display menu. */
#include <glcd.h>

/* Include the font definition */
#include <font5x7.h>

/* Function used for reading image data from external memory */
unsigned char read_ext_memory(GLCDMEMADDR_t addr)
{
unsigned char data;
/* Place your code here ... */

return data;
}

/* Function used for writing image data to external memory */
void write_ext_memory(GLCDMEMADDR_t addr, unsigned char data)
{
/* Place your code here ... */

}

void main(void)
{
GLCDINIT_t init;

/* Specify the current font */
init.font=font5x7;

/* Specify the function used for reading data from external memory.
   If not used, set value to NULL */
init.readxmem=read_ext_memory;

/* Specify the function used for writing data to external memory.
   If not used, set value to NULL */
init.writexmem=write_ext_memory;

/* Initialize the display controller and graphics */
glcd_init(&init);

/* Follows the rest of the code ... */
}
```

**void glcd_display(bool on)**

Turns display on/off.

*Parameter:*

**on** specifies display on/off state.

**void glcd_setcolor(GLCDCOL_t foreground_color)**

Sets the current foreground color that will be used for displaying text and graphics.

*Parameter:*

**foreground_color** specifies the foreground color.

**void glcd_setbkcolor(GLCDCOL_t background_color)**

Sets the current background color that will be used for displaying text and graphics.

*Parameter:*

**background_color** specifies the background color.

**void glcd_settpcolor(GLCDCOL_t transparent_color)**

Sets the transparency color for image displaying in transparent mode.

*Parameter:*

**transparent_color** specifies the transparency color.
When an image pixel with this color must be displayed in transparent mode **GLCD_PUTTP**, the background color at the pixel coordinates will be used instead.

**Note:** This function is available only for displays with more than 2 colors.

**GLCDCOL_t glcd_getcolor(void)**

Returns the display's current foreground color.

**GLCDCOL_t glcd_getbkcolor(void)**

Returns the display's current background color.

**GLCDCOL_t glcd_gettpcolor(void)**

Returns the current transparency color for image displaying in transparent mode.
When an image pixel with this color must be displayed in transparent mode **GLCD_PUTTP**, the background color at the pixel coordinates will be used instead.

**Note:** This function is available only for displays with more than 2 colors.

**GLCDCOL_t glcd_getmaxcolor(void)**

Returns the highest color value for a pixel in ON state.

**GLCDX_t glcd_getmaxx(void)**

Returns the maximum X horizontal coordinate value.

**GLCDY_t glcd_getmaxy(void)**

Returns the maximum Y horizontal coordinate value.

**void glcd_clear(void)**

Clears the display by setting it's color to the current background color.

**void glcd_putpixel(GLCDX_t x, GLCDY_t y, GLCDCOL_t color)**

Sets the color of the pixel at specified coordinates.
**Note:** The current pixel plot coordinates are not affected by this function.

*Parameters:*

**x** specifies the horizontal pixel coordinate
**y** specifies the vertical pixel coordinate
**color** specifies the color that must be assigned to the pixel.

**void glcd_setpixel(GLCDX_t x, GLCDY_t y)**

Sets the color of the pixel at specified coordinates to the current foreground color.
**Note:** The current pixel plot position coordinates are not affected by this function.

*Parameters:*

**x** specifies the horizontal pixel coordinate
**y** specifies the vertical pixel coordinate.

**void glcd_clrpixel(GLCDX_t x, GLCDY_t y)**

Sets the color of the pixel at specified coordinates to the current background color.
**Note:** The current pixel plot position coordinates are not affected by this function.

*Parameters:*

**x** specifies the horizontal pixel coordinate
**y** specifies the vertical pixel coordinate.

**GLCDCOL_t glcd_getpixel(GLCDX_t x, GLCDY_t y)**

Returns the color of the pixel at specified coordinates. If the pixel coordinates are outside the display area, the returned color will be 0.
**Note:** The current pixel plot position coordinates are not affected by this function.

*Parameters:*

**x** specifies the horizontal pixel coordinate
**y** specifies the vertical pixel coordinate.

**void glcd_moveto(GLCDX_t x, GLCDY_t y)**

Moves the current pixel plot position to the specified coordinates.

*Parameters:*

**x** specifies the horizontal pixel coordinate
**y** specifies the vertical pixel coordinate.

**void glcd_moverel(GLCDDX_t dx, GLCDDY_t dy)**

Moves the current pixel plot position to a new relative position.

*Parameters:*

**dx** specifies the horizontal displacement relative to the current pixel plot position
**dy** specifies the vertical displacement relative to the current pixel plot position.

**GLCDX_t glcd_getx(void)**

Returns the value of the current pixel plot position horizontal coordinate.

**GLCDY_t glcd_gety(void)**

Returns the value of the current pixel plot position vertical coordinate.

**void glcd_setfont(flash unsigned char *font_name)**

Specifies the current font used for displaying text.

*Parameters:*

**font_name** points to an array located in FLASH memory, that holds the font definition.

**void glcd_settextjustify(unsigned char horiz, unsigned char vert)**

Sets the horizontal and vertical text justification values.

*Parameters:*

**horiz** specifies the horizontal spacing between displayed characters, measured in pixels
**vert** specifies the vertical spacing between displayed characters, measured in pixels.

**unsigned char glcd_charwidth(char c)**

Returns the width (in pixels) of a character for the current font, including the horizontal justification.

*Parameter:*

**c** specifies the code of the character for which the width must be returned.

**unsigned char glcd_textheight(void)**

Returns the text height (in pixels) for the current font, including the vertical justification.

**GLCDX_t glcd_textwidth(char *str)**

Returns the text width (in pixels) of a NULL terminated literal char string located in RAM for the current font, including the horizontal justification.

*Parameter:*

**str** pointer to the literal char string.

**GLCDX_t glcd_textwidthf(flash char *str)**

Returns the text width (in pixels) of a NULL terminated literal char string located in FLASH for the current font, including the horizontal justification.

*Parameter:*

**str** pointer to the literal char string.

**GLCDX_t glcd_textwidthe(eeprom char *str)**

Returns the text width (in pixels) of a NULL terminated literal char string located in EEPROM for the current font, including the horizontal justification.

*Parameter:*

**str** pointer to the literal char string.

**void glcd_transparent(bool on)**

Controls displaying text in transparent mode.
If this mode is enabled, the background color of the area where the character is displayed is preserved, otherwise the color specified by **glcd_setbkcolor** is used.

*Parameter:*

**on** enables or disables displaying text in transparent mode.

**Note:** The **glcd_transparent** function is defined only for displays with more than 2 colors.

**void glcd_putcharxy(GLCDX_t x, GLCDY_t y, char c)**

Displays a character using the current font at the specified coordinates.
After the character is displayed, the current horizontal pixel plot coordinate is increased to the next display position by the width of the character + horizontal text justification.
If the new horizontal pixel plot coordinate will result outside the right display margin, then the new horizontal coordinate will be set to 0 and the vertical pixel plot coordinate will be increased by the height of the font + vertical text justification.

*Parameters:*

**x** specifies the horizontal coordinate of the left top corner of the displayed character
**y** specifies the vertical coordinate of the left top corner of the displayed character
**c** specifies the code of the character that must be displayed.

**void glcd_putchar(char c)**

Displays a character using the current font at the current pixel plot position.

*Parameter:*

**c** specifies the code of the character that must be displayed.

**void glcd_outtextxy(GLCDX_t x, GLCDY_t y, char *str)**

Displays a NULL terminated literal char string located in RAM at the specified coordinates.
The new display position will be located at the end of the displayed text.

*Parameters:*

**x** specifies the horizontal coordinate of the left top corner of the first displayed character
**y** specifies the vertical coordinate of the left top corner of the first displayed character
**str** pointer to the literal char string.

**void glcd_outtextxyf(GLCDX_t x, GLCDY_t y, flash char *str)**

Displays a NULL terminated literal char string located in FLASH at the specified coordinates.
The new display position will be located at the end of the displayed text.

*Parameters:*

**x** specifies the horizontal coordinate of the left top corner of the first displayed character
**y** specifies the vertical coordinate of the left top corner of the first displayed character
**str** pointer to the literal char string.

**void glcd_outtextxye(GLCDX_t x, GLCDY_t y, eeprom char *str)**

Displays a NULL terminated literal char string located in EEPROM at the specified coordinates.
The new display position will be located at the end of the displayed text.

*Parameters:*

**x** specifies the horizontal coordinate of the left top corner of the first displayed character
**y** specifies the vertical coordinate of the left top corner of the first displayed character
**str** pointer to the literal char string.

**void glcd_outtext(char *str)**

Displays a NULL terminated literal char string located in RAM at the current display position.
The new display position will be located at the end of the displayed text.

*Parameter:*

**str** pointer to the literal char string.

**void glcd_outtextf(char *str)**

Displays a NULL terminated literal char string located in FLASH at the current display position.
The new display position will be located at the end of the displayed text.

*Parameter:*

**str** pointer to the literal char string.

**void glcd_outtexte(char *str)**

Displays a NULL terminated literal char string located in EEPROM at the current display position.
The new display position will be located at the end of the displayed text.

*Parameter:*

**str** pointer to the literal char string.

**void glcd_block(GLCDX_t left, GLCDY_t top, GLCDX_t width, GLCDY_t height, GLCDMEM_t memt, GLCDMEMADDR_t addr, GLCDBLOCKMODE_t mode)**

Writes/reads a block of bytes to/from a rectangular graphics display area at specified coordinates.

*Parameters:*

**left** specifies the horizontal coordinate of the left top corner of the rectangular display area
**top** specifies the vertical coordinate of the left top corner of the rectangular display area
**width** specifies the horizontal size of the rectangular display area
**height** specifies the vertical size of the rectangular display area
**memt** specifies the memory type to/from which the data will be written/read.
It may take one of the following values:

- **GLCD_MEM_RAM** RAM access
- **GLCD_MEM_FLASH** FLASH access
- **GLCD_MEM_EEPROM** EEPROM access
- **GLCD_MEM_EXT** external memory accessed using special functions.

**addr** specifies the memory address to/from which the data will be written/read
**mode** specifies the operation to be performed:

- **GLCD_PUTCOPY** copy a bitmap from memory to display overwriting previous display data
- **GLCD_PUTTP** copy a bitmap from memory to display overwriting previous display data using transparent mode (when an image pixel with the current transparency color must be displayed, the background color at the pixel coordinates will be used instead)
- **GLCD_PUTXOR** copy a bitmap from memory to display performing a XOR with previous display data
- **GLCD_PUTOR** copy a bitmap from memory to display performing an OR with previous display data
- **GLCD_PUTNOT** copy a bitmap from memory to display performing a bit negation
- **GLCD_PUTAND** copy a bitmap from memory to display performing an AND with previous display data
- **GLCD_GET** read a block of data from specified coordinates.

**Notes:**

- The **glcd_block** function doesn't access the text overlay display, if present.
- The **GLCD_PUTXOR**, **GLCD_PUTOR**, **GLCD_PUTNOT**, **GLCD_PUTAND** modes are supported only for displays with 2 colors (monochrome).
- The **GLCD_PUTTP** mode is supported only for displays with more than 2 colors.

**unsigned long glcd_imagesize(GLCDX_t width, GLCDY_t height)**

Returns the memory size in bytes needed to store a rectangular bitmap image.

*Parameters:*

**width** specifies the horizontal size of the rectangular display area
**height** specifies the vertical size of the rectangular display area.

*Return values:*

image size or 0 if the **width** or **height** values are not valid.

**unsigned long glcd_putimage(GLCDX_t left, GLCDY_t top, unsigned char *pimg, GLCDBLOCKMODE_t mode)**

Displays a bitmap image located in RAM at specified coordinates.

*Parameters:*

**left** specifies the horizontal coordinate of the left top corner of the image
**top** specifies the vertical coordinate of the left top corner of the image
**pimg** pointer to the image data, which is located in RAM
**mode** specifies how the display operation must be performed:

- **GLCD_PUTCOPY** copy a bitmap from memory to display overwriting previous display data
- **GLCD_PUTTP** copy a bitmap from memory to display overwriting previous display data using transparent mode (when an image pixel with the current transparency color must be displayed, the background color at the pixel coordinates will be used instead)
- **GLCD_PUTXOR** copy a bitmap from memory to display performing a XOR with previous display data
- **GLCD_PUTOR** copy a bitmap from memory to display performing an OR with previous display data
- **GLCD_PUTNOT** copy a bitmap from memory to display performing a bit negation
- **GLCD_PUTAND** copy a bitmap from memory to display performing an AND with previous display data.

*Return values:*

image size or 0 if the coordinate values are not valid.

**Notes:**
- The **GLCD_PUTXOR**, **GLCD_PUTOR**, **GLCD_PUTNOT**, **GLCD_PUTAND** modes are supported only for displays with 2 colors (monochrome).
- The **GLCD_PUTTP** mode is supported only for displays with more than 2 colors.

**unsigned long glcd_putimagef(GLCDX_t left, GLCDY_t top, flash unsigned char *pimg, GLCDBLOCKMODE_t mode)**

Displays a bitmap image located in FLASH at specified coordinates.

*Parameters:*

**left** specifies the horizontal coordinate of the left top corner of the image
**top** specifies the vertical coordinate of the left top corner of the image
**pimg** pointer to the image data, which is located in FLASH
**mode** specifies how the display operation must be performed:

- **GLCD_PUTCOPY** copy a bitmap from memory to display overwriting previous display data
- **GLCD_PUTTP** copy a bitmap from memory to display overwriting previous display data using transparent mode (when an image pixel with the current transparency color must be displayed, the background color at the pixel coordinates will be used instead)
- **GLCD_PUTXOR** copy a bitmap from memory to display performing a XOR with previous display data
- **GLCD_PUTOR** copy a bitmap from memory to display performing an OR with previous display data
- **GLCD_PUTNOT** copy a bitmap from memory to display performing a bit negation
- **GLCD_PUTAND** copy a bitmap from memory to display performing an AND with previous display data.

*Return values:*

image size or 0 if the coordinate values are not valid.

**Notes:**
- The **GLCD_PUTXOR**, **GLCD_PUTOR**, **GLCD_PUTNOT**, **GLCD_PUTAND** modes are supported only for displays with 2 colors (monochrome).
- The **GLCD_PUTTP** mode is supported only for displays with more than 2 colors.

**unsigned long glcd_putimagee(GLCDX_t left, GLCDY_t top, eeprom unsigned char *pimg, GLCDBLOCKMODE_t mode)**

Displays a bitmap image located in EEPROM at specified coordinates.

*Parameters:*

**left** specifies the horizontal coordinate of the left top corner of the image
**top** specifies the vertical coordinate of the left top corner of the image
**pimg** pointer to the image data, which is located in EEPROM
**mode** specifies how the display operation must be performed:

- **GLCD_PUTCOPY** copy a bitmap from memory to display overwriting previous display data
- **GLCD_PUTTP** copy a bitmap from memory to display overwriting previous display data using transparent mode (when an image pixel with the current transparency color must be displayed, the background color at the pixel coordinates will be used instead)
- **GLCD_PUTXOR** copy a bitmap from memory to display performing a XOR with previous display data
- **GLCD_PUTOR** copy a bitmap from memory to display performing an OR with previous display data
- **GLCD_PUTNOT** copy a bitmap from memory to display performing a bit negation
- **GLCD_PUTAND** copy a bitmap from memory to display performing an AND with previous display data.

*Return values:*

image size or 0 if the coordinate values are not valid.

**Notes:**
- The **GLCD_PUTXOR**, **GLCD_PUTOR**, **GLCD_PUTNOT**, **GLCD_PUTAND** modes are supported only for displays with 2 colors (monochrome).
- The **GLCD_PUTTP** mode is supported only for displays with more than 2 colors.

**unsigned long glcd_putimagex(GLCDX_t left, GLCDY_t top, GLCDMEMADDR_t addr, GLCDBLOCKMODE_t mode)**

Displays a bitmap image located in external memory at specified coordinates.
**Note:** External memory read must be implemented through an user defined function, specified during initialization by **glcd_init**.

*Parameters:*

**left** specifies the horizontal coordinate of the left top corner of the image
**top** specifies the vertical coordinate of the left top corner of the image
**addr** specifies the external memory address from which the image will be read
**mode** specifies how the display operation must be performed:

- **GLCD_PUTCOPY** copy a bitmap from memory to display overwriting previous display data
- **GLCD_PUTTP** copy a bitmap from memory to display overwriting previous display data using transparent mode (when an image pixel with the current transparency color must be displayed, the background color at the pixel coordinates will be used instead)
- **GLCD_PUTXOR** copy a bitmap from memory to display performing a XOR with previous display data
- **GLCD_PUTOR** copy a bitmap from memory to display performing an OR with previous display data
- **GLCD_PUTNOT** copy a bitmap from memory to display performing a bit negation
- **GLCD_PUTAND** copy a bitmap from memory to display performing an AND with previous display data.

*Return values:*

image size or 0 if the coordinate values are not valid.

**Notes:**
- The **GLCD_PUTXOR**, **GLCD_PUTOR**, **GLCD_PUTNOT**, **GLCD_PUTAND** modes are supported only for displays with 2 colors (monochrome).
- The **GLCD_PUTTP** mode is supported only for displays with more than 2 colors.

**unsigned long glcd_getimage(GLCDX_t left, GLCDY_t top, GLCDX_t width, GLCDY_t height, unsigned char *pimg)**

Saves a rectangular display area to RAM as a bitmapped image.

*Parameters:*

**left** specifies the horizontal coordinate of the left top corner of the rectangular display area
**top** specifies the vertical coordinate of the left top corner of the rectangular display area
**width** specifies the horizontal size of the rectangular display area
**height** specifies the vertical size of the rectangular display area
**pimg** points to the byte array that will hold the image data.

*Return values:*

image size or 0 if the coordinate values are not valid.

**unsigned long glcd_getimagee(GLCDX_t left, GLCDY_t top, GLCDX_t width, GLCDY_t height, eeprom unsigned char *pimg)**

Saves a rectangular display area to EEPROM as a bitmapped image.

*Parameters:*

**left** specifies the horizontal coordinate of the left top corner of the rectangular display area
**top** specifies the vertical coordinate of the left top corner of the rectangular display area
**width** specifies the horizontal size of the rectangular display area
**height** specifies the vertical size of the rectangular display area
**pimg** points to the byte array that will hold the image data.

*Return values:*

image size or 0 if the coordinate values are not valid.

**unsigned long glcd_getimagex(GLCDX_t left, GLCDY_t top, GLCDX_t width, GLCDY_t height, GLCDMEMADDR_t addr)**

Saves a rectangular display area to external memory as a bitmapped image.
**Note:** External memory write must be implemented through an user defined function, specified during initialization by **glcd_init**.

*Parameters:*

**left** specifies the horizontal coordinate of the left top corner of the rectangular display area
**top** specifies the vertical coordinate of the left top corner of the rectangular display area
**width** specifies the horizontal size of the rectangular display area
**height** specifies the vertical size of the rectangular display area
**addr** specifies the external memory address to which the image will be written.

*Return values:*

image size or 0 if the coordinate values are not valid.

**void glcd_setlinestyle(unsigned char thickness, unsigned char bit_pattern)**

Sets the current line displaying style.

*Parameters:*

**thickness** specifies the thickness of the lines to be drawn on the display, measured in pixels
**bit_pattern** specifies the pattern of the eight successive pixels of the lines to be drawn.
There are the following predefined patterns:

- **GLCD_LINE_SOLID** solid line pattern
- **GLCD_LINE_DOT_SMALL** small dots line pattern
- **GLCD_LINE_DOT_LARGE** large dots line pattern.

**void glcd_setlinethick(unsigned char thickness)**

Sets the current line thickness.

*Parameter:*

**thickness** specifies the thickness of the lines to be drawn on the display, measured in pixels.

**unsigned char glcd_getlinethick(void)**

Returns current line thickness setting, measured in pixels.

**unsigned char glcd_getlinepattern(void)**

Returns current line bit pattern setting.

**void glcd_line(GLCDX_t x0, GLCDY_t y0, GLCDX_t x1, GLCDY_t y1)**

Draws a line with the current foreground color, thickness and bit pattern. The current pixel plot position will be updated to the line's ending point coordinates.

*Parameters:*

**x0** specifies the line's starting point horizontal coordinate
**y0** specifies the line's starting point vertical coordinate
**x1** specifies the line's ending point horizontal coordinate
**y1** specifies the line's ending point vertical coordinate.

**void glcd_lineto(GLCDX_t x, GLCDY_t y)**

Draws a line from the current pixel plot position to a new position using the current foreground color, thickness and bit pattern.
The current pixel plot position will be updated to the line's ending point coordinates.

*Parameters:*

**x** specifies the line's ending point horizontal coordinate
**y** specifies the line's ending point vertical coordinate.

**void glcd_linerel(GLCDDX_t dx, GLCDDY_t dy)**

Draws a line from the current pixel plot position to a new relative position using the current foreground color, thickness and bit pattern.
The current pixel plot position will be updated to the line's ending point coordinates.

*Parameters:*

**dx** specifies the horizontal displacement of the line's ending point relative to the current pixel plot position
**dy** specifies the vertical displacement of the line's ending point relative to the current pixel plot position.

**void glcd_rectangle(GLCDX_t left, GLCDY_t top, GLCDX_t right, GLCDY_t bottom)**

Draws a rectangle with the current foreground color, line thickness and bit pattern, using absolute coordinates.
The current pixel plot position will be updated to the left top corner of the rectangle.

*Parameters:*

**left** specifies the horizontal coordinate of the left top corner of the rectangle
**top** specifies the vertical coordinate of the left top corner of the rectangle
**right** specifies the horizontal coordinate of the right bottom corner of the rectangle
**bottom** specifies the vertical coordinate of the right bottom corner of the rectangle.

**void glcd_rectrel(GLCDX_t left, GLCDY_t top, GLCDDX_t width, GLCDDY_t height)**

Draws a rectangle with the current foreground color, line thickness and bit pattern, using relative coordinates.
The current pixel plot position will be updated to the left top corner of the rectangle.

*Parameters:*

**left** specifies the horizontal coordinate of the left top corner of the rectangle
**top** specifies the vertical coordinate of the left top corner of the rectangle
**width** specifies the horizontal size of the rectangle
**height** specifies the vertical size of the rectangle.

**void glcd_rectround(GLCDX_t left, GLCDY_t top, GLCDDX_t width, GLCDDY_t height, GLCDRAD_t radius)**

Draws a rectangle with rounded corners, using the current foreground color and line thickness.
The current pixel plot position will be updated to the left top corner of the rectangle.

*Parameters:*

**left** specifies the horizontal coordinate of the left top corner of the rectangle
**top** specifies the vertical coordinate of the left top corner of the rectangle
**width** specifies the horizontal size of the rectangle
**height** specifies the vertical size of the rectangle
**radius** specifies the radius of each corner's circle arc.

**void glcd_drawpoly(unsigned char npoints, flash GLCDPOINT_t *polypoints)**

Draws a polygon using the current foreground color, line thickness and bit pattern.
The current pixel plot position will be updated to the ending point of the last line of the polygon.

*Parameters:*

**npoints** specifies the number of points of the polygon
**polypoints** points to a an array of polygon point coordinates located in FLASH.

Example:

```
/* Include the graphic display driver functions.
   The display controller type and connections must be specified in
   the Project|Configure|C Compiler|Libraries|Graphic Display menu. */
#include <glcd.h>

/* Array located in FLASH that holds the hexagon point coordinates.
   The coordinate of the last point must match the ones of the
   first point, so that the polygon will be closed.
   So there will be 6+1=7 points for a hexagon. */
flash GLCDPOINT_t hexagon[7]=
{{40,0},{57,10},{57,30},{40,40},{23,30},{23,10},{40,0}};

void main(void)
{
GLCDINIT_t init;

/* We will not display any text, so there's need for a font */
init.font=NULL;

/* No need for reading data from external memory */
init.readxmem=NULL;

/* No need for writing data to external memory */
init.writexmem=NULL;

/* Initialize the display controller and graphics */
glcd_init(&init);

/* Draw the hexagon */
glcd_drawpoly(7,hexagon);

/* Stop here */
while (1);
}
```

**void glcd_circle(GLCDX_t x, GLCDY_t y, GLCDRAD_t radius)**

Draws a circle at specified center coordinates using the current foreground color and line thickness.

*Parameters:*

**x** specifies the horizontal coordinate of the circle's center
**y** specifies the vertical coordinate of the circle's center
**radius** specifies the circle's radius.

**void glcd_arc(GLCDX_t x, GLCDY_t y, unsigned short start_angle, unsigned short end_angle, GLCDRAD_t radius)**

Draws an arc of a circle at specified center coordinates using the current foreground color and line thickness.
The angles are measured in degrees, starting from the three o'clock position counter-clockwise.

*Parameters:*

**x** specifies the horizontal coordinate of the circle's center
**y** specifies the vertical coordinate of the circle's center
**start_angle** specifies the arc's starting angle
**end_angle** specifies the arc's ending angle
**radius** specifies the circle's radius.

**void glcd_getarccoords(GLCDARCCOORDS_t *arccoords)**

Fills a **GLCDARCCOORDS_t** type structure with information about the last call to the **glcd_arc** function.

*Parameter:*

**arccoords** points to a **GLCDARCCOORDS_t** type structure.

**void glcd_setfill(unsigned char *pattern, GLCDCOL_t color)**

Sets an user defined 8x8 pixel fill pattern from RAM, used by the **glcd_bar** and **glcd_barrel** functions, and the fill color.

*Parameters:*

**pattern** points to an 8 byte array that holds the fill pattern
**color** specifies the current color used by the filling functions.

**void glcd_setfillf(flash unsigned char *pattern, GLCDCOL_t color)**

Sets an user defined 8x8 pixel fill pattern from FLASH, used by the **glcd_bar** and **glcd_barrel** functions, and the fill color.

*Parameters:*

**pattern** points to an 8 byte array that holds the fill pattern
**color** specifies the current color used by the filling functions.

**void glcd_setfille(eeprom unsigned char *pattern, GLCDCOL_t color)**

Sets an user defined 8x8 pixel fill pattern from EEPROM, used by the **glcd_bar** and **glcd_barrel** functions, and the fill color.

*Parameters:*

**pattern** points to an 8 byte array that holds the fill pattern
**color** specifies the current color used by the filling functions.

**void glcd_setfillcolor(GLCDCOL_t color)**

Sets the fill color.

*Parameters:*

**color** specifies the current color used by the filling functions.

**GLCDCOL_t glcd_getfillcolor(void)**

Returns the current fill color used by the filling functions.

**void glcd_bar(GLCDX_t left, GLCDY_t top, GLCDX_t right, GLCDY_t bottom)**

Draws a filled-in rectangular bar, using absolute coordinates, without drawing an outline.
The bar is filled using the current fill pattern and fill color.

*Parameters:*

**left** specifies the horizontal coordinate of the left top corner of the bar
**top** specifies the vertical coordinate of the left top corner of the bar
**right** specifies the horizontal coordinate of the right bottom corner of the bar
**bottom** specifies the vertical coordinate of the right bottom corner of the bar.

**void glcd_barrel(GLCDX_t left, GLCDY_t top, GLCDDX_t width, GLCDDY_t height)**

Draws a filled-in rectangular bar, using relative coordinates, without drawing an outline.
The bar is filled using the current fill pattern and fill color.

*Parameters:*

**left** specifies the horizontal coordinate of the left top corner of the bar
**top** specifies the vertical coordinate of the left top corner of the bar
**width** specifies the horizontal size of the bar
**height** specifies the vertical size of the bar.

**void glcd_floodfill(GLCDX_t x, GLCDY_t y, GLCDCOL_t border)**

Fills a closed polygon or area with the current fill color.

*Parameters:*

**x** specifies the horizontal coordinate of a point inside the area to be filled
**y** specifies the vertical coordinate of a point inside the area to be filled
**border** specifies the color of the border of the area to be filled, where the fill process must stop.

**void glcd_fillcircle(GLCDX_t x, GLCDY_t y, GLCDRAD_t radius)**

Draws and fills a circle at specified center coordinates using the current fill color.

*Parameters:*

**x** specifies the horizontal coordinate of the circle's center
**y** specifies the vertical coordinate of the circle's center
**radius** specifies the circle's radius.

**void glcd_pieslice(GLCDX_t x, GLCDY_t y, unsigned short start_angle,
unsigned short end_angle, GLCDRAD_t radius)**

Draws a pie slice at specified center coordinates using the current foreground color and line thickness. After that the pie slice is filled with the current fill color.
The angles are measured starting from from 3 o'clock counter-clockwise.

*Parameters:*

**x** specifies the horizontal coordinate of the circle's center
**y** specifies the vertical coordinate of the circle's center
**start_angle** specifies the arc's starting angle
**end_angle** specifies the arc's ending angle
**radius** specifies the circle's radius.

## 5.12.1 Graphic LCD Functions Specific to the ILI9325 Controller

The ILI9325 library functions supplied with the CodeVisionAVR Advanced license, can operate the controller in 16 bit interface mode with 256 or 64k colors.
In order to take full advantage of the ILI9325 controller's features the following specific functions, declared in the **glcd_ili9325.h** header file, were implemented:

**void ili9325_wrcmd(unsigned char cmd)**

Writes a command to the ILI9325 controller, used to access a specific command register.

*Parameter:*

**cmd** command register index to be sent to the controller.

This index may take one of the values defined in the following macros from the **glcd_ili9325.h** header file:

```
/* ILI9325 command register definitions */
#define ILI9325_CMD_OSC 0x00 /* Oscillator register */
#define ILI9325_CMD_DRIVER_OUT 0x01 /* Driver output control */
#define ILI9325_CMD_DRIVING_WAVEFORM 0x02 /* LCD driving waveform control
*/
#define ILI9325_CMD_ENTRY_MODE 0x03 /* Entry mode control */
#define ILI9325_RESIZE_CONTROL 0x04 /* Resizing control register */
#define ILI9325_CMD_DISPLAY_CONTROL1 0x07 /* Display control register 1 */
#define ILI9325_CMD_DISPLAY_CONTROL2 0x08 /* Display control register 2 */
#define ILI9325_CMD_DISPLAY_CONTROL3 0x09 /* Display control register 3 */
#define ILI9325_CMD_DISPLAY_CONTROL4 0x0a /* Display control register 4 */
#define ILI9325_CMD_RGB_IF_CONTROL1 0x0c /* RGB display interface control
register 1 */
#define ILI9325_FRAME_MARKER_POS 0x0d /* Frame marker position */
#define ILI9325_CMD_RGB_IF_CONTROL2 0x0f /* RGB display interface control
register 2 */
#define ILI9325_CMD_POWER_CONTROL1 0x10 /* Power control 1 register */
#define ILI9325_CMD_POWER_CONTROL2 0x11 /* Power control 2 register */
#define ILI9325_CMD_POWER_CONTROL3 0x12 /* Power control 3 register */
#define ILI9325_CMD_POWER_CONTROL4 0x13 /* Power control 4 register */
#define ILI9325_CMD_GDDRAMX 0x20 /* Set GRAM X address counter register */
#define ILI9325_CMD_GDDRAMY 0x21 /* Set GRAM Y address counter register */
#define ILI9325_CMD_GDDRAM_DATA 0x22 /* GRAM read/write data register */
#define ILI9325_CMD_POWER_CONTROL7 0x29 /* Power control 7 register */
#define ILI9325_CMD_FRAME_RATE 0x2b /* Frame rate frequency control
register */

#define ILI9325_CMD_GAMMA_CONTROL1 0x30 /* Gamma control 1 */
#define ILI9325_CMD_GAMMA_CONTROL2 0x31 /* Gamma control 2 */
#define ILI9325_CMD_GAMMA_CONTROL3 0x32 /* Gamma control 3 */
#define ILI9325_CMD_GAMMA_CONTROL4 0x35 /* Gamma control 4 */
#define ILI9325_CMD_GAMMA_CONTROL5 0x36 /* Gamma control 5 */
#define ILI9325_CMD_GAMMA_CONTROL6 0x37 /* Gamma control 6 */
#define ILI9325_CMD_GAMMA_CONTROL7 0x38 /* Gamma control 7 */
define ILI9325_CMD_GAMMA_CONTROL8 0x39 /* Gamma control 8 */
#define ILI9325_CMD_GAMMA_CONTROL9 0x3c /* Gamma control 9 */
#define ILI9325_CMD_GAMMA_CONTROL10 0x3d /* Gamma control 10 */
```

```
#define ILI9325_CMD_HORIZ_RAM_ADDR_START 0x50 /* Address of horizontal
start window positions */
#define ILI9325_CMD_HORIZ_RAM_ADDR_END 0x51 /* Address of horizontal end
window positions */
#define ILI9325_CMD_VERT_RAM_ADDR_START 0x52 /* Address of vertical start
window positions */
#define ILI9325_CMD_VERT_RAM_ADDR_END 0x53 /* Address of vertical end
window positions */
#define ILI9325_CMD_DRIVER_OUT_CONTROL2 0x60 /* Driver output control */
#define ILI9325_CMD_BASE_IMG_DISPLAY_CONTROL 0x61 /* Base image display
control */
#define ILI9325_CMD_VERT_SCROLL_CONTROL 0x6a /* Vertical scroll control */
#define ILI9325_CMD_PARTIAL_IMG1_DISPLAY_POS 0x80 /* Partial image 1
display position */
#define ILI9325_CMD_PARTIAL_IMG1_START_LINE 0x81 /* Partial image 1 display
start line */
#define ILI9325_CMD_PARTIAL_IMG1_END_LINE 0x82 /* Partial image 1 display
end line */
#define ILI9325_CMD_PARTIAL_IMG2_DISPLAY_POS 0x83 /* Partial image 2
display position */
#define ILI9325_CMD_PARTIAL_IMG2_START_LINE 0x84 /* Partial image 2 display
start line */
#define ILI9325_CMD_PARTIAL_IMG2_END_LINE 0x85 /* Partial image 2 display
end line */
#define ILI9325_CMD_PANEL_IF_CONTROL1 0x90 /* Panel interface control 1 */
#define ILI9325_CMD_PANEL_IF_CONTROL2 0x92 /* Panel interface control 2 */
#define ILI9325_CMD_PANEL_IF_CONTROL4 0x95 /* Panel interface control 4 */
#define ILI9325_CMD_OTP_VCM_PROG_CONTROL 0xA1 /* OTP VCM programming
control */
#define ILI9325_CMD_OTP_VCM_STATUS_ENABLE 0xA2 /* OTP VCM status and enable
*/
#define ILI9325_CMD_OTP_PGM_ID_KEY 0xA5 /* OTP programming ID key */
```

A detailed description of the above mentioned command registers can be found in the ILI9325 datasheet.

**void ili9325_wrreg(unsigned char index, unsigned short data)**

Writes data to a command register of the ILI9325 controller.

*Parameters:*

**index** command register index
**data** to be written.

**unsigned short ili9325_rdreg(unsigned char index)**

Reads the contents of a command register of the ILI9325 controller.

*Parameters:*

**index** command register index.

**void ili9325_wrdata(unsigned short data)**

Writes data to the ILI9325 controller's Graphic Display RAM.

*Parameters:*

**data** to be written.

**unsigned short ili9325_rddata(void)**

Reads data from the ILI9325 controller's Graphic Display RAM.

The **glcd_ili9325.h** header file also contains the definition of the **GLCDINIT_t** type specific for the ILI9325 controller, used as parameter for the **glcd_init** function:

```
typedef struct
        {
        flash unsigned char *font; /* default font after initialization */
        /* pointer to the function used for reading a byte from external
           memory */
        unsigned char (*readxmem) (GLCDMEMADDR_t addr);
        /* pointer to the function used for writing a byte to external
           memory */
        void (*writexmem) (GLCDMEMADDR_t addr, unsigned char data);
        unsigned char reverse_x:1; /* reverse display horizontally */
        unsigned char reverse_y:1; /* reverse display vertically */
        unsigned char cl_bits_order:1; /* selects the color bits writing
                                          order to the display RAM
                                          =0 -> RGB, =1 ->BGR */
        /* power control registers bits */
        unsigned char stepup_factor:3; /* step-up factor of the step-up
                                          circuit, see BT0..BT2 bits in
                                          the datasheet */
        unsigned char stepup_freq1:3;  /* controls the frequency for the
                                          step-up circuit 1 */
        unsigned char stepup_freq2:3;  /* controls the frequency for the
                                          step-up circuit 2 */
        unsigned char crt_source:3; /* adjusts the amount of current
                                       from the constant current source
                                       in the internal op. amplifier
                                       circuit (AP0..AP2 bits) */
        unsigned char vreg1out:4;  /* adjusts the VREG1OUT voltage */
        unsigned char vcom:5;  /* adjusts the amplitude of the Vcom
                                  alternating drive voltage based on
                                  VREG1OUT voltage */
        unsigned char vcomh:6; /* adjusts the amplitude of the VcomH
                                  voltage based on VREG1OUT voltage
                                  VcomH=VREG1OUT*(vcomh*0.005+0.685) [V] */
        unsigned char frame_freq:4; /* LCD frame frequency */
```

```
        /* positive gamma control registers bits */
        unsigned char kp00:3; /* KP00..KP02 positive gamma micro adj. */
        unsigned char kp10:3; /* KP10..KP12 positive gamma micro adj. */
        unsigned char kp20:3; /* KP20..KP22 positive gamma micro adj. */
        unsigned char kp30:3; /* KP30..KP32 positive gamma micro adj. */
        unsigned char kp40:3; /* KP40..KP42 positive gamma micro adj. */
        unsigned char kp50:3; /* KP50..KP52 positive gamma micro adj. */
        unsigned char rp00:3; /* RP00..RP02 positive gamma gradient
                                 adj. */
        unsigned char rp10:3; /* RP10..RP12 positive gamma gradient
                                 adj. */
        unsigned char vrp00:4; /* VRP00..VRP03 positive gamma amplification
                                 adj. */
        unsigned char vrp10:5; /* VRP10..VRP14 positive gamma amplification
                                 adj. */
        /* negative gamma control registers bits */
        unsigned char kn00:3; /* KN00..KN02 negative gamma micro adj. */
        unsigned char kn10:3; /* KN10..KN12 negative gamma micro adj. */
        unsigned char kn20:3; /* KN20..KN22 positive gamma micro adj. */
        unsigned char kn30:3; /* KN30..KN32 positive gamma micro adj. */
        unsigned char kn40:3; /* KN40..KN42 negative gamma micro adj. */
        unsigned char kn50:3; /* KN50..KN52 negative gamma micro adj. */
        unsigned char rn00:3; /* RN00..RN02 negative gamma gradient
                                 adj. */
        unsigned char rn10:3; /* RN10..RN12 negative gamma gradient
                                 adj. */
        unsigned char vrn00:4; /* VRN00..VRN03 negative gamma amplification
                                 adj. */
        unsigned char vrn10:5; /* VRN10..VRN14 negative gamma
                                 amplification adj. */
        } GLCDINIT_t;
```

The following macros are defined for initializing the members of the **GLCDINIT_t** structure:

```
/* Initialization values for reverse_x */
#define ILI9325_REVX_NORM 0 /* No horizontal reverse */
#define ILI9325_REVX_REV 1 /* Horizontal reverse */

/* Initialization values for reverse_y */
#define ILI9325_REVY_NORM 0 /* No vertical reverse */
#define ILI9325_REVY_REV 1 /* Vertical reverse */

/* Initialization values for cl_bits_order */
#define ILI9325_CL_BITS_RGB 0 /* Write color bits to display RAM
                                 in RGB order */
#define ILI9325_CL_BITS_BGR 1 /* Write color bits to display RAM
                                 in BGR order */

/* Initialization values for the VREG1OUT voltage */
#define ILI9325_VREG1OUT_4V000 8 /* 4.000 V */
#define ILI9325_VREG1OUT_4V125 9 /* 4.125 V */
#define ILI9325_VREG1OUT_4V250 10 /* 4.250 V */
#define ILI9325_VREG1OUT_4V375 11 /* 4.375 V */
#define ILI9325_VREG1OUT_4V500 12 /* 4.500 V */
#define ILI9325_VREG1OUT_4V625 13 /* 4.625 V */
#define ILI9325_VREG1OUT_4V750 14 /* 4.750 V */
#define ILI9325_VREG1OUT_4V875 15 /* 4.875 V */
#define ILI9325_VREG1OUT_5V000 1 /* 5.000 V */
#define ILI9325_VREG1OUT_5V125 2 /* 5.125 V */
```

```
#define ILI9325_VREG1OUT_5V250 3 /* 5.250 V */
#define ILI9325_VREG1OUT_5V500 4 /* 5.500 V */
#define ILI9325_VREG1OUT_5V750 5 /* 5.750 V */
#define ILI9325_VREG1OUT_6V000 6 /* 6.000 V */

/* Initialization values for the Vcom voltage */
#define ILI9325_VCOM_0_70 0 /* Vcom=VREG1OUT*0.70 */
#define ILI9325_VCOM_0_72 1 /* Vcom=VREG1OUT*0.72 */
#define ILI9325_VCOM_0_74 2 /* Vcom=VREG1OUT*0.74 */
#define ILI9325_VCOM_0_76 3 /* Vcom=VREG1OUT*0.76 */
#define ILI9325_VCOM_0_78 4 /* Vcom=VREG1OUT*0.78 */
#define ILI9325_VCOM_0_80 5 /* Vcom=VREG1OUT*0.80 */
#define ILI9325_VCOM_0_82 6 /* Vcom=VREG1OUT*0.82 */
#define ILI9325_VCOM_0_84 7 /* Vcom=VREG1OUT*0.84 */
#define ILI9325_VCOM_0_86 8 /* Vcom=VREG1OUT*0.86 */
#define ILI9325_VCOM_0_88 9 /* Vcom=VREG1OUT*0.88 */
#define ILI9325_VCOM_0_90 10 /* Vcom=VREG1OUT*0.90 */
#define ILI9325_VCOM_0_92 11 /* Vcom=VREG1OUT*0.92 */
#define ILI9325_VCOM_0_94 12 /* Vcom=VREG1OUT*0.94 */
#define ILI9325_VCOM_0_96 13 /* Vcom=VREG1OUT*0.96 */
#define ILI9325_VCOM_0_98 14 /* Vcom=VREG1OUT*0.98 */
#define ILI9325_VCOM_1_00 15 /* Vcom=VREG1OUT*1.00 */
#define ILI9325_VCOM_1_02 0x14 /* Vcom=VREG1OUT*1.02 */
#define ILI9325_VCOM_1_04 0x15 /* Vcom=VREG1OUT*1.04 */
#define ILI9325_VCOM_1_06 0x16 /* Vcom=VREG1OUT*1.06 */
#define ILI9325_VCOM_1_08 0x17 /* Vcom=VREG1OUT*1.08 */
#define ILI9325_VCOM_1_10 0x18 /* Vcom=VREG1OUT*1.10 */
#define ILI9325_VCOM_1_12 0x19 /* Vcom=VREG1OUT*1.12 */
#define ILI9325_VCOM_1_14 0x1A /* Vcom=VREG1OUT*1.14 */
#define ILI9325_VCOM_1_16 0x1B /* Vcom=VREG1OUT*1.16 */
#define ILI9325_VCOM_1_18 0x1C /* Vcom=VREG1OUT*1.18 */
#define ILI9325_VCOM_1_20 0x1D /* Vcom=VREG1OUT*1.20 */
#define ILI9325_VCOM_1_22 0x1E /* Vcom=VREG1OUT*1.22 */
#define ILI9325_VCOM_1_24 0x1F /* Vcom=VREG1OUT*1.24 */

/* Initialization value for VcomH
   VcomH=VREG1OUT*(VREG1OUT_MULT1000/1000)
   VREG1OUT_MULT1000=685..1000 */
#define ILI9325_VCOMH(VREG1OUT_MULT1000) ((VREG1OUT_MULT1000-685)/5)

/* Initialization values for stepup_freq1 */
#define ILI9325_STEPUP1_FOSC 0 /* FDCDC1=Fosc */
#define ILI9325_STEPUP1_FOSC2 1 /* FDCDC1=Fosc/2 */
#define ILI9325_STEPUP1_FOSC4 2 /* FDCDC1=Fosc/4 */
#define ILI9325_STEPUP1_FOSC8 3 /* FDCDC1=Fosc/8 */
#define ILI9325_STEPUP1_FOSC16 4 /* FDCDC1=Fosc/16 */
#define ILI9325_STEPUP1_FOSC32 5 /* FDCDC1=Fosc/32 */
#define ILI9325_STEPUP1_FOSC64 6 /* FDCDC1=Fosc/64 */
#define ILI9325_STEPUP1_HALT 7 /* Halt step-up circuit 1 */

/* Initialization values for stepup_freq2 */
#define ILI9325_STEPUP2_FOSC4 0 /* FDCDC2=Fosc/4 */
#define ILI9325_STEPUP2_FOSC8 1 /* FDCDC2=Fosc/8 */
#define ILI9325_STEPUP2_FOSC16 2 /* FDCDC2=Fosc/16 */
#define ILI9325_STEPUP2_FOSC32 3 /* FDCDC2=Fosc/32 */
#define ILI9325_STEPUP2_FOSC64 4 /* FDCDC2=Fosc/64 */
#define ILI9325_STEPUP2_FOSC128 5 /* FDCDC2=Fosc/128 */
#define ILI9325_STEPUP2_FOSC256 6 /* FDCDC2=Fosc/256 */
#define ILI9325_STEPUP2_HALT 7 /* Halt step-up circuit 2 */
```

```
/* Initialization values for frame_freq */
#define ILI9325_FRAME40 0 /* 40Hz */
#define ILI9325_FRAME43 1 /* 43Hz */
#define ILI9325_FRAME45 2 /* 45Hz */
#define ILI9325_FRAME48 3 /* 48Hz */
#define ILI9325_FRAME51 4 /* 51Hz */
#define ILI9325_FRAME55 5 /* 55Hz */
#define ILI9325_FRAME59 6 /* 59Hz */
#define ILI9325_FRAME64 7 /* 64Hz */
#define ILI9325_FRAME70 8 /* 70Hz */
#define ILI9325_FRAME77 9 /* 77Hz */
#define ILI9325_FRAME85 10 /* 85Hz */
#define ILI9325_FRAME96 11 /* 96Hz */
#define ILI9325_FRAME110 12 /* 110Hz */
#define ILI9325_FRAME128 13 /* 128Hz */

/* Default value for reverse_x */
#define ILI9325_DEFAULT_REVX ILI9325_REVX_NORM /* No horizontal reverse */
/* Default value for reverse_y */
#define ILI9325_DEFAULT_REVY ILI9325_REVY_NORM /* No vertical reverse */

/* Default value for cl_bits_order
  (color bits writing order to display RAM),
  write in RGB order */
#define ILI9325_DEFAULT_CL_BITS ILI9325_CL_BITS_RGB

/* Power control 1 BT0..BT2 step-up factor of the step-up circuit
   DDVDH=Vci1*2, VCL=-Vci1, VGH=Vci1*6, VGL=-Vci1*3 */
#define ILI9325_DEFAULT_STEPUP_FACTOR 2
/* Power control 1 AP0..AP2 adjusts the amount of current
   from the constant current source in the
   internal op. amplifier circuit */
#define ILI9325_DEFAULT_CRT_SOURCE 2
/* Power control 2 DC00..DC02 step-up circuit 1 frequency */
#define ILI9325_DEFAULT_STEPUP1_FREQ ILI9325_STEPUP1_FOSC4
/* Power control 2 DC10..DC12 step-up circuit 2 frequency */
#define ILI9325_DEFAULT_STEPUP2_FREQ ILI9325_STEPUP2_FOSC128
/* Default value for VREG1OUT voltage */
#define ILI9325_DEFAULT_VREG1OUT ILI9325_VREG1OUT_4V000
/* Default value for Vcom alternating drive voltage */
#define ILI9325_DEFAULT_VCOM ILI9325_VCOM_0_94
/* Default value for VcomH voltage
   VcomH=VREG1OUT*0.835 */
#define ILI9325_DEFAULT_VCOMH ILI9325_VCOMH(835)
/* Default value for LCD frame frequency */
#define ILI9325_DEFAULT_FRAME_FREQ ILI9325_FRAME96

/* Default initialization values for the gamma control register bits */
/* KP00..KP02 positive gamma micro adj. */
#define ILI9325_DEFAULT_KP00 7
/* KP10..KP12 positive gamma micro adj. */
#define ILI9325_DEFAULT_KP10 7
/* KP20..KP22 positive gamma micro adj. */
#define ILI9325_DEFAULT_KP20 4
/* KP30..KP32 positive gamma micro adj. */
#define ILI9325_DEFAULT_KP30 2
/* KP40..KP42 positive gamma micro adj. */
#define ILI9325_DEFAULT_KP40 4
```

```
/* KP50..KP52 positive gamma micro adj. */
#define ILI9325_DEFAULT_KP50 2
/* RP00..RP02 positive gamma gradient adj. */
#define ILI9325_DEFAULT_RP00 2
/* RP10..RP12 positive gamma gradient adj. */
#define ILI9325_DEFAULT_RP10 5
/* VRP00..VRP03 positive gamma amplification adj. */
#define ILI9325_DEFAULT_VRP00 2
/* VRP10..VRP14 positive gamma amplification adj. */
#define ILI9325_DEFAULT_VRP10 3
/* KN00..KN02 negative gamma micro adj. */
#define ILI9325_DEFAULT_KN00 7
/* KN10..KN12 negative gamma micro adj. */
#define ILI9325_DEFAULT_KN10 5
/* KN20..KN22 positive gamma micro adj. */
#define ILI9325_DEFAULT_KN20 4
/* KN30..KN32 positive gamma micro adj. */
#define ILI9325_DEFAULT_KN30 2
/* KN40..KN42 negative gamma micro adj. */
#define ILI9325_DEFAULT_KN40 4
/* KN50..KN52 negative gamma micro adj. */
#define ILI9325_DEFAULT_KN50 2
/* RN00..RN02 negative gamma gradient adj. */
#define ILI9325_DEFAULT_RN00 2
/* RN10..RN12 negative gamma gradient adj. */
#define ILI9325_DEFAULT_RN10 5
/* VRN00..VRN03 negative gamma amplification adj. */
#define ILI9325_DEFAULT_VRN00 2
/* VRN10..VRN14 negative gamma amplification adj. */
#define ILI9325_DEFAULT_VRN10 3
```

The following colors are redefined in the **glcd_ili9325.h** header file:

GLCD_CL_BLACK
GLCD_CL_WHITE
GLCD_CL_GRAY
GLCD_CL_LIGHT_GRAY
GLCD_CL_GREEN
GLCD_CL_LIME
GLCD_CL_BLUE
GLCD_CL_RED
GLCD_CL_AQUA
GLCD_CL_YELLOW
GLCD_CL_MAGENTA
GLCD_CL_CYAN
GLCD_CL_DARK_CYAN
GLCD_CL_ORANGE
GLCD_CL_PINK
GLCD_CL_BROWN
GLCD_CL_VIOLET
GLCD_CL_SILVER
GLCD_CL_GOLD
GLCD_CL_NAVY
GLCD_CL_MAROON
GLCD_CL_PURPLE
GLCD_CL_OLIVE

The ILI9325 library functions can operate the display in 256 or 64k color modes.
For 256 color mode the following color bit allocation in a data byte is used:

- Bits 0..2 - Blue color bits 0..2
- Bits 3..5 - Green color bits 0..2
- Bits 6..7 - Red color bits 0..1.

For 64k color mode the following color bit allocation in a 16bit data word is used:

- Bits 0..4 - Blue color bits 0..4
- Bits 5..10 - Green color bits 0..5
- Bits 11..15 - Red color bits 0..4.

**Notes:**

- In order to reduce image storage size and imrove speed, it is recommended to use the 256 color mode, if possible.
- The **glcd_ili9325.h** header file is automatically #included when the main **glcd.h** header file is #included. Therefore there is no need to explicitly #include **glcd_ili9325.h**.
- The **..\EXAMPLES\Graphic LCD\ILI9325** directory contains fully functional code samples that may be used as references for ILI9325 initialization and usage.

## 5.12.2 Graphic LCD Functions Specific to the PCD8544 Controller

In order to take full advantage of the PCD8544 controller's features the following specific functions, declared in the **glcd_pcd8544.h** header file, were implemented:

**void pcd8544_wrcmd(unsigned char cmd)**

Writes a command to the PCD8544 controller.

*Parameter:*

**cmd** command to be sent to the controller.

This command may take one of the values defined in the following macros from the **glcd_pcd8544.h** header file:

```
#define PCD8544_FUNCTION_SET 0x20 /* Function set, horizontal addressing */
/* Puts controller in power-down mode,
   must be combined with PCD8544_FUNCTION_SET */
#define PCD8544_POWER_DOWN 0x04
/* Use extended instruction set,
   must be combined with PCD8544_FUNCTION_SET */
#define PCD8544_EXT_INST 0x01

#define PCD8544_DISPLAY_BLANK 0x08 /* Sets display blank */
#define PCD8544_DISPLAY_NORMAL 0x0C /* Sets display normal mode */
#define PCD8544_DISPLAY_ALL_ON 0x09 /* Sets all display segments on */
#define PCD8544_DISPLAY_INVERSE 0x0D /* Sets inverse video mode */
#define PCD8544_SETX 0X80  /* Sets X address of display RAM */
#define PCD8544_SETY 0X40  /* Sets Y address of display RAM */

/* Extended instruction set enabled by
   PCD8544_FUNCTION_SET+PCD8544_EXT_INST */
#define PCD8544_TEMP_CTRL 0x04 /* Sets temperature coefficient */
#define PCD8544_BIAS_SYSTEM 0x10 /* Sets bias system */
#define PCD8544_VLCD 0x80 /* Sets VLCD value */
```

A detailed description of the above mentioned commands can be found in the PCD8544 datasheet.

**void pcd8544_setvlcd(unsigned char vlcd)**

Controls the LCD contrast.

*Parameter:*

**vlcd** value for the VLCD voltage, allowed range is 0..127.

The **glcd_pcd8544.h** header file also contains the definition of the **GLCDINIT_t** type specific for the PCD8544 controller, used as parameter for the **glcd_init** function:

```
typedef struct
        {
        flash unsigned char *font; /* default font after initialization */

        /* pointer to the function used for reading a byte from external
           memory */
        unsigned char (*readxmem) (GLCDMEMADDR_t addr);

        /* pointer to the function used for writing a byte to external
           memory */
        void (*writexmem) (GLCDMEMADDR_t addr, unsigned char data);

        unsigned char temp_coef:2; /* temperature coefficient
                                      bits [0..3] */
        unsigned char bias:3; /* bias system bits [0..7] */
        unsigned char vlcd:7; /* VLCD set bits [0..127] */
        } GLCDINIT_t;
```

The detailed description of the above mentioned initialization parameters can be found in the PCD8544 datasheet.

**Notes:**

- The **glcd_pcd8544.h** header file is automatically #included when the main **glcd.h** header file is #included. Therefore there is no need to explicitly #include **glcd_pcd8544.h**.
- The **..\EXAMPLES\Graphic LCD\PCD8544** directory contains fully functional code samples that may be used as references for PCD8544 initialization and usage.

## 5.12.3 Graphic LCD Functions Specific to the S1D13700 Controller

In order to take full advantage of the S1D13700 controller's features the following specific functions, declared in the **glcd_s1d13700.h** header file, were implemented:

**void s1d_wrcmd(unsigned char cmd)**

Writes a command to the S1D13700 controller.

*Parameter:*

**cmd** command to be sent to the controller.

This command may take one of the values defined in the following macros from the **glcd_s1d13700.h** header file:

```
#define S1D13700_SYSTEM_SET 0X40  /* Initialize device and display */
#define S1D13700_MWRITE 0X42  /* Write to display memory */
#define S1D13700_MREAD 0x43  /* Read from display memory */
#define S1D13700_SCROLL 0X44  /* Set display start address and display
regions */
#define S1D13700_CSRW 0X46 /* Set cursor address */
#define S1D13700_CSRR 0X47 /* Read cursor address */
#define S1D13700_CSRDIR_RIGHT 0X4C  /* Set direction of cursor movement
                                       to right */
#define S1D13700_POWER_SAVE 0X53  /* Enter standby mode */
#define S1D13700_DISP_ON_OFF 0X58  /* Enable/disable display and
                                      display flashing */
#define S1D13700_HDOT_SCR 0X5A  /* Set horizontal scroll position */
#define S1D13700_OVLAY 0X5B  /* Set display overlay format */
#define S1D13700_CGRAM_ADDR 0X5C  /* Set start address of character
                                     generator RAM */
#define S1D13700_CSRFORM 0X5D  /* Set cursor type */
#define S1D13700_GRAYSCALE 0x60 /* Set grayscale depth */
```

A detailed description of the above mentioned commands can be found in the S1D13700 datasheet.

**void s1d_wrdata(unsigned char data)**

Writes a data byte to the S1D13700 controller.

*Parameter:*

**data** byte to be sent to the controller.

**unsigned char s1d_rddata(void)**

Reads a data byte from the S1D13700 controller.

**Notes:**

- The **glcd_s1d13700.h** header file is automatically #included when the main **glcd.h** header file is #included. Therefore there is no need to explicitly #include **glcd_s1d13700.h**.
- The **..\EXAMPLES\Graphic LCD\S1D13700** directory contains fully functional code samples that may be used as references for S1D13700 initialization and usage.

## 5.12.4 Graphic LCD Functions Specific to the SED1335 Controller

In order to take full advantage of the SED1335 controller's features the following specific functions, declared in the **glcd_sed1335.h** header file, were implemented:

**void sed1335_wrcmd(unsigned char cmd)**

Writes a command to the SED1335 controller.

*Parameter:*

**cmd** command to be sent to the controller.

This command may take one of the values defined in the following macros from the **glcd_sed1335.h** header file:

```
#define SED1335_SYSTEM_SET 0X40  /* Initialize device and display */
#define SED1335_MWRITE 0X42  /* Write to display memory */
#define SED1335_MREAD 0x43  /* Read from display memory */
#define SED1335_SCROLL 0X44  /* Set display start address and display
                                regions */
#define SED1335_CSRW 0X46 /* Set cursor address */
#define SED1335_CSRR 0X47 /* Read cursor address */
#define SED1335_CSRDIR_RIGHT 0X4C  /* Set direction of cursor movement
                                      to right */
#define SED1335_SLEEP_IN 0X53  /* Enter standby mode */
#define SED1335_DISP_ON_OFF 0X58  /* Enable/disable display and
                                     display flashing */
#define SED1335_HDOT_SCR 0X5A  /* Set horizontal scroll position */
#define SED1335_OVLAY 0X5B  /* Set display overlay format */
#define SED1335_CGRAM_ADDR 0X5C  /* Set start address of character
                                    generator RAM */
#define SED1335_CSRFORM 0X5D  /* Set cursor type */
```

A detailed description of the above mentioned commands can be found in the SED1335 datasheet.

**void sed1335_wrdata(unsigned char data)**

Writes a data byte to the SED1335 controller.

*Parameter:*

**data** byte to be sent to the controller.

**unsigned char sed1335_rddata(void)**

Reads a data byte from the SED1335 controller.

**void sed1335_fastmode(unsigned char on)**

Specifies if the BUSY flag should be tested on data read/write in order to reduce display flicker.

*Parameter:*

**on** if set to a non-zero value specifies that the BUSY flag will not be tested in order to increase display speed, however flicker will appear.

**Notes:**
- When the **glcd_init** function is called, it enables BUSY flag testing, so that the display will not flicker.
- The **glcd_sed1335.h** header file is automatically #included when the main **glcd.h** header file is #included. Therefore there is no need to explicitly #include **glcd_sed1335.h**.
- The **..\EXAMPLES\Graphic LCD\SED1335** directory contains fully functional code samples that may be used as references for SED1335 initialization and usage.

## 5.12.5 Graphic LCD Functions Specific to the SED1530 Controller

In order to take full advantage of the SED1530 controller's features the following specific functions, declared in the **glcd_sed1530.h** header file, were implemented:

**void sed1530_wrcmd(unsigned char cmd)**

Writes a command to the SED1530 controller.

*Parameter:*

**cmd** command to be sent to the controller.

This command may take one of the values defined in the following macros from the **glcd_sed1530.h** header file:

```
#define SED1530_CMD_START_LINE 0x40 /* set display start line */
#define SED1530_CMD_SET_PAGE 0xB0 /* set display page address */
#define SED1530_CMD_SET_ADDR_LOW 0x00 /* set column address bits 0..3 */
#define SED1530_CMD_SET_ADDR_HIGH 0x10 /* set column address bits 4..7 */
#define SED1530_CMD_ADC_SELECT_NORM 0xA0 /* set relationship between RAM
                                            column address and display
                                            driver: normal */
#define SED1530_CMD_ADC_SELECT_REV 0xA1  /* set relationship between RAM
                                            column address and display
                                            driver: reversed */
#define SED1530_CMD_DISP_NORMAL 0xA6 /* set normal display mode */
#define SED1530_CMD_DISP_REVERSE 0xA7 /* set reversed display mode */
#define SED1530_CMD_PIXELS_NORMAL 0xA4 /* display the graphic RAM
                                          contents */
#define SED1530_CMD_ALL_PIXELS_ON 0xA5 /* all display pixels are on */
#define SED1530_CMD_DISP_OFF 0xAE /* display off */
#define SED1530_CMD_DISP_ON 0xAF /* display on */
#define SED1530_CMD_LCD_BIAS_16 0xA2 /* sets voltage ratio for LCD bias
                                        to 1/6 */
#define SED1530_CMD_LCD_BIAS_15 0xA3 /* sets voltage ratio for LCD bias
                                        to 1/5 */
#define SED1530_CMD_COM0_63 0xC0 /* sets the COM output scan
                                    direction 0->63 */
#define SED1530_CMD_COM63_0 0xC8 /* sets the COM output scan
                                    direction 63->0 */
#define SED1530_CMD_POWER_CTRL 0x28 /* turns on/off the
                                       voltage follower (| bit 0),
                                       voltage regulator (| bit 1),
                                       voltage booster (| bit 2) */
#define SED1530_VOLT_FOLLOWER_ON (1<<0)  /* enable voltage follower */
#define SED1530_VOLT_REGULATOR_ON (1<<1) /* enable voltage regulator */
#define SED1530_VOLT_BOOSTER_ON (1<<2)   /* enable voltage booster */

#define SED1530_CMD_ELECTRONIC_VOLUME 0x80 /* sets the electronic volume
                                              register in order to control
                                              the V5 LCD drive voltage */
#define SED1530_CMD_RESET 0xE2 /* resets the controller */
```

A detailed description of the above mentioned commands can be found in the SED1530 datasheet.

**void sed1530_setcontrast(unsigned char contrast)**

Controls the LCD contrast.

*Parameter:*

**contrast** sets the value of the V5 LCD drive voltage, allowed range is 0..31.

The **glcd_sed1530.h** header file also contains the definition of the **GLCDINIT_t** type specific for the SED1530 controller, used as parameter for the **glcd_init** function:

```
typedef struct
        {
        flash unsigned char *font; /* default font after initialization */
        /* pointer to the function used for reading a byte
           from external memory */
        unsigned char (*readxmem) (GLCDMEMADDR_t addr);
        /* pointer to the function used for writing a byte
           to external memory */
        void (*writexmem) (GLCDMEMADDR_t addr, unsigned char data);
        unsigned char lcd_bias:1; /* =0 1/6 LCD bias, =1 1/5 LCD bias */
        unsigned char reverse_x:1; /* reverse display horizontally (ADC) */
        unsigned char rev132_x0:1; /* set to 1 for displays that use
                                      reversed RAM column address
                                      (reverse_x=1) driver and the pixel
                                      with x=0 is connected to column
                                      driver #132 */
        unsigned char reverse_y:1; /* reverse display vertically (COM) */
        unsigned char lcd_contrast:4; /* V5 LCD drive voltage: [0..31] */
        } GLCDINIT_t;
```

The following macros are defined for initializing the members of the **GLCDINIT_t** structure:

```
/* values used for lcd_bias initialization */
#define SED1530_LCD_BIAS_16 0 /* sets LCD bias drive ratio 1/6 */
#define SED1530_LCD_BIAS_15 1 /* sets LCD bias drive ratio 1/5 */

/* values used for reverse_x initialization */
#define SED1530_REVX_NORM 0 /* set relationship between RAM column address
                               and display driver: normal (ADC=0) */
#define SED1530_REVX_REV 1  /* set relationship between RAM column address
                               and display driver: reversed (ADC=1) */

/* values used for rev132_x0 initilization,
   effective only when reverse_x=1 (SED1530_REVX_REV) */
#define SED1530_REV132_X0NC 0  /* pixel with x=0 is not connected to
                                  column driver #132 when ADC=1 */
#define SED1530_REV132_X0CON 1 /* pixel with x=0 is connected to
                                  column driver #132 when ADC=1 */

/* values used for reverse_y initialization */
#define SED1530_REVY_NORM 0 /* sets the vertical COM output scan
                               direction 0->63 */
#define SED1530_REVY_REV 1  /* sets the vertical COM output scan
                               direction 63->0 */
```

```
/* default initialization values */
/* default value for LCD bias */
#define SED1530_DEFAULT_LCD_BIAS SED1530_LCD_BIAS_16
/* default value for reverse_x */
#define SED1530_DEFAULT_REVX SED1530_REVX_NORM
/* default value for rev132_x0,
   effective only when reverse_x=1 (SED1530_REVX_REV) */
#define SED1530_DEFAULT_REV132_X0 SED1530_REV132_X0NC
/* default value for reverse_y */
#define SED1530_DEFAULT_REVY SED1530_REVY_NORM
/* default contrast */
#define SED1530_DEFAULT_CONTRAST 7
```

The detailed description of the above mentioned initialization parameters can be found in the SED1530 datasheet.

**Notes:**

- The **glcd_sed1530.h** header file is automatically #included when the main **glcd.h** header file is #included. Therefore there is no need to explicitly #include **glcd_sed1530.h**.
- The **..\EXAMPLES\Graphic LCD\SED1530** directory contains fully functional code samples that may be used as references for SED1530 initialization and usage.

## 5.12.6 Graphic LCD Functions Specific to the SPLC501C Controller

In order to take full advantage of the SPLC501C controller's features the following specific functions, declared in the **glcd_splc501.h** header file, were implemented:

**void splc501_wrcmd(unsigned char cmd)**

Writes a command to the SPLC501C controller.

*Parameter:*

**cmd** command to be sent to the controller.

This command may take one of the values defined in the following macros from the **glcd_splc501.h** header file:

```
#define SPLC501_CMD_START_LINE 0x40 /* set display start line */
#define SPLC501_CMD_SET_PAGE 0xB0 /* set display page address */
#define SPLC501_CMD_SET_ADDR_LOW 0x00 /* set column address bits 0..3 */
#define SPLC501_CMD_SET_ADDR_HIGH 0x10 /* set column address bits 4..7 */
#define SPLC501_CMD_ADC_SELECT_NORM 0xA0 /* set relationship between
                                            RAM column address and display
                                            driver: normal */
#define SPLC501_CMD_ADC_SELECT_REV 0xA1  /* set relationship between
                                            RAM column address and display
                                            driver: reversed */
#define SPLC501_CMD_DISP_NORMAL 0xA6 /* set normal display mode */
#define SPLC501_CMD_DISP_REVERSE 0xA7 /* set reversed display mode */
#define SPLC501_CMD_PIXELS_NORMAL 0xA4 /* display the graphic RAM
                                          contents */
#define SPLC501_CMD_ALL_PIXELS_ON 0xA5 /* all display pixels are on */
#define SPLC501_CMD_DISP_OFF 0xAE /* display off */
#define SPLC501_CMD_DISP_ON 0xAF /* display on */
#define SPLC501_CMD_LCD_BIAS_19 0xA2 /* sets voltage ratio for LCD bias
                                        to 1/9 */
#define SPLC501_CMD_LCD_BIAS_17 0xA3 /* sets voltage ratio for LCD bias
                                        to 1/7 */
#define SPLC501_CMD_COM0_63 0xC0 /* sets the COM output scan
                                    direction 0->63 */
#define SPLC501_CMD_COM63_0 0xC8 /* sets the COM output scan
                                    direction 63->0 */
#define SPLC501_CMD_POWER_CTRL 0x28 /* turns on/off the
                                       voltage follower (| bit 0),
                                       voltage regulator (| bit 1),
                                       voltage booster (| bit 2) */
#define SPLC501_VOLT_FOLLOWER_ON (1<<0)  /* enable voltage follower */
#define SPLC501_VOLT_REGULATOR_ON (1<<1) /* enable voltage regulator */
#define SPLC501_VOLT_BOOSTER_ON (1<<2)   /* enable voltage booster */

#define SPLC501_CMD_VOLT_REG_V5 0x20 /* sets the V5 voltage regulator
                                        internal resistor ratio */
#define SPLC501_CMD_ELECTRONIC_VOLUME 0x81 /* sets the electronic volume
                                              register in order to control
                                              the V5 LCD drive voltage */
#define SPLC501_CMD_SET_DRIVING_MODE 0xD2 /* used to set the LCD driving
                                             mode */
#define SPLC501_CMD_RESET 0xE2 /* resets the controller */
```

A detailed description of the above mentioned commands can be found in the SPLC501C datasheet.

**void splc501_setcontrast(unsigned char contrast)**

Controls the LCD contrast.

*Parameter:*

**contrast** sets the value of the V5 LCD drive voltage, allowed range is 0..63.

The **glcd_splc501.h** header file also contains the definition of the **GLCDINIT_t** type specific for the SPLC501C controller, used as parameter for the **glcd_init** function:

```
typedef struct
        {
        flash unsigned char *font; /* default font after initialization */
        /* pointer to the function used for reading a byte from external
           memory */
        unsigned char (*readxmem) (GLCDMEMADDR_t addr);
        /* pointer to the function used for writing a byte to external
           memory */
        void (*writexmem) (GLCDMEMADDR_t addr, unsigned char data);

        unsigned char lcd_bias:1; /* =0 1/9 LCD bias, =1 1/7 LCD bias */
        unsigned char lcd_bias:1; /* =0 1/6 LCD bias, =1 1/5 LCD bias */
        unsigned char reverse_x:1; /* reverse display horizontally (ADC) */
        unsigned char rev132_x0:1; /* set to 1 for displays that use
                                      reversed RAM column address
                                      (reverse_x=1) driver and the pixel
                                      with x=0 is connected to column
                                      driver #132 */
        unsigned char reverse_y:1; /* reverse display vertically (COM) */
        unsigned char volt_reg_v5:3; /* set V5 voltage regulator internal
                                        resistor ratio [0..7] */
        unsigned char driving_mode:2; /* set LCD driving mode:
                                         0 - mode 1, 1 - mode 2,
                                         2 - mode 3, 3 - mode 4 */
        unsigned char lcd_contrast:5; /* LCD contrast voltage: [0..63] */
        } GLCDINIT_t;
```

The following macros are defined for initializing the members of the **GLCDINIT_t** structure:

```
/* values used for lcd_bias initialization */
#define SPLC501_LCD_BIAS_19 0 /* sets LCD bias drive ratio 1/9 */
#define SPLC501_LCD_BIAS_17 1 /* sets LCD bias drive ratio 1/7 */

/* values used for reverse_x initialization */
#define SPLC501_REVX_NORM 0 /* set relationship between RAM column address
and display driver: normal (ADC=0) */
#define SPLC501_REVX_REV 1  /* set relationship between RAM column address
and display driver: reversed (ADC=1) */

/* values used for rev132_x0 initilization,
   effective only when reverse_x=1 (SPLC501_REVX_REV) */
#define SPLC501_REV132_X0NC 0  /* pixel with x=0 is not connected to column
driver #132 when ADC=1 */
#define SPLC501_REV132_X0CON 1 /* pixel with x=0 is connected to column
driver #132 when ADC=1 */
```

```
/* values used for reverse_y initialization */
#define SPLC501_REVY_NORM 0 /* sets the vertical COM output scan direction
0->63 */
#define SPLC501_REVY_REV 1  /* sets the vertical COM output scan direction
63->0 */

/* values used for driving_mode initialization */
#define SPLC501_DRIVING_MODE1 0 /* driving mode 1 */
#define SPLC501_DRIVING_MODE2 1 /* driving mode 2 */
#define SPLC501_DRIVING_MODE3 2 /* driving mode 3 */
#define SPLC501_DRIVING_MODE4 3 /* driving mode 4 */

/* default initialization values */
/* default value for LCD bias */
#define SPLC501_DEFAULT_LCD_BIAS SPLC501_LCD_BIAS_19
/* default value for reverse_x */
#define SPLC501_DEFAULT_REVX SPLC501_REVX_NORM
/* default value for rev132_x0,
   effective only when reverse_x=1 (SPLC501_REVX_REV) */
#define SPLC501_DEFAULT_REV132_X0 SPLC501_REV132_X0NC
/* default value for reverse_y */
#define SPLC501_DEFAULT_REVY SPLC501_REVY_NORM
/* default V5 voltage regulator internal resistor ratio */
#define SPLC501_DEFAULT_VOLT_REG_V5 6
/* default LCD driving mode */
#define SPLC501_DEFAULT_DRIVING_MODE SPLC501_DRIVING_MODE1
/* default contrast */
#define SPLC501_DEFAULT_CONTRAST 7
```

The detailed description of the above mentioned initialization parameters can be found in the SPLC501C datasheet.

**Notes:**

- The **glcd_splc501.h** header file is automatically #included when the main **glcd.h** header file is #included. Therefore there is no need to explicitly #include **glcd_splc501.h**.
- The **..\EXAMPLES\Graphic LCD\SPLC501C** directory contains fully functional code samples that may be used as references for SPLC501C initialization and usage.

## 5.12.7 Graphic LCD Functions Specific to the SSD1289 Controller

The SSD1289 library functions supplied with the CodeVisionAVR Advanced license, operate the controller in 16 bit interface mode.
In order to take full advantage of the SSD1289 controller's features the following specific functions, declared in the **glcd_ssd1289.h** header file, were implemented:

**void ssd1289_wrcmd(unsigned char cmd)**

Writes a command to the SSD1289 controller, used to access a specific command register.

*Parameter:*

**cmd** command register index to be sent to the controller.

This index may take one of the values defined in the following macros from the **glcd_ssd1289.h** header file:

```
/* SSD1289 command register definitions */
#define SSD1289_CMD_OSC 0x00 /* Oscillator register */
#define SSD1289_CMD_DRIVER_OUT 0x01 /* Driver output control
#define SSD1289_CMD_DRIVING_WAVEFORM 0x02 /* LCD driving waveform
                                             control */
#define SSD1289_CMD_POWER_CONTROL1 0x03 /* Power control 1 */
#define SSD1289_CMD_CMP_REG1 0x05 /* Compare register 1 */
#define SSD1289_CMD_CMP_REG2 0x06 /* Compare register 2 */
#define SSD1289_CMD_DISPLAY_CONTROL 0x07 /* Display control register */
#define SSD1289_CMD_FRAME_CYCLE 0x0b /* Frame cycle control register */
#define SSD1289_CMD_POWER_CONTROL2 0x0c /* Power control 2 register */
#define SSD1289_CMD_POWER_CONTROL3 0x0d /* Power control 3 register */
#define SSD1289_CMD_POWER_CONTROL4 0x0e /* Power control 4 register */
#define SSD1289_CMD_GATE_SCAN_POS 0x0f /* Gate scan position register */
#define SSD1289_CMD_SLEEP_MODE 0x10 /* Sleep mode register */
#define SSD1289_CMD_ENTRY_MODE 0x11 /* Entry mode register */
#define SSD1289_CMD_GENERIC_IF_CTRL 0x15 /* Generic interface control
                                            register */
#define SSD1289_CMD_HORIZ_PORCH 0x16 /* Horizontal porch register */
#define SSD1289_CMD_VERT_PORCH 0x17 /* Vertical porch register */
#define SSD1289_CMD_POWER_CONTROL5 0x1e /* Power control 5 register */
#define SSD1289_CMD_GDDRAM_DATA 0x22 /* GDDRAM read/write data register */
#define SSD1289_CMD_GDDRAM_WR_MASK1 0x23 /* GDDRAM write data mask 1
                                            register */
#define SSD1289_CMD_GDDRAM_WR_MASK2 0x24 /* GDDRAM write data mask 2
                                            register */
#define SSD1289_CMD_FRAME_FREQ 0x25 /* Frame frequency control register */
#define SSD1289_CMD_GAMMA_CONTROL1 0x30 /* Gamma control 1 */
#define SSD1289_CMD_GAMMA_CONTROL2 0x31 /* Gamma control 2 */
#define SSD1289_CMD_GAMMA_CONTROL3 0x32 /* Gamma control 3 */
#define SSD1289_CMD_GAMMA_CONTROL4 0x33 /* Gamma control 4 */
#define SSD1289_CMD_GAMMA_CONTROL5 0x34 /* Gamma control 5 */
#define SSD1289_CMD_GAMMA_CONTROL6 0x35 /* Gamma control 6 */
#define SSD1289_CMD_GAMMA_CONTROL7 0x36 /* Gamma control 7 */
#define SSD1289_CMD_GAMMA_CONTROL8 0x37 /* Gamma control 8 */
#define SSD1289_CMD_GAMMA_CONTROL9 0x3a /* Gamma control 9 */
#define SSD1289_CMD_GAMMA_CONTROL10 0x3b /* Gamma control 10 */
#define SSD1289_CMD_VERT_SCROLL1 0x41 /* Vertical scroll control for
                                         screen 1 */
#define SSD1289_CMD_VERT_SCROLL2 0x42 /* Vertical scroll control for
                                         screen 2 */
```

```
#define SSD1289_CMD_HORIZ_RAM_ADDR 0x44 /* Addresses of horizontal
                                           start/end window positions */
#define SSD1289_CMD_VERT_RAM_ADDR_START 0x45 /* Address of vertical start
                                                window positions */
#define SSD1289_CMD_VERT_RAM_ADDR_END 0x46 /* Address of vertical end
                                              window positions */
#define SSD1289_CMD_DRV_POS_START1 0x48 /* Driving start line position for
                                           screen 1 */
#define SSD1289_CMD_DRV_POS_END1 0x49 /* Driving end line position for
                                         screen 1 */
#define SSD1289_CMD_DRV_POS_START2 0x4a /* Driving start line position for
                                           screen 2 */
#define SSD1289_CMD_DRV_POS_END2 0x4b /* Driving end line position for
                                         screen 2 */
#define SSD1289_CMD_GDDRAMX 0x4e /* Set GDDRAM X address counter
                                    register */
#define SSD1289_CMD_GDDRAMY 0x4f /* Set GDDRAM Y address counter
                                    register
```

A detailed description of the above mentioned command registers can be found in the SSD1289 datasheet.

**void ssd1289_wrreg(unsigned char index, unsigned short data)**

Writes data to a command register of the SSD1289 controller.

*Parameters:*

**index** command register index
**data** to be written.

**unsigned short ssd1289_rdreg(unsigned char index)**

Reads the contents of a command register of the SSD1289 controller.

*Parameters:*

**index** command register index.

**void ssd1289_wrdata(unsigned short data)**

Writes data to the SSD1289 controller's Graphic Display RAM.

*Parameters:*

**data** to be written.

**unsigned short ssd1289_rddata(void)**

Reads data from the SSD1289 controller's Graphic Display RAM.

The **glcd_ssd1289.h** header file also contains the definition of the **GLCDINIT_t** type specific for the SSD1289 controller, used as parameter for the **glcd_init** function:

```
typedef struct
        {
        flash unsigned char *font; /* default font after initialization */
        /* pointer to the function used for reading a byte from external
           memory */
        unsigned char (*readxmem) (GLCDMEMADDR_t addr);
        /* pointer to the function used for writing a byte to external
           memory */
        void (*writexmem) (GLCDMEMADDR_t addr, unsigned char data);
        unsigned char reverse_x:1; /* reverse display horizontally */
        unsigned char reverse_y:1; /* reverse display vertically */
        unsigned char cl_bits_order:1; /* selects the color bits writing
                                          order to the display RAM
                                          =0 -> RGB, =1 ->BGR */
        /* power control registers bits */
        unsigned char stepup_factor:3; /* step-up factor of the step-up
                                          circuit, see BT0..BT2 bits in
                                          the datasheet */
        unsigned char stepup_cycle:4;  /* controls the cycle for the
                                          step-up circuit */
        unsigned char crt_source:3; /* adjusts the amount of current
                                       from the constant current source
                                       in the internal op. amplifier
                                       circuit (AP0..AP2 bits) */
        unsigned char vcix2:3;  /* adjusts the VCIX2 voltage */
        unsigned char vlcd63:4; /* adjusts the VLCD63 voltage */
        unsigned char vcoml:5;  /* adjusts the amplitude of the
                                   VcomL alternating drive voltage */
        unsigned char vcomh:5;  /* adjusts the amplitude of the VcomH
                                   voltage
                                   VcomH=VLCD63*(0.35+vcomh*0.01) [V] */
        unsigned char frame_freq:4; /* LCD frame frequency */
        /* positive gamma control registers bits */
        unsigned char pkp00:3; /* PKP00..PKP02 positive gamma micro adj. */
        unsigned char pkp10:3; /* PKP10..PKP12 positive gamma micro adj. */
        unsigned char pkp20:3; /* PKP20..PKP22 positive gamma micro adj. */
        unsigned char pkp30:3; /* PKP30..PKP32 positive gamma micro adj. */
        unsigned char pkp40:3; /* PKP40..PKP42 positive gamma micro adj. */
        unsigned char pkp50:3; /* PKP50..PKP52 positive gamma micro adj. */
        unsigned char prp00:3; /* PRP00..PRP02 positive gamma gradient
                                  adj. */
        unsigned char prp10:3; /* PRP10..PRP12 positive gamma gradient
                                  adj. */
        unsigned char vrp00:4; /* VRP00..VRP03 positive gamma amplification
                                  adj. */
        unsigned char vrp10:5; /* VRP10..VRP14 positive gamma amplification
                                  adj. */
        /* negative gamma control registers bits */
        unsigned char pkn00:3; /* PKN00..PKN02 negative gamma micro adj. */
        unsigned char pkn10:3; /* PKN10..PKN12 negative gamma micro adj. */
        unsigned char pkn20:3; /* PKN20..PKN22 positive gamma micro adj. */
        unsigned char pkn30:3; /* PKN30..PKN32 positive gamma micro adj. */
        unsigned char pkn40:3; /* PKN40..PKN42 negative gamma micro adj. */
        unsigned char pkn50:3; /* PKN50..PKN52 negative gamma micro adj. */
        unsigned char prn00:3; /* PRN00..PRN02 negative gamma gradient
                                  adj. */
```

```
        unsigned char prn10:3; /* PRN10..PRN12 negative gamma gradient
                                  adj. */
        unsigned char vrn00:4; /* VRN00..VRN03 negative gamma amplification
                                  adj. */
        unsigned char vrn10:5; /* VRN10..VRN14 negative gamma
                                  amplification adj. */
        } GLCDINIT_t;
```

The following macros are defined for initializing the members of the **GLCDINIT_t** structure:

```
/* Initialization values for reverse_x */
#define SSD1289_REVX_NORM 0 /* No horizontal reverse */
#define SSD1289_REVX_REV 1 /* Horizontal reverse */

/* Initialization values for reverse_y */
#define SSD1289_REVY_NORM 0 /* No vertical reverse */
#define SSD1289_REVY_REV 1 /* Vertical reverse */

/* Initialization values for cl_bits_order */
#define SSD1289_CL_BITS_RGB 0 /* Write color bits to display RAM
                                 in RGB order */
#define SSD1289_CL_BITS_BGR 1 /* Write color bits to display RAM
                                 in BGR order */

/* Initilization values for dc30 step-up circuit cycle */
#define SSD1289_STEPUP_FLINE24 0 /* Fline *24 */
#define SSD1289_STEPUP_FLINE16 1 /* Fline *16 */
#define SSD1289_STEPUP_FLINE12 2 /* Fline *12 */
#define SSD1289_STEPUP_FLINE8 3 /* Fline *8 */
#define SSD1289_STEPUP_FLINE6 4 /* Fline *6 */
#define SSD1289_STEPUP_FLINE5 5 /* Fline *5 */
#define SSD1289_STEPUP_FLINE4 6 /* Fline *4 */
#define SSD1289_STEPUP_FLINE3 7 /* Fline *3 */
#define SSD1289_STEPUP_FLINE2 8 /* Fline *2 */
#define SSD1289_STEPUP_FLINE1 9 /* Fline *1 */
#define SSD1289_STEPUP_FOSC4 10 /* Fosc /4  (Fosc=510kHz) */
#define SSD1289_STEPUP_FOSC6 11 /* Fosc /6 */
#define SSD1289_STEPUP_FOSC8 12 /* Fosc /8 */
#define SSD1289_STEPUP_FOSC10 13 /* Fosc /10 */
#define SSD1289_STEPUP_FOSC12 14 /* Fosc /12 */
#define SSD1289_STEPUP_FOSC16 15 /* Fosc /16 */

/* Initialization values for the VCIX2 voltage */
#define SSD1289_VCIX2_5V1 0 /* 5.1V */
#define SSD1289_VCIX2_5V2 1 /* 5.2V */
#define SSD1289_VCIX2_5V3 2 /* 5.3V */
#define SSD1289_VCIX2_5V4 3 /* 5.4V */
#define SSD1289_VCIX2_5V5 4 /* 5.5V */
#define SSD1289_VCIX2_5V6 5 /* 5.6V */
#define SSD1289_VCIX2_5V7 6 /* 5.7V */
#define SSD1289_VCIX2_5V8 7 /* 5.8V */
```

```
/* Initialization values for the VLCD63 voltage */
#define SSD1289_VLCD63_3V08 0 /* VLCD63=3.08V */
#define SSD1289_VLCD63_3V24 1 /* VLCD63=3.24V */
#define SSD1289_VLCD63_3V40 2 /* VLCD63=3.40V */
#define SSD1289_VLCD63_3V56 3 /* VLCD63=3.56V */
#define SSD1289_VLCD63_3V70 4 /* VLCD63=3.70V */
#define SSD1289_VLCD63_3V86 5 /* VLCD63=3.86V */
#define SSD1289_VLCD63_4V04 6 /* VLCD63=4.04V */
#define SSD1289_VLCD63_4V18 7 /* VLCD63=4.18V */
#define SSD1289_VLCD63_4V33 8 /* VLCD63=4.33V */
#define SSD1289_VLCD63_4V49 9 /* VLCD63=4.49V */
#define SSD1289_VLCD63_4V67 10 /* VLCD63=4.67V */
#define SSD1289_VLCD63_4V80 11 /* VLCD63=4.80V */
#define SSD1289_VLCD63_5V00 12 /* VLCD63=5.00V */
#define SSD1289_VLCD63_5V14 13 /* VLCD63=5.14V */
#define SSD1289_VLCD63_5V29 14 /* VLCD63=5.29V */
#define SSD1289_VLCD63_5V45 15 /* VLCD63=5.45V */

/* Initialization values for the VcomL voltage */
#define SSD1289_VCOML_0_60 0 /* VcomL=VLCD63*0.60 */
#define SSD1289_VCOML_0_63 1 /* VcomL=VLCD63*0.63 */
#define SSD1289_VCOML_0_66 2 /* VcomL=VLCD63*0.66 */
#define SSD1289_VCOML_0_69 3 /* VcomL=VLCD63*0.69 */
#define SSD1289_VCOML_0_72 4 /* VcomL=VLCD63*0.72 */
#define SSD1289_VCOML_0_75 5 /* VcomL=VLCD63*0.75 */
#define SSD1289_VCOML_0_78 6 /* VcomL=VLCD63*0.78 */
#define SSD1289_VCOML_0_81 7 /* VcomL=VLCD63*0.81 */
#define SSD1289_VCOML_0_84 8 /* VcomL=VLCD63*0.84 */
#define SSD1289_VCOML_0_87 9 /* VcomL=VLCD63*0.87 */
#define SSD1289_VCOML_0_90 10 /* VcomL=VLCD63*0.90 */
#define SSD1289_VCOML_0_93 11 /* VcomL=VLCD63*0.93 */
#define SSD1289_VCOML_0_96 12 /* VcomL=VLCD63*0.96 */
#define SSD1289_VCOML_0_99 13 /* VcomL=VLCD63*0.99 */
#define SSD1289_VCOML_1_02 14 /* VcomL=VLCD63*1.02 */
#define SSD1289_VCOML_EXT_RES 15 /* VcomL is set by an external
                                    variable resistor */
#define SSD1289_VCOML_1_05 16 /* VcomL=VLCD63*1.05 */
#define SSD1289_VCOML_1_08 17 /* VcomL=VLCD63*1.08 */
#define SSD1289_VCOML_1_11 18 /* VcomL=VLCD63*1.11 */
#define SSD1289_VCOML_1_14 19 /* VcomL=VLCD63*1.14 */
#define SSD1289_VCOML_1_17 20 /* VcomL=VLCD63*1.17 */
#define SSD1289_VCOML_1_20 21 /* VcomL=VLCD63*1.20 */
#define SSD1289_VCOML_1_23 22 /* VcomL=VLCD63*1.23 */

/* Initialization values for frame_freq */
#define SSD1289_FRAME50 0 /* 50Hz */
#define SSD1289_FRAME55 2 /* 55Hz */
#define SSD1289_FRAME60 5 /* 60Hz */
#define SSD1289_FRAME65 8 /* 65Hz */
#define SSD1289_FRAME70 0x0A /* 70Hz */
#define SSD1289_FRAME75 0x0C /* 75Hz */
#define SSD1289_FRAME80 0x0E /* 80Hz */
```

```
/* Default value for reverse_x */
#define SSD1289_DEFAULT_REVX SSD1289_REVX_NORM /* No horizontal reverse */
/* Default value for reverse_y */
#define SSD1289_DEFAULT_REVY SSD1289_REVY_NORM /* No vertical reverse */

/* Default value for cl_bits_order
  (color bits writing order to display RAM),
  write in RGB order */
#define SSD1289_DEFAULT_CL_BITS SSD1289_CL_BITS_RGB

/* Power control 1 BT0..BT2 step-up factor of the step-up circuit */
#define SSD1289_DEFAULT_STEPUP_FACTOR 4
/* Power control 1 DC0..DC3 step-up circuit cycle */
#define SSD1289_DEFAULT_STEPUP_CYCLE SSD1289_STEPUP_FOSC4
/* Power control 1 AP0..AP2 adjusts the amount of current
   from the constant current source in the internal operational
   amplifier circuit */
#define SSD1289_DEFAULT_CRT_SOURCE 2
/* Default value for VCIX2 voltage */
#define SSD1289_DEFAULT_VCIX2 SSD1289_VCIX2_5V1
/* Default value for VLCD63 voltage */
#define SSD1289_DEFAULT_VLCD63 SSD1289_VLCD63_4V80
/* Default value for VcomL alternating drive voltage */
#define SSD1289_DEFAULT_VCOML SSD1289_VCOML_0_72
/* Default value for VcomH=VLCD63*(0.35+0x1A*0.01) */
#define SSD1289_DEFAULT_VCOMH 0x1A
/* Default value for driving waveform control FLD bit,
   splits one frame into 3 fields to reduce flicker */
#define SSD1289_DEFAULT_FLD 1
/* Default value for LCD frame frequency */
#define SSD1289_DEFAULT_FRAME_FREQ SSD1289_FRAME80

/* Default initialization values for the gamma control register bits */
/* PKP00..PKP02 positive gamma micro adj. */
#define SSD1289_DEFAULT_PKP00 7
/* PKP10..PKP12 positive gamma micro adj. */
#define SSD1289_DEFAULT_PKP10 7
/* PKP20..PKP22 positive gamma micro adj. */
#define SSD1289_DEFAULT_PKP20 4
/* PKP30..PKP32 positive gamma micro adj. */
#define SSD1289_DEFAULT_PKP30 2
/* PKP40..PKP42 positive gamma micro adj. */
#define SSD1289_DEFAULT_PKP40 4
/* PKP50..PKP52 positive gamma micro adj. */
#define SSD1289_DEFAULT_PKP50 2
/* PRP00..PRP02 positive gamma gradient adj. */
#define SSD1289_DEFAULT_PRP00 2
/* PRP10..PRP12 positive gamma gradient adj. */
#define SSD1289_DEFAULT_PRP10 5
/* VRP00..VRP03 positive gamma amplification adj. */
#define SSD1289_DEFAULT_VRP00 2
/* VRP10..VRP14 positive gamma amplification adj. */
#define SSD1289_DEFAULT_VRP10 3
/* PKN00..PKN02 negative gamma micro adj. */
#define SSD1289_DEFAULT_PKN00 7
/* PKN10..PKN12 negative gamma micro adj. */
#define SSD1289_DEFAULT_PKN10 5
/* PKN20..PKN22 positive gamma micro adj. */
#define SSD1289_DEFAULT_PKN20 4
/* PKN30..PKN32 positive gamma micro adj. */
```

```
#define SSD1289_DEFAULT_PKN30 2
/* PKN40..PKN42 negative gamma micro adj. */
#define SSD1289_DEFAULT_PKN40 4
/* PKN50..PKN52 negative gamma micro adj. */
#define SSD1289_DEFAULT_PKN50 2
/* PRN00..PRN02 negative gamma gradient adj. */
#define SSD1289_DEFAULT_PRN00 2
/* PRN10..PRN12 negative gamma gradient adj. */
#define SSD1289_DEFAULT_PRN10 5
/* VRN00..VRN03 negative gamma amplification adj. */
#define SSD1289_DEFAULT_VRN00 2
/* VRN10..VRN14 negative gamma amplification adj. */
#define SSD1289_DEFAULT_VRN10 3
```

The following colors are predefined in the **glcd_ssd1289.h** header file:

GLCD_CL_BLACK
GLCD_CL_WHITE
GLCD_CL_GRAY
GLCD_CL_LIGHT_GRAY
GLCD_CL_GREEN
GLCD_CL_LIME
GLCD_CL_BLUE
GLCD_CL_RED
GLCD_CL_AQUA
GLCD_CL_YELLOW
GLCD_CL_MAGENTA
GLCD_CL_CYAN
GLCD_CL_DARK_CYAN
GLCD_CL_ORANGE
GLCD_CL_PINK
GLCD_CL_BROWN
GLCD_CL_VIOLET
GLCD_CL_SILVER
GLCD_CL_GOLD
GLCD_CL_NAVY
GLCD_CL_MAROON
GLCD_CL_PURPLE
GLCD_CL_OLIVE

The SSD1289 library functions can operate the display in 256 or 64k color modes.
For 256 color mode the following color bit allocation in a data byte is used:
- Bits 0..2 - Blue color bits 0..2
- Bits 3..5 - Green color bits 0..2
- Bits 6..7 - Red color bits 0..1.

For 64k color mode the following color bit allocation in a 16bit data word is used:
- Bits 0..4 - Blue color bits 0..4
- Bits 5..10 - Green color bits 0..5
- Bits 11..15 - Red color bits 0..4.

**Notes:**
- In order to reduce image storage size and improve speed, it is recommended to use the 256 color mode, if possible.
- The **glcd_ssd1289.h** header file is automatically #included when the main **glcd.h** header file is #included. Therefore there is no need to explicitly #include **glcd_ssd1289.h**.
- The **..\EXAMPLES\Graphic LCD\SSD1289** directory contains fully functional code samples that may be used as references for SSD1289 initialization and usage.

## 5.12.8 Graphic OLED Display Functions Specific to the SSD1303 and SH1101A Controllers

In order to take full advantage of the Solomon Systech SSD1303 and Sino Wealth SH1101A OLED controllers' features, the following specific functions, declared in the **glcd_st7565.h** header file, were implemented:

**void ssd1303_wrcmd(unsigned char cmd)**

Writes a command to the SSD1303 controller.

*Parameter:*

**cmd** command to be sent to the controller.

This command may take one of the values defined in the following macros from the **glcd_ssd1303.h** header file:

```
#define SSD1303_CMD_SET_ADDR_LOW 0x00 /* set column address bits 0..3 */
#define SSD1303_CMD_SET_ADDR_HIGH 0x10 /* set column address bits 4..7 */
#define SSD1303_CMD_SET_HORIZ_SCROLL 0x26 /* horizontal scroll setup */
#define SSD1303_CMD_HORIZ_SCROLL_OFF 0x2E /* deactivate horizontal scroll */
#define SSD1303_CMD_HORIZ_SCROLL_ON 0x2F /* activate horizontal scroll */
#define SSD1303_CMD_START_LINE 0x40 /* set display start line */
#define SSD1303_CMD_SET_PAGE 0xB0 /* set display page address */

#define SSD1303_CMD_SET_CONTRAST 0x81 /* sets the contrast control register */
#define SSD1303_CMD_SET_BRIGHTNESS 0x82 /* sets the brightness control register */
#define SSD1303_CMD_SET_LUT 0x91 /* sets the Look Up Table */
#define SSD1303_CMD_SET_COLOR_BANK1_16 0x92 /* sets the colors for banks 1-16 */
#define SSD1303_CMD_SET_COLOR_BANK17_32 0x93 /* sets the colors for banks 17-32 */

#define SSD1303_CMD_ADC_SELECT_NORM 0xA0 /* set the relationship between RAM column
                                            address and display driver: normal */
#define SSD1303_CMD_ADC_SELECT_REV 0xA1    /* set the relationship between RAM column
                                            address and display driver: reversed */
#define SSD1303_CMD_DISP_NORMAL 0xA6 /* set normal display mode */
#define SSD1303_CMD_DISP_REVERSE 0xA7 /* set reversed display mode */
#define SSD1303_CMD_MUX_RATIO 0xA8 /* set multiplex ratio */
#define SSD1303_CMD_PIXELS_NORMAL 0xA4 /* display the graphic RAM contents */
#define SSD1303_CMD_ALL_PIXELS_ON 0xA5 /* all display pixels are on */
#define SSD1303_CMD_DC_DC 0xAD /* turn the DC/DC converter on/off */
#define SSD1303_DC_DC_OFF 0x8A /* second byte of the command: DC/DC off */
#define SSD1303_DC_DC_ON 0x8B /* second byte of the command: DC/DC on */

#define SSD1303_CMD_DISP_OFF 0xAE /* display off */
#define SSD1303_CMD_DISP_ON 0xAF /* display on */

#define SSD1303_CMD_COM0_63 0xC0 /* sets the COM output scan direction 0->63 */
#define SSD1303_CMD_COM63_0 0xC8 /* sets the COM output scan direction 63->0 */

#define SSD1303_CMD_DISPL_OFFSET 0xD3 /* set display offset */
#define SSD1303_CMD_DISPL_CLK 0xD5 /* set display clock division ratio and
                                      oscillator frequency */
#define SSD1303_CMD_AREA_COLOR_POWER_SAVE 0xD8 /* set area color and
                                                  power-save modes */
#define SDD1303_CMD_POWER_SAVE_OFF 0 /* second command byte for power-save off */
#define SDD1303_CMD_POWER_SAVE_ON 0x05 /* second command byte for power-save on */
```

#define SSD1303_CMD_PRECHARGE_PERIOD 0xD9 /* set pre-charge period for Phase 1 and Phase 2 */
#define SSD1303_CMD_COM_CONFIG 0xDA /* set COM pins hardware configuration: interlaced or non-interlaced */
#define SSD1303_CMD_INTERLACED 0x12 /* second command byte for interlaced operation */
#define SSD1303_CMD_NON_INTERLACED 0x02 /* second command byte for non-interlaced operation */

#define SSD1303_CMD_VCOM_DESELECT 0xDB /* set VCOM deselect level */

A detailed description of the above mentioned commands can be found in the SSD1303 datasheet.

**void ssd1303_setcontrast(unsigned char contrast)**

Controls the OLED display contrast.

*Parameter:*

**contrast** sets the contrast value, the allowed range being 0..255.

The **glcd_ssd1303.h** header file also contains the definition of the **GLCDINIT_t** type specific for the SSD1303 controller, used as parameter for the **glcd_init** function:

```
typedef struct
        {
        flash unsigned char *font; /* default font after initialization */
        /* pointer to the function used for reading a byte from external
memory */
        unsigned char (*readxmem) (GLCDMEMADDR_t addr);
        /* pointer to the function used for writing a byte to external
memory */
        void (*writexmem) (GLCDMEMADDR_t addr, unsigned char data);

        unsigned char reverse_x:1; /* reverse display horizontally (ADC) */
        unsigned char reverse_xoffs:3; /* specify the X offset when
                                          plotting pixels if reverse_x=1,
                                          the usual value is
                                          132-_GLCD_MAXX_ */
        unsigned char reverse_y:1; /* reverse display vertically (COM) */
        unsigned char interlaced:1; /* use vertically interlaced display */
        unsigned char contrast; /* OLED display contrast */
        } GLCDINIT_t;
```

The following macros are defined for initializing the members of the **GLCDINIT_t** structure:

```
/* values used for reverse_x initialization */
#define SSD1303_REVX_NORM 0 /* set the relationship between RAM column
                               address and display driver: normal */
#define SSD1303_REVX_REV 1  /* set the relationship between RAM column
                               address and display driver: reversed */

/* values used for reverse_y initialization */
#define SSD1303_REVY_NORM 0 /* sets the vertical COM output scan direction
0->63 */
#define SSD1303_REVY_REV 1  /* sets the vertical COM output scan direction
63->0 */
```

```
/* values used for interlaced initialization */
#define SSD1303_NON_INTERLACED 0
#define SSD1303_INTERLACED 1

/* default initialization values */
/* default value for reverse_x */
#define SSD1303_DEFAULT_REVX SSD1303_REVX_NORM
/* default value for reverse_xoffs
   effective only when reverse_x=1 (SSD1303_REVX_REV) */
#define SSD1303_DEFAULT_REV_XOFFS (132-_GLCD_MAXX_)
/* default value for reverse_y */
#define SSD1303_DEFAULT_REVY SSD1303_REVY_NORM
/* default value for interlaced */
#define SSD1303_DEFAULT_INTERLACED SSD1303_INTERLACED
/* default contrast */
#define SSD1303_DEFAULT_CONTRAST 128
```

The detailed description of the above mentioned initialization parameters can be found in the SSD1303 datasheet.

**Notes:**

- The **glcd_ssd1303.h** header file is automatically #included when the main **glcd.h** header file is #included. Therefore there is no need to explicitly #include **glcd_ssd1303.h**.
- The **..\EXAMPLES\Graphic LCD\SSD1303** directory contains fully functional code samples that may be used as references for SSD1303, SH1101A initialization and usage.

## 5.12.9 Graphic LCD Functions Specific to the SSD1963 Controller

The SSD1963 library functions supplied with the CodeVisionAVR Advanced license, can operate the controller in 8 and 16 bit interface modes. To obtain higher display speed, the 16 bit interface mode is recommended.

In order to take full advantage of the SSD1963 controller's features the following specific functions, declared in the **glcd_ssd1963.h** header file, were implemented:

**void ssd1963_wrcmd(unsigned char cmd)**

Writes a command to the SSD1963 controller.

*Parameter:*

**cmd** command to be sent to the controller.

This command may take one of the values defined in the following macros from the **glcd_ssd1963.h** header file:

```
#define SSD1963_CMD_NOP 0x00 /* No operation */
#define SSD1963_CMD_SOFT_RESET 0x01 /* Software reset */
#define SSD1963_CMD_GET_PWR_MODE 0x0A /* Get the current power mode */
#define SSD1963_CMD_GET_ADDR_MODE 0x0B /* Get the frame buffer to the
display panel read order */
#define SSD1963_CMD_GET_DISPLAY_MODE 0x0D /* Returns the display
                                             image mode status */
#define SSD1963_CMD_GET_SIGNAL_MODE 0x0E /* Get the current display signal
                                            mode from the peripheral */
#define SSD1963_CMD_ENT_SLEEP 0x10 /* Enter sleep mode */
#define SSD1963_CMD_EXIT_SLEEP 0x11 /* Exit from sleep mode */
#define SSD1963_CMD_ENT_PARTIAL_MODE 0x12 /* Enter partial display mode */
#define SSD1963_CMD_ENT_NORMAL_MODE 0x13 /* Enter normal display mode */
#define SSD1963_CMD_EXIT_INVERT_MODE 0x20 /* Exit from inverted display
                                             mode */
#define SSD1963_CMD_ENT_INVERT_MODE 0x21 /* Enter inverted display mode */
#define SSD1963_CMD_SET_GAMMA 0x26 /* Selects the gamma curve used by the
                                      display device */
#define SSD1963_CMD_BLANK_DISPLAY 0x28 /* Set display off, without clearing
                                          the frame buffer */
#define SSD1963_CMD_ON_DISPLAY 0x29 /* Set display on */
#define SSD1963_CMD_SET_COLUMN_ADDR 0x2A /* Set the column extent of the
                                            frame buffer accessed with the
                                            SSD1963_CMD_RD_MEM_CONT and
                                            SSD1963_CMD_WR_MEM_CONT
                                            commands */
#define SSD1963_CMD_SET_PAGE_ADDR 0x2B /* Set the page extent of the frame
                                          buffer accessed with the
                                          SSD1963_CMD_RD_MEM_CONT and
                                          SSD1963_CMD_WR_MEM_CONT
                                          commands */
#define SSD1963_CMD_WR_MEM_START 0x2C /* Transfer image information from uC
                                         to SSD1963 starting with the
                                         location specified by
                                         SSD1963_CMD_SET_COLUMN_ADDR and
                                         SSD1963_CMD_SET_PAGE_ADDR */
```

```
#define SSD1963_CMD_RD_MEM_START 0x2E /* Transfer image information from
                                         SSD1963 to uC starting with the
                                         location specified by
                                         SSD1963_CMD_SET_COLUMN_ADDR and
                                      /* SSD1963_CMD_SET_PAGE_ADDR */
#define SSD1963_CMD_SET_PARTIAL_AREA 0x30 /* Defines the partial display
                                             mode area */
#define SSD1963_CMD_SET_SCROLL_AREA 0x33 /* Defines the vertical scrolling
                                            and fixed areas */
#define SSD1963_CMD_SET_TEAR_OFF 0x34 /* Disable sending synchronization
                                         information from the display */
#define SSD1963_CMD_SET_TEAR_ON 0x35 /* Enable sending synchronization
                                        information from the display */
#define SSD1963_CMD_SET_ADDR_MODE 0x36 /* Set read order from uC to frame
                                          buffer and from frame buffer to
                                          the display panel */
#define SSD1963_CMD_SET_SCROLL_START 0x37 /* Set the start of the vertical
                                             scrolling area in the frame
                                             buffer */
#define SSD1963_CMD_EXIT_IDLE_MODE 0x38 /* Exit idle mode */
#define SSD1963_CMD_ENT_IDLE_MODE 0x39 /* Enter idle mode */
#define SSD1963_CMD_SET_PIXEL_FORMAT 0x3A /* Set pixel format: */
#define SSD1963_PIXEL_3BIT 0x10 /* 3-bit/pixel */
#define SSD1963_PIXEL_8BIT 0x20 /* 8-bit/pixel */
#define SSD1963_PIXEL_12BIT 0x30 /* 12-bit/pixel */
#define SSD1963_PIXEL_16BIT 0x50 /* 16-bit/pixel */
#define SSD1963_PIXEL_18BIT 0x60 /* 18-bit/pixel */
#define SSD1963_PIXEL_24BIT 0x70 /* 24-bit/pixel */

#define SSD1963_CMD_WR_MEM_CONT 0x3C /* Transfer image data from uC to
                                        SSD1963 continuing from the
                                        last SSD1963_CMD_WR_MEM_START or
                                        SSD1963_CMD_WR_MEM_CONT */
#define SSD1963_CMD_RD_MEM_CONT 0x3E /* Transfer image data from SSD1963 to
                                        uC continuing from the
                                        last SSD1963_CMD_RD_MEM_START or
                                        SSD1963_CMD_RD_MEM_CONT */
#define SSD1963_CMD_SET_TEAR_SCANLINE 0x44 /* Enable sending the TE signal
                                              to the uC when the
                                              display refresh reaches the
                                              provided scanline */
#define SSD1963_CMD_GET_TEAR_SCANLINE 0x45 /* Get the current scanline */
#define SSD1963_CMD_RD_DDB 0xA1 /* Read the Device Descriptor Block
                                   of SSD1963: */
#define SSD1963_SUPPLIER_ID 0x5701 /* Solomon Systech supplier ID */
#define SSD1963_PRODUCT_ID 0x61 /* SSD1963 product ID */
#define SSD1963_REVISION 0x01 /* SSD1963 minimal revision */

#define SSD1963_CMD_SET_LCD_MODE 0xB0 /* Set LCD panel mode and
                                         resolution */
#define SSD1963_CMD_GET_LCD_MODE 0xB1 /* Get LCD panel mode and
                                         resolution */
#define SSD1963_CMD_SET_HOR_PERIOD 0xB4 /* Set front porch settings */
#define SSD1963_CMD_GET_HOR_PERIOD 0xB5 /* Get front porch settings */
#define SSD1963_CMD_SET_VER_PERIOD 0xB6 /* Set the vert. blanking interval
                                           between last scan line
                                           and next LFRAME pulse */
#define SSD1963_CMD_GET_VER_PERIOD 0xB7 /* Get the vert. blanking interval
                                           between last scan line
                                           and next LFRAME pulse */
```

```
#define SSD1963_CMD_SET_GPIO_CONF 0xB8  /* Set GPIO configuration */
#define SSD1963_CMD_GET_GPIO_CONF 0xB9  /* Get GPIO configuration */
#define SSD1963_CMD_SET_GPIO_VAL 0xBA /* Write data to the GPIOs configured
                                         as outputs */
#define SSD1963_CMD_GET_GPIO_STATUS 0xBB /* Read data from the GPIOs
                                            configured as inputs */
#define SSD1963_CMD_SET_POST_PROC 0xBC /* Set the image post processor */
#define SSD1963_CMD_GET_POST_PROC 0xBD /* Get the image post processor */
#define SSD1963_CMD_SET_PWM_CONF 0xBE /* Set PWM configuration */
#define SSD1963_CMD_GET_PWM_CONF 0xBF /* Get PWM configuration */
#define SSD1963_CMD_SET_LCD_GEN0 0xC0 /* Set the rise, fall, period and
                                         toggling properties
                                         of LCD signal generator 0 */
#define SSD1963_CMD_GET_LCD_GEN0 0xC1 /* Get the rise, fall, period and
                                         toggling properties
                                         of LCD signal generator 0 */
#define SSD1963_CMD_SET_LCD_GEN1 0xC2 /* Set the rise, fall, period and
                                         toggling properties
                                         of LCD signal generator 1 */
#define SSD1963_CMD_GET_LCD_GEN1 0xC3 /* Get the rise, fall, period and
                                         toggling properties
                                         of LCD signal generator 1 */
#define SSD1963_CMD_SET_LCD_GEN2 0xC4 /* Set the rise, fall, period and
                                         toggling properties
                                         of LCD signal generator 2 */
#define SSD1963_CMD_GET_LCD_GEN2 0xC5 /* Get the rise, fall, period and
                                         toggling properties
                                         of LCD signal generator 2 */
#define SSD1963_CMD_SET_LCD_GEN3 0xC6 /* Set the rise, fall, period and
                                         toggling properties
                                         of LCD signal generator 3 */
#define SSD1963_CMD_GET_LCD_GEN3 0xC7 /* Get the rise, fall, period and
                                         toggling properties
                                         of LCD signal generator 3 */
#define SSD1963_CMD_SET_GPIO0_ROP 0xC8 /* Set GPIO0 with respect to LCD
                                          signal generators using
                                          ROP3 operation */
#define SSD1963_CMD_GET_GPIO0_ROP 0xC9 /* Get GPIO0 properties with respect
                                          to LCD signal generators */
#define SSD1963_CMD_SET_GPIO1_ROP 0xCA /* Set GPIO1 with respect to LCD
                                          signal generators using
                                          ROP3 operation */
#define SSD1963_CMD_GET_GPIO1_ROP 0xCB /* Get GPIO1 properties with respect
                                          to LCD signal generators */
#define SSD1963_CMD_SET_GPIO2_ROP 0xCC /* Set GPIO2 with respect to LCD
                                          signal generators using
                                          ROP3 operation */
#define SSD1963_CMD_GET_GPIO2_ROP 0xCD /* Get GPIO2 properties with respect
                                          to LCD signal generators */
#define SSD1963_CMD_SET_GPIO3_ROP 0xCE /* Set GPIO3 with respect to LCD
                                          signal generators using
                                          ROP3 operation */
#define SSD1963_CMD_GET_GPIO3_ROP 0xCF /* Get GPIO3 properties with respect
                                          to LCD signal generators */
#define SSD1963_CMD_SET_DBC_CONF 0xD0 /* Set Dynamic Backlight Control
                                         configuration */
#define SSD1963_CMD_GET_DBC_CONF 0xD1 /* Get Dynamic Backlight Control
                                         configuration */
#define SSD1963_CMD_SET_DBC_THRES 0xD4 /* Set the threshold for each level
                                          of power saving */
```

```
#define SSD1963_CMD_GET_DBC_THRES 0xD5 /* Get the threshold for each level
                                          of power saving */
#define SSD1963_CMD_SET_PLL 0xE0 /* Start the PLL */
#define SSD1963_CMD_SET_PLL_MN 0xE2 /* Set the PLL divider (M) and
                                       multiplier (N) */
#define SSD1963_CMD_GET_PLL_MN 0xE3 /* Get the PLL divider (M) and
                                       multiplier (N) */
#define SSD1963_CMD_GET_PLL_STATUS 0xE4   /* Get the current PLL status */
#define SSD1963_CMD_DEEP_SLEEP 0xE5 /* Set deep sleep mode, PLL will be
                                       stopped */
#define SSD1963_CMD_SET_PCLK 0xE6   /* Set pixel clock (LSHIFT signal)
                                       frequency */
#define SSD1963_CMD_GET_PCLK 0xE7   /* Get pixel clock (LSHIFT signal)
                                       frequency settings */
#define SSD1963_CMD_SET_PDATA_INTERFACE 0xF0 /* Set the pixel data format
                                                used for parallel mode
                                                communication with
                                                the uC: */
#define SSD1963_PIXEL_DATA_8BIT 0 /* 8-bit */
#define SSD1963_PIXEL_DATA_9BIT 6 /* 9-bit */
#define SSD1963_PIXEL_DATA_12BIT 1 /* 12-bit */
#define SSD1963_PIXEL_DATA_16BIT 2 /* 16-bit packed */
#define SSD1963_PIXEL_DATA_16BIT565 3 /* 16-bit 565 format */
#define SSD1963_PIXEL_DATA_18BIT 4 /* 18-bit */
#define SSD1963_PIXEL_DATA_24BIT 5 /* 24-bit */

#define SSD1963_CMD_GET_PDATA_INTERFACE 0xF1 /* Get the pixel data format
                                                used for parallel mode
                                                communication with
                                                the uC */
```

**void ssd1963_wrdata(unsigned char data)**

Writes data byte(s) to the SSD1963 controller after a command was issued using the **ssd1963_wrcmd** function.

*Parameter:*

**data** to be sent to the controller.

**unsigned char ssd1963_rddata(void)**

Reads result data byte(s) from the SSD1963 controller after a command was issued using the **ssd1963_wrcmd** function.

**void ssd1963_wrdram(GLCDCOL_t color)**

Writes color data for 1 pixel to the SSD1963 controller's display RAM.

*Parameter:*

**color** data to be sent to the controller.

**Note:** Before calling this function a **SSD1963_CMD_WR_MEM_START** or **SSD1963_CMD_WR_MEM_CONT** command must be issued to the controller using the **ssd1963_wrcmd** function.

**GLCDCOL_t ssd1963_rddram(void)**

Reads color data for 1 pixel from the SSD1963 controller's display RAM.

**Note:** Before calling this function a **SSD1963_CMD_RD_MEM_START** or **SSD1963_CMD_RD_MEM_CONT** command must be issued to the controller using the **ssd1963_wrcmd** function.

**void ssd1963_sleep(bool on)**

Puts the SSD1963 controller in sleep mode or exit from sleep mode.

*Parameter:*

**on** when **true** puts the controller in sleep mode, when **false** exits the sleep mode.

**Notes:**
- The function automatically inserts a 5ms delay after entering or exiting the sleep mode.
- A delay of minimum 120ms must be present after exiting the sleep mode and entering sleep mode again.

The **glcd_ssd1963.h** header file also contains the definition of the **GLCDINIT_t** type specific for the SSD1963 controller, used as parameter for the **glcd_init** function:

```
typedef struct
        {
        flash unsigned char *font; /* default font after initialization */
        /* pointer to the function used for reading a byte from external
           memory */
        unsigned char (*readxmem) (GLCDMEMADDR_t addr);
        /* pointer to the function used for writing a byte to external
           memory */
        void (*writexmem) (GLCDMEMADDR_t addr, unsigned char data);
        unsigned short ctrl_clk; /* SSD1963 controller external clock
                                    (crystal) frequency [kHz] */
        unsigned short tft_pixel_clk; /* TFT pixel clock frequency [kHz] */
        unsigned char hpulse_width; /* TFT panel horizontal pulse width
                                       [pixel clock cycles] */
        unsigned char hfront_porch; /* TFT panel horizontal front porch
                                       width [pixel clock cycles] */
        unsigned char hback_porch; /* TFT panel horizontal back porch width
                                      [pixel clock cycles] */
        unsigned char vpulse_width; /* TFT panel vertical pulse width
                                       [line cycles] */
        unsigned char vfront_porch; /* TFT panel vertical front porch width
                                       [line cycles] */
        unsigned char vback_porch; /* TFT panel vertical back porch width
                                      [line cycles] */
        unsigned char tft24bit:1; /* specify TFT panel data width
                                     =0 - 18bit, =1 - 24bit */
        unsigned char reverse_x:1; /* reverse display horizontally */
        unsigned char reverse_y:1; /* reverse display vertically */
        unsigned char cl_bits_order:1; /* selects the color bits writing
                                          order to the display RAM
                                          =0 -> RGB, =1 ->BGR */
        } GLCDINIT_t;
```

The following macros are defined for initializing the members of the **GLCDINIT_t** structure:

```
/* Initialization values for tft24bit */
#define SSD1963_TFT_DATA_WIDTH18 0 /* TFT panel data width 18 bit */
#define SSD1963_TFT_DATA_WIDTH24 1 /* TFT panel data width 24 bit */

/* Initialization values for reverse_x */
#define SSD1963_REVX_NORM 0 /* No horizontal reverse */
#define SSD1963_REVX_REV 1 /* Horizontal reverse */

/* Initialization values for reverse_y */
#define SSD1963_REVY_NORM 0 /* No vertical reverse */
#define SSD1963_REVY_REV 1 /* Vertical reverse */

/* Initialization values for cl_bits_order */
/* Write color bits to display RAM in RGB order */
#define SSD1963_CL_BITS_RGB 0
/* Write color bits to display RAM in BGR order */
#define SSD1963_CL_BITS_BGR 1

/* Default initialization values for GLCDINIT_t */
#define SSD1963_DEFAULT_CTRL_CLK 10000 /* external clock frequency [kHz] */

#if ((_GLCD_MAXX_==320) && (_GLCD_MAXY_==240))
/* 3.5" 320x240 display */
/* TFT LCD pixel clock frequency [kHz] */
#define SSD1963_DEFAULT_TFT_PIXEL_CLK_FREQ 6500

#define SSD1963_DEFAULT_TFT_HOR_PULSE_WIDTH 20
#define SSD1963_DEFAULT_TFT_HOR_FRONT_PORCH 20
#define SSD1963_DEFAULT_TFT_HOR_BACK_PORCH 48

#define SSD1963_DEFAULT_TFT_VER_PULSE_WIDTH 2
#define SSD1963_DEFAULT_TFT_VER_FRONT_PORCH 4
#define SSD1963_DEFAULT_TFT_VER_BACK_PORCH 16

/* default tft24bit to 18 bit TFT */
#define SSD1963_DEFAULT_TFT_DATA_WIDTH SSD1963_TFT_DATA_WIDTH18

#elif ((_GLCD_MAXX_==480) && (_GLCD_MAXY_==272))
/* 4.3" 480x272 display */
/* TFT LCD pixel clock frequency [kHz] */
#define SSD1963_DEFAULT_TFT_PIXEL_CLK_FREQ 9000

#define SSD1963_DEFAULT_TFT_HOR_PULSE_WIDTH 41
#define SSD1963_DEFAULT_TFT_HOR_FRONT_PORCH 2
#define SSD1963_DEFAULT_TFT_HOR_BACK_PORCH 2

#define SSD1963_DEFAULT_TFT_VER_PULSE_WIDTH 10
#define SSD1963_DEFAULT_TFT_VER_FRONT_PORCH 2
#define SSD1963_DEFAULT_TFT_VER_BACK_PORCH 2

/* default tft24bit to 18 bit TFT */
#define SSD1963_DEFAULT_TFT_DATA_WIDTH SSD1963_TFT_DATA_WIDTH18
```

```
#elif ((_GLCD_MAXX_==800) && (_GLCD_MAXY_==480))
/* 7" 800x480 display */
/* TFT LCD pixel clock frequency [kHz] */
#define SSD1963_DEFAULT_TFT_PIXEL_CLK_FREQ 30000

#define SSD1963_DEFAULT_TFT_HOR_PULSE_WIDTH 1
#define SSD1963_DEFAULT_TFT_HOR_FRONT_PORCH 45
#define SSD1963_DEFAULT_TFT_HOR_BACK_PORCH 210

#define SSD1963_DEFAULT_TFT_VER_PULSE_WIDTH 1
#define SSD1963_DEFAULT_TFT_VER_FRONT_PORCH 10
#define SSD1963_DEFAULT_TFT_VER_BACK_PORCH 34

/* default tft24bit to 24 bit TFT */
#define SSD1963_DEFAULT_TFT_DATA_WIDTH SSD1963_TFT_DATA_WIDTH24
#endif

/* Default value for reverse_x */
/* No horizontal reverse */
#define SSD1963_DEFAULT_REVX SSD1963_REVX_NORM

/* Default value for reverse_y */
/* No vertical reverse */
#define SSD1963_DEFAULT_REVY SSD1963_REVY_NORM

/* Default value for cl_bits_order */
/* Write in RGB order */
#define SSD1963_DEFAULT_CL_BITS SSD1963_CL_BITS_RGB
```

The following colors are predefined in the **glcd_ssd1963.h** header file:

GLCD_CL_BLACK
GLCD_CL_WHITE
GLCD_CL_GRAY
GLCD_CL_LIGHT_GRAY
GLCD_CL_GREEN
GLCD_CL_LIME
GLCD_CL_BLUE
GLCD_CL_RED
GLCD_CL_AQUA
GLCD_CL_YELLOW
GLCD_CL_MAGENTA
GLCD_CL_CYAN
GLCD_CL_DARK_CYAN
GLCD_CL_ORANGE
GLCD_CL_PINK
GLCD_CL_BROWN
GLCD_CL_VIOLET
GLCD_CL_SILVER
GLCD_CL_GOLD
GLCD_CL_NAVY
GLCD_CL_MAROON
GLCD_CL_PURPLE
GLCD_CL_OLIVE

The SSD1963 library functions can operate the display in 256 or 64k color modes.
For 256 color mode the following color bit allocation in a data byte is used:

- Bits 0..2 - Blue color bits 0..2
- Bits 3..5 - Green color bits 0..2
- Bits 6..7 - Red color bits 0..1.

For 64k color mode the following color bit allocation in a 16bit data word is used:

- Bits 0..4 - Blue color bits 0..4
- Bits 5..10 - Green color bits 0..5
- Bits 11..15 - Red color bits 0..4.

**Notes:**

- In order to reduce image storage size and improve speed, it is recommended to use the 256 color mode, if possible.
- The **glcd_ssd1963.h** header file is automatically #included when the main **glcd.h** header file is #included. Therefore there is no need to explicitly #include **glcd_ssd1963.h**.
- The **..\EXAMPLES\Graphic LCD\SSD1963** directory contains fully functional code samples that may be used as references for SSD1963 initialization and usage.

## 5.12.10 Graphic LCD Functions Specific to the SSD2119 Controller

The SSD2119 library functions supplied with the CodeVisionAVR Advanced license, can operate the controller in 8bit and 16 bit interface modes.
In order to take full advantage of the SSD2119 controller's features the following specific functions, declared in the **glcd_ssd2119.h** header file, were implemented:

**void ssd2119_wrcmd(unsigned char cmd)**

Writes a command to the SSD2119 controller, used to access a specific command register.

*Parameter:*

**cmd** command register index to be sent to the controller.

This index may take one of the values defined in the following macros from the **glcd_ssd2119.h** header file:

```
/* SSD2119 command register definitions */
#define SSD2119_CMD_OSC 0x00 /* Oscillator register */
#define SSD2119_CMD_DRIVER_OUT 0x01 /* Driver output control
#define SSD2119_CMD_DRIVING_WAVEFORM 0x02 /* LCD driving waveform
                                            control */
#define SSD2119_CMD_POWER_CONTROL1 0x03 /* Power control 1 */
#define SSD2119_CMD_DISPLAY_CONTROL 0x07 /* Display control register */
#define SSD2119_CMD_FRAME_CYCLE 0x0b /* Frame cycle control register */
#define SSD2119_CMD_POWER_CONTROL2 0x0c /* Power control 2 register */
#define SSD2119_CMD_POWER_CONTROL3 0x0d /* Power control 3 register */
#define SSD2119_CMD_POWER_CONTROL4 0x0e /* Power control 4 register */
#define SSD2119_CMD_GATE_SCAN_POS 0x0f /* Gate scan position register */
#define SSD2119_CMD_SLEEP_MODE 0x10 /* Sleep mode register */
#define SSD2119_CMD_ENTRY_MODE 0x11 /* Entry mode register */
#define SSD2119_CMD_DEEP_SLEEP_MODE 0x12 /* Deep sleep mode register */
#define SSD2119_CMD_GENERIC_IF_CTRL 0x15 /* Generic interface control
                                           register */
#define SSD2119_CMD_HORIZ_PORCH 0x16 /* Horizontal porch register */
#define SSD2119_CMD_VERT_PORCH 0x17 /* Vertical porch register */
#define SSD2119_CMD_POWER_CONTROL5 0x1e /* Power control 5 register */
#define SSD2119_CMD_GDDRAM_DATA 0x22 /* GDDRAM read/write data register */
#define SSD2119_CMD_FRAME_FREQ 0x25 /* Frame frequency control register */
#define SSD2119_CMD_GAMMA_CONTROL1 0x30 /* Gamma control 1 */
#define SSD2119_CMD_GAMMA_CONTROL2 0x31 /* Gamma control 2 */
#define SSD2119_CMD_GAMMA_CONTROL3 0x32 /* Gamma control 3 */
#define SSD2119_CMD_GAMMA_CONTROL4 0x33 /* Gamma control 4 */
#define SSD2119_CMD_GAMMA_CONTROL5 0x34 /* Gamma control 5 */
#define SSD2119_CMD_GAMMA_CONTROL6 0x35 /* Gamma control 6 */
#define SSD2119_CMD_GAMMA_CONTROL7 0x36 /* Gamma control 7 */
#define SSD2119_CMD_GAMMA_CONTROL8 0x37 /* Gamma control 8 */
#define SSD2119_CMD_GAMMA_CONTROL9 0x3a /* Gamma control 9 */
#define SSD2119_CMD_GAMMA_CONTROL10 0x3b /* Gamma control 10 */
#define SSD2119_CMD_VERT_SCROLL1 0x41 /* Vertical scroll control for
                                        screen 1 */
#define SSD2119_CMD_VERT_SCROLL2 0x42 /* Vertical scroll control for
                                        screen 2 */
```

```
#define SSD2119_CMD_HORIZ_RAM_ADDR 0x44 /* Addresses of horizontal
                                           start/end window positions */
#define SSD2119_CMD_VERT_RAM_ADDR_START 0x45 /* Address of vertical start
                                                window positions */
#define SSD2119_CMD_VERT_RAM_ADDR_END 0x46 /* Address of vertical end
                                              window positions */
#define SSD2119_CMD_DRV_POS_START1 0x48 /* Driving start line position for
                                           screen 1 */
#define SSD2119_CMD_DRV_POS_END1 0x49 /* Driving end line position for
                                         screen 1 */
#define SSD2119_CMD_DRV_POS_START2 0x4a /* Driving start line position for
                                           screen 2 */
#define SSD2119_CMD_DRV_POS_END2 0x4b /* Driving end line position for
                                         screen 2 */
#define SSD2119_CMD_GDDRAMX 0x4e /* Set GDDRAM X address counter
                                    register */
#define SSD2119_CMD_GDDRAMY 0x4f /* Set GDDRAM Y address counter
                                    register
```

A detailed description of the above mentioned command registers can be found in the SSD2119 datasheet.

**void ssd2119_wrreg(unsigned char index, unsigned short data)**

Writes data to a command register of the SSD2119 controller.

*Parameters:*

**index** command register index
**data** to be written.

**unsigned short ssd2119_rdreg(unsigned char index)**

Reads the contents of a command register of the SSD2119 controller.

*Parameters:*

**index** command register index.

**void ssd2119_wrdata(unsigned short data)**

Writes data to the SSD2119 controller's Graphic Display RAM.

*Parameters:*

**data** to be written.

**unsigned short ssd2119_rddata(void)**

Reads data from the SSD2119 controller's Graphic Display RAM.

The **glcd_ssd2119.h** header file also contains the definition of the **GLCDINIT_t** type specific for the SSD2119 controller, used as parameter for the **glcd_init** function:

```
typedef struct
        {
        flash unsigned char *font; /* default font after initialization */
        /* pointer to the function used for reading a byte from external
           memory */
        unsigned char (*readxmem) (GLCDMEMADDR_t addr);
        /* pointer to the function used for writing a byte to external
           memory */
        void (*writexmem) (GLCDMEMADDR_t addr, unsigned char data);
        unsigned char reverse_x:1; /* reverse display horizontally */
        unsigned char reverse_y:1; /* reverse display vertically */
        unsigned char first_out_gate:1; /* selects the first output gate
                                           GD bit of Driver Output Control
                                           register
                                           =0 Gate 1 on left side of
                                           display
                                           =1 Gate 2 on left side of
                                           display */
        unsigned char cl_bits_order:1; /* selects the color bits writing
                                          order to the display RAM
                                          =0 -> RGB, =1 ->BGR */
        /* power control registers bits */
        unsigned char stepup_factor:3; /* step-up factor of the step-up
                                          circuit, see BT0..BT2 bits in
                                          the datasheet */
        unsigned char stepup_cycle:4;  /* controls the cycle for the
                                          step-up circuit */
        unsigned char crt_source:3; /* adjusts the amount of current
                                       from the constant current source
                                       in the internal op. amplifier
                                       circuit (AP0..AP2 bits) */
        unsigned char vcix2:3;  /* adjusts the VCIX2 voltage */
        unsigned char vlcd63:4; /* adjusts the VLCD63 voltage */
        unsigned char vcoml:5;  /* adjusts the amplitude of the
                                   VcomL alternating drive voltage */
        unsigned char vcomh:5;  /* adjusts the amplitude of the VcomH
                                   voltage
                                   VcomH=VLCD63*(0.35+vcomh*0.01) [V] */
        unsigned char frame_freq:4; /* LCD frame frequency */
        /* positive gamma control registers bits */
        unsigned char pkp00:3; /* PKP00..PKP02 positive gamma micro adj. */
        unsigned char pkp10:3; /* PKP10..PKP12 positive gamma micro adj. */
        unsigned char pkp20:3; /* PKP20..PKP22 positive gamma micro adj. */
        unsigned char pkp30:3; /* PKP30..PKP32 positive gamma micro adj. */
        unsigned char pkp40:3; /* PKP40..PKP42 positive gamma micro adj. */
        unsigned char pkp50:3; /* PKP50..PKP52 positive gamma micro adj. */
        unsigned char prp00:3; /* PRP00..PRP02 positive gamma gradient
                                  adj. */
        unsigned char prp10:3; /* PRP10..PRP12 positive gamma gradient
                                  adj. */
        unsigned char vrp00:4; /* VRP00..VRP03 positive gamma amplification
                                  adj. */
        unsigned char vrp10:5; /* VRP10..VRP14 positive gamma amplification
                                  adj. */
```

```
/* negative gamma control registers bits */
        unsigned char pkn00:3; /* PKN00..PKN02 negative gamma micro adj. */
        unsigned char pkn10:3; /* PKN10..PKN12 negative gamma micro adj. */
        unsigned char pkn20:3; /* PKN20..PKN22 positive gamma micro adj. */
        unsigned char pkn30:3; /* PKN30..PKN32 positive gamma micro adj. */
        unsigned char pkn40:3; /* PKN40..PKN42 negative gamma micro adj. */
        unsigned char pkn50:3; /* PKN50..PKN52 negative gamma micro adj. */
        unsigned char prn00:3; /* PRN00..PRN02 negative gamma gradient
                                  adj. */
        unsigned char prn10:3; /* PRN10..PRN12 negative gamma gradient
                                  adj. */
        unsigned char vrn00:4; /* VRN00..VRN03 negative gamma amplification
                                  adj. */
        unsigned char vrn10:5; /* VRN10..VRN14 negative gamma
                                  amplification adj. */
        } GLCDINIT_t;
```

The following macros are defined for initializing the members of the **GLCDINIT_t** structure:

```
/* Initialization values for reverse_x */
#define SSD2119_REVX_NORM 0 /* No horizontal reverse */
#define SSD2119_REVX_REV 1 /* Horizontal reverse */

/* Initialization values for reverse_y */
#define SSD2119_REVY_NORM 0 /* No vertical reverse */
#define SSD2119_REVY_REV 1 /* Vertical reverse */

/* Initialization values for first_out_gate */
#define SSD2119_FIRST_OUT_GATE1 0 /* Gate 1 on left side of display */
#define SSD2119_FIRST_OUT_GATE2 1 /* Gate 2 on left side of display */

/* Initialization values for cl_bits_order */
#define SSD2119_CL_BITS_RGB 0 /* Write color bits to display RAM
                                 in RGB order */
#define SSD2119_CL_BITS_BGR 1 /* Write color bits to display RAM
                                 in BGR order */

/* Initilization values for dc30 step-up circuit cycle */
#define SSD2119_STEPUP_FLINE24 0 /* Fline *24 */
#define SSD2119_STEPUP_FLINE16 1 /* Fline *16 */
#define SSD2119_STEPUP_FLINE12 2 /* Fline *12 */
#define SSD2119_STEPUP_FLINE8 3 /* Fline *8 */
#define SSD2119_STEPUP_FLINE6 4 /* Fline *6 */
#define SSD2119_STEPUP_FLINE5 5 /* Fline *5 */
#define SSD2119_STEPUP_FLINE4 6 /* Fline *4 */
#define SSD2119_STEPUP_FLINE3 7 /* Fline *3 */
#define SSD2119_STEPUP_FLINE2 8 /* Fline *2 */
#define SSD2119_STEPUP_FLINE1 9 /* Fline *1 */
#define SSD2119_STEPUP_FOSC4 10 /* Fosc /4  (Fosc=510kHz) */
#define SSD2119_STEPUP_FOSC6 11 /* Fosc /6 */
#define SSD2119_STEPUP_FOSC8 12 /* Fosc /8 */
#define SSD2119_STEPUP_FOSC10 13 /* Fosc /10 */
#define SSD2119_STEPUP_FOSC12 14 /* Fosc /12 */
#define SSD2119_STEPUP_FOSC16 15 /* Fosc /16 */
```

```
/* Initialization values for the VCIX2 voltage
  (Power Control Reg. 2) */
#define SSD2119_VCIX2_5V1 0 /* 5.1V */
#define SSD2119_VCIX2_5V3 1 /* 5.3V */
#define SSD2119_VCIX2_5V5 2 /* 5.5V */
#define SSD2119_VCIX2_5V7 3 /* 5.7V */
#define SSD2119_VCIX2_5V9 4 /* 5.9V */
#define SSD2119_VCIX2_6V1 5 /* 6.1V */

/* Initialization values for the VLCD63 voltage
  (bits VRH0..3 from Power Control Reg. 3) */
#define SSD2119_VLCD63_3V56 3 /* VLCD63=3.56V */
#define SSD2119_VLCD63_3V70 4 /* VLCD63=3.70V */
#define SSD2119_VLCD63_3V86 5 /* VLCD63=3.86V */
#define SSD2119_VLCD63_4V04 6 /* VLCD63=4.04V */
#define SSD2119_VLCD63_4V18 7 /* VLCD63=4.18V */
#define SSD2119_VLCD63_4V33 8 /* VLCD63=4.33V */
#define SSD2119_VLCD63_4V49 9 /* VLCD63=4.49V */
#define SSD2119_VLCD63_4V67 10 /* VLCD63=4.67V */
#define SSD2119_VLCD63_4V80 11 /* VLCD63=4.80V */
#define SSD2119_VLCD63_5V00 12 /* VLCD63=5.00V */
#define SSD2119_VLCD63_5V14 13 /* VLCD63=5.14V */
#define SSD2119_VLCD63_5V29 14 /* VLCD63=5.29V */
#define SSD2119_VLCD63_5V45 15 /* VLCD63=5.45V */
#define SSD2119_VLCD63_5V62 0 /* VLCD63=5.62V */
#define SSD2119_VLCD63_5V80 1 /* VLCD63=5.80V */
#define SSD2119_VLCD63_6V00 2 /* VLCD63=6.00V */

/* Initialization values for the VcomL voltage
   (bits VDV0..4 from Power Control Reg. 4) */
#define SSD2119_VCOML_0_60 0 /* VcomL=VLCD63*0.60 */
#define SSD2119_VCOML_0_63 1 /* VcomL=VLCD63*0.63 */
#define SSD2119_VCOML_0_66 2 /* VcomL=VLCD63*0.66 */
#define SSD2119_VCOML_0_69 3 /* VcomL=VLCD63*0.69 */
#define SSD2119_VCOML_0_72 4 /* VcomL=VLCD63*0.72 */
#define SSD2119_VCOML_0_75 5 /* VcomL=VLCD63*0.75 */
#define SSD2119_VCOML_0_78 6 /* VcomL=VLCD63*0.78 */
#define SSD2119_VCOML_0_81 7 /* VcomL=VLCD63*0.81 */
#define SSD2119_VCOML_0_84 8 /* VcomL=VLCD63*0.84 */
#define SSD2119_VCOML_0_87 9 /* VcomL=VLCD63*0.87 */
#define SSD2119_VCOML_0_90 10 /* VcomL=VLCD63*0.90 */
#define SSD2119_VCOML_0_93 11 /* VcomL=VLCD63*0.93 */
#define SSD2119_VCOML_0_96 12 /* VcomL=VLCD63*0.96 */
#define SSD2119_VCOML_0_99 13 /* VcomL=VLCD63*0.99 */
#define SSD2119_VCOML_1_02 14 /* VcomL=VLCD63*1.02 */
#define SSD2119_VCOML_EXT_RES 15 /* VcomL is set by an external
                                    variable resistor */
#define SSD2119_VCOML_1_05 16 /* VcomL=VLCD63*1.05 */
#define SSD2119_VCOML_1_08 17 /* VcomL=VLCD63*1.08 */
#define SSD2119_VCOML_1_11 18 /* VcomL=VLCD63*1.11 */
#define SSD2119_VCOML_1_14 19 /* VcomL=VLCD63*1.14 */
#define SSD2119_VCOML_1_17 20 /* VcomL=VLCD63*1.17 */
#define SSD2119_VCOML_1_20 21 /* VcomL=VLCD63*1.20 */
#define SSD2119_VCOML_1_23 22 /* VcomL=VLCD63*1.23 */
```

```
/* Initialization values for frame_freq */
#define SSD2119_FRAME50 0 /* 50Hz */
#define SSD2119_FRAME55 2 /* 55Hz */
#define SSD2119_FRAME60 5 /* 60Hz */
#define SSD2119_FRAME65 8 /* 65Hz */
#define SSD2119_FRAME70 0x0A /* 70Hz */
#define SSD2119_FRAME75 0x0C /* 75Hz */
#define SSD2119_FRAME80 0x0E /* 80Hz */

/* Default value for reverse_x */
#define SSD2119_DEFAULT_REVX SSD2119_REVX_NORM /* No horizontal reverse */
/* Default value for reverse_y */
#define SSD2119_DEFAULT_REVY SSD2119_REVY_NORM /* No vertical reverse */

/* Default value for first_out_gate
   Gate 2 on left side of display */
#define SSD2119_DEFAULT_FIRST_OUT_GATE SSD2119_FIRST_OUT_GATE2

/* Default value for cl_bits_order
  (color bits writing order to display RAM),
  write in RGB order */
#define SSD2119_DEFAULT_CL_BITS SSD2119_CL_BITS_RGB

/* Power control 1 BT0..BT2 step-up factor of the step-up circuit */
#define SSD2119_DEFAULT_STEPUP_FACTOR 4
/* Power control 1 DC0..DC3 step-up circuit cycle */
#define SSD2119_DEFAULT_STEPUP_CYCLE SSD2119_STEPUP_FOSC4
/* Power control 1 AP0..AP2 adjusts the amount of current
   from the constant current source in the internal operational
   amplifier circuit */
#define SSD2119_DEFAULT_CRT_SOURCE 2
/* Default value for VCIX2 voltage */
#define SSD2119_DEFAULT_VCIX2 SSD2119_VCIX2_5V1
/* Default value for VLCD63 voltage */
#define SSD2119_DEFAULT_VLCD63 SSD2119_VLCD63_4V80
/* Default value for VcomL alternating drive voltage */
#define SSD2119_DEFAULT_VCOML SSD2119_VCOML_0_72
/* Default value for VcomH=VLCD63*(0.35+0x1A*0.01) */
#define SSD2119_DEFAULT_VCOMH 0x1A
/* Default value for driving waveform control FLD bit,
   splits one frame into 3 fields to reduce flicker */
#define SSD2119_DEFAULT_FLD 1
/* Default value for LCD frame frequency */
#define SSD2119_DEFAULT_FRAME_FREQ SSD2119_FRAME80

/* Default initialization values for the gamma control register bits */
/* PKP00..PKP02 positive gamma micro adj. */
#define SSD2119_DEFAULT_PKP00 7
/* PKP10..PKP12 positive gamma micro adj. */
#define SSD2119_DEFAULT_PKP10 7
/* PKP20..PKP22 positive gamma micro adj. */
#define SSD2119_DEFAULT_PKP20 4
/* PKP30..PKP32 positive gamma micro adj. */
#define SSD2119_DEFAULT_PKP30 2
/* PKP40..PKP42 positive gamma micro adj. */
#define SSD2119_DEFAULT_PKP40 4
/* PKP50..PKP52 positive gamma micro adj. */
#define SSD2119_DEFAULT_PKP50 2
/* PRP00..PRP02 positive gamma gradient adj. */
#define SSD2119_DEFAULT_PRP00 2
```

```
/* PRP10..PRP12 positive gamma gradient adj. */
#define SSD2119_DEFAULT_PRP10 5
/* VRP00..VRP03 positive gamma amplification adj. */
#define SSD2119_DEFAULT_VRP00 2
/* VRP10..VRP14 positive gamma amplification adj. */
#define SSD2119_DEFAULT_VRP10 3
/* PKN00..PKN02 negative gamma micro adj. */
#define SSD2119_DEFAULT_PKN00 7
/* PKN10..PKN12 negative gamma micro adj. */
#define SSD2119_DEFAULT_PKN10 5
/* PKN20..PKN22 positive gamma micro adj. */
#define SSD2119_DEFAULT_PKN20 4
/* PKN30..PKN32 positive gamma micro adj. */
#define SSD2119_DEFAULT_PKN30 2
/* PKN40..PKN42 negative gamma micro adj. */
#define SSD2119_DEFAULT_PKN40 4
/* PKN50..PKN52 negative gamma micro adj. */
#define SSD2119_DEFAULT_PKN50 2
/* PRN00..PRN02 negative gamma gradient adj. */
#define SSD2119_DEFAULT_PRN00 2
/* PRN10..PRN12 negative gamma gradient adj. */
#define SSD2119_DEFAULT_PRN10 5
/* VRN00..VRN03 negative gamma amplification adj. */
#define SSD2119_DEFAULT_VRN00 2
/* VRN10..VRN14 negative gamma amplification adj. */
#define SSD2119_DEFAULT_VRN10 3
```

The following colors are predefined in the **glcd_ssd2119.h** header file:

GLCD_CL_BLACK
GLCD_CL_WHITE
GLCD_CL_GRAY
GLCD_CL_LIGHT_GRAY
GLCD_CL_GREEN
GLCD_CL_LIME
GLCD_CL_BLUE
GLCD_CL_RED
GLCD_CL_AQUA
GLCD_CL_YELLOW
GLCD_CL_MAGENTA
GLCD_CL_CYAN
GLCD_CL_DARK_CYAN
GLCD_CL_ORANGE
GLCD_CL_PINK
GLCD_CL_BROWN
GLCD_CL_VIOLET
GLCD_CL_SILVER
GLCD_CL_GOLD
GLCD_CL_NAVY
GLCD_CL_MAROON
GLCD_CL_PURPLE
GLCD_CL_OLIVE

The SSD2119 library functions can operate the display in 256 or 64k color modes.
For 256 color mode the following color bit allocation in a data byte is used:

- Bits 0..2 - Blue color bits 0..2
- Bits 3..5 - Green color bits 0..2
- Bits 6..7 - Red color bits 0..1.

For 64k color mode the following color bit allocation in a 16bit data word is used:

- Bits 0..4 - Blue color bits 0..4
- Bits 5..10 - Green color bits 0..5
- Bits 11..15 - Red color bits 0..4.

**Notes:**

- In order to reduce image storage size and improve speed, it is recommended to use the 256 color mode, if possible.
- The **glcd_ssd2119.h** header file is automatically #included when the main **glcd.h** header file is #included. Therefore there is no need to explicitly #include **glcd_ssd2119.h**.
- The **..\EXAMPLES\Graphic LCD\SSD2119** directory contains fully functional code samples that may be used as references for SSD2119 initialization and usage.

## 5.12.11 Graphic LCD Functions Specific to the ST7565 Controller

In order to take full advantage of the ST7565 controller's features the following specific functions, declared in the **glcd_st7565.h** header file, were implemented:

**void st7565_wrcmd(unsigned char cmd)**

Writes a command to the ST7565 controller.

*Parameter:*

**cmd** command to be sent to the controller.

This command may take one of the values defined in the following macros from the **glcd_st7565.h** header file:

```
#define ST7565_CMD_START_LINE 0x40 /* set display start line */
#define ST7565_CMD_SET_PAGE 0xB0 /* set display page address */
#define ST7565_CMD_SET_ADDR_LOW 0x00 /* set column address bits 0..3 */
#define ST7565_CMD_SET_ADDR_HIGH 0x10 /* set column address bits 4..7 */
#define ST7565_CMD_ADC_SELECT_NORM 0xA0 /* set relationship between RAM
                                          column address and display
                                          driver: normal */
#define ST7565_CMD_ADC_SELECT_REV 0xA1  /* set relationship between RAM
                                          column address and display
                                          driver: reversed */
#define ST7565_CMD_DISP_NORMAL 0xA6 /* set normal display mode */
#define ST7565_CMD_DISP_REVERSE 0xA7 /* set reversed display mode */
#define ST7565_CMD_PIXELS_NORMAL 0xA4 /* display the graphic RAM
                                        contents */
#define ST7565_CMD_ALL_PIXELS_ON 0xA5 /* all display pixels are on */
#define ST7565_CMD_STATIC_INDICATOR_OFF 0xAC /* turn off static
                                               indicator */
#define ST7565_CMD_STATIC_INDICATOR_ON 0xAD /* turn on static indicator */
#define ST7565_CMD_DISP_OFF 0xAE /* display off */
#define ST7565_CMD_DISP_ON 0xAF /* display on */
#define ST7565_CMD_LCD_BIAS_LOW 0xA2  /* sets voltage ratio for LCD bias to
                                        1/9 (duty cycle=1/65),
                                        1/8 (duty cycle=1/49),
                                        1/6 (duty cycle=1/33),
                                        1/8 (duty cycle=1/55),
                                        1/8 (duty cycle=1/53) */
#define ST7565_CMD_LCD_BIAS_HIGH 0xA3 /* sets voltage ratio for LCD bias to
                                        1/7 (duty cycle=1/65),
                                        1/6 (duty cycle=1/49),
                                        1/5 (duty cycle=1/33),
                                        1/6 (duty cycle=1/55),
                                        1/6 (duty cycle=1/53) */
#define ST7565_CMD_COM0_63 0xC0 /* sets the COM output scan
                                  direction 0->63 */
#define ST7565_CMD_COM63_0 0xC8 /* sets the COM output scan
                                  direction 63->0 */
```

```
#define ST7565_CMD_POWER_CTRL 0x28 /* turns on/off the
                                      voltage follower (| bit 0),
                                      voltage regulator (| bit 1),
                                      voltage booster (| bit 2) */
#define ST7565_VOLT_FOLLOWER_ON (1<<0)  /* enable voltage follower */
#define ST7565_VOLT_REGULATOR_ON (1<<1) /* enable voltage regulator */
#define ST7565_VOLT_BOOSTER_ON (1<<2)   /* enable voltage booster */

#define ST7565_CMD_VOLT_REG_V5 0x20 /* sets the V5 voltage regulator
                                       internal resistor ratio */
#define ST7565_CMD_ELECTRONIC_VOLUME 0x81 /* sets the electronic volume
                                             register in order to control
                                             the V5 LCD drive voltage */
#define ST7565_CMD_SET_DRIVING_MODE 0xD2 /* used to set the LCD driving
                                            mode */
#define ST7565_CMD_RESET 0xE2 /* resets the controller */
```

A detailed description of the above mentioned commands can be found in the ST7565 datasheet.

**void st7565_setcontrast(unsigned char contrast)**

Controls the LCD contrast.

*Parameter:*

**contrast** sets the value of the V5 LCD drive voltage, allowed range is 0..63.

The **glcd_st7565.h** header file also contains the definition of the **GLCDINIT_t** type specific for the ST7565 controller, used as parameter for the **glcd_init** function:

```
typedef struct
        {
        flash unsigned char *font; /* default font after initialization */
        /* pointer to the function used for reading a byte
           from external memory */
        unsigned char (*readxmem) (GLCDMEMADDR_t addr);
        /* pointer to the function used for writing a byte
           to external memory */
        void (*writexmem) (GLCDMEMADDR_t addr, unsigned char data);

        unsigned char lcd_bias:1; /* =0 LCD bias ratio low,
                                     =1 LCD bias ratio high */
        unsigned char reverse_x:1; /* reverse display horizontally (ADC) */
        unsigned char rev132_x0:1; /* set to 1 for displays that use
                                      reversed RAM column address
                                      (reverse_x=1) driver and the pixel
                                      with x=0 is connected to column
                                      driver #132 */
        unsigned char reverse_y:1; /* reverse display vertically (COM) */
        unsigned char volt_reg_v5:3; /* set V5 voltage regulator
                                        internal resistor ratio [0..7] */
        unsigned char driving_mode:2; /* set LCD driving mode:
                                         0 - mode 1, 1 - mode 2,
                                         2 - mode 3, 3 - mode 4 */
        unsigned char lcd_contrast:5; /* LCD contrast voltage: [0..63] */
        } GLCDINIT_t;
```

The following macros are defined for initializing the members of the **GLCDINIT_t** structure:

```
/* values used for lcd_bias initialization */
#define ST7565_LCD_BIAS_19 0 /* sets LCD bias drive ratio 1/9, 1/8, 1/7 */
#define ST7565_LCD_BIAS_17 1 /* sets LCD bias drive ratio 1/7, 1/6, 1/5 */

/* values used for reverse_x initialization */
#define ST7565_REVX_NORM 0 /* set relationship between RAM column address
                              and display driver: normal (ADC=0) */
#define ST7565_REVX_REV 1  /* set relationship between RAM column address
                              and display driver: reversed (ADC=1) */

/* values used for rev132_x0 initilization,
   effective only when reverse_x=1 (ST7565_REVX_REV) */
#define ST7565_REV132_X0NC 0  /* pixel with x=0 is not connected to column
driver #132 when ADC=1 */
#define ST7565_REV132_X0CON 1 /* pixel with x=0 is connected to column
driver #132 when ADC=1 */

/* values used for reverse_y initialization */
#define ST7565_REVY_NORM 0 /* sets the vertical COM output scan
                              direction 0->63 */
#define ST7565_REVY_REV 1  /* sets the vertical COM output scan
                              direction 63->0 */

/* values used for driving_mode initialization */
#define ST7565_DRIVING_MODE1 0 /* driving mode 1 */
#define ST7565_DRIVING_MODE2 1 /* driving mode 2 */
#define ST7565_DRIVING_MODE3 2 /* driving mode 3 */
#define ST7565_DRIVING_MODE4 3 /* driving mode 4 */

/* default initialization values */
/* default value for LCD bias */
#define ST7565_DEFAULT_LCD_BIAS ST7565_LCD_BIAS_LOW
/* default value for reverse_x */
#define ST7565_DEFAULT_REVX ST7565_REVX_NORM
/* default value for rev132_x0,
   effective only when reverse_x=1 (ST7565_REVX_REV) */
#define ST7565_DEFAULT_REV132_X0 ST7565_REV132_X0NC
/* default value for reverse_y */
#define ST7565_DEFAULT_REVY ST7565_REVY_NORM
/* default V5 voltage regulator internal resistor ratio */
#define ST7565_DEFAULT_VOLT_REG_V5 6
/* default LCD driving mode */
#define ST7565_DEFAULT_DRIVING_MODE ST7565_DRIVING_MODE1
/* default contrast */
#define ST7565_DEFAULT_CONTRAST 7
```

The detailed description of the above mentioned initialization parameters can be found in the ST7565 datasheet.

**Notes:**

- The **glcd_st7565.h** header file is automatically #included when the main **glcd.h** header file is #included. Therefore there is no need to explicitly #include **glcd_st7565.h**.
- The **..\EXAMPLES\Graphic LCD\ST7565** directory contains fully functional code samples that may be used as references for ST7565 initialization and usage.

## 5.12.12 Graphic LCD Functions Specific to the ST7920 Controller

In order to take full advantage of the ST7920 controller's features the following specific functions, declared in the **glcd_st7920.h** header file, were implemented:

**void st7920_wrcmd(unsigned char cmd)**

Writes a command to the ST7920 controller.

*Parameter:*

**cmd** command to be sent to the controller.

This command may take one of the values defined in the following macros from the **glcd_st7920.h** header file:

```
/* Basic command set */
/* clear display */
#define ST7920_DISPLAY_CLEAR 0x01

/* return cursor to origin */
#define ST7920_RETURN_HOME 0x02

/* set cursor position & display shift when doing write or read */
#define ST7920_ENTRY_MODE_SET 0x04
/* entry mode bits: */
/* cursor moves right, address counter+=1 */
#define ST7920_ENTRY_CURSOR_RIGHT 0x02
/* shift entire display */
#define ST7920_ENTRY_SHIFT_DISPL 0x01

/* display control */
#define ST7920_DISPLAY_CTRL 0x08
/* display control bits: */
/* display on */
#define ST7920_DISPLAY_ON 0x04
/* cursor on */
#define ST7920_CURSOR_ON 0x02
/* character blinks at cursor position */
#define ST7920_BLINK 0x01

/* cursor/display shift control */
#define ST7920_CURSOR_DISPLAY_SHIFT_CTRL 0x10
/* cursor/display shift control bits: */
/* shift display */
#define ST7920_SHIFT_DISPLAY 0x08
/* shifts right */
#define ST7920_SHIFT_RIGHT 0x04
```

```
/* function set */
#define ST7920_FUNC_SET 0x20
/* function set control bits:
   must not be applied at the same time!
   change ST7920_8BIT or ST7920_GRAPHICS_ON first, then ST7920_EXTENDED */
/* 8 bit interface */
#define ST7920_8BIT 0x10
/* select the extended command set */
#define ST7920_EXTENDED 0x04
/* turn graphics display on */
#define ST7920_GRAPHICS_ON 0x02

/* writes CGRAM address into address counter AC */
#define ST7920_SET_CGRAM_ADDR 0x40

/* writes DDRAM address into address counter AC:
   start for line #1 of text AC=0x00
   start for line #2 of text AC=0x10
   start for line #3 of text AC=horizontal_display_resolution/16
   start for line #4 of text AC=0x10+horizontal_display_resolution/16 */
#define ST7920_SET_DDRAM_ADDR 0x80

/* Extended command set */
#define ST7920_STANDBY 0x01 /* enter standby mode */

/* enable the ST7920_SET_CGRAM_ADDR command */
#define ST7920_ENABLE_SET_CGRAM_ADDR 0x02

/* enable vertical scroll mode and disable the
   ST7920_SET_CGRAM_ADDR command */
#define ST7920_VERT_SCROLL 0x03

/* toggle reverse condition for a line of text */
#define ST7920_REVERSE_LINE 0x04

/* set vertical scroll displacement address */
#define ST7920_SET_SCROLL_ADDR 0x40

/* set the vertical and horizontal addresses
   for the graphic display RAM into address counter */
#define ST7920_SET_GDRAM_ADDR 0x80
```

A detailed description of the above mentioned commands can be found in the ST7920 datasheet.

**void st7920_wrdata(unsigned char data)**

Writes a data byte to the ST7920 controller.

*Parameter:*

**data** byte to be sent to the controller.

**unsigned char st7920_rddata(void)**

Reads a data byte from the ST7920 controller.

**void glcd_cleartext(void)**

Clears the text overlay area when the character generator is used and sets the text display position at row 0 and column 0.

**void glcd_cleargraphics(void)**

Clears the LCD graphics overlay area by setting it's color to the current background color.

The ST7920 controller has a built-in alphanumeric character font generator which can be used for displaying text over the graphics.
In order to take advantage of this capability, the following high level functions, compatible with the alphanumeric LCDs, were implemented:

**void lcd_clear(void)**

Clears the text on the LCD and sets the text display position at row 0 and column 0.
**Note:** This function is equivalent to **glcd_cleartext** and was defined for compatibility.

**void lcd_gotoxy(unsigned char x, unsigned char y)**

Sets the current text display position at column x and row y. The row and column numbering starts from 0.

**void lcd_putchar(char c)**

Displays the character c at the current display position.

For displaying ASCII set characters, c may take the values: 0x02..0x7F. These characters are displayed in a 8x16 pixel matrix.
For displaying a Chinese BIG5 character the function must be called twice, first with the MSB of the character code, then with the LSB.
Example:

```
/* display the Chinese BIG5 character with code: 0xA140 */
lcd_putchar(0xA1);
lcd_putchar(0x40);
```

The Chinese BIG5 characters may take the values: 0xA140..0xD75F.

For displaying an user defined character the **lcd_putchar** function must be also called twice, first with the parameter 0, then with the character code which may be 0, 2, 4 or 6.
Example:

```
lcd_putchar(0);
lcd_putchar(4); /* display the user defined character with code: 4 */
```

**Note:** Chinese BIG 5 and user defined characters use a 16x16 pixel matrix, therefore may be displayed only on even horizontal text x coordinates.

**void glcd_definechar(unsigned char c,flash unsigned char *data)**

Defines a character in the LCD controller's character generator RAM.

*Parameters:*

**c** specifies the defined character's code, must be 0, 2, 4 or 6 for the ST7920 controller
**data** points to a 32 byte array located in FLASH that contains the character's definition.

The data in the array is organized as 16 horizontal rows, each row containing 2 bytes of pixels:
byte 0, bit 0 -> pixel 7, on row 0
byte 0, bit 1 -> pixel 6, on row 0
..
byte 0, bit 7 -> pixel 0, on row 0
byte 1, bit 0 -> pixel 15, on row 0
byte 1, bit 1 -> pixel 14, on row 0
..
byte 1, bit 7 -> pixel 8, on row 0

byte 2, bit 0 -> pixel 7, on row 1
byte 2, bit 1 -> pixel 6, on row 1
..
byte 2, bit 7 -> pixel 0, on row 1
byte 3, bit 0 -> pixel 15, on row 1
byte 3, bit 1 -> pixel 14, on row 1
..
byte 3, bit 7 -> pixel 8, on row 1
..

Example:

```
#include <glcd.h>
#include <delay.h>

/* User defined characters */
flash char user_def_char[4][32]=
{
/* Character with code: 0 */
{
0b00000000,0b11111111,
0b00000000,0b00000011,
0b00000000,0b00000101,
0b00000000,0b00001001,
0b00000000,0b00010001,
0b00000000,0b00100001,
0b00000000,0b01000001,
0b00000000,0b10000001,
0b00000001,0b00000000,
0b00000010,0b00000000,
0b00000100,0b00000000,
0b00001000,0b00000000,
0b00010000,0b00000000,
0b00100000,0b00000000,
0b01000000,0b00000000,
0b10000000,0b00000000
},
```

```
/* Character with code: 2 */
{
0b00000000,0b00000001,
0b00000000,0b00000010,
0b00000000,0b00000100,
0b00000000,0b00001000,
0b00000000,0b00010000,
0b00000000,0b00100000,
0b00000000,0b01000000,
0b00000000,0b10000000,
0b10000001,0b00000000,
0b10000010,0b00000000,
0b10000100,0b00000000,
0b10001000,0b00000000,
0b10010000,0b00000000,
0b10100000,0b00000000,
0b11000000,0b00000000,
0b11111111,0b00000000
},

/* Character with code: 4 */
{
0b11111111,0b11111111,
0b10000000,0b00000011,
0b10000000,0b00000101,
0b10000000,0b00001001,
0b10000000,0b00010001,
0b10000000,0b00100001,
0b10000000,0b01000001,
0b10000000,0b10000001,
0b10000001,0b00000001,
0b10000010,0b00000001,
0b10000100,0b00000001,
0b10001000,0b00000001,
0b10010000,0b00000001,
0b10100000,0b00000001,
0b11000000,0b00000001,
0b11111111,0b11111111
},

/* Character with code: 6 */
{
0b11111111,0b11111111,
0b11000000,0b00000001,
0b10100000,0b00000001,
0b10010000,0b00000001,
0b10001000,0b00000001,
0b10000100,0b00000001,
0b10000010,0b00000001,
0b10000001,0b00000001,
0b10000000,0b10000001,
0b10000000,0b01000001,
0b10000000,0b00100001,
0b10000000,0b00010001,
0b10000000,0b00001001,
0b10000000,0b00000101,
0b10000000,0b00000011,
0b11111111,0b11111111
}
};
```

```
void main()
{
unsigned char c;
GLCDINIT_t init;

/* No font is used */
init.font=NULL;

/* No need for reading data from external memory */
init.readxmem=NULL;

/* No need for reading data from external memory */
init.writexmem=NULL;

/* Initialize the LCD controller and graphics */
glcd_init(&init);

/* Display text using the built-in character generator */
lcd_putsf("Sitronix ST7920\nCodeVisionAVR\nDemo");

/* 2 seconds delay */
delay_ms(2000);
lcd_clear();

/* Define 4 characters with codes: 0, 2, 4, 6 */
for (c=0; c<=3; c++)
    glcd_definechar(c*2,user_def_char[c]);

lcd_putsf("User defined\n"
          "characters:\n");

/* User defined characters must be located on even x coordinates */
/* Display user defined characters with codes: 0, 2, 4, 6 */
for (c=0; c<=6; c+=2)
    {
    lcd_putchar(0); /* Always first write MSB=0 */
    lcd_putchar(c); /* Write LSB=character code */
    }
}
```

**void lcd_puts(char *str)**

Displays at the current display position the string str, located in RAM.

**void lcd_putsf(char flash *str)**

Displays at the current display position the string str, located in FLASH.

**void lcd_putse(char eeprom *str)**

Displays at the current display position the string str, located in EEPROM.

**Notes:**

- The **glcd_st7920.h** header file is automatically #included when the main **glcd.h** header file is #included. Therefore there is no need to explicitly #include **glcd_st7920.h**.
- The **..\EXAMPLES\Graphic LCD\ST7920** directory contains fully functional code samples that may be used as references for ST7920 initialization and usage.

## 5.12.13 Graphic LCD Functions Specific to the T6963C Controller

In order to take full advantage of the T6963C controller's features the following specific functions, declared in the **glcd_t6963.h** header file, were implemented:

**void t6963_busy(void)**

Waits for the T6963 controller to become ready for reading or writing data by polling the STA0 and STA1 flags. Also sets C /D=1.

**void t6963_wrcmd(unsigned char cmd)**

Writes a command to the T6963C controller.

*Parameter:*

**cmd** command to be sent to the controller.

This command may take one of the values defined in the following macros from the **glcd_t6963.h** header file:

```
#define T6963_SET_CURSOR_PTR 0x21 /* Set cursor pointer */
#define T6963_SET_OFFS_REG 0x22 /* Set offset register */
#define T6963_SET_ADDR_PTR 0x24 /* Set address pointer */

#define T6963_SET_TXT_HOME_ADDR 0x40 /* Set text RAM starting address */
#define T6963_SET_TXT_AREA 0x41 /* Set the number of text columns */
#define T6963_SET_GFX_HOME_ADDR 0x42 /* Set graphics RAM starting
                                        address */
#define T6963_SET_GFX_AREA 0x43 /* Set the number of columns for
                                   graphic mode */

#define T6963_SET_MODE_OR 0x80
#define T6963_SET_MODE_XOR 0x81
#define T6963_SET_MODE_AND 0x83
#define T6963_SET_MODE_TXT_ATTR 0x84 /* Set text attributes */
#define T6963_SET_MODE_INTCG 0x80 /* Use internal character generator */
#define T6963_SET_MODE_EXTCG 0x88 /* Use external character generator */

#define T6963_DISPLAY_OFF 0x90 /* Display off */
#define T6963_CURSORON_BLINKOFF 0x92 /* Cusror on, blink off */
#define T6963_CURSORON_BLINKON 0x93 /* Cusror on, blink on */
#define T6963_TXTON_GFXOFF 0x94 /* Text on, graphics off */
#define T6963_TXTOFF_GFXON 0x98 /* Text off, graphics on */
#define T6963_TXTON_GFXON 0x9C /* Text on, graphics on */

#define T6963_CURSOR_1LINE 0xA0 /* Selects 1 line cursor */
#define T6963_CURSOR_2LINE 0xA1 /* Selects 2 lines cursor */
#define T6963_CURSOR_3LINE 0xA2 /* Selects 3 lines cursor */
#define T6963_CURSOR_4LINE 0xA3 /* Selects 4 lines cursor */
#define T6963_CURSOR_5LINE 0xA4 /* Selects 5 lines cursor */
#define T6963_CURSOR_6LINE 0xA5 /* Selects 6 lines cursor */
#define T6963_CURSOR_7LINE 0xA6 /* Selects 7 lines cursor */
#define T6963_CURSOR_8LINE 0xA7 /* Selects 8 lines cursor */

#define T6963_DATA_AUTO_WR 0xB0 /* Data auto write */
#define T6963_DATA_AUTO_RD 0xB1 /* Data auto read */
#define T6963_AUTO_RESET 0xB2 /* Use to exit from auto mode */
```

```
#define T6963_DATA_WR_INC 0xC0 /* Write data and increment ADP */
#define T6963_DATA_RD_INC 0xC1 /* Read data and increment ADP */
#define T6963_DATA_WR 0xC4 /* Write data without modifying ADP */
#define T6963_DATA_RD 0xC5 /* Read data without modifying ADP */

#define T6963_SCREEN_PEEK 0xE0 /* Transfers 1 byte of displayed data
                                  to the data stack */
#define T6963_SCREEN_COPY 0xE8 /* Copies a single raster line of data
                                  to the graphic area */
```

A detailed description of the above mentioned commands can be found in the T6963C datasheet.

**void t6963_wrdata(unsigned char data)**

Writes a data byte to the T6963C controller.

*Parameter:*

**data** byte to be sent to the controller.

**unsigned char t6963_rddata(void)**

Reads a data byte from the T6963C controller.

**void glcd_cleartext(void)**

Clears the text overlay area when the character generator is used and sets the text display position at row 0 and column 0.

**void glcd_cleargraphics(void)**

Clears the LCD graphics overlay area by setting it's color to the current background color.

**void glcd_definechar(unsigned char c,flash unsigned char *data)**

Defines a character in the LCD controller's character generator external RAM.

*Parameters:*

**c** specifies the defined character's code, must be >= 0x80
**data** points to a byte array that contains the character's definition.
The array dimension depends on the height of the character generator font and is 8.

**Notes:**
- The **glcd_t6963.h** header file is automatically #included when the main **glcd.h** header file is #included. Therefore there is no need to explicitly #include **glcd_t6963.h**.
- The **..\EXAMPLES\Graphic LCD\T6963C** directory contains fully functional code samples that may be used as references for T6963C initialization and usage.

## 5.12.14 Graphic LCD Functions Specific to the UC1701 Controller

In order to take full advantage of the UC1701 controller's features the following specific functions, declared in the **glcd_uc1701.h** header file, were implemented:

**void uc1701_wrcmd(unsigned char cmd)**

Writes a command to the UC1701 controller.

*Parameter:*

**cmd** command to be sent to the controller.

This command may take one of the values defined in the following macros from the **glcd_uc1701.h** header file:

```
#define UC1701_CMD_SET_SCROLL_LINE 0x40 /* set the scroll line number */
#define UC1701_CMD_SET_PAGE 0xB0 /* set display page address */
#define UC1701_CMD_SET_ADDR_LOW 0x00 /* set column address bits 0..3 */
#define UC1701_CMD_SET_ADDR_HIGH 0x10 /* set column address bits 4..7 */
#define UC1701_CMD_SEG_DIR_NORM 0xA0 /* set relationship between RAM column
                                        address and display driver:
                                        normal (MX=0) */
#define UC1701_CMD_SEG_DIR_REV 0xA1 /* set relationship between RAM column
                                       address and display driver:
                                       reversed (MX=1) */
#define UC1701_CMD_DISP_NORMAL 0xA6 /* set normal display mode */
#define UC1701_CMD_DISP_REVERSE 0xA7 /* set reversed display mode */
#define UC1701_CMD_PIXELS_NORMAL 0xA4 /* display the graphic RAM
                                         contents */
#define UC1701_CMD_ALL_PIXELS_ON 0xA5 /* all display pixels are on */
#define UC1701_CMD_STATIC_INDICATOR_OFF 0xAC /* turn off static
                                                indicator */
#define UC1701_CMD_STATIC_INDICATOR_ON 0xAD /* turn on static indicator */
#define UC1701_CMD_DISP_OFF 0xAE /* turn display off */
#define UC1701_CMD_DISP_ON 0xAF /* turn display on */
#define UC1701_CMD_LCD_BIAS_LOW 0xA2  /* sets voltage ratio for LCD bias to
                                         1/9 (duty cycle=1/65),
                                         1/8 (duty cycle=1/49),
                                         1/6 (duty cycle=1/33),
                                         1/8 (duty cycle=1/55),
                                         1/8 (duty cycle=1/53) */
#define UC1701_CMD_LCD_BIAS_HIGH 0xA3 /* sets voltage ratio for LCD bias to
                                         1/7 (duty cycle=1/65),
                                         1/6 (duty cycle=1/49),
                                         1/5 (duty cycle=1/33),
                                         1/6 (duty cycle=1/55),
                                         1/6 (duty cycle=1/53) */
#define UC1701_CMD_COM0_63 0xC0 /* sets the COM output scan
                                   direction 0->63 (MY=0) */
#define UC1701_CMD_COM63_0 0xC8 /* sets the COM output scan
                                   direction 63->0 (MY=1) */
#define UC1701_CMD_POWER_CTRL 0x28 /* turns on/off the voltage follower
                                      (| bit 0),
                                      voltage regulator (| bit 1),
                                      voltage booster (| bit 2) */
#define UC1701_VOLT_FOLLOWER_ON (1<<0)  /* enable voltage follower */
#define UC1701_VOLT_REGULATOR_ON (1<<1) /* enable voltage regulator */
#define UC1701_VOLT_BOOSTER_ON (1<<2)   /* enable voltage booster */
```

```
#define UC1701_CMD_VLCD_RES 0x20 /* sets the VLCD voltage regulator
                                    internal resistor ratio */
#define UC1701_CMD_ELECTRONIC_VOLUME 0x81 /* sets the electronic
                                             volume register in order
                                             to control the VLCD
                                             drive voltage */

#define UC1701_CMD_ADV_PGM_CTRL0 0xFA /* advanced program control 0,
                                         used to set LCD temperature
                                         compensation coefficient (TC) */

#define UC1701_CMD_RESET 0xE2 /* resets the controller */

/* set cursor update mode when writing to display RAM,
   reading from display RAM doesn't affect the column address
   register CA */
#define UC1701_CMD_CURSOR_UPDATE_ON 0xE0 /* the column address CA will be
                                            incremented when writing to
                                            display RAM */
#define UC1701_CMD_CURSOR_UPDATE_OFF 0xEE /* the column address CA will
                                             not be incremented when
                                             writing to display RAM */
```

A detailed description of the above mentioned commands can be found in the UC1701 datasheet.

**void uc1701_setcontrast(unsigned char contrast)**

Controls the LCD contrast.

*Parameter:*

**contrast** sets the value of the VLCD drive voltage, allowed range is 0..63.

The **glcd_uc1701.h** header file also contains the definition of the **GLCDINIT_t** type specific for the UC1701 controller, used as parameter for the **glcd_init** function:

```
typedef struct
        {
        flash unsigned char *font; /* default font after initialization */
        /* pointer to the function used for reading a byte
           from external memory */
        unsigned char (*readxmem) (GLCDMEMADDR_t addr);
        /* pointer to the function used for writing a byte
           to external memory */
        void (*writexmem) (GLCDMEMADDR_t addr, unsigned char data);
        unsigned char lcd_bias:1; /* =0 LCD bias voltage ratio low,
                                     =1 LCD bias voltage ratio high */
        unsigned char reverse_x:1; /* reverse display horizontally (MX) */
        unsigned char rev132_x0:1; /* set to 1 for displays that use
                                      reversed RAM column address
                                      (reverse_x=1) driver and the pixel
                                      with x=0 is connected to column
                                      driver #132 */
        unsigned char reverse_y:1; /* reverse display vertically (MY) */
```

```
        unsigned char volt_reg_vlcd:3; /* set VLCD voltage regulator
                                          internal resistor ratio [0..7] */
        unsigned char lcd_contrast:5; /* LCD contrast voltage: [0..63] */
        unsigned char lcd_temp_comp:1; /* LCD temperature compensation
                                          coefficient (TC)
                                          (=0 -0.05%/C, =1 -0.11%/C) */
        } GLCDINIT_t;
```

The following macros are defined for initializing the members of the **GLCDINIT_t** structure:

```
/* values used for lcd_bias initialization */
#define UC1701_LCD_BIAS_19 0 /* sets LCD bias drive ratio 1/9, 1/8, 1/7 */
#define UC1701_LCD_BIAS_17 1 /* sets LCD bias drive ratio 1/7, 1/6, 1/5 */

/* values used for reverse_x initialization */
#define UC1701_REVX_NORM 0 /* set relationship between RAM column address
                              and display driver: normal (MX=0) */
#define UC1701_REVX_REV 1  /* set relationship between RAM column address
                              and display driver: reversed (MX=1) */

/* values used for rev132_x0 initilization,
   effective only when reverse_x=1 (UC1701_REVX_REV) */
#define UC1701_REV132_X0NC 0  /* pixel with x=0 is not connected to column
                                 driver #132 when MX=1 */
#define UC1701_REV132_X0CON 1 /* pixel with x=0 is connected to column
                                 driver #132 when MX=1 */

/* values used for reverse_y initialization */
/* sets the vertical output scan direction 0->63 */
#define UC1701_REVY_NORM 0
/* sets the vertical output scan direction 63->0 */
#define UC1701_REVY_REV 1

/* values used for lcd_temp_comp initialization */
#define UC1701_LCD_TEMP_COMP_005 0 /* -0.05%/C */
#define UC1701_LCD_TEMP_COMP_011 1 /* -0.11%/C */

/* default initialization values */
/* default value for LCD bias */
#define UC1701_DEFAULT_LCD_BIAS UC1701_LCD_BIAS_LOW
/* default value for reverse_x */
#define UC1701_DEFAULT_REVX UC1701_REVX_NORM
/* default value for rev132_x0,
   effective only when reverse_x=1 (UC1701_REVX_REV) */
#define UC1701_DEFAULT_REV132_X0 UC1701_REV132_X0NC
/* default value for reverse_y */
#define UC1701_DEFAULT_REVY UC1701_REVY_NORM
/* default VLCD voltage regulator internal resistor ratio */
#define UC1701_DEFAULT_VLCD_RES 6
/* default contrast */
#define UC1701_DEFAULT_CONTRAST 7
/* default LCD temperature compensation coefficient */
#define UC1701_DEFAULT_LCD_TEMP_COMP UC1701_LCD_TEMP_COMP_011
```

The detailed description of the above mentioned initialization parameters can be found in the UC1701 datasheet.

**Notes:**

- The **glcd_uc1701.h** header file is automatically #included when the main **glcd.h** header file is #included. Therefore there is no need to explicitly #include **glcd_uc1701.h**.

## 5.13 1 Wire Protocol Functions

The 1 Wire Functions are intended for easy interfacing between C programs and various peripherals using the Maxim/Dallas Semiconductor 1 Wire protocol.
These functions treat the microcontroller as a bus master and the peripherals as slaves.
The prototypes for these functions are placed in the file **1wire.h**, located in the .\INC subdirectory. This file must be **#include -d** before using the functions.

The 1 Wire functions must be configured, by specifying the I/O port and bit used for communication through the 1 Wire protocol.
This is accomplished in the **Project|Configure|C Compiler|Libraries|1 Wire** menu:

- the **Enable 1 Wire Bus Interface Support** option must be activated
- the **I/O Port** and **Bit** must be specified in **Data Connection.**

**Note:** For compatibility with projects developed with CodeVisionAVR prior to V2.04.7, the 1 Wire functions can also be configured as outlined in the example below.
However in this case, the **Enable 1 Wire Bus Interface Support** option must be disabled in the **Project|Configure|C Compiler|Libraries|1 Wire** menu.

Example:

```
/* the 1 Wire bus is connected to ATmega8515 PORTB
   the data signal is bit 2 */
#asm
    .equ __w1_port=0x18
    .equ __w1_bit=2
#endasm

/* now you can include the 1 Wire Functions */
#include <1wire.h>
```

<u>This method is not recommended for new projects and it also does not support the XMEGA chips.</u>

Because the 1 Wire Functions require precision time delays for correct operation, the interrupts must be disabled during their execution.
Also it is very important to specify the correct AVR chip **Clock** frequency in the **Project|Configure|C Compiler|Code Generation** menu.

The 1 Wire Functions are:

**unsigned char w1_init(void)**

this function initializes the 1 Wire devices on the bus.
It returns 1 if there were devices present or 0 if not.

**unsigned char w1_read(void)**

this function reads a byte from the 1 Wire bus.

**unsigned char w1_write(unsigned char data)**

this function writes the byte **data** to the 1 Wire bus.
It returns 1 if the write process completed normally or 0 if not.

**unsigned char w1_search(unsigned char cmd,void *p)**

this function returns the number of devices connected to the 1 Wire bus.
If no devices were detected then it returns 0.
The byte **cmd** represents the Search ROM (F0h), Alarm Search (ECh) for the DS1820/DS18S20, or other similar commands, sent to the 1 Wire device.

The pointer **p** points to an area of RAM where are stored the 8 bytes ROM codes returned by the device. After the eighth byte, the function places a ninth status byte which contains a status bit returned by some 1 Wire devices (e.g. DS2405).
Thus the user must allocate 9 bytes of RAM for each device present on the 1 Wire bus.
If there is more then one device connected to the 1 Wire bus, than the user must first call the **w1_search** function to identify the ROM codes of the devices and to be able to address them at a later stage in the program.
Example:

```
#include <mega8515.h>

/* the ATmega8515 port and bit used for the 1 Wire bus must be specified in
   the Project|Configure|C Compiler|Libraries 1 Wire menu */

/* include the 1 Wire bus functions prototypes */
#include <1wire.h>

/* include the printf function prototype */
#include <stdio.h>

/* specify the maximum number of devices connected
   to the 1 Wire bus */
#define MAX_DEVICES 8

/* allocate RAM space for the ROM codes & status bit */
unsigned char rom_codes[MAX_DEVICES][9];

/* quartz crystal frequency [Hz] */
#define xtal 4000000L

/* Baud rate */
#define baud 9600

void main(void) {
unsigned char i,j,devices;

/* initialize the USART control register
   TX enabled, no interrupts, 8 data bits */
UCSRA=0x00;
UCSRB=0x08;
UCSRC=0x86;

/* initialize the USART's baud rate */
UBRRH=(xtal/16/baud-1) >> 8;
UBRRL=(xtal/16/baud-1) & 0xFF;

/* detect how many DS1820/DS18S20 devices
   are connected to the bus and
   store their ROM codes in the rom_codes array */
devices=w1_search(0xf0,rom_codes);
```

```
/* display the ROM codes for each detected device */
printf("%-u DEVICE(S) DETECTED\n\r",devices);
if (devices) {
   for (i=0;i<devices;i++) {
       printf("DEVICE #%-u ROM CODE IS:", i+1);
       for (j=0;j<8;j++) printf("%-X ",rom_codes[i][j]);
       printf("\n\r");
       };
   };
while (1); /* loop forever */
}
```

**unsigned char w1_dow_crc8(void *p, unsigned char n)**

this function checks the 8 bit DOW CRC for a block of bytes with the length **n**, pointed by **p**.
It returns 1 if the DOW CRC of the first **n-1** bytes from the block equals the value of the **n**-th byte, or 0 if it doesn't.

## 5.14 Two Wire Interface Functions for non-XMEGA Devices

The **TWI Functions for non-XMEGA Devices** are intended for easy interfacing between C programs and various external peripherals using the $I^2C$ bus.
These functions can operate the AVR microcontroller as bus master or slave.
The function prototypes, along with helper variable and macro definitions, are placed in the header file **twi.h**, located in the .\INC subdirectory.
This file must be **#include -d** before using the TWI functions.

**Notes:**
- The **twi.h** header file automatically includes the **io.h** header file that contains the I/O modules definitions for the AVR device selected in the project configuration.
- These functions operate using interrupts, so the **interrupts must be globally enabled** using the *#asm("sei")* inline assembly code before attempting any communication through the $I^2C$ bus.
- **For proper operation, the TWI Functions require the presence of 3.3k - 4.7k pull-up resistors to +5V on the SCL and SDA signals.**

### 5.14.1 Two Wire Interface Functions for Master Mode Operation

The following functions are used for operating the TWI in master mode:

**void twi_master_init(unsigned int bit_rate)**

enables and initializes the TWI hardware for operating in master mode.

*Parameters:*

**bit_rate** specifies the SCL clock frequency in kHz.

**bool twi_master_trans(**
**unsigned char slave_addr,**
**unsigned char *tx_data,**
**unsigned char tx_count,**
**unsigned char *rx_data,**
**unsigned char rx_count)**

performs a TWI transaction using the master module.

*Parameters:*

**slave_addr** specifies the 7 bit bus address of the slave with which the transaction should be performed.
**tx_data** represents a pointer to the buffer that holds the data that must be transmitted to the slave during the transaction.
**tx_count** specifies the number of bytes to transmit during the transaction. If no data must be transmitted to the slave, the **tx_data** parameter should be a NULL pointer and **tx_count** must be 0.
**rx_data** represents a pointer to the buffer that will hold the data received from the slave during the transaction.
**rx_count** specifies the number of bytes to be received from the slave during the transaction. If no data must be received from the slave, the **rx_data** parameter should be a NULL pointer and **rx_count** must be 0.

*Return values:*

**true** on success
**false** in case of error.

The nature of the error can be established by reading the value of the **twi_result** global variable, which can take the values defined in the following macros:

#define TWI_RES_OK 0
#define TWI_RES_BUFFER_OVERFLOW 1
#define TWI_RES_ARBITRATION_LOST 2
#define TWI_RES_BUS_ERROR 3
#define TWI_RES_NACK_RECEIVED 4
#define TWI_RES_BUS_TIMEOUT 5
#define TWI_RES_FAIL 6
#define TWI_RES_UNKNOWN 7

**Note: Operating the TWI in master mode requires the interrupts to be globally enabled. This must be done by using the *#asm("sei")* inline assembly code, before attempting any communication through the $I^2C$ bus.**

TWI master operation example:

```
/* accessing an external AT24C16B EEPROM using the
   TWI running in master mode */

/* TWI functions */
#include <twi.h>

/* delay functions */
#include <delay.h>

/* 7 bit TWI bus slave address of the AT24C16B 2kbyte EEPROM */
#define EEPROM_TWI_BUS_ADDRESS (0xA0 >> 1)

void main(void)
{
struct
     {
     struct
          {
          unsigned char msb;
          unsigned char lsb;
          } addr;
     unsigned char data;
     } twi_eeprom;

unsigned char eeprom_rd_data;

/* the TWI master for SCL bit rate of 100 kHz */
twi_master_init(100);

/* globally enable interrupts */
#asm("sei")

/* write the byte 0x55 to the AT24C16B EEPROM address 0x210 */
twi_eeprom.addr.msb=0x02;
twi_eeprom.addr.lsb=0x10;
twi_eeprom.data=0x55;
twi_master_trans(EEPROM_TWI_BUS_ADDRESS,(unsigned char *)
&twi_eeprom,3,0,0);

/* 10ms delay to complete the write operation */
delay_ms(10);
```

```
/* read the byte back into the eeprom_rd_data variable */
twi_master_trans(EEPROM_TWI_BUS_ADDRESS,(unsigned char *)
&twi_eeprom,2,&eeprom_rd_data,1);

/* stop here */
while (1);
}
```

## 5.14.2 Two Wire Interface Functions for Slave Mode Operation

The following function is used for operating the TWI in slave mode:

**void twi_slave_init(**
**bool match_any_addr,**
**unsigned char addr,**
**unsigned char *rx_buffer,**
**unsigned char rx_buffer_size,**
**unsigned char *tx_buffer,**
**bool (*slave_rx_handler) (bool rx_complete),**
**unsigned char (*slave_tx_handler) (bool tx_complete));**

enables and initializes the TWI hardware for operating in slave mode.

*Parameters:*

**match_any_addr** enables the TWI slave to respond to any slave address supplied by the master when starting a transaction.
**addr** represents the 7 bit slave address to which the slave will respond if the **match_any_addr** parameter is **false**.
**rx_buffer** represents a pointer to the buffer that will hold the data received by the slave during the transaction.
**rx_buffer_size** represents the size of the receive buffer, specified in bytes.
**tx_buffer** represents a pointer to the buffer that holds the data to be transmitted by the slave to the master during the transaction.
**slave_rx_handler** represents a pointer to the TWI slave receive processing function.
This function is called each time a byte is received from the master.
It can handle the received data from the receive buffer using the value from the **twi_rx_index** global variable.
The TWI interrupt service routine embedded in **twi.lib**, when calling this function, will pass the **rx_complete** argument as **true** when the master transmission to the slave has finished.
If the slave wishes to terminate the reception of bytes from the slave, the function pointed by the **slave_rx_handler** must return the value **false** when it is called.
If the slave can accept more data bytes from the master, this function should return the value **true**.
**slave_tx_handler** represents a pointer to the TWI slave transmission processing function.
This function is caled twice by the TWI interrupt service routine embedded in **twi.lib.**
On its first call, when the master is ready to receive the data transmitted by the slave, this function is called with the **tx_complete** parameter set as **false** and should return the number of bytes from the transmit buffer that must be transmitted to the master during the ongoing transaction.
After the master has finished receiving the data bytes transmitted by the slave, this function is called for the second time with the **tx_complete** parameter set as **true**, signaling that the transaction on the bus has finished.
In this case the function should return the value 0, as there are no more bytes to be transmitted from the slave to the master.
After the transaction has finished, the **twi_tx_index** global variable will be the index of the last byte in the transmission buffer, that was transmitted by the slave to the master.

After a transaction between the master and the slave has finished, its status can be read from the **twi_result** global variable, which may take one of the values specified by these macros defined in the **twi.h** header file:

#define TWI_RES_OK 0
#define TWI_RES_BUFFER_OVERFLOW 1
#define TWI_RES_BUS_ERROR 3
#define TWI_RES_NACK_RECEIVED 4
#define TWI_RES_UNKNOWN 7

**Note: Operating the TWI in slave mode requires the interrupts to be globally enabled. This must be done by using the *#asm("sei")* inline assembly code, before attempting any communication through the $I^2C$ bus.**

TWI slave operation example:

```
/* The slave will receive the bytes sent by the master and
   will display them on the LCD in ASCII form.
   After a packet of bytes has been received, the slave
   will transmit the char string "Data packet received OK\n\r"
   back to the master.

   The LCD connections must be specified in the
   Project|Configure|C Compiler|Libraries|Alphanumeric LCD menu */

#include <io.h>
#include <stdio.h>
#include <twi.h>
#include <delay.h>
#include <alcd.h>
#include <string.h>

/* 7 bit slave I2C address */
#define TWI_SLAVE_ADDR 0x50

/* slave receive buffer */
char rx_buffer[32];

/* slave transmission buffer */
char tx_buffer[32];

/* flag that signals that the TWI slave reception was OK */
bit received_ok=false;

/* status messages */
flash char * flash status_msg[8]=
{
"OK",
"Buffer overflow",
"Arbitration lost",
"Bus error",
"NACK received",
"Bus timeout",
"Fail",
"Unknown error"
};
```

```
bool slave_rx_handler(bool rx_complete)
{
if (twi_result==TWI_RES_OK)
   received_ok=true; /* signal that data was received without errors */
else
   {
   /* TWI receive error, display the twi_result value on the LCD */
   lcd_clear();
   lcd_putsf("Receive error:\n");
   lcd_putsf(status_msg[twi_result]);
   received_ok=false; /* signal that data was received with errors */
   return false; /* stop reception */
   }

if (rx_complete)
   /* the TWI master has finished transmitting data */
   return false; /* no more bytes to receive */

/* signal to the TWI master that the TWI slave is ready
   to accept more data as long as there is space in the
   receive buffer */
return (twi_rx_index<sizeof(rx_buffer));
}

unsigned char slave_tx_handler(bool tx_complete)
{
unsigned char i;
if (tx_complete==false)
   {
   /* transmission from slave to master is about to start
      copy the text to transmit to the TWI master in the
      transmission buffer */
   strcpyf(tx_buffer,"Data packet received OK\n\r");

   /* # of bytes to transmit from the TWI slave to the TWI master */
   return strlen(tx_buffer);
   }

/* transmission from slave to master has already started,
   no more bytes to send in this transaction */
if (received_ok)
   {
   /* no TWI receive error, display the received data on the LCD */
   lcd_clear();
   for (i=0;i<twi_rx_index;i++) lcd_putchar(rx_buffer[i]);
   }
return 0;
}

void main(void)
{
/* initialize the LCD */
lcd_init(16);
lcd_clear();
```

```
/* initialize the TWI slave */
twi_slave_init(false,TWI_SLAVE_ADDR,rx_buffer,sizeof(rx_buffer),
tx_buffer,slave_rx_handler,slave_tx_handler);

lcd_putsf("TWI slave OK");
delay_ms(2000);

/* enable interrupts to start TWI communication */
#asm("sei")

/* all processing is performed by TWI interrupts inside twi.lib */
while (1);
}
```

## 5.15 Two Wire Interface Functions for XMEGA Devices

The **TWI Functions for XMEGA Devices** are intended for easy interfacing between C programs and various external peripherals using the $I^2C$ bus and SMBus.
These functions can operate the XMEGA AVR microcontroller as both bus master and slave.
The function prototypes, along with helper structure and macro definitions, are placed in the header file **twix.h**, located in the .\INC subdirectory.
This file must be **#include -d** before using the TWI functions.

**Notes:**

- The **twix.h** header file automatically includes the **io.h** header file that contains the I/O modules definitions for the XMEGA device selected in the project configuration.
- The **TWI Functions for XMEGA Devices** operate using interrupts, so the interrupt priority level(s) used by them must be activated. Interrupts must be also globally enabled using the *#asm("sei")* inline assembly code.
- **For proper operation, the TWI Functions require the presence of 3.3k - 4.7k pull-up resistors to +3.3V on the SCL and SDA signals.**

General initialization of the TWI module associated with an XMEGA I/O port is performed by the

**void twi_init(TWI_t *module, bool ext_driver_intf, unsigned char sda_hold)**

function.

*Parameters:*

**module** represents a pointer to the TWI module associated with the I/O port.
**ext_driver** enables the external driver interface. In this situation, the internal TWI drivers with input filtering and slew rate control are bypassed and I/O pin direction must be configured by the user software.
**sda_hold** enables an internal hold time on the SDA signal with respect to the negative edge of SCL.

For XMEGA AU chips **sda_hold** can take one of the following predefined values:

| | |
|---|---|
| **TWI_SDAHOLD_OFF_gc** | - SDA Hold Time off |
| **TWI_SDAHOLD_50NS_gc** | - SDA Hold Time 50 ns |
| **TWI_SDAHOLD_300NS_gc** | - SDA Hold Time 300 ns |
| **TWI_SDAHOLD_400NS_gc** | - SDA Hold Time 400 ns. |

For the rest of XMEGA chips **sda_hold** can take one of the following values:

| | |
|---|---|
| **0** | - SDA Hold Time off |
| **TWI_SDAHOLD_bm** | - SDA Hold Time on. |

## 5.15.1 Two Wire Interface Functions for Master Mode Operation

The following structure data type is defined in the **twix.h** header file for operating the XMEGA TWI in master mode:

```
typedef struct
{
TWI_t *module;                /* pointer to the used TWI interface module */
unsigned char slave_address; /* I2C slave address */
unsigned char *tx_buffer;    /* pointer to the transmit buffer */
unsigned char bytes_to_tx;   /* number of bytes to transmit to the slave */
unsigned char tx_counter;    /* number of transmitted bytes */
unsigned char *rx_buffer;    /* pointer to receive buffer */
unsigned char bytes_to_rx;   /* number of bytes to receive from the slave
*/
unsigned char rx_counter;    /* number of received bytes */
unsigned char result;        /* transaction result */
} TWI_MASTER_INFO_t;
```

The **TWI_MASTER_INFO_t** data type is used for declaring the structure variables, used to hold the information required by each TWI module when operating in master mode.
These structure variables are updated automatically by the TWI functions during bus transactions.

The result of a TWI master transaction is returned in the **result** member of the **TWI_MASTER_INFO_t** structure data type, which may take the values defined by the following macros from **twix.h**:

```
#define TWIM_RES_UNKNOWN 0
#define TWIM_RES_OK 1
#define TWIM_RES_BUFFER_OVERFLOW 2
#define TWIM_RES_ARBITRATION_LOST 3
#define TWIM_RES_BUS_ERROR 4
#define TWIM_RES_NACK_RECEIVED 5
#define TWIM_RES_FAIL 6
```

The macro**:**

**TWI_BAUD_REG(SYS_CLK, TWI _CLK_RATE)**

is used for calculating the value of the TWI **MASTER.BAUD** register for the desired TWI clock rate **TWI_CLK_RATE**, expressed in Hz, based on the System Clock **SYS_CLK** value, expressed in Hz.

The following functions are used for operating the TWI in master mode:

**void twi_master_init(**
**TWI_MASTER_INFO_t *twi,**
**TWI_t *module,**
**TWI_MASTER_INTLVL_t int_level,**
**unsigned char baud_reg)**

enables and initializes a TWI module for operating in master mode.

*Parameters:*

**twi** represents a pointer to the **TWI_MASTER_INFO_t** data type variable, that will be used to hold all the required information for operating in master mode, for a particular TWI module.
**module** represents a pointer to the TWI module, associated with an XMEGA I/O port, that will be enabled and initialized for operation in master mode.
**int_level** specifies the interrupt priority level used by the TWI module when operating in master mode.

**baud_reg** specifies the value used for initializing the TWI module's **MASTER.BAUD** register. Usually the value of this parameter is calculated using the **TWI_BAUD_REG** macro.

**void twi_master_int_handler(TWI_MASTER_INFO_t *twi)**

represents the interrupt handler used for processing the interrupts generated by a TWI module operating in master mode.

*Parameters:*

**twi** represents a pointer to the **TWI_MASTER_INFO_t** data type variable, that is used to hold all the required information for master operation of a particular TWI module associated with an I/O port. The **TWI_MASTER_INFO_t** data type variable must have been first initialized by a call to **twi_master_init**.

The **twi_master_int_handler** function must be called inside the interrupt service routine associated with the master interrupt of a particular TWI module.

**bool twi_master_trans(**
**TWI_MASTER_INFO_t *twi,**
**unsigned char slave_addr,**
**unsigned char *tx_data,**
**unsigned char tx_count,**
**unsigned char *rx_data,**
**unsigned char rx_count)**

performs a TWI transaction using the master module.

*Parameters:*

**twi** represents a pointer to the **TWI_MASTER_INFO_t** data type variable, that is used to hold all the required information for master operation of a particular TWI module associated with an I/O port. The **TWI_MASTER_INFO_t** data type variable must have been first initialized by a call to **twi_master_init**.
**slave_addr** specifies the 7 bit bus address of the slave with which the transaction should be performed.
**tx_data** represents a pointer to the buffer that holds the data that must be transmitted to the slave during the transaction.
**tx_count** specifies the number of bytes to transmit during the transaction. If no data must be transmitted to the slave, the **tx_data** parameter should be a NULL pointer and **tx_count** must be 0.
**rx_data** represents a pointer to the buffer that will hold the data received from the slave during the transaction.
**rx_count** specifies the number of bytes to be received from the slave during the transaction. If no data must be received from the slave, the **rx_data** parameter should be a NULL pointer and **rx_count** must be 0.

*Return values:*

**true** on success
**false** in case of error.

The nature of the error can be established by reading the value of the **result** member of the **TWI_MASTER_INFO_t** structure data type variable pointed by **twi**.

TWI master operation example:

```
/* accessing an external AT24C16B EEPROM using the
   TWID module running in master mode */

/* TWI functions for XMEGA devices */
#include <twix.h>

/* delay functions */
#include <delay.h>

/* TWI clock rate [Hz] */
#define TWI_CLK_RATE 100000

/* 7 bit TWI bus slave address of the AT24C16B 2kbyte EEPROM */
#define EEPROM_TWI_BUS_ADDRESS (0xA0 >> 1)

/* structure that holds information used by the TWID master
   for performing a TWI bus transaction */
TWI_MASTER_INFO_t twid_master;

/* interrupt service routine for TWID master */
interrupt [TWID_TWIM_vect] void twid_master_isr(void)
{
twi_master_int_handler(&twid_master);
}

void main(void)
{
struct
     {
     struct
          {
          unsigned char msb;
          unsigned char lsb;
          } addr;
     unsigned char data;
     } twi_eeprom;

unsigned char eeprom_rd_data;

/* general TWID initialization
   no external driver interface
   no SDA hold time */
twi_init(&TWID,false,false);

/* enable and initialize the TWID master
   interrupt level: low */
twi_master_init(&twid_master,&TWID,TWI_MASTER_INTLVL_LO_gc,
TWI_BAUD_REG(_MCU_CLOCK_FREQUENCY_,TWI_CLK_RATE));

/* enable the Low interrupt level */
PMIC.CTRL|=PMIC_LOLVLEN_bm;

/* globally enable interrupts */
#asm("sei")
```

```
/* write the byte 0x55 to the AT24C16B EEPROM address 0x210 */
twi_eeprom.addr.msb=0x02;
twi_eeprom.addr.lsb=0x10;
twi_eeprom.data=0x55;
twi_master_trans(&twid_master,EEPROM_TWI_BUS_ADDRESS,(unsigned char *)
&twi_eeprom,3,0,0);

/* 10ms delay to complete the write operation */
delay_ms(10);

/* read the byte back into the eeprom_rd_data variable */
twi_master_trans(&twid_master,EEPROM_TWI_BUS_ADDRESS,(unsigned char *)
&twi_eeprom,2,&eeprom_rd_data,1);

/* stop here */
while (1);
}
```

## 5.15.2 Two Wire Interface Functions for Slave Mode Operation

The following structure data type is defined in the **twix.h** header file for operating the XMEGA TWI in slave mode:

```
typedef struct
{
TWI_t *module;                 /* pointer to the used TWI interface module
*/
unsigned char *rx_buffer;      /* pointer to receive buffer */
unsigned char rx_buffer_size; /* receive buffer size */
unsigned char rx_index;        /* index in the receive buffer of the last
received byte */
unsigned char *tx_buffer;      /* pointer to transmit buffer */
unsigned char tx_index;        /* index in the transmit buffer of the last
transmitted byte */
unsigned char bytes_to_tx;     /* number of bytes to transmit to the master
*/
void (*twi_trans) (void);      /* pointer to TWI slave transaction
processing function */
unsigned result;               /* transaction result */
} TWI_SLAVE_INFO_t;
```

The **TWI_SLAVE_INFO_t** data type is used for declaring the structure variables used to hold the information required by each TWI module when operating in slave mode.
These structure variables are updated automatically by the TWI functions during bus transactions.

The result of a TWI slave transaction is returned in the **result** member of the **TWI_SLAVE_INFO_t** structure data type, which may take the values defined by the following macros from **twix.h**:

```
#define TWIS_RES_UNKNOWN 0
#define TWIS_RES_OK 1
#define TWIS_RES_ADDR_MATCH 2
#define TWIS_RES_BUFFER_OVERFLOW 3
#define TWIS_RES_TRANSMIT_COLLISION 4
#define TWIS_RES_BUS_ERROR 5
#define TWIS_RES_FAIL 6
#define TWIS_RES_HALT 7
```

The following functions are used for operating the TWI in slave mode:

**void twi_slave_init(**
    **TWI_SLAVE_INFO_t *twi,**
    **TWI_t *module,**
    **TWI_SLAVE_INTLVL_t int_level,**
    **bool match_any_addr,**
    **unsigned char addr,**
    **unsigned char addr_mask_reg,**
    **unsigned char *rx_buffer,**
    **unsigned char rx_buffer_size,**
    **unsigned char *tx_buffer,**
    **void (*twi_slave_trans)(void));**

enables and initializes the TWI module for operating in slave mode.

*Parameters:*

**twi** represents a pointer to the **TWI_SLAVE_INFO_t** data type variable, that will be used to hold all the required information for operating in slave mode, for a particular TWI module.
**module** represents a pointer to the TWI module, associated with an XMEGA I/O port, that will be enabled and initialized for operation in slave mode.
**int_level** specifies the interrupt priority level used by the TWI module when operating in slave mode.
**match_any_addr** enables the TWI slave to respond to any slave address supplied by the master when starting a transaction.
**addr** represents the 7 bit slave address to which the slave will respond if the **match_any_addr** parameter is **false**.
**addr_mask_reg** specfies the value used to initialize the TWI module's **SLAVE.ADDRMASK** register. If bit 0 of **addr_mask_reg** is 0, then bits 1 to 7 will represent the 7 bit slave address bit mask, otherwise these bits will represent a second 7 bit slave address to which the slave will respond. When address mask mode is used, if a bit in the address mask is set to 1, the address match between the incoming address bit and the corresponding bit from the slave address is ignored, i.e. masked bits will always match.
**rx_buffer** represents a pointer to the buffer that will hold the data received by the slave during the transaction.
**rx_buffer_size** represents the size of the receive buffer, specified in bytes.
**tx_buffer** represents a pointer to the buffer that holds the data to be transmitted by the slave to the master during the transaction.
**twi_slave_trans** represents a pointer to the TWI slave transaction processing function.
This function is called each time a byte is received from the master.
It will handle the received data from the receive buffer using the value from the **rx_index** member of **TWI_SLAVE_INFO_t**.
Also on its first call, when a transaction was started, it must initialize the **bytes_to_tx** member to the number of bytes from the transmit buffer that must be send to the master during the ongoing transaction.
After the current transaction is finished, **bytes_to_tx** will be automatically reset to 0, along with **rx_index**, by the *TWI slave interrupt handler*, so it can be ready to be initialized when the next transaction will start.

**void twi_slave_int_handler(TWI_SLAVE_INFO_t *twi)**

represents the interrupt handler used for processing the interrupts generated by a TWI module operating in master mode.

*Parameters:*

**twi** represents a pointer to the **TWI_SLAVE_INFO_t** data type variable, that is used to hold all the required information for slave operation of a particular TWI module associated with an I/O port.
The **TWI_SLAVE_INFO_t** data type variable must have been first initialized by a call to **twi_slave_init**.

The **twi_slave_int_handler** function must be called inside the interrupt service routine associated with the slave interrupt of a particular TWI module.

**void twi_slave_halt_trans(TWI_SLAVE_INFO_t *twi)**

is used by the slave to halt a TWI transaction.
Usually this function must be called from within the TWI slave transaction processing function specified by the **twi_slave_trans** parameter of **twi_slave_init**.

*Parameters:*

**twi** represents a pointer to the **TWI_SLAVE_INFO_t** data type variable, that is used to hold all the required information for slave operation of a particular TWI module associated with an I/O port.

TWI master and slave operation example:

```
/*
Sample program to test the XMEGA TWIC master and
TWID slave operation.

If one or several switches SW0..SW7 are pressed
on the STK600 board, their state is transmitted
by the TWIC master to the TWID slave, which
displays the received data on the LED0..LED7.
The slave sends to the master the contents of the
test_data array.

Use a STK600 development board with STK600-TQFP100 and
STK600-RC100X-13 addapters

The STK600 programmer must be set in JTAG programming mode
in the Tools|Programmer menu.

Make sure that the VTARGET and VREF voltages are set to 3.6V
using AVR Studio.
The VTARGET LED on the STK600 board must be lighted.

Make the following connections on the STK600:

PC0 - PD0 SDA pin - 4.7K resistor to VTG
PC1 - PD1 SCL pin - 4.7K resistor to VTG
SW0..SW7 - PE0..PE7 using a ribbon cable with 10 pin connectors
LED0..LED7 - PA0..PA7 using a ribbon cable with 10 pin connectors.
*/

/* TWI functions for XMEGA devices */
#include <twix.h>

/* string functions */
#include <string.h>

/* delay functions */
#include <delay.h>

/* TWI clock rate [Hz] */
#define TWI_CLK_RATE 100000

/* 7 bit TWI slave address */
#define SLAVE_ADDR 0x5A
```

```
/* structure that holds information used by the TWIC master
   for performing a TWI bus transaction */
TWI_MASTER_INFO_t twi_master;

/* TWIC master interrupt service routine */
interrupt [TWIC_TWIM_vect] void twic_master_isr(void)
{
twi_master_int_handler(&twi_master);
}

/* TWID slave receive and transmit buffers */
unsigned char twi_slave_rx_buffer[16];
/* data to be transmitted to the master */
unsigned char test_data[]="TWI test";

/* structure that holds information used by the TWID slave
   for performing a TWI bus transaction */
TWI_SLAVE_INFO_t twi_slave;

/* TWID slave interrupt service routine */
interrupt [TWID_TWIS_vect] void twic_slave_isr(void)
{
twi_slave_int_handler(&twi_slave);
}

/* TWID slave transaction processing function */
void twi_slave_trans(void)
{
/* read the received data from the buffer and
   output it to the LEDs on PORTA */
PORTA.OUT=twi_slave.rx_buffer[twi_slave.rx_index];

/* prepare to transmit the contents of the
   test_data array to the master,
   initialize the number of bytes to transmit
   only once at the beginning of the transmission */
if (twi_slave.bytes_to_tx==0) twi_slave.bytes_to_tx=sizeof(test_data);

/* if needed, the function:

   twi_slave_halt(&twi_slave);

   can be called here in order to halt
   the transaction by the slave */
}

void main(void)
{
unsigned char switches;
/* received data from the slave */
unsigned char rx_data[sizeof(test_data)];

/* initialize PORTA as inverted outputs, used for driving LEDs */
PORTA.OUT=0x00; /* all LEDs are initially off */
PORTA.DIR=0xFF;
```

```
PORTA.PIN0CTRL=PORT_INVEN_bm | PORT_OPC_TOTEM_gc;
PORTA.PIN1CTRL=PORT_INVEN_bm | PORT_OPC_TOTEM_gc;
PORTA.PIN2CTRL=PORT_INVEN_bm | PORT_OPC_TOTEM_gc;
PORTA.PIN3CTRL=PORT_INVEN_bm | PORT_OPC_TOTEM_gc;
PORTA.PIN4CTRL=PORT_INVEN_bm | PORT_OPC_TOTEM_gc;
PORTA.PIN5CTRL=PORT_INVEN_bm | PORT_OPC_TOTEM_gc;
PORTA.PIN6CTRL=PORT_INVEN_bm | PORT_OPC_TOTEM_gc;
PORTA.PIN7CTRL=PORT_INVEN_bm | PORT_OPC_TOTEM_gc;

/* initialize PORTE as inputs used for reading switches
   pullup resistors are already present on the STK600 board */
PORTE.DIR=0x00;

/* general TWIC initialization
   no external driver interface
   no SDA hold time */
twi_init(&TWIC,false,false);

/* initialize the TWIC master
   use low priority level interrupt */
twi_master_init(&twi_master,&TWIC,TWI_MASTER_INTLVL_LO_gc,
    TWI_BAUD_REG(_MCU_CLOCK_FREQUENCY_,TWI_CLK_RATE));

/* general TWID initialization
   no external driver interface
   no SDA hold time */
twi_init(&TWID,false,false);

/* initialize the TWID slave
   use low priority level interrupt */
twi_slave_init(&twi_slave,&TWID,TWI_SLAVE_INTLVL_LO_gc,
    false,SLAVE_ADDR,0,
    twi_slave_rx_buffer,sizeof(twi_slave_rx_buffer),
    test_data,twi_slave_trans);

/* enable low interrupt level interrupts */
PMIC.CTRL|=PMIC_LOLVLEN_bm;

/* globaly enable interrupts */
#asm("sei")

while (1)
      {
      /* read the SW0..7 switches
         the switches value is inverted because
         the connection is established to GND
         when a switch is pressed */
      switches= ~PORTE.IN;

      /* check if at least one switch was pressed */
      if (switches)
         {
         /* yes, transmit the switches state to the slave
            and receive the contents of the test_data array
            sent by the slave in  rx_data */
         twi_master_trans(&twi_master,SLAVE_ADDR,
                          &switches,sizeof(switches),
                          rx_data,sizeof(rx_data));
```

```
        /* check that correct data was received from the slave */
        if (strncmp(rx_data,test_data,sizeof(test_data)))
           /* if rx_data doesn't match test_data... */
           while (1)
                 {
                 /* flash all LEDs to signal the mismatch */
                 PORTA.OUT=0xFF;
                 delay_ms(200);
                 PORTA.OUT=0x00;
                 delay_ms(200);
                 }
        }
     }
}
```

## 5.16 Software Bit-Banged $I^2C$ Bus Functions

The $I^2C$ Functions are intended for easy interfacing between C programs and various peripherals using the Philips $I^2C$ bus.
These functions treat the microcontroller as a bus master and the peripherals as slaves.
The $I^2C$ Functions are implemented in software, using the "bit-banging" method.
The hardware TWI is not used.

**Notes:**

- **For proper operation, the $I^2C$ Functions require the presence of 3.3k - 4.7k pull-up resistors to +5V on the SCL and SDA signals.**
- The bit-banged $I^2C$ Functions functions do not support the XMEGA chips.

The prototypes for these functions are placed in the file **i2c.h**, located in the .\INC subdirectory. This file must be **#include -d** before using the functions.

These functions must be configured, by specifying the I/O port and bits used for communication through the $I^2C$ bus and the bit rate of the SCL clock.
This is accomplished in the **Project|Configure|C Compiler|Libraries|I2C** menu:

- the **Enable Bit-Banged I2C Support** option must be activated
- the **I/O Port**, **SDA** and **SCL** bits must be specified in **Data Connection**
- the Bit Rate of the SCL signal must be set.

**Note:** For compatibility with projects developed with CodeVisionAVR prior to V2.05.1, the $I^2C$ Functions can also be configured as outlined in the example below.
However in this case, the **Enable Bit-Banged I2C Support** option must be disabled in the **Project|Configure|C Compiler|Libraries|I2C** menu.

Example:

```
/* ATmega8515 PORTB bits 0 and 1 are used for the I2C bus
   signals SDA and SCL */
#asm
    .equ __i2c_port=0x18
    .equ __sda_bit=0
    .equ __scl_bit=1
#endasm

/* now you can include the I2C Functions */
#include <i2c.h>
```

This method is not recommended for new projects.

The I$^2$C Functions are:

**void i2c_init(void)**

this function initializes the I$^2$C bus.
This is the first function that must be called prior to using the other I$^2$C Functions.

**unsigned char i2c_start(void)**

issues a START condition.
Returns 1 if bus is free or 0 if the I$^2$C bus is busy.

**void i2c_stop(void)**

issues a STOP condition.

**unsigned char i2c_read(unsigned char ack)**

reads a byte from the bus.

The **ack** parameter specifies if an acknowledgement is to be issued after the byte was read.
Set **ack** to 0 for no acknowledgement or 1 for acknowledgement.

**unsigned char i2c_write(unsigned char data)**

writes the byte data to the bus.
Returns 1 if the slave acknowledges or 0 if not.
Example how to access an Atmel 24C02 256 byte I$^2$C EEPROM:

```
/* include the I2C bus functions
   The I2C bus connections and bit rate must be specified in the
   Project|Configure|C Compiler|Libraries|I2C menu */
#include <i2c.h>

/* function declaration for delay_ms */
#include <delay.h>

#define EEPROM_BUS_ADDRESS 0xa0

/* read a byte from the EEPROM */
unsigned char eeprom_read(unsigned char address) {
unsigned char data;
i2c_start();
i2c_write(EEPROM_BUS_ADDRESS);
i2c_write(address);
i2c_start();
i2c_write(EEPROM_BUS_ADDRESS | 1);
data=i2c_read(0);
i2c_stop();
return data;
}
```

```
/* write a byte to the EEPROM */
void eeprom_write(unsigned char address, unsigned char data) {
i2c_start();
i2c_write(EEPROM_BUS_ADDRESS);
i2c_write(address);
i2c_write(data);
i2c_stop();
/* 10ms delay to complete the write operation */
delay_ms(10);
}

void main(void) {
unsigned char i;
/* initialize the I2C bus */
i2c_init();
/* write the byte 55h at address AAh */
eeprom_write(0xaa,0x55);
/* read the byte from address AAh */
i=eeprom_read(0xaa);
while (1); /* loop forever */
}
```

## 5.17 SPI Functions

The SPI Functions are intended for easy interfacing between C programs and various peripherals using the SPI bus.
The prototypes for these functions are placed in the file **spi.h**, located in the .\INC subdirectory. This file must be **#include -d** before using the functions.
The SPI functions are:

**unsigned char spi(unsigned char data)**

this function sends the byte data, simultaneously receiving a byte.

Prior to using the **spi** function, you must configure the SPI Control Register SPCR according to the Atmel Data Sheets.
Because the **spi** function uses polling for SPI communication, there is no need to set the SPI Interrupt Enable Bit SPIE.

For the XMEGA chips the **spi** function use by default the **SPIC** controller on I/O port **PORTC**.
If you wish to use another SPI controller, you must define the **_ATXMEGA_SPI_** and **_ATXMEGA_SPI_PORT_** preprocessor macros prior to #include the **spi.h** header file, like in the example below:

```
/* use the ATxmega128A1 SPID for the spi function */
#define _ATXMEGA_SPI_ SPID
#define _ATXMEGA_SPI_PORT_ PORTD

/* use the SPI functions */
#include <spi.h>
```

The **_ATXMEGA_SPI_** and **_ATXMEGA_SPI_PORT_** macros needs to be defined only once in the whole program, as the compiler will treat them like they are globally defined.

For the XMEGA chips the SPI library also contains the following function:

**void spi_init(bool master_mode, bool lsb_first, SPI_MODE_t mode, bool clk2x, SPI_PRESCALER_t clock_div, unsigned char ss_pin)**

this function initializes the SPI controller and corresponding I/O port as defined by the **_ATXMEGA_SPI_** and **_ATXMEGA_SPI_PORT_** macros.
If the **master_mode** parameter is true, the SPI controller will function in master mode, otherwise it will function in slave mode.
If the **lsb_first** parameter is true, the data byte sent/received on the bus will start with bit 0, otherwise it will start with bit 7.
The **mode** parameter specifies the SPI clock polarity and phase. The **SPI_MODE_t** data type and SPI modes are defined in the header file **xmbits_a1.h**:

**SPI_MODE_0_gc** for SPI mode 0
**SPI_MODE_1_gc** for SPI mode 1
**SPI_MODE_2_gc** for SPI mode 2
**SPI_MODE_3_gc** for SPI mode 3.

If the **clk2x** parameter is true, the SPI master will function in double speed mode.

The **clock_div** parameter specifies the SPI clock prescaler division factor. The **SPI_PRESCALER_t** data type and SPI prescaler values are defined in the header file **xmbits_a1.h**:

**SPI_PRESCALER_DIV4_gc** for System Clock/4
**SPI_PRESCALER_DIV16_gc** for System Clock/16
**SPI_PRESCALER_DIV64_gc** for System Clock/64
**SPI_PRESCALER_DIV128_gc** for System Clock/128.

The **ss_pin** parameter specifies the SPI I/O port pin that is used for SS. Its values are defined in the header file **xmbits_a1.h**.

Example of using the **spi** function for interfacing to an AD7896 ADC:

```
/*
   Digital voltmeter using an
   Analog Devices AD7896 ADC
   connected to an AT90mega8515
   using the SPI bus

   Chip: AT90mega8515
   Memory Model: SMALL
   Data Stack Size: 128 bytes
   Clock frequency: 4MHz

   AD7896 connections to the ATmega8515

   [AD7896]  [ATmega8515 DIP40]
    1 Vin
    2 Vref=5V
    3 AGND  - 20 GND
    4 SCLK  - 8  SCK
    5 SDATA - 7  MISO
    6 DGND  - 20 GND
    7 CONVST- 2  PB1
    8 BUSY  - 1  PB0

   Use an 2x16 alphanumeric LCD connected
   to PORTC as follows:

  [LCD]   [ATmega8515 DIP40]
   1 GND- 20 GND
   2 +5V- 40 VCC
   3 VLC
   4 RS - 21 PC0
   5 RD - 22 PC1
   6 EN - 23 PC2
  11 D4 - 25 PC4
  12 D5 - 26 PC5
  13 D6 - 27 PC6
  14 D7 - 28 PC7 */
```

```
/* include the LCD Functions
   The connections must be specified in the
   Project|Configure|C Compiler|Libraries|Alphanumeric LCD menu */
#include <alcd.h>
/* SPI driver function */
#include <spi.h>
#include <mega8515.h>
#include <stdio.h>
#include <delay.h>

/* AD7896 reference voltage [mV] */
#define VREF 5000L

/* AD7896 control signals PORTB bit allocation */
#define ADC_BUSY PINB.0
#define NCONVST PORTB.1

/* LCD display buffer */
char lcd_buffer[33];

unsigned read_adc(void)
{
unsigned result;
/* start conversion in mode 1, (high sampling performance) */
NCONVST=0;
NCONVST=1;
/* wait for the conversion to complete */
while (ADC_BUSY);
/* read the MSB using SPI */
result=(unsigned) spi(0)<<8;
/* read the LSB using SPI and combine with MSB */
result|=spi(0);
/* calculate the voltage in [mV] */
result=(unsigned) (((unsigned long) result*VREF)/4096L);
/* return the measured voltage */
return result;
}

void main(void)
{
/* initialize PORTB
   PB.0 input from AD7896 BUSY
   PB.1 output to AD7896 /CONVST
   PB.2 & PB.3 inputs
   PB.4 output (SPI /SS pin)
   PB.5 input
   PB.6 input (SPI MISO)
   PB.7 output to AD7896 SCLK */
DDRB=0x92;

/* initialize the SPI in master mode
   no interrupts, MSB first, clock phase negative
   SCK low when idle, clock phase=0
   SCK=fxtal/4 */
SPCR=0x54;
SPSR=0x00;
```

```
/* the AD7896 will work in mode 1 (high sampling performance)
   /CONVST=1, SCLK=0 */
PORTB=2;

/* initialize the LCD */
lcd_init(16);

lcd_putsf("AD7896 SPI bus\nVoltmeter");
delay_ms(2000);
lcd_clear();

/* read and display the ADC input voltage */
while (1)
      {
      sprintf(lcd_buffer,"Uadc=%4umV",read_adc());
      lcd_clear();
      lcd_puts(lcd_buffer);
      delay_ms(100);
   };
}
```

## 5.18 Power Management Functions

The Power Management Functions are intended for putting the AVR chip in one of its low power consumption modes.
The prototypes for these functions are placed in the file **sleep.h**, located in the .\INC subdirectory. This file must be **#include -d** before using the functions.
The Power Management Functions are:

**void sleep_enable(void)**

this function enables entering the low power consumption modes.

**void sleep_disable(void)**

this function disables entering the low power consumption modes.
It is used to disable accidental entering the low power consumption modes.

**void idle(void)**

this function puts the AVR chip in the idle mode.
Prior to using this function, the **sleep_enable** function must be invoked to allow entering the low power consumption modes.
In this mode the CPU is stopped, but the Timers/Counters, Watchdog and interrupt system continue operating.
The CPU can wake up from external triggered interrupts as well as internal ones.

**void powerdown(void)**

this function puts the AVR chip in the powerdown mode.
Prior to using this function, the **sleep_enable** function must be invoked to allow entering the low power consumption modes.
In this mode the external oscillator is stopped.
The AVR can wake up only from an external reset, Watchdog time-out or external level triggered interrupt.

**void powersave(void)**

this function puts the AVR chip in the powersave mode.
Prior to using this function, the **sleep_enable** function must be invoked to allow entering the low power consumption modes.
This mode is similar to the powerdown mode with some differences, please consult the Atmel Data Sheet for the particular chip that you use.

**void standby(void)**

this function puts the AVR chip in the standby mode.
Prior to using this function, the **sleep_enable** function must be invoked to allow entering the low power consumption modes.
This mode is similar to the powerdown mode with the exception that the external clock oscillator keeps on running.
Consult the Atmel Data Sheet for the particular chip that you use, in order to see if the standby mode is available for it.

**void extended_standby(void)**

this function puts the AVR chip in the extended standby mode.
Prior to using this function, the **sleep_enable** function must be invoked to allow entering the low power consumption modes.
This mode is similar to the powersave mode with the exception that the external clock oscillator keeps on running.
Consult the Atmel Data Sheet for the particular chip that you use, in order to see if the standby mode is available for it.

**Note:** There are specific situations where the power management functions can't be used because of the timing limitations.
For example the ATmega168P chip has a feature which is not available in ATmega168: Brown-Out Detection disable during sleep.
If we wish to use this feature, we need to enter in sleep mode in maximum 4 clocks after the **BODS** bit is set in the **MCUCR** register.
But calling and executing the **powersave** function requires a longer time than than this, so this example code will not function correctly:

```
unsigned char tmp;

sleep_enable();
/* Disable brown out detection in sleep */
tmp = MCUCR | (1<<BODS) | (1<<BODSE);
MCUCR = tmp;
MCUCR = tmp & (~(1<<BODSE));
powersave();  /* Takes too long until the sleep instruction is executed */
```

This is the correct code:

```
unsigned char tmp;

/* Prepare the sleep in power save mode*/
SMCR |= (1<<SE) | (1<<SM1) | (1<<SM0);
/* Disable brown out detection in sleep */
tmp = MCUCR | (1<<BODS) | (1<<BODSE);
MCUCR = tmp;
MCUCR = tmp & (~(1<<BODSE));
/* Enter sleep mode */
#asm("sleep");
```

## 5.19 Delay Functions

These functions are intended for generating delays in C programs.
The prototypes for these functions are placed in the file **delay.h**, located in the .\INC subdirectory. This file must be **#include -d** before using the functions.
Before calling the functions the interrupts must be disabled, otherwise the delays will be much longer then expected.
Also it is very important to specify the correct AVR chip clock frequency in the **Project|Configure|C Compiler|Code Generation** menu.

The functions are:

**void delay_us(unsigned int n)**

generates a delay of n μseconds. n must be a constant expression.

**void delay_ms(unsigned int n)**

generates a delay of n milliseconds.
This function automatically resets the wtachdog timer every 1ms by generating the **wdr** instruction.

Example:

```
void main(void) {
/* disable interrupts */
#asm("cli")

/* 100µs delay */
delay_us(100);

/* ............. */

/* 10ms delay */
delay_ms(10);

/* enable interrupts */
#asm("sei")

/* ............. */
}
```

## 5.20 MMC/SD/SD HC FLASH Memory Card Driver Functions

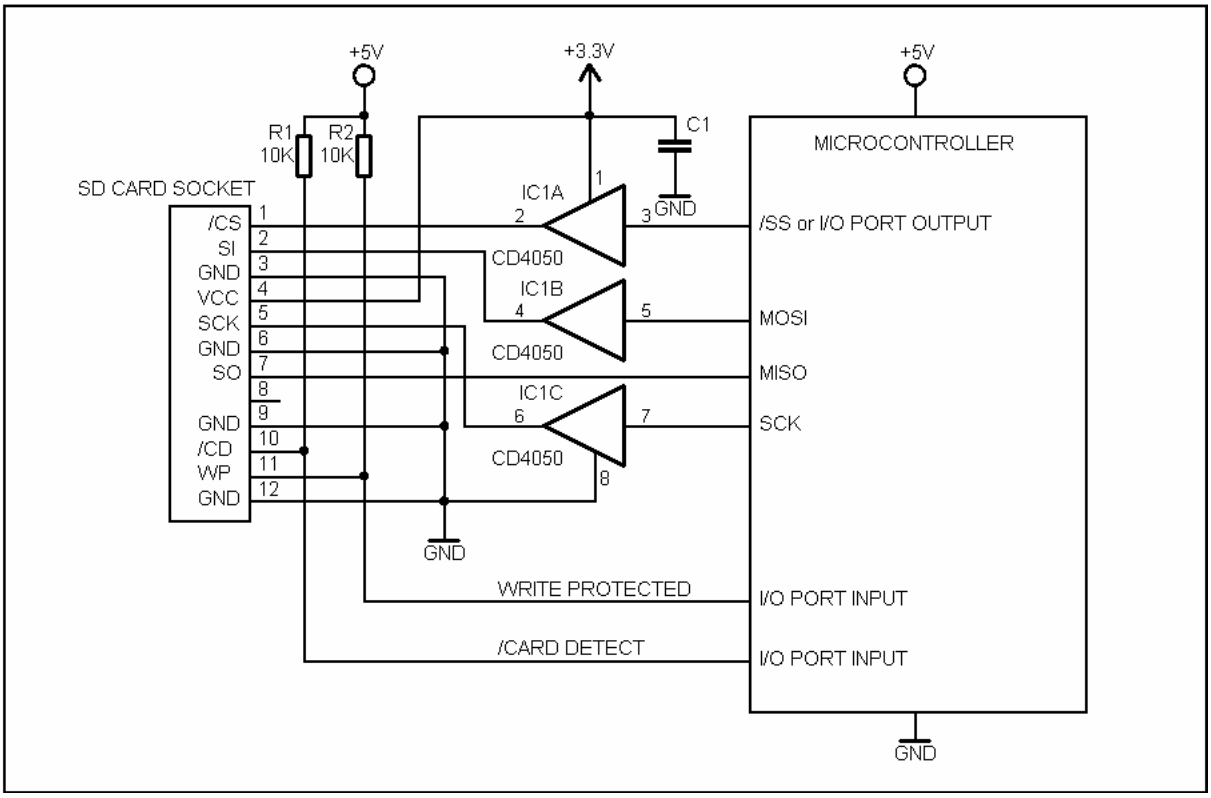
The MMC/SD/SD HC FLASH Memory Card Driver Functions are intended for interfacing between C programs and MMC, SD, SD HC cards using the SPI bus interface.
These low level functions are referenced by the high level **FAT Access Functions**.

The unctions are based on the open source drivers provided by Mr. ChaN from Electronic Lives Mfg.
http://elm-chan.org

Before using the card driver functions, the I/O port signals employed for communication with the MMC/SD/SD HC card must be configured in the **Project|Configure|C Compiler|Libraries|MMC/SD/SD HC Card** menu.

**Note:** The MMC/SD/SD HC card driver functions are not re-entrant. They must not be called from interrupt service routines.

The MMC/SD/SD HC card must be connected to the AVR microcontroller using a CD4050 CMOS buffer that will translate the 5V logic signals to 3.3V as needed by the card.
The connection schematic is provided below:




**Note:** The drivers can be also used with hardware designs which set the **WP** signal to logic 0 when the MMC/SD/SD HC Card is write protected. In this case the **WP Active Low** option must be enabled in the **Project|Configure|C Compiler|Libraries|MMC/SD/SD HC Card** menu.

The MMC/SD/SD HC card driver function prototypes, helper type definitions and macros are placed in the header file **sdcard.h**, located in the .\INC subdirectory. This file must be **#include -d** before using the functions.

The MMC/SD/SD HC card driver functions are:

**void disk_timerproc (void)**

is a low level timing function that must be called every 10ms by a Timer interrupt.

**Note:** It is mandatory to ensure that this function is called every 10ms in your program.
Otherwise the MMC/SD/SD HC card driver functions will lock in an endless loop when testing for disk operations timeout.

Example:

```
/* ATmega128 I/O register definitions */
#include <mega128.h>
/* MMC/SD/SD HC card support */
#include <sdcard.h>

/* Timer1 overflow interrupt frequency [Hz] */
#define T1_OVF_FREQ 100
/* Timer1 clock prescaler value */
#define T1_PRESC 1024L
/* Timer1 initialization value after overflow */
#define T1_INIT (0x10000L-(_MCU_CLOCK_FREQUENCY_/(T1_PRESC*T1_OVF_FREQ)))

/* 100Hz timer interrupt generated by ATmega128 Timer1 overflow */
interrupt [TIM1_OVF] void timer_comp_isr(void)
{
/* re-initialize Timer1 */
TCNT1H=T1_INIT>>8;
TCNT1L=T1_INIT&0xFF;
/* card access low level timing function */
disk_timerproc();

/* the rest of the interrupt service routine */
/* .... */

}

void main(void)
{
/* initialize Timer1 overflow interrupts in Mode 0 (Normal) */
TCCR1A=0x00;
/* clkio/1024 */
TCCR1B=(1<<CS12)|(1<<CS10);
/* timer overflow interrupts will occur with 100Hz frequency */
TCNT1H=T1_INIT>>8;
TCNT1L=T1_INIT&0xFF;
/* enable Timer1 overflow interrupt */
TIMSK=1<<TOIE1;
/* globally enable interrupts */
#asm("sei")

/* the rest of the program */
/* .... */

while(1)
       {
       /* .... */
       }
}
```

**unsigned char disk_initialize(unsigned char drv)**

performs the initialization, including the SPI bus interface and I/O ports, of a physical drive located on a MMC, SD or SD HC card.

*Parameters:*

**drv** represents the drive number. Drive numbering starts with 0.

*Return value:*

The function returns 1 byte containing the disk status flags, specified by the following macros defined in **sdcard.h**:

- STA_NOINIT (=0x01, bit 0 of function result) Disk drive not initialized. This flag is set after microcontroller reset, card removal or when the **disk_initialize** function has failed.
- STA_NODISK (=0x02, bit 1 of function result) This flag is set if no card is inserted in the socket. **Note:** the STA_NOINIT flag is also set in this situation.
- STA_PROTECT (=0x04, bit 2 of function result) Card is write protected. If the STA_NODISK flag is also set, the STA_PROTECT flag is not valid.

On success, the function returns 0, which means all status flags are reset.

**Note:**

- For the MMC/SD/SD HC card driver using the SPI interface, the **drv** parameter must be always 0, otherwise the function will return with the STA_NOINIT flag set.
- The **disk_initialize** function will always configure the I/O port pin, where the SPI /SS signal is present, as an output. This is required for the correct operation of the SPI in master mode.

The /SS signal can be used as a general purpose output, without affecting the MMC/SD/SD HC card driver operation.

Example:

```
/* ATmega128 I/O register definitions */
#include <mega128.h>
/* MMC/SD/SD HC card support */
#include <sdcard.h>
/* delay functions */
#include <delay.h>

/* include the LCD Functions
   The connections must be specified in the
   Project|Configure|C Compiler|Libraries|Alphanumeric LCD menu */
#include <alcd.h>

/* Timer1 overflow interrupt frequency [Hz] */
#define T1_OVF_FREQ 100
/* Timer1 clock prescaler value */
#define T1_PRESC 1024L
/* Timer1 initialization value after overflow */
#define T1_INIT (0x10000L-(_MCU_CLOCK_FREQUENCY_/(T1_PRESC*T1_OVF_FREQ)))
```

```
/* 100Hz timer interrupt generated by ATmega128 Timer1 overflow */
interrupt [TIM1_OVF] void timer_comp_isr(void)
{
/* re-initialize Timer1 */
TCNT1H=T1_INIT>>8;
TCNT1L=T1_INIT&0xFF;
/* card access low level timing function */
disk_timerproc();

/* the rest of the interrupt service routine */
/* .... */

}

void main(void)
{
unsigned char status;

/* initialize Timer1 overflow interrupts in Mode 0 (Normal) */
TCCR1A=0x00;
/* clkio/1024 */
TCCR1B=(1<<CS12)|(1<<CS10);
/* timer overflow interrupts will occur with 100Hz frequency */
TCNT1H=T1_INIT>>8;
TCNT1L=T1_INIT&0xFF;
/* enable Timer1 overflow interrupt */
TIMSK=1<<TOIE1;
/* initialize the LCD, 16 characters/line */
lcd_init(16);
/* globally enable interrupts */
#asm("sei")
/* initialize SPI interface and card driver */
status=disk_initialize(0);
/* clear the LCD */
lcd_clear();
/* display disk initialization result on the LCD */
if (status & STA_NODISK) lcd_puts("Card not present");
else
if (status & STA_NOINIT) lcd_puts("Disk init failed");
else
if (status & STA_PROTECT) lcd_puts("Card write\nprotected");
/* all status flags are 0, disk initialization OK */
else lcd_puts("Init OK");
/* wait 2 seconds */
delay_ms(2000);

/* the rest of the program */
/* .... */

while(1)
       {
       /* .... */

       }
}
```

**bool sdcard_present(void)**

is used for detecting the presence of a FLASH memory card inserted in the socket. It returns **true** in this case.

**Notes:**
- This function must be called before performing memory card accesses, in the situation when the usage of the **/CD** (card detect) signal is disabled in the in the **Project|Configure|C Compiler|Libraries|MMC/SD/SD HC Card** menu.
- If the function is called before **disk_initialize**, it will automatically first perform the initialization, including the SPI bus interface and I/O ports, of the physical drive.

**unsigned char disk_status(unsigned char drv)**

returns the current disk status of a physical drive located on a MMC, SD or SD HC card.

*Parameters:*

**drv** represents the drive number. Drive numbering starts with 0.

*Return value:*

The function returns 1 byte containing the disk status flags, specified by the following macros defined in **sdcard.h**:

- STA_NOINIT (=0x01, bit 0 of function result) Disk drive not initialized. This flag is set after microcontroller reset, card removal or when the **disk_initialize** function has failed.
- STA_NODISK (=0x02, bit 1 of function result) This flag is set if no card is inserted in the socket. **Note:** the STA_NOINIT flag is also set in this situation.
- STA_PROTECT (=0x04, bit 2 of function result) Card is write protected. If the STA_NODISK flag is also set, the STA_PROTECT flag is not valid.

On success, the function returns 0, which means all status flags are reset.

**Note:** For the MMC/SD/SD HC card driver using the SPI interface, the **drv** parameter must be always 0, otherwise the function will return with the STA_NOINIT flag set.

The **DRESULT** enumeration data type is defined in **sdcard.h**:

```
typedef enum
      {
      RES_OK=0,           /* 0: Successful */
      RES_ERROR,          /* 1: R/W Error */
      RES_WRPRT,          /* 2: Write Protected */
      RES_NOTRDY,         /* 3: Not Ready */
      RES_PARERR          /* 4: Invalid Parameter */
      } DRESULT;
```

It is used for returning the result of the following driver functions:

**DRESULT disk_read (unsigned char drv, unsigned char* buff, unsigned long sector, unsigned char count)**

reads sectors from a physical drive.

*Parameters:*

**drv** represents the drive number. Drive numbering starts with 0.
**buff** points to the char array where read data will be stored.
**sector** represents the Logical Block Address number of the first sector to be read.
**count** represents the number of sectors to be read (1..255).

*Return value:*

RES_OK - success
RES_ERROR - a write error occured
RES_WRPRT - the MMC/SD/SD HC card is write protected
RES_NOTRDY - the disk drive has not been initialized
RES_PARERR - invalid parameters were passed to the function.

**Note:** For the MMC/SD/SD HC card driver using the SPI interface, the **drv** parameter must be always 0, otherwise the function will return with the STA_NOINIT flag set.

**DRESULT disk_write (unsigned char drv, unsigned char* buff, unsigned long sector, unsigned char count)**

writes sectors to a physical drive.

*Parameters:*

**drv** represents the drive number. Drive numbering starts with 0.
**buff** points to the char array where the data to be written is stored.
**sector** represents the Logical Block Address number of the first sector to be written.
**count** represents the number of sectors to be written (1..255).

*Return value:*

RES_OK - success
RES_ERROR - a write error occured
RES_WRPRT - the SD/SD HC card is write protected
RES_NOTRDY - disk drive has not been initialized
RES_PARERR - invalid parameters were passed to the function.

**Note:** For the MMC/SD/SD HC card driver using the SPI interface, the **drv** parameter must be always 0, otherwise the function will return with the STA_NOINIT flag set.

**DRESULT disk_ioctl (unsigned char drv, unsigned char ctrl, void* buff)**

this function is used for controlling MMC/SD/SD HC card specific features and other disk functions.

*Parameters*

**drv** represents the drive number. Drive numbering starts with 0.
**ctrl** specifies the command code.
**buff** points to the buffer that will hold function results depending on the command code.
When not used, a NULL pointer must be passed as parameter.

*Return value:*

RES_OK - success
RES_ERROR - an error occured
RES_NOTRDY - the disk drive has not been initialized
RES_PARERR - invalid parameters were passed to the function.

**Note:** For the MMC/SD/SD HC card driver using the SPI interface, the **drv** parameter must be always 0, otherwise the function will return with the STA_NOINIT flag set.

The following **ctrl** command codes, specified by the macros defined in the **sdcard.h** header file, can be issued to the **disk_ioctl** function:

- CTRL_SYNC - wait until the disk drive has finished the write process. The **buff** pointer must be NULL.
- GET_SECTOR_SIZE - returns the size of the drive's sector. The **buff** pointer must point to a 16bit **unsigned int** variable, that will contain the sector size. For MMC/SD/SD HC cards the returned sector size will be 512 bytes.
- GET_SECTOR_COUNT - returns the total number of sectors on the drive. The **buff** pointer must point to a 32bit **unsigned long int** variable, that will contain the sector count.
- GET_BLOCK_SIZE - returns the erase block size of the drive's memory array in sectors count. The **buff** pointer must point to a 32bit **unsigned long int** variable, that will contain the block size. If the erase block size is not known, the returned value will be 1.

Example:

```
/* ATmega128 I/O register definitions */
#include <mega128.h>
/* MMC/SD/SD HC card support */
#include <sdcard.h>
/* delay functions */
#include <delay.h>
/* sprintf */
#include <stdio.h>

/* include the LCD Functions
   The connections must be specified in the
   Project|Configure|C Compiler|Libraries|Alphanumeric LCD menu */
#include <alcd.h>

/* Timer1 overflow interrupt frequency [Hz] */
#define T1_OVF_FREQ 100
/* Timer1 clock prescaler value */
#define T1_PRESC 1024L
/* Timer1 initialization value after overflow */
#define T1_INIT (0x10000L-(_MCU_CLOCK_FREQUENCY_/(T1_PRESC*T1_OVF_FREQ)))
```

```
/* 100Hz timer interrupt generated by ATmega128 Timer1 overflow */
interrupt [TIM1_OVF] void timer_comp_isr(void)
{
/* re-initialize Timer1 */
TCNT1H=T1_INIT>>8;
TCNT1L=T1_INIT&0xFF;
/* card access low level timing function */
disk_timerproc();

/* the rest of the interrupt service routine */
/* .... */

}

void main(void)
{
char display_buffer[64]; /* buffer used by sprintf */
unsigned char status;
unsigned int sector_size;
unsigned long int sector_count;

/* initialize Timer1 overflow interrupts in Mode 0 (Normal) */
TCCR1A=0x00;
/* clkio/1024 */
TCCR1B=(1<<CS12)|(1<<CS10);
/* timer overflow interrupts will occur with 100Hz frequency */
TCNT1H=T1_INIT>>8;
TCNT1L=T1_INIT&0xFF;
/* enable Timer1 overflow interrupt */
TIMSK=1<<TOIE1;
/* initialize the LCD */
lcd_init(16);
/* globally enable interrupts */
#asm("sei")
/* initialize SPI interface and card driver */
status=disk_initialize(0);
/* clear the LCD */
lcd_clear();
/* display disk initialization result on the LCD */
if (status & STA_NOINIT) lcd_puts("Disk init failed");
else
if (status & STA_NODISK) lcd_puts("Card not present");
else
if (status & STA_PROTECT) lcd_puts("Card write\nprotected");
/* all status flags are 0, disk initialization OK */
else
    {
    lcd_puts("Init OK");
    /* wait 2 seconds */
    delay_ms(2000);
    /* clear the LCD */
    lcd_clear();
```

```
    /* get the sector size */
    if (disk_ioctl(0,GET_SECTOR_SIZE,&sector_size)==RES_OK)
       {
       /* sector size read OK, display it */
       sprintf(display_buffer,"Sector size=%u",sector_size);
       lcd_puts(display_buffer);
       /* wait 2 seconds */
       delay_ms(2000);
       /* clear the LCD */
       lcd_clear();
       /* get the sector count */
       if (disk_ioctl(0,GET_SECTOR_COUNT,&sector_count)==RES_OK)
          {
          /* sector count read OK, display it */
          sprintf(display_buffer,"Sector count=%lu",sector_count);
          lcd_puts(display_buffer);
          }
       else lcd_puts("Error reading\nsector count");
       }
    else lcd_puts("Error reading\nsector size");
    }

/* wait 2 seconds */
delay_ms(2000);

/* the rest of the program */
/* .... */

while(1)
       {
       /* .... */

       }
}
```

**Note:** When compiling the above example, make sure that the **(s)printf Features** option in the **Project|Configure|C Compiler|Code Generation** menu will be set to: **long, width**.
This will ensure that the unsigned long int **sector_count** variable will be displayed correctly by the **sprintf** function.

## 5.21 FAT Access Functions

These functions are intended for high level data access to MMC/SD/SD HC FLASH memory cards formated using the FAT12, FAT16 or FAT32 standards.

The FAT access functions are based on FATFS open source library by Mr. ChaN from Electronic Lives Mfg. http://elm-chan.org

The FAT access function prototypes, helper type definitions and macros are placed in the header file **ff.h**, located in the .\INC subdirectory. This file must be **#include -d** before using the functions.

The FAT access functions call the low level **MMC/SD/SD HC Card Driver** functions, so the I/O port signals employed for communication with the MMC/SD/SD HC card must be configured in the **Project|Configure|C Compiler|Libraries|MMC/SD/SD HC Card** menu.

**Notes:**
- The FAT access functions are not re-entrant. They must not be called from interrupt service routines.
- Currently the FAT access functions support only the DOS short 8.3 file name format. Long file names are not supported.
- The file/directory names are encoded using 8bit ASCII, unicode characters are not supported.
- Before beeing accessed using the FAT functions, the MMC/SD/SD HC card must be partitioned and formated to FAT12, FAT16 or FAT32 system on a PC.

The following helper data types are defined in **ff.h**:

- The **FRESULT** type is used for returning the result of the FAT access functions:

```
typedef enum
      {
      FR_OK = 0,           /* (0) Succeeded */
      FR_DISK_ERR,         /* (1) A hard error occured in the low level
                                  disk I/O layer */
      FR_INT_ERR,          /* (2) Assertion failed */
      FR_NOT_READY,        /* (3) The physical drive doesn't work */
      FR_NO_FILE,          /* (4) Could not find the file */
      FR_NO_PATH,          /* (5) Could not find the path */
      FR_INVALID_NAME,     /* (6) The path name format is invalid */
      FR_DENIED,           /* (7) Acces denied due to prohibited access or
                                  directory full */
      FR_EXIST,            /* (8) Acces denied due to prohibited access */
      FR_INVALID_OBJECT,   /* (9) The file/directory object is invalid */
      FR_WRITE_PROTECTED,  /* (10) The physical drive is write protected */
      FR_INVALID_DRIVE,    /* (11) The logical drive number is invalid */
      FR_NOT_ENABLED,      /* (12) The volume has no work area */
      FR_NO_FILESYSTEM,    /* (13) There is no valid FAT volume */
      FR_MKFS_ABORTED,     /* (14) f_mkfs() aborted due to a parameter
                                   error */
      FR_TIMEOUT,          /* (15) Could not access the volume within
                                   the defined period */
      FR_INVALID_PARAMETER=19 /* (19) Given parameter is invalid */
      } FRESULT;
```

• The **FATFS** type structure is used for holding the work area associated with each logical drive volume:

```
typedef struct _FATFS_
      {
      unsigned char     fs_type;    /* FAT sub type */
      unsigned char     drive;      /* Physical drive number */
      unsigned char     csize;      /* Number of sectors per cluster */
      unsigned char     n_fats;     /* Number of FAT copies */
      unsigned char     wflag;      /* win[] dirty flag (1:must be written
                                       back) */
      unsigned short    id;         /* File system mount ID */
      unsigned short    n_rootdir;  /* Number of root directory entries
                                       (0 on FAT32) */
      unsigned char     fsi_flag;   /* fsinfo dirty flag
                                       (1:must be written back) */
      unsigned long     last_clust; /* Last allocated cluster */
      unsigned long     free_clust; /* Number of free clusters */
      unsigned long     fsi_sector; /* fsinfo sector */
      unsigned long     cdir;       /* Current directory (0:root)*/
      unsigned long     sects_fat;  /* Sectors per fat */
      unsigned long     max_clust;  /* Maximum cluster# + 1.
                                       Number of clusters is max_clust-2 */
      unsigned long     fatbase;    /* FAT start sector */
      unsigned long     dirbase;    /* Root directory start sector
                                      (Cluster# on FAT32) */
      unsigned long     database;   /* Data start sector */
      unsigned long     winsect;    /* Current sector appearing in the
                                       win[] */
      unsigned char     win[512];   /* Disk access window for Directory/FAT
                                       */
      } FATFS;
```

A **FATFS** type object is allocated by the **f_mount** function for each logical drive.

- The **FIL** type structure is used to hold the state of an open file:

```
typedef struct _FIL_
      {
      FATFS*   fs;                  /* Pointer to the owner file system
                                       object */
      unsigned short    id;         /* Owner file system mount ID */
      unsigned char     flag;       /* File status flags */
      unsigned char     csect;      /* Sector address in the cluster */
      unsigned long     fptr;       /* File R/W pointer */
      unsigned long     fsize;      /* File size */
      unsigned long     org_clust;  /* File start cluster */
      unsigned long     curr_clust; /* Current cluster */
      unsigned long     dsect;      /* Current data sector */
      unsigned long     dir_sect;   /* Sector containing the directory
                                       entry */
      unsigned char*    dir_ptr;    /* Pointer to the directory entry in
                                       the window */
      unsigned char     buf[512];   /* File R/W buffer */
      } FIL;
```

This structure is initialized by the **f_open** and discarded by the **f_close** functions.

- The **FILINFO** type structure is used to hold the information returned by the **f_stat** and **f_readdir** functions:

```
typedef struct _FILINFO_
      {
      unsigned long     fsize;      /* File size */
      unsigned short    fdate;      /* Last modified date */
      unsigned short    ftime;      /* Last modified time */
      unsigned char     fattrib;    /* Attribute */
      char  fname[13];  /* Short file name (DOS 8.3 format) */
      } FILINFO;
```

The **fdate** structure member indicates the date when the file was modified or the directory was created.
It has the following format:
bits 0:4 - Day: 1...31
bits 5:8 - Month: 1...12
bits 9:15 - Year starting with 1980: 0...127

The **ftime** structure member indicates the time when the file was modified or the directory was created.
It has the following format:
bits 0:4 - Second/2: 0...29
bits 5:10 - Minute: 0...59
bits 11:15 - Hour: 0...23

The **fattrib** structure member indicates the file or directory attributes combination defined by the following macros:

AM_RDO - Read Only attribute
AM_HID - Hidden attribute
AM_SYS - System attribute
AM_VOL - Volume attribute
AM_DIR - Directory attribute
AM_ARC - Archive attribute
AM_MASK - Mask of all defined attributes.

- The **DIR** type structure is used for holding directory information returned by the **f_opendir** and **f_readdir** functions:

```
typedef struct _DIR_
      {
      FATFS*      fs;           /* Pointer to the owner file system object
*/
      unsigned short    id;     /* Owner file system mount ID */
      unsigned short    index;  /* Current read/write index number */
      unsigned long     sclust; /* Table start cluster (0:Static table) */
      unsigned long     clust;  /* Current cluster */
      unsigned long     sect;   /* Current sector */
      unsigned char*    dir;    /* Pointer to the current SFN entry in the
                                   win[] */
      unsigned char*    fn;     /* Pointer to the SFN (in/out)
                                   {file[8],ext[3],status[1]} */
      } DIR;
```

The FAT access functions require the presence of a Real Time Clock in the system, in order to be able to set the time stamp of files or directories.
The following pointers to functions, that allow reading the time and date from the Real Time Clock, are declared in the **ff.h** header file:

**void (*prtc_get_time) (unsigned char *hour, unsigned char *min, unsigned char *sec)**

pointer to a Real Time Clock function used for reading time.

**void (*prtc_get_date) (unsigned char *date, unsigned char *month, unsigned int *year)**

pointer to a Real Time Clock function used for reading date.

On program startup these pointers need to be initialized to point to the appropriate RTC functions, like in the following example:

```
/* FAT on MMC/SD/SD HC card support */
#include <ff.h>

/* include the PCF8563 functions
   The I2C bus connections and bit rate must be specified in the
   Project|Configure|C Compiler|Libraries|I2C menu */
#include <PCF8563.h>

void main(void)
{
/* init the PCF8563 RTC */
rtc_init(0,RTC_CLKOUT_OFF,RTC_TIMER_OFF);

/* init the pointer to the RTC function used for reading time */
prtc_get_time=
    (void (*)(unsigned char *,unsigned char *,unsigned char *))
    rtc_get_time;

/* init the pointer to the RTC function used for reading time */
prtc_get_date=
    (void (*)(unsigned char *,unsigned char *,unsigned int *))
    rtc_get_date;

/* follows the rest of the program */
/* ... */
}
```

**Notes:**

- If the return type of the RTC functions is different from **void**, like required by the **prtc_get_time** and **prtc_get_date** pointer declarations, then casting to the appropriate type must be performed, like in the above example.
- If the system doesn't have a Real Time Clock, then these pointers must not be initialized at program startup. In this situation, they will be automatically initialized to NULL in the FAT access library and all files or directories created or modified by the FAT access functions will have the time stamp: January 1, 2009 00:00:00.

The FAT access functions are:

**FRESULT f_mount(unsigned char vol, FATFS *fs)**

allocates/deallocates a work area of memory for a logical drive volume.
This function must be called first before any other FAT access function.
In order to deallocate a work area associated with a logical drive, a NULL pointer must be passed as **fs.**

**Note:** This function only initializes the work area, no physical disk access is performed at this stage. The effective volume mount is performed on first file access after the function was called or after a media change.

*Parameters:*

**vol** specifies the logical drive number (0...9).
**fs** is a pointer to the FATFS type data structure associated with the logical drive that must be allocated/deallocated.

*Return Values:*

FR_OK - success
FR_INVALID_DRIVE - the drive number is invalid.

**FRESULT f_open(FIL* fp, const char* path, unsigned char mode)**

creates a file object **FIL** structure which will be used for accessing the file. The file read/write pointer is set to the start of the file.

*Parameters:*

**fp** points to the **FIL** type structure to be created. After the **f_open** function succeeds, this structure can be used by the other functions to access the file.
**path** points to a RAM based NULL terminated char string that represents the path name for the file to be created or opened.
The path name has the following format:

[logical_drive_number:][/][directory_name/]file_name

Examples:

*file.txt* - a file located in the current directory (specified previously by the **f_chdir** function) on the current drive (specified previously by the **f_chdrive** function).

*/file.txt* - a file located in the root directory of the current drive.

*0:file.txt* - a file located in the current directory (specified previously by the **f_chdir** function) on the logical drive 0.

*0:/* - the root directory of logical drive 0.

*0:/file.txt* - a file located in the root directory of logical drive 0.

. - current directory.

.. - parent directory of the current directory.

The file_name must have the DOS 8.3 short file name format.

**mode** is the file access type and open method, represented by a combination of the flags specified by the following macros:

FA_READ - Read access to the object. Data can be read from the file. For read-write access it must be combined with FA_WRITE.
FA_WRITE - Write access to the object. Data can be written to the file. For read-write access it must be combined with FA_READ.
FA_OPEN_EXISTING - Opens the file. If the file doesn't exist, the function will fail.
FA_OPEN_ALWAYS - If the file exists, it will be opened. If the file doesn't exist, it will be first created and then opened.
FA_CREATE_NEW - Creates a new file. If the file already exists, the function will fail.
FA_CREATE_ALWAYS - Creates a new file. If the file already exists, it will be overwritten and its size set to 0.

*Return values:*

FR_OK - success.
FR_NO_FILE - couldn't find the file.
FR_NO_PATH - couldn't find the path.
FR_INVALID_NAME - the file name is invalid.
FR_INVALID_DRIVE - the drive number is invalid.
FR_EXIST - the file already exists.
FR_DENIED - file access was denied because one of the following reasons:
- trying to open a read-only file in write mode
- file couldn't be created because a file with the same name or read-only attribute already exists
- file couldn't be created because the directory table or disk are full.

FR_NOT_READY - no disk access was possible due to missing media or other reason.
FR_WRITE_PROTECTED - opening in write mode or creating a file was not possible because the media is write protected.
FR_DISK_ERR - the function failed because of a physical disk access function failure.
FR_INT_ERR - the function failed due to a wrong FAT structure or an internal error.
FR_NOT_ENABLED - the logical drive was not mounted with **f_mount**.
FR_NO_FILESYSTEM - there is no valid FAT partition on the disk.

**FRESULT f_read(FIL* fp, void* buff, unsigned int btr, unsigned int* br)**

reads data from a file previously opened with **f_open**.
After the function is executed, the file read/write pointer advances with the number of bytes read from the file.

*Parameters:*

**fp** points to the **FIL** type structure that contains the file parameters. This structure must have been previously initialized by calling the **f_open** function.
**buff** points to a byte buffer array, located in RAM, that will hold the data read from the file. The size of the buffer must be large enough so that the data will fit in.
**btr** specifies the number of bytes to be read from the file.
**br** points to an unsigned int variable that will hold the number of bytes of data effectively read from the file. On function success, if the number of effectively read bytes is smaller than the **btr** value, then the file read/write pointer reached the end of the file.

*Return values:*

FR_OK - success.
FR_DENIED - file access was denied because it was opened in write-only mode.
FR_NOT_READY - no disk access was possible due to missing media or other reason.
FR_DISK_ERR - the function failed because of a physical disk access function failure.
FR_INT_ERR - the function failed due to a wrong FAT structure or an internal error.
FR_INVALID_OBJECT - the file was not opened with **f_open**.

**FRESULT f_write(FIL* fp, const void* buff, unsigned int btw, unsigned int* bw)**

writes data to a file previously opened with **f_open**.
After the function is executed, the file read/write pointer advances with the number of bytes written to the file.

*Parameters:*

**fp** points to the **FIL** type structure that contains the file parameters. This structure must have been previously initialized by calling the **f_open** function.
**buff** points to a byte buffer array, located in RAM, that holds the data to be written to the file.
**btw** specifies the number of bytes to be written to the file.
**bw** points to an unsigned int variable that will hold the number of bytes of data effectively written to the file.

*Return values:*

FR_OK - success.
FR_DENIED - file access was denied because it was opened in read-only mode.
FR_NOT_READY - no disk access was possible due to missing media or other reason.
FR_DISK_ERR - the function failed because of a physical disk access function failure.
FR_INT_ERR - the function failed due to a wrong FAT structure or an internal error.
FR_INVALID_OBJECT - the file was not opened with **f_open**.

**FRESULT f_lseek(FIL* fp, unsigned long ofs)**

moves the file read/write pointer of a file previously opened with **f_open**.
In write-mode, this function can be also used to extend the file size, by moving the file read/write pointer past the end of the file. On success the value of the **fptr** member of the **FIL** structure, pointed by **fp**, must be checked to see if the file read/write pointer effectively advanced to the correct position and the drive didn't get full.
In read-mode, trying to advance the file read/write pointer past the end, will limit its position to the end of the file. In this case the **fptr** member of the **FIL** structure, pointed by **fp**, will hold the size of the file.

*Parameters:*

**fp** points to the **FIL** type structure that contains the file parameters. This structure must have been previously initialized by calling the **f_open** function.
**ofs** represents the byte position where the file read/write pointer must be placed starting with the begining of the file.

*Return values:*

FR_OK - success.
FR_NOT_READY - no disk access was possible due to missing media or other reason.
FR_DISK_ERR - the function failed because of a physical disk access function failure.
FR_INT_ERR - the function failed due to a wrong FAT structure or an internal error.
FR_INVALID_OBJECT - the file was not opened with **f_open**.

**FRESULT f_truncate(FIL* fp)**

truncates the file's size to the current position of the file read/write pointer.
If the read/write pointer is already at the end of the file, the function will have no effect.

*Parameter:*

**fp** points to the **FIL** type structure that contains the file parameters. This structure must have been previously initialized by calling the **f_open** function.

*Return values:*

FR_OK - success.
FR_DENIED - file access was denied because it was opened in read-only mode.
FR_NOT_READY - no disk access was possible due to missing media or other reason.
FR_DISK_ERR - the function failed because of a physical disk access function failure.
FR_INT_ERR - the function failed due to a wrong FAT structure or an internal error.
FR_INVALID_OBJECT - the file was not opened with **f_open**.

**FRESULT f_close(FIL* fp)**

closes a file previously opened using **f_open**.
If any data was written to the file, the cached information is written to the disk.
After the function succeeded, the **FIL** type structure pointed by **fp**, is not valid anymore and the RAM memory allocated for it can be released.
If the file was opened in read-only mode, the memory allocated for the **FIL** type structure, pointed by **fp**, can be released without the need for previously calling the **f_close** function.

*Parameter:*

**fp** points to the **FIL** type structure that contains the file parameters. This structure must have been previously initialized by calling the **f_open** function.

*Return values:*

FR_OK - success.
FR_NOT_READY - no disk access was possible due to missing media or other reason.
FR_DISK_ERR - the function failed because of a physical disk access function failure.
FR_INT_ERR - the function failed due to a wrong FAT structure or an internal error.
FR_INVALID_OBJECT - the file was not opened with **f_open**.

**FRESULT f_sync(FIL* fp)**

flushes the cached data when writing a file.
This function is useful for applications when a file is opened for a long time in write mode.
Calling **f_sync** periodically or right after **f_write** minimizes the risk of data loss due to power failure or media removal from the drive.

**Note:** There is no need to call **f_sync** before **f_close**, as the later also performs a write cache flush.

*Parameter:*

**fp** points to the **FIL** type structure that contains the file parameters. This structure must have been previously initialized by calling the **f_open** function.

*Return values:*

FR_OK - success.
FR_NOT_READY - no disk access was possible due to missing media or other reason.
FR_DISK_ERR - the function failed because of a physical disk access function failure.
FR_INT_ERR - the function failed due to a wrong FAT structure or an internal error.
FR_INVALID_OBJECT - the file was not opened with **f_open**.

**FRESULT f_opendir(DIR* dj, const char* path)**

opens an existing directory and initializes the **DIR** type structure that holds directory information, which may be used by other FAT access functions.
The memory allocated for the **DIR** type structure may be de-allocated at any time.

*Parameters:*

**dj** points to the **DIR** type structure that must be initialized.
**path** points to a RAM based NULL terminated char string that represents the path name for the directory to be opened.

*Return values:*

FR_OK - success.
FR_NO_PATH - couldn't find the path.
FR_INVALID_NAME - the directory name is invalid.
FR_INVALID_DRIVE - the drive number is invalid.
FR_NOT_READY - no disk access was possible due to missing media or other reason.
FR_DISK_ERR - the function failed because of a physical disk access function failure.
FR_INT_ERR - the function failed due to a wrong FAT structure or an internal error.
FR_NOT_ENABLED - the logical drive was not mounted with **f_mount**.
FR_NO_FILESYSTEM - there is no valid FAT partition on the disk.

**FRESULT f_readdir(DIR* dj, FILINFO* fno)**

sequentially reads directory entries.
In order to read all the items in a directory this function must be called repeatedly.
When all items were read, the function will return a empty NULL char string in the **fname** member of the **FILINFO** structure, without any error.

**Note:** The "." and ".." directory entries are not filtered and will appear in the read entries.

*Parameters:*

**dj** points to the **DIR** type structure that holds directory information previously initialized by calling the **f_opendir** function.
**fno** points to the **FILINFO** type structure that will hold the file information for a read directory entry. If a NULL pointer is passed as **fno**, the directory entry read process will start from the begining.

*Return values:*

FR_OK - success.
FR_NOT_READY - no disk access was possible due to missing media or other reason.
FR_DISK_ERR - the function failed because of a physical disk access function failure.
FR_INT_ERR - the function failed due to a wrong FAT structure or an internal error.
FR_NO_FILESYSTEM - there is no valid FAT partition on the disk.

Example:

```
/* ATmega128 I/O register definitions */
#include <mega128.h>
/* FAT on MMC/SD/SD HC card support */
#include <ff.h>
/* printf */
#include <stdio.h>
/* string functions */
#include <string.h>

/* Timer1 overflow interrupt frequency [Hz] */
#define T1_OVF_FREQ 100
/* Timer1 clock prescaler value */
#define T1_PRESC 1024L
/* Timer1 initialization value after overflow */
#define T1_INIT (0x10000L-(_MCU_CLOCK_FREQUENCY_/(T1_PRESC*T1_OVF_FREQ)))

/* USART Baud rate */
#define BAUD_RATE 19200
#define BAUD_INIT (_MCU_CLOCK_FREQUENCY_/(BAUD_RATE*16L)-1)

/* 100Hz timer interrupt generated by ATmega128 Timer1 overflow */
interrupt [TIM1_OVF] void timer_comp_isr(void)
{
/* re-initialize Timer1 */
TCNT1H=T1_INIT>>8;
TCNT1L=T1_INIT&0xFF;
/* MMC/SD/SD HC card access low level timing function */
disk_timerproc();
}
```

```
/* error message list */
flash char * flash error_msg[]=
{
"", /* not used */
"FR_DISK_ERR",
"FR_INT_ERR",
"FR_INT_ERR",
"FR_NOT_READY",
"FR_NO_FILE",
"FR_NO_PATH",
"FR_INVALID_NAME",
"FR_DENIED",
"FR_EXIST",
"FR_INVALID_OBJECT",
"FR_WRITE_PROTECTED",
"FR_INVALID_DRIVE",
"FR_NOT_ENABLED",
"FR_NO_FILESYSTEM",
"FR_MKFS_ABORTED",
"FR_TIMEOUT"
};

/* display error message and stop */
void error(FRESULT res)
{
if ((res>=FR_DISK_ERR) && (res<=FR_TIMEOUT))
   printf("ERROR: %p\r\n",error_msg[res]);
/* stop here */
while(1);
}

/* will hold file/directory information returned by f_readdir*/
FILINFO file_info;

/* recursively scan directory entries and display them */
FRESULT directory_scan(char *path)
{
/* will hold the directory information */
DIR directory;
/* FAT function result */
FRESULT res;
int i;
```

```
if ((res=f_opendir(&directory,path))==FR_OK)
   {
   while (((res=f_readdir(&directory,&file_info))==FR_OK) &&
         file_info.fname[0])
         {
         /* display file/directory name and associated information */
         printf("%c%c%c%c%c %02u/%02u/%u %02u:%02u:%02u %9lu"
                "  %s/%s\r\n",
                (file_info.fattrib & AM_DIR) ? 'D' : '-',
                (file_info.fattrib & AM_RDO) ? 'R' : '-',
                (file_info.fattrib & AM_HID) ? 'H' : '-',
                (file_info.fattrib & AM_SYS) ? 'S' : '-',
                (file_info.fattrib & AM_ARC) ? 'A' : '-',
                file_info.fdate & 0x1F,(file_info.fdate >> 5) & 0xF,
                (file_info.fdate >> 9)+1980,
                file_info.ftime >> 11,(file_info.ftime >> 5) & 0x3F,
                (file_info.ftime & 0xF) << 1,
                file_info.fsize,path,file_info.fname);
         if (file_info.fattrib & AM_DIR)
            {
            /* it's a subdirectory */
            /* make sure to skip past "." and ".." when recursing */
            if (file_info.fname[0]!='.')
               {
               i=strlen(path);
               /* append the subdirectory name to the path */
               if (path[i-1]!='/') strcatf(path,"/");
               strcat(path,file_info.fname);
               /* scan subdirectory */
               res=directory_scan(path);
               /* restore the old path name */
               path[i]=0;
               /* remove any eventual '/' from the end of the path */
               --i;
               if (path[i]=='/') path[i]=0;
               /* stop if an error occured */
               if (res!=FR_OK) break;
               }
            }
          }
   }
return res;
}

void main(void)
{
/* FAT function result */
FRESULT res;
/* will hold the information for logical drive 0: */
FATFS drive;
/* root directory path */
char path[256]="0:/";
```

```
/* initialize Timer1 overflow interrupts in Mode 0 (Normal) */
TCCR1A=0x00;
/* clkio/1024 */
TCCR1B=(1<<CS12)|(1<<CS10);
/* timer overflow interrupts will occur with 100Hz frequency */
TCNT1H=T1_INIT>>8;
TCNT1L=T1_INIT&0xFF;
/* enable Timer1 overflow interrupt */
TIMSK=1<<TOIE1;

/* initialize the USART0 TX, 8N1, Baud rate: 19200 */
UCSR0A=0;
UCSR0B=1<<TXEN0;
UCSR0C=(1<<UCSZ01)|(1<<UCSZ00);
UBRR0H=BAUD_INIT>>8;
UBRR0L=BAUD_INIT&0xFF;

/* globally enable interrupts */
#asm("sei")

printf("Directory listing for root of logical drive 0:\r\n");

/* mount logical drive 0: */
if ((res=f_mount(0,&drive))==FR_OK)
   printf("Logical drive 0: mounted OK\r\n");
else
   /* an error occured, display it and stop */
   error(res);

/* repeateadly read directory entries and display them */
if ((res=directory_scan(path))!=FR_OK)
   /* if an error occured, display it and stop */
   error(res);

/* stop here */
while(1);
}
```

**Note:** When compiling the above example, make sure that the **(s)printf Features** option in the **Project|Configure|C Compiler|Code Generation** menu will be set to: **long, width**.
This will ensure that the unsigned long int file sizes will be displayed correctly by the **printf** function.

**FRESULT f_stat(const char* path, FILINFO* fno)**

gets the file or directory status in a **FILINFO** type structure.

*Parameters:*

**path** points to a RAM based NULL terminated char string that represents the path name for the file or directory.
**fno** points to the **FILINFO** type structure that will hold the status information.

*Return values:*

FR_OK - success.
FR_NO_FILE - couldn't find the file.
FR_NO_PATH - couldn't find the path.
FR_INVALID_NAME - the file name is invalid.
FR_INVALID_DRIVE - the drive number is invalid.
FR_NOT_READY - no disk access was possible due to missing media or other reason.
FR_WRITE_PROTECTED - opening in write mode or creating a file was not possible because the media is write protected.
FR_DISK_ERR - the function failed because of a physical disk access function failure.
FR_INT_ERR - the function failed due to a wrong FAT structure or an internal error.
FR_NOT_ENABLED - the logical drive was not mounted with **f_mount**.
FR_NO_FILESYSTEM - there is no valid FAT partition on the disk.

**FRESULT f_getfree(const char* path, unsigned long* nclst, FATFS** fatfs)**

gets the number of free clusters on the drive.

*Parameters:*

**path** points to a RAM based NULL terminated char string that represents the path name of the root directory of the logical drive.
**nclst** points to an unsigned long int variable that will hold the number of free clusters.
**fatfs** points to a pointer to the FATFS type structure associated with the logical drive.

*Return values:*

FR_OK - success.
FR_INVALID_DRIVE - the drive number is invalid.
FR_NOT_READY - no disk access was possible due to missing media or other reason.
FR_WRITE_PROTECTED - opening in write mode or creating a file was not possible because the media is write protected.
FR_DISK_ERR - the function failed because of a physical disk access function failure.
FR_INT_ERR - the function failed due to a wrong FAT structure or an internal error.
FR_NOT_ENABLED - the logical drive was not mounted with **f_mount**.
FR_NO_FILESYSTEM - there is no valid FAT partition on the disk.

The **csize** member of the FATFS structure represents the number of sectors/cluster, so the free size in bytes can be calculated using the example below:

```
/* ATmega128 I/O register definitions */
#include <mega128.h>
/* FAT on MMC/SD/SD HC card support */
#include <ff.h>
/* printf */
#include <stdio.h>

/* Timer1 overflow interrupt frequency [Hz] */
#define T1_OVF_FREQ 100
/* Timer1 clock prescaler value */
#define T1_PRESC 1024L
/* Timer1 initialization value after overflow */
#define T1_INIT (0x10000L-(_MCU_CLOCK_FREQUENCY_/(T1_PRESC*T1_OVF_FREQ)))

/* USART Baud rate */
#define BAUD_RATE 19200
#define BAUD_INIT (_MCU_CLOCK_FREQUENCY_/(BAUD_RATE*16L)-1)

/* 100Hz timer interrupt generated by ATmega128 Timer1 overflow */
interrupt [TIM1_OVF] void timer_comp_isr(void)
{
/* re-initialize Timer1 */
TCNT1H=T1_INIT>>8;
TCNT1L=T1_INIT&0xFF;
/* card access low level timing function */
disk_timerproc();
}

/* error message list */
flash char * flash error_msg[]=
{
"", /* not used */
"FR_DISK_ERR",
"FR_INT_ERR",
"FR_INT_ERR",
"FR_NOT_READY",
"FR_NO_FILE",
"FR_NO_PATH",
"FR_INVALID_NAME",
"FR_DENIED",
"FR_EXIST",
"FR_INVALID_OBJECT",
"FR_WRITE_PROTECTED",
"FR_INVALID_DRIVE",
"FR_NOT_ENABLED",
"FR_NO_FILESYSTEM",
"FR_MKFS_ABORTED",
"FR_TIMEOUT"
};
```

```
/* display error message and stop */
void error(FRESULT res)
{
if ((res>=FR_DISK_ERR) && (res<=FR_TIMEOUT))
   printf("ERROR: %p\r\n",error_msg[res]);
/* stop here */
while(1);
}

void main(void)
{
/* FAT function result */
FRESULT res;
/* will hold the information for logical drive 0: */
FATFS fat;
/* pointer to the FATFS type structure */
FATFS *pfat;
/* number of free clusters on logical drive 0:*/
unsigned long free_clusters;
/* number of free kbytes on logical drive 0: */
unsigned long free_kbytes;
/* root directory path for logical drive 0: */
char root_path[]="0:/";

/* initialize Timer1 overflow interrupts in Mode 0 (Normal) */
TCCR1A=0x00;
/* clkio/1024 */
TCCR1B=(1<<CS12)|(1<<CS10);
/* timer overflow interrupts will occur with 100Hz frequency */
TCNT1H=T1_INIT>>8;
TCNT1L=T1_INIT&0xFF;
/* enable Timer1 overflow interrupt */
TIMSK=1<<TOIE1;

/* initialize the USART0 TX, 8N1, Baud rate: 19200 */
UCSR0A=0;
UCSR0B=1<<TXEN0;
UCSR0C=(1<<UCSZ01)|(1<<UCSZ00);
UBRR0H=BAUD_INIT>>8;
UBRR0L=BAUD_INIT&0xFF;

/* globally enable interrupts */
#asm("sei")

/* point to the FATFS structure that holds
information for the logical drive 0: */
pfat=&fat;

/* mount logical drive 0: */
if ((res=f_mount(0,pfat))==FR_OK)
   printf("Logical drive 0: mounted OK\r\n");
else
   /* an error occured, display it and stop */
   error(res);
```

```
/* get the number of free clusters */
if ((res=f_getfree(root_path,&free_clusters,&pfat))==FR_OK)
   {
   /* calculate the number of free bytes */
   free_kbytes=free_clusters *
               /* cluster size in sectors */
               pfat->csize
               /* divide by 2 to obtain the sector size in kbytes
               512 (sector size in bytes)/1024 = 1/2 kbytes
               we need to do the division by 2 directly,
               in order to prevent unsigned long multiplication
               overflow for 8GB+ SD HC cards */
               /2;
   /* display the number of free kbytes */
   printf("Free space on logical drive 0: %lu kbytes\r\n",free_kbytes);
   }
else
   /* an error occured, display it and stop */
   error(res);

/* stop here */
while(1);
}
```

**Note:** When compiling the above example, make sure that the **(s)printf Features** option in the **Project|Configure|C Compiler|Code Generation** menu will be set to: **long, width**.
This will ensure that the unsigned long int file sizes will be displayed correctly by the **printf** function.

**FRESULT f_mkdir (const char* path)**

creates a new directory.

*Parameter:*

**path** points to a RAM based NULL terminated char string that represents the path name for the directory to be created.

*Return values:*

FR_OK - success.
FR_NO_PATH - couldn't find the path.
FR_INVALID_NAME - the directory name is invalid.
FR_INVALID_DRIVE - the drive number is invalid.
FR_EXIST - the directory already exists.
FR_DENIED - the directory couldn't be created because the directory table or disk are full.
FR_NOT_READY - no disk access was possible due to missing media or other reason.
FR_WRITE_PROTECTED - creating the directory was not possible because the media is write protected.
FR_DISK_ERR - the function failed because of a physical disk access function failure.
FR_INT_ERR - the function failed due to a wrong FAT structure or an internal error.
FR_NOT_ENABLED - the logical drive was not mounted with **f_mount**.
FR_NO_FILESYSTEM - there is no valid FAT partition on the disk.

**FRESULT f_unlink(const char* path)**

deletes an existing file or directory.

*Parameter:*

**path** points to a RAM based NULL terminated char string that represents the path name for the file or directory to be deleted.

*Return values:*

FR_OK - success.
FR_NO_FILE - couldn't find the file or directory.
FR_NO_PATH - couldn't find the path.
FR_INVALID_NAME - the file or directory name is invalid.
FR_INVALID_DRIVE - the drive number is invalid.
FR_DENIED - access was denied because one of the following reasons:
- file or directory read-only attribute is set
- the directory is not empty.

FR_NOT_READY - no disk access was possible due to missing media or other reason.
FR_WRITE_PROTECTED - the media in the drive is write protected.
FR_DISK_ERR - the function failed because of a physical disk access function failure.
FR_INT_ERR - the function failed due to a wrong FAT structure or an internal error.
FR_NOT_ENABLED - the logical drive was not mounted with **f_mount**.
FR_NO_FILESYSTEM - there is no valid FAT partition on the disk.

**FRESULT f_chmod (const char* path, unsigned char value, unsigned char mask)**

changes the attribute of a file or directory.

*Parameters:*

**path** points to a RAM based NULL terminated char string that represents the path name for the file or directory.
**value** specifies the new combination of attribute flags to be set.
**mask** specifies the combination of which attribute flags must be changed.

The attribute is obtained by combining the following predefined macros:

AM_RDO - Read Only attribute flag
AM_HID - Hidden attribute flag
AM_SYS - System attribute flag
AM_ARC - Archive attribute flag

using the **|** binary OR operator.

*Return values:*

FR_OK - success.
FR_NO_FILE - couldn't find the file or directory.
FR_NO_PATH - couldn't find the path.
FR_INVALID_NAME - the file or directory name is invalid.
FR_INVALID_DRIVE - the drive number is invalid.
FR_NOT_READY - no disk access was possible due to missing media or other reason.
FR_WRITE_PROTECTED - the media in the drive is write protected.
FR_DISK_ERR - the function failed because of a physical disk access function failure.
FR_INT_ERR - the function failed due to a wrong FAT structure or an internal error.
FR_NOT_ENABLED - the logical drive was not mounted with **f_mount**.
FR_NO_FILESYSTEM - there is no valid FAT partition on the disk.

**FRESULT f_utime (const char* path, const FILINFO* fno)**

changes the time stamp of a file or directory.

*Parameters:*

**path** points to a RAM based NULL terminated char string that represents the path name for the file or directory.
**fno** points to the **FILINFO** type structure that holds the file information and has the time stamp to be set contained in the **fdate** and **ftime** members.

*Return values:*

FR_OK - success.
FR_NO_FILE - couldn't find the file or directory.
FR_NO_PATH - couldn't find the path.
FR_INVALID_NAME - the file or directory name is invalid.
FR_INVALID_DRIVE - the drive number is invalid.
FR_NOT_READY - no disk access was possible due to missing media or other reason.
FR_WRITE_PROTECTED - the media in the drive is write protected.
FR_DISK_ERR - the function failed because of a physical disk access function failure.
FR_INT_ERR - the function failed due to a wrong FAT structure or an internal error.
FR_NOT_ENABLED - the logical drive was not mounted with **f_mount**.
FR_NO_FILESYSTEM - there is no valid FAT partition on the disk.

Example:

```
/* ATmega128 I/O register definitions */
#include <mega128.h>

/* FAT on MMC/SD/SD HC card support */
#include <ff.h>

/* printf */
#include <stdio.h>

/* include the PCF8563 functions
   The I2C bus connections and bit rate must be specified in the
   Project|Configure|C Compiler|Libraries|I2C menu */
#include <PCF8563.h>

/* Timer1 overflow interrupt frequency [Hz] */
#define T1_OVF_FREQ 100
/* Timer1 clock prescaler value */
#define T1_PRESC 1024L
/* Timer1 initialization value after overflow */
#define T1_INIT (0x10000L-(_MCU_CLOCK_FREQUENCY_/(T1_PRESC*T1_OVF_FREQ)))

/* USART Baud rate */
#define BAUD_RATE 19200
#define BAUD_INIT (_MCU_CLOCK_FREQUENCY_/(BAUD_RATE*16L)-1)

/* FAT function result */
FRESULT res;
/* number of bytes written/read to the file */
unsigned int nbytes;
/* will hold the information for logical drive 0: */
FATFS fat;
/* will hold the file information */
FIL file;
/* will hold file attributes, time stamp information */
FILINFO finfo;
/* file path */
char path[]="0:/file.txt";
/* text to be written to the file */
char text[]="I like CodeVisionAVR!";
/* file read buffer */
char buffer[256];

/* 100Hz timer interrupt generated by ATmega128 Timer1 overflow */
interrupt [TIM1_OVF] void timer_comp_isr(void)
{
/* re-initialize Timer1 */
TCNT1H=T1_INIT>>8;
TCNT1L=T1_INIT&0xFF;
/* card access low level timing function */
disk_timerproc();
}
```

```
/* error message list */
flash char * flash error_msg[]=
{
"", /* not used */
"FR_DISK_ERR",
"FR_INT_ERR",
"FR_INT_ERR",
"FR_NOT_READY",
"FR_NO_FILE",
"FR_NO_PATH",
"FR_INVALID_NAME",
"FR_DENIED",
"FR_EXIST",
"FR_INVALID_OBJECT",
"FR_WRITE_PROTECTED",
"FR_INVALID_DRIVE",
"FR_NOT_ENABLED",
"FR_NO_FILESYSTEM",
"FR_MKFS_ABORTED",
"FR_TIMEOUT"
};

/* display error message and stop */
void error(FRESULT res)
{
if ((res>=FR_DISK_ERR) && (res<=FR_TIMEOUT))
   printf("ERROR: %p\r\n",error_msg[res]);
/* stop here */
while(1);
}

/* display file's attribute, size and time stamp */
void display_status(char *file_name)
{
if ((res=f_stat(file_name,&finfo))==FR_OK)
   printf("File: %s, Attributes: %c%c%c%c%c\r\n"
          "Date: %02u/%02u/%u, Time: %02u:%02u:%02u\r\n"
          "Size: %lu bytes\r\n",
          finfo.fname,
          (finfo.fattrib & AM_DIR) ? 'D' : '-',
          (finfo.fattrib & AM_RDO) ? 'R' : '-',
          (finfo.fattrib & AM_HID) ? 'H' : '-',
          (finfo.fattrib & AM_SYS) ? 'S' : '-',
          (finfo.fattrib & AM_ARC) ? 'A' : '-',
          finfo.fdate & 0x1F, (finfo.fdate >> 5) & 0xF,
          (finfo.fdate >> 9) + 1980,
          (finfo.ftime >> 11), (finfo.ftime >> 5) & 0x3F,
          (finfo.ftime & 0xF) << 1,
          finfo.fsize);
else
   /* an error occured, display it and stop */
   error(res);
}
```

```
void main(void)
{
/* initialize Timer1 overflow interrupts in Mode 0 (Normal) */
TCCR1A=0x00;
/* clkio/1024 */
TCCR1B=(1<<CS12)|(1<<CS10);
/* timer overflow interrupts will occur with 100Hz frequency */
TCNT1H=T1_INIT>>8;
TCNT1L=T1_INIT&0xFF;
/* enable Timer1 overflow interrupt */
TIMSK=1<<TOIE1;

/* initialize the USART0 TX, 8N1, Baud rate: 19200 */
UCSR0A=0;
UCSR0B=1<<TXEN0;
UCSR0C=(1<<UCSZ01)|(1<<UCSZ00);
UBRR0H=BAUD_INIT>>8;
UBRR0L=BAUD_INIT&0xFF;

/* init the PCF8563 RTC */
rtc_init(0,RTC_CLKOUT_OFF,RTC_TIMER_OFF);

/* init the pointer to the RTC function used for reading time */
prtc_get_time=
    (void (*)(unsigned char *,unsigned char *,unsigned char *))
    rtc_get_time;

/* init the pointer to the RTC function used for reading date */
prtc_get_date=
    (void (*)(unsigned char *,unsigned char *,unsigned int *))
    rtc_get_date;

/* globally enable interrupts */
#asm("sei")


/* mount logical drive 0: */
if ((res=f_mount(0,&fat))==FR_OK)
   printf("Logical drive 0: mounted OK\r\n");
else
   /* an error occured, display it and stop */
   error(res);

printf("%s \r\n",path);

/* create a new file in the root of drive 0:
   and set write access mode */
if ((res=f_open(&file,path,FA_CREATE_ALWAYS | FA_WRITE))==FR_OK)
   printf("File %s created OK\r\n",path);
else
   /* an error occured, display it and stop */
   error(res);

/* write some text to the file,
   without the NULL string terminator sizeof(data)-1 */
if ((res=f_write(&file,text,sizeof(text)-1,&nbytes))==FR_OK)
   printf("%u bytes written of %u\r\n",nbytes,sizeof(text)-1);
else
   /* an error occured, display it and stop */
   error(res);
```

```
/* close the file */
if ((res=f_close(&file))==FR_OK)
   printf("File %s closed OK\r\n",path);
else
   /* an error occured, display it and stop */
   error(res);

/* open the file in read mode */
if ((res=f_open(&file,path,FA_READ))==FR_OK)
   printf("File %s opened OK\r\n",path);
else
   /* an error occured, display it and stop */
   error(res);

/* read and display the file's content.
   make sure to leave space for a NULL terminator
   in the buffer, so maximum sizeof(buffer)-1 bytes can be read */
if ((res=f_read(&file,buffer,sizeof(buffer)-1,&nbytes))==FR_OK)
   {
   printf("%u bytes read\r\n",nbytes);
   /* NULL terminate the char string in the buffer */
   buffer[nbytes+1]=NULL;
   /* display the buffer contents */
   printf("Read text: \"%s\"\r\n",buffer);
   }
else
   /* an error occured, display it and stop */
   error(res);

/* close the file */
if ((res=f_close(&file))==FR_OK)
   printf("File %s closed OK\r\n",path);
else
   /* an error occured, display it and stop */
   error(res);

/* display file's attribute, size and time stamp */
display_status(path);

/* change file's attributes, set the file to be Read-Only */
if ((res=f_chmod(path,AM_RDO,AM_RDO))==FR_OK)
   printf("Read-Only attribute set OK\r\n",path);
else
   /* an error occured, display it and stop */
   error(res);
```

```
/* change file's time stamp */
#define DAY (6)
#define MONTH (3)
#define YEAR (2000)
#define SECOND (0)
#define MINUTE (40)
#define HOUR (14)

finfo.fdate=DAY | (MONTH<<5) | ((YEAR-1980)<<9);
finfo.ftime=(SECOND>>1) | (MINUTE<<5) | (HOUR<<11);
if ((res=f_utime(path,&finfo))==FR_OK)
   printf("New time stamp %02u/%02u/%u %02u:%02u:%02u set OK\r\n",
   DAY,MONTH,YEAR,HOUR,MINUTE,SECOND);
else
   /* an error occured, display it and stop */
   error(res);

/* display file's new attribute and time stamp */
display_status(path);

/* change file's attributes, clear the Read-Only attribute */
if ((res=f_chmod(path,0,AM_RDO))==FR_OK)
   printf("Read-Only attribute cleared OK\r\n",path);
else
   /* an error occured, display it and stop */
   error(res);

/* display file's new attribute and time stamp */
display_status(path);

/* stop here */
while(1);
}
```

**Note:** When compiling the above example, make sure that the **(s)printf Features** option in the **Project|Configure|C Compiler|Code Generation** menu will be set to: **int, width**.

**FRESULT f_rename(const char* path_old, const char* path_new)**

renames a file or directory.
If the new path contains a different directory than the old path, the file will be also moved to this directory.
The logical drive is determined by the old path, it must not be specified in the new path.

*Parameters:*

**path_old** points to a RAM based NULL terminated char string that represents the path name for the file or directory to be renamed.
**path_new** points to a RAM based NULL terminated char string that represents the new path name for the file or directory.

*Return values:*

FR_OK - success.
FR_NO_FILE - couldn't find the file or directory.
FR_NO_PATH - couldn't find the path.
FR_INVALID_NAME - the file or directory name is invalid.
FR_INVALID_DRIVE - the drive number is invalid.
FR_NOT_READY - no disk access was possible due to missing media or other reason.
FR_EXIST - the file or directory already exists.
FR_DENIED - the file or directory couldn't be created or moved from any reason..
FR_WRITE_PROTECTED - the media in the drive is write protected.
FR_DISK_ERR - the function failed because of a physical disk access function failure.
FR_INT_ERR - the function failed due to a wrong FAT structure or an internal error.
FR_NOT_ENABLED - the logical drive was not mounted with **f_mount**.
FR_NO_FILESYSTEM - there is no valid FAT partition on the disk.

**FRESULT f_chdir(const char* path)**

changes the current directory of the current logical drive.
When the drive is mounted, the current directory is the root directory.

**Note:** After **f_chdir** is called, all subsequent file access function operations will be performed by default in the new current directory, if no other directory is specified when calling these functions.

*Parameter:*

**path** points to a RAM based NULL terminated char string that represents the path name for the directory to go.

*Return values:*

FR_OK - success.
FR_NO_PATH - couldn't find the path.
FR_INVALID_NAME - the file or directory name is invalid.
FR_INVALID_DRIVE - the drive number is invalid.
FR_NOT_READY - no disk access was possible due to missing media or other reason.
FR_DISK_ERR - the function failed because of a physical disk access function failure.
FR_INT_ERR - the function failed due to a wrong FAT structure or an internal error.
FR_NOT_ENABLED - the logical drive was not mounted with **f_mount**.
FR_NO_FILESYSTEM - there is no valid FAT partition on the disk.

**FRESULT f_chdrive(unsigned char drv)**

changes the current logical drive.The initial logical drive is 0.

**Note:** After **f_chdrive** is called, all subsequent file/directory access function operations will be performed by default on the new logical drive, if no other drive is specified when calling these functions.

*Parameter:*

**drv** specifies the logical drive number (0...9) to be set as current drive.

*Return values:*

FR_OK - success.
FR_INVALID_DRIVE - the drive number is invalid.

**FRESULT f_mkfs(unsigned char drv, unsigned char number_fats,
unsigned short bytes_cluster)**

creates a single primary partition on the drive and formats it.

*Parameter:*

**drv** specifies the logical drive number (0...9) to be formatted.
**number_fats** specifies the number of FATs to be created on the drive during formatting.
This parameter may take the values 1 or 2.
**bytes_cluster** specifies the number of bytes allocated for one cluster.

*Return values:*

FR_OK - success.
FR_INVALID_DRIVE - the drive number is invalid.
FR_INVALID_PARAMETER – the **number_fats** or **bytes_cluster** parameters are not valid
FR_NOT_READY - no disk access was possible due to missing media or other reason.
FR_DISK_ERR - the function failed because of a physical disk access function failure.
FR_NOT_ENABLED - the logical drive was not mounted with **f_mount**.
FR_WRITE_PROTECTED - the media in the drive is write protected.
FR_MKFS_ABORTED – the formatted disk size is too small or the **bytes_cluster** parameter is not correct for the disk size.

## 5.22 Peripheral Chips Functions

The CodeVisionAVR C Compiler has a rich set of library functions for accessing a large variety of peripheral chips using the $I^2C$ (TWI), 1 Wire and SPI buses.

### 5.22.1 Philips PCF8563 Real Time Clock Functions

These functions are intended for easy interfacing between C programs and the PCF8563 $I^2C$ bus real time clock (RTC), using both the hardware TWI and software bit-banged $I^2C$ bus functions.
One of the following header files, located in the .\INC subdirectory must be **#include** -d before using these functions:

- **pcf8563_twi.h** for hardware TWI
- **pcf8563.h** for software bit-banged $I^2C$ bus.

The appropriate header files for hardware TWI (**twi.h** or **twix.h**) or software bit-banged $I^2C$ bus (**i2c.h**) functions prototypes are automatically **#include** -d with the **pcf8563_twi.h** or **pcf8563.h**.

The **Project|Configure|C Compiler|Libraries|I2C** menu must be used for specifying the I/O port allocation of the **SCL** and **SDA** signals, along the **bit rate** of the **SCL** clock, for bit-banged $I^2C$ bus (not the hardware TWI) communication.

**Note: For proper operation, the PCF8563 Functions require the presence of 3.3k - 4.7k pull-up resistors to +5V (+3.3V for XMEGA devices) on the SCL and SDA signals.**

The PCF8563 Functions are:

**void rtc_init(unsigned char ctrl2, unsigned char clkout, unsigned char timer_ctrl)**

this function initializes the PCF8563 chip.
Before calling this function, the TWI hardware, respectively $I^2C$ bus, must be initialized by calling the **twi_master_init** (**twi_init**, **twi_master_init** and **pcf8563_twi_init** for XMEGA devices), respectively **i2c_init** functions.
This is the first function that must be called prior to using the other PCF8563 Functions.
Only one PCF8563 chip can be connected to the $I^2C$ bus.

The **ctrl2** parameter specifies the initialization value for the PCF8563 **Control/Status 2** register.
The **pcf8563.h** header file defines the following macros which allow the easy setting of the **ctrl2** parameter:

- **RTC_TIE_ON** sets the **Control/Status 2** register bit **TIE** to 1
- **RTC_AIE_ON** sets the **Control/Status 2** register bit **AIE** to 1
- **RTC_TP_ON** sets the **Control/Status 2** register bit **TI/TP** to 1

These macros can be combined using the **|** operator in order to set more bits to 1.

The **clkout** parameter specifies the initialization value for the PCF8563 **CLKOUT Frequency** register.
The **pcf8563.h** header file defines the following macros which allow the easy setting of the **clkout** parameter:

- **RTC_CLKOUT_OFF** disables the generation of pulses on the PCF8563 CLKOUT output
- **RTC_CLKOUT_1** generates 1Hz pulses on the PCF8563 CLKOUT output
- **RTC_CLKOUT_32** generates 32Hz pulses on the PCF8563 CLKOUT output
- **RTC_CLKOUT_1024** generates 1024Hz pulses on the PCF8563 CLKOUT output
- **RTC_CLKOUT_32768** generates 32768Hz pulses on the PCF8563 CLKOUT output.

The **timer_ctrl** parameter specifies the initialization value for the PCF8563 **Timer Control** register.
The **pcf8563.h** header file defines the following macros which allow the easy setting of the **timer_ctrl** parameter:

- **RTC_TIMER_OFF** disables the PCF8563 Timer countdown
- **RTC_TIMER_CLK_1_60** sets the PCF8563 Timer countdown clock frequency to 1/60Hz
- **RTC_TIMER_CLK_1** sets the PCF8563 Timer countdown clock frequency to 1Hz
- **RTC_TIMER_CLK_64** sets the PCF8563 Timer countdown clock frequency to 64Hz
- **RTC_TIMER_CLK_4096** sets the PCF8563 Timer countdown clock frequency to 4096Hz.

Refer to the PCF8563 data sheet for more information.

**void pcf8563_twi_init(TWI_MASTER_INFO_t *ptwim)**

this function is used to initialize the PCF8563 library's internal variables when using the **TWI Functions for Master Mode Operation for XMEGA Devices**.
It is not used for non-XMEGA devices.

The **ptwim** parameter must point to a **TWI_MASTER_INFO_t** structure variable, that is used to hold the information required by the TWI module when operating in master mode.
The **pcf8563_twi_init** function must be called before **rtc_init**.
Refer to the supplied example at the end of this chapter for more details.

**unsigned char rtc_read(unsigned char address)**

this function reads the byte stored in a PCF8563 register at **address**.

**void rtc_write(unsigned char address, unsigned char data)**

this function stores the byte **data** in the PCF8563 register at **address**.

**unsigned char rtc_get_time(unsigned char *hour, unsigned char *min, unsigned char *sec)**

this function returns the current time measured by the RTC .
The ***hour**, ***min** and ***sec** pointers must point to the variables that must receive the values of hour, minutes and seconds.
The function return the value 1 if the read values are correct.
If the function returns 0 then the chip supply voltage has dropped below the Vlow value and the time values are incorrect.

**void rtc_set_time(unsigned char hour, unsigned char min, unsigned char sec)**

this function sets the current time of the RTC .
The **hour**, **min** and **sec** parameters represent the values of hour, minutes and seconds.

**void rtc_get_date(unsigned char *day, unsigned char *month, unsigned *year)**

this function returns the current date measured by the RTC .
The ***day**, ***month** and ***year** pointers must point to the variables that must receive the values of day, month and year.

**void rtc_set_date(unsigned char day, unsigned char month, unsigned year)**

this function sets the current date of the RTC .

**void rtc_alarm_off(void)**

this function disables the RTC alarm function.

**void rtc_alarm_on(void)**

this function enables the RTC alarm function.

**void rtc_get_alarm(unsigned char *day, unsigned char *hour, unsigned char *min)**

this function returns the alarm time and date of the RTC.
The ***day**, ***hour** and ***min** pointers must point to the variables that must receive the values of day, hour and minutes.

**void rtc_set_alarm(unsigned char day, unsigned char hour, unsigned char min)**

this function sets the alarm time and date of the RTC.
The **day**, **hour** and **min** parameters represent the values of day, hours and minutes.
If **day** is set to 0, then this parameter will be ignored.
After calling this function the alarm will be turned off. It must be enabled using the **rtc_alarm_on** function.

**void rtc_set_timer(unsigned char val)**

this function sets the countdown value of the PCF8563 Timer.

PCF8563 example using the hardware **TWI Functions for Master Mode Operation for non-XMEGA Devices**:

```
/* include the PCF8563 functions for TWI */
#include <pcf8563_twi.h>

/* include the LCD Functions
   The connections must be specified in the
   Project|Configure|C Compiler|Libraries|Alphanumeric LCD menu */
#include <alcd.h>

/* include the prototype for sprintf */
#include <stdio.h>

/* include the prototypes for the delay functions */
#include <delay.h>

char display_buffer[17]; /* LCD display buffer for 1 line */

void main(void)
{
unsigned char hour,min,sec,day,month;
unsigned int year;

/* initialize the LCD, 2 rows by 16 columns */
lcd_init(16);

/* initialize the TWI in master mode with 100 kHz bit rate */
twi_master_init(100);

/* enable interrupts so that TWI can be used */
#asm("sei")
```

```
/* initialize the RTC,
   Timer interrupt enabled,
   Alarm interrupt enabled,
   CLKOUT frequency=1Hz
   Timer clock frequency=1Hz */
rtc_init(RTC_TIE_ON | RTC_AIE_ON,RTC_CLKOUT_1,RTC_TIMER_CLK_1);

rtc_set_time(12,0,0); /* set time 12:00:00 */

rtc_set_date(1,2,2011); /* set date 1/02/2011 */

/* display the time and date continuously */
while (1)
      {
      /* read the time from the RTC */
      rtc_get_time(&hour,&min,&sec);

      /* read the date from the RTC */
      rtc_get_date(&day,&month,&year);

      /* display the time on the LCD */
      sprintf(display_buffer,"Time: %2d:%02d:%02d\n",hour,min,sec);
      lcd_clear();
      lcd_puts(display_buffer);

      /* display the date on the LCD */
      sprintf(display_buffer,"Date: %2d/%02d/%d",day,month,year);
      lcd_puts(display_buffer);

      delay_ms(500); /* 0.5 second delay */
      }
}
```

PCF8563 example using the hardware **TWI Functions for Master Mode Operation for XMEGA Devices**.
The chip is connected to the TWI of PORTD (TWID) of an ATxmega128A1.

```
/* include the PCF8563 functions for TWI */
#include <pcf8563_twi.h>

/* include the LCD Functions
   The connections must be specified in the
   Project|Configure|C Compiler|Libraries|Alphanumeric LCD menu */
#include <alcd.h>

/* include the prototype for sprintf */
#include <stdio.h>

/* include the prototypes for the delay functions */
#include <delay.h>

char display_buffer[17]; /* LCD display buffer for 1 line */

/* structure that holds information used by the TWID master
   for performing a TWI bus transaction */
TWI_MASTER_INFO_t twid_master;
```

```
/* interrupt service routine for TWID master */
interrupt [TWID_TWIM_vect] void twid_master_isr(void)
{
twi_master_int_handler(&twid_master);
}

/* TWI clock rate [Hz] */
#define TWI_CLK_RATE 100000

void main(void)
{
unsigned char hour,min,sec,day,month;
unsigned int year;

/* initialize the LCD, 2 rows by 16 columns */
lcd_init(16);

/* general TWID initialization: no external driver interface,
   no SDA hold time */
twi_init(&TWID,false,false);

/* enable and initialize the TWID master interrupt level: low */
twi_master_init(&twid_master,&TWID,TWI_MASTER_INTLVL_LO_gc,
TWI_BAUD_REG(_MCU_CLOCK_FREQUENCY_,TWI_CLK_RATE));

/* enable the Low interrupt level */
PMIC.CTRL|=PMIC_LOLVLEN_bm;

/* set the PCF8563 functions to use TWID */
pcf8563_twi_init(&twid_master);

/* enable interrupts so that TWI can be used */
#asm("sei")

/* initialize the RTC,
   Timer interrupt enabled,
   Alarm interrupt enabled,
   CLKOUT frequency=1Hz
   Timer clock frequency=1Hz */
rtc_init(RTC_TIE_ON | RTC_AIE_ON,RTC_CLKOUT_1,RTC_TIMER_CLK_1);

rtc_set_time(12,0,0); /* set time 12:00:00 */

rtc_set_date(1,2,2011); /* set date 1/02/2011 */

/* display the time and date continuously */
while (1)
      {
      /* read the time from the RTC */
      rtc_get_time(&hour,&min,&sec);

      /* read the date from the RTC */
      rtc_get_date(&day,&month,&year);

      /* display the time on the LCD */
      sprintf(display_buffer,"Time: %2d:%02d:%02d\n",hour,min,sec);
      lcd_clear();
      lcd_puts(display_buffer);
```

```
      /* display the date on the LCD */
      sprintf(display_buffer,"Date: %2d/%02d/%d",day,month,year);
      lcd_puts(display_buffer);

      delay_ms(500); /* 0.5 second delay */
      }
}
```

PCF8563 example using the **Software Bit-Banged I2C Bus Functions**:

```
/* include the PCF8563 functions for bit-banged I2C
   The I2C bus connections and bit rate must be specified in the
   Project|Configure|C Compiler|Libraries|I2C menu */
#include <pcf8563.h>

/* include the LCD Functions
   The connections must be specified in the
   Project|Configure|C Compiler|Libraries|Alphanumeric LCD menu */
#include <alcd.h>

/* include the prototype for sprintf */
#include <stdio.h>

/* include the prototypes for the delay functions */
#include <delay.h>

char display_buffer[17]; /* LCD display buffer for 1 line */

void main(void)
{
unsigned char hour,min,sec,day,month;
unsigned int year;

/* initialize the LCD, 2 rows by 16 columns */
lcd_init(16);

/* initialize the bit-banged I2C functions */
i2c_init();

/* initialize the RTC: Timer interrupt enabled, Alarm interrupt enabled,
   CLKOUT frequency=1Hz, Timer clock frequency=1Hz */
rtc_init(RTC_TIE_ON | RTC_AIE_ON,RTC_CLKOUT_1,RTC_TIMER_CLK_1);

rtc_set_time(12,0,0); /* set time 12:00:00 */

rtc_set_date(1,2,2011); /* set date 1/02/2011 */

/* display the time and date continuously */
while (1)
      {
      /* read the time from the RTC */
      rtc_get_time(&hour,&min,&sec);

      /* read the date from the RTC */
      rtc_get_date(&day,&month,&year);
```

```
      /* display the time on the LCD */
      sprintf(display_buffer,"Time: %2d:%02d:%02d\n",hour,min,sec);
      lcd_clear();
      lcd_puts(display_buffer);

      /* display the date on the LCD */
      sprintf(display_buffer,"Date: %2d/%02d/%d",day,month,year);
      lcd_puts(display_buffer);

      delay_ms(500); /* 0.5 second delay */
      }
}
```

## 5.22.2 Philips PCF8583 Real Time Clock Functions

These functions are intended for easy interfacing between C programs and the PCF8583 $I^2C$ bus real time clock (RTC), using both the hardware TWI and software bit-banged $I^2C$ bus functions.
One of the following header files, located in the .\INC subdirectory must be **#include** -d before using these functions:

- **pcf8583_twi.h** for hardware TWI
- **pcf8583.h** for software bit-banged $I^2C$ bus.

The appropriate header files for hardware TWI (**twi.h** or **twix.h**) or software bit-banged $I^2C$ bus (**i2c.h**) functions prototypes are automatically **#include** -d with the **pcf8583_twi.h** or **pcf8583.h**.

The **Project|Configure|C Compiler|Libraries|I2C** menu must be used for specifying the I/O port allocation of the **SCL** and **SDA** signals, along the **bit rate** of the **SCL** clock, for bit-banged $I^2C$ bus (<u>not the hardware TWI</u>) communication.

**Note: For proper operation, the PCF8583 Functions require the presence of 3.3k - 4.7k pull-up resistors to +5V (+3.3V for XMEGA devices) on the SCL and SDA signals.**

The PCF8583 Functions are:

**void rtc_init(unsigned char chip, unsigned char dated_alarm)**

this function initializes the PCF8583 chip.
Before calling this function, the TWI hardware, respectively $I^2C$ bus, must be initialized by calling the **twi_master_init** (**twi_init**, **twi_master_ini**t and **pcf8583_twi_init** for XMEGA devices), respectively **i2c_init** functions.
This is the first function that must be called prior to using the other PCF8583 Functions.

If more then one chip is connected to the $I^2C$ bus, then the function must be called for each one, specifying accordingly the function parameter **chip**.
Maximum 2 PCF8583 chips can be connected to the $I^2C$ bus, their chip address can be 0 or 1.
The **dated_alarm** parameter specifies if the RTC alarm takes in account both the time and date (dated_alarm=1), or only the time (dated_alarm=0).
Refer to the PCF8583 data sheet for more information.
After calling this function the RTC alarm is disabled.

**void pcf8583_twi_init(TWI_MASTER_INFO_t *ptwim)**

this function is used to initialize the PCF8583 library's internal variable swhen using the **TWI Functions for Master Mode Operation for XMEGA Devices**.
It is not used for non-XMEGA devices.

The **ptwim** parameter must point to a **TWI_MASTER_INFO_t** structure variable, that is used to hold the information required by the TWI module when operating in master mode.
The **pcf8583_twi_init** function must be called before **rtc_init**.
Refer to the supplied example at the end of this chapter for more details.

**unsigned char rtc_read(unsigned char chip, unsigned char address)**

this function reads the byte stored in the PCF8583 SRAM.

**void rtc_write(unsigned char chip, unsigned char address, unsigned char data)**

this function stores the byte **data** in the PCF8583 SRAM.
When writing to the SRAM the user must take in account that locations at addresses 10h and 11h are used for storing the current year value.

**unsigned char rtc_get_status(unsigned char chip)**

this function returns the value of the PCF8583 control/status register.
By calling this function the global variables **rtc_status** and **rtc_alarm** are automatically updated.
The **rtc_status** variable holds the value of the PCF8583 control/status register.
The **rtc_alarm** variable takes the value 1 if an RTC alarm occurred.

**void rtc_get_time(unsigned char chip, unsigned char *hour, unsigned char *min, unsigned char *sec, unsigned char *hsec)**

this function returns the current time measured by the RTC.
The ***hour**, ***min**, ***sec** and ***hsec** pointers must point to the variables that must receive the values of hour, minutes, seconds and hundreds of a second.

**void rtc_set_time(unsigned char chip, unsigned char hour, unsigned char min, unsigned char sec, unsigned char hsec)**

this function sets the current time of the RTC.
The **hour**, **min**, **sec** and **hsec** parameters represent the values of hour, minutes, seconds and hundreds of a second.

**void rtc_get_date(unsigned char chip, unsigned char *day, unsigned char *month, unsigned *year)**

this function returns the current date measured by the RTC.
The ***day**, ***month** and ***year** pointers must point to the variables that must receive the values of day, month and year.

**void rtc_set_date(unsigned char chip, unsigned char day, unsigned char month, unsigned year)**

this function sets the current date of the RTC.

**void rtc_alarm_off(unsigned char chip)**

this function disables the RTC alarm function.

**void rtc_alarm_on(unsigned char chip)**

this function enables the RTC alarm function.

**void rtc_get_alarm_time(unsigned char chip, unsigned char *hour, unsigned char *min, unsigned char *sec, unsigned char *hsec)**

this function returns the alarm time of the RTC.
The ***hour**, ***min**, ***sec** and ***hsec** pointers must point to the variables that must receive the values of hours, minutes, seconds and hundreds of a second.

**void rtc_set_alarm_time(unsigned char chip, unsigned char hour, unsigned char min, unsigned char sec, unsigned char hsec)**

this function sets the alarm time of the RTC.
The **hour**, **min**, **sec** and **hsec** parameters represent the values of hours, minutes, seconds and hundreds of a second.

**void rtc_get_alarm_date(unsigned char chip, unsigned char *day, unsigned char *month)**

this function returns the alarm date of the RTC.
The ***day** and ***month** pointers must point to the variables that must receive the values of date and month.

**void rtc_set_alarm_date(unsigned char chip, unsigned char day, unsigned char month)**

this function sets the alarm date of the RTC.

PCF8583 example using the hardware **TWI Functions for Master Mode Operation for non-XMEGA Devices**:

```
/* include the PCF8583 functions for TWI */
#include <pcf8583_twi.h>

/* include the LCD Functions
   The connections must be specified in the
   Project|Configure|C Compiler|Libraries|Alphanumeric LCD menu */
#include <alcd.h>

/* include the prototype for sprintf */
#include <stdio.h>

/* include the prototypes for the delay functions */
#include <delay.h>

char display_buffer[17]; /* LCD display buffer for 1 line */

void main(void)
{
unsigned char hour,min,sec,hsec,day,month;
unsigned int year;

/* initialize the LCD, 2 rows by 16 columns */
lcd_init(16);

/* initialize the TWI in master mode with 100 kHz bit rate */
twi_master_init(100);

/* enable interrupts so that TWI can be used */
#asm("sei")

/* initialize the RTC 0, no dated alarm */
rtc_init(0,0);

rtc_set_time(0,12,0,0,0); /* set time 12:00:00.00 on RTC 0*/

rtc_set_date(0,1,2,2011); /* set date 1/02/2011 on RTC 0 */
```

```
/* display the time and date continuously */
while (1)
      {
      /* read the time from the RTC 0 */
      rtc_get_time(0,&hour,&min,&sec,&hsec);

      /* read the date from the RTC 0 */
      rtc_get_date(0,&day,&month,&year);

      /* display the time on the LCD */
      sprintf(display_buffer,"Time: %2d:%02d:%02d\n",hour,min,sec);
      lcd_clear();
      lcd_puts(display_buffer);

      /* display the date on the LCD */
      sprintf(display_buffer,"Date: %2d/%02d/%d",day,month,year);
      lcd_puts(display_buffer);

      delay_ms(500); /* 0.5 second delay */
      }
}
```

PCF8583 example using the hardware **TWI Functions for for Master Mode Operation XMEGA Devices**.
The chip is connected to the TWI of PORTD (TWID) of an ATxmega128A1.

```
/* include the PCF8583 functions for TWI */
#include <pcf8583_twi.h>

/* include the LCD Functions
   The connections must be specified in the
   Project|Configure|C Compiler|Libraries|Alphanumeric LCD menu */
#include <alcd.h>

/* include the prototype for sprintf */
#include <stdio.h>

/* include the prototypes for the delay functions */
#include <delay.h>

char display_buffer[17]; /* LCD display buffer for 1 line */

/* structure that holds information used by the TWID master
   for performing a TWI bus transaction */
TWI_MASTER_INFO_t twid_master;

/* interrupt service routine for TWID master */
interrupt [TWID_TWIM_vect] void twid_master_isr(void)
{
twi_master_int_handler(&twid_master);
}

/* TWI clock rate [Hz] */
#define TWI_CLK_RATE 100000
```

```
void main(void)
{
unsigned char hour,min,sec,hsec,day,month;
unsigned int year;

/* initialize the LCD, 2 rows by 16 columns */
lcd_init(16);

/* general TWID initialization
   no external driver interface
   no SDA hold time */
twi_init(&TWID,false,false);

/* enable and initialize the TWID master
   interrupt level: low */
twi_master_init(&twid_master,&TWID,TWI_MASTER_INTLVL_LO_gc,
TWI_BAUD_REG(_MCU_CLOCK_FREQUENCY_,TWI_CLK_RATE));

/* enable the Low interrupt level */
PMIC.CTRL|=PMIC_LOLVLEN_bm;

/* set the PCF8583 functions to use TWID */
pcf8583_twi_init(&twid_master);

/* enable interrupts so that TWI can be used */
#asm("sei")

/* initialize the RTC 0, no dated alarm */
rtc_init(0,0);

rtc_set_time(0,12,0,0,0); /* set time 12:00:00.00 on RTC 0*/

rtc_set_date(0,1,2,2011); /* set date 1/02/2011 on RTC 0 */

/* display the time and date continuously */
while (1)
      {
      /* read the time from the RTC 0 */
      rtc_get_time(0,&hour,&min,&sec,&hsec);

      /* read the date from the RTC 0 */
      rtc_get_date(0,&day,&month,&year);

      /* display the time on the LCD */
      sprintf(display_buffer,"Time: %2d:%02d:%02d\n",hour,min,sec);
      lcd_clear();
      lcd_puts(display_buffer);

      /* display the date on the LCD */
      sprintf(display_buffer,"Date: %2d/%02d/%d",day,month,year);
      lcd_puts(display_buffer);

      delay_ms(500); /* 0.5 second delay */
      }
}
```

PCF8583 example using the **Software Bit-Banged I2C Bus Functions**:

```
/* include the PCF8583 functions for bit-banged I2C
   The I2C bus connections and bit rate must be specified in the
   Project|Configure|C Compiler|Libraries|I2C menu */
#include <pcf8583.h>

/* include the LCD Functions
   The connections must be specified in the
   Project|Configure|C Compiler|Libraries|Alphanumeric LCD menu */
#include <alcd.h>

/* include the prototype for sprintf */
#include <stdio.h>

/* include the prototypes for the delay functions */
#include <delay.h>

char display_buffer[17]; /* LCD display buffer for 1 line */

void main(void)
{
unsigned char hour,min,sec,hsec,day,month;
unsigned int year;

/* initialize the LCD, 2 rows by 16 columns */
lcd_init(16);

/* initialize the bit-banged I2C functions */
i2c_init();

/* initialize the RTC 0, no dated alarm */
rtc_init(0,0);

rtc_set_time(0,12,0,0,0); /* set time 12:00:00.00 on RTC 0*/

rtc_set_date(0,1,2,2011); /* set date 1/02/2011 on RTC 0 */

/* display the time and date continuously */
while (1)
      {
      /* read the time from the RTC 0 */
      rtc_get_time(0,&hour,&min,&sec,&hsec);

      /* read the date from the RTC 0 */
      rtc_get_date(0,&day,&month,&year);

      /* display the time on the LCD */
      sprintf(display_buffer,"Time: %2d:%02d:%02d\n",hour,min,sec);
      lcd_clear();
      lcd_puts(display_buffer);

      /* display the date on the LCD */
      sprintf(display_buffer,"Date: %2d/%02d/%d",day,month,year);
      lcd_puts(display_buffer);

      delay_ms(500); /* 0.5 second delay */
      }
}
```

## 5.22.3 Maxim/Dallas Semiconductor DS1302 Real Time Clock Functions

These functions are intended for easy interfacing between C programs and the DS1302 real time clock (RTC).
The prototypes for these functions are placed in the file **ds1302.h**, located in the .\INC subdirectory. This file must be **#include -d** before using the functions.

The DS1302 RTC Functions functions do not yet support the XMEGA chips.

Prior to **#include** -ing the **ds1302.h** file, you must declare which microcontroller port and port bits are used for communication with the DS1302.
Example:

```
/* the DS1302 is connected to ATmega8515 PORTB
   the IO signal is bit 3
   the SCLK signal is bit 4
   the RST signal is bit 5 */
#asm
    .equ __ds1302_port=0x18
    .equ __ds1302_io=3
    .equ __ds1302_sclk=4
    .equ __ds1302_rst=5
#endasm

/* now you can include the DS1302 Functions */
#include <ds1302.h>
```

**Note:** For XMEGA chips a virtual port must be mapped to the I/O port used for connecting to the DS1302 chip, and the address of the virtual port **VPORT*n*_OUT** register must be specified.

The DS1302 Functions are:

**void rtc_init(unsigned char tc_on, unsigned char diodes, unsigned char res)**

this function initializes the DS1302 chip.
This is the first function that must be called prior to using the other DS1302 Functions.
If the **tc_on** parameter is set to 1 then the DS1302's trickle charge function is enabled.
The **diodes** parameter specifies the number of diodes used when the trickle charge function is enabled. This parameter can take the value 1 or 2.
The **res** parameter specifies the value of the trickle charge resistor:

- 0 for no resistor
- 1 for a 2kΩ resistor
- 2 for a 4kΩ resistor
- 3 for a 8kΩ resistor.

Refer to the DS1302 data sheet for more information.

**unsigned char ds1302_read(unsigned char addr)**

this function reads a byte stored at address **addr** in the DS1302 registers or SRAM.

**void ds1302_write(unsigned char addr, unsigned char data)**

this function stores the byte **data** at address **addr** in the DS1302 registers or SRAM.

**void rtc_get_time(unsigned char *hour, unsigned char *min, unsigned char *sec)**

this function returns the current time measured by the RTC.
The ***hour**, ***min** and ***sec** pointers must point to the variables that must receive the values of hours, minutes and seconds.

**void rtc_set_time(unsigned char hour, unsigned char min, unsigned char sec)**

this function sets the current time of the RTC.
The **hour**, **min** and **sec** parameters represent the values of hour, minutes and seconds.

**void rtc_get_date(unsigned char *day, unsigned char *month, unsigned char *year)**

this function returns the current date measured by the RTC.
The ***day**, ***month** and ***year** pointers must point to the variables that must receive the values of day, month and year.

**void rtc_set_date(unsigned char day, unsigned char month, unsigned char year)**

this function sets the current date of the RTC.

Example program for a DS1302 connected to an ATmega8515 chip:

```
/* the DS1302 is connected to ATmega8515 PORTB
   the IO signal is bit 3
   the SCLK signal is bit 4
   the RST signal is bit 5 */
#asm
    .equ __ds1302_port=0x18 ; Address of the PORTB register
    .equ __ds1302_io=3
    .equ __ds1302_sclk=4
    .equ __ds1302_rst=5
#endasm

/* include the DS1302 */
#include <ds1302.h>

/* include the LCD Functions
   The connections must be specified in the
   Project|Configure|C Compiler|Libraries|Alphanumeric LCD menu */
#include <alcd.h>

/* include the prototype for sprintf */
#include <stdio.h>

/* include the prototypes for the delay functions */
#include <delay.h>

char display_buffer[17]; /* LCD display buffer for 1 line */

void main(void)
{
unsigned char hour,min,sec,day,month,year;

/* initialize the LCD, 2 rows by 16 columns */
lcd_init(16);
```

```
/* initialize the DS1302 RTC:
   use trickle charge,
   with 1 diode and 8K resistor */
rtc_init(1,1,3);

rtc_set_time(12,0,0); /* set time 12:00:00 */

rtc_set_date(1,2,12); /* set date 1/02/2013 */

/* display the time and date continuously */
while (1)
      {
      /* read the time from the RTC */
      rtc_get_time(&hour,&min,&sec);

      /* read the date from the RTC */
      rtc_get_date(&day,&month,&year);

      /* display the time on the LCD */
      sprintf(display_buffer,"Time: %2d:%02d:%02d\n",hour,min,sec);
      lcd_clear();
      lcd_puts(display_buffer);

      /* display the date on the LCD */
      sprintf(display_buffer,"Date: %2d/%02d/%d",day,month,2000+year);
      lcd_puts(display_buffer);

      delay_ms(500); /* 0.5 second delay */
      }
}
```

Example program for a DS1302 connected to an ATxmega128A1 chip on PORTA pins 0, 1 and 2.
The PORTA is mapped to the virtual port VPORT0.

```
/* the DS1302 is connected to ATxmega 128A1 VPORT0
   the IO signal is bit 0
   the SCLK signal is bit 1
   the RST signal is bit 2 */
#asm
    .equ __ds1302_port=0x11 ; Address of the VPORT0_OUT register
    .equ __ds1302_io=0
    .equ __ds1302_sclk=1
    .equ __ds1302_rst=2
#endasm

/* include the DS1302 */
#include <ds1302.h>

/* include the ATxmega128A1 I/O register definitions */
#include <io.h>

/* include the LCD Functions
   The connections must be specified in the
   Project|Configure|C Compiler|Libraries|Alphanumeric LCD menu */
#include <alcd.h>

/* include the prototype for sprintf */
#include <stdio.h>
```

```
/* include the prototypes for the delay functions */
#include <delay.h>

char display_buffer[17]; /* LCD display buffer for 1 line */

void main(void)
{
unsigned char hour,min,sec,day,month,year;

/* PORTA is mapped to VPORT0
   PORTB is mapped to VPORT1 */
PORTCFG.VPCTRLA=PORTCFG_VP1MAP_PORTB_gc | PORTCFG_VP0MAP_PORTA_gc;

/* initialize the LCD, 2 rows by 16 columns */
lcd_init(16);

/* initialize the DS1302 RTC:
   use trickle charge,
   with 1 diode and 8K resistor */
rtc_init(1,1,3);

rtc_set_time(12,0,0); /* set time 12:00:00 */

rtc_set_date(1,2,2011); /* set date 1/02/2011 */

/* display the time and date continuously */
while (1)
      {
      /* read the time from the RTC */
      rtc_get_time(&hour,&min,&sec);

      /* read the date from the RTC */
      rtc_get_date(&day,&month,&year);

      /* display the time on the LCD */
      sprintf(display_buffer,"Time: %2d:%02d:%02d\n",hour,min,sec);
      lcd_clear();
      lcd_puts(display_buffer);

      /* display the date on the LCD */
      sprintf(display_buffer,"Date: %2d/%02d/%d",day,month,2000+year);
      lcd_puts(display_buffer);

      delay_ms(500); /* 0.5 second delay */
      }
}
```

## 5.22.4 Maxim/Dallas Semiconductor DS1307 Real Time Clock Functions

These functions are intended for easy interfacing between C programs and the DS1307 $I^2C$ bus real time clock (RTC), using both the hardware TWI and software bit-banged $I^2C$ bus functions.
One of the following header files, located in the .\INC subdirectory must be **#include** -d before using these functions:

- **ds1307_twi.h** for hardware TWI
- **ds1307.h** for software bit-banged $I^2C$ bus.

The appropriate header files for hardware TWI (**twi.h** or **twix.h**) or software bit-banged $I^2C$ bus (**i2c.h**) functions prototypes are automatically **#include** -d with the **ds1307_twi.h** or **ds1307.h**.

The **Project|Configure|C Compiler|Libraries|I2C** menu must be used for specifying the I/O port allocation of the **SCL** and **SDA** signals, along the **bit rate** of the **SCL** clock, for bit-banged $I^2C$ bus (<u>not the hardware TWI</u>) communication.

**Note: For proper operation, the DS1307 Functions require the presence of 3.3k - 4.7k pull-up resistors to +5V (+3.3V for XMEGA devices) on the SCL and SDA signals.**

The DS1307 Functions are:

**void rtc_init(unsigned char rs, unsigned char sqwe, unsigned char out)**

this function initializes the DS1307 chip.
Before calling this function, the TWI hardware, respectively $I^2C$ bus, must be initialized by calling the **twi_master_init** (**twi_init**, **twi_master_ini**t and **ds1307_twi_init** for XMEGA devices), respectively **i2c_init** functions.
This is the first function that must be called prior to using the other DS1307 Functions.

The **rs** parameter specifies the value of the square wave output frequency on the SQW/OUT pin:

- 0 for 1 Hz
- 1 for 4096 Hz
- 2 for 8192 Hz
- 3 for 32768 Hz.

If the **sqwe** parameter is set to 1 then the square wave output on the SQW/OUT pin is enabled.
The **out** parameter specifies the logic level on the SQW/OUT pin when the square wave output is disabled (sqwe=0).
Refer to the DS1307 data sheet for more information.

**void ds1307_twi_init(TWI_MASTER_INFO_t *ptwim)**

this function is used to initialize the DS1307 library's internal variables when using the **TWI Functions for Master Mode Operation for XMEGA Devices**.
It is not used for non-XMEGA devices.

The **ptwim** parameter must point to a **TWI_MASTER_INFO_t** structure variable, that is used to hold the information required by the TWI module when operating in master mode.
The **ds1307_twi_init** function must be called before **rtc_init**.
Refer to the supplied example at the end of this chapter for more details.

**void rtc_get_time(unsigned char *hour, unsigned char *min, unsigned char *sec)**

this function returns the current time measured by the RTC.
The ***hour**, ***min** and ***sec** pointers must point to the variables that must receive the values of hours, minutes and seconds.

**void rtc_set_time(unsigned char hour, unsigned char min, unsigned char sec)**

this function sets the current time of the RTC.
The **hour**, **min** and **sec** parameters represent the values of hour, minutes and seconds.

**void rtc_get_date(unsigned char *week_day, unsigned char *day, unsigned char *month, unsigned char *year)**

this function returns the current date measured by the RTC.
The ***week_day**, ***day**, ***month** and ***year** pointers must point to the variables that must receive the values of day of week, day, month and year.

**void rtc_set_date(unsigned char week_day, unsigned char day, unsigned char month, unsigned char year)**

this function sets the current date of the RTC.

DS1307 example using the hardware **TWI Functions for Master Mode Operation for non-XMEGA Devices**:

```
/* include the DS1307 functions for TWI */
#include <ds1307_twi.h>

/* include the LCD Functions
   The connections must be specified in the
   Project|Configure|C Compiler|Libraries|Alphanumeric LCD menu */
#include <alcd.h>

/* include the prototype for sprintf */
#include <stdio.h>

/* include the prototypes for the delay functions */
#include <delay.h>

char display_buffer[17]; /* LCD display buffer for 1 line */

void main(void)
{
unsigned char hour,min,sec,week_day,day,month,year;

/* initialize the LCD, 2 rows by 16 columns */
lcd_init(16);

/* initialize the TWI in master mode with 100 kHz bit rate */
twi_master_init(100);

/* enable interrupts so that TWI can be used */
#asm("sei")

/* initialize the RTC
   square wave output is disabled
   SQW/OUT has logic state 0 */
rtc_init(0,0,0);

rtc_set_time(12,0,0); /* set time 12:00:00 */

rtc_set_date(2,1,2,11); /* set date Tuesday 1/02/2011 */
```

```
/* display the time and date continuously */
while (1)
      {
      /* read the time from the RTC */
      rtc_get_time(&hour,&min,&sec);

      /* read the date from the RTC */
      rtc_get_date(&week_day,&day,&month,&year);

      /* display the time on the LCD */
      sprintf(display_buffer,"Time: %2d:%02d:%02d\n",hour,min,sec);
      lcd_clear();
      lcd_puts(display_buffer);

      /* display the date on the LCD */
      sprintf(display_buffer,"Date: %2d/%02d/%d",day,month,2000+year);
      lcd_puts(display_buffer);

      delay_ms(500); /* 0.5 second delay */
      }
}
```

DS1307 example using the hardware **TWI Functions for Master Mode Operation for XMEGA Devices**.
The chip is connected to the TWI of PORTD (TWID) of an ATxmega128A1.

```
/* include the DS1307 functions for TWI */
#include <ds1307_twi.h>

/* include the LCD Functions
   The connections must be specified in the
   Project|Configure|C Compiler|Libraries|Alphanumeric LCD menu */
#include <alcd.h>

/* include the prototype for sprintf */
#include <stdio.h>

/* include the prototypes for the delay functions */
#include <delay.h>

char display_buffer[17]; /* LCD display buffer for 1 line */

/* structure that holds information used by the TWID master
   for performing a TWI bus transaction */
TWI_MASTER_INFO_t twid_master;

/* interrupt service routine for TWID master */
interrupt [TWID_TWIM_vect] void twid_master_isr(void)
{
twi_master_int_handler(&twid_master);
}

/* TWI clock rate [Hz] */
#define TWI_CLK_RATE 100000
```

```
void main(void)
{
unsigned char hour,min,sec,week_day,day,month,year;

/* initialize the LCD, 2 rows by 16 columns */
lcd_init(16);

/* general TWID initialization
   no external driver interface
   no SDA hold time */
twi_init(&TWID,false,false);

/* enable and initialize the TWID master
   interrupt level: low */
twi_master_init(&twid_master,&TWID,TWI_MASTER_INTLVL_LO_gc,
TWI_BAUD_REG(_MCU_CLOCK_FREQUENCY_,TWI_CLK_RATE));

/* enable the Low interrupt level */
PMIC.CTRL|=PMIC_LOLVLEN_bm;

/* set the DS1307 functions to use TWID */
ds1307_twi_init(&twid_master);

/* enable interrupts so that TWI can be used */
#asm("sei")

/* initialize the RTC
   square wave output is disabled
   SQW/OUT has logic state 0 */
rtc_init(0,0,0);

rtc_set_time(12,0,0); /* set time 12:00:00 */

rtc_set_date(2,1,2,11); /* set date Tuesday 1/02/2011 */

/* display the time and date continuously */
while (1)
      {
      /* read the time from the RTC */
      rtc_get_time(&hour,&min,&sec);

      /* read the date from the RTC */
      rtc_get_date(&week_day,&day,&month,&year);

      /* display the time on the LCD */
      sprintf(display_buffer,"Time: %2d:%02d:%02d\n",hour,min,sec);
      lcd_clear();
      lcd_puts(display_buffer);

      /* display the date on the LCD */
      sprintf(display_buffer,"Date: %2d/%02d/%d",day,month,2000+year);
      lcd_puts(display_buffer);

      delay_ms(500); /* 0.5 second delay */
      }
}
```

DS1307 example using the **Software Bit-Banged I2C Bus Functions**:

```
/* include the DS1307 functions for bit-banged I2C
   The I2C bus connections and bit rate must be specified in the
   Project|Configure|C Compiler|Libraries|I2C menu */
#include <ds1307.h>

/* include the LCD Functions
   The connections must be specified in the
   Project|Configure|C Compiler|Libraries|Alphanumeric LCD menu */
#include <alcd.h>

/* include the prototype for sprintf */
#include <stdio.h>

/* include the prototypes for the delay functions */
#include <delay.h>

char display_buffer[17]; /* LCD display buffer for 1 line */

void main(void)
{
unsigned char hour,min,sec,week_day,day,month,year;

/* initialize the LCD, 2 rows by 16 columns */
lcd_init(16);

/* initialize the bit-banged I2C functions */
i2c_init();

/* initialize the RTC
   square wave output is disabled
   SQW/OUT has logic state 0 */
rtc_init(0,0,0);

rtc_set_time(12,0,0); /* set time 12:00:00 */

rtc_set_date(2,1,2,11); /* set date Tuesday 1/02/2011 */

/* display the time and date continuously */
while (1)
      {
      /* read the time from the RTC */
      rtc_get_time(&hour,&min,&sec);

      /* read the date from the RTC */
      rtc_get_date(&week_day,&day,&month,&year);

      /* display the time on the LCD */
      sprintf(display_buffer,"Time: %2d:%02d:%02d\n",hour,min,sec);
      lcd_clear();
      lcd_puts(display_buffer);

      /* display the date on the LCD */
      sprintf(display_buffer,"Date: %2d/%02d/%d",day,month,2000+year);
      lcd_puts(display_buffer);

      delay_ms(500); /* 0.5 second delay */
      }
}
```

## 5.22.5 Maxim/Dallas Semiconductor DS1621 Thermometer/ Thermostat Functions

These functions are intended for easy interfacing between C programs and the DS1621 $I^2C$ bus thermometer/thermostat, using both the hardware TWI and software bit-banged $I^2C$ bus functions.
One of the following header files, located in the .\INC subdirectory must be **#include** -d before using these functions:

- **ds1621_twi.h** for hardware TWI
- **ds1621.h** for software bit-banged $I^2C$ bus.

The appropriate header files for hardware TWI (**twi.h** or **twix.h**) or software bit-banged $I^2C$ bus (**i2c.h**) functions prototypes are automatically **#include** -d with the **ds1621_twi.h** or **ds1621.h**.

The **Project|Configure|C Compiler|Libraries|I2C** menu must be used for specifying the I/O port allocation of the **SCL** and **SDA** signals, along the **bit rate** of the **SCL** clock, for bit-banged $I^2C$ bus (not the hardware TWI) communication.

**Note: For proper operation, the DS1621 Functions require the presence of 3.3k - 4.7k pull-up resistors to +5V (+3.3V for XMEGA devices) on the SCL and SDA signals.**

The DS1621 Functions are:

**void ds1621_init(unsigned char chip,signed char tlow,signed char thigh, unsigned char pol)**

this function initializes the DS1621 chip.
Before calling this function, the TWI hardware, respectively $I^2C$ bus, must be initialized by calling the **twi_master_init** (**twi_init**, **twi_master_init** and **ds1621_twi_init** for XMEGA devices), respectively **i2c_init** functions.
This is the first function that must be called prior to using the other DS1621 Functions.

If more then one chip is connected to the $I^2C$ bus, then the function must be called for each one, specifying accordingly the function parameter **chip**.
Maximum 8 DS1621 chips can be connected to the $I^2C$ bus, their **chip** address can be from 0 to 7.
Besides measuring temperature, the DS1621 functions also like a thermostat.
The Tout output becomes active when the temperature exceeds the **thigh** limit, and leaves the active state when the temperature drops below the **tlow** limit.
Both **tlow** and **thigh** are expressed in °C.
**pol** represents the polarity of the DS1621 Tout output in active state.
If **pol** is 0, the output is active low and if **pol** is 1, the output is active high.
Refer to the DS1621 data sheet for more information.

**void ds1621_twi_init(TWI_MASTER_INFO_t *ptwim)**

this function is used to initialize the DS1621 library's internal variables when using the **TWI Functions for Master Mode Operation for XMEGA Devices**.
It is not used for non-XMEGA devices.

The **ptwim** parameter must point to a **TWI_MASTER_INFO_t** structure variable, that is used to hold the information required by the TWI module when operating in master mode.
The **ds1621_twi_init** function must be called before **ds1621_init**.
Refer to the supplied example at the end of this chapter for more details.

**unsigned char ds1621_get_status(unsigned char chip)**

this function reads the contents of the configuration/status register of the DS1621 with address **chip**.
Refer to the DS1621 data sheet for more information about this register.

**void ds1621_set_status(unsigned char chip, unsigned char data)**

this function sets the contents of the configuration/status register of the DS1621 with address **chip**.
Refer to the DS1621 data sheet for more information about this register.

**void ds1621_start(unsigned char chip)**

this functions exits the DS1621, with address **chip**, from the power-down mode and starts the temperature measurements and the thermostat.

**void ds1621_stop(unsigned char chip)**

this functions enters the DS1621, with address **chip**, in power-down mode and stops the temperature measurements and the thermostat.

**int ds1621_temperature_10(unsigned char chip)**

this function returns the temperature of the DS1621 sensor with the address **chip**.
The temperature is in °C and is multiplied by 10.

Example how to display the temperature of two DS1621 sensors with addresses 0 and 1.
The chips are accessed using the **TWI Functions for Master Mode Operation for non-XMEGA Devices**.

```
/* include the DS1621 functions for TWI */
#include <ds1621_twi.h>

/* include the LCD Functions
   The connections must be specified in the
   Project|Configure|C Compiler|Libraries|Alphanumeric LCD menu */
#include <alcd.h>

/* include the prototype for sprintf */
#include <stdio.h>

/* include the prototype for abs */
#include <stdlib.h>

char display_buffer[33];

void main(void)
{
int t0,t1;

/* initialize the LCD, 2 rows by 16 columns */
lcd_init(16);

/* initialize the TWI in master mode with 100 kHz bit rate */
twi_master_init(100);

/* enable interrupts so that TWI can be used */
#asm("sei")

/* initialize the DS1621 sensor with address 0 */
/* tlow=20°C thigh=25°C */
ds1621_init(0,20,25,0);
```

```
/* initialize the DS1621 sensor with address 1 */
/* tlow=30°C thigh=35°C */
ds1621_init(1,30,35,0);

/* temperature display loop */
while (1)
      {
      /* read the temperature of DS1621 #0 *10°C */
      t0=ds1621_temperature_10(0);

      /* read the temperature of DS1621 #1 *10°C */
      t1=ds1621_temperature_10(1);

      /* prepare the displayed temperatures */
      /* in the display_buffer */
      sprintf(display_buffer,"t0=%-i.%-u%cC\nt1=%-i.%-u%cC",
      t0/10,abs(t0%10),0xdf,t1/10,abs(t1%10),0xdf);

      /* display the temperatures */
      lcd_clear();
      lcd_puts(display_buffer);
      };
}
```

The same example, but the DS1621 chips are accessed using the **TWI Functions for Master Mode Operation for XMEGA Devices**.
The chips are connected to the TWI of PORTD (TWID) of an ATxmega128A1.

```
/* include the DS1621 functions for TWI */
#include <ds1621_twi.h>

/* include the LCD Functions
   The connections must be specified in the
   Project|Configure|C Compiler|Libraries|Alphanumeric LCD menu */
#include <alcd.h>

/* include the prototype for sprintf */
#include <stdio.h>

/* include the prototype for abs */
#include <stdlib.h>

char display_buffer[33];

/* structure that holds information used by the TWID master
   for performing a TWI bus transaction */
TWI_MASTER_INFO_t twid_master;

/* interrupt service routine for TWID master */
interrupt [TWID_TWIM_vect] void twid_master_isr(void)
{
twi_master_int_handler(&twid_master);
}

/* TWI clock rate [Hz] */
#define TWI_CLK_RATE 100000
```

```
void main(void)
{
int t0,t1;

/* initialize the LCD, 2 rows by 16 columns */
lcd_init(16);

/* general TWID initialization
   no external driver interface
   no SDA hold time */
twi_init(&TWID,false,false);

/* enable and initialize the TWID master
   interrupt level: low */
twi_master_init(&twid_master,&TWID,TWI_MASTER_INTLVL_LO_gc,
TWI_BAUD_REG(_MCU_CLOCK_FREQUENCY_,TWI_CLK_RATE));

/* enable the Low interrupt level */
PMIC.CTRL|=PMIC_LOLVLEN_bm;

/* set the DS1621 functions to use TWID */
ds1621_twi_init(&twid_master);

/* enable interrupts so that TWI can be used */
#asm("sei")

/* initialize the DS1621 sensor with address 0 */
/* tlow=20°C thigh=25°C */
ds1621_init(0,20,25,0);

/* initialize the DS1621 sensor with address 1 */
/* tlow=30°C thigh=35°C */
ds1621_init(1,30,35,0);

/* temperature display loop */
while (1)
      {
      /* read the temperature of DS1621 #0 *10°C */
      t0=ds1621_temperature_10(0);

      /* read the temperature of DS1621 #1 *10°C */
      t1=ds1621_temperature_10(1);

      /* prepare the displayed temperatures */
      /* in the display_buffer */
      sprintf(display_buffer,"t0=%-i.%-u%cC\nt1=%-i.%-u%cC",
      t0/10,abs(t0%10),0xdf,t1/10,abs(t1%10),0xdf);

      /* display the temperatures */
      lcd_clear();
      lcd_puts(display_buffer);
      };
}
```

The same example, but the DS1621 chips are accessed using the **Software Bit-Banged I2C Bus Functions**.

```
/* include the DS1621 functions
   The I2C bus connections and bit rate must be specified in the
   Project|Configure|C Compiler|Libraries|I2C menu */
#include <ds1621.h>

/* include the LCD Functions
   The connections must be specified in the
   Project|Configure|C Compiler|Libraries|Alphanumeric LCD menu */
#include <alcd.h>

/* include the prototype for sprintf */
#include <stdio.h>

/* include the prototype for abs */
#include <stdlib.h>

char display_buffer[33];

void main(void)
{
int t0,t1;

/* initialize the LCD, 2 rows by 16 columns */
lcd_init(16);

/* initialize the I²C bus */
i2c_init();

/* initialize the DS1621 sensor with address 0 */
/* tlow=20°C thigh=25°C */
ds1621_init(0,20,25,0);

/* initialize the DS1621 sensor with address 1 */
/* tlow=30°C thigh=35°C */
ds1621_init(1,30,35,0);

/* temperature display loop */
while (1)
      {
      /* read the temperature of DS1621 #0 *10°C */
      t0=ds1621_temperature_10(0);

      /* read the temperature of DS1621 #1 *10°C */
      t1=ds1621_temperature_10(1);

      /* prepare the displayed temperatures */
      /* in the display_buffer */
      sprintf(display_buffer,"t0=%-i.%-u%cC\nt1=%-i.%-u%cC",
      t0/10,abs(t0%10),0xdf,t1/10,abs(t1%10),0xdf);

      /* display the temperatures */
      lcd_clear();
      lcd_puts(display_buffer);
      };
}
```

## 5.22.6 Maxim/Dallas Semiconductor DS1820/DS18S20 Temperature Sensors Functions

These functions are intended for easy interfacing between C programs and the DS1820/DS18S20 1 Wire bus temperature sensors.
The prototypes for these functions are placed in the file **ds1820.h**, located in the .\INC subdirectory. This file must be **#include -d** before using the functions.
The 1 Wire bus functions prototypes are automatically #include -d with the **ds1820.h**.

The 1 Wire functions must be configured, by specifying the I/O port and bit used for communication through the 1 Wire protocol.
This is accomplished in the **Project|Configure|C Compiler|Libraries|1 Wire** menu:

- the **Enable 1 Wire Bus Interface Support** option must be activated
- the **I/O Port** and **Bit** must be specified in **Data Connection.**

The DS1820/DS18S20 functions are:

**unsigned char ds1820_read_spd(unsigned char *addr)**

this function reads the contents of the SPD for the DS1820/DS18S20 sensor with the ROM code stored in an array of 8 bytes located at address **addr**.
The functions returns the value 1 on succes and 0 in case of error.
If only one DS1820/DS18S20 sensor is used, no ROM code array is necessary and the pointer **addr** must be NULL (0).
The contents of the SPD will be stored in the structure:

```
struct __ds1820_scratch_pad_struct
       {
       unsigned char temp_lsb,temp_msb,
                temp_high,temp_low,
                res1,res2,
                cnt_rem,cnt_c,
                crc;
       } __ds1820_scratch_pad;
```

defined in the **ds1820.h** header file.

**int ds1820_temperature_10(unsigned char *addr)**

this function returns the temperature of the DS1820/DS18S20 sensor with the ROM code stored in an array of 8 bytes located at address **addr**.
The temperature is measured in °C and is multiplied by 10. In case of error the function returns the value -9999.
If only one DS1820/DS18S20 sensor is used, no ROM code array is necessary and the pointer **addr** must be NULL (0).

If several sensors are used, then the program must first identify the ROM codes for all the sensors. Only after that the **ds1820_temperature_10** function may be used, with the **addr** pointer pointing to the array which holds the ROM code for the needed device.
Example:

```
#include <mega8515.h>

/* the ATmega8515 port and bit used for the 1 Wire bus must be specified in
   the Project|Configure|C Compiler|Libraries 1 Wire menu */

/* include the DS1820/DS18S20 functions prototypes */
#include <ds1820.h>

/* include the printf function prototype */
#include <stdio.h>

/* include the abs function prototype */
#include <stdlib.h>

/* quartz crystal frequency [Hz] */
#define xtal 4000000L

/* Baud rate */
#define baud 9600

/* maximum number of DS1820/DS18S20 connected to the bus */
#define MAX_DEVICES 8

/* DS1820/DS18S20 devices ROM code storage area,
   9 bytes are used for each device
   (see the w1_search function description),
   but only the first 8 bytes contain the ROM code
   and CRC */
unsigned char rom_codes[MAX_DEVICES][9];

main()
{
unsigned char i,devices;
int temp;

/* initialize the USART control register
   TX enabled, no interrupts, 8 data bits */
UCSRA=0x00;
UCSRB=0x08;
UCSRC=0x86;

/* initialize the USART's baud rate */
UBRRH=(xtal/16/baud-1) >> 8;
UBRRL=(xtal/16/baud-1) & 0xFF;

/* detect how many DS1820/DS18S20 devices
   are connected to the bus and
   store their ROM codes in the rom_codes array */
devices=w1_search(0xf0,rom_codes);
```

```
/* display the number */
printf("%-u DEVICE(S) DETECTED\n\r",devices);

/* if no devices were detected then halt */
if (devices==0) while (1); /* loop forever */

/* measure and display the temperature(s) */
while (1)
      {
      for (i=0;i<devices;)
          {
          temp=ds1820_temperature_10(&rom_codes[i][0]);
          printf("t%-u=%-i.%-u\xf8C\n\r",++i,temp/10,
          abs(temp%10));
          };
      };
}
```

**unsigned char ds1820_set_alarm(unsigned char *addr,signed char temp_low, signed char temp_high)**

this function sets the low (**temp_low**) and high (**temp_high**) temperature alarms of the DS1820/DS18S20.
In case of success the function returns the value 1, else it returns 0.
The alarm temperatures are stored in both the DS1820/DS18S20's scratchpad RAM and its EEPROM.
The ROM code needed to address the device is stored in an array of 8 bytes located at address **addr**.
If only one DS1820/DS18S20 sensor is used, no ROM code array is necessary and the pointer **addr** must be NULL (0).

The alarm status for all the DS1820/DS18S20 devices on the 1 Wire bus can be determined by calling the **w1_search** function with the Alarm Search (ECh) command.
Example:

```
#include <mega8515.h>

/* the ATmega8515 port and bit used for the 1 Wire bus must be specified in
   the Project|Configure|C Compiler|Libraries 1 Wire menu */

/* include the DS1820/DS18S20 functions prototypes */
#include <ds1820.h>

/* include the printf function prototype */
#include <stdio.h>

/* include the abs function prototype */
#include <stdlib.h>

/* maximum number of DS1820/DS18S20 connected to the bus */
#define MAX_DEVICES 8

/* DS1820/DS18S20 devices ROM code storage area,
   9 bytes are used for each device
   (see the w1_search function description),
   but only the first 8 bytes contain the ROM code and CRC */
unsigned char rom_codes[MAX_DEVICES][9];
```

```
/* allocate space for ROM codes of the devices
   which generate an alarm */
unsigned char alarm_rom_codes[MAX_DEVICES][9];

#define xtal 4000000L /* quartz crystal frequency [Hz] */
#define baud 9600 /* Baud rate */

main()
{
unsigned char i,devices;
int temp;

/* initialize the USART control register
   TX enabled, no interrupts, 8 data bits */
UCSRA=0x00;
UCSRB=0x08;
UCSRC=0x86;

/* initialize the USART's baud rate */
UBRRH=(xtal/16/baud-1) >> 8;
UBRRL=(xtal/16/baud-1) & 0xFF;

/* detect how many DS1820/DS18S20 devices
   are connected to the bus and
   store their ROM codes in the rom_codes array */
devices=w1_search(0xf0,rom_codes);

/* display the number */
printf("%-u DEVICE(S) DETECTED\n\r",devices);

/* if no devices were detected then halt */
if (devices==0) while (1); /* loop forever */

/* set the temperature alarms for all the devices
   temp_low=25°C temp_high=35°C */
for (i=0;i<devices;i++)
    {
    printf("INITIALIZING DEVICE #%-u ", i+1);
    if (ds1820_set_alarm(&rom_codes[i][0],25,35))
       putsf("OK"); else putsf("ERROR");
    };

while (1)
      {
      /* measure and display the temperature(s) */
      for (i=0;i<devices;)
          {
          temp=ds1820_temperature_10(&rom_codes[i][0]);
          printf("t%-u=%-i.%-u\xf8C\n\r",++i,temp/10,
          abs(temp%10));
          };

      /* display the number of devices which
         generated an alarm */
      printf("ALARM GENERATED BY %-u DEVICE(S)\n\r",
      w1_search(0xec,alarm_rom_codes));
      };
}
```

Refer to the DS1820/DS18S20 data sheet for more information.

## 5.22.7 Maxim/Dallas Semiconductor DS18B20 Temperature Sensor Functions

These functions are intended for easy interfacing between C programs and the DS18B20 1 Wire bus temperature sensor.
The prototypes for these functions are placed in the file **ds18b20.h**, located in the .\INC subdirectory. This file must be **#include -d** before using the functions.
The 1 Wire bus functions prototypes are automatically #include -d with the **ds18b20.h**.

The 1 Wire functions must be configured, by specifying the I/O port and bit used for communication through the 1 Wire protocol.
This is accomplished in the **Project|Configure|C Compiler|Libraries|1 Wire** menu:

- the **Enable 1 Wire Bus Interface Support** option must be activated
- the **I/O Port** and **Bit** must be specified in **Data Connection.**

The DS18B20 functions are:

**unsigned char ds18b20_read_spd(unsigned char *addr)**

this function reads the contents of the SPD for the DS18B20 sensor with the ROM code stored in an array of 8 bytes located at address **addr**.
The functions returns the value 1 on succes and 0 in case of error.
If only one DS18B20 sensor is used, no ROM code array is necessary and the pointer **addr** must be NULL (0).
The contents of the SPD will be stored in the structure:

```
struct __ds18b20_scratch_pad_struct
       {
       unsigned char temp_lsb,temp_msb,
                temp_high,temp_low,
                conf_register,
                res1,
                res2,
                res3,
                crc;
       } __ds18b20_scratch_pad;
```

defined in the **ds18b20.h** header file.

**unsigned char ds18b20_init(unsigned char *addr,signed char temp_low,signed char temp_high,unsigned char resolution)**

this function sets the low (**temp_low**) and high (**temp_high**) temperature alarms and specifies the temperature measurement **resolution** of the DS18B20.
The resolution argument may take the value of one of the following macros defined in the **ds18b20.h** header file:

DS18B20_9BIT_RES for 9 bit tempearture measurement resolution (0.5°C)
DS18B20_10BIT_RES for 10 bit tempearture measurement resolution (0.25°C)
DS18B20_11BIT_RES for 11 bit tempearture measurement resolution (0.125°C)
DS18B20_12BIT_RES for 12 bit tempearture measurement resolution (0.0625°C)

In case of success the function returns the value 1, else it returns 0.
The alarm temperatures and resolution are stored in both the DS18B20's scratchpad SRAM and its EEPROM.
The ROM code needed to address the device is stored in an array of 8 bytes located at address **addr**.
If only one DS18B20 sensor is used, no ROM code array is necessary and the pointer **addr** must be NULL (0).
The alarm status for all the DS18B20 devices on the 1 Wire bus can be determined by calling the **w1_search** function with the Alarm Search (ECh) command.

**float ds18b20_temperature(unsigned char *addr)**

this function returns the temperature of the DS18B20 sensor with the ROM code stored in an array of 8 bytes located at address **addr**.
The temperature is measured in °C. In case of error the function returns the value -9999.
If only one DS18B20 sensor is used, no ROM code array is necessary and the pointer **addr** must be NULL (0).
Prior on calling the the **ds18b20_temperature** function for the first time, the **ds18b20_init** function must be used to specify the desired temperature measurement resolution.
If more several sensors are used, then the program must first identify the ROM codes for all the sensors.
Only after that the **ds18b20_temperature** function may be used, with the **addr** pointer pointing to the array which holds the ROM code for the needed device.

Example:

```
#include <mega8515.h>

/* the ATmega8515 port and bit used for the 1 Wire bus must be specified in
   the Project|Configure|C Compiler|Libraries 1 Wire menu */

/* include the DS18B20 functions prototypes */
#include <ds18b20.h>

/* include the printf function prototype */
#include <stdio.h>

/* quartz crystal frequency [Hz] */
#define xtal 4000000L

/* Baud rate */
#define baud 9600

/* maximum number of DS18B20 connected to the bus */
#define MAX_DEVICES 8

/* DS18B20 devices ROM code storage area,
   9 bytes are used for each device
   (see the w1_search function description),
   but only the first 8 bytes contain the ROM code
   and CRC */
unsigned char rom_codes[MAX_DEVICES][9];

/* allocate space for ROM codes of the devices
   which generate an alarm */
unsigned char alarm_rom_codes[MAX_DEVICES][9];
```

```
main()
{
unsigned char i,devices;

/* initialize the USART control register
   TX enabled, no interrupts, 8 data bits */
UCSRA=0x00;
UCSRB=0x08;
UCSRC=0x86;

/* initialize the USART's baud rate */
UBRRH=(xtal/16/baud-1) >> 8;
UBRRL=(xtal/16/baud-1) & 0xFF;

/* detect how many DS18B20 devices
   are connected to the bus and
   store their ROM codes in the rom_codes array */
devices=w1_search(0xf0,rom_codes);

/* display the number */
printf("%-u DEVICE(S) DETECTED\n\r",devices);

/* if no devices were detected then halt */
if (devices==0) while (1); /* loop forever */

/* set the temperature alarms & temperature
   measurement resolutions for all the devices
   temp_low=25°C temp_high=35°C resolution 12bits */
for (i=0;i<devices;i++)
    {
    printf("INITIALIZING DEVICE #%-u ",i+1);
    if (ds18b20_init(&rom_codes[i][0],25,35,DS18B20_12BIT_RES))
       putsf("OK"); else putsf("ERROR");
    };

while (1)
      {
      /* measure and display the temperature(s) */
      for (i=0;i<devices;)
          printf("t%u=%+.3f\xf8C\n\r",i+1,
          ds18b20_temperature(&rom_codes[i++][0]));

      /* display the number of devices which
         generated an alarm */
      printf("ALARM GENERATED BY %-u DEVICE(S)\n\r",
      w1_search(0xec,alarm_rom_codes));
      };
}
```

Refer to the DS18B20 data sheet for more information.

## 5.22.8 Maxim/Dallas Semiconductor DS2430 EEPROM Functions

These functions are intended for easy interfacing between C programs and the DS2430 1 Wire bus EEPROM.
The prototypes for these functions are placed in the file **ds2430.h**, located in the .\INC subdirectory. This file must be **#include -d** before using the functions.
The 1 Wire bus functions prototypes are automatically #include -d with the **ds2430.h**.

The 1 Wire functions must be configured, by specifying the I/O port and bit used for communication through the 1 Wire protocol.
This is accomplished in the **Project|Configure|C Compiler|Libraries|1 Wire** menu:

- the **Enable 1 Wire Bus Interface Support** option must be activated
- the **I/O Port** and **Bit** must be specified in **Data Connection.**

The DS2430 functions are:

**unsigned char ds2430_read_block(unsigned char *romcode,unsigned char *dest, unsigned char addr,unsigned char size);**

this function reads a block of **size** bytes starting from the DS2430 EEPROM memory address **addr** and stores it in the string **dest** located in RAM.
It returns 1 if successful, 0 if not.
The DS2430 device is selected using its ROM code stored in an array of 8 bytes located at address **romcode**.

**unsigned char ds2430_read(unsigned char *romcode,unsigned char addr, unsigned char *data);**

this function reads a byte from the DS2430 EEPROM memory address **addr** and stores it in the RAM memory location pointed by **data**.
It returns 1 if successful, 0 if not.
The DS2430 device is selected using its ROM code stored in an array of 8 bytes located at address **romcode**.

**unsigned char ds2430_write_block(unsigned char *romcode, unsigned char *source,unsigned char addr,unsigned char size);**

this function writes a block of **size** bytes, from the string **source**, located in RAM, in the DS2430 EEPROM starting from memory address **addr**.
It returns 1 if successful, 0 if not.
The DS2430 device is selected using its ROM code stored in an array of 8 bytes located at address **romcode**.

**unsigned char ds2430_write(unsigned char *romcode, unsigned char addr,unsigned char data);**

this function writes the byte **data** at DS2430 EEPROM memory address **addr**.
It returns 1 if successful, 0 if not.
The DS2430 device is selected using its ROM code stored in an array of 8 bytes located at address **romcode**.

**unsigned char ds2430_read_appreg_block(unsigned char *romcode, unsigned char *dest,unsigned char addr,unsigned char size);**

this function reads a block of **size** bytes starting from the DS2430 application register address **addr** and stores it in the string **dest** located in RAM.
It returns 1 if successful, 0 if not.
The DS2430 device is selected using its ROM code stored in an array of 8 bytes located at address **romcode**.

**unsigned char ds2430_write_appreg_block(unsigned char *romcode, unsigned char *source,unsigned char addr,unsigned char size);**

this function reads a block of **size** bytes starting from the DS2430 application register address **addr** and stores it in the string **dest** located in RAM.
It returns 1 if successful, 0 if not.
The DS2430 device is selected using its ROM code stored in an array of 8 bytes located at address **romcode**.

If only one DS2430 EEPROM is used, no ROM code array is necessary and the pointer **romcode** must be NULL (0).
If several 1 Wire device are used, then the program must first identify the ROM codes for all the devices. Only after that the DS2430 functions may be used, with the **romcode** pointer pointing to the array which holds the ROM code for the needed device.
Example:

```
/* The ATmega8515 port and bit used for the 1 Wire bus must be specified
   in the Project|Configure|C Compiler|Libraries 1 Wire menu

   The DS2430 devices are connected to
   bit 6 of PORTA of the ATmegaS8515 as follows:

   [DS2430]      [STK200 PORTA HEADER]
    1 GND         -   9  GND
    2 DATA        -   7  PA6

   All the devices must be connected in parallel

   AN 4.7k PULLUP RESISTOR MUST BE CONNECTED
   BETWEEN DATA (PA6) AND +5V !
*/

/* include the DS2430 functions */
#include <ds2430.h>
#include <mega8515.h>
#include <stdio.h>

/* DS2430 devices ROM code storage area,
   9 bytes are used for each device
   (see the w1_search function description),
   but only the first 8 bytes contain the ROM code
   and CRC */
#define MAX_DEVICES 8

unsigned char rom_code[MAX_DEVICES][9];
```

```
char text[]="Hello world!";
char buffer[32];
#define START_ADDR 2

/* ATmega8515 clock frequency [Hz] */
#define xtal 4000000L

/* Baud rate */
#define baud 9600

main() {
unsigned char i,devices;

/* initialize the USART control register
   TX enabled, no interrupts, 8 data bits */
UCSRA=0x00;
UCSRB=0x08;
UCSRC=0x86;

/* initialize the USART's baud rate */
UBRRH=(xtal/16/baud-1) >> 8;
UBRRL=(xtal/16/baud-1) & 0xFF;

/* detect how many 1 Wire devices are present on the bus */
devices=w1_search(0xF0,&rom_code[0][0]);
printf("%-u 1 Wire devices found\n\r",devices);
for (i=0;i<devices;i++)
   /* make sure to select only the DS2430 types
      0x14 is the DS2430 family code */
   if (rom_code[i][0]==DS2430_FAMILY_CODE)
      {
      printf("\n\r");
      /* write text in each DS2430 at START_ADDR */
      if (ds2430_write_block(&rom_code[i][0],
         text,START_ADDR,sizeof(text)))
         {
         printf("Data written OK in DS2430 #%-u!\n\r",i+1);
         /* display the text written in each DS2430 */
         if (ds2430_read_block(&rom_code[i][0],buffer,START_ADDR,
            sizeof(text)))
            printf("Data read OK!\n\rDS2430 #%-u text: %s\n\r",
            i+1,buffer);
         else printf("Error reading data from DS2430 #%-u!\n\r",
            i+1);
         }
      else printf("Error writing data to DS2430 #%-u!\n\r",i+1);
      };
/* stop */
while (1);
}
```

Refer to the DS2430 data sheet for more information.

## 5.22.9 Maxim/Dallas Semiconductor DS2433 EEPROM Functions

These functions are intended for easy interfacing between C programs and the DS2433 1 Wire bus EEPROM.
The prototypes for these functions are placed in the file **ds2433.h**, located in the .\INC subdirectory. This file must be **#include -d** before using the functions.
The 1 Wire bus functions prototypes are automatically #include -d with the **ds2433.h**.

The 1 Wire functions must be configured, by specifying the I/O port and bit used for communication through the 1 Wire protocol.
This is accomplished in the **Project|Configure|C Compiler|Libraries|1 Wire** menu:

- the **Enable 1 Wire Bus Interface Support** option must be activated
- the **I/O Port** and **Bit** must be specified in **Data Connection.**

The DS2433 functions are:

**unsigned char ds2433_read_block(unsigned char *romcode,unsigned char *dest, unsigned int addr,unsigned int size);**

this function reads a block of **size** bytes starting from the DS2433 EEPROM memory address **addr** and stores it in the string **dest** located in RAM.
It returns 1 if successful, 0 if not.
The DS2433 device is selected using its ROM code stored in an array of 8 bytes located at address **romcode**.

**unsigned char ds2433_read(unsigned char *romcode,unsigned int addr, unsigned char *data);**

this function reads a byte from the DS2433 EEPROM memory address **addr** and stores it in the RAM memory location pointed by **data**.
It returns 1 if successful, 0 if not.
The DS2433 device is selected using its ROM code stored in an array of 8 bytes located at address **romcode**.

**unsigned char ds2433_write_block(unsigned char *romcode, unsigned char *source,unsigned int addr,unsigned int size);**

this function writes a block of **size** bytes, from the string **source**, located in RAM, in the DS2433 EEPROM starting from memory address **addr**.
It returns 1 if successful, 0 if not.
The DS2433 device is selected using its ROM code stored in an array of 8 bytes located at address **romcode**.

**unsigned char ds2433_write(unsigned char *romcode,unsigned int addr, unsigned char data);**

this function writes the byte **data** at DS2433 EEPROM memory address **addr**.
It returns 1 if successful, 0 if not.
The DS2433 device is selected using its ROM code stored in an array of 8 bytes located at address **romcode**.

If only one DS2433 EEPROM is used, no ROM code array is necessary and the pointer **romcode** must be NULL (0).
If several 1 Wire device are used, then the program must first identify the ROM codes for all the devices. Only after that the DS2433 functions may be used, with the **romcode** pointer pointing to the array which holds the ROM code for the needed device.
Example:

```
/* The ATmega8515 port and bit used for the 1 Wire bus must be specified
   in the Project|Configure|C Compiler|Libraries 1 Wire menu

   The DS2433 devices are connected to
   bit 6 of PORTA of the ATmega8515 as follows:

   [DS2433]      [STK200 PORTA HEADER]
    1 GND         -   9  GND
    2 DATA        -   7  PA6

   All the devices must be connected in parallel

   AN 4.7k PULLUP RESISTOR MUST BE CONNECTED
   BETWEEN DATA (PA6) AND +5V !
*/
#asm
    .equ __w1_port=0x1b
    .equ __w1_bit=6
#endasm

// test the DS2433 functions
#include <ds2433.h>
#include <mega8515.h>
#include <stdio.h>

/* DS2433 devices ROM code storage area,
   9 bytes are used for each device
   (see the w1_search function description),
   but only the first 8 bytes contain the ROM code
   and CRC */
#define MAX_DEVICES 8
unsigned char rom_code[MAX_DEVICES][9];

char text[]="This is a long text to \
be able to test writing across the \
scratchpad boundary";

char buffer[100];

#define START_ADDR 2

/* ATmega8515 clock frequency [Hz] */
#define xtal 4000000L

/* Baud rate */
#define baud 9600
```

```
main() {
unsigned char i,devices;

/* initialize the USART control register
   TX enabled, no interrupts, 8 data bits */
UCSRA=0x00;
UCSRB=0x08;
UCSRC=0x86;

/* initialize the USART's baud rate */
UBRRH=(xtal/16/baud-1) >> 8;
UBRRL=(xtal/16/baud-1) & 0xFF;

/* detect how many 1 Wire devices are present on the bus */
devices=w1_search(0xF0,&rom_code[0][0]);
printf("%-u 1 Wire devices found\n\r",devices);

for (i=0;i<devices;i++)
   /* make sure to select only the DS2433 types
      0x23 is the DS2433 family code */
   if (rom_code[i][0]==DS2433_FAMILY_CODE)
      {
      printf("\n\r");
      /* write text in each DS2433 at START_ADDR */
      if (ds2433_write_block(&rom_code[i][0],
         text,START_ADDR,sizeof(text)))
         {
         printf("Data written OK in DS2433 #%-u!\n\r",i+1);
         /* display the text written in each DS2433 */
         if (ds2433_read_block(&rom_code[i][0],buffer,START_ADDR,
            sizeof(text)))
            printf("Data read OK!\n\rDS2433 #%-u text: %s\n\r",
            i+1,buffer);
         else printf("Error reading data from DS2433 #%-u!\n\r",i+1);
         }
      else printf("Error writing data to DS2433 #%-u!\n\r",i+1);
      };
/* stop */
while (1);
}
```

Refer to the DS2433 data sheet for more information.

## 5.22.10 National Semiconductor LM75 Temperature Sensor Functions

These functions are intended for easy interfacing between C programs and the LM75 $I^2C$ bus temperature sensor, using both the hardware TWI and software bit-banged $I^2C$ bus functions.
One of the following header files, located in the .\INC subdirectory must be **#include** -d before using these functions:

- **lm75_twi.h** for hardware TWI
- **lm75.h** for software bit-banged $I^2C$ bus.

The appropriate header files for hardware TWI (**twi.h** or **twix.h**) or software bit-banged $I^2C$ bus (**i2c.h**) functions prototypes are automatically **#include** -d with the **lm75_twi.h** or **lm75.h**.

The **Project|Configure|C Compiler|Libraries|I2C** menu must be used for specifying the I/O port allocation of the **SCL** and **SDA** signals, along the **bit rate** of the **SCL** clock, for bit-banged $I^2C$ bus (<u>not the hardware TWI</u>) communication.

**Note: For proper operation, the LM75 Functions require the presence of 3.3k - 4.7k pull-up resistors to +5V (+3.3V for XMEGA devices) on the SCL and SDA signals.**

The LM75 Functions are:

**void lm75_init(unsigned char chip,signed char thyst,signed char tos, unsigned char pol)**

this function initializes the LM75 sensor chip.
Before calling this function, the TWI hardware, respectively $I^2C$ bus, must be initialized by calling the **twi_master_init** (**twi_init**, **twi_master_ini**t and **lm75_twi_init** for XMEGA devices), respectively **i2c_init** functions.This is the first function that must be called prior to using the other LM75 Functions.

If more then one chip is connected to the $I^2C$ bus, then the function must be called for each one, specifying accordingly the function parameter **chip**.
Maximum 8 LM75 chips can be connected to the $I^2C$ bus, their **chip** address can be from 0 to 7.
The LM75 is configured in comparator mode, where it functions like a thermostat.
The O.S. output becomes active when the temperature exceeds the **tos** limit, and leaves the active state when the temperature drops below the **thyst** limit.
Both **thyst** and **tos** are expressed in °C.
**pol** represents the polarity of the LM75 O.S. output in active state.
If **pol** is 0, the output is active low and if **pol** is 1, the output is active high.
Refer to the LM75 data sheet for more information.

**void lm75_twi_init(TWI_MASTER_INFO_t *ptwim)**

this function is used to initialize the LM75 library's internal variables when using the **TWI Functions for Master Mode Operation for XMEGA Devices**.
It is not used for non-XMEGA devices.

The **ptwim** parameter must point to a **TWI_MASTER_INFO_t** structure variable, that is used to hold the information required by the TWI module when operating in master mode.
The **lm75_twi_init** function must be called before **lm75_init**.
Refer to the supplied example at the end of this chapter for more details.

**int lm75_temperature_10(unsigned char chip)**

this function returns the temperature of the LM75 sensor with the address **chip**.
The temperature is in °C and is multiplied by 10.
A 300ms delay must be present between two successive calls to the **lm75_temperature_10** function.

Example how to display the temperature of two LM75 sensors with addresses 0 and 1.
The chips are accessed using the **TWI Functions for Master Mode Operation for non-XMEGA Devices**.

```
/* include the LM75 functions for TWI */
#include <lm75_twi.h>

/* include the LCD Functions
   The connections must be specified in the
   Project|Configure|C Compiler|Libraries|Alphanumeric LCD menu */
#include <alcd.h>

/* include the prototype for sprintf */
#include <stdio.h>

/* include the prototype for abs */
#include <stdlib.h>

/* include the prototypes for the delay functions */
#include <delay.h>

char display_buffer[33];

void main(void) {
int t0,t1;

/* initialize the LCD, 2 rows by 16 columns */
lcd_init(16);

/* initialize the TWI in master mode with 100 kHz bit rate */
twi_master_init(100);

/* enable interrupts so that TWI can be used */
#asm("sei")

/* initialize the LM75 sensor with address 0 */
/* thyst=20°C tos=25°C */
lm75_init(0,20,25,0);

/* initialize the LM75 sensor with address 1 */
/* thyst=30°C tos=35°C */
lm75_init(1,30,35,0);

/* temperature display loop */
while (1)
      {
      /* read the temperature of sensor #0 *10°C */
      t0=lm75_temperature_10(0);
      /* 300ms delay */
      delay_ms(300);

      /* read the temperature of sensor #1 *10°C */
      t1=lm75_temperature_10(1);
      /* 300ms delay */
      delay_ms(300);
```

```
      /* prepare the displayed temperatures */
      /* in the display_buffer */
      sprintf(display_buffer,
      "t0=%-i.%-u%cC\nt1=%-i.%-u%cC",
      t0/10,abs(t0%10),0xdf,t1/10,abs(t1%10),0xdf);

      /* display the temperatures */
      lcd_clear();
      lcd_puts(display_buffer);
      };
}
```

The same example, but the LM75 chips are accessed using the **TWI Functions for Master Mode Operation for XMEGA Devices** using the TWI of PORTD (TWID) of an ATxmega128A1 chip.

```
/* include the LM75 functions for TWI */
#include <lm75_twi.h>

/* include the LCD Functions
   The connections must be specified in the
   Project|Configure|C Compiler|Libraries|Alphanumeric LCD menu */
#include <alcd.h>

/* include the prototype for sprintf */
#include <stdio.h>

/* include the prototype for abs */
#include <stdlib.h>

/* include the prototypes for the delay functions */
#include <delay.h>

char display_buffer[33];

/* structure that holds information used by the TWID master
   for performing a TWI bus transaction */
TWI_MASTER_INFO_t twid_master;

/* interrupt service routine for TWID master */
interrupt [TWID_TWIM_vect] void twid_master_isr(void)
{
twi_master_int_handler(&twid_master);
}

/* TWI clock rate [Hz] */
#define TWI_CLK_RATE 100000

void main(void) {
int t0,t1;

/* initialize the LCD, 2 rows by 16 columns */
lcd_init(16);

/* general TWID initialization
   no external driver interface
   no SDA hold time */
twi_init(&TWID,false,false);
```

```
/* enable and initialize the TWID master
   interrupt level: low */
twi_master_init(&twid_master,&TWID,TWI_MASTER_INTLVL_LO_gc,
TWI_BAUD_REG(_MCU_CLOCK_FREQUENCY_,TWI_CLK_RATE));

/* enable the Low interrupt level */
PMIC.CTRL|=PMIC_LOLVLEN_bm;

/* set the LM75 functions to use TWID */
lm75_twi_init(&twid_master);

/* enable interrupts so that TWI can be used */
#asm("sei")

/* initialize the LM75 sensor with address 0 */
/* thyst=20°C tos=25°C */
lm75_init(0,20,25,0);

/* initialize the LM75 sensor with address 1 */
/* thyst=30°C tos=35°C */
lm75_init(1,30,35,0);

/* temperature display loop */
while (1)
      {
      /* read the temperature of sensor #0 *10°C */
      t0=lm75_temperature_10(0);
      /* 300ms delay */
      delay_ms(300);

      /* read the temperature of sensor #1 *10°C */
      t1=lm75_temperature_10(1);
      /* 300ms delay */
      delay_ms(300);

      /* prepare the displayed temperatures */
      /* in the display_buffer */
      sprintf(display_buffer,
      "t0=%-i.%-u%cC\nt1=%-i.%-u%cC",
      t0/10,abs(t0%10),0xdf,t1/10,abs(t1%10),0xdf);

      /* display the temperatures */
      lcd_clear();
      lcd_puts(display_buffer);
      };
}
```

The same example, but the LM75 chips are accessed using the **Software Bit-Banged I2C Bus Functions**.

```
/* include the LM75 functions
   The I2C bus connections and bit rate must be specified in the
   Project|Configure|C Compiler|Libraries|I2C menu */
#include <lm75.h>

/* include the LCD Functions
   The connections must be specified in the
   Project|Configure|C Compiler|Libraries|Alphanumeric LCD menu */
#include <alcd.h>

/* include the prototype for sprintf */
#include <stdio.h>

/* include the prototype for abs */
#include <stdlib.h>

/* include the prototypes for the delay functions */
#include <delay.h>

char display_buffer[33];

void main(void) {
int t0,t1;

/* initialize the LCD, 2 rows by 16 columns */
lcd_init(16);

/* initialize the I²C bus */
i2c_init();

/* initialize the LM75 sensor with address 0 */
/* thyst=20°C tos=25°C */
lm75_init(0,20,25,0);

/* initialize the LM75 sensor with address 1 */
/* thyst=30°C tos=35°C */
lm75_init(1,30,35,0);

/* temperature display loop */
while (1)
      {
      /* read the temperature of sensor #0 *10°C */
      t0=lm75_temperature_10(0);
      /* 300ms delay */
      delay_ms(300);

      /* read the temperature of sensor #1 *10°C */
      t1=lm75_temperature_10(1);
      /* 300ms delay */
      delay_ms(300);
```

```
      /* prepare the displayed temperatures */
      /* in the display_buffer */
      sprintf(display_buffer,
      "t0=%-i.%-u%cC\nt1=%-i.%-u%cC",
      t0/10,abs(t0%10),0xdf,t1/10,abs(t1%10),0xdf);

      /* display the temperatures */
      lcd_clear();
      lcd_puts(display_buffer);
      };
}
```

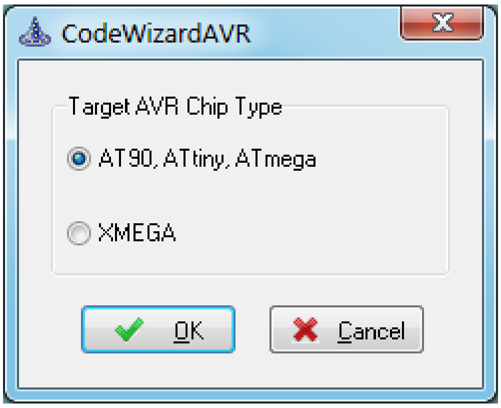
# 6. CodeWizardAVR Automatic Program Generator

The CodeWizardAVR Automatic Program Generator allows you to easily write all the code needed for implementing the following functions:

- External memory access setup
- Chip reset source identification
- Input/Output Port initialization
- External Interrupts initialization
- Timers/Counters initialization
- Watchdog Timer initialization
- UART initialization and interrupt driven buffered serial communication
- Analog Comparator initialization
- ADC initialization
- SPI Interface initialization
- $I^2C$ Bus, LM75 Temperature Sensor, DS1621 Thermometer/Thermostat, PCF8563, PCF8583, DS1302 and DS1307 Real Time Clocks initialization
- 1 Wire Bus and DS1820/DS18S20 Temperature Sensors initialization
- LCD module initialization.

The Automatic Program Generator is invoked using the **Tools|CodeWizardAVR** menu command or by clicking on the toolbar button.

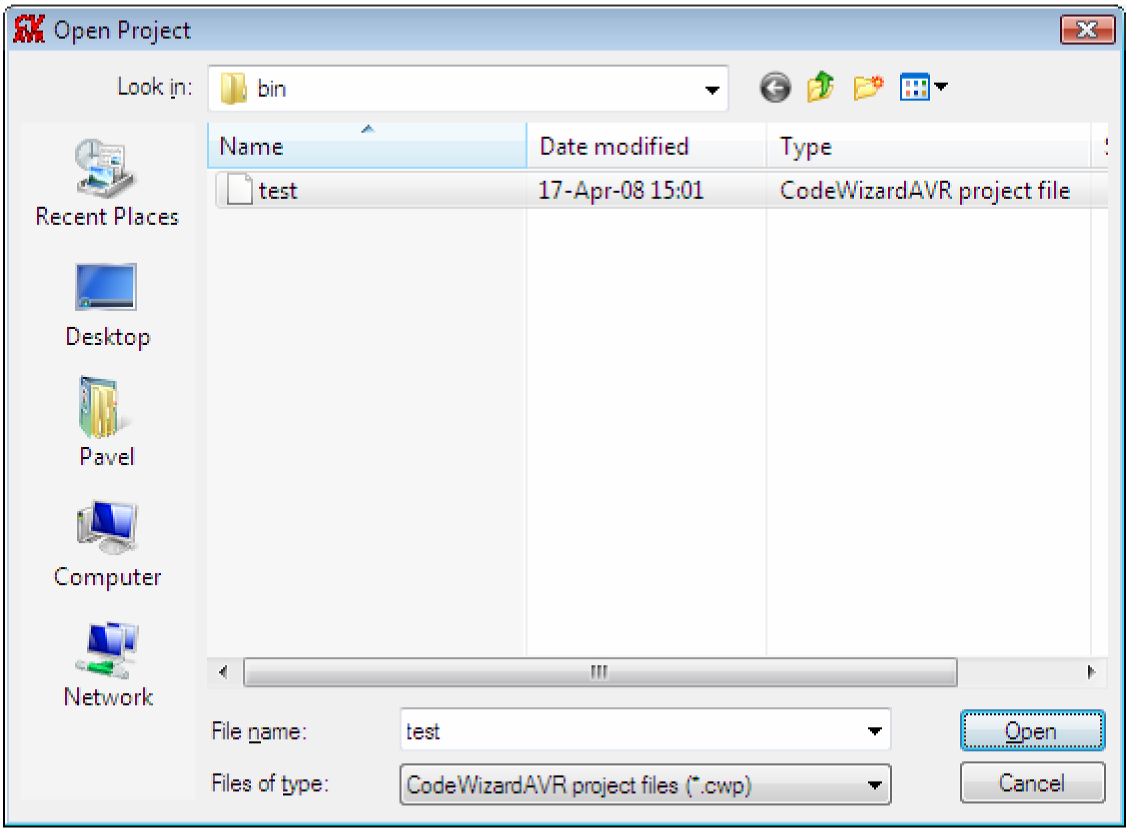
The following dialog box will open



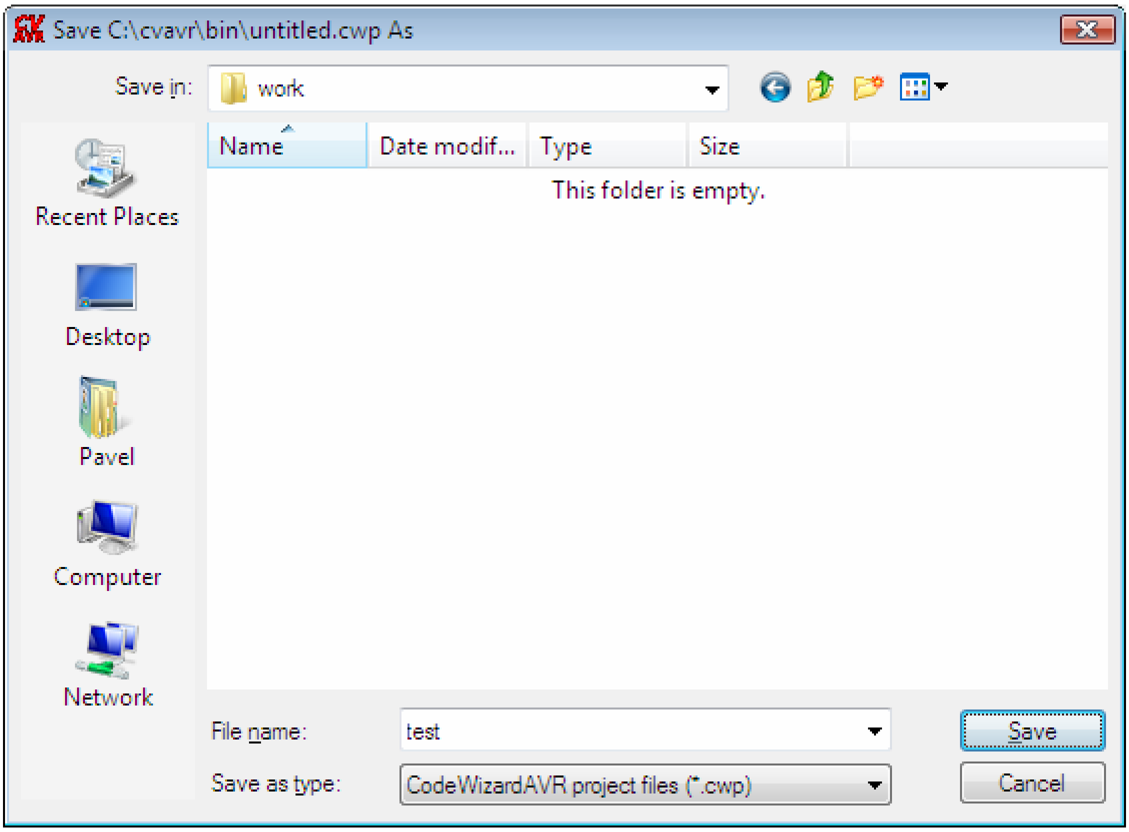

allowing to select between the AVR chip families for which automatic code generation will be performed.

The **File|New** menu command or the toolbar button allow creating a new CodeWizardAVR project. This project will be named by default **untitled.cwp** .

The **File|Open** menu command or the toolbar button allow loading an existing CodeWizardAVR project:



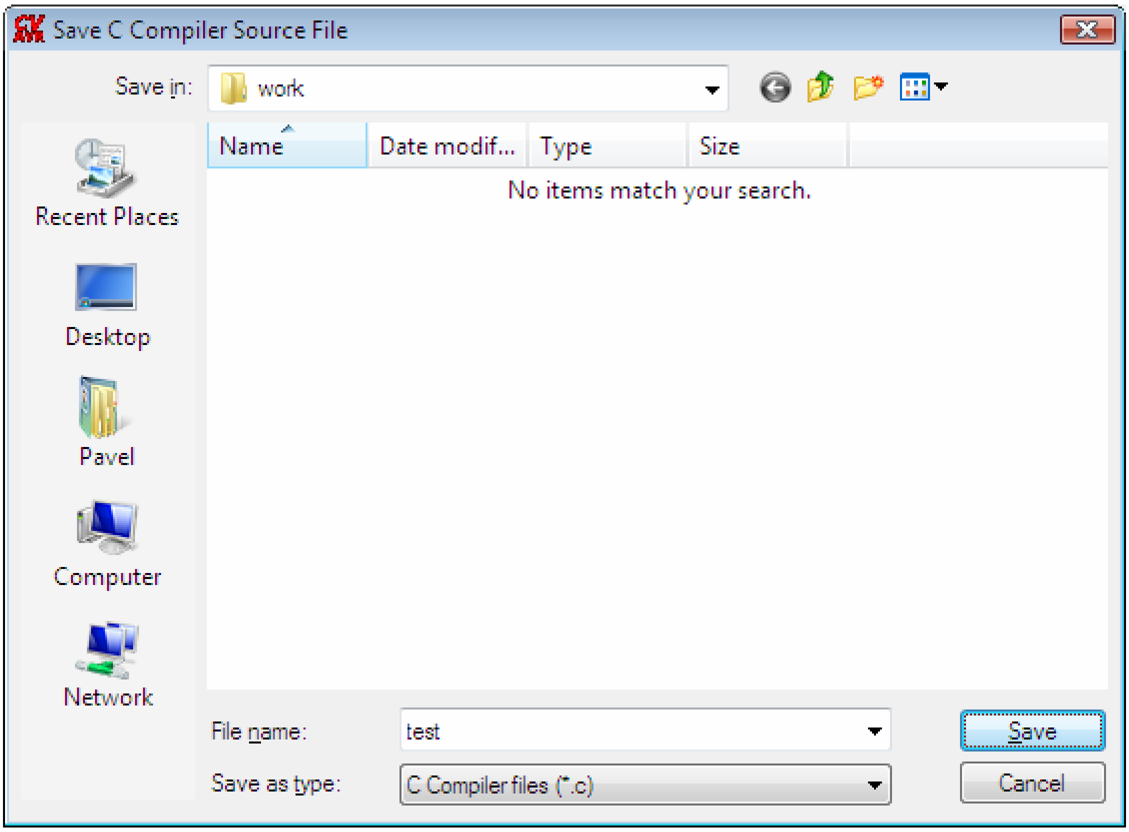
The **File|Save** menu command or the toolbar button allow saving the currently opened CodeWizardAVR project.
The **File|Save As** menu command or the toolbar button allow saving the currently opened CodeWizardAVR project under a new name:



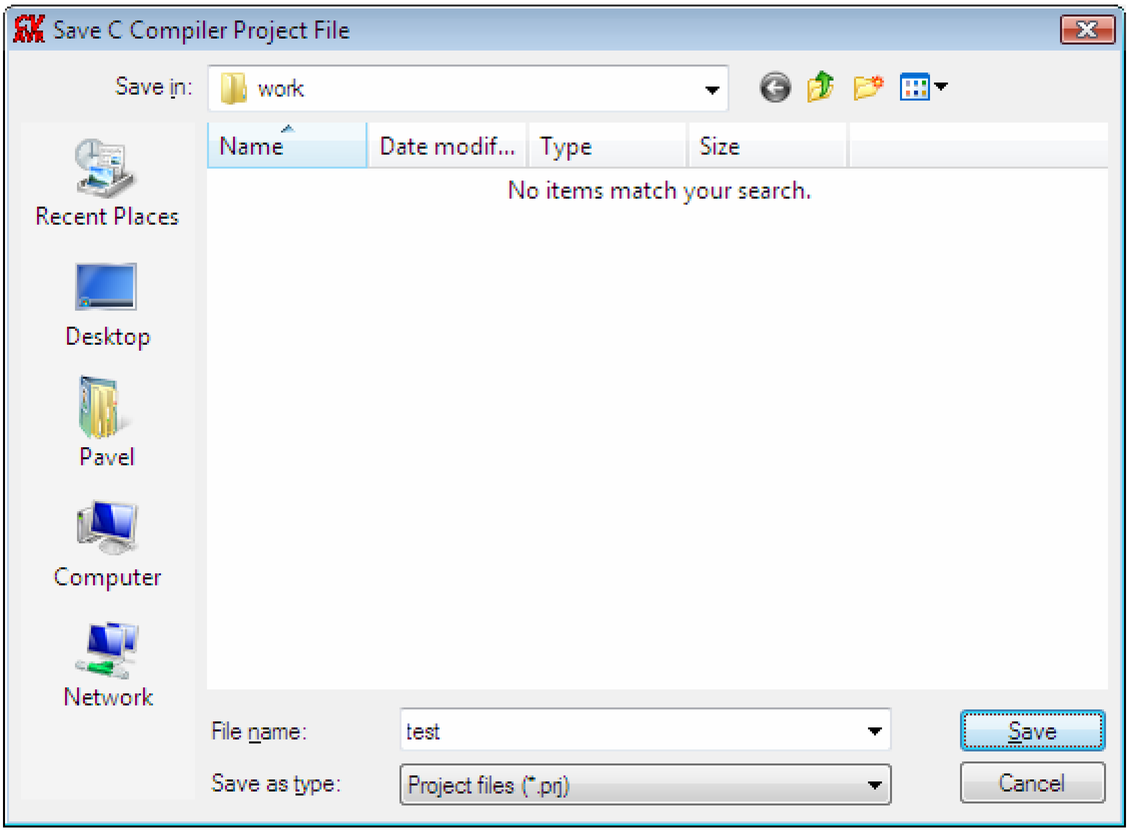

By selecting the **Program|Generate** menu option or by clicking on the toolbar button, the code generated by CodeWizardAVR can be viewed in the **Program Preview** window. This may be useful when applying changes to an existing project, as portions of code generated by the CodeWizardAVR can be selected, copied to the clipboard and then pasted in the project's source files.

If the **Program|Generate, Save and Exit** menu option is selected or the toolbar button is clicked, CodeWizardAVR will generate the main .C source and project .PRJ files, save the CodeWizardAVR project .CWP file and return to the CodeVisionAVR IDE.
Eventual pin function conflicts will be prompted to the user, allowing him to correct the errors.

In the course of program generation the user will be prompted for the name of the main C file:




and for the name of the project file:

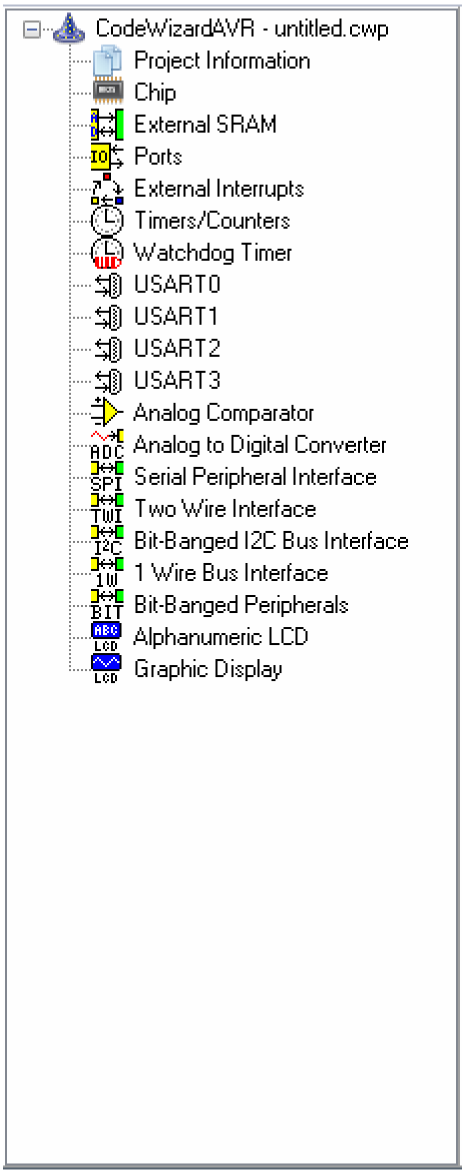


**Note:** When a **.prj** project for the CodeVisionAVR IDE is created, a corresponding **.cproj** project file for Atmel Studio will be created too.
This allows editing/compiling the same project in both Atmel Studio and CodeVisionAVR IDE.

Selecting the **File|Exit** menu option allows the user to exit the CodeWizardAVR without generating any program files.

By selecting the **Help|Help Topics** menu option, by pressing the F1 key or by clicking on the toolbar button, the user can see the help topic that corresponds to the current CodeWizardAVR configuration menu.

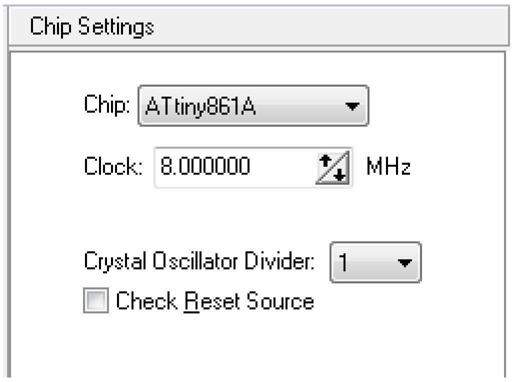
The AVR peripheral that needs to be configured can be selected by clicking on the corresponding node of the CodeWizardAVR selection tree.




If program code was already generated and is available for display in the Program Preview window, clicking on a peripheral node, will position the cursor at the beginning of the initialization code for that peripheral.

## 6.1 Setting the AVR Chip Options

The AVR chip options can be specified by clicking on the **Chip** node of the CodeWizardAVR selection tree.



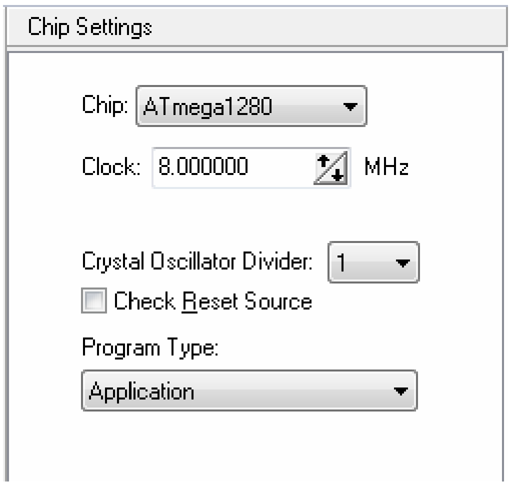

The chip type can be specified using the **Chip** list box.
The chip clock frequency in MHz can be specified using the **Clock** spinedit box.

For the AVR chips that contain a crystal oscillator divider, a supplementary **Crystal Oscillator Divider Enabled** check box is visible.
This check box allows you to enable or disable the crystal oscillator divider.
If the crystal oscillator is enabled, you can specify the division ratio using the **Crystal Oscillator Divider** spinedit box.

For the AVR chips that allow the identification of the reset source, a supplementary **Check Reset Source** check box is visible. If it's checked then the CodeWizardAVR will generate code that allows identification of the conditions that caused the chip reset.

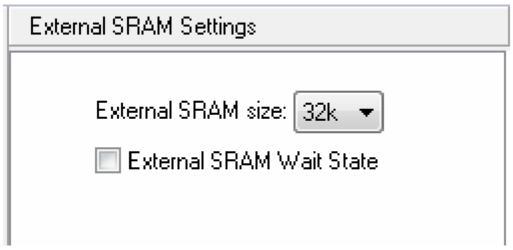
For the AVR chips that allow self-programming, a supplementary **Program Type** list box is visible.
It allows to select the type of the generated code:

- **Application**
- **Boot Loader**



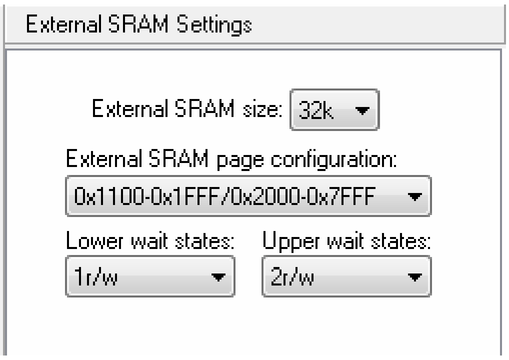
## 6.2 Setting the External SRAM

For the AVR chips that allow connection of external SRAM, you can specify the size of this memory and wait state insertion by clicking on the **External SRAM** node of the CodeWizardAVR selection tree.




The size of external SRAM can be specified using the **External SRAM Size** list box.
Additional wait states in accessing the external SRAM can be inserted by checking the **External SRAM Wait State** check box.
The MCUCR register in the startup initialization code is configured automatically according to these settings.

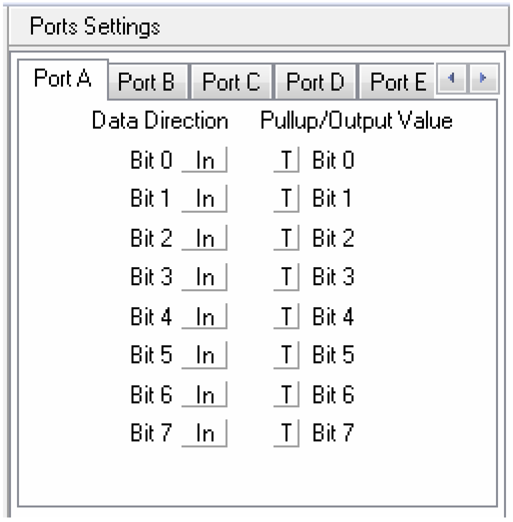
For devices, like the ATmega1280, that allow splitting the external SRAM in two pages, the External SRAM configuration window will look like this:




The **External SRAM page configuration** list box allows selection of the splitting address for the two external SRAM pages .
The wait states that are inserted during external SRAM access, can be specified for the lower, respectively upper, memory pages using the **Lower wait states**, respectively **Upper wait states** list boxes.
The MCUCR, EMCUCR, XMCRA registers in the startup initialization code are configured automatically according to these settings.

## 6.3 Setting the Input/Output Ports

The input/output Ports configuration can be specified by clicking on the **Ports** node of the CodeWizardAVR selection tree.




You can chose which port you want to configure by selecting the appropriate **PORT x** tab.
By clicking on the corresponding **Data Direction** bit you can set the chip pin to be output (Out) or input (In).
The DDRx register will be initialized according to these settings.

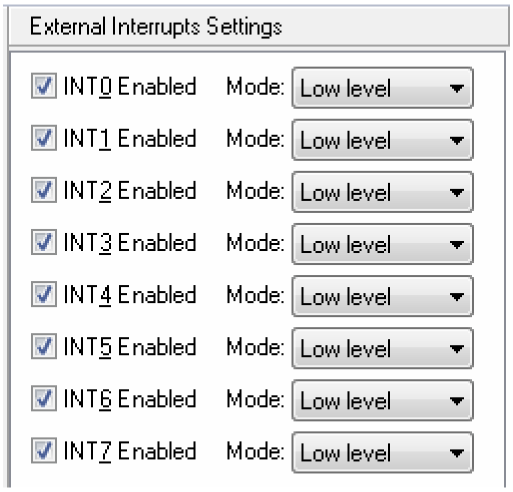
By clicking on the corresponding **Pullup/Output Value** bit you can set the following options:

- if the pin is an input, it can be tri-stated (T) or have an internal pull-up (P) resistor connected to the positive power supply.
- if the pin is an output, its value can be initially set to 0 or 1.

The PORTx register will be initialized according to these settings.

## 6.4 Setting the External Interrupts

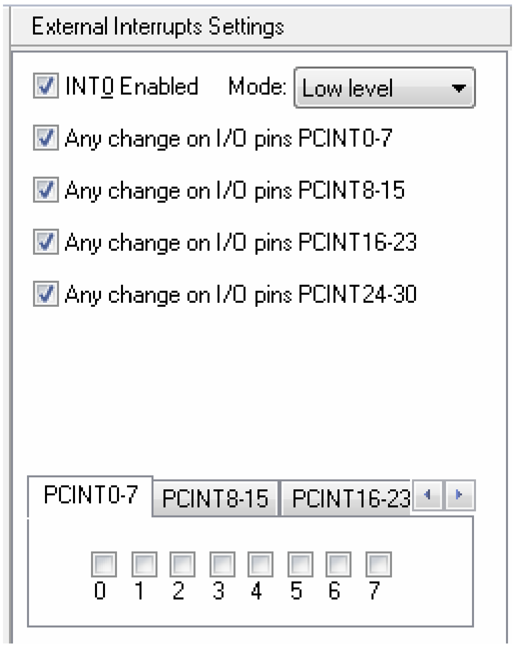
The external interrupt configuration can be specified by clicking on the **External IRQ** node of the CodeWizardAVR selection tree.




Checking the appropriate **INTx Enabled** check box enables the corresponding external interrupt.
If the AVR chip supports this feature, you can select if the interrupt will be edge or level triggered using the corresponding **Mode** list box.

For each enabled external interrupt the CodeWizardAVR will define an **ext_intx_isr** interrupt service routine, where **x** is the number of the external interrupt.

For some devices, like the ATmega3290, the following options may be present:



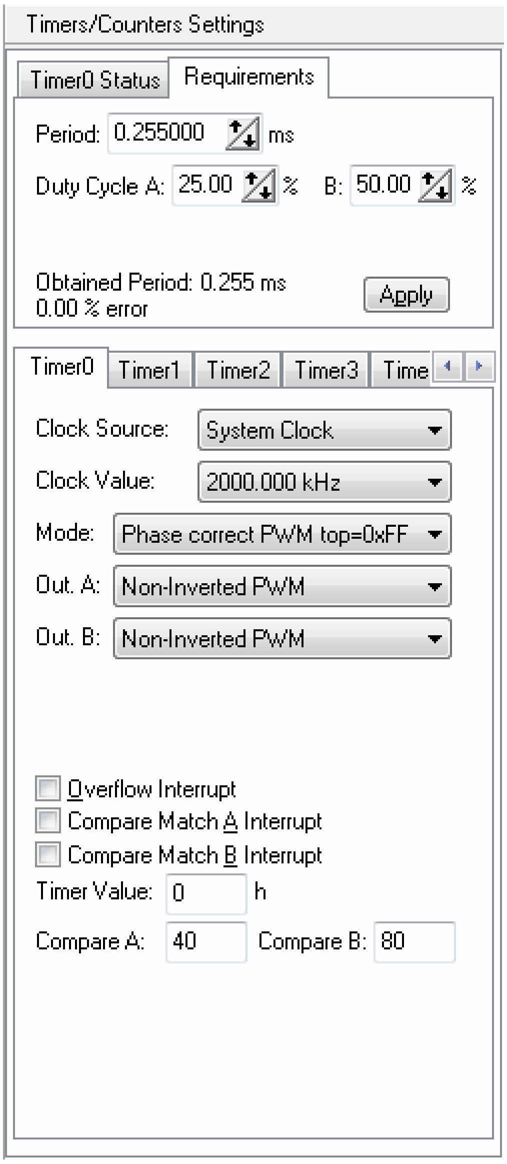

The **Any change on I/O pins** check boxes, if checked, will specify which of the PCINT I/O pins will trigger an external interrupt.
The interrupt service routines for these interrupts will be **pin_change_isr0** for PCINT0-7, **pin_change_isr1** for PCINT8-15, **pin_change_isr2** for PCINT16-23 and **pin_change_isr3** for PCINT24-30.

## 6.5 Setting the Timers/Counters

The timers/counters configuration can be specified by clicking on the **Timers/Counters** node of the CodeWizardAVR selection tree.

A number of **Timer** tabs will be displayed according to the AVR chip type.

By selecting the **Timer 0** tab you can have the following options:




- **Clock Source** specifies the timer/counter 0 clock pulse source
- **Clock Value** specifies the timer/counter 0 clock frequency
- **Mode** specifies if the timer/counter 0 functioning mode
- **Outp. A** specifies the function of the timer/counter 0 compare A output and depends of the functioning mode
- **Outp. B** specifies the function of the timer/counter 0 compare B output and depends of the functioning mode
- **Overflow Interrupt** specifies if an interrupt is to be generated on timer/counter 0 overflow

- **Compare Match A Interrupt** specifies if an interrupt is to be generated on timer/counter 0 compare A match
- **Compare Match B Interrupt** specifies if an interrupt is to be generated on timer/counter 0 compare B match
- **Timer Value** specifies the initial value of timer/counter 0 at startup
- **Compare A** specifies the initial value of timer/counter 0 output compare A register
- **Compare B** specifies the initial value of timer/counter 0 output compare B register.

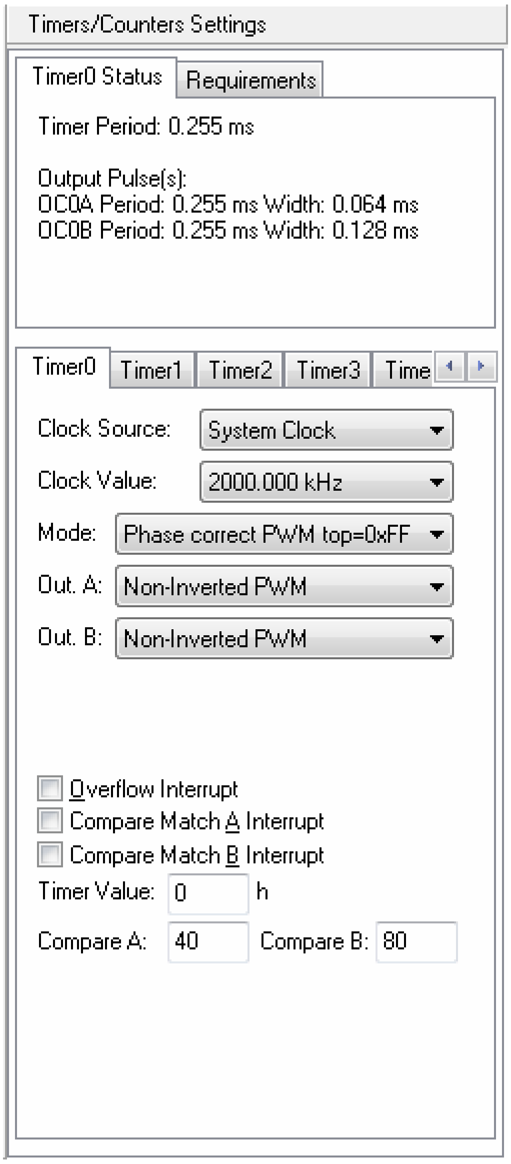
If timer/counter 0 interrupts are used the following interrupt service routines may be defined by the CodeWizardAVR:

- **timer0_ovf_isr** for timer/counter overflow
- **timer0_compa_isr** for timer/counter output compare A match
- **timer0_compb_isr** for timer/counter output compare B match

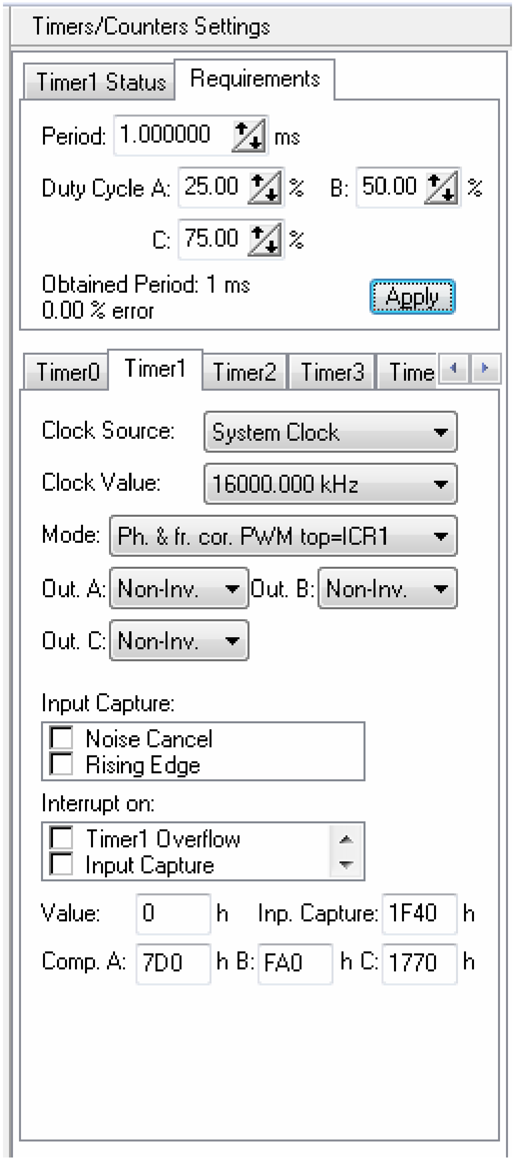
**Note:** Depending of the used AVR chip some of these options may not be present.
For more information you must consult the corresponding Atmel data sheet.

The CodeWizardAVR features an automatic timer configurator which is invoked by selecting the **Requirements** tab.
Here the user can enter the required timer period in ms and the duty cycle(s) for the enabled timer's outputs for PWM modes.
By pressing the **Apply** button all the corresponding timer configuration registers are set accordingly, in order to obtain the closest possible values for the required parameters, using the timer operating mode specified by the **Mode** list box.

Selecting the **Timer 0 Status** tab allows to see the obtained output pulse parameters for the currently set timer clock frequency, operating mode and configuration registers values:



By selecting the **Timer 1** tab you can have the following options:



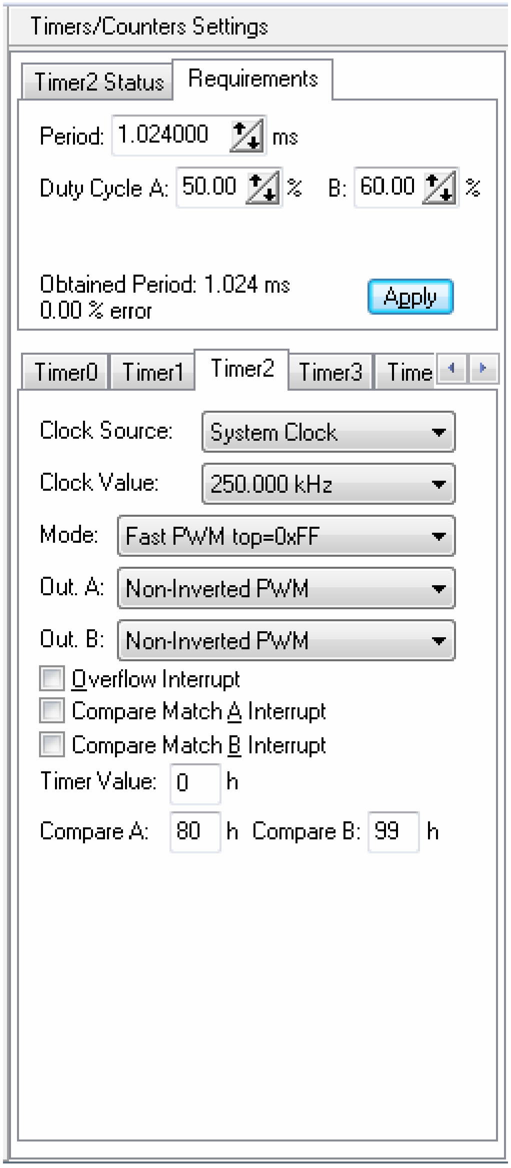

- **Clock Source** specifies the timer/counter 1 clock pulse source
- **Clock Value** specifies the timer/counter 1 clock frequency
- **Mode** specifies if the timer/counter 1 functioning mode
- **Out. A** specifies the function of the timer/counter 1 output A and depends of the functioning mode
- **Out. B** specifies the function of the timer/counter 1 output B and depends of the functioning mode
- **Out. C** specifies the function of the timer/counter 1 output C and depends of the functioning mode
- **Inp Capt.** specifies the timer/counter 1 capture trigger edge and if the noise canceler is to be used
- **Interrupt on** specifies if an interrupt is to be generated on timer/counter 1 overflow, input capture and compare match
- **Timer Value** specifies the initial value of timer/counter 1 at startup
- **Comp. A, B** and **C** specifies the initial value of timer/counter 1 output compare registers A, B and C.

If timer/counter 1 interrupts are used, the following interrupt service routines may be defined by the CodeWizardAVR:

- **timer1_ovf_isr** for timer/counter overflow
- **timer1_comp_isr** or **timer1_compa_isr**, **timer1_compb_isr** and **timer1_copmc_isr** for timer/counter output compare match
- **timer1_capt_isr** for timer/counter input capture

**Note:** Depending of the used AVR chip some of these options may not be present.
For more information you must consult the corresponding Atmel data sheet.

By selecting the **Timer 2** tab you can have the following options:



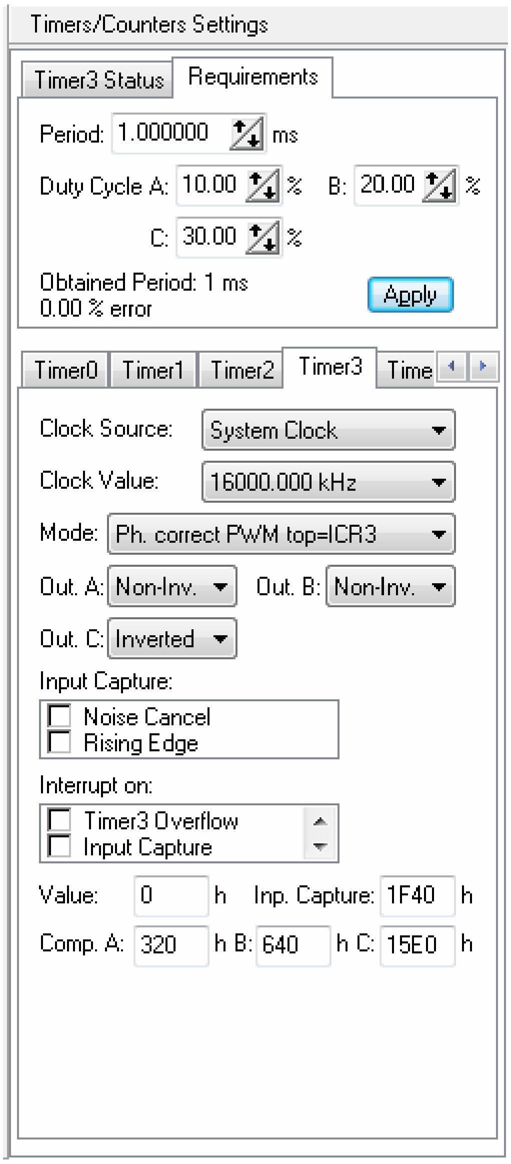

- **Clock Source** specifies the timer/counter 2 clock pulse source
- **Clock Value** specifies the timer/counter 2 clock frequency
- **Mode** specifies if the timer/counter 2 functioning mode
- **Out. A** specifies the function of the timer/counter 2 output A and depends of the functioning mode
- **Out. B** specifies the function of the timer/counter 2 output B and depends of the functioning mode
- **Overflow Interrupt** specifies if an interrupt is to be generated on timer/counter 2 overflow
- **Compare Match A Interrupt** specifies if an interrupt is to be generated on timer/counter 2 compare register A match
- **Compare Match B Interrupt** specifies if an interrupt is to be generated on timer/counter 2 compare register B match
- **Timer Value** specifies the initial value of timer/counter 2 at startup
- **Compare A** specifies the initial value of timer/counter 2 output compare A register
- **Compare B** specifies the initial value of timer/counter 2 output compare B register

If timer/counter 2 interrupts are used, the following interrupt service routines may be defined by the CodeWizardAVR:

- **timer2_ovf_isr** for timer/counter overflow
- **timer2_comp_isra** and **timer2_compb_isr** for timer/counter output compare match.

**Note:** Depending of the used AVR chip some of these options may not be present.
For more information you must consult the corresponding Atmel data sheet.

By selecting the **Timer 3** tab you can have the following options:




- **Clock Source** specifies the timer/counter 3 clock pulse source
- **Clock Value** specifies the timer/counter 3 clock frequency
- **Mode** specifies if the timer/counter 3 functioning mode
- **Out. A** specifies the function of the timer/counter 3 output A and depends of the functioning mode
- **Out. B** specifies the function of the timer/counter 3 output B and depends of the functioning mode
- **Out. C** specifies the function of the timer/counter 3 output C and depends of the functioning mode
- **Inp Capt.** specifies the timer/counter 3 capture trigger edge and if the noise canceler is to be used
- **Interrupt on** specifies if an interrupt is to be generated on timer/counter 3 overflow, input capture and compare match
- **Timer Value** specifies the initial value of timer/counter 3 at startup
- **Comp. A, B** and **C** specifies the initial value of timer/counter 3 output compare registers A, B and C.

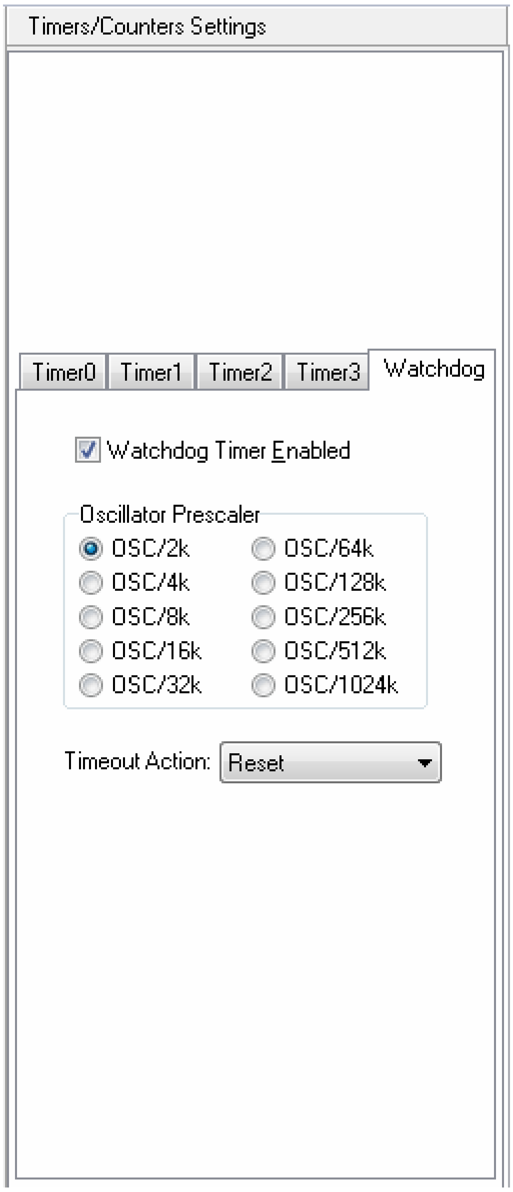
If timer/counter 3 interrupts are used, the following interrupt service routines may be defined by the CodeWizardAVR:

- **timer3_ovf_isr** for timer/counter overflow
- **timer3_comp_isr** or **timer3_compa_isr**, **timer3_compb_isr** and **timer3_compc_isr** for timer/counter output compare match
- **timer3_capt_isr** for timer/counter input capture

**Notes:**

- Depending of the used AVR chip some of these options may not be present.
- Some AVR chips may have additional timers, which can be configured the same way as described above.

By selecting the **Watchdog** tab you can configure the watchdog timer.




Checking the **Watchdog Timer Enabled** check box activates the watchdog timer.
You will have then the possibility to set the watchdog timer's **Oscillator Prescaller**.

The **Timeout Action** list box allows to specify what action will be performed on watchdog timer overflow:

- hardware **Reset**
- **Interrupt** only
- **Interrupt** and then hardware **Reset**.

If  interrupt generation is enabled, the **wdt_timeout_isr** interrupt service function will be created.

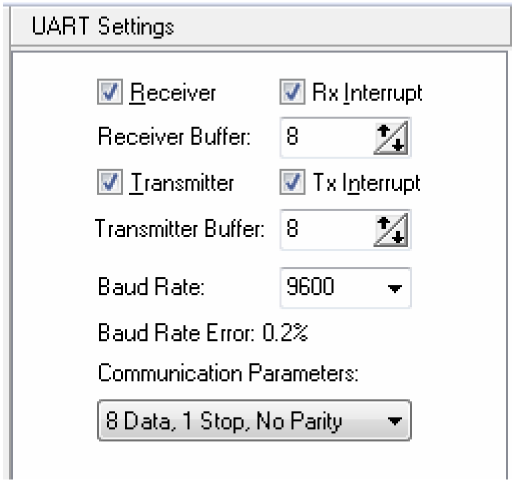
In case the watchdog timer is enabled, you must include yourself the appropriate code sequences to reset it periodically. Example:

```
#asm("wdr")
```

For more information about the watchdog timer you must consult the Atmel data sheet for the chip that you use.

## 6.6 Setting the UART or USART

The UART or USART configuration can be specified by clicking on the **USART** node(s) of the CodeWizardAVR selection tree.




Checking the **Receiver** check box activates the UART receiver.
The receiver can function in the following modes:

- polled, the **Rx Interrupt** check box isn't checked
- interrupt driven circular buffer, the **Rx Interrupt** check box is checked.

In the interrupt driven mode you can specify the size of the circular buffer using the **Receiver Buffer** spinedit box.

Checking the **Transmitter** check box activates the UART transmitter.
The transmitter can function in the following modes:

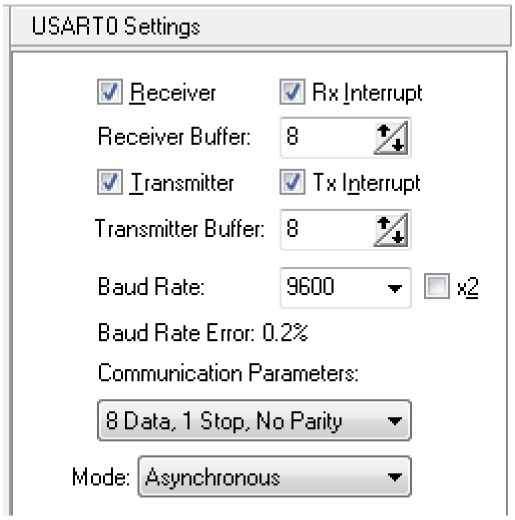
- polled, the **Tx Interrupt** check box isn't checked
- interrupt driven circular buffer, the **Tx Interrupt** check box is checked.

In the interrupt driven mode you can specify the size of the circular buffer using the **Transmitter Buffer** spinedit box.

The communication Baud rate can be specified using the **UART Baud Rate** list box.
CodeWizardAVR will automatically set the UBRR according to the Baud rate and AVR chip clock frequency. The Baud rate error for these parameters will be calculated and displayed.

The **Communications Parameters** list box allows you to specify the number of data bits, stop bits and parity used for serial communication.

For devices featuring an **USART** there will be an additional **Mode** list box.




It allows you to specify the following communication modes:
- Asynchronous
- Synchronous Master, with the UCSRC register's UCPOL bit set to 0
- Synchronous Master, with the UCSRC register's UCPOL bit set to 1
- Synchronous Slave, with the UCSRC register's UCPOL bit set to 0
- Synchronous Slave, with the UCSRC register's UCPOL bit set to 1.

The serial communication is realized using the Standard Input/Output Functions **getchar**, **gets**, **scanf**, **putchar**, **puts** and **printf**.
For interrupt driven serial communication, CodeWizardAVR automatically redefines the basic **getchar** and **putchar** functions.

The receiver buffer is implemented using the global array **rx_buffer**.
The global variable **rx_wr_index** is the **rx_buffer** array index used for writing received characters in the buffer.
The global variable **rx_rd_index** is the **rx_buffer** array index used for reading received characters from the buffer by the **getchar** function.
The global variable **rx_counter** contains the number of characters received in **rx_buffer** and not yet read by the **getchar** function.
If the receiver buffers overflows the **rx_buffer_overflow** global bit variable will be set.

The transmitter buffer is implemented using the global array **tx_buffer**.
The global variable **tx_wr_index** is the **tx_buffer** array index used for writing in the buffer the characters to be transmitted.
The global variable **tx_rd_index** is the **tx_buffer** array index used for reading from the buffer the characters to be transmitted by the **putchar** function.
The global variable **tx_counter** contains the number of characters from **tx_buffer** not yet transmitted by the interrupt system.

For devices with 2 UARTs, respectively 2 USARTs, there will be two tabs present: **UART0** and **UART1**, respectively **USART0** and **USART1**.
The functions of configuration check and list boxes will be the same as described above.

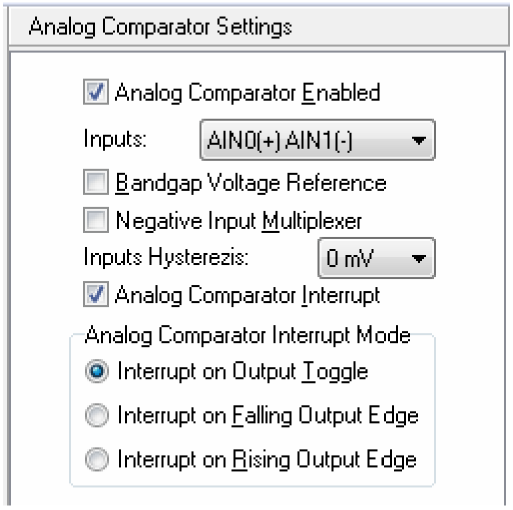
The UART0 (USART0) will use the normal **putchar** and **getchar** functions.
In case of interrupt driven buffered communication, UART0 (USART0) will use the following variables:
**rx_buffer0**, **rx_wr_index0**, **rx_rd_index0**, **rx_counter0**, **rx_buffer_overflow0**, **tx_buffer0**, **tx_wr_index0**, **tx_rd_index0**, **tx_counter0**.

The UART1 (USART1) will use the **putchar1** and **getchar1** functions.
In case of interrupt driven buffered communication, UART1 (USART1) will use the following variables:
**rx_buffer1**, **rx_wr_index1**, **rx_rd_index1**, **rx_counter1**, **rx_buffer_overflow1**,
**tx_buffer1**, **tx_wr_index1**, **tx_rd_index1**, **tx_counter1**.

All serial I/O using functions declared in **stdio.h**, will be done using UART0 (USART0).

## 6.7 Setting the Analog Comparator

The configuration of the **Analog Comparator** can be specified by clicking on the corresponding node of the CodeWizardAVR selection tree.




Checking the **Analog Comparator Enabled** check box enables the on-chip analog comparator.
Checking the **Bandgap Voltage Reference** check box will connect an internal voltage reference to the analog comparator's positive input.

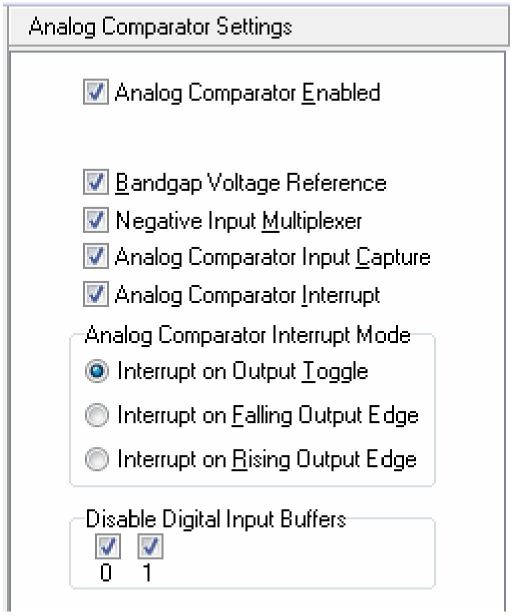
Checking the **Negative Input Multiplexer** check box will connect the analog comparator's negative input to the ADC's analog multiplexer.
If the **Negative Input Multiplexer** option is not enabled, the **Inputs** list box allows to select which of the ADC's analog multiplexer inputs will be connected to the analog comparator's positive and negative inputs.

The **Inputs Hysterezis** list box allows to select the amount of hysterezis of the analog comparator inputs.

If you want to generate interrupts if the analog comparator's output changes state, then you must check the **Analog Comparator Interrupt** check box.
The type of output change that triggers the interrupt can be specified in the **Analog Comparator Interrupt Mode** settings.

For some AVR chips the analog comparator's output may be to be used for capturing the state of timer/counter 1.
In this case the **Analog Comparator Input Capture** check box must be checked if present.

The **Disable Digital Input Buffer on AIN0**, respectively **Disable Digital Input Buffer on AIN1** check boxes, if checked, will deactivate the digital input buffers on the AIN0, respectively AIN1 pins, thus reducing the power consumption of the chip.

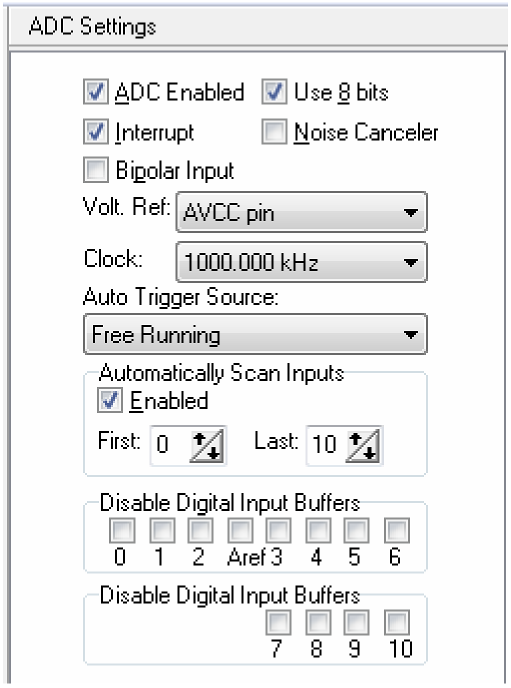



The corresponding bits in the PIN registers will always read 0 in this case.

Some of this check boxes may not be present on all the AVR chips.
If the analog comparator interrupt is enabled, the CodeWizardAVR will define the **ana_comp_isr** interrupt service routine.

## 6.8 Setting the Analog to Digital Converter

Some AVR chips contain an Analog to Digital Converter (ADC).
The ADC configuration can be specified by clicking on the **Analog to Digital Converter** node of the CodeWizardAVR selection tree.




Checking the **ADC Enabled** check box enables the on-chip ADC.

On some AVR devices only the 8 most significant bits of the AD conversion result can be used. This feature is enabled by checking the **Use 8 bits** check box.

The ADC may be operated in bipolar mode if the **Bipolar Input** check box is checked.

Some AVR devices allow the ADC to use a high speed conversion mode, but with lower precision. This feature is enabled by checking the **High Speed** check box, if present.

If the ADC has an internal reference voltage source, than it can be selected using the **Volt. Ref.** list box or activated by checking the **ADC Bandgap** check box.

The ADC clock frequency can be selected using the **Clock** list box.

If you want to generate interrupts when the ADC finishes the conversion, then you must check the **Interrupt** check box.

If ADC interrupts are used you have the possibility to enable the following functions:

- by checking the **Noise Canceler** check box, the chip is placed in idle mode during the conversion process, thus reducing the noise induced on the ADC by the chip's digital circuitry
- by checking the **Automatically Scan Inputs Enabled** check box, the CodeWizardAVR will generate code to scan an ADC input domain and put the results in an array. The start, respectively the end, of the domain are specified using the **First Input**, respectively the **Last Input**, spinedit boxes.

Some AVR devices allow the AD conversion to be triggered by an event which can be selected using the **Auto Trigger Source** list box.
If the automatic inputs scanning is disabled, then a single analog-digital conversion can be executed using the function:

**unsigned int read_adc(unsigned char adc_input)**

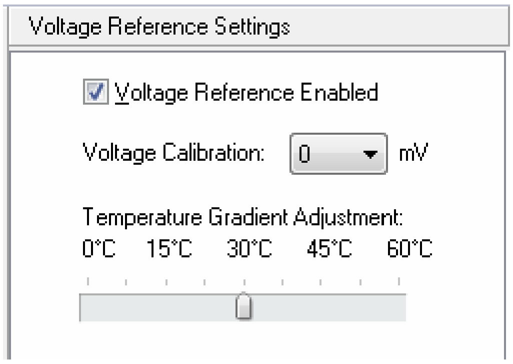
This function will return the analog-digital conversion result for the input **adc_input**. The input numbering starts from 0.
If interrupts are enabled the above function will use an additional interrupt service routine **adc_isr**. This routine will store the conversion result in the **adc_data** global variable.

If the automatic inputs scanning is enabled, the **adc_isr** service routine will store the conversion results in the **adc_data** global array. The user program must read the conversion results from this array.

For some chips there is also the possibility to disable the digital input buffers on the inputs used by the ADC, thus reducing the power consumption of the chip.
This is accomplished by checking the corresponding **Disable Digital Input Buffers** check boxes.
If the **Automatically Scan Inputs** option is enabled, then the corresponding digital input buffers are automatically disabled for the ADC inputs in the scan range.

## 6.9 Setting the ATmega406 Voltage Reference

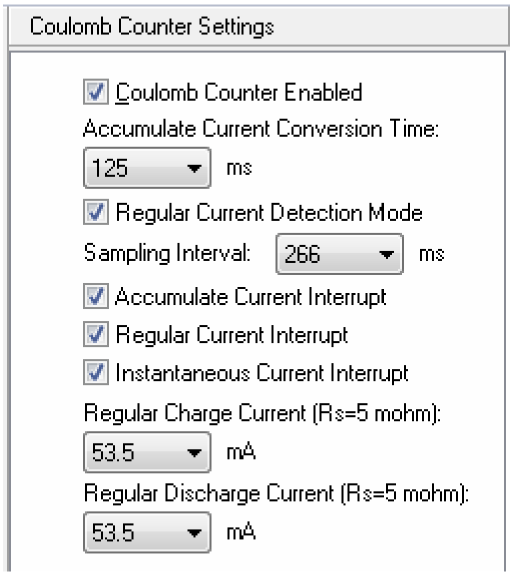
Some AVR chips, like the ATmega406, contain a low power precision bang-gap voltage reference, which can be configured by clicking on the **Voltage Reference** node of the CodeWizardAVR selection tree.




Checking the **Voltage Reference Enabled** check box enables the precision voltage reference.
The **Voltage Calibration** list box allows for precision adjustment of the nominal value of the reference voltage in 2mV steps.
The **Temperature Gradient Adjustment** slider allows shifting the top of the $V_{REF}$ versus temperature curve to the center of the temperature range of interest, thus minimizing the voltage drift in this range.
The Atmega406 datasheet may be consulted for more details.

## 6.10 Setting the ATmega406 Coulomb Counter

The ATmega406 chip, contains a dedicated Sigma-Delta ADC optimized for Coulomb Counting to sample the charge or discharge current flowing through an external sense resistor Rs.
This ADC can be configured by clicking on the **Coulomb Counter** node of the CodeWizardAVR selection tree.




Checking the **Coulomb Counter Enabled** check box enables the Coulomb Counter Sigma-Delta ADC.
The **Accumulate Current Conversion Time** list box specifies the conversion time for the Accumulate Current output.
The **Regular Current Detection Mode** check box specifies that the Coulomb Counter will repeatedly do one instantaneous current conversion, before it is turned of for a timing interval specified by the **Sampling Interval** list box.
The interval selected using the above-mentioned list box includes a sampling time, having a typical value of 16ms.

The **Accumulate Current Interrupt** check box enable the generation of an interrupt after the accumulate current conversion has completed. This interrupt is serviced by the **ccadc_acc_isr** ISR.

The **Regular Current Interrupt** check box enable the generation of an interrupt when the absolute value of the result of the last AD conversion is greater, or equal to, the values of the CADRCC and CADRDC registers. This interrupt is serviced by the **ccadc_reg_cur_isr** ISR.

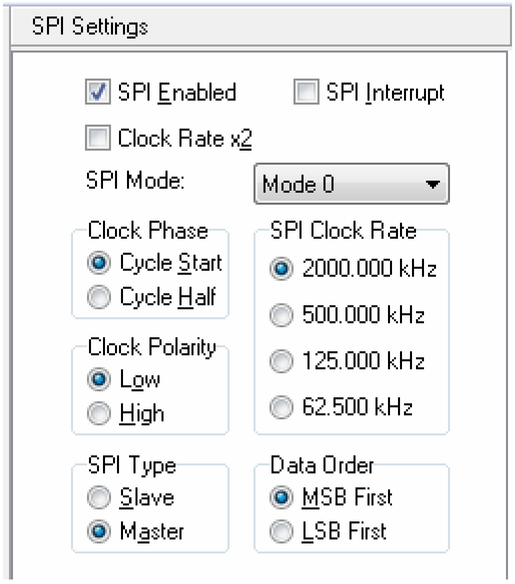
The **Instantaneous Current Interrupt** check box enables the generation of an interrupt when an instantaneous current conversion has completed. This interrupt is serviced by the **ccadc_conv_isr** ISR.

The **Regular Charge Current**, respectively **Regular Discharge Current**, list boxes determine the threshold levels for the *regular charge*, respectively *regular discharge* currents, setting the values for the CADRCC, respectively CADRDC, registers used for generating the *Regular Current Interrupt*.

The Atmega406 datasheet may be consulted for more details about the Coulomb Counter.

## 6.11 Setting the SPI Interface

The SPI interface configuration can be specified by clicking on the **Serial Peripheral Interface** node of the CodeWizardAVR selection tree.




Checking the **SPI Enabled** check box enables the on-chip SPI interface.

If you want to generate interrupts upon completion of a SPI transfer, then you must check the **SPI Interrupt** check box.

You have the possibility to specify the following parameters:

- **SPI Clock Rate** used for the serial transfer
- **Clock Phase**: the position of the SCK strobe signal edge relative to the data bit
- **Clock Polarity**: low or high in idle state
- **SPI Type**: the AVR chip is master or slave
- **Data Order** in the serial transfer.

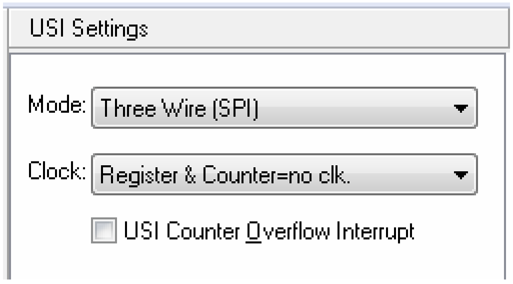
Checking the **Clock Rate x2** check box, available for some AVR chips, will double the **SPI Clock Rate**.

For communicating through the SPI interface, with disabled SPI interrupt, you must use the **SPI Functions**.
If the SPI interrupt is enabled, you must use the **spi_isr** interrupt service routine, declared by the CodeWizardAVR.

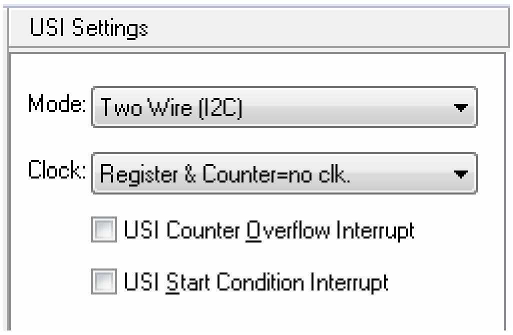
## 6.12 Setting the Universal Serial Interface - USI

The **USI** configuration can be specified by clicking on the corresponding node of the CodeWizardAVR selection tree.

The USI operating mode can be selected using the **Mode** list box.
One of the USI operating modes is the **Three Wire (SPI)** mode:




The USI can also operate in the **Two Wire (I2C)** mode:




The **Shift Reg. Clock** list box sets the clock source for the USI Shift Register and Counter.

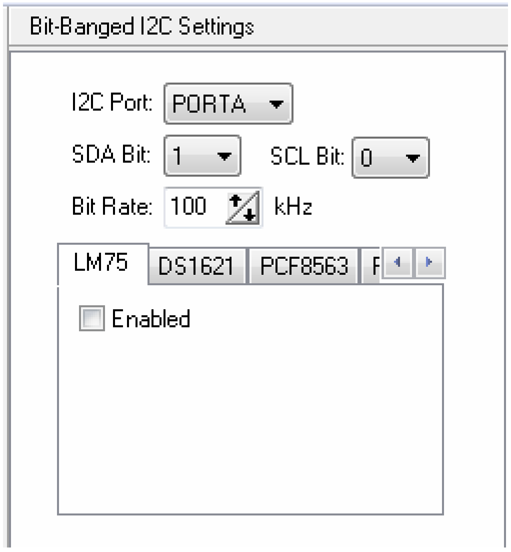
As both the USI Shift Register and Counter are clocked from the same clock source, the USI Counter may be used to count the number of received or transmitted bits and generate an overflow interrupt when the data transfer is complete.
Checking the **USI Counter Overflow Interrupt** check box will generate code for an interrupt service routine that will be executed upon the overflow of the USI Counter.

If the **USI Start Condition Interrupt** check box is checked then the CodeWizardAVR will generate code for an interrupt service routine that will be executed when a Start Condition is detected on the I2C bus in USI **Two Wire** operating mode.

## 6.13 Setting the $I^2C$ Bus

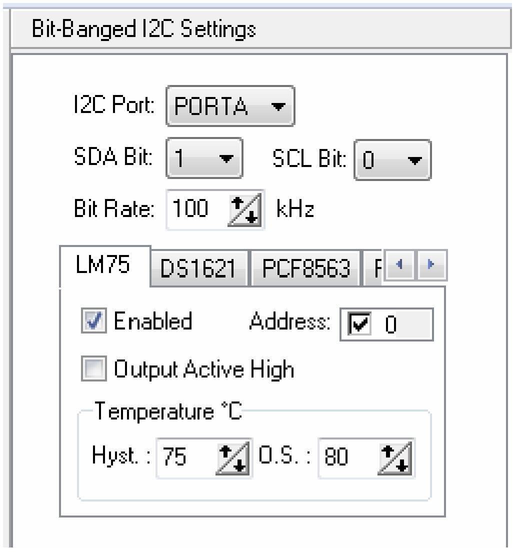
The $I^2C$ bus configuration can be specified by clicking on the **Bit-Banged $I^2C$ Bus Interface** node of the CodeWizardAVR selection tree.




Using the **$I^2C$ Port** list box you can specify which port is used for the implementation of the $I^2C$ bus. The **SDA Bit** and **SCL Bit** list boxes allow you to specify which port bits the $I^2C$ bus uses.

## 6.13.1 Setting the LM75 devices

If you use the LM75 temperature sensor, you must select the **LM75** tab and check the **LM75 Enabled** check box.

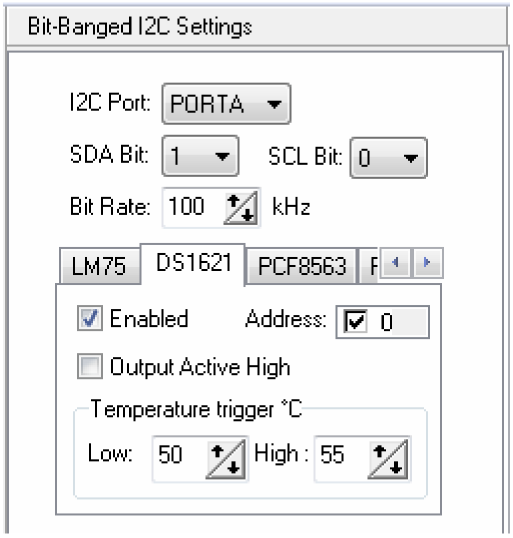



The **LM75 Address** list box allows you to specify the 3 lower bits of the $I^2C$ addresses of the LM75 devices connected to the bus. Maximum 8 LM75 devices can be used.
The **Output Active High** check box specifies the active state of the LM75 O.S. output.
The **Hyst.**, respectively **O.S.** , spinedit boxes specify the hysterezis, respectively O.S. temperatures.

The LM75 devices are accessed through the **National Semiconductor LM75 Temperature Sensor Functions**.

## 6.13.2 Setting the DS1621 devices

If you use the DS1621 thermometer/thermostat, you must select the **DS1621** tab and check the **DS1621 Enabled** check box.

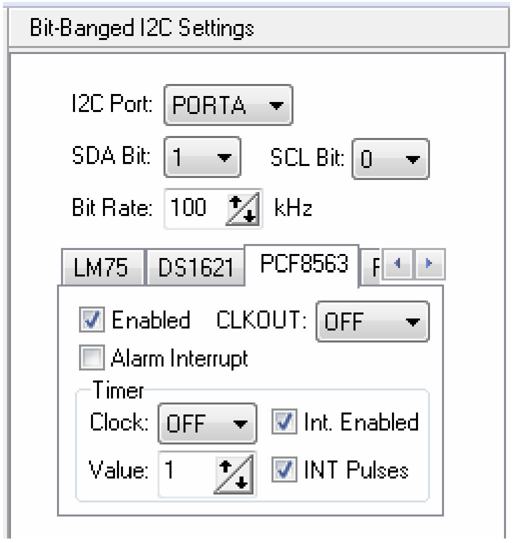



The **Output Active High** check box specifies the active state of the DS1621 Tout output.
The **Low**, respectively **High**, spinedit boxes specify the low, respectively high temperatures trigger temperatures for the Tout output.

The DS1621 devices are accessed through the **Maxim/Dallas Semiconductor DS1621 Thermometer/Thermostat** functions.

### 6.13.3 Setting the PCF8563 devices

If you use the PCF8563 RTC, you must select the PCF8563 tab and check the PCF8563 Enabled check box.

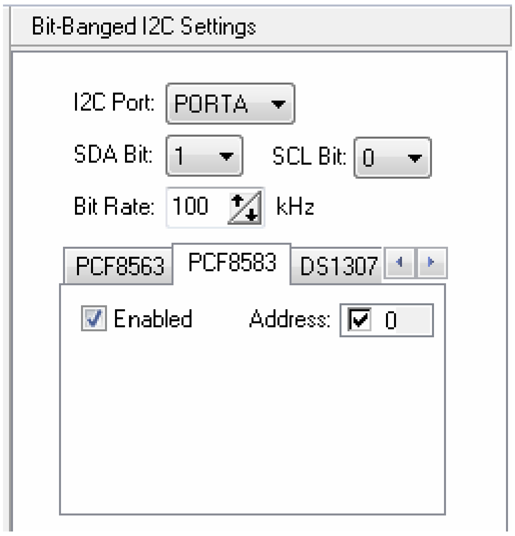



The **CLKOUT** list box specifies the frequency of the pulses on the CLKOUT output.
The **Alarm Interrupt** check box enables the generation of interrupts, on the INT pin, when the alarm conditions are met.
The **Timer|Clock** list box specifies the countdown frequency of the PCF8563 Timer.
If the **Int. Enabled** check box is checked, an interrupt will be generated when the Timer countdown value will be 0.
If the **INT Pulses** check box is checked, the INT pin will issue short pulses when the Timer countdown value reaches 0.
The **Timer|Value** spinedit box specifies the Timer reload value when the countdown reaches 0.

The PCF8563 devices are accessed through the **Philips PCF8563 Real Time Clock Functions**.

## 6.13.4 Setting the PCF8583 devices

If you use the PCF8583 RTC, you must select the **PCF8583** tab and check the **PCF8583 Enabled** check box.

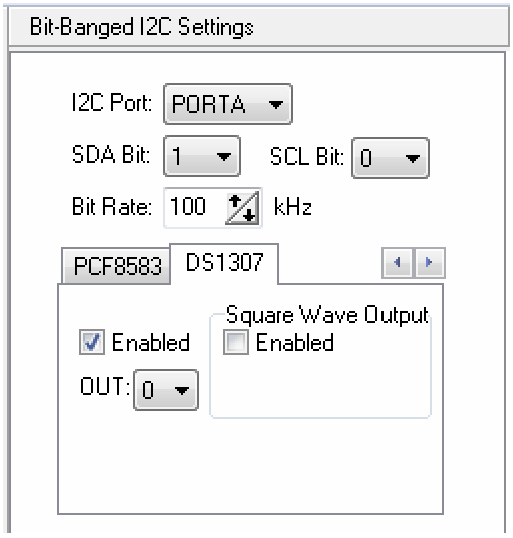



The **PCF8583 Address** list box allows you to specify the low bit of the $I^2C$ addresses of the PCF8583 devices connected to the bus. Maximum 2 PCF8583 devices can be used.

The PCF8583 devices are accessed through the **Philips PCF8583 Real Time Clock Functions**.

## 6.13.5 Setting the DS1307 devices

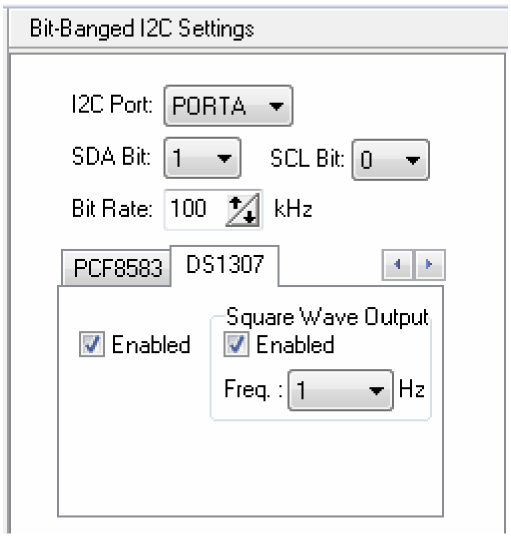
If you use the DS1307 RTC, you must select the **DS1307** tab and check the **DS1307 Enabled** check box.




The DS1307 device is accessed through the **Maxim/Dallas Semiconductor DS1307 Real Time Clock Functions**.

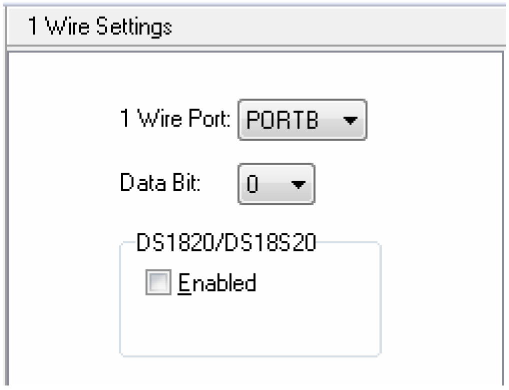
In case the square wave signal output is disabled, the state of the SQW/OUT pin can be specified using the **OUT** list box.

By checking the **Square Wave Output Enabled** check box a square wave signal will be available on the DS1307's SQW/OUT pin. The frequency of the square wave can be selected using the **Freq.** list box:



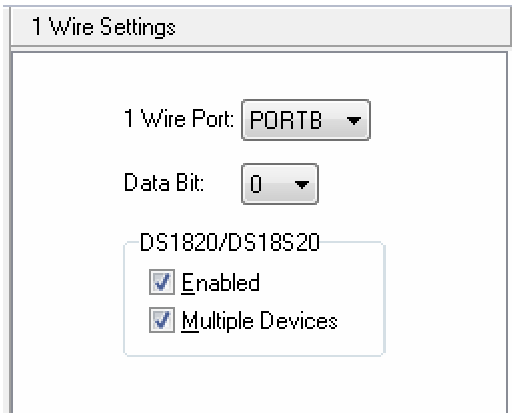
## 6.14 Setting the 1 Wire Bus

The 1 Wire bus configuration can be specified by clicking on the **1 Wire Bus Interface** node of the CodeWizardAVR selection tree.




Using the **1 Wire Port** list box you can specify which port is used for the implementation of the 1 Wire bus.
The **Data Bit** list box allows you to specify which port bit the 1 Wire bus uses.

If you use the DS1820/DS18S20 temperature sensors, you must check the **DS1820/DS18S20 Enabled** check box.

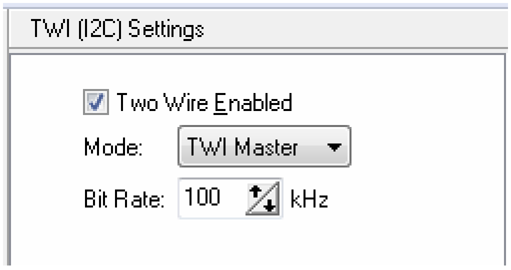



If you use several DS1820/DS18S20 devices connected to the 1 Wire bus, you must check the **Multiple Devices** check box. Maximum 8 DS1820/DS18S20 devices can be connected to the bus.
The ROM codes for these devices will be stored in the **ds1820_rom_codes** array.

The DS1820/DS18S20 devices can be accessed using the **Maxim/Dallas Semiconductor DS1820/DS18S20 Temperature Sensors Functions**.

## 6.15 Setting the Two Wire Bus Interface

The **Two Wire Interface** configuration can be specified by clicking on the corresponding node of the CodeWizardAVR selection tree.

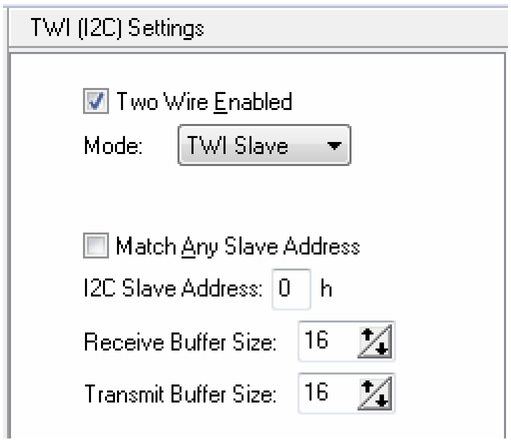



The AVR chip's Two Wire interface can be enabled by checking the **Two Wire Enabled** check box.
One of the two operating modes can be selected:

- **TWI Master**
- **TWI Slave**.

In **TWI Master** mode the **Bit Rate** list box allows to specify maximum frequency of the pulses on the SCL Two Wire bus line. This value will be passed to the **twi_master_init** function (**twi.h**), called for initialization, and will affect the value of the **TWBR** register.
When operating in **TWI Master** mode the **twi_master_trans** function from **twi.h** must be used for bus communication.

In **TWI Slave** mode the following options are available:




The **Slave Address** edit box sets the 7 bit slave address of the Two Wire serial bus unit. This address must be specified in hexadecimal and will be used to initialize the bits 1..7 of the TWAR register.

Checking the **Match Any Slave Address** check box, enables the slave to acknowledge for any slave address issued by the master.

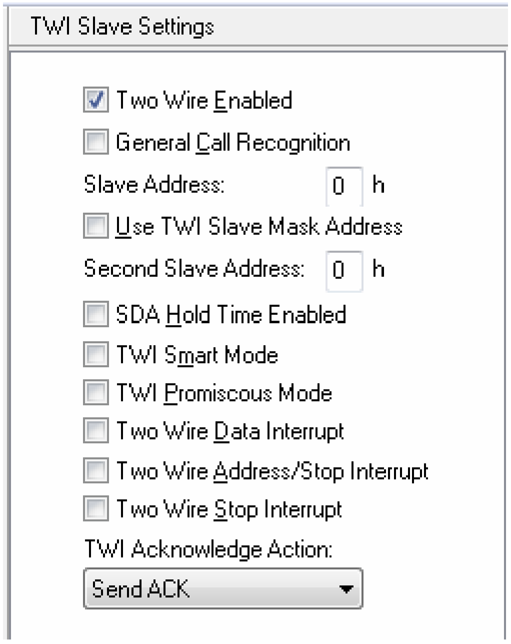
The sizes of the **Receive** and **Transmit Buffers** can be set accordingly.

All the above mentioned options will be passed to the **twi_slave_init** function (**twi.h**), called for initialization.

The CodeWizardAVR will also generate code for the **twi_rx_handler** and **twi_tx_handler** functions used for TWI slave reception and transmission.
An example of usage for these functions can be found in the chapter **4.13.2 Two Wire Interface Functions for Slave Mode Operation.**

## 6.16 Setting the Two Wire Bus Slave Interface for the ATtiny20/40 chips

The **Two Wire Interface** slave configuration for the ATtiny20/40 chips can be specified by clicking on the corresponding TWI node of the CodeWizardAVR selection tree.




The AVR chip's Two Wire Slave interface can be enabled by checking the **Two Wire Enabled** check box.
If the **General Call Recognition** check box is checked the general call recognition logic is enabled and bit 0 of the TWSA register will be set.

The **Slave Address** edit box sets the slave address of the Two Wire serial bus unit. This address must be specified in hexadecimal and will be used to initialize the bits 1..7 of the TWSA register.

If the **Use TWI Slave Mask Address** option is enabled, the contents of bits 1..7 of the TWSAM register will be used to mask (disable) the corresponding bits in the TWSA register. If the mask bit is one, the address match between the incoming address bit and the corresponding bit in TWSA is ignored. In other words, masked bits will always match.

If the **Use TWI Slave Mask Address** option is disabled, the contents of bits 1..7 of the TWSAM register will be used as a second slave address. In this mode, the slave will match on two unique addresses, one in TWSA register and the other in TWSAM register.

The **Second Slave Address** or **Slave Mask Address** edit box sets the contents of bits 1..7 of the TWSAM register. The value must be specified in hexadecimal.

The **SDA Hold Time Enabled** option specifies if the internal hold time on SDA with respect to the negative edge on SCL must be generated.

If the **TWI Smart Mode** option is enabled, the TWI slave enters Smart Mode, where the TWI Acknowledge Action is sent immediately after the TWI data register (TWSD) has been read.

When the **TWI Promiscous Mode** option is enabled, the address match logic of the TWI slave responds to all received addresses, ignoring the contents of the TWSA and TWSAM registers.

The **Two Wire Data Interrupt** option enables the generation of an interrupt when a data byte has been successfully received in the TWSD register, i.e. no bus errors or collisions have occurred during the operation.

The **Two Wire Address/Stop Interrupt** option enables the generation of an interrupt when the slave detects that a valid address has been received, a transmit collision or a STOP condition have been detected on the bus.

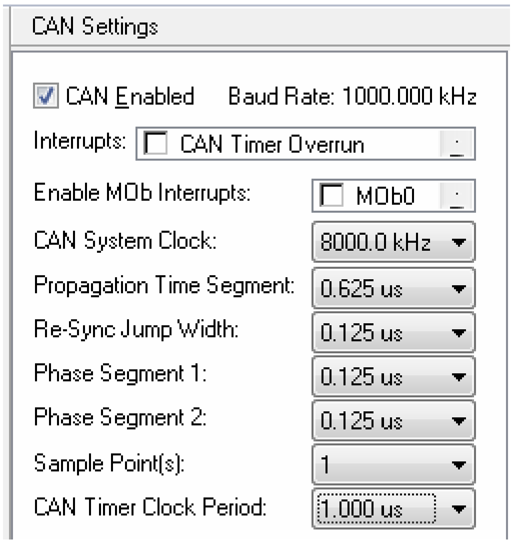
The **Two Wire Stop Interrupt** option enables the generation of an interrupt when a STOP condition has been detected on the bus.

The **TWI Acknowledge Action** list box specifies which action will be performed when a valid command has been written to TWCMD0 and TWCMD1 bits of the TWSCRB register, or when the TWSD data register has been read after a data byte has been received from the master.

More details about the TWI Slave Interface can be found in the ATtiny 20 datasheet.

## 6.17 Setting the CAN Controller

The **CAN Interface** configuration can be specified by clicking on the corresponding node of the CodeWizardAVR selection tree.




The AVR chip's CAN interface can be enabled by checking the **CAN Enabled** check box.
The **Interrupts** list box allows enabling/disabling the following interrupts generated by the CAN controller:

- **CAN Timer Overrun** interrupt, serviced by the **can_timer_isr** function
- **General Errors** (bit error, stuff error, CRC error, form error, acknowledge error) interrupt, serviced by the **can_isr** function
- **Frame Buffer Full** interrupt, serviced by the **can_isr** function
- **MOb Errors** interrupt, serviced by the **can_isr** function
- **Transmit** completed OK interrupt, serviced by the **can_isr** function
- **Receive** completed OK interrupt, serviced by the **can_isr** function
- **Bus Off** interrupt, serviced by the **can_isr** function
- **All** interrupts, except Timer Overrun, serviced by the **can_isr** function

The **Enable MOb Registers** list box allows for individual enabling/disabling of the CAN Message Object registers.
The **Enable MOb Interrupts** list box allows for enabling/disabling the interrupts generated by individual Message Object registers.
The **Highest Interrupt Priority MOb** list box allows selecting the Message Object register that has the highest interrupt priority.
The **CAN System Clock** list box allows selecting the frequency of the CAN controller system clock.

The **Propagation Time Segment** list box allows for compensation of physical delay times within the network. The duration of the propagation time segment must be twice the sum of the signal propagation time on the bus line, the input comparator delay and the output driver delay.

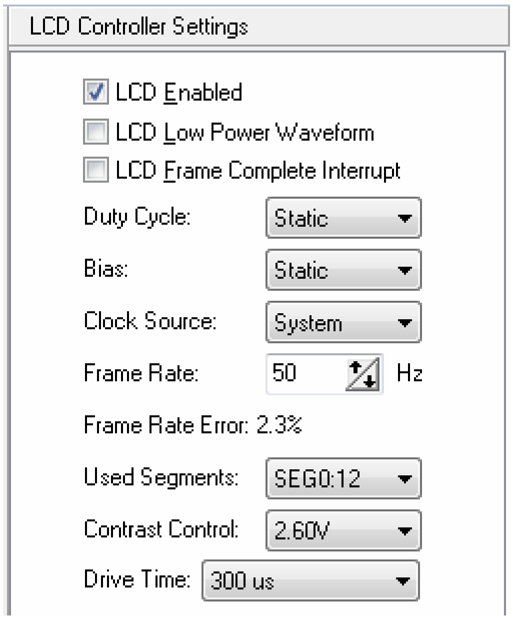
The **Re-Sync Jump Width** list box allows for compensation of phase shifts between clock oscillators of different bus controllers, by controller re-synchronization on any relevant signal edge of the current transmission.
The **Phase Segment 1** and **Phase Segment 2** list boxes allow for compensation of phase edge errors.

The **Sample Point(s)** list box allows selecting the number of times (1 or 3) the bus is sampled.
The **CAN Timer Clock Period** list box allows selecting the period of the CAN timer clock pulses.

The **CAN Baud Rate** is calculated based on the durations of the **CAN System Clock**, **Propagation Time Segment**, **Phase Segment 1** and **Phase Segment 2** parameters.
If the CAN Baud Rate value is correct its value is displayed in black color, otherwise it is displayed in red and must be corrected by modifying the above mentioned parameters.

## 6.18 Setting the ATmega169/329/3290/649/6490 LCD Controller

The configuration of the **LCD Controller** built in the ATmega169/329/3290/649/6490 chips can be specified by clicking on the corresponding node of the CodeWizardAVR selection tree.




The ATmega169V/L on chip LCD controller can be enabled by checking the **LCD Enabled** check box.
By checking the **LCD Low Power Waveform** check box, the low power waveform will be outputted on the LCD pins. This allows reducing the power consumption of the LCD.

If the **LCD Frame Complete Interrupt** check box is checked, the LCD controller will generate an interrupt at the beginning of a new frame. In low power waveform mode this interrupt will be generated every second frame. The frame complete interrupt will be serviced by the **lcd_sof_isr** function.

The **LCD Duty Cycle** list box selects one of the following duty cycles: Static, 1/2, 1/3 or 1/4.

The **LCD Bias** list box selects the 1/3 or 1/2 bias. Please refer to the documentation of the LCD manufacturer for bias selection.

The **Clock Source** list box selects the system clock or an external asynchronous clock as the LCD controller clock source.

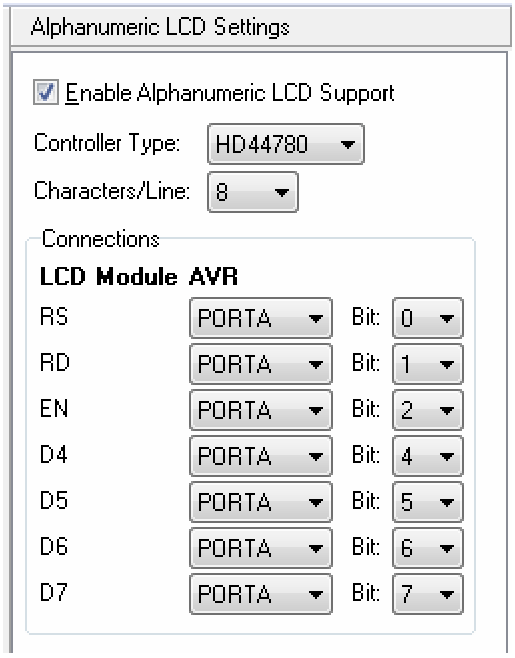
The **Frame Rate** spin edit allows specifying the LCD frame rate.
The LCD Frame Rate Register (LCDFRR) is initialized based on the frequency of the clock source and the obtainable frame rate, that is as close as possible to the one that was specified.

The **Frame Rate Error** is calculated based on the specified **Frame Rate** and the real one obtained from LCDFRR.

The **Used Segments** list box setting determine the number of port pins used as LCD segment drivers.
The **Contrast Control** list box specifies the maximum voltage on LCD segment and common pins **VLCD**. The **VLCD** range is between 2.60 and 3.35 Vcc.

## 6.19 Setting the Alphanumeric LCD

The I/O port allocation for the **LCD Functions for displays with up to 2x40 characters** can be configured by clicking on the **Alphanumeric LCD** node of the CodeWizardAVR selection tree.



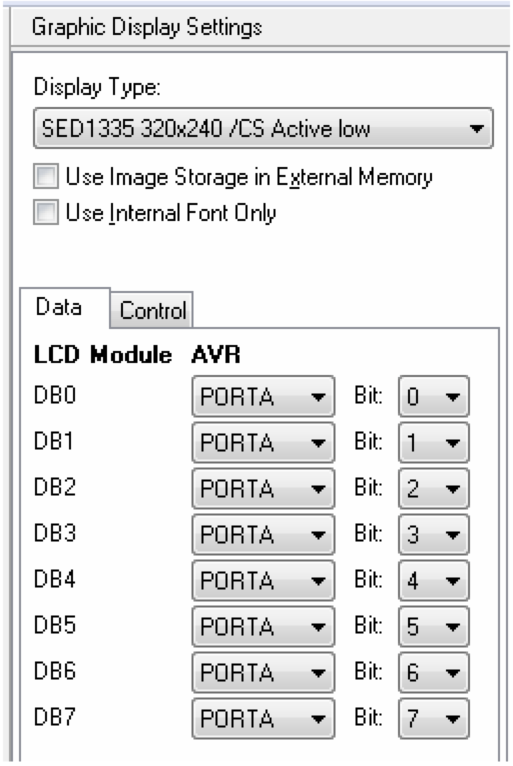

The **Enable Alhanumeric LCD Support** check box activates the configuration specified for the *alcd.h* library functions.

The **Characters/Line** list box allows to specify the number of characters per line supported by the LCD module.

The connections between the LCD module and the AVR I/O ports can be specified individually for each signal in the **Connections** group box.

## 6.20 Setting the Graphic LCD

The I/O port allocation for the **Graphic LCD Functions** can be configured by clicking on the **Graphic LCD** node of the CodeWizardAVR selection tree.

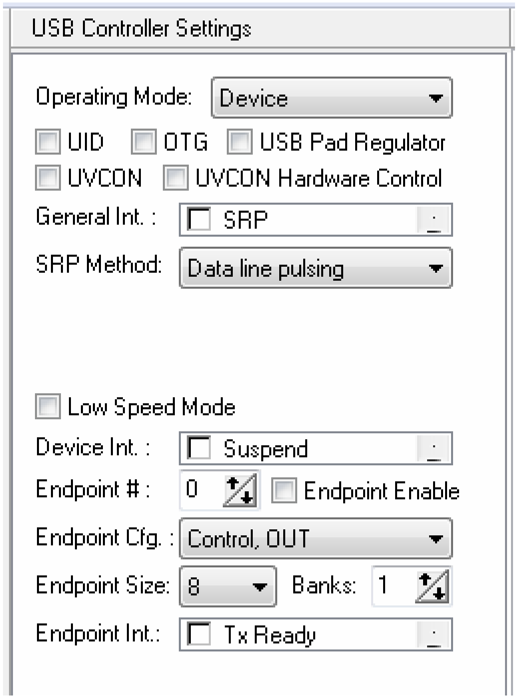



The **Display Type** list box allows to select the graphic controller type and LCD resolution.
The **Use Image Storage in External Memory** check box specifies if additional code will be generated for functions needed to read or write data from external memory, used for graphic image storage.
The **Use Internal Font Only** check box specifies for the **glcd_init** function, that only the internal character generator of the controller is used for displaying text.
**Note:** This option is available only for graphic LCD controllers that have a built-in character generator.

The connections between the graphic LCD module and the AVR I/O ports can be specified individually for each signal in the **Data** and **Control** tabs.
**Note:** In order to obtain maximum performance, it is advised to set the LCD controller's **data bus** bits to match the bits with the same numbers of the same AVR I/O port.

## 6.21 Setting the USB Controller

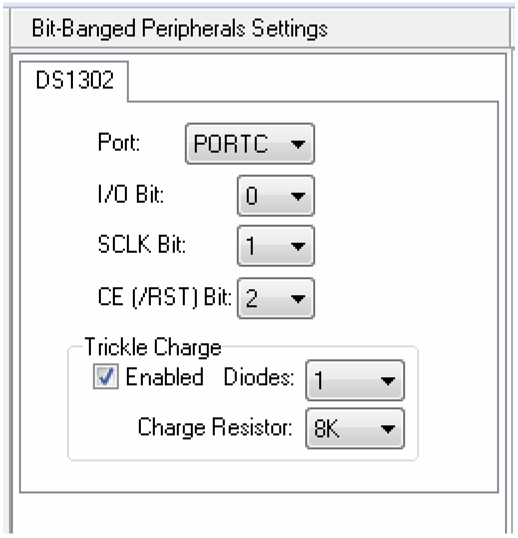
The configuration of the **USB Controller** for the AT90USB646, AT90USB647, AT90USB1286 and AT90USB1287 chips can be specified by clicking on the corresponding node of the CodeWizardAVR selection tree.
The USB controller can operate in two modes: **Device** and **Host**, specified using the **Operating Mode** list box.




The operation of the USB controller in both modes and the various settings for them are described in detail in the AT90USB datasheet.

## 6.22 Setting Bit-Banged Peripherals

The configuration of the peripherals connected using the bit-banging method can be specified by clicking on the **Bit-Banged Peripherals** node of the CodeWizardAVR selection tree.
If you use the DS1302 RTC, you must select the **DS1302** tab.



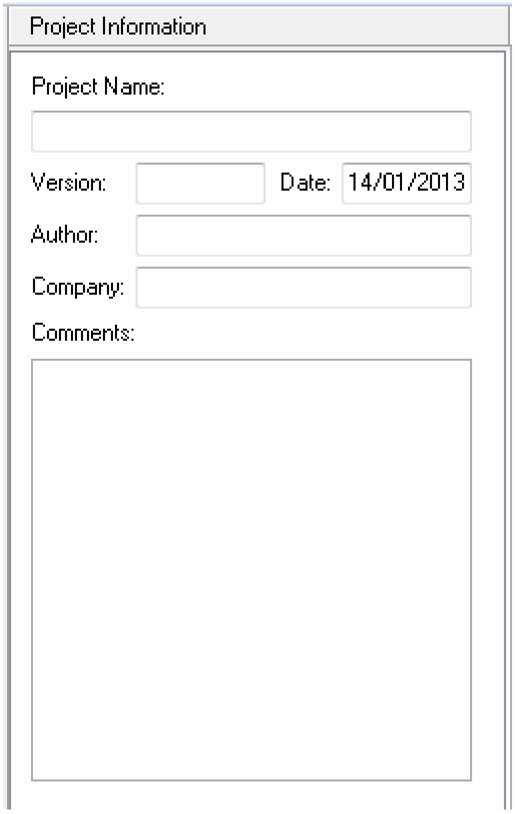

Using the **Port** list box you can specify which port is used for connecting with the DS1302.
The **I/O Bit**, **SCLK Bit** and **/RST Bit** list boxes allow you to specify which port bits are used for this.

The DS1302's trickle charge function can be activated by checking the **Trickle Charge|Enabled** check box.
The number of diodes, respectively the charge resistor value, can be specified using the **Trickle Charge|Diodes**, respectively **Trickle Charge|Resistors**, list boxes.

The DS1302 device is accessed through the **Maxim/Dallas Semiconductor DS1302 Real Time Clock Functions**.

## 6.23 Specifying the Project Information

The information placed in the comment header, located at the beginning of the C source file produced by CodeWizardAVR, can be specified by clicking on the **Project Information** node of the selection tree.



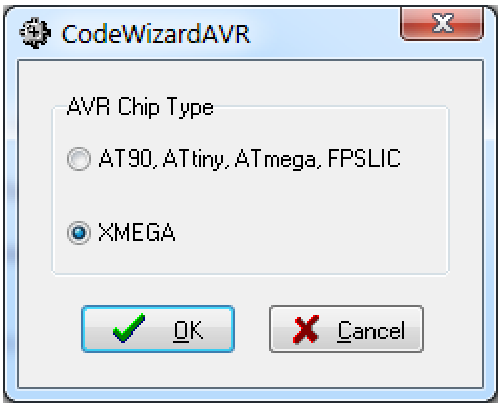

You can specify the **Project Name**, **Date**, **Author**, **Company** and **Comments**.

# 7. CodeWizardAVR Automatic Program Generator for the XMEGA Chips

The CodeWizardAVR Automatic Program Generator allows you to easily write all the code needed for initializing the XMEGA on-chip peripherals.

The Automatic Program Generator is invoked using the **Tools|CodeWizardAVR** menu command or by clicking on the toolbar button.

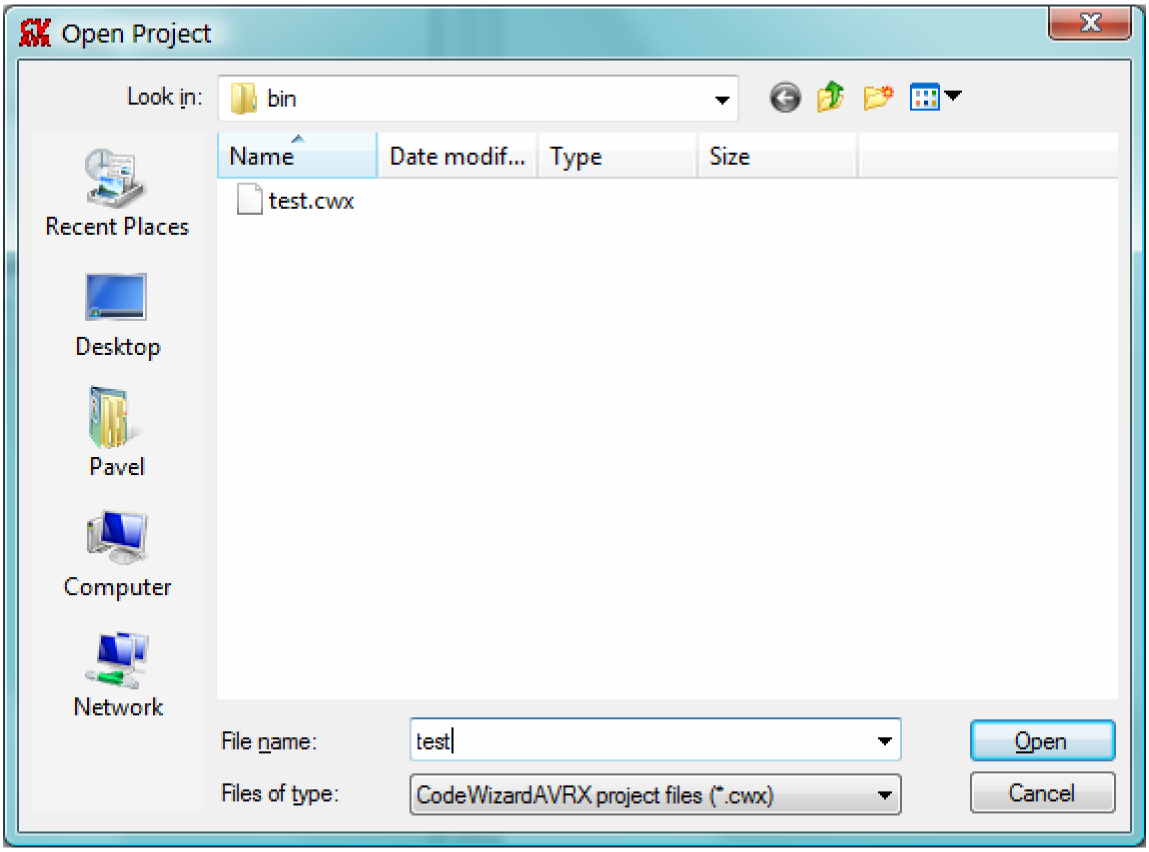
The following dialog box will open



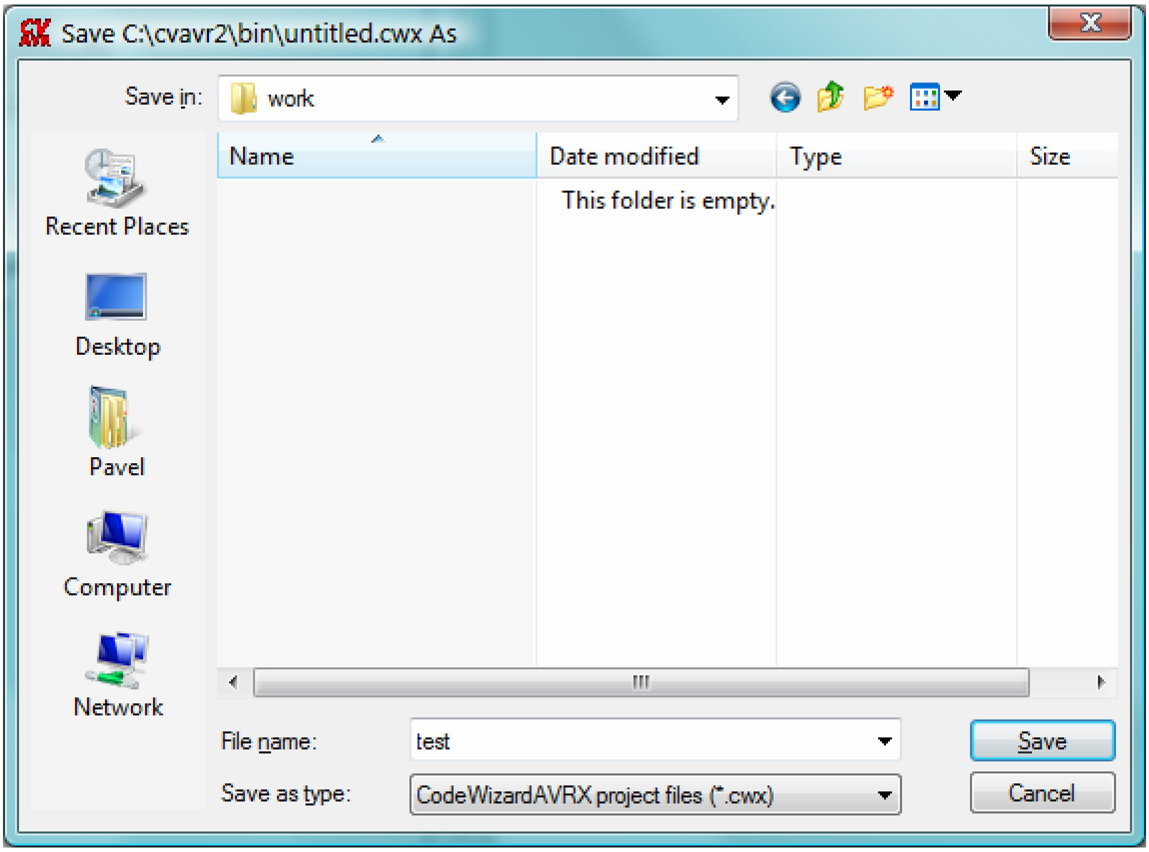

allowing to select between the AVR chip families for which automatic code generation will be performed.

The **File|New** menu command or the toolbar button allow creating a new CodeWizardAVR project. This project will be named by default **untitled.cwx** .

The **File|Open** menu command or the toolbar button allow loading an existing CodeWizardAVR project:



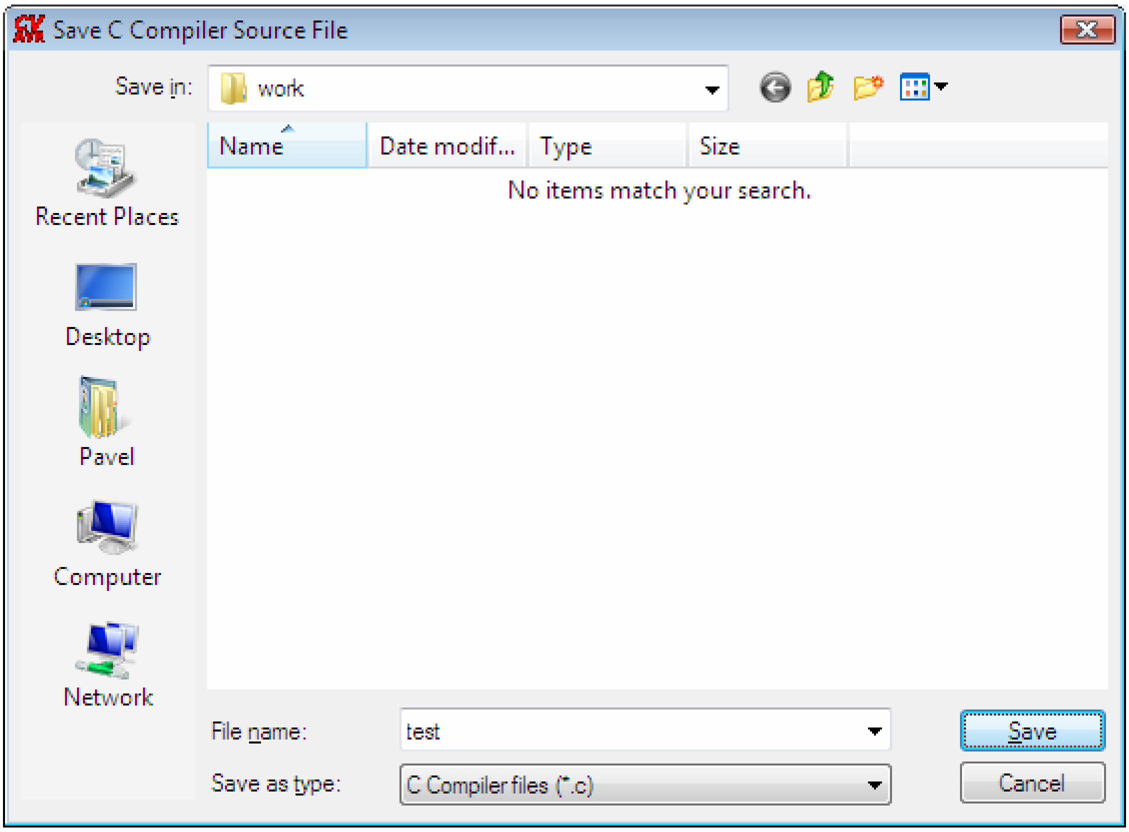
The **File|Save** menu command or the toolbar button allow saving the currently opened CodeWizardAVR project.
The **File|Save As** menu command or the toolbar button allow saving the currently opened CodeWizardAVR project under a new name:



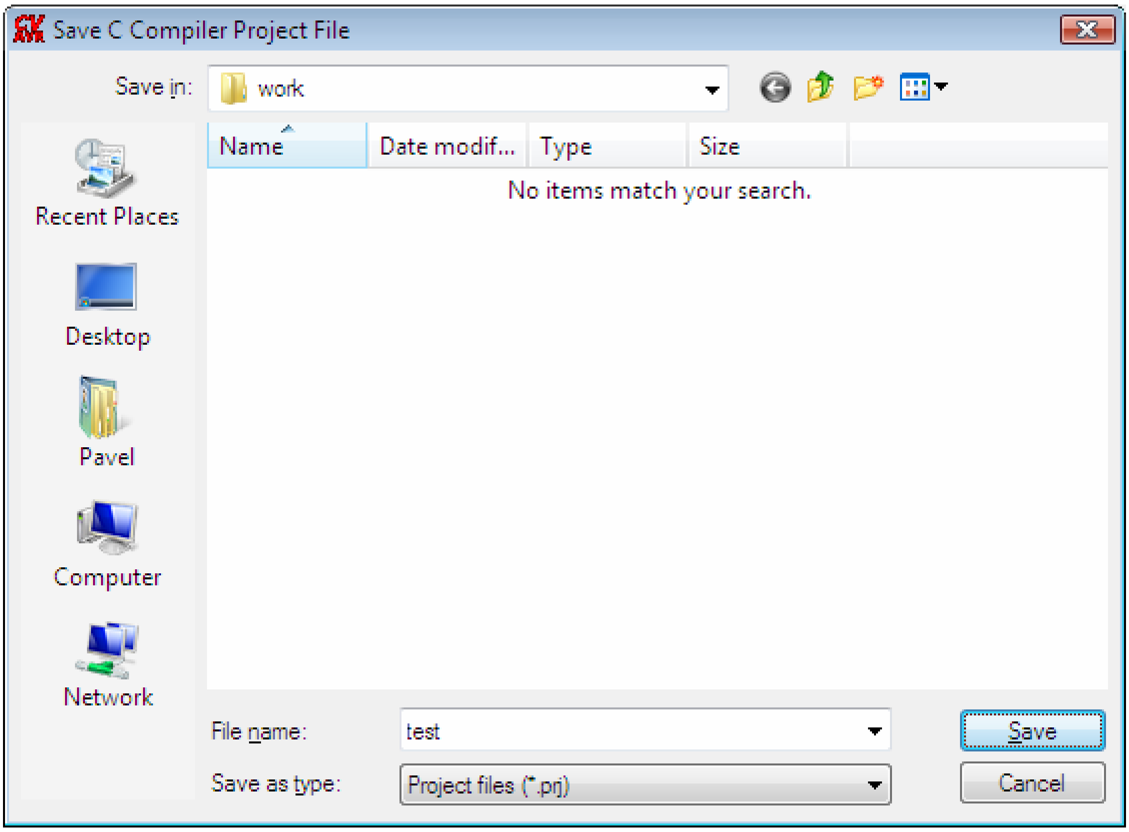

By selecting the **Program|Generate** menu option or by clicking on the toolbar button, the code generated by CodeWizardAVR can be viewed in the **Program Preview** window. This may be useful when applying changes to an existing project, as portions of code generated by the CodeWizardAVR can be selected, copied to the clipboard and then pasted in the project's source files.

If the **Program|Generate, Save and Exit** menu option is selected or the toolbar button is clicked, CodeWizardAVR will generate the main .C source and project .PRJ files, save the CodeWizardAVR project .CWX file and return to the CodeVisionAVR IDE.
Eventual peripheral configuration conflicts will be prompted to the user, allowing him to correct the errors.

In the course of program generation the user will be prompted for the name of the main C file:




and for the name of the project file:

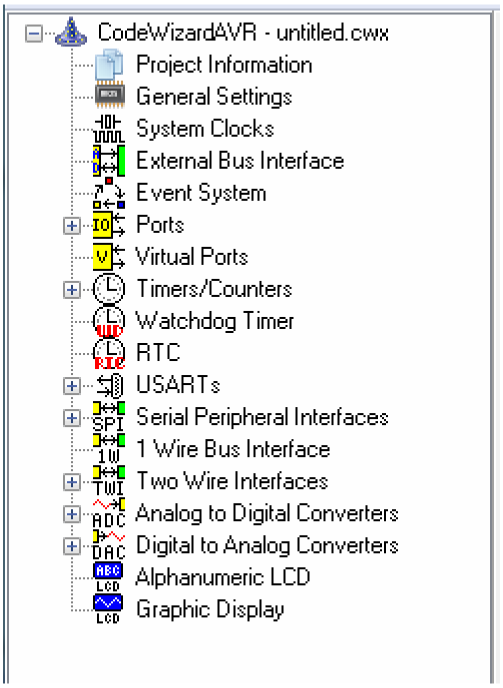


**Note:** When a **.prj** project for the CodeVisionAVR IDE is created, a corresponding **.cproj** project file for Atmel Studio will be created too.
This allows editing/compiling the same project in both Atmel Studio and CodeVisionAVR IDE.

Selecting the **File|Exit** menu option allows the user to exit the CodeWizardAVR without generating any program files.

By selecting the **Help|Help Topics** menu option, by pressing the F1 key or by clicking on the  toolbar button, the user can see the help topic that corresponds to the current CodeWizardAVR configuration menu.

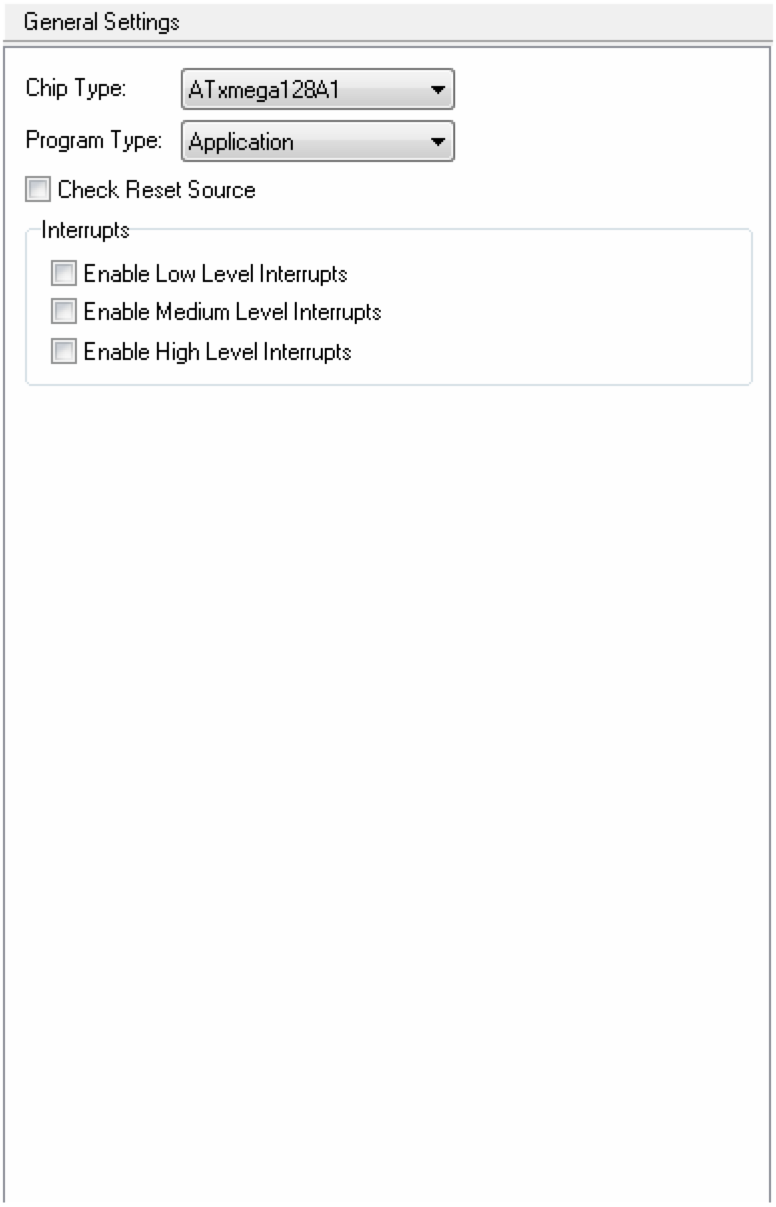
The XMEGA peripheral that needs to be configured can be selected by clicking on the corresponding node of the CodeWizardAVR selection tree.




If program code was already generated and is available for display in the Program Preview window, clicking on a peripheral node, will position the cursor at the beginning of the initialization code for that peripheral.

## 7.1 Setting the General Chip Options

The general chip options can be specified by clicking on the General Settings node of the CodeWizardAVR selection tree.




The **Chip Type** list box allows to select the XMEGA device for which code will be generated.

The **Program Type** list box allows to select the type of the generated code:

- Application
- Boot Loader

The **Check Reset Source** check box enables the generation of code that allows the identification of the conditions that caused the XMEGA chip reset:

- Power-On Reset
- External Reset
- Brown Out Reset
- Watchdog Reset
- Program and Debug Interface Reset
- Software Reset.

The **Interrupts** group box allows to specify the settings for **Programmable Multi-level Interrupt Controller** initialization code generation.
The following groups of interrupts can be individually enabled or disabled:

- Low Level Interrupts
- Medium Level Interrupts
- High Level Interrupts.

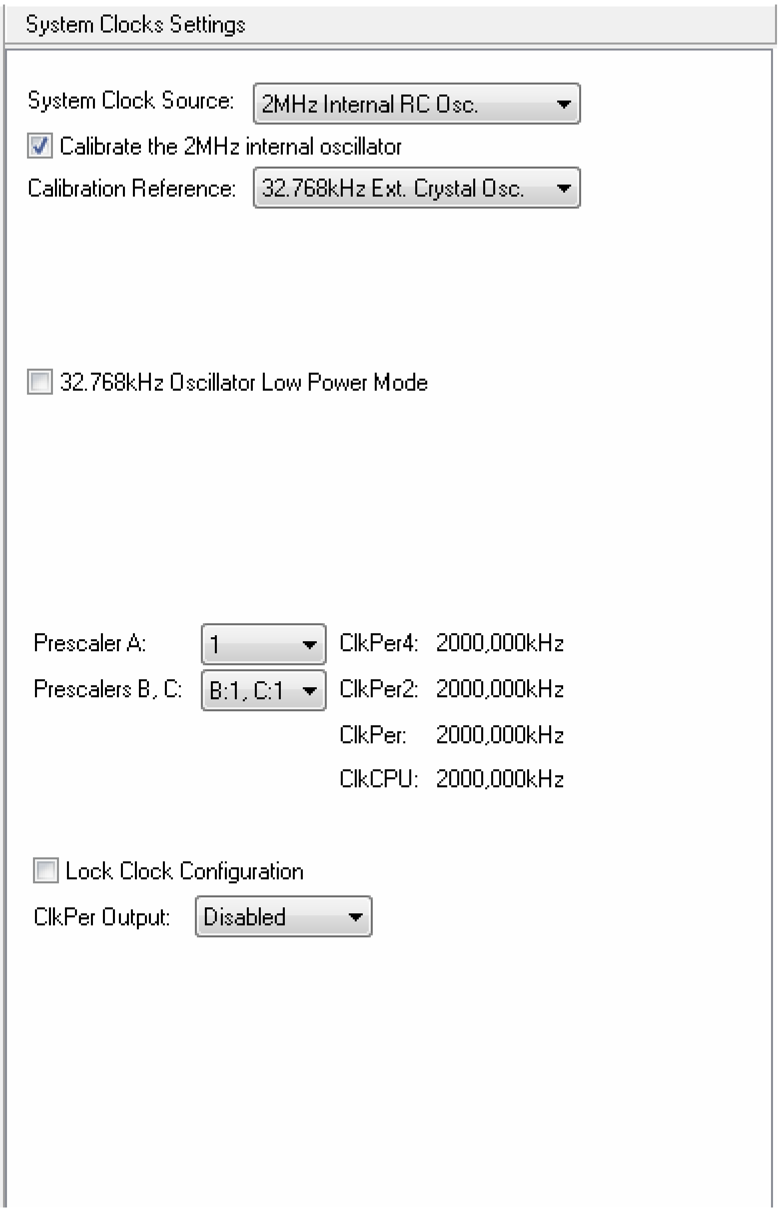
## 7.2 Setting the System Clocks

The various XMEGA clock source options can be specified by clicking on the **System Clocks** node of the CodeWizardAVR selection tree.

The **System Clock Source** list box allows to select between the following options:

- 2MHz Internal RC Oscillator
- 32MHz Internal RC Oscillator
- 32.768kHz Internal RC Oscillator
- External Oscillator or Clock
- Phase Locked Loop.

If one of the internal RC oscillators is used as a system clock source, the following options are available:



If the **Calibrate Internal Oscillator** option is enabled, the internal 2MHz or 32MHz RC oscillator, used as system clock source, will be calibrated using one of the **Calibration Reference** sources:
- 32.768kHz Internal RC Oscillator
- 32.768kHz External Crystal Oscillator.

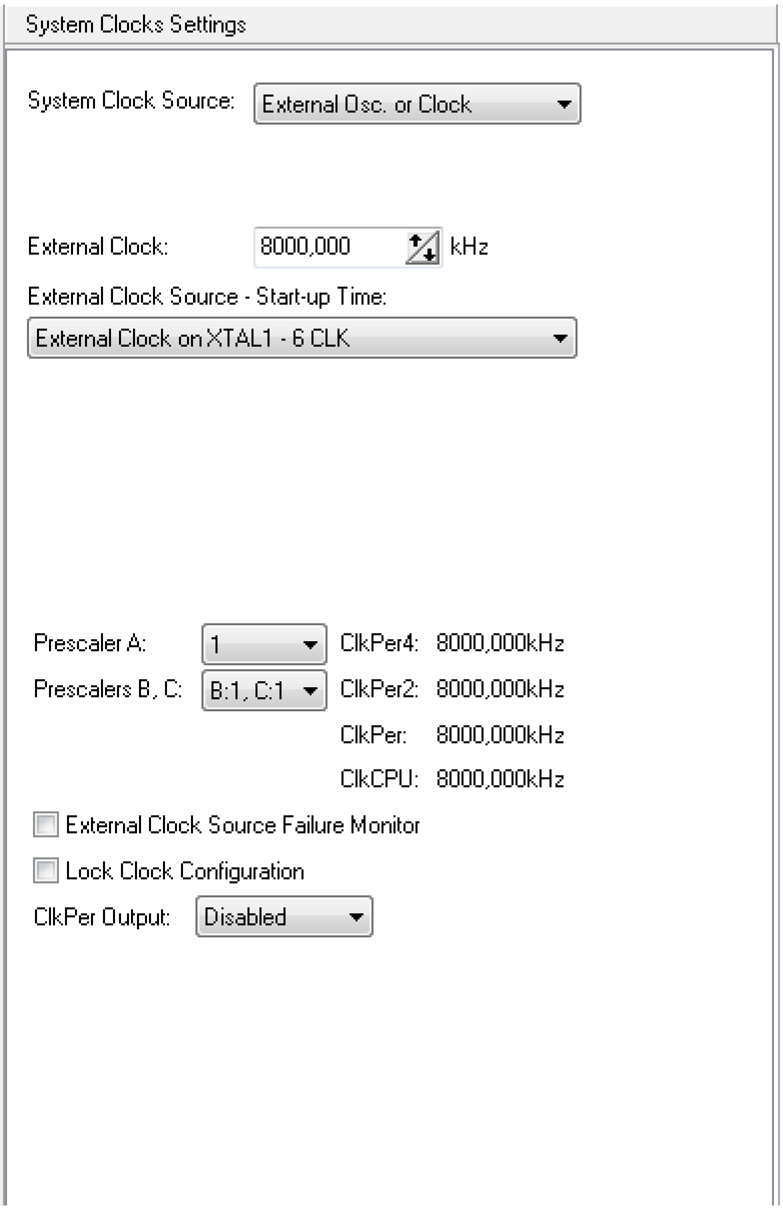
The **32.768kHz Oscillator Low Power Mode** option allows to run this external crystal oscillator with reduced voltage swing on the **TOSC2** pin.

The **Prescaler A** option allows to divide the system clock by a factor between 1 and 512, obtaining the **ClkPer4** peripheral clock.
The **Prescaler B, C** option allows to divide the **ClkPer4** peripheral clock by a factor of 1, 2 or 4, obtaining the **ClkPer2**, **ClkPer** peripheral clocks and the **ClkCPU** clock used by the CPU and Non-Volatile Memory.

If the **Lock Clock Configuration** option is enabled, the system clock selection and prescaler settings are protected against further updates until the next chip reset.

The **ClkPer Output** list box allows to specify if the **ClkPer** signal will be fed to the bit 7 of **PORTC**, **PORTD** or **PORTE**.

If an **External Oscillator or Clock** is used as **System Clock Source**, the following specific configuration options are available:



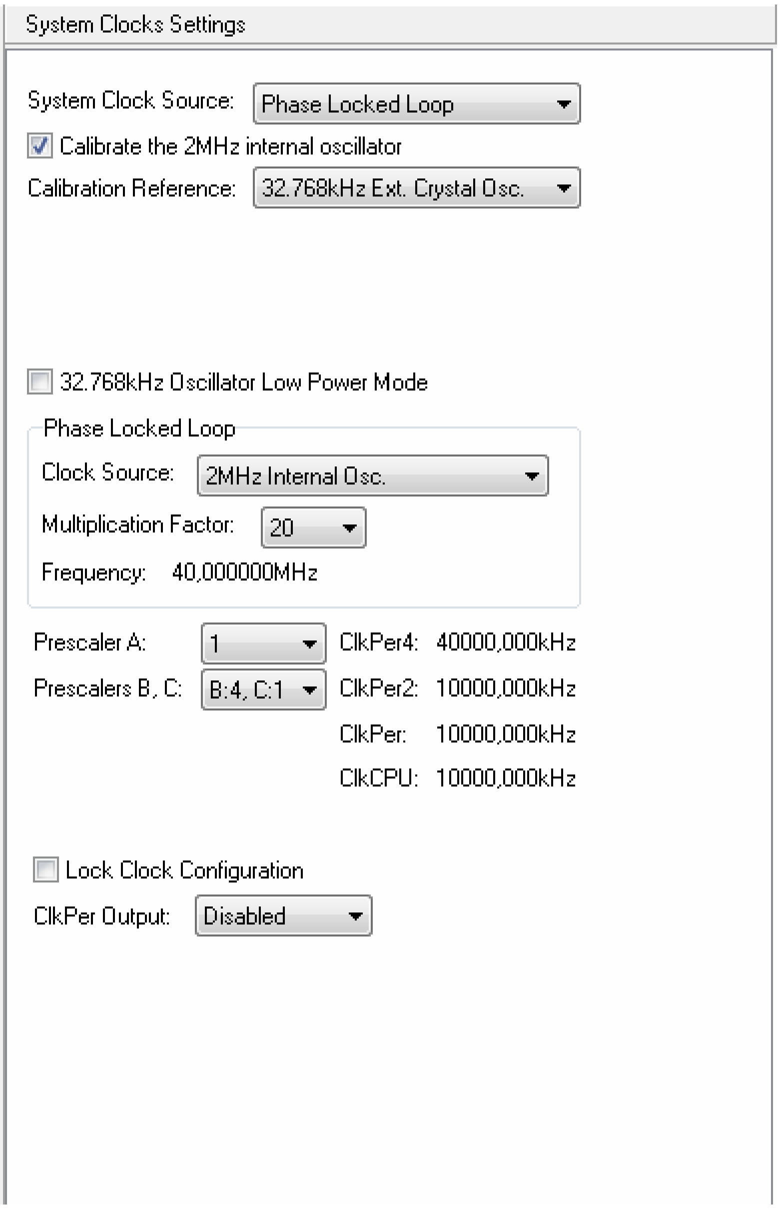

The **External Clock** option specifies the value of the external clock frequency in kHz.

The **External Clock Source - Start-up Time** list box allows to select the type of external clock source: external clock signal, crystal or ceramic resonator, and its start-up time.

If the **External Clock Source Failure Monitor** option is enabled, the device will perform the following actions if the external clock stops:

- switch to the 2MHz internal oscillator, independently of any clock system lock setting, by reseting the **System Clock Selection Register** to its default value
- reset the **Oscillator Control Register Register** to its default value
- Set the **External Clock Source Failure Detection Interrupt Flag** in the **XOSC Failure Detection Register**
- Issue a **Non-Maskable Interrupt** (NMI).

If a **Phase Locked Loop** (PLL) is used as **System Clock Source**, the following specific configuration options are available:



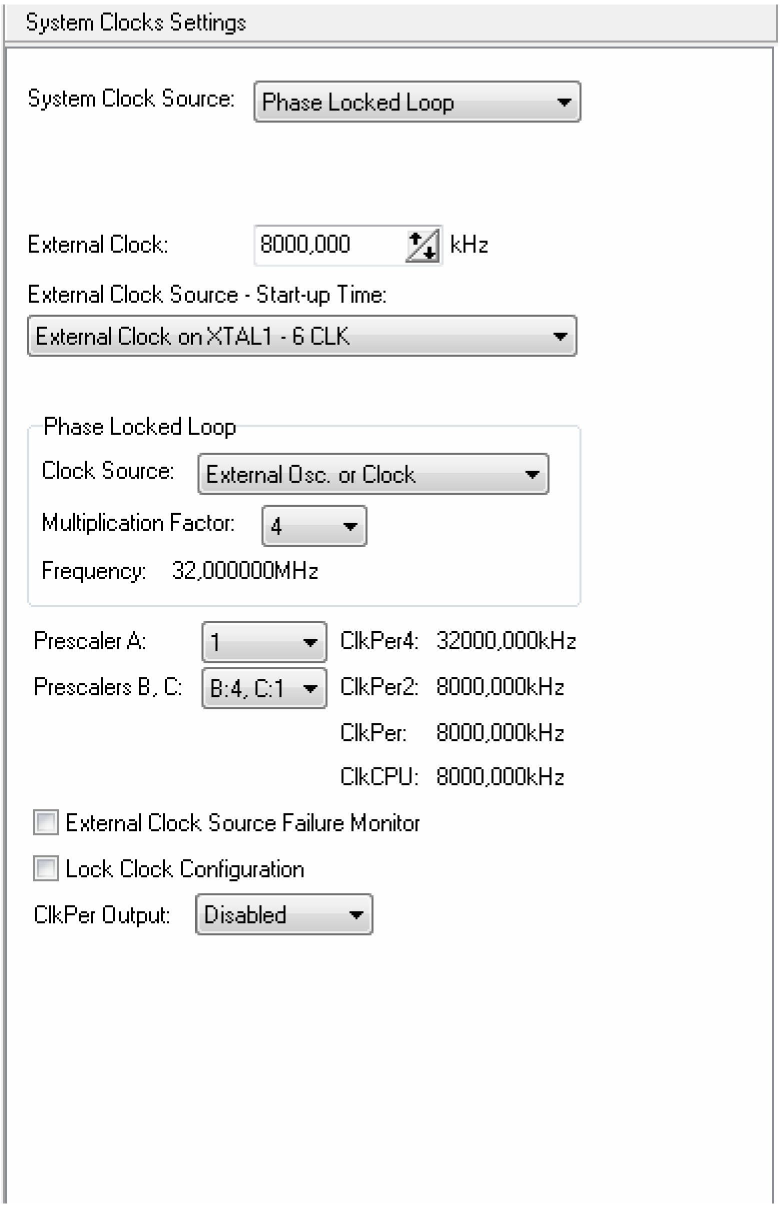

The **Clock Source** list box allows to select one of the following clocks for the **PLL**:

- 2MHz Internal RC Oscillator
- 32MHz Internal RC Oscillator divided by 4
- External Oscillator or Clock.

For the the two internal RC oscillators, we can find the specific calibration options, that were explained previously.

The **Multiplication Factor** list box allows to select a factor between 1 and 31, by which the **PLL** will multiply its clock source frequency.

If an **External Oscillator or Clock** is selected as **PLL** clock source, we can find the specific options, that were explained previously:



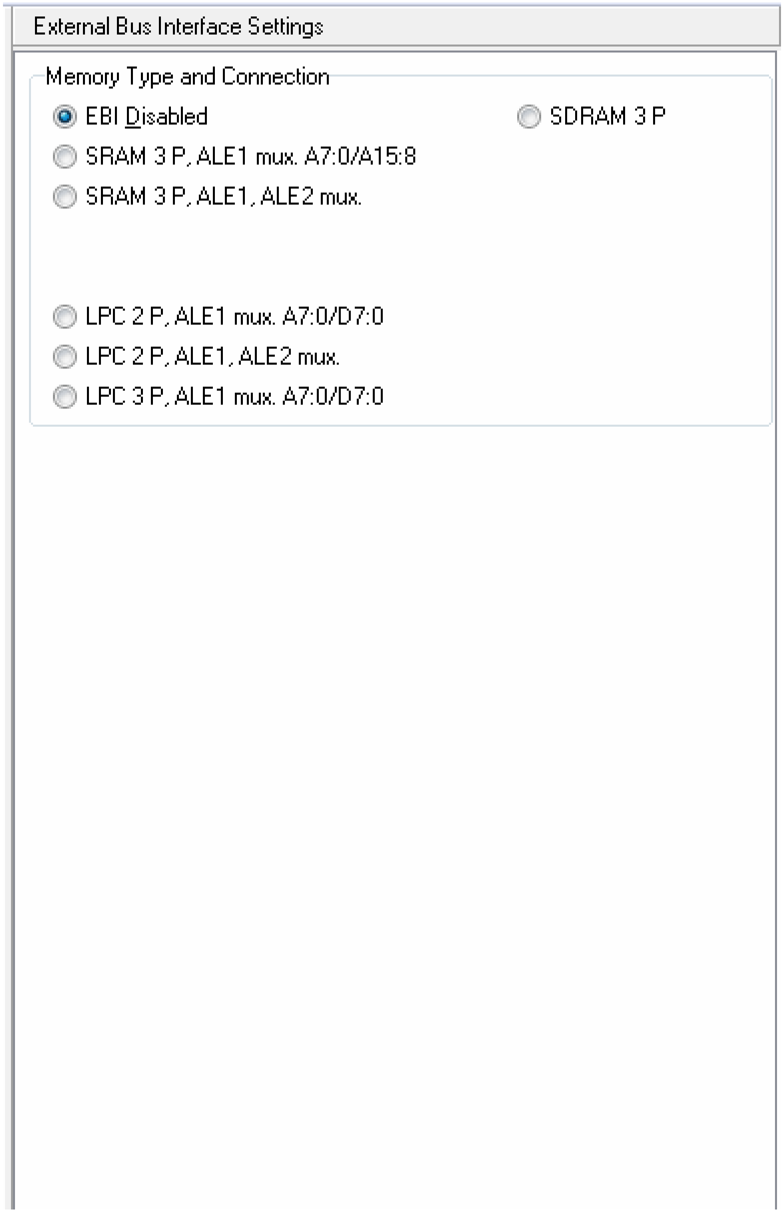

The **System Clocks** initialization is performed by the:

**void system_clocks_init(void)**

function generated by the CodeWizardAVR.

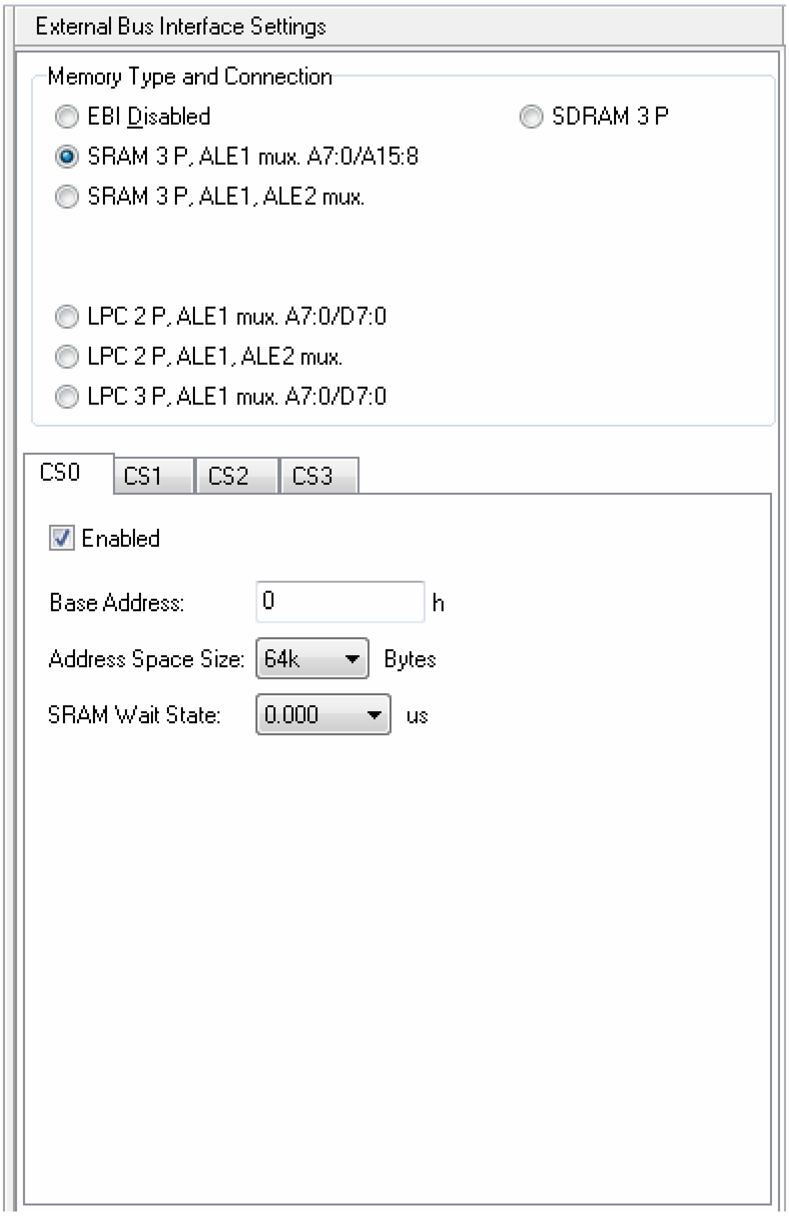
## 7.3 Setting the External Bus Interface

The **External Bus Interface** (**EBI**) program generation options can be specified by clicking on the **External Bus Interface** node of the CodeWizardAVR selection tree.



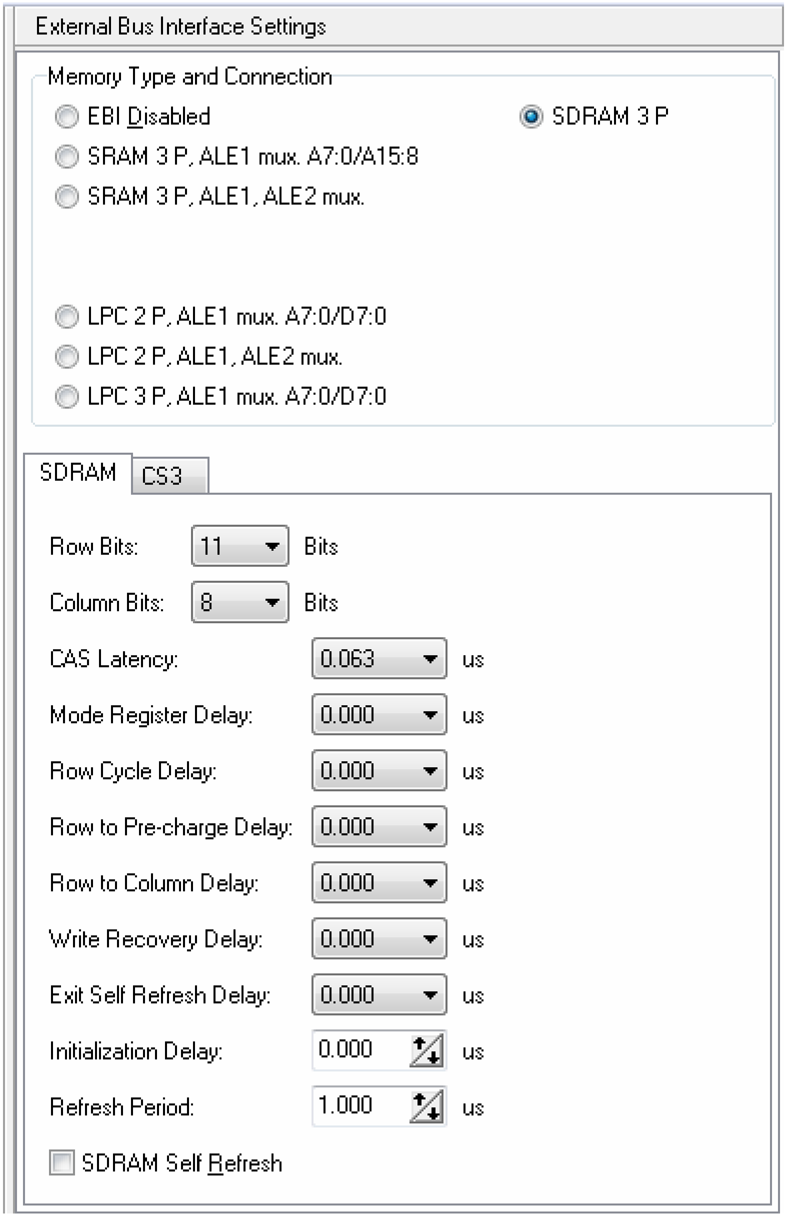

The check boxes from the **Memory Type and Connection** group allow to specify the **EBI** operating mode and kind of external RAM connected to the chip.

Depending on the **EBI** operating mode, additional tabs are displayed for the configuration of the **CS0...CS3** chip select signals:




The **Enable** check box activates the usage of the corresponding **CSn** chip select signal.
The **Base Address** represents the starting address, in hexadecimal, of the **Address Space** for which the chip select signal becomes active.
The **Address Space Size** list box allows to specify the address range size for which the chip select signal is active.
The **SRAM Wait State** list box allows inserting additional wait states when accessing slow external memory.

Specific options can be set if **SDRAM** chips are connected to the XMEGA chip:




These options are described in detail in Atmel's **XMEGA A Manual** in the **EBI - External Bus Interface** chapter.

**Note:** All the necessary code for **EBI** setup will be automatically added by the compiler in the startup initialization that is executed immediately after the chip reset. There is no need for the programmer to write his own code for this purpose.

When **SDRAM** is used as external memory and a different clock source is used instead of the internal 2MHz oscillator, it is necessary to execute the function that configures the system clocks **before** the **EBI** setup sequence, which will ensure that correct timing is used for later **SDRAM** access by the startup code.
This can be achieved by using the **__reset** attribute applied to the clock initialization function:

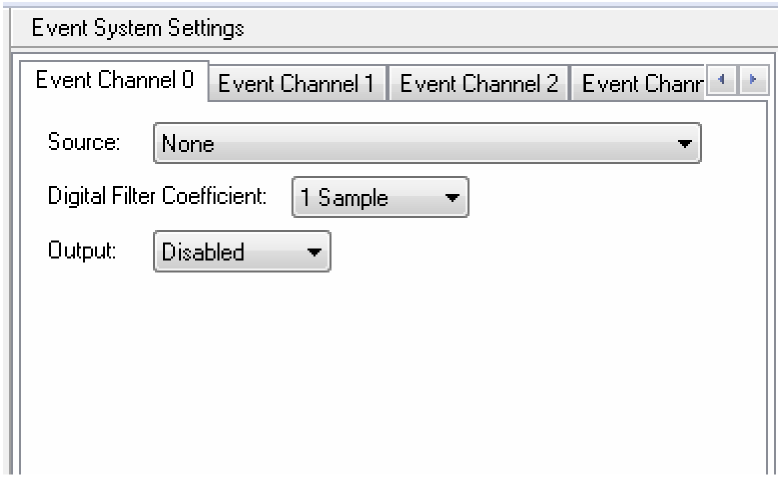
```
__reset void system_clocks_init(void)
{
// Initialization code ...

}
```

The code generated by the CodeWizardAVR for XMEGA chips, automatically handles such situations.

## 7.4 Setting the Event System

The XMEGA Event System can be configured by clicking on the **Event System** node of the CodeWizardAVR selection tree.

The following options are available:




The **Event Channel Source** list box allow to select the events that will trigger the corresponding channel.

The **Digital Filter Coefficient** option allows to specify the length of digital filtering used.
Events will be passed through to the event channel, only when the event source has been active and sampled with the same level, for the specified number of peripheral clock cycles.
The **Event Channel 0 Output** list box allows to specify if the signal triggered by **Event Channel 0** will be fed to the bit 7 of **PORTC**, **PORTD** or **PORTE**.

Additional **Event System** specific options are present in the CodeWizardAVR configuration pages for each XMEGA peripheral.

The **Event System** initialization is performed by the:

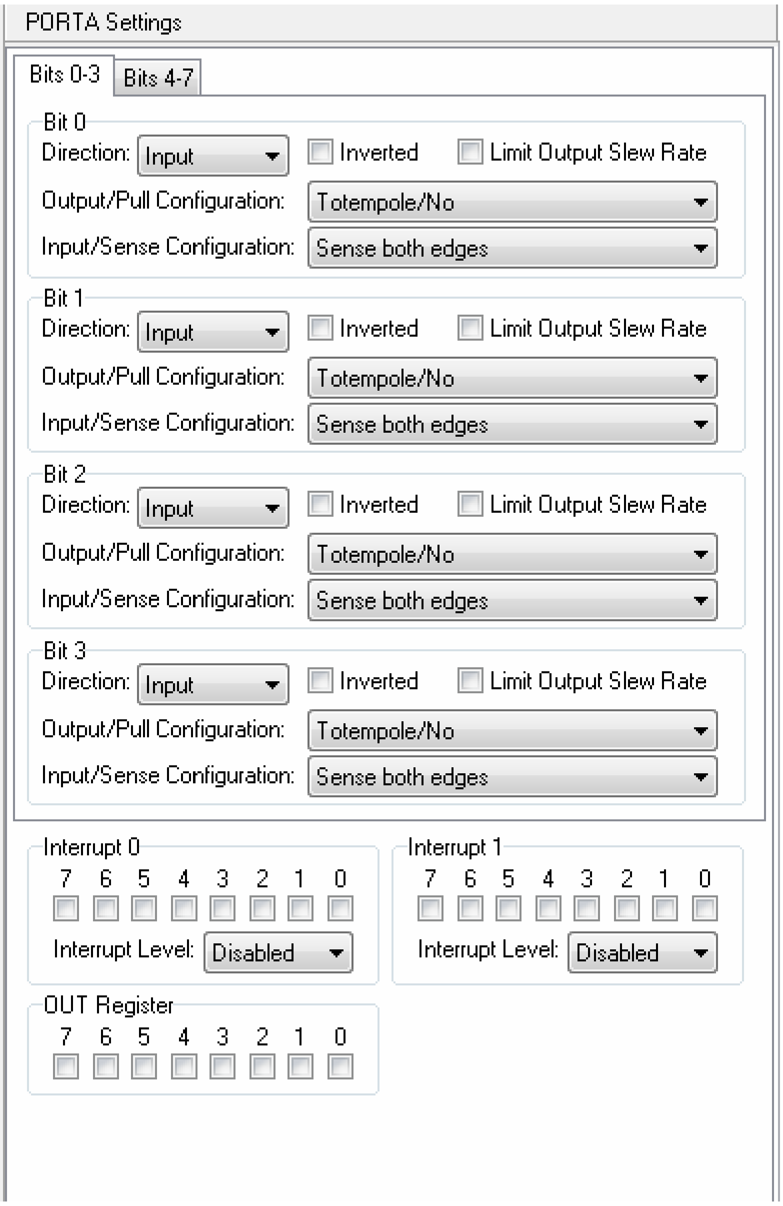
**void event_system_init(void)**

function generated by the CodeWizardAVR.

## 7.5 Setting the Input/Output Ports

The XMEGA Input/Output Ports can be configured by clicking on the **Ports** and **PORTn** nodes of the CodeWizardAVR selection tree.

The following options are available for configuring each bit of an I/O port:




The **Direction** list box specifies if the pin associated with the I/O port bit will be an input or output.
The input/output data on the port pin can be **Inverted** by enabling this option.
The **Limit Output Slew Rate** option will enable slew rate limiting on the corresponding output pin.

The **Output/Pull Configuration** list box allows to specify the corresponding configurations for the port pin.
**Output** configurations can be:

- Totempole
- Wired OR
- Wired AND.

**Pull** configurations can be:

- None
- Bus keeper
- Pull down, if the pin is an input
- Pull up, if the pin is an input.

The **Input/Sense Configuration** list box allows to specify how the pin, configured as input, can trigger port interrupts and events.

The **Interrupt 0**, respectively **Interrupt 1,** group boxes allow to individually enable/disable port interrupt 0, respectively port interrupt 1, triggering by each pin.
For both **Interrupt 0** and **Interrupt 1**, the enabled state and priority can be specified by the corresponding **Interrupt Level** list boxes.

The **OUT** group box allows to individually set the values of each bit of the port output register, written during initialization.

The **Input/Output Ports** initialization is performed by the:

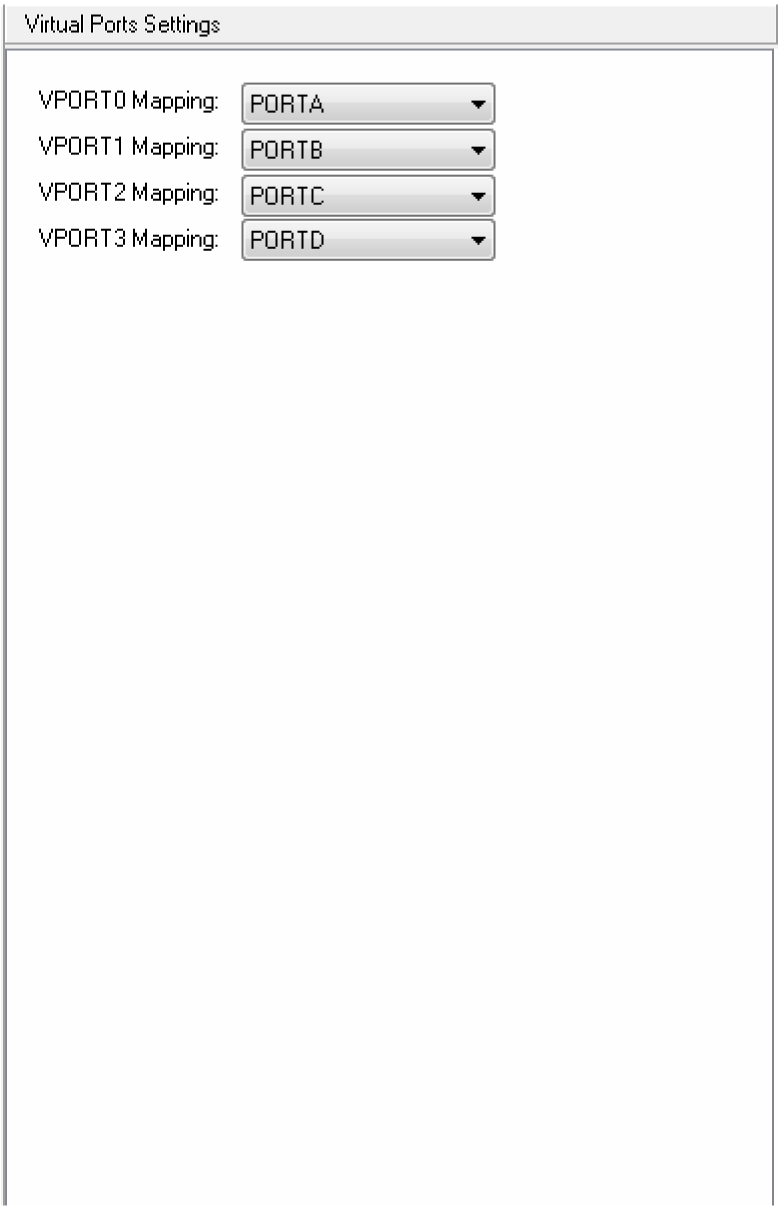
**void ports_init(void)**

function generated by the CodeWizardAVR.

## 7.6 Setting the Virtual Ports

The XMEGA Virtual Ports can be configured by clicking on the **Virtual Ports**  node of the CodeWizardAVR selection tree.

The following options are available:




The **VPORT0 Mapping**, **VPORT1 Mapping**, **VPORT2 Mapping** and **VPORT3 Mapping** list boxes allow to select which I/O port, mapped in the extended I/O memory space, will be mapped virtually to the I/O memory space, allowing it to be accessed using more efficient IN and OUT instructions.

The **Virtual Ports** initialization is performed by the:

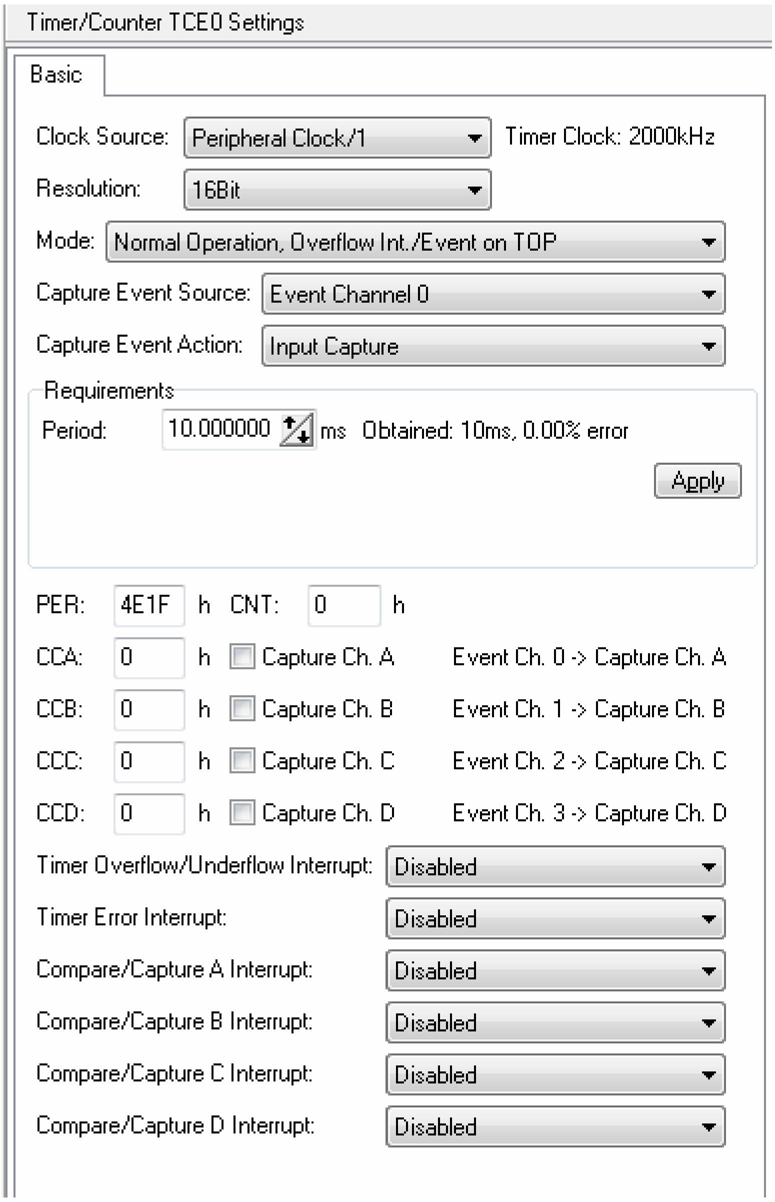
**void vports_init(void)**

function generated by the CodeWizardAVR.

## 7.7 Setting the Timers/Counters

The XMEGA Timers/Counters can be configured by clicking on the **Timers/Counters** and **TCn** nodes of the CodeWizardAVR selection tree.

The Timer/Counter can be activated by selecting a **Clock Source**:




The clock source can be the **Peripheral Clock** divided by a factor between 1 and 1024 or an event from **Event Channels** 0 to 7.

The **Resolution** list box allows to select one of the following options:

- 16Bit
- 16Bit, with High Resolution Extension enabled
- 8Bit.

The **Mode** list box allows to select one of the following Timer/Counter operating modes:

- Normal Operation
- Frequency Waveform Generation
- Single Slope Pulse Width Modulation (PWM) Generation
- Dual Slope PWM Generation.

In **Normal** operating mode, the timer can capture an event specified by the **Capture Event Action** option.
The **Capture Event Source** option specifies the capture source for **Capture Channel A (CCA)**.
The event sources for the rest of the capture channels: **CCB**, **CCC** and **CCD** are the next **Event Channels** in ascending order, as can be seen from the above example picture.

Each **Capture Channel** can be enabled using the corresponding **Capture Ch. A**, **Capture Ch. B**, **Capture Ch. C** or **Capture Ch. D** check boxes.

The initial values of the **CCA**, **CCB**, **CCC** and **CCD** capture channel registers can be specified using the corresponding edit controls.

The **Requirements** group box allows the user to specify the desired timer **Period** in ms.
Pressing the **Apply** button will perform automatic timer configuration (**Clock Source** and **PER** period register values), so that the required timer period will be obtained for a given **Peripheral Clock** value.

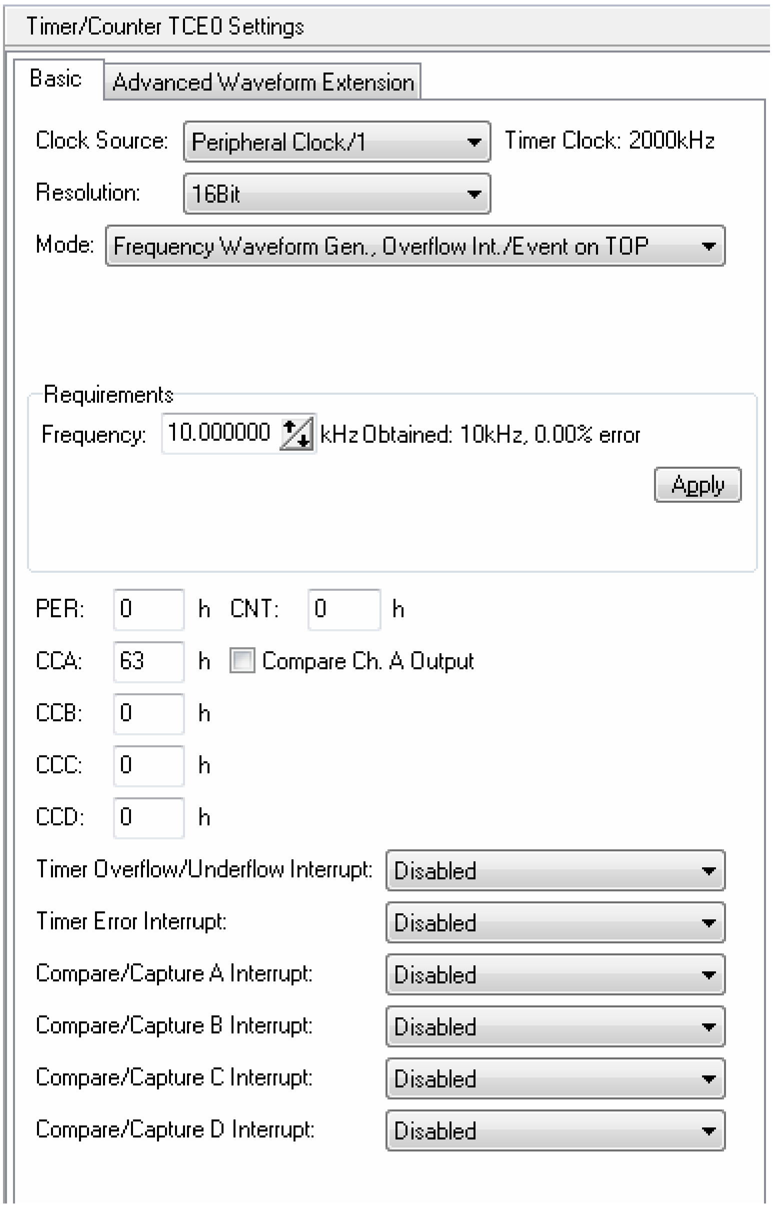
The initial value for the Timer/Counter CNT register can be specified using the corresponding edit control.

The Timer/Counter can generate several types of interrupts:

- Timer Overflow/Underflow Interrupt
- Timer Error Interrupt
- Compare/Capture A Interrupt
- Compare/Capture B Interrupt
- Compare/Capture C Interrupt
- Compare/Capture D Interrupt.

Each type of interrupt can be individually enabled and its priority set, using the corresponding list boxes.

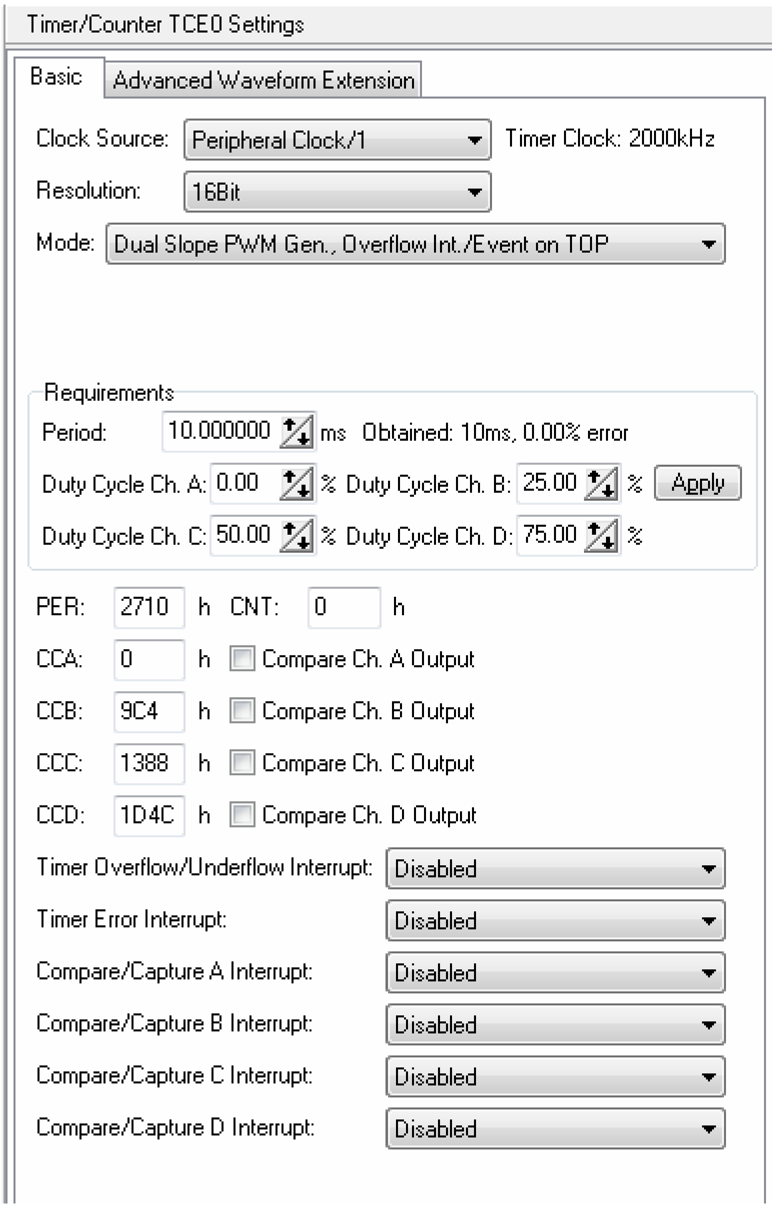
In **Frequency Waveform Generation** mode the following specific options are available:




The **Requirements** group box allows the user to specify the desired timer **Frequency** in kHz.
Pressing the **Apply** button will perform automatic timer configuration (**Clock Source** and **CCA** register values), so that the required timer frequency will be obtained for a given **Peripheral Clock** value.
**Note:** The **PER** register is not used in this operating mode.

If the **Compare Ch. A Output** option is enabled, the corresponding waveform generation (WG) output will be toggled on each compare match between the **CNT** and **CCA** registers.
The duty cycle of this signal will be 50%.

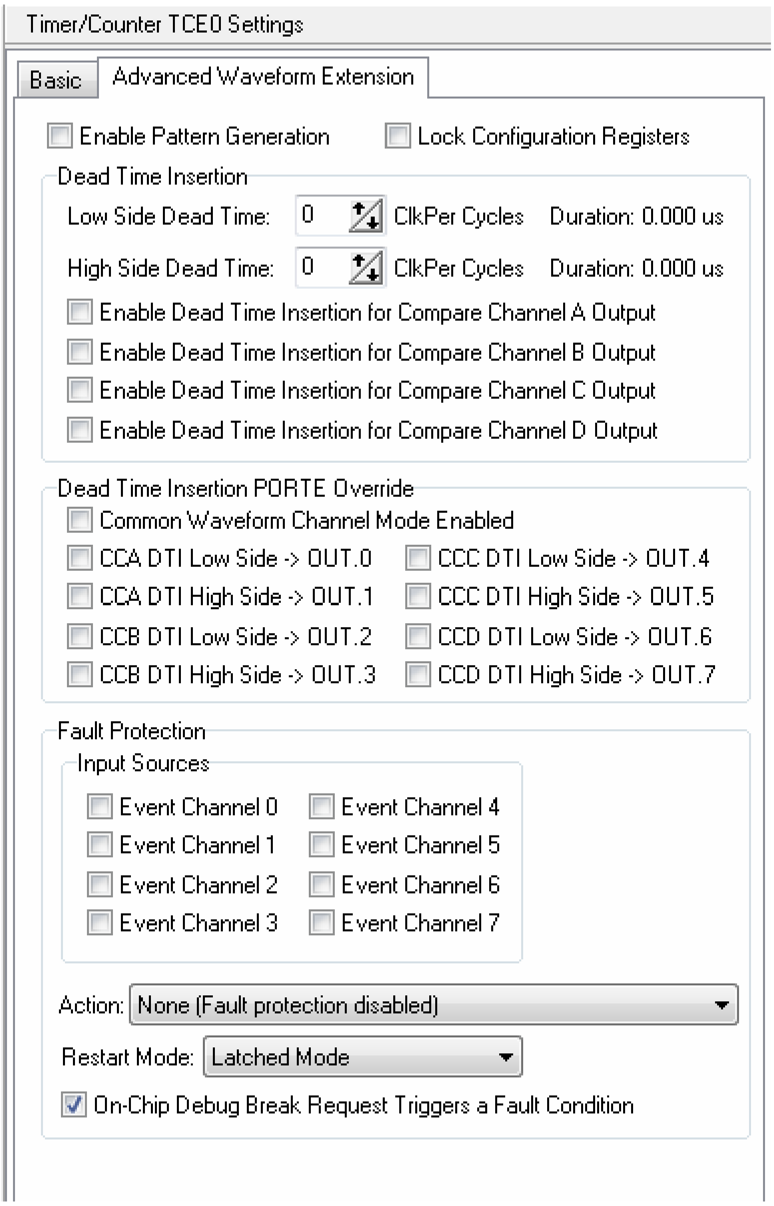
In **Single Slope PWM Generation** and **Double Slope PWM Generation** modes the following specific options are available:




The **Requirements** group box allows the user to specify the desired timer **Period** in ms and the **Duty Cycles** for the **Compare/Capture Channels** A, B, C and D.
Pressing the **Apply** button will perform automatic timer configuration (**Clock Source**, **PER**, **CCA**, **CCB**, **CCC** and **CCD** register values), so that the required timer period and duty cycles will be obtained for a given **Peripheral Clock** value.

If the **Compare Ch. A Output** option is enabled, the corresponding waveform generation (WG) output will be activated for compare matches between the **CNT** and **CCA** registers.
The same applies for the **Compare Ch. B Output**, **Compare Ch. C Output** and **Compare Ch. D Output** options and the corresponding **CCB**, **CCC** and **CCD** registers.

When operating in waveform generation modes the Timer/Counter can also use the **Advanced Waveform Extension (AWeX)** that provides some additional features.
It can be accessed by selecting the **Advanced Waveform Extension** tab:




The **Dead Time Insertion** group box contains the settings for the **Dead Time Insertion** (DTI) unit, that enables the generation of the non-inverted **Low Side** (LS) and inverted **High Side** (HS) waveforms on the corresponding I/O port pins.

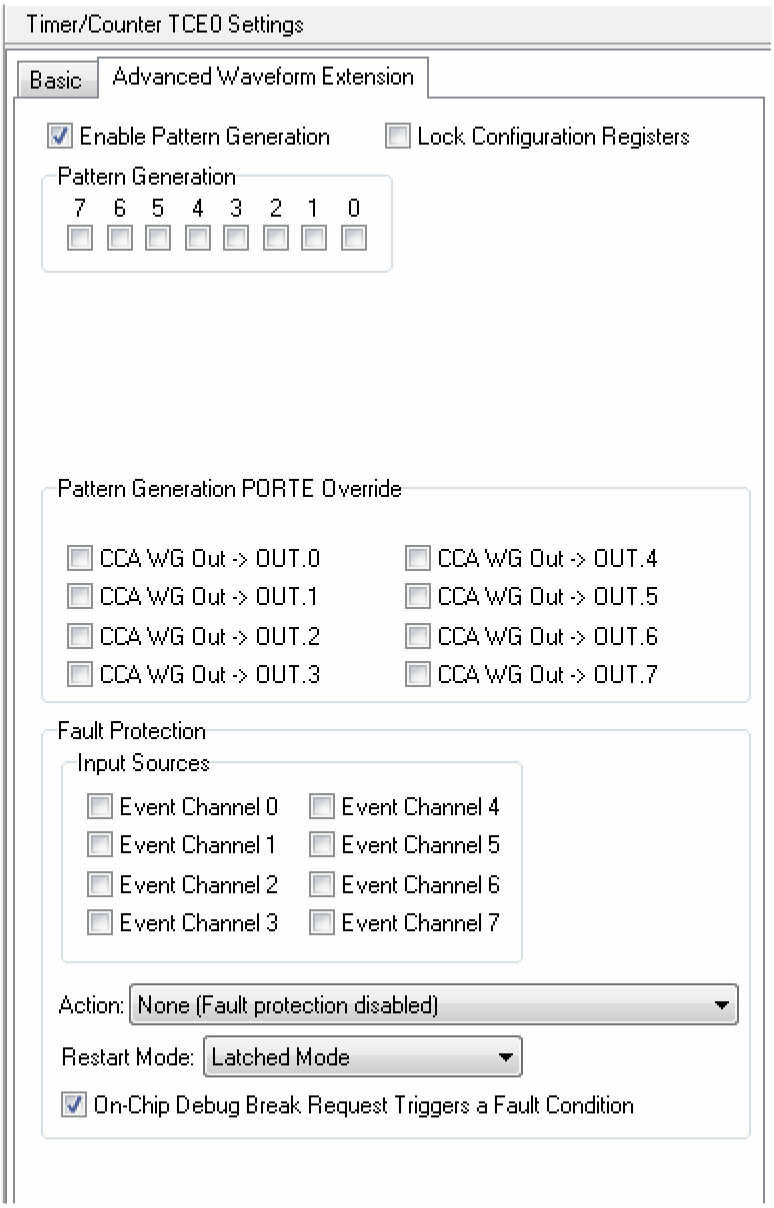
Dead times are inserted between **LS** and **HS** switching. These can be specified using the **Low Side Dead Time** and **High Side Dead Time** edit boxes.
Dead time insertion can be individually activated for each compare channel, using the corresponding **Enable Dead Time Insertion for Compare Channel Output** option.

The **Dead Time Insertion PORT Override** group box allows to individually specify which **LS** or **HS** waveforms will be outputed on the I/O port associated with the timer.
If the **Common Waveform Channel Mode Enabled** option is activated, the **Compare Channel A** waveform will be used as input for all the dead time generators. The waveforms of **Compare Channels** B, C and D will be ignored.

If the **Enable Pattern Generation** option is activated, the pattern generator extension will be used to produce a synchronized bit pattern on the I/O port associated with the timer.
The **DTI** unit is not activated in this case, its registers will be used by the pattern generator.

The pattern can be specified using the **Pattern Generation** check boxes.
This value will be used to initialize the **DTIHS** register.

The **Pattern Generation PORT Override** check boxes allow to specify to which I/O port pins, the waveform generated by the **Compare Channel A** will be outputed when an UPDATE condition is set by the waveform generation mode.
This value will be used to initialize the **DTILS** register.




The **Fault Protection** group box allows to specify the **Input Sources** and **Action** to be performed when a fault is detected.
The fault protection beeing event controlled, any event from the **Event System** can be used to trigger a fault action.

The following event **Actions** are possible:
- **None**
- **Clear all Override Enable Bits** in the **OUTOVEN** register, disabling the output override on all Timer/Counter outputs
- **Clear all Direction Bits** in the I/O port **DIR** register, setting all port pins as tri-stated inputs.

The **Restart Mode** list box allows to specify how the **AWeX** and Timer/Counter can return from the fault state and restore normal operation, when the fault condition is no longer active:
- **Latched Mode** - the waveform output will remain in the fault state until the the fault condition is no longer active and the fault detection flag **FDF** in the **AWEXn.STATUS** register will be cleared by software. When both these conditions are met, the waveform will return to normal operation at the next **UPDATE** condition.
- **Cycle-by-Cycle Mode** - the waveform output will remain in the fault state until the the fault condition is no longer active. When this condition is met, the waveform will return to normal operation at the next **UPDATE** condition.

The **On-Chip Debug Break Request Triggers a Fault Condition** option specifies if an OCD break request will be treated as a fault.

The **Change Protection** option allows to prevent unintentional changes to the Timer/Counter: **CTRLA** and AWeX: **OUTOVEN**, **FDEMASK** registers.

In order to initialize enabled Timers/Counters the CodeWizardAVR generates the functions:

**void tc*mn*_init(void)**

where: *m* - is the lowercase suffix of the I/O port where the Timer/Counter is implemented
*n* - is the number of the Timer/Counter on the port, starting with 0.

An unused Timer/Counter of type 0 can be disabled by calling the function:

**void tc0_disable(TC0_t *ptc)**

where **ptc** is a pointer to the correspoding **TC0_t** structure.
Example:

```
/* TCC0 is not used, so disable it */
tc0_disable(&TCC0);
```

An unused Timer/Counter of type 1 can be disabled by calling the function:

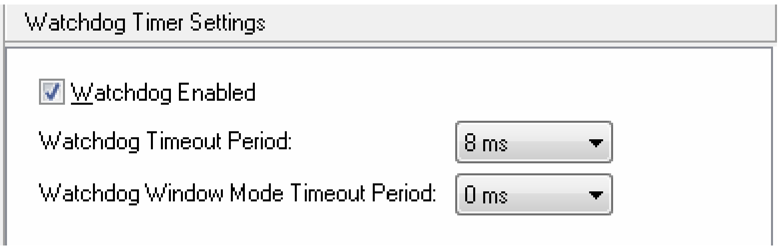
**void tc1_disable(TC1_t *ptc)**

where **ptc** is a pointer to the correspoding **TC1_t** structure.

## 7.8 Setting the Watchdog Timer

The XMEGA Watchdog Timer (WDT) can be configured by clicking on the **Watchdog Timer** node of the CodeWizardAVR selection tree.

The following options are available:




The **Watchdog Enabled** option allows to activate the **WDT**.

The **Watchdog Timeout Period** list box allows to specify the **open window** time period *after* which the **WDT** will issue a system reset (in **Normal** and **Window** operating modes), if the application code has not reset the **WDT** using the **WDR** instruction.

The **Watchdog Window Mode Timeout Period** allows to specify the length of the **closed window** time period (in **Window** operating mode), in which if the application code uses the **WDR** instruction, to try to reset the **WDT**, the **WDT** will issue a system reset.
If the application code resets the **WDT** *after* the **closed window** time elapses, but during the **open window,** no system reset will occur.

**Note:** If the **Watchdog Window Mode Timeout Period** is set to 0, the **WDT** will operate in **Normal** mode.

The **WDT** initialization is performed by the:

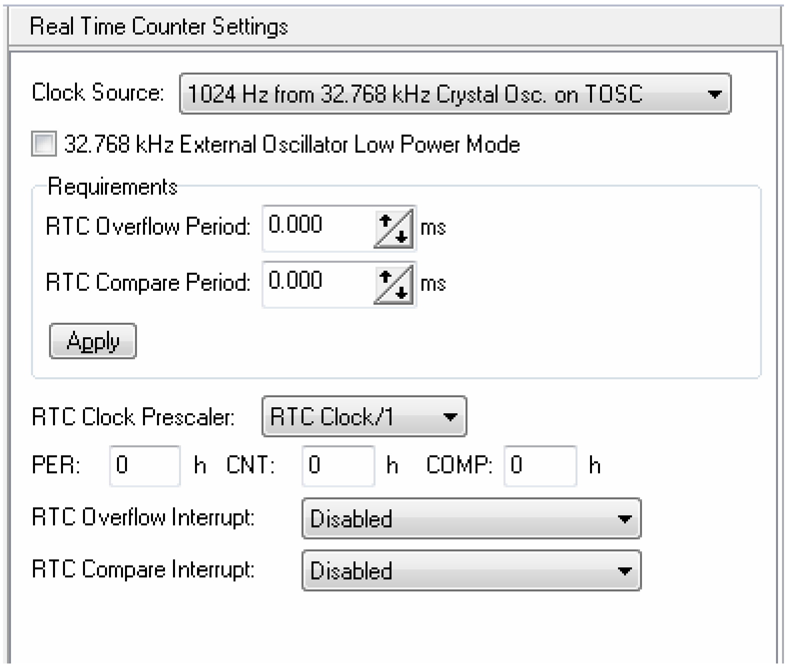
**void watchdog_init(void)**

function generated by the CodeWizardAVR.

## 7.9 Setting the 16-Bit Real Time Counter

The XMEGA 16-Bit Real Time Counter (RTC) can be configured by clicking on the **Real Time Counter** node of the CodeWizardAVR selection tree.

The following options are available:




The **Clock Source** list box allows to select the signal that will be used as **RTC** clock:

- 1024 Hz obtained from the 32 kHz internal Ultra Low Power oscillator
- 1024 Hz obtained from the 32.768 kHz external crystal oscillator on the TOSC1 and TOSC2 pins
- 1024 Hz obtained from the 32 kHz internal RC oscillator
- 32.768 kHz obtained from the 32.768 kHz external crystal oscillator on the TOSC1 and TOSC2 pins.

If the 32.768 kHz external crystal oscillator is used, it can be configured to operate in **Low Power Mode** by checking the appropriate option.

The **RTC** can be configured automatically by specifying the **RTC Overflow** and **RTC Compare** periods and clicking on the **Apply** button from the **Requirements** group box.
This will set the optimal values for the **RTC Clock Prescaler** list box, **PER** (period) and **COMP** (compare) 16-bit registers.

The initial value for the **CNT** (count) 16-bit register can be specified in hexadecimal, using the appropriate edit box.

The **RTC** can generate two types of interrupts:

- RTC Overflow Interrupt
- RTC Compare Interrupt.

Each type of interrupt can be individually enabled and its priority set, using the corresponding list boxes.

The **RTC** initialization is performed by the:

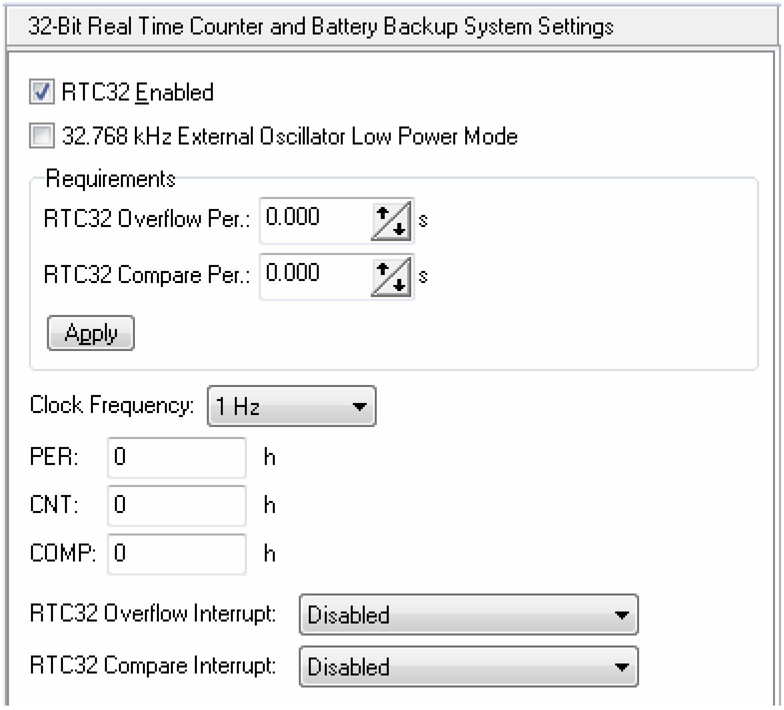
**void rtcxm_init(void)**

function generated by the CodeWizardAVR.

## 7.10 Setting the 32-Bit Real Time Counter and Battery Backup System

The XMEGA 32-Bit Real Time Counter (RTC32) and Battery Backup System can be configured by clicking on the **RTC32 and Battery Backup** node of the CodeWizardAVR selection tree.

The following options are available:




The **RTC32 Enabled** option allows to activate the operation the 32-Bit Real Time Counter and associated Battery Backup System.

The **RTC32** can be clocked by a 1 Hz or 1024 Hz signal, selected using the **Clock Frequency** list box. This signal is obtained by dividing the output of the 32.768 kHz external crystal oscillator, which can be configured to operate in **Low Power Mode** by checking the appropriate option.

The **RTC32** can be configured automatically by specifying the **RTC32 Overflow** and **RTC32 Compare** periods and clicking on the **Apply** button from the **Requirements** group box.
This will set the optimal values for the **Clock Frequency** list box, **PER** (period) and **COMP** (compare) 32-bit registers.

The initial value for the **CNT** (count) 32-bit register can be specified in hexadecimal, using the appropriate edit box.

The **RTC32** can generate two types of interrupts:

- RTC32 Overflow Interrupt
- RTC32 Compare Interrupt.

Each type of interrupt can be individually enabled and its priority set, using the corresponding list boxes.

When it is enabled, the **RTC32** initialization is performed by the:

**void rtc32_battery_backup_init(void)**

function generated by the CodeWizardAVR.

If the **RTC32** is disabled, the initialization is performed by the:

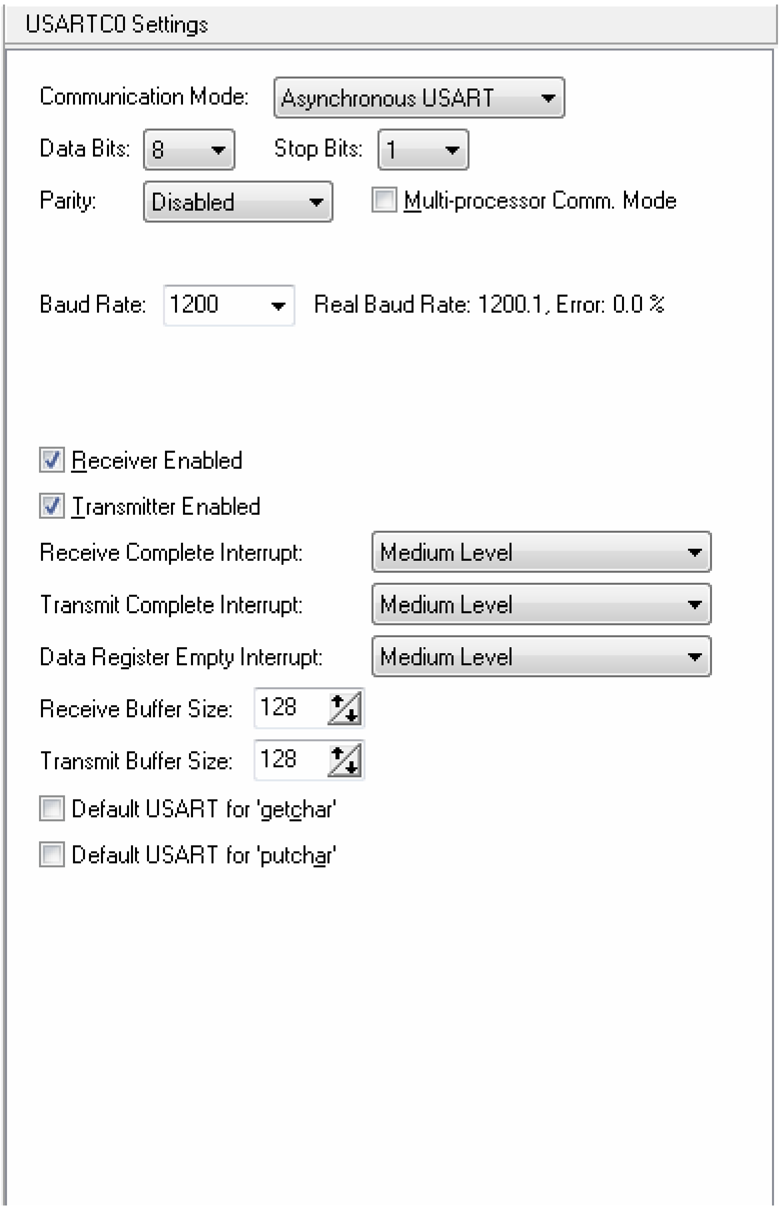
**void rtc32_battery_backup_disable(void)**

function.

## 7.11 Setting the USARTs

The XMEGA USARTs can be configured by clicking on the **USARTs** and **USARTn** nodes of the CodeWizardAVR selection tree.

The following options are available for configuring each USART:




The **Communication Mode** list box allows to select one of the following operating modes:

- Asynchronous USART
- Synchronous USART
- Infrared Module (IRDA 1.4)
- Master SPI.

For the **Asynchronous**, **Synchronous** and **Infrared Module**, the following specific options are available:

The **Data Bits** option specifies the number of data bits in a data frame: 5 to 9.
The **Stop Bits** option specifies the number of stop bits in a data frame: 1 or 2.
The **Parity** bit in a data frame can be:
- Disabled
- Even
- Odd.

If the **Multi-processor Comm. Mode** option is enabled, a dedicated bit in the frame is used to indicate whether the frame is an address or data frame.
If the Receiver is set up to receive frames that contain 5 to 8 data bits, the first stop bit is used to indicate the frame type. If the Receiver is set up for frames with 9 data bits, the ninth bit is used for this purpose.

The **Baud Rate** list box allows to select the communication data rate.
The CodeWizardAVR automatically calculates the values for the **BSEL** and **SCALE**, for the current **Baud Rate** and **Peripheral Clock** values.
The **Real Baud Rate** and **Error** are displayed.

The Receiver, respectively Transmitter, can be activated using the **Receiver Enabled**, respectively **Transmitter Enabled** check boxes.

The USART can generate several types of interrupts:
- Receive Complete Interrupt
- Transmit Complete  Interrupt
- Data Register Empty Interrupt.

Each type of interrupt can be individually enabled and its priority set, using the corresponding list boxes.

If buffered interrupt driven serial communication will be used, the sizes of the Receiver, respectively Transmitter, buffers can be specified using the **Receiver Buffer Size**, respectively **Transmitter Buffer Size** edit boxes.

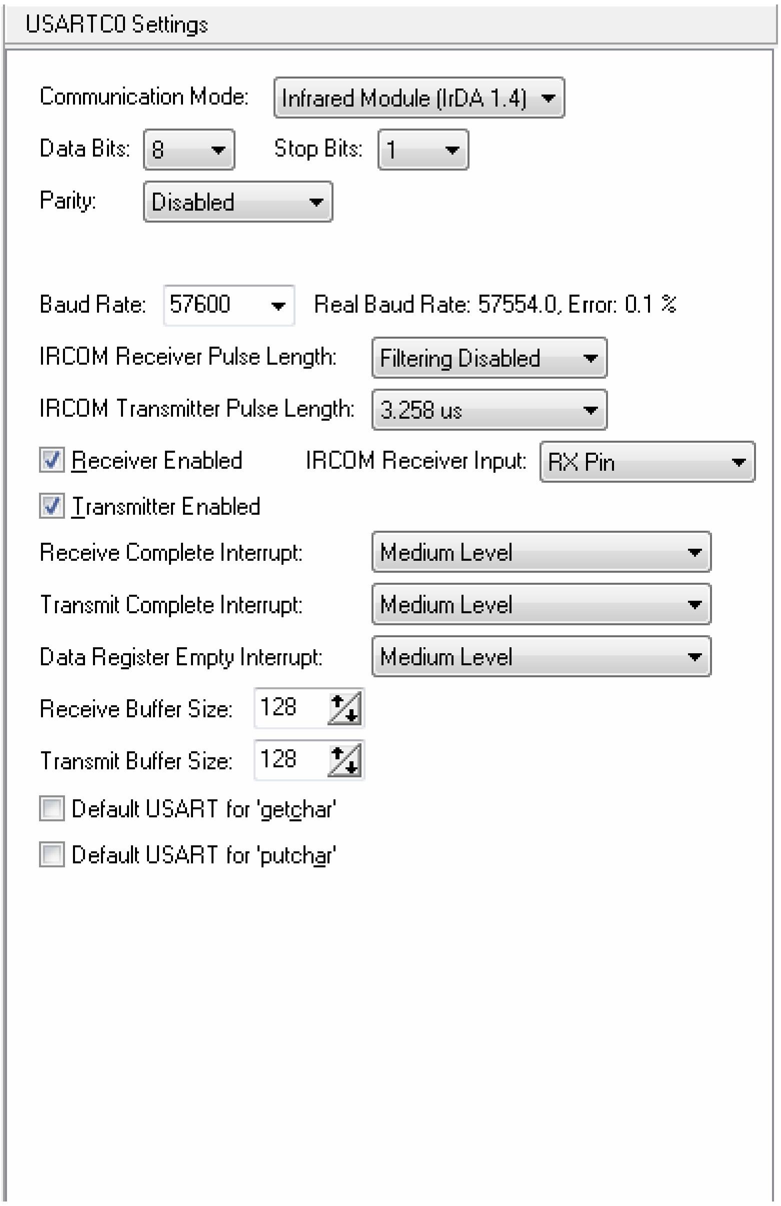
In order to allow receiving, respectively transmitting data using the USART, the CodeWizardAVR will define the **getchar_usart*pn***, respectively **putchar_usart*pn*** functions, where ***p*** is the I/O port letter and ***n*** is the USART number in the port.

One of the USARTs can be chosen as the default communication device to be used by the **getchar**, respectively **putchar**, **Standard C Input/Output Functions** by enabling the **Default USART for 'getchar'**, respectively the **Default USART for 'putchar'** options.
In this situation the standard **getchar** and **putchar** functions are redefined in the generated program.

For interrupt driven serial communication, some additional global variables will be declared during code generation.
The receiver buffer is implemented using the global array **rx_buffer_usart*pn*.**
The global variable **rx_wr_index_usart*pn*** is the **rx_buffer_usart*pn*** array index used for writing received characters in the buffer.
The global variable **rx_rd_index_usart*pn*** is the **rx_buffer_usart*pn*** array index used for reading received characters from the buffer by the **getchar_usart*pn*** function.
The global variable **rx_counter_usart*pn*** contains the number of characters received in **rx_buffer_usart*pn*** and not yet read by the **getchar_usart*pn*** function.
If the receiver buffers overflows the **rx_buffer_overflow_usart*pn*** global bit variable will be set.

The transmitter buffer is implemented using the global array **tx_buffer_usart*pn***.
The global variable **tx_wr_index_usart*pn*** is the **tx_buffer_usart*pn*** array index used for writing in the buffer the characters to be transmitted.
The global variable **tx_rd_index_usart*pn*** is the **tx_buffer_usart*pn*** array index used for reading from the buffer the characters to be transmitted by the **putchar_usart*pn*** function.

If the **Infrared Module** communication mode is used, some specific options are available:

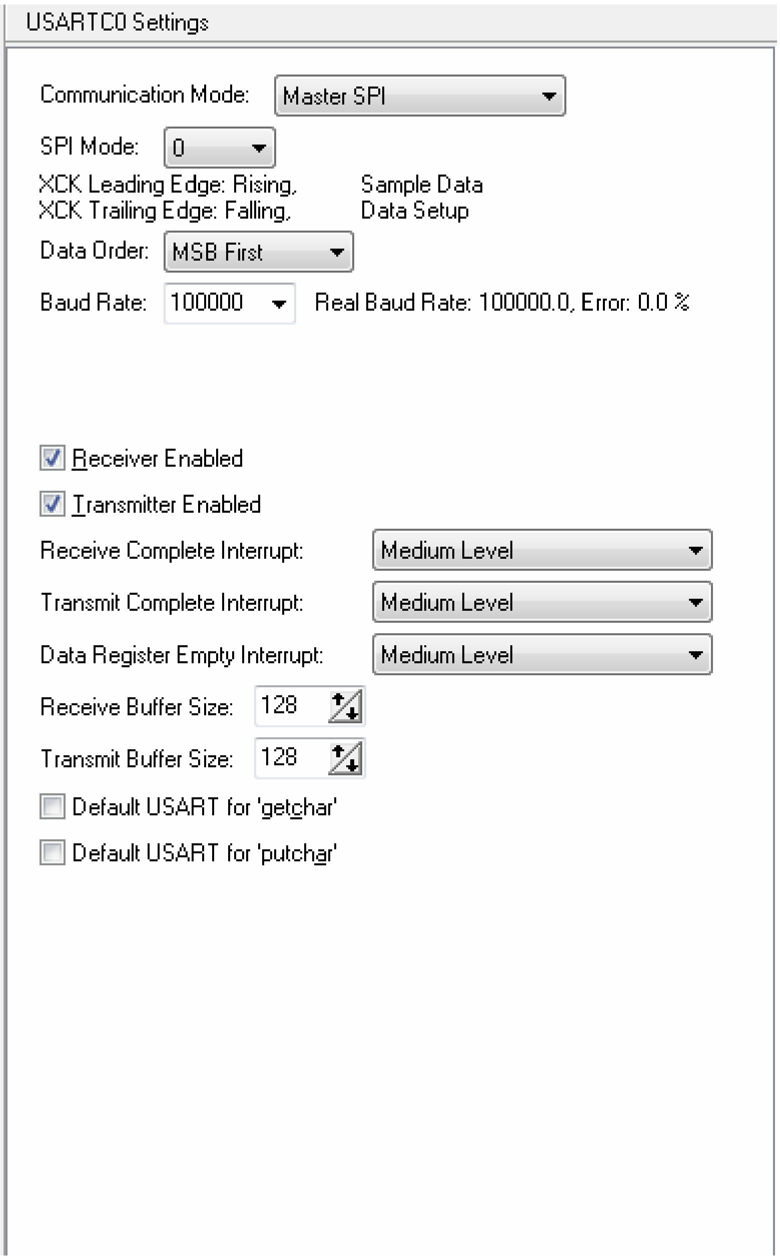



The **IRCOM Receiver Pulse Length** option sets the filter coefficient for the IRCOM Receiver.
If enabled, it represent the number of samples required for pulse to be accepted.

The **IRCOM Transmitter Pulse Length** sets the pulse modulation scheme for the IRCOM Transmitter.

The **IRCOM Receiver Input** list box allows to select if the input of the IRCOM Receiver will be connected to the **RX** port pin or to one of the **Event System Channels** 0 to 7.

If the **Master SPI** communication mode is used, some specific options are available:




The **SPI Mode** can be:

- Mode 0
- Mode 1
- Mode 2
- Mode 3.

The **Data Order** in the frame can be:

- MSB First
- LSB First.

For enabled **USART**s the CodeWizardAVR generates the functions:

**void usart*mn*_init(void)**

where: *m* - is the lowercase suffix of the I/O port where the **USART** is implemented
*n* - is the number of the **USART** on the port, starting with 0.

An unused **USART** can be disabled by calling the function:

**void usart_disable(USART_t *pu)**

where **pu** is a pointer to the correspoding **USART_t** structure.
Example:

```
/* USARTC1 is not used, so disable it */
usart_disable(&USARTC1);
```

For transmitting data the CodeWizardAVR generates the functions:

**void putchar_usart*mn*(char c)**

where: c - is the character to be transmitted
*m* - is the lowercase suffix of the I/O port where the **USART** is implemented
*n* - is the number of the **USART** on the port, starting with 0.

For receiving data the CodeWizardAVR generates the functions:

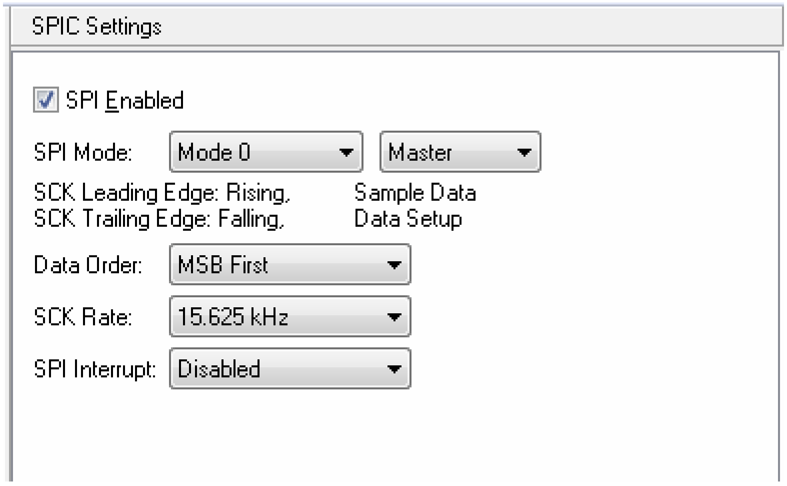
**char getchar_usart*mn*(void)**

where: *m* - is the lowercase suffix of the I/O port where the **USART** is implemented
*n* - is the number of the **USART** on the port, starting with 0.

## 7.12 Setting the Serial Peripheral Interfaces

The XMEGA Serial Peripheral Interfaces (SPI) can be configured by clicking on the **Serial Peripheral Interfaces** SPI nodes of the CodeWizardAVR selection tree.

The following options are available:




The **SPI Enabled** check box allows to activate the corresponding Serial Peripheral Interface.

The **SPI Mode** can be:
- Mode 0
- Mode 1
- Mode 2
- Mode 3.

The **SPI** can operate as:
- Master
- Slave.

The **Data Order** in the frame can be:
- MSB First
- LSB First.

If the **SPI** operates as a **Master**, it will generate the **SCK** clock signal for the slave(s).
The frequency of this signal, obtained by dividing the **ClkPer** peripheral clock, can be selected using the **SCK Rate** list box.

The **SPI Interrupt** can be enabled and its priority set, using the corresponding list box.

The initialization of each **SPI** peripheral is performed by the:

**void spi*m*_init(void)**

functions generated by the CodeWizardAVR, where ***m*** is the lowercase suffix of the I/O port where the **SPI** is implemented.

If the **SPI Interrupt** is disabled, the **SPI** will operate in polled mode.
The following transmit/receive function will be generated by the CodeWizardAVR for this situation for **Master** mode:

**unsigned char spi*m*_master_tx_rx(unsigned char c)**

where: *m* - is the lowercase suffix of the I/O port where the **SPI** is implemented
c - is the byte to be transmitted to the slave.

The function will return the byte received from the slave.
The **SPI** beeing operated in polled mode, this function will be blocking, as the state of the **SPIF** flag from the **STATUS** register will be tested in an endless loop, until one byte will be transmitted/received.
**Note:** The **spi*m*_master_tx_rx** function doesn't control the **/SS** signal.
The **/SS** line must be set low in order to select the slave before calling this function.
The **SET_SPI*M*_SS_LOW** macro is defined by the CodeWizardAVR for this purpose, where ***M*** is the suffix of the I/O port where the **SPI** is implemented.
After all communication is finished on the bus, the **/SS** line must be set high in order to deselect the slave.
This is accomplished using the **SET_SPI*M*_SS_HIGH** macro defined by the CodeWizardAVR for this purpose, where ***M*** is the suffix of the I/O port where the **SPI** is implemented.
Example for **SPIC** operating as a master:

```
/* Select the SPI slave */
SET_SPIC_SS_LOW
/* Send two bytes of data to the slave */
spic_master_tx_rx(0x12);
spic_master_tx_rx(0x34);
/* Deselect the SPI slave */
SET_SPIC_SS_HIGH
```

When operating as a **Slave**, the CodeWizardAVR will generate the function:

**unsigned char spi*m*_slave_tx_rx(unsigned char c)**

where: *m* - is the lowercase suffix of the I/O port where the **SPI** is implemented
c - is the byte to be transmitted to the master.

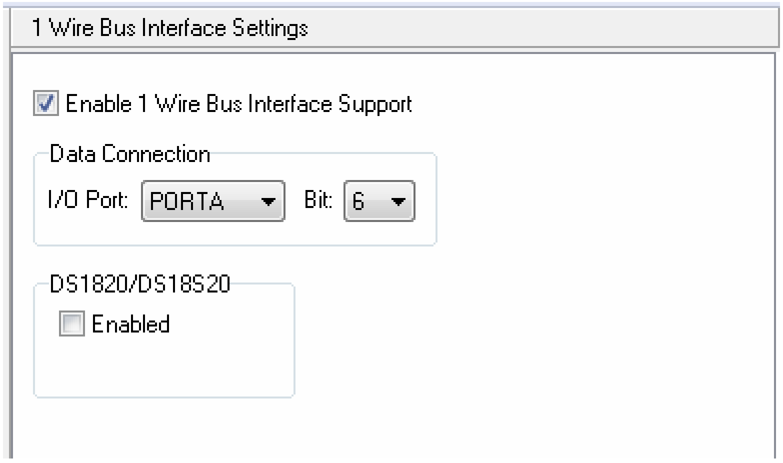
The function will return the byte received from the master.
The **SPI** beeing operated in polled mode, this function will be blocking, as the state of the **SPIF** flag from the **STATUS** register will be tested in an endless loop, until one byte will be transmitted/received.

In order to prevent such situations it is recommended to enable the **SPI Interrupt**.
The CodeWizardAVR will then generate the **spi*m*_isr SPI** interrupt service routine, where ***m*** is the lowercase suffix of the I/O port where the **SPI** is implemented.
Inside this function, the received data will be processed only when its received, and new data will be prepared to be transmitted, without blocking the execution of the rest of the application.

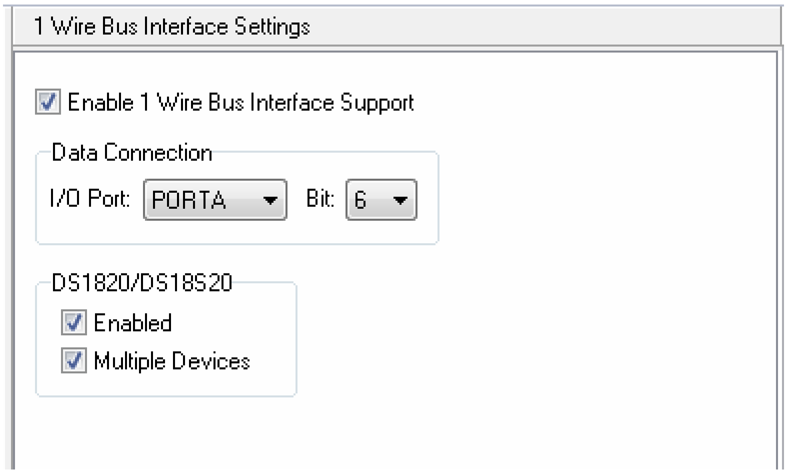
## 7.13 Setting the 1 Wire Bus

The 1 Wire Protocol Functions for the XMEGA chips can be configured by clicking on the **1 Wire**  node of the CodeWizardAVR selection tree.

The following settings are available:




- **Enable 1 Wire Bus Interface Support** allows the activation of the **1 Wire Protocol Functions**.
- **I/O Port** and **Bit** specify in **Data Connection**, the port and bit used for 1 Wire bus communication
- **DS1820/DS18S20 Enabled** check box activates the generation of support code for accessing the DS1820/DS18S20 temperature sensor devices:



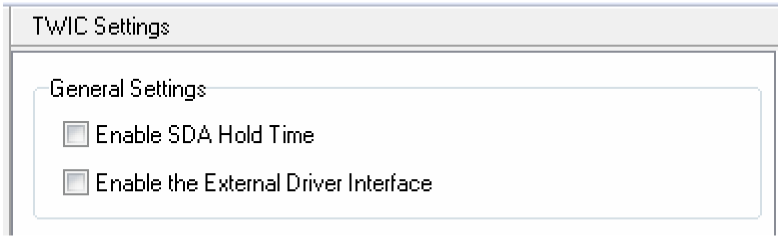

If several DS1820/DS18S20 devices are connected to the 1 Wire bus, the **Multiple Devices** option must be checked.
A maximum of 8 DS1820/DS18S20 devices can be connected to the bus.
The ROM codes for these devices will be stored in the **ds1820_rom_codes** array.

The DS1820/DS18S20 devices can be accessed using the **Maxim/Dallas Semiconductor DS1820/DS18S20 Temperature Sensors Functions**.

## 7.14 Setting the Two Wire Interfaces

The XMEGA Two Wire Interfaces (TWI) can be configured by clicking on the **Two Wire Interfaces** nodes of the CodeWizardAVR selection tree.

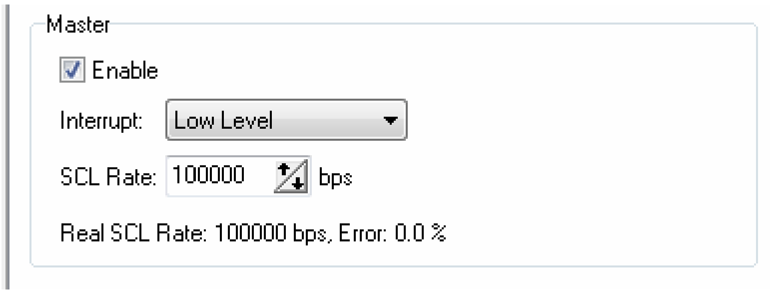
The following **General Settings** are available:




- **Enable SDA Hold Time** allows to add an internal hold time to the SDA signal with respect to the negative edge of SCL.
- **Enable the External Driver Interface** activates the usage of external TWI compliant tri-state drivers for the SDA and SCL signals.

In this situation the internal TWI drivers with input filtering and slew rate control are bypassed.
The normal I/O port pin function is used and the direction must be configured by the user software.

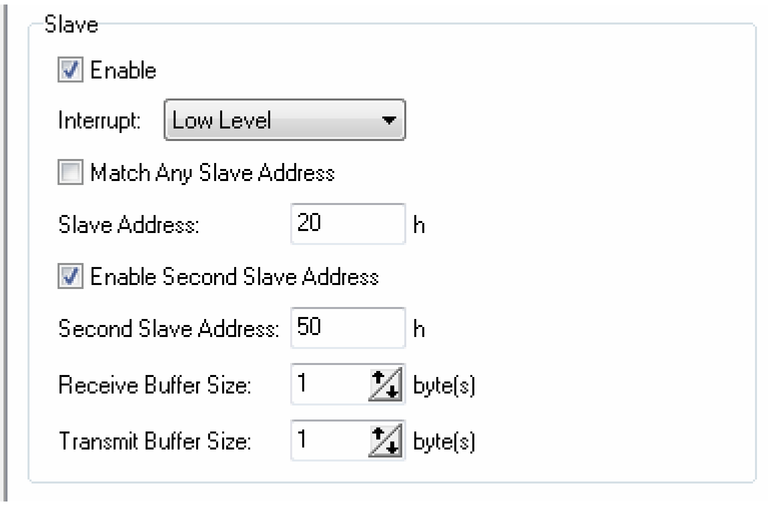
The following settings are available for operating the TWI module in **Master** mode:



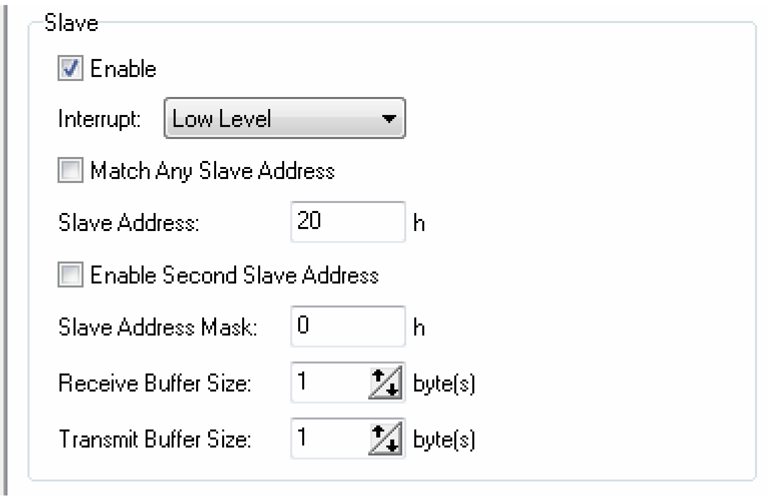

- **Enable** activates the operation of the TWI module in master mode.
- **Interrupt** specifies the interrupt priority level used by the TWI module when operating in master mode.
- **SCL Rate** specifies the required TWI clock rate on the SCL pin.

The **Real SCL Rate** is calculated and displayed based on the **System Clock** value.

The following settings are available for operating the TWI module in **Slave** mode:




- **Enable** activates the operation of the TWI module in slave mode.
- **Interrupt** specifies the interrupt priority level used by the TWI module when operating in slave mode.
- **Match Any Slave Address** enables the TWI slave to respond to any slave address supplied by the master when starting a transaction.
- **Slave Address** represents the 7 bit slave address to which the slave will respond if the **Match Any Slave Address** option is **disabled**.
- **Enable Second Slave Address**, when **enabled**, allows to specify a **Second Slave Address** to which the slave should respond.
- **Slave Address Mask**, when the **Enable Second Slave Address** option is **disabled**, represents the 7 bit slave address bit mask applied to the **Slave Address**:




If a bit in the **Slave Address Mask** is set to 1, the address match between the incoming address bit and the corresponding bit from the **Slave Address** is ignored, i.e. masked bits will always match.

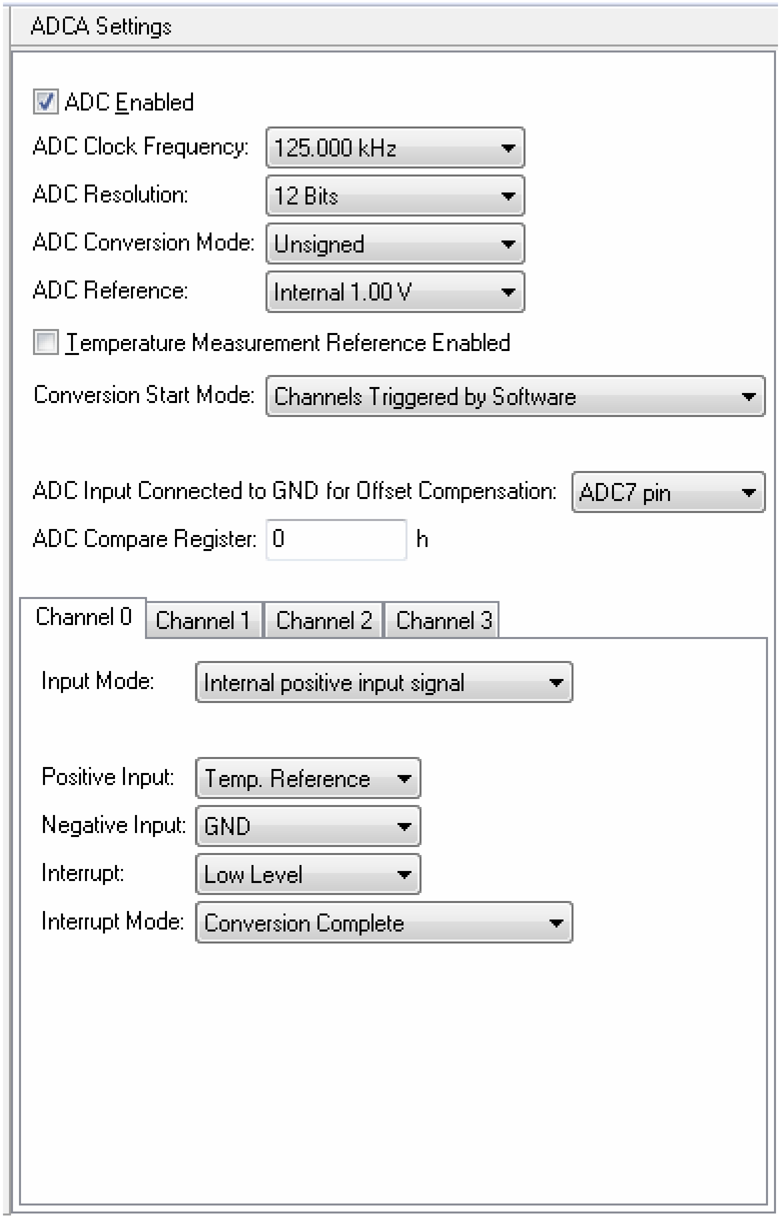
- **Receive Buffer Size** specifies the size of the receive buffer in bytes.
- **Transmit Buffer Size** specifies the size of the transmit buffer in bytes.

After the TWI is configured, the CodeWizardAVR will generate code that uses the **Two Wire Interface Functions for XMEGA Devices** library.

## 7.15 Setting the Analog to Digital Converters

The XMEGA Analog to Digital Converter(s) (ADC) can be configured by clicking on the **Analog to Digital Converters** nodes of the CodeWizardAVR selection tree.

The following settings are available:



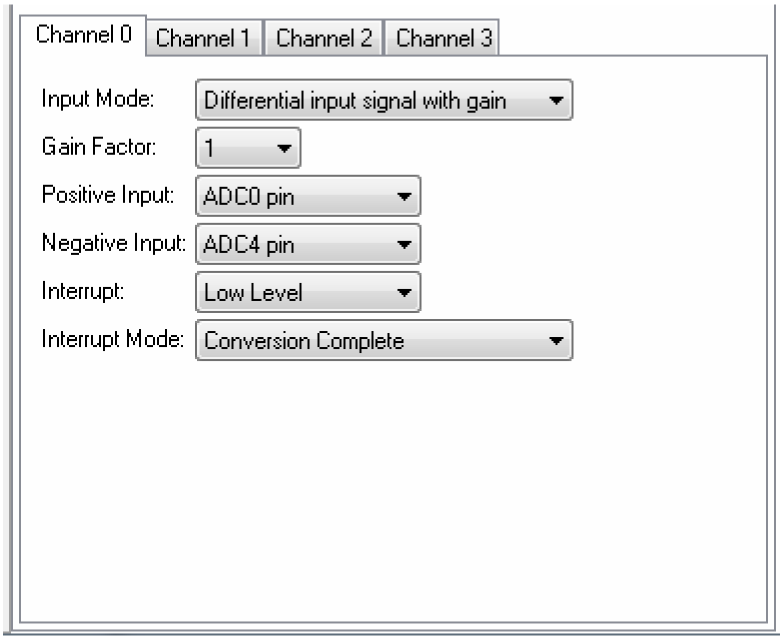

- **ADC Enabled** allows the activation of the selected ADC
- **ADC Clock Frequency** specifies the frequency of the clock signal used by the ADC
- **ADC Resolution** allows to select **8 Bit** or **12 Bit** analog to digital conversion resolution
- **ADC Conversion Mode** allows to select **Signed** or **Unsigned** analog to digital conversion
- **ADC Reference** allows to specify the voltage reference used by the ADC
- **Temperature Measurement Reference Enabled** activates the on-chip temperature sensor
- **Conversion Start Mode** allows to select how an analog to digital conversion is started for each ADC channel: **Triggered by Software**, **Free Running**, **Triggered by the Event System** and **Triggered by the Event System, Synchronized**

- **ADC Input Connected to GND for Offset Compensation** allows to specify which ADC input is connected to GND, in **Unsigned** conversion mode, so that the ADC offset voltage can be measured at start-up and substracted from subsequent conversion results. For **Signed** conversion mode, there is no need to connect one of the ADC inputs to GND, because the offset is read using channel 0 in differential mode, with both + and - inputs connected internally together.
- **ADC Compare Register** allows to specify the initial value for the ADC **CMPL** and **CMPH** registers

For each ADC channel there is the possibility to specify the following options:

- **Input Mode** specifies which signals are applied to the ADC channel: **Internal positive input**, **Single-ended external positive input signal**, **Differential external input signal** and **Differential external input signal with gain**.

For the last input mode there is also the option to select the **Gain Factor**:

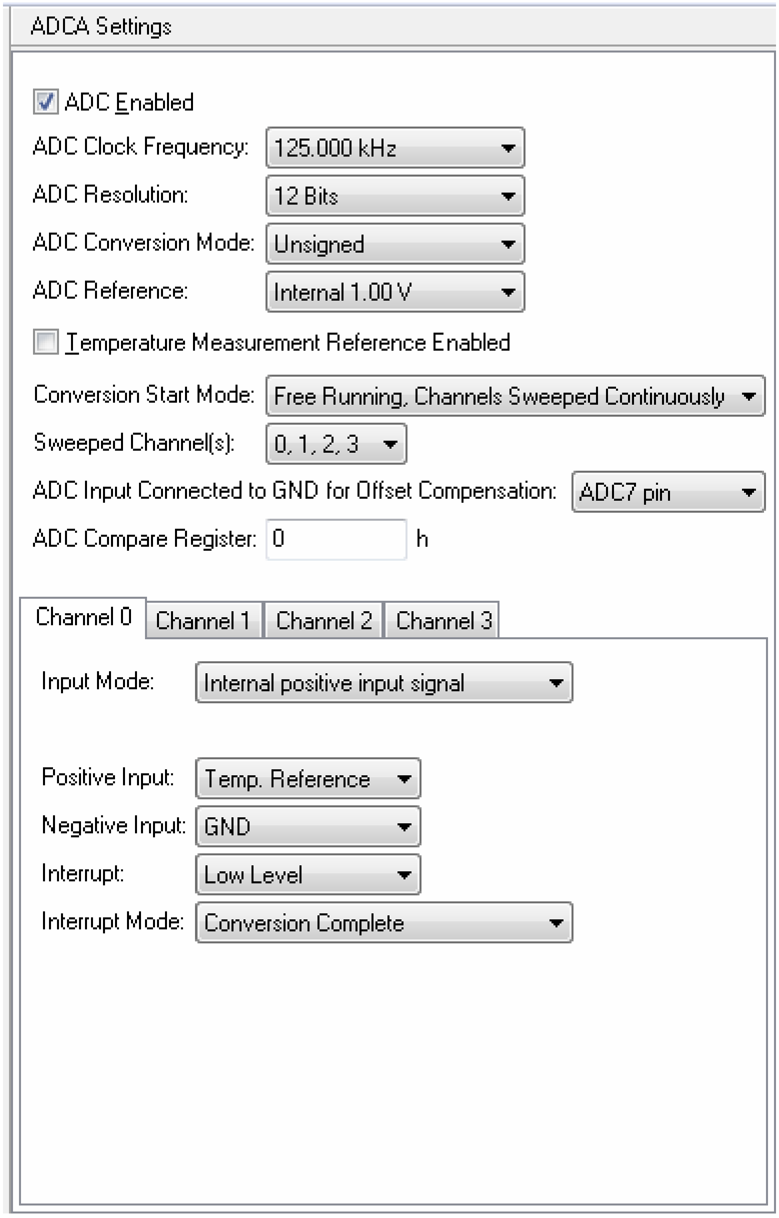



**Note:** Differential input modes are available only for **Signed** ADC conversion mode.

- **Positive Input** allows to select which ADC pin will be used as channel's positive input for modes that use external input signals
- **Negative Input** allows to select which ADC pin will be used as channel's negative input for **Differential** input modes
- **Interrupt** allows to specify if an interrupt will be generated for the ADC channel, by one of the events specified in the **Interrupt Mode** option: **Conversion Complete**, **Compare Result Below Threshold** and **Compare Result Above Threshold.**

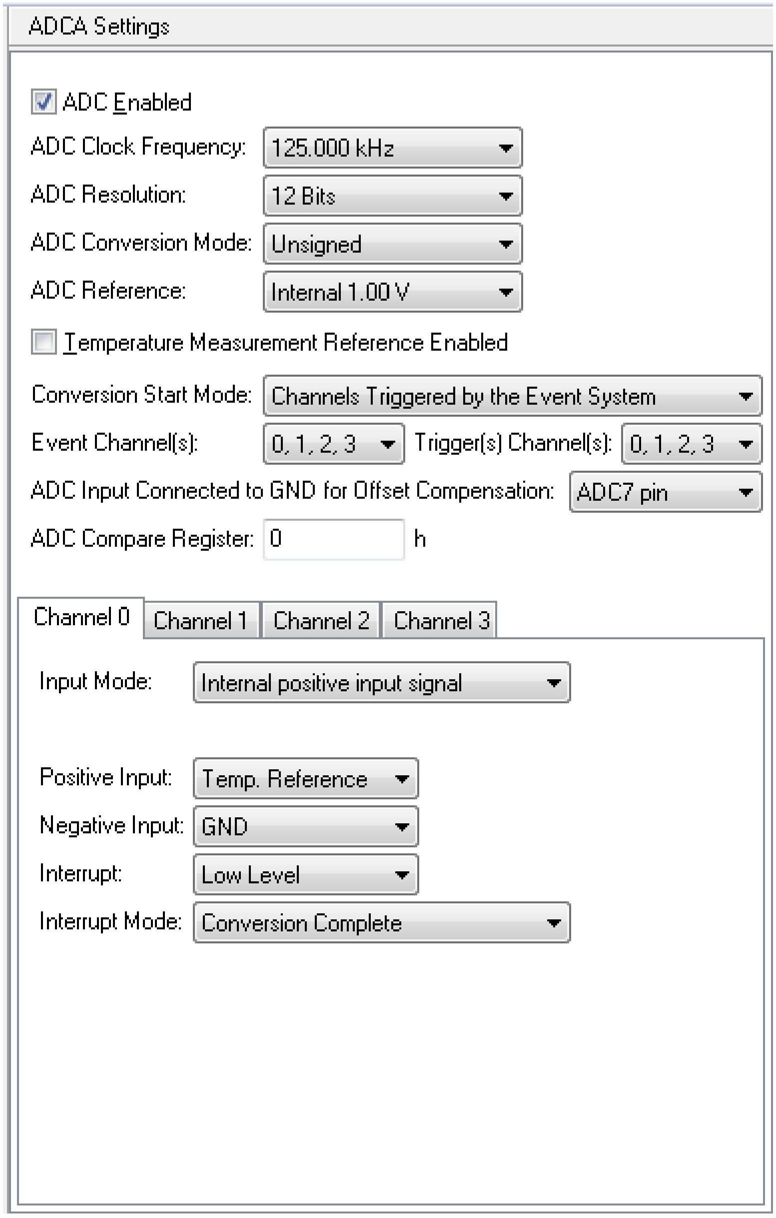
For the last two **Interrupt Modes**, the analog to digital conversion result is compared with the value of the **CMPL** and **CMPH** registers, an interrupt being generated if the specified condition is met.

In the **Free Running** conversion start mode, the ADC will continuously sweep the channels specified by the **Sweeped Channel(s)** list box:



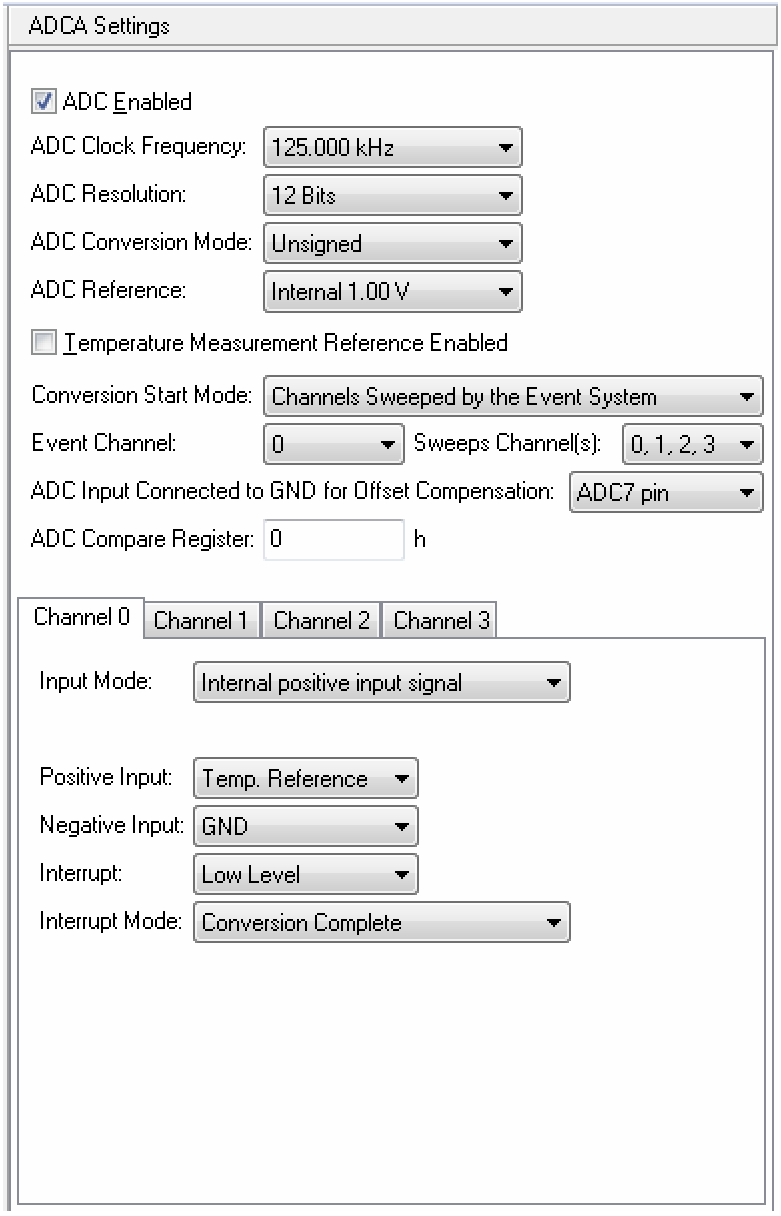

In the above example figure, the ADC will continuously sweep the inputs of channels 0, 1, 2 and 3.

In the **Channels Triggered by the Event System** conversion start mode, each **Event Channel** will trigger the analog to digital conversion for the ADC channels specified by the **Trigger(s) Channels(s)** list box:




In the above example figure, event channel 0 will trigger ADC channel 0, event channel 1 will trigger ADC channel 1, event channel 2 will trigger ADC channel 2 and event channel 3 will trigger ADC channel 3.

In the **Channels Sweeped by the Event System** and **Channels Sweeped by the Event System, Synchronized** conversion start modes, an event that occurs on **Event Channel** will trigger a sweep of all the ADC channels specified by the **Sweeps Channel(s)** list box:




In the above example figure, an event that occurs on event channel 0 will start the sweep of ADC channels' 0, 1, 2 and 3 inputs.

The following functions are generated by the CodeWizardAVR:

**void adc*n*_init(void)**

This function will initialize the ADC*n* peripheral, where *n* is the name of the PORT where the ADC inputs are located.
The function will also perform the ADC calibration in 12 Bit mode (by reading the calibration value from the signature row), the measurement of its offset voltage and will store the offset value in the **adc*n*_offset** variable. This value will be later substracted for offset compensation when performing analog to digital conversions.

**void adc*n*_conv_start(unsigned char channel)**

This function will start an analog to digital conversion for the specified **channel** of ADC*n*.
It is generated only in **Channels Triggered by Software** conversion start mode, if interrupts are enabled for at least one of the ADC*n* channels.
After the conversion is complete, its result can be read in the ADC*n* channel's interrupt service routine.

If software polling is used (interrupts are disabled for the ADC*n* channel) then the following functions can be used:

**unsigned int adc*n*_read(unsigned char channel)**

for **Unsigned** ADC conversion mode

**int adc*n*_read(unsigned char channel)**

for **Signed** ADC conversion mode.

When called, these functions will start an analog to digital conversion for the specified **channel** of ADC*n*, will wait for the conversion to complete, will compensate the ADC offset voltage and will return the conversion result.
**Note:** The analog to digital conversion will be started by the **adc*n*_read** functions only for **Channels Triggered by Software** conversion start mode. For the other start modes it will be automatically started by the ADC itself (**Free Running** mode) or by the **Event System**.

For the **Channels Triggered by Software, Channels Sweeped by the Event System** and **Channels Sweeped by the Event System, Synchronized** conversion start modes and if software polling is used (interrupts being disabled for at least one of the ADC*n* channels), the following functions are generated:

**void adc*n*_sweep_read(unsigned int *pdata)**

for **Unsigned** ADC conversion mode

**void adc*n*_sweep_read(int *pdata)**

for **Signed** ADC conversion mode.

When called, these functions will start analog to digital conversions for all the ADC*n* channels specified in the **Sweep** listbox, will wait for the conversions to complete, will compensate the ADC offset voltage and will store the conversion results in the array pointed by **pdata**.
**Note:** The analog to digital conversions will be started by the **adc*n*_sweep** functions only for **Channels Triggered by Software** conversion start mode. For the other start modes they will be automatically started by the **Event System**.

If interrupts are enabled for an ADC channel, then the corresponding interrupt service routine will be generated for it:

**void adc*n*_ch*m*_isr(void)**

where *n* is the name of the PORT where the ADC inputs are located and *m* is the number of the ADC*n* channel.
The interrupt service routine will contain code to read the conversion result from the **RESL** and **RESH** registers of ADC*n* channel *m* and compensate the ADC*n* offset voltage.

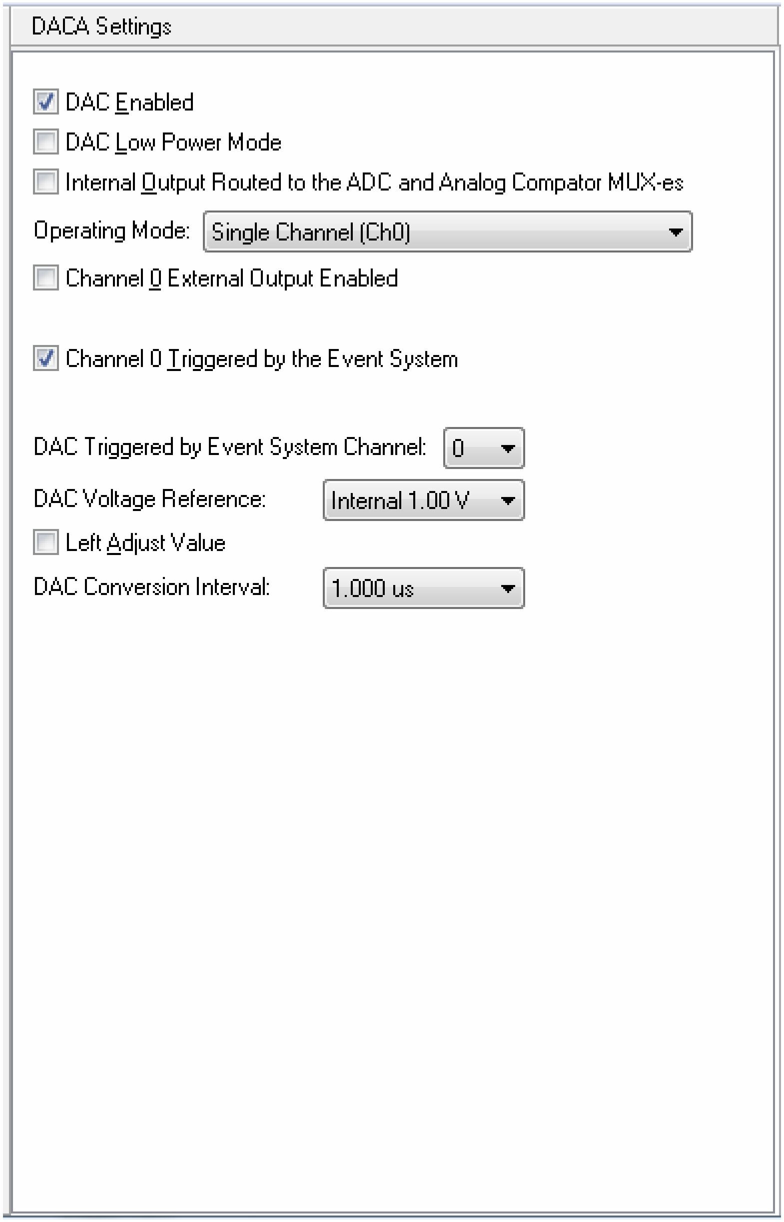
More details about ADC operation for the XMEGA chips can be found in the following Atmel documents:

- AVR1300: Using the XMEGA ADC
- XMEGA A Manual
- XMEGA D Manual.

## 7.16 Setting the Digital to Analog Converters

The XMEGA Digital to Analog Converter(s) (DAC) can be configured by clicking on the **Digital to Analog Converters** nodes of the CodeWizardAVR selection tree.

The following settings are available:




- **DAC Enabled** allows the activation of the selected DAC
- **DAC Low Power Mode** allows the operation of the DAC in low power consumption mode
- **Internal Output Routed to the ADC and Analog Comparator MUX-es** allows to internally connect the DAC's output to the corresponding ADC and/or Analog Comparator multiplexers
- **Operating Mode** allows to select one of the two DAC operating modes: **Single Channel** (Ch0) or **Dual Channel** (Ch0 and Ch1).

In **Single Channel** operating mode, the DAC output is directly connected to the Channel 0 output, without using a sample and hold circuit.
This output can be activated using the **Channel 0 External Output Enabled** check box.

A Digital to Analog conversion can be triggered by:
- writing a new value to the **CH0DATAL** and **CH0DATAH** registers
- the **Event System**, if the **Channel 0 Triggered by the Event System** option is enabled.

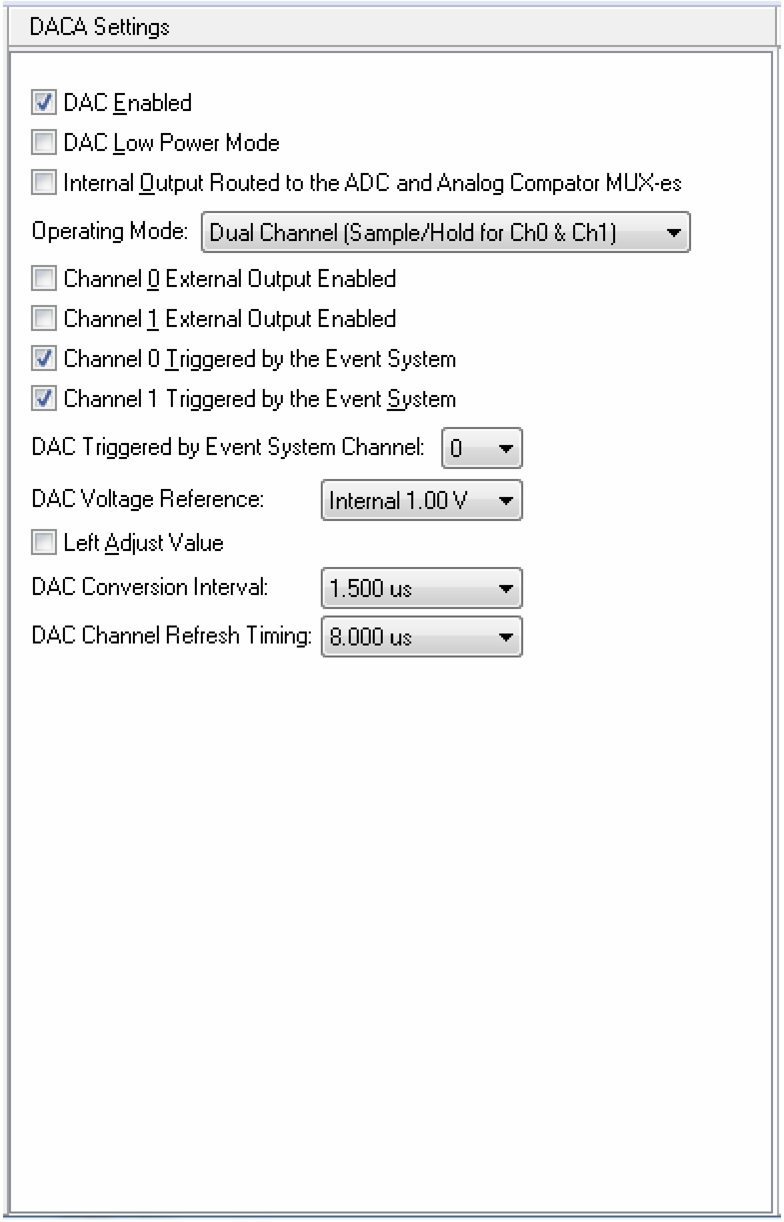
In this later case, the **DAC Triggered by Event System Channel** option allows to select the channel on which an event will trigger a conversion.

The **DAC Voltage** reference option allows to select the reference used by the DAC:
- 1.0 V internal reference
- the AVCC pin
- the AREF pin on PORTA
- the AREF pin on PORTB.

The **Left Adjust Value** option, if enabled, allows to distribute the data bits 4..11 to the **CH0DATAH,** and data bits 0..3 to the bits 4..7 of the **CH0DATAL** registers.
This also allows for 8-bit Digital to Analog conversions if only the **CH0DATAH** register is written.
If the option is disabled, the data bits 0..7 are distributed to the **CH0DATAL,** and data bits 8..11 to the bits 8..11 of the **CH0DATAH** registers.

The **DAC Conversion Interval** specifies the time interval between a completed channel conversion until starting a new conversion.
Its minimal value depends of the **Peripheral Clock** value and can't be less than 1 us in **Single Channel** operating mode.

In **Dual Channel** operating mode the DAC output is fed into two different pins, using a sample and hold circuit.
In this operating mode the following additional options are available:




- **Channel 1 External Output Enabled** check box allows to activate the output of DAC Channel 1 on the corresponding port pin
- **Channel 1 Triggered by the Event System** check box allows to trigger a Digital To Analog conversion for Channel 1 too, when an event occurs on the **Event System Channel** selected by the corresponding option
- **DAC Channel Refresh Timing** allows to specify the time interval between each time a channel is updated. This interval can't exceed 30 us.

The following functions are generated by the CodeWizardAVR:

**void dac*n*_init(void)**

This function will initialize the DAC*n* peripheral, where *n* is the name of the PORT where the DAC outputs are located.

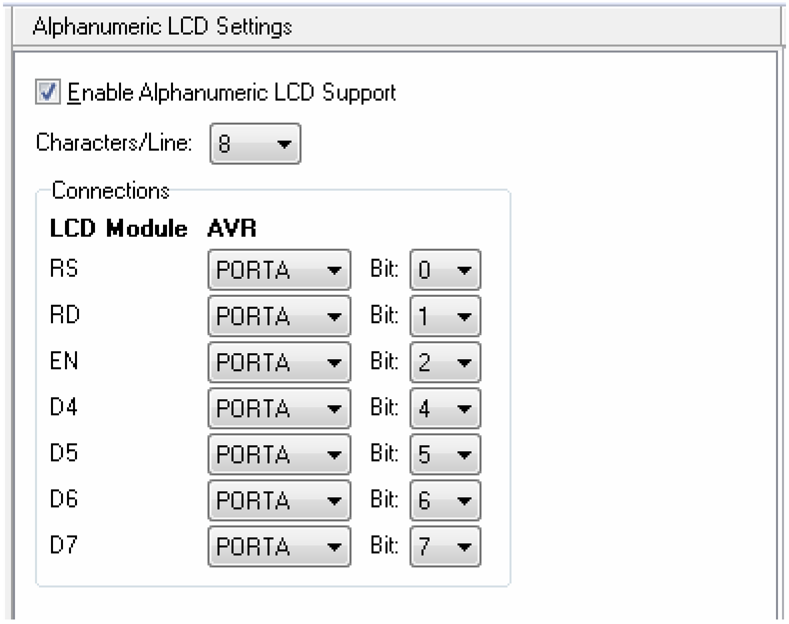
**void dac*n*_write(unsigned char ch, unsigned int data)**

This function will write the **data** value to the output of DAC*n* channel **ch**.
If triggering by the **Event System** is disabled for channel **ch**, the Digital to Analog conversion is automatically started by calling this function.
If triggering is enabled, the conversion will start only after the function is called *and* an event occurs on the corresponding **Event System** channel.

More details about DAC operation for the XMEGA chips can be found in the Atmel **XMEGA A Manual**.

## 7.17 Setting the Alphanumeric LCD

The I/O port allocation for the **LCD Functions for displays with up to 2x40 characters** can be configured by clicking on the **Alphanumeric LCD** node of the CodeWizardAVR selection tree.



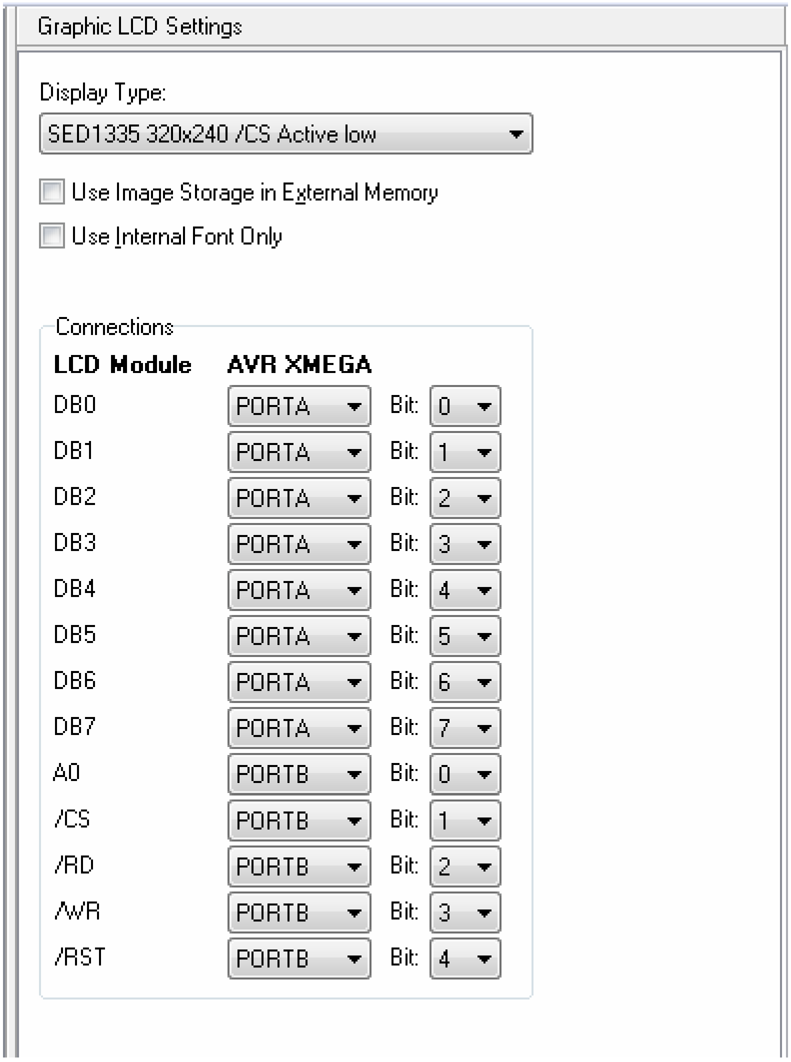

The **Enable Alhanumeric LCD Support** check box activates the configuration specified for the *alcd.h* library functions.

The **Characters/Line** list box allows to specify the number of characters per line supported by the LCD module.

The connections between the LCD module and the AVR I/O ports can be specified individually for each signal in the **Connections** group box.

## 7.18 Setting the Graphic LCD

The I/O port allocation for the **Graphic LCD Functions** can be configured by clicking on the **Graphic LCD** node of the CodeWizardAVR selection tree.

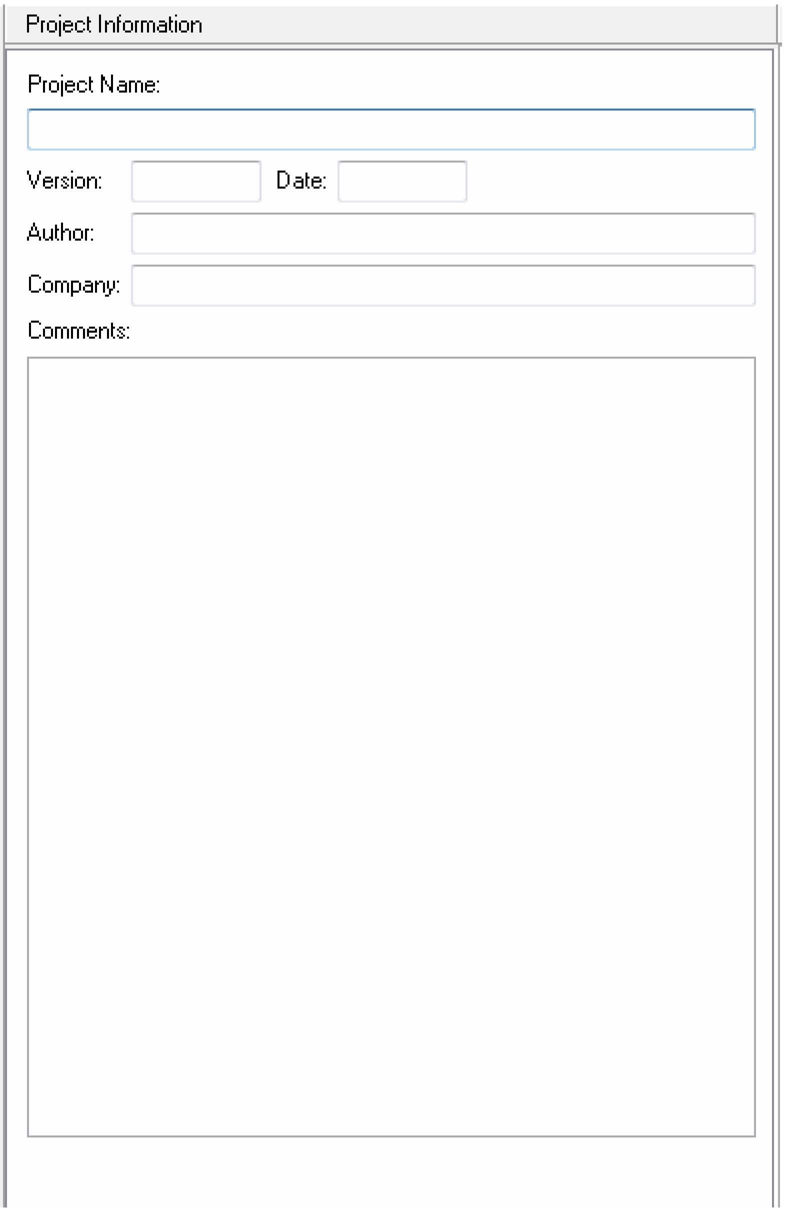



The **Display Type** list box allows to select the graphic controller type and LCD resolution.
The **Use Image Storage in External Memory** check box specifies if additional code will be generated for functions needed to read or write data from external memory, used for graphic image storage.
The **Use Internal Font Only** check box specifies for the **glcd_init** function, that only the internal character generator of the controller is used for displaying text.
**Note:** This option is available only for graphic LCD controllers that have a built-in character generator.

The connections between the graphic LCD module and the AVR I/O ports can be specified individually for each signal in the **Connections** group box.
**Note:** In order to obtain maximum performance, it is advised to set the LCD controller's **data bus** bits to match the bits with the same numbers of the same AVR I/O port.

## 7.19 Specifying the Project Information

The information placed in the comment header, located at the beginning of the C source file produced by CodeWizardAVR, can be specified by clicking on the **Project Information** node of the selection tree.



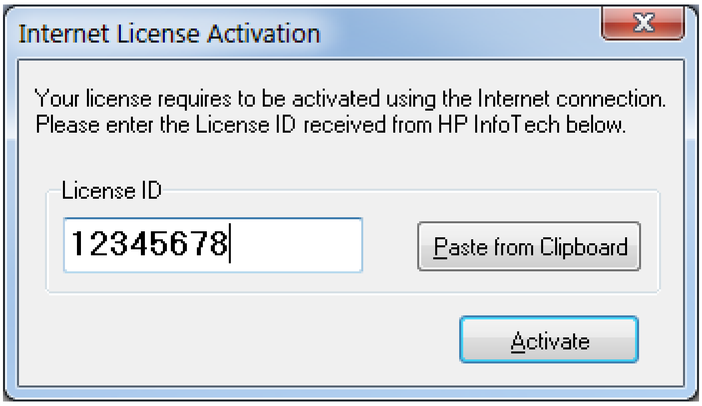

You can specify the **Project Name**, **Date**, **Author**, **Company** and **Comments**.

# 8. Licensing System

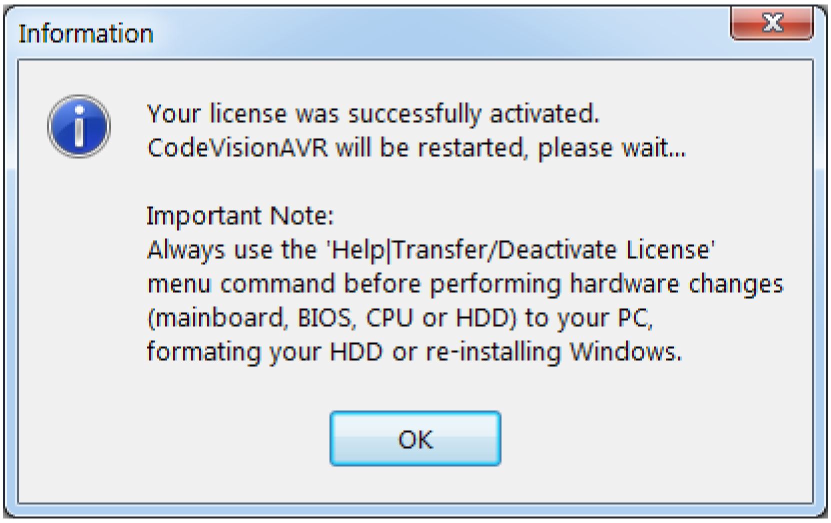
## 8.1 Activating the License

The CodeVisionAVR license is locked to the hardware of the PC on which it is installed.
If the license is not yet activated, the following dialog will be displayed at program startup:




You must enter the **License ID** that was supplied by HP InfoTech when the license was initially purchased.
The **License ID** can be also pasted from the clipboard by pressing the **Paste from Clipboard** button.
After that the license can be activated by pressing the **Activate** button.

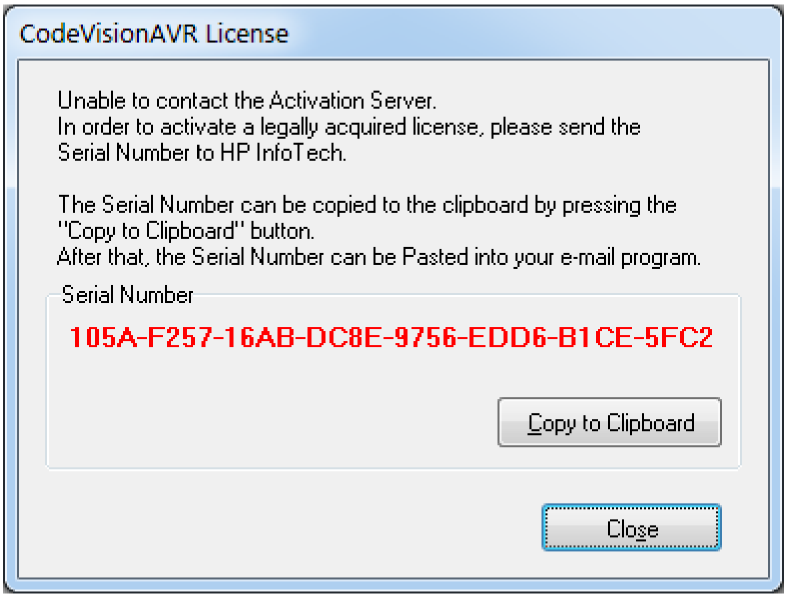
CodeVisionAVR will contact the **Activation Server** using the Internet.
After several seconds, upon successful activation, the following confirmation message will be displayed:




After pressing the **OK** button, CodeVisionAVR will be restarted and ready to use.

If there was an error contacting the **Activation Server**, the license needs to be activated manually by contacting HP InfoTech.
In this case the following message will be displayed with a **Serial Number** specific to the computer on which CodeVisionAVR is installed:




The **Serial Number** can be copied to the clipboard using the **Copy to Clipboard** button and then pasted from the clipboard in the e-mail program.

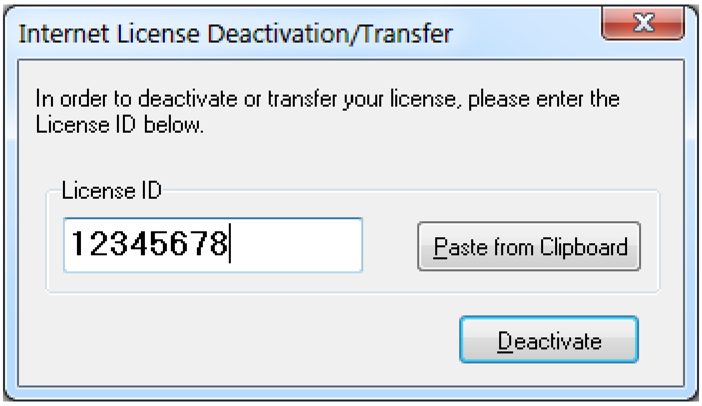
**Note:** If you experience problems contacting the **Activation Server**, please make sure that no firewall or antivirus program are blocking CodeVisionAVR to access the Internet.

## 8.2 Transferring or Deactivating the License

The CodeVisionAVR license is locked to the hardware of the PC on which the software is installed.

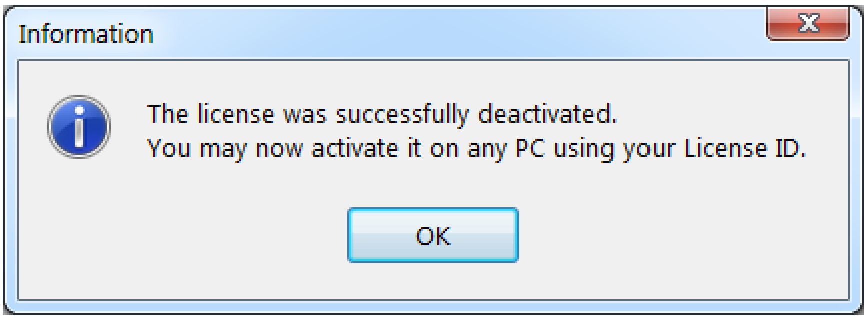
The **Help|Transfer/Deactivate License** menu command must be used in order to transfer the license to another computer.

The following dialog will be displayed:

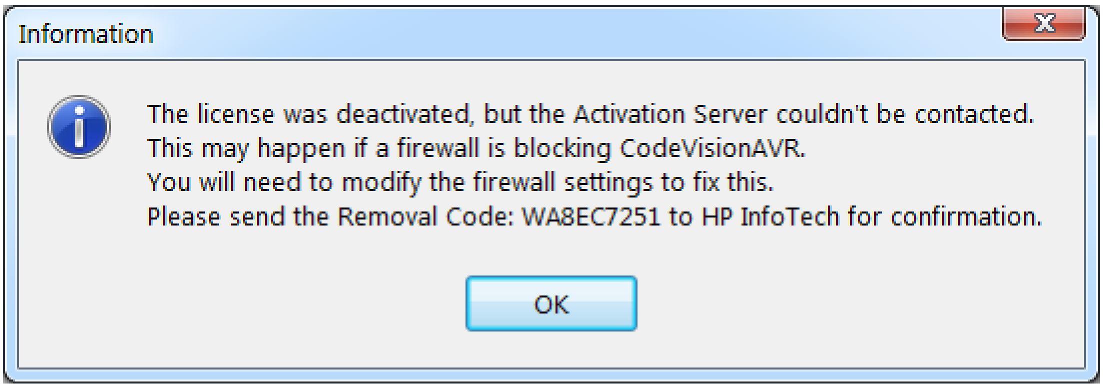



You must enter the **License ID** that was supplied by HP InfoTech when the license was initially purchased.
This is necessary in order to prevent accidental license deactivation by unauthorized persons.
The **License ID** can be also pasted from the clipboard by pressing the **Paste from Clipboard** button.
After that the license can be deactivated by pressing the **Deactivate** button.

CodeVisionAVR will contact the **Activation Server** using the Internet.
After several seconds, upon successful deactivation, the following confirmation message will be displayed:




If the **Activation Server** could not be contacted, the following message will be displayed:



The **Removal Code** must be sent by e-mail to HP InfoTech in order to confirm that the license was indeed deactivated.

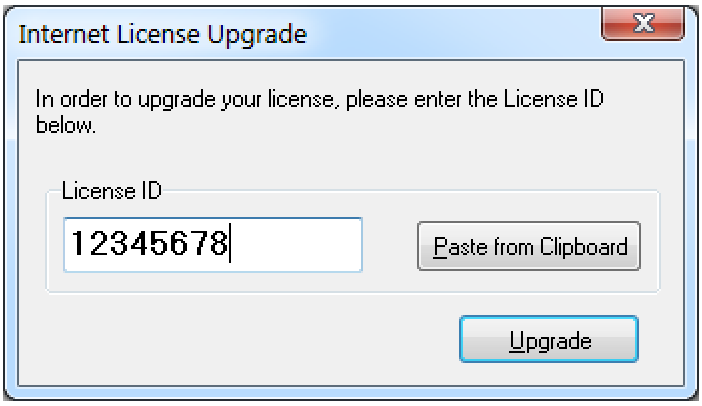
**Note:** If you experience problems contacting the **Activation Server**, please make sure that no firewall or antivirus program are blocking CodeVisionAVR to access the Internet.

If you wish to perform hardware changes to your computer (mainboard, BIOS or HDD) or format the HDD and reinstall Windows, you need to temporary deactivate the license using the above mentioned procedure, and once the changes are done, activate the license again as described in Chapter 7.1.

## 8.3 Upgrading the License

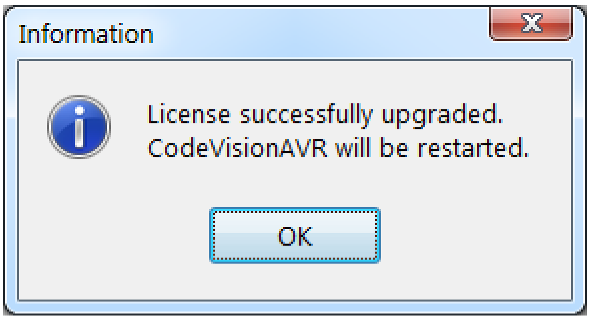
The **Help|Upgrade License** menu command must be used if you have purchased a license upgrade or update extension.

The following dialog will be displayed:

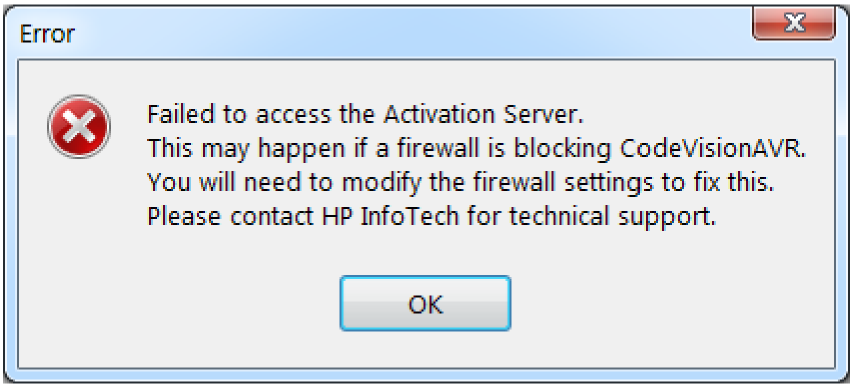



You must enter the **License ID** that was supplied by HP InfoTech when the license was initially purchased.
This is necessary in order to prevent accidental license upgrade by unauthorized persons.
The **License ID** can be also pasted from the clipboard by pressing the **Paste from Clipboard** button.
After that the license can be upgraded by pressing the **Upgrade** button.

CodeVisionAVR will contact the **Activation Server** using the Internet.
After several seconds, upon successful upgrade, the following confirmation message will be displayed:



If there was an error contacting the **Activation Server**, the following error message will be displayed:




In this situation you will need to contact HP InfoTech in order to manually upgrade your license.

**Note:** If you experience problems contacting the **Activation Server**, please make sure that no firewall or antivirus program are blocking CodeVisionAVR to access the Internet.

# 9. License Agreement

## 9.1 Software License

The use of CodeVisionAVR indicates your understanding and acceptance of the following terms and conditions. This license shall supersede any verbal or prior verbal or written, statement or agreement to the contrary. If you do not understand or accept these terms, or your local regulations prohibit "after sale" license agreements or limited disclaimers, you must cease and desist using this product immediately.

This product is © Copyright 1998-2013 by Pavel Haiduc and HP InfoTech S.R.L., all rights reserved. International copyright laws, international treaties and all other applicable national or international laws protect this product. This software product and documentation may not, in whole or in part, be copied, photocopied, translated, or reduced to any electronic medium or machine readable form, without prior consent in writing, from HP InfoTech S.R.L. and according to all applicable laws.
The sole owners of this product are Pavel Haiduc and HP InfoTech S.R.L.

## 9.2 Liability Disclaimer

This product and/or license is provided as is, without any representation or warranty of any kind, either express or implied, including without limitation any representations or endorsements regarding the use of, the results of, or performance of the product, its appropriateness, accuracy, reliability, or correctness.
The user and/or licensee assume the entire risk as to the use of this product.
Pavel Haiduc and HP InfoTech S.R.L. do not assume liability for the use of this program beyond the original purchase price of the software. In no event will Pavel Haiduc or HP InfoTech S.R.L. be liable for additional direct or indirect damages including any lost profits, lost savings, or other incidental or consequential damages arising from any defects, or the use or inability to use these programs, even if Pavel Haiduc or HP InfoTech S.R.L. have been advised of the possibility of such damages.

## 9.3 Restrictions

You may not use, copy, modify, translate, or transfer the programs, documentation, or any copy except as expressly defined in this agreement. You may not attempt to unlock or bypass any "copy-protection" or authentication algorithm utilized by the program. You may not remove or modify any copyright notice or the method by which it may be invoked.

## 9.4 Operating License

You have the non-exclusive right to use the program only by a single person, on a single computer at a time. You may physically transfer the program from one computer to another, provided that the program is used only by a single person, on a single computer at a
time. In-group projects where multiple persons will use the program, you must purchase an individual license for each member of the group.
Use over a "local area network" (within the same locale) is permitted provided that only a single person, on a single computer uses the program at a time. Use over a "wide area network" (outside the same locale) is strictly prohibited under any and all circumstances.

## 9.5 Back-up and Transfer

You may make one copy of the program solely for "back-up" purposes, as prescribed by international copyright laws. You must reproduce and include the copyright notice on the back-up copy. You may transfer the product to another party only if the other party agrees to the terms and conditions of this agreement, and completes and returns registration information (name, address, etc.) to Pavel Haiduc and HP InfoTech S.R.L. within 30 days of the transfer. If you transfer the program you must at the same time transfer the documentation and back-up copy, or transfer the documentation and destroy the back-up copy. You may not retain any portion of the program, in any form, under any circumstance.

## 9.6 Terms

This license is effective until terminated. You may terminate it by destroying the program, the documentation and copies thereof. This license will also terminate if you fail to comply with any terms or conditions of this agreement. You agree upon such termination to destroy all copies of the program and of the documentation, or return them to Pavel Haiduc or HP InfoTech S.R.L. for disposal. Note that by registering this product you give Pavel Haiduc and HP InfoTech S.R.L. permission to reference your name in product advertisements.

## 9.7 Other Rights and Restrictions

All other rights and restrictions not specifically granted in this license are reserved by Pavel Haiduc and HP InfoTech S.R.L.

# 10. Technical Support and Updates

Registered users of commercial versions of CodeVisionAVR receive one-year of free updates and technical support starting from the date of license purchase.

The technical support is provided by e-mail in English or French languages.
The e-mail support address is: **office@hpinfotech.com**

## 11. Contact Information

HP InfoTech S.R.L. can be contacted at:

HP INFOTECH S.R.L.
BD. DECEBAL NR. 3
BL. S12B, SC. 2, AP. 29
SECTOR 3
BUCHAREST
ROMANIA

Phone: +(40)-723469754
Fax: +(40)-213261876

e-mail: office@hpinfotech.com
Internet: http://www.hpinfotech.com
http://www.hpinfotech.biz
http://www.hpinfotech.ro