

# **CrossWorks for ARM**

## **User Guide**

Rowley Associates Limited





# Contents

## Part I

## Preface

### Chapter 1

#### **3 Introduction**

- 4 What is CrossWorks?
- 5 What we don't tell you
- 6 Activating your product
- 7 Text conventions
- 8 Standard syntactic metalanguage
- 9 Requesting support and reporting problems

## Part II

## CrossStudio Reference

### Chapter 2

#### **13 CrossStudio Tutorial**

- 14 Creating a project
- 16 Managing files in a project
- 19 Setting project options
- 21 Building projects
- 22 Exploring projects
- 27 Using the debugger
- 30 Low-level debugging

### Chapter 3

#### **35 Overview**

- 35 CrossStudio standard layout
- 36 The title bar
- 37 The menu bar

	38	The status bar
	40	The editing workspace
<b>Chapter 4</b>	<b>43</b>	<b>Project management</b>
	44	Project system
	46	Creating a project
	47	Adding existing files to a project
	48	Adding new files to a project
	48	Removing a file, folder, project, or project link
	49	Project properties
	52	Project configurations
	53	Project dependencies and build order
	54	Project macros
<b>Chapter 5</b>	<b>57</b>	<b>Building projects</b>
	57	Build configurations and their uses
	58	Building your applications
	60	Correcting errors after building
<b>Chapter 6</b>	<b>61</b>	<b>Source code control</b>
	61	Configuring source control
	62	Using source control
<b>Chapter 7</b>	<b>65</b>	<b>Debug expressions</b>
<b>Chapter 8</b>	<b>67</b>	<b>Basic editing</b>
	67	Elements of the code editor
	68	Navigation
	70	Bookmarks
	72	Changing text
	74	Using the clipboard
	75	Drag and drop editing
	76	Undo and redo
	77	Indentation
	79	File management
	81	Find and replace
	83	Regular expressions
	85	Advanced editor features
	88	Code templates
<b>Chapter 9</b>	<b>89</b>	<b>Memory map editor</b>
<b>Chapter 10</b>	<b>93</b>	<b>Package management</b>

<b>Chapter 11</b>	<b>95</b>	<b>Linking and Section placement</b>
<b>Chapter 12</b>	<b>99</b>	<b>CrossStudio Windows</b>
	100	Clipboard ring window
	102	Build log window
	102	Breakpoints window
	107	Call stack window
	111	Execution counts window
	112	Globals window
	114	Locals window
	116	Memory window
	118	Register windows
	120	Threads window
	122	Trace window
	122	Watch window
	125	Help window
	127	Output window
	128	Project explorer
	130	Properties window
	131	Source navigator window
	133	Symbol browser
	139	Targets window
<b>Chapter 13</b>	<b>145</b>	<b>Target interfaces</b>
	146	Macraigor Wiggler (20 and 14 pin) Target interface
	148	CrossConnect Target interface
	149	Segger J-Link Target interface
	151	ARM Simulator Target interface
<b>Chapter 14</b>	<b>153</b>	<b>ARM Target Support</b>
	154	Target startup code
	155	Startup code
	157	Memory map files
	158	Project configurations
	159	Target script file
	166	Program loading
	167	ARM Device Specific Target Support
<b>Chapter 15</b>	<b>169</b>	<b>Dialogs</b>
	169	Debug file search editor
	170	Environment options

<b>Chapter 16</b>	<b>173</b>	<b>CrossStudio menu summary</b>
	173	File menu
	175	New menu
	176	Edit menu
	177	Clipboard menu
	179	Clipboard Ring menu
	179	Macros menu
	180	Edit Selection menu
	182	Bookmarks menu
	183	Advanced menu
	183	View menu
	185	Other Windows menu
	186	Browser menu
	187	Toolbars menu
	188	Search menu
	189	Project menu
	190	Build menu
	192	Debug menu
	194	Debug Control menu
	195	Breakpoint menu
	196	Debug Windows menu
	198	Tools menu
	199	Window menu
	200	Help menu

## Part III Tasking Library Tutorial

<b>Chapter 17</b>	<b>205</b>	<b>Overview</b>
	205	Tasks
	206	Task synchronization and resource allocation
	207	Timer support
	207	Interrupt service routine support
	207	Memory block allocation support
	207	C library support
<b>Chapter 18</b>	<b>209</b>	<b>Tasks</b>
<b>Chapter 19</b>	<b>213</b>	<b>Event sets</b>
<b>Chapter 20</b>	<b>217</b>	<b>Semaphores</b>
<b>Chapter 21</b>	<b>221</b>	<b>Message queues</b>

<b>Chapter 22</b>	<b>225</b>	<b>Byte queues</b>
<b>Chapter 23</b>	<b>229</b>	<b>Global interrupts control</b>
<b>Chapter 24</b>	<b>231</b>	<b>Timer support</b>
<b>Chapter 25</b>	<b>233</b>	<b>Programmable interrupt handling</b>
<b>Chapter 26</b>	<b>235</b>	<b>Low-level interrupt handling</b>
<b>Chapter 27</b>	<b>237</b>	<b>Memory areas</b>

## Part IV Library Reference

<b>Chapter 28</b>	<b>241</b>	<b>Standard C Library Reference</b>
	242	<assert.h> - Diagnostics
	243	<ctype.h> - Character handling
	247	<errno.h> - Errors
	248	<limits.h> - Integer numerical limits
	252	<math.h> - Mathematics
	276	<setjmp.h> - Non-local jumps
	278	<stdarg.h> - Variable arguments
	280	<stdio.h> - Input/output functions
	295	<stdlib.h> - General utilities
	310	<string.h> - String handling

## Part V Command Line Tools

<b>Chapter 29</b>	<b>321</b>	<b>CrossBuild</b>
<b>Chapter 30</b>	<b>323</b>	<b>CrossLoad</b>
<b>Chapter 31</b>	<b>327</b>	<b>mkhdr - C/C++ header file generator</b>
<b>Chapter 32</b>	<b>329</b>	<b>mkpkg - package creator</b>
<b>Chapter 33</b>	<b>331</b>	<b>GCC Online Documentation</b>
<b>Chapter 34</b>	<b>333</b>	<b>BINUTILS Online Documentation</b>

<b>Chapter 35</b>	<b>337</b>	<b>File formats</b>
	337	Memory Map file format
	339	Section Placement file format
	340	Project file format
	341	Project Templates file format
<b>Chapter 36</b>	<b>345</b>	<b>Property Reference</b>
	345	Build Properties
	350	Compilation Properties
	362	External Build Project Properties
	365	Folder Properties
	366	Library Properties
	367	Linker Properties
	379	Staging And Combining Project Properties
	383	Target Interface Properties
	397	ARM Simulator Properties
<b>Chapter 37</b>	<b>399</b>	<b>Code editor command summary</b>
<b>Chapter 38</b>	<b>405</b>	<b>Binary editor command summary</b>
<b>Chapter 39</b>	<b>407</b>	<b>Glossary</b>



# Part I

---

## Preface

# Preface



# Introduction

This guide is divided into a number of sections:

- **Introduction (page 3).** Covers installing CrossWorks on your machine and verifying that it operates correctly, followed by a brief guide to the operation of the CrossStudio integrated development environment, debugger, and other software supplied in the CrossWorks package.
- **CrossStudio Tutorial (page 13).** Describes how to get started with CrossStudio and runs through all the steps from creating a project to debugging it on hardware.
- **CrossStudio Reference (page 11).** Contains information on how to use the CrossStudio development environment to manage your projects, build, and debug your applications.
- **Tasking Library Tutorial (page 203).** Contains documentation on using the CrossWorks tasking library to write multi-threaded applications.
- **ARM Library Reference.** Contains documentation for the functions that are specific to the ARM.
- **Standard C Library Reference (page 241).** Contains documentation for the functions in the standard C library supplied in the package.
- **ARM Target Support (page 153).** Contains a description of system files used for startup and debugging of ARM applications.
- **Target interfaces (page 145).** Contains a description of the support for programming ARM microcontrollers.

# What is CrossWorks?

CrossWorks for ARM is a complete C development system for ARM7 microprocessors. It comprises of the ARM GCC C compiler, the CrossWorks C Library and the CrossStudio integrated development environment.

In order to use CrossWorks for ARM you will need:

- Windows 98, Windows Me, Windows NT 4.0, Windows 2000 or Windows XP.
- A Macgraigor Wiggler for ARM (WNPJ-ARM-20/WNPJ-ARM-14) or compatible parallel port to JTAG interface.
- An ARM 7 target board with 20 or 14 pin JTAG connector. CrossWorks for ARM provides support for several ARM based microcontrollers out of the box in the form of examples and target configurations. CrossWorks can also be easily modified to support other ARM 7 targets, see **ARM Target Support** (page 153) for more information.

## GNU Compiler Collection

CrossWorks for ARM comes with a pre-built version of the GNU Compiler Collection (GCC) C and C++ compiler, assembler, linker and other tools to enable you to immediately begin developing applications for ARM.

## CrossWorks C Library

CrossWorks for ARM has it's own royalty-free ANSI and ISO C compliant C library that has been specifically designed for use within embedded systems.

## CrossStudio IDE

CrossStudio for ARM is a streamlined integrated development environment (IDE) for building, testing, and deploying ARM applications. CrossStudio provides:

- **Source Code Editor** A powerful source code editor with multi-level undo and redo, makes editing your code a breeze.
- **Project System** A complete project system organises your source code and build rules.
- **Build System** With a single key press you can build all your applications in a solution, ready for them to be loaded onto a developer card or into the debugger.
- **ARM Hardware Debug** With the Macgraigor Wiggler attached, you can use the integrated debugger to step through and diagnose problems in your software on your target board.

- **Integrated Debugger** The debugger will help you to quickly find problems in your ARM and Thumb applications.
- **ARM Flash Programming and Debug** You can download your programs directly into Flash and debug them seamlessly from within the IDE.
- **Integrated Help system** The built-in help system provides context-sensitive help and a complete reference to the CrossStudio IDE and tools.

## What we don't tell you

This documentation does not attempt to teach the C or assembly language programming; rather, you should seek out one of the many introductory texts available. And similarly the documentation doesn't cover the ARM architecture or microcontroller application development in any great depth.

We also assume that you're fairly familiar with the operating system of the host computer being used. For Microsoft Windows development environment we recommend Windows 2000 or Windows XP, but you can use Windows NT 4, Windows 95, Windows 98, or Windows Me if you wish.

### C programming guides

- Kernighan, B.W. and Ritchie, D.M., *The C Programming Language* (2nd edition, 1988). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-110362-8. The original C bible, updated to cover the essentials of ANCI C (1990 version).
- Harbison, S.P. and Steele, G.L., *A C Reference Manual* (second edition, 1987). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-109802-0. A nice reference guide to C, including a useful amount of information on ANSI C. Written by Guy Steele, a noted language expert.

### ANSI C reference

- ISO/IEC 9899:1990, C Standard and ISO/IEC 9899:1999, C Standard. The standard is available from your national standards body or directly from ISO at [www.iso.ch](http://www.iso.ch).

### ARM microcontrollers

- ARM technical reference manuals, specifications, user guides and white papers for can be found at <http://www.arm.com/Documentation/>.

## GNU compiler collection

- The latest GCC documentation, news and downloads can be found at <http://gcc.gnu.org/>.

## Activating your product

Each copy of CrossWorks must be licensed and registered before it can be used. Each time you purchase a CrossWorks license, you, as a single user, can use CrossWorks on the computers you need to develop and deploy your application. This covers the usual scenario of using both a laptop and desktop and, optionally, a laboratory computer.

### Evaluating CrossWorks

If you are evaluating CrossWorks on your computer, you must activate it. To activate your software for evaluation, follow these instructions:

- Install CrossWorks on your computer using the CrossWorks installer and accept the license agreement.
- Run the **CrossStudio** application.
- From the **Tools** menu, click **License Manager**.
- If you have a default mailer, click the **By Mail** button underneath the text "If you wish to evaluate CrossWorks...".
- Using e-mail, send the registration key to the e-mail address **license@rowley.co.uk**.
- If you do not have a default mailer, click the **Manually** button underneath the text "If you wish to evaluate CrossWorks...".
- CrossStudio copies a registration key onto the clipboard. Send the registration key to the e-mail address **license@rowley.co.uk**.

By return you will receive an **activation key**. To activate CrossWorks for evaluation, do the following::

- Run the **CrossStudio** application.
- From the **Tools** menu, click **License Manager**.
- Click the **Activate Product** button.
- Type in or paste the returned activation key into the dialog and click **OK**.

If you need more time to evaluate CrossWorks, simply request a new evaluation key when the issued one expires or is about to expire.

### After purchasing CrossWorks

When you purchase CrossStudio, either directly from ourselves or through a distributor, you will be issued a Product Key which uniquely identifies your purchase. To permanently activate your software, follow these instructions:

- Install CrossWorks on your computer using the CrossWorks installer and accept the license agreement.
- Run the **CrossStudio** application.
- From the **Tools** menu, click **License Manager**.
- If you have a default mailer, click the **By Mail** button underneath the text "If you have a product key...".
- Using e-mail, send the registration key to the e-mail address **license@rowley.co.uk**.
- If you do have a default mailer, click the **Manually** button underneath the text "If you have a product key...".
- CrossStudio copies a registration key onto the clipboard. Send the registration key to the e-mail address **license@rowley.co.uk**.

By return you will receive an **activation key**. To activate CrossWorks:

- Run the **CrossStudio** application.
- From the **Tools** menu, click **License Manager**.
- Click the **Activate Product** button.
- Type in or paste the returned activation key into the dialog and click **OK**.

As CrossWorks is licensed per developer, you can install the software on any computer that you use such as a desktop, laptop, and laboratory computer, but on each of these you must go through activation using your issued product key.

## Text conventions

Throughout the documentation, text printed **in this typeface** represents verbatim communication with the computer: for example, pieces of C text, commands to the operating system, or responses from the computer. In examples, text printed *in this typeface* is not to be used verbatim: it represents a class of items, one of which should be used. For example, this is the format of one kind of compilation command:

**hcl** *source-file*

This means that the command consists of:

- The word **hcl**, typed exactly like that.
- A *source-file*: not the text **source-file**, but an item of the *source-file* class, for example **'myprog.c'**.

Whenever commands to and responses from the computer are mixed in the same example, the commands (i.e. the items which you enter) will be presented in this typeface. For example, here is a dialogue with the computer using the format of the compilation command given above:

```
c:\crossworks\examples>hcl -v myprog.c
CrossWorks MSP430 Compiler Driver   Release 1.0.0
Copyright (c) 1997-2004 Rowley Associates Ltd.
```

The user types the text **hcl -v myprog.c**, and then presses the enter key (which is assumed and is not shown); the computer responds with the rest.

## Standard syntactic metalanguage

In a formal description of a computer language, it is often convenient to use a more precise language than English. This language-description language is referred to as a *metalanguage*. The metalanguage which will be used to describe the C language is that specified by British Standard 6154. A tutorial introduction to the standard syntactic metalanguage is available from the National Physical Laboratory.

The BS6154 standard syntactic metalanguage is similar in concept to many other metalanguages, particularly those of the well-known Backus-Naur family. It therefore suffices to give a very brief informal description here of the main points of BS6154; for more detail, the standard itself should be consulted.

- Terminal strings of the language—those built up by rules of the language—are enclosed in quotation marks.
- Non-terminal phrases are identified by names, which may consist of several words.
- When numbers are used in the text they will usually be decimal. When we wish to make clear the base of a number, the base is used as a subscript, for example  $15_8$  is the number 15 in base eight and 13 in decimal,  $2F_{16}$  is the number 2F in hexadecimal and 47 in decimal.
- A sequence of items may be built up by connecting the components with commas.
- Alternatives are separated by vertical bars ('|').
- Optional sequences are enclosed in square brackets ('[' and ']').



- Sequences which may be repeated zero or more times are enclosed in braces ('{' and '}').
- Each phrase definition is built up using an equals sign to separate the two sides, and a semicolon to terminate the right hand side.

## Requesting support and reporting problems

With software as complex as CrossWorks, it's almost inevitable that you'll need assistance at some point. Here are some pointers on what to do when you think you've found a problem.

### Requesting help

If you need some help working with CrossWorks, please contact our support department by e-mail, [support@rowley.co.uk](mailto:support@rowley.co.uk).

### Reporting a bug

Should you have a problem with this product which you consider a bug, please report it by e-mail to our support department, [bugs@rowley.co.uk](mailto:bugs@rowley.co.uk).

### Support and suggestions

If you have any comments or suggestions regarding the software or documentation, please send these in an e-mail to [support@rowley.co.uk](mailto:support@rowley.co.uk) or in writing to:

CrossWorks Customer Support  
Rowley Associates Limited  
Suite 4B/4C Drake House  
Drake Lane  
Dursley  
Gloucestershire GL11 4HS  
UNITED KINGDOM

Tel: +44 1453 549536

Fax: +44 1453 544068



# Part II

## CrossStudio Reference

# CrossStudio Reference

This section is a reference to the CrossStudio integrated development environment.

### In this section

- **Overview (page 35).** Contains an overview of CrossStudio and its layout.
- **Project management (page 43).** Describes how to manage files and projects in CrossStudio.
- **Building projects (page 57).** Describes how to build projects and correct errors in them.
- **Source code control (page 61).** Describes how to set up your source code control provider so that it works seamlessly with CrossStudio.
- **Debug expressions (page 65).** Describes the type and form of expression that CrossStudio can evaluate when debugging an application.
- **Basic editing (page 67).** Describes CrossStudio's integrated code editor and how to get the most from it.
- **Memory map editor (page 89).** Describes how to edit memory map files that guide section placement.
- **Linking and Section placement (page 95).** Describes how your project is partitioned and placed into the target device's memory.

- **CrossStudio Windows (page 99).** **CrossStudio Windows** (page 99) Describes each of CrossStudio's window individually.
- **CrossStudio menu summary (page 173).** Summarizes each of the menus presented in CrossStudio.

# CrossStudio Tutorial

CrossStudio allows you to organize your collection of projects into a workspace or *solution*. We provide a number of project templates for popular evaluation and demonstration boards with the product which you can use as a springboard to start your application development. A project is typically organized into groups, where each group gathers together files that are related—for example, header files, source files, and documentation files can all have their own group in a project.

This section will take you through creating, compiling, and debugging a simple application using the build-in simulator to prepare you for starting your own projects using CrossStudio.

## In this section

- **Creating a project (page 14).** Describes how to start a project, select your target processor, and other common options.
- **Managing files in a project (page 16).** Describes how to add existing and new files to a project and how to remove items from a project.
- **Setting project options (page 19).** Describes how to set options on project items and how project option inheritance works.
- **Building projects (page 21).** Describes how to build the project, correct compilation and linkage errors, and find out how big your applications are.
- **Exploring projects (page 22).** Describes how to use the Project Explorer, Symbol Browser, and Source Navigator to find out how much memory your project takes and navigate around the files that make up the project. It also describes the similarities and differences between the three windows.
- **Using the debugger (page 27).** Describes the debugger and how to find and fix problems at a high level when executing your application.

- **Low-level debugging (page 30).** Describes how to use debugger features to debug your program at the machine level by watching registers and tracing instructions.

## Creating a project

To start developing an application, you create a new project. To create a new project, do the following:

- From the **File** menu, click **New** then **New Project...**

The **New Project** dialog appears. This dialog displays the set of project types and project templates.

We'll create a project to develop our application in C:

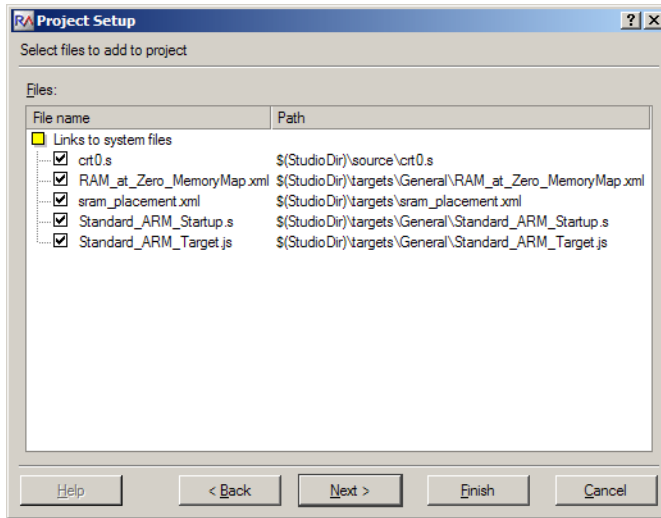
- Click the **Executable** icon in the **Templates** pane which selects the type of project to add.
- Type `Tutorial` in the **Name** edit box, which names the project.
- You can use the **Location** edit box or the **Browse** button to locate where you want the project to be created.
- Click **OK**.

This will create a project for a generic ARM target that has RAM mapped at address 0x00000000, as we are going to run this example on the simulator this is fine. ARM hardware however is rarely so accommodating as memory will be mapped at different addresses, target specific startup code may be required to initialize peripherals, different techniques need to be employed to reset the target and target specific loader applications are required to program FLASH. To create a project to run on hardware you should instead select a template from the project type matching your target, this will create a project with the memory maps, startup code, reset script and FLASH loader for your target.

Once created, the project setup wizard prompts you to set some common options for the project.

Here you can specify an additional file format to be output when the application is linked, and what library support to include if you use **printf** and **scanf**. You can change these settings after the project is created using the Project Explorer.

Clicking **Next** displays the files that will be added to the project.



The **Links to system files** group shows the links that will be created in the project to CrossStudio system files. Project links are fully explained in **Project management** (page 43), and we can ignore these for now.

Clicking **Next** displays the configurations that will be added to the project.

Here you can specify the default configurations that will be added to the project. Project configurations are fully explained in **Project management** (page 43), and we can ignore these for now.

Complete the project creation by clicking **Finish**.

The **Project Explorer** shows the overall structure of your project. To see the project explorer, do one of the following:

- From the **View** menu, click **Project Explorer**.

or

- Type **Ctrl+Alt+P**.

or

- Right click the tool bar area.
- From the popup menu, select **Project Explorer**.

This is what our project looks like in the Project Explorer:

You'll notice that the project name is shown in bold which indicates that it is the active project (and in our case, the only project). If you have more than one project then you can set the active project using the dropdown box on the build tool bar or the context menu of the project explorer.

The files are arranged into two groups:

- **Source Files** contains the main source files for your application which will typically be header files, C files, and assembly code files. You may want to add files with other extensions or documentation files in HTML format, for instance.
- **System Files** contains links to source files that are not part of the project yet are required when the project is built and run. In this case, the system files are `crt0.s` which is the C runtime startup written in assembly code, `RAM_at_Zero_MemoryMap.xml` a memory map file that describes a target with RAM located at address 0x00000000, `sram_placement.xml` which directs the linker on how to arrange program sections in memory, `Standard_ARM_Startup.s` which contains the target specific start code and exception vectors and `Standard_ARM_Target.js` which contains the target specific target script which instructs the debugger on how to reset the target and what to do when the processor stops or starts. Files which are stored outside of the projects home directory are shown by a small purple shortcut indicator at the bottom left of the icon, as above.

These folders have nothing to do with directories on disk, they are simply a means to group related files together in the project explorer. You can create new folders and specify filters based on the file extension so that when you add a new file to the project it will be placed in the folder whose filter matches the file extension.

## Managing files in a project

We'll now set up the project with some files that demonstrate features of the CrossStudio IDE. For this, we will add one pre-prepared and one new file to the project.

### Adding an existing file to a project

We will add one of the tutorial files to the project. To add an existing file to the project, do the following:

- From the **File** menu, click **Add Existing File**.

or

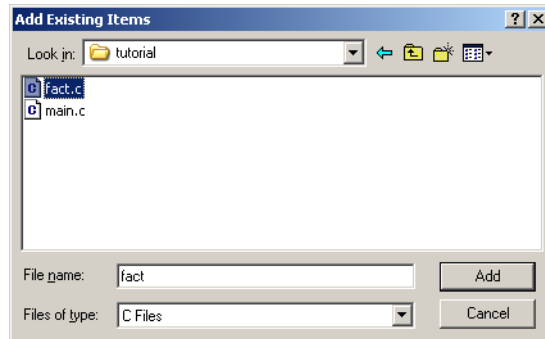
- Type **Ctrl+D**.

or



- In the **Project Explorer**, right click the **Tutorial** project node.
- Select **Add Existing File** from the context menu.

When you've done this, CrossStudio displays a standard file locator dialog. Navigate to the CrossStudio installation directory, then to the `tutorial` folder, select the `fact.c`.



Now click **OK** to add the file to the project. The Project Explorer will show `fact.c` with a shortcut arrow because the file is not in the project's home directory. Rather than edit the file in the tutorial directory, we'll take a copy of it and put it into the project home directory:

- In the **Project Explorer**, right click the `fact.c` node.
- From the popup menu, click **Import**.

The shortcut arrow disappears from the `fact.c` node which indicates that the file is now in our home directory.

We can open a file for editing by double clicking the node in the Project Explorer. Double clicking `fact.c` brings it into the code editor:

```

Help | main.c | fact.c
// CrossWorks Tutorial

int fact(int n)
{
    if (n <= 1)
        return 1;
    else
        return fact(n-1) * n;
}

```

### Adding a new file to a project

Our project isn't complete as `fact.c` is only part of an application. We'll add a new C file to the project which will contain the `main()` function. To add a new file to the project, do the following:

- From the **Project** menu, click **New File**.

or

- On the **Project Explorer** tool bar, click the **Add New File** tool button.

or

- In the **Project Explorer**, right click the `Tutorial` node.
- From the context menu, click **Add New File**.

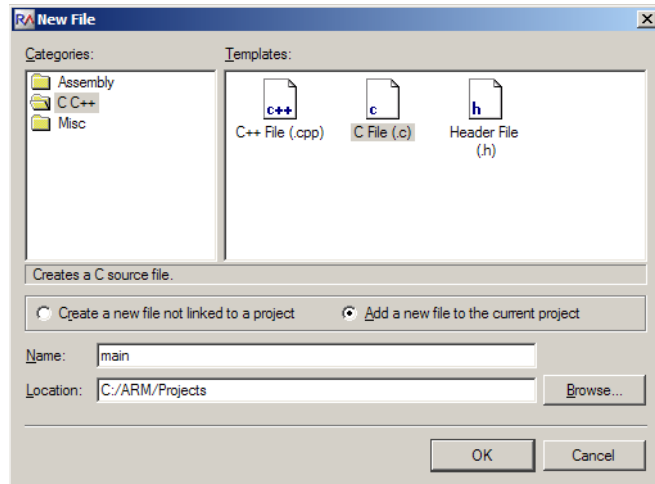
or

- Type **Ctrl+N**.

The **New File** dialog appears.

- Ensure that the **C File (.c)** icon is selected.
- In the **Name** edit box, type `main`.

The dialog box will now look like this:



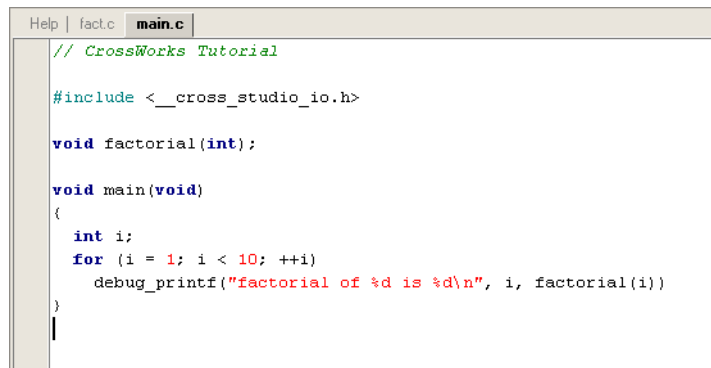
Click **OK** to add the new file.

CrossStudio opens an editor with the new file ready for editing. Rather than type in the program from scratch, we'll add it from a file stored on disk.

- From the **Edit** menu, click **Insert File** or type **Ctrl+K, Ctrl+I**.
- Using the file browser, navigate to the `tutorial` directory.

- Select the `main.c` file.
- Click **OK**.

Your `main.c` file should now look like this:



```
Help | fact.c | main.c
// CrossWorks Tutorial

#include <__cross_studio_io.h>

void factorial(int);

void main(void)
{
    int i;
    for (i = 1; i < 10; ++i)
        debug_printf("factorial of %d is %d\n", i, factorial(i))
}
```

Next, we'll set up some project options.

## Setting project options

You have now created a simple project, and in this section we will set some options for the project.

You can set project options on any node of the solution. That is, you can set options on a solution-wide basis, on a project-wide basis, on a project group basis, or on an individual file basis. For instance, options that you set on a solution are inherited by all projects in that solution, by all groups in each of those projects, and then by all files in each of those groups. If you set an option further down in the hierarchy, that setting will be inherited by nodes that are children of (or grandchildren of) that node. The way that options are inherited provides a very powerful way to customize and manage your projects.

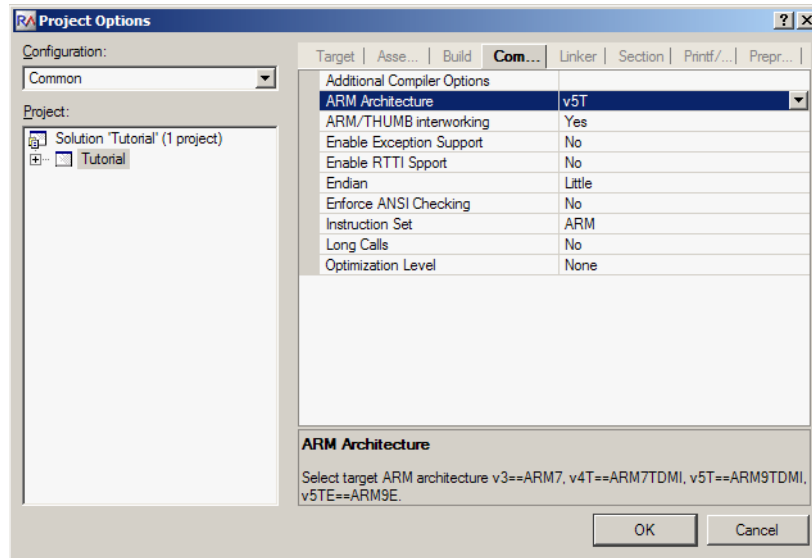
### Changing the ARM architecture

In this instance, we will set up the targeted ARM architecture to be v5T. As we will be running the example on the simulator it doesn't matter which architecture we target as the simulator will simulate the architecture specified in the project. To change the targeted ARM architecture:

- Right click the **Tutorial** project in the Project Explorer and select **Properties** from the menu the **Project Options** dialog appears.
- Click the **Configuration** dropdown and change to the **Common** configuration.

- Click the **Compiler** tab to display the code generation options.
- Click the **ARM Architecture** option and change this from **v4T** to **v5T**.

The dialog box will now look like this:



Notice that when you change between **Debug** and **Release** configurations, the code generation options change. This dialog shows which options are used when building a project (or anything in a project) in a given configuration. Because we have set the target processor in the **Common** configuration, both **Debug** and **Release** configurations will use this setting. We could, however, set the processor type to be different in **Debug** and **Release** configurations, allowing us to develop on a processor with a large amount of code memory and hardware emulation support, but elect to deploy on a smaller, more cost effective variant.

Now click **OK** to accept the changes made to the project.

### Using the Properties Window

If you click on the project node, the **Properties Window** will show the properties of the project these have all been inherited from the solution. If you modify a property when the project node is selected then you'll find that its value is highlighted because you have overridden the property value that was inherited from the solution. You can restore the inherited value of a property by right clicking the property and selecting **Use Inherited Value** from the menu.

Next, we'll build the project.

## Building projects

Now that the project is created and set up, it's time to build it. Unfortunately, there are some deliberate errors in the program which we need to correct.

### Building the project

To build the project, do the following:

- From the **Project** menu, click **Build**.

—or—

- On the **Build** tool bar, click the **Build** tool button.

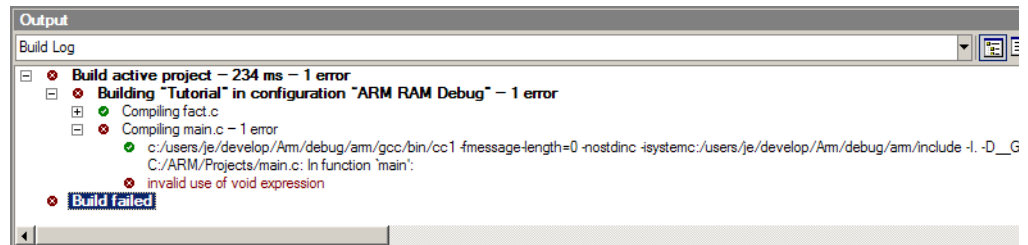
—or—

- Type F7.

Alternatively, to build the **Tutorial** project using a context menu, do the following:

- In the **Project Explorer**, right click the **Tutorial** project node.
- Select **Build** from the context menu.

CrossStudio starts compiling the project files but finishes after detecting an error. The Output Window shows the Build Log which contains the errors found in the project:



### Correcting compilation and linkage errors

CrossStudio compiled `fact.c` without errors, but `main.c` contains two errors. After compilation, CrossStudio moves the cursor to the line containing the first reported error. As well as this, the line is marked in the gutter and highlighted by underlining it red. (You can change this behaviour using the **Environment Options** dialog.)

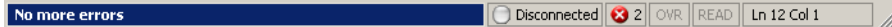
```
int i;
for (i = 1; i < 10; ++i)
    debug_printf("factorial of %d is %d\n", i, factorial(i))
}
```

The status bar also updates to indicate two build errors and shows the first error message.

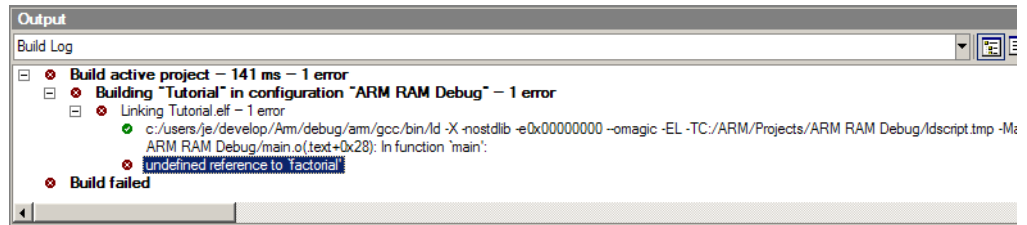


To correct the error, change the return type of `factorial` from `void` to `int` in its prototype.

To move the cursor to the line containing the next error, type **F4** or from the **Search** menu, click **Next Location**. The cursor is now positioned at the `debug_printf` statement which is missing a terminating semicolon add the semicolon to the end of the line. Using **F4** again indicates that we have corrected all errors:



Pressing **F4** again wraps around and moves the cursor to the first error, and you can use **Shift+F4** or **Previous Location** in the **Search** menu to move back through errors. Now that the errors are corrected, compile the project again. The build log still shows that we have a problem.



Notice that `fact.c` has not been recompiled because it was compiled correctly before and is up to date. The remaining error is a linkage error. Double click on `fact.c` in the Project Explorer to open it for editing and change the two occurrences of `fact` to `factorial`. Recompile the project—this time, the project compiles correctly:

## Exploring projects

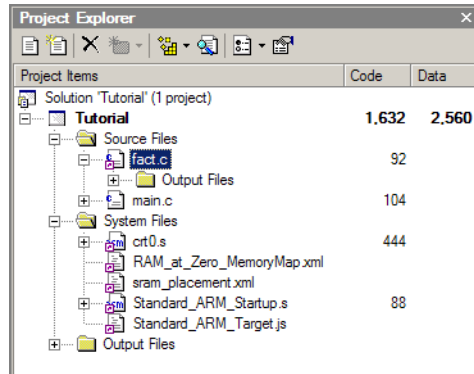
Now that the project has no errors and builds correctly, we can turn our attention to uncovering exactly how our application fits in memory and how to navigate around it.

### Using Project Explorer features

The Project Explorer is the central focus for arranging your source code into projects, and it's a good place to show ancillary information gathered when CrossStudio builds your applications. This section will cover the features that the Project Explorer offers to give you an overview of your project.

## Project code and data sizes

Developers are always interested in how much memory their applications take up, and with small embedded microcontrollers this is especially true. The Project Explorer can display the code and data sizes for each project and individual source file that is successfully compiled. To do this, click the **Options** dropdown on the **Project Explorer** tool bar and make sure that **Show Code/Data Size** is checked. Once checked, the Project Explorer displays two additional columns, **Code** and **Data**.



The screenshot shows the Project Explorer window with a table of project items. The table has three columns: 'Project Items', 'Code', and 'Data'. The 'Tutorial' project is expanded to show its sub-items.

Project Items	Code	Data
Solution 'Tutorial' (1 project)	1.632	2.560
Tutorial		
Source Files		
fact.c	92	
Output Files		
main.c	104	
System Files		
crt0.s		444
RAM_at_Zero_MemoryMap.xml		
sram_placement.xml		
Standard_ARM_Startup.s	88	
Standard_ARM_Target.js		
Output Files		

The **Code** column displays the total code space required for the project and the **Data** column displays the total data space required. The code and data sizes for each C and assembly source file are *estimates*, but good estimates nonetheless. Because the linker removes any unreferenced code and data and performs a number of optimizations, the sizes for the linked project may not be the sum of the sizes of each individual file. The code and data sizes for the project, however, *are* accurate. As before, your numbers may not match these exactly.

## Dependencies

The Project Explorer is very versatile: not only can you display the code and data sizes for each element of a project and the project as a whole, you can also configure the Project Explorer to show the *dependencies* for a file. As part of the compilation process, CrossStudio finds and records the relationships between files that is, it finds which files are dependent upon other files. CrossStudio uses these relationships when it comes to build the project again so that it does the minimum amount of work to bring the project up to date.

To show the dependencies for a project, click the **Options** button on the **Project Explorer** tool bar and ensure that **Show Dependencies** is checked in the menu. Once checked, dependent files are shown as sub-nodes of the file which depends upon them.

In this case, `main.c` is dependent upon `cross_studio_io.h` because it includes it with a **#include** directive. You can open `cross_studio_io.h` in an editor by double clicking it, so having dependencies turned on is an effective way of navigating to and summarising the files that a source file includes.

### Output files

Another useful piece of information is knowing the files output files when compiling and linking the application. Needless to say, CrossStudio can display this information too. To turn on output file display, click the **Options** button on the **Project Explorer** tool bar and ensure that **Show Output Files** is checked in the menu. Once checked, output files are shown in an **Output Files** folder underneath the node that generates them.

In the above figure, we can see that the object files `fact.o`, `main.o`, and `crt0.o` are object files produced by compiling their corresponding source files; the map file `Tutorial.map` and the linked executable `Tutorial.elf` are produced by the linker. As a convenience, double clicking an object file or a linked executable file in the Project Explorer will open an editor showing the disassembled contents of the file.

### Disassembling a project or file

You can disassemble a project either by double clicking the corresponding file as described above, or you can use the Disassemble tool to do it.

To disassemble a project or file, do one of the following:

- Click the appropriate project or file in the **Project Explorer** view.
- On the **Project Explorer** tool bar, click the **Disassemble** tool button .



or

- Right click the appropriate project or file in the **Project Explorer** view.
- From the popup menu, click the **Disassemble**.

CrossStudio opens a new read-only editor and places a disassembled listing into it. If you change your project and rebuild it, causing a change in the object or executable file, the disassembly updates to keep the display up-to-date with the file on disk.

## Using Symbol Browser features

Whilst a map file produced by the linker is *traditionally* the best way (and in some cases, the only way) to see how your application is laid out in memory, CrossStudio provides a much better way to examine and navigate your application: the *Symbol Browser*. You can use the Symbol Browser to navigate your application, see which data objects and functions have been linked into your application, what their sizes are, which section they are in, and where they are placed in memory.



## Displaying the Symbol Browser

To display the **Symbol Browser** window if it is hidden, do one of the following:

- From the **View** menu, click **Symbol Browser**.

or

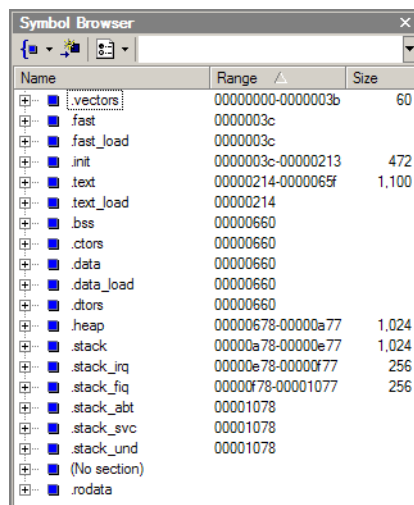
- Type **Ctrl+Alt+Y**.

or

- Right click the tool bar area to display the **View** menu.
- From the popup menu, click **Symbol Browser**.

## Drilling down into the application

The **Tutorial** project shows this in the Symbol Browser:



From this you can see:

- The **.vectors** section containing the ARM exception vectors is placed in memory between address 0x00000000 and 0x0000003B.
- The **.fast** section containing performance critical code and data is empty.
- The **.init** section containing the system startup code is placed in memory between address 0x0000003C and 0x00000213.
- The **.text** section containing the program code is placed in memory between address 0x00000214 and 0x0000065F.
- The **.text\_load** section which is the section that the **.text** section image is loaded from is at the same address as **.text**. If the **.text** and **.text\_load** section start addresses

differ then the startup code will copy the contents of the **.text\_load** section to the **.text** section before the program enters main. This feature allows the **.text** section to run from RAM in ROM based applications.

- The **.bss** section containing zeroed data is empty.
- The **.ctors** containing the global constructor table is empty.
- The **.data** section containing initialized data is empty.
- The **.data\_load** section which is the section that the **.data** section image is loaded from is at the same address as **.data**. If the **.data** and **.data\_load** section start addresses differ then the startup code will copy the contents of the **.data\_load** section to the **.data** section before the program enters main. This feature is required for ROM based applications so that data can be initialized in RAM on startup.
- The **.dtors** section containing the global destructor table is empty.
- The **.heap** section is 1024 bytes in length and located at 0x00000678. Note that the size of the heap can be adjusted by modifying the size of the **.heap** section in the section placement file (sram\_placement.xml in this example).
- The **.stack** section which contains the User/System mode stack is 1024 bytes in length and located at 0x00000A78. Note that the sizes of the stacks can be adjusted by modifying the size of the **.stack** sections in the section placement file (sram\_placement.xml in this example).
- The **.stack\_irq** section which contains the IRQ mode stack is 256 bytes in length and located at 0x00000E78.
- The **.stack\_fiq** section which contains the FIQ mode stack is 256 bytes in length and located at 0x00000F78.
- The **.stack\_abt** section which contains the Abort mode stack is 0 bytes in length.
- The **.stack\_svc** section which contains the Supervisor mode stack is 0 bytes in length.
- The **.stack\_und** section which contains the Undefined mode stack is 0 bytes in length.

To drill down, open the **.text** node by double clicking it: CrossStudio displays the individual functions that have been placed in memory and their sizes:

Here, we can see that **main** is 104 bytes in size and is placed in memory between addresses 0x00000270 and 0x000002D7 inclusive. Just as in the Project Explorer, you can double click a function and CrossStudio moves the cursor to the line containing the definition of that function, so you can easily navigate around your application using the Symbol Browser.

### Printing Symbol Browser contents

You can print the contents of the Symbol Browser by focusing the Symbol Browser window and selecting **Print** from the **File** menu, or **Print Preview** if you want to see what it will look like before printing. CrossStudio prints only the columns that you have selected for display, and prints items in the same order they are displayed in the Symbol Browser, so you can choose which columns to print and how to print symbols by configuring the Symbol Browser display before you print.

We have touched on only some of the features that the Symbol Browser offers; to find out more, refer to **Symbol browser** (page 133) where it is described in detail.

## Using the debugger

Our sample application, which we have just compiled and linked, is now built and ready to run. In this section we'll concentrate on downloading and debugging this application, and using the features of CrossStudio to see how it performs.

### Getting set up

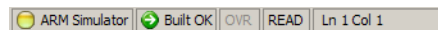
Before running your application, you need to select the target to run it on. The Targets window lists each target interface that is defined, as does the Targets menu, and you use these to connect CrossStudio to a target. For this tutorial, you'll be debugging on the simulator, not hardware, to simplify matters. To connect to the simulator, do one of the following:

- From the **Target** menu, click **Connect ARM Simulator**.

—or—

- From the **View** menu, click **Targets** to focus the Targets window.
- In the Targets window, double click **ARM Simulator**.

After connecting, the connected target is shown in the status bar:




The color of the LED in the Target Status panel changes according to what CrossStudio and the target are doing:

- **White** — No target is connected.
- **Yellow** — Target is connected.
- **Solid green** — Target is free running, not under control of CrossStudio or the debugger.

- **Flashing green** — Target is running under control of the debugger.
- **Solid red** — Target is stopped at a breakpoint or because execution is paused.
- **Flashing red** — CrossStudio is programming the application into the target.

## Setting a breakpoint

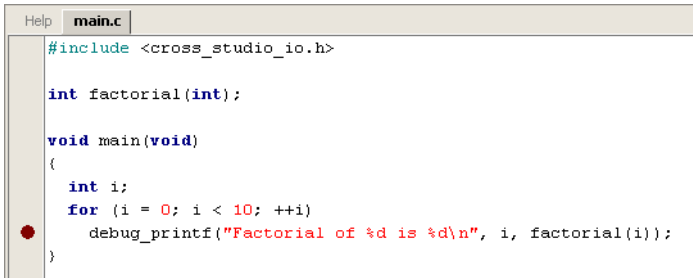
CrossStudio will run a program until it hits a breakpoint. We'll place a breakpoint on the call to `debug_printf` in `main.c`. To set the breakpoint, Move the cursor to the line containing `debug_printf` and do one of the following:

- On the **Build** tool bar, click the **Toggle Breakpoint** button — .  


—or—

- Type **F9**.

Alternatively, you can set a breakpoint without moving the cursor by clicking in the gutter of the line to set the breakpoint on.



```

Help  main.c
#include <cross_studio_io.h>

int factorial(int);

void main(void)
{
    int i;
    for (i = 0; i < 10; ++i)
        debug_printf("Factorial of %d is %d\n", i, factorial(i));
}

```


The gutter displays an icon on lines where the breakpoints are set. The Breakpoints window updates to show where each breakpoint is set and whether it's set, disabled, or invalid—you can find more detailed information in the **Breakpoints window** (page 102) section. The breakpoints that you set are stored in the session file associated with the project which means that your breakpoints are remembered if you exit and re-run CrossStudio.

## Starting the application

You can now start the program in one of these ways:

- From the **Debug** menu, click **Start Debugging**.

—or—

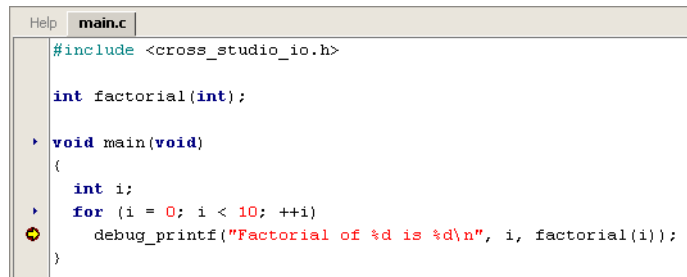
- On the **Build** tool bar, click the **Start Debugging** button — .  


—or—

- Type F5.

The workspace will change from the standard Editing workspace to the Debugging workspace. You can choose which windows to display in both these workspaces and manage them independently. CrossStudio loads the active project into the target and places the breakpoints that you have set. During loading, the the Target Log in the Output Window shows its progress and any problems:

The program stops at our breakpoint and a yellow arrow indicates where the program is paused.

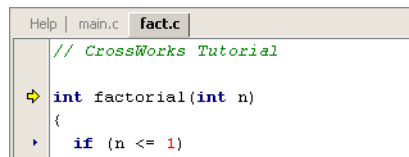


```
Help | main.c
#include <cross_studio_io.h>

int factorial(int);

void main(void)
{
    int i;
    for (i = 0; i < 10; ++i)
        debug_printf("Factorial of %d is %d\n", i, factorial(i));
}
```

You can step over a statement by selecting **Debug > Step Over**, by typing **F10** or by clicking the **Step Over** button on the **Debug** tool bar. Right now, we'll step into the next function, `factorial`, and trace its execution. To step into `factorial`, select **Debug > Step Into**, type **F11**, or click the **Step Into** button on the **Debug** tool bar. Now the display changes to show that you have entered `factorial` and execution is paused there.



```
Help | main.c | fact.c
// CrossWorks Tutorial

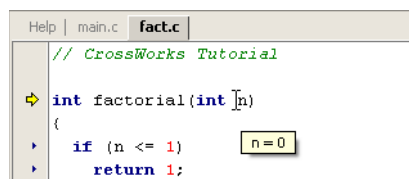
int factorial(int n)
{
    if (n <= 1)
```

You can also step to a specific statement using **Debug > Run To Cursor**. To restart your application to run to the next breakpoint use **Debug > Go**.

Note that when single stepping you may step into a function that the debugger cannot locate source code for. In this case the debugger will display the instructions of the application, you can step out to get back to source code or continue to debug at the instruction code level. There are may be cases in which the debugger cannot display the instructions, in these cases you will informed of this with a dialog and you should step out.

## Inspecting data

Being able to control execution isn't very helpful if you can't look at the values of variables, registers, and peripherals. Hovering the mouse pointer over a variable will show its value as a *data tip*:



```
Help | main.c | fact.c
// CrossWorks Tutorial
int factorial(int n)
{
    if (n <= 1)
        return 1;
}
```

You can configure CrossStudio to display data tips in a variety of formats at the same time using the Environment Options dialog.

The Call Stack window shows the function calls that have been made but have not yet finished, i.e. the active set of functions. To display the Call Window, select **Debug > Debug Windows > Call Stack**, or type **Ctrl+Alt+S**.

You can find out about the call stack window in the **Call stack window** (page 107) section.

## Program output

The tutorial application uses the function `debug_printf` to output a string to the Debug Console in the Output Window. The Debug Console appears automatically whenever something is written to it—pressing **F5** to continue program execution and you will notice that the Debug Console appears. In fact, the program runs forever, writing the same messages over and over again. To pause the program, select **Debug > Break** or type **Ctrl+.** (control-period).

In the next section we'll cover low-level debugging at the machine level.

## Low-level debugging

This section describes how to debug your application at the register and instruction level. Debugging at a high level is all very well, but there are occasions where you need to look a little more closely into the way that your program executes to track down the causes of difficult-to-find bugs and CrossStudio provides the tools you need to do just this.

## Setting up again

What we'll now do is run the sample application, but look at how it executes at the machine level. If you haven't done so already, stop the program executing by typing **Shift+F5**, by selecting **Stop Debugging** from the **Debug** menu, or clicking the **Stop Debugging** button on the **Debug** tool bar. Now run the program so that it stops at the first breakpoint again.

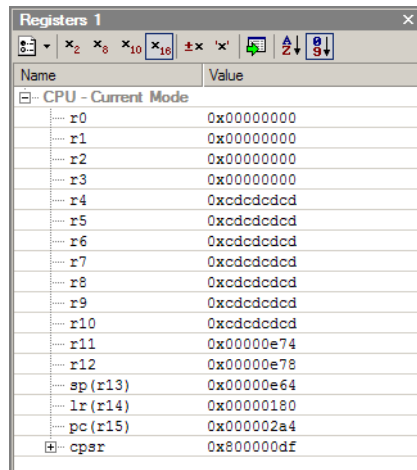
You can see the current processor state in the **Register** windows. To show the first registers window, do one of the following:

- From the **Debug** menu, click **Debug Windows** then **Registers 1**.

—or—

- Type **Ctrl+T, R, 1**.

Your registers window will look something like this:



Name	Value
CPU - Current Mode	
r0	0x00000000
r1	0x00000000
r2	0x00000000
r3	0x00000000
r4	0xcdcdcdcd
r5	0xcdcdcdcd
r6	0xcdcdcdcd
r7	0xcdcdcdcd
r8	0xcdcdcdcd
r9	0xcdcdcdcd
r10	0xcdcdcdcd
r11	0x00000e74
r12	0x00000e78
sp (r13)	0x00000e64
lr (r14)	0x00000180
pc (r15)	0x000002a4
cpsr	0x800000df

This register view is displaying the registers for the active processor mode. You can also display the entire set of ARM registers, to do this select **CPU - ALL** from the **Groups** menu on the toolbar.

There are four register windows so you can open and display four sets of peripheral registers at the same time.

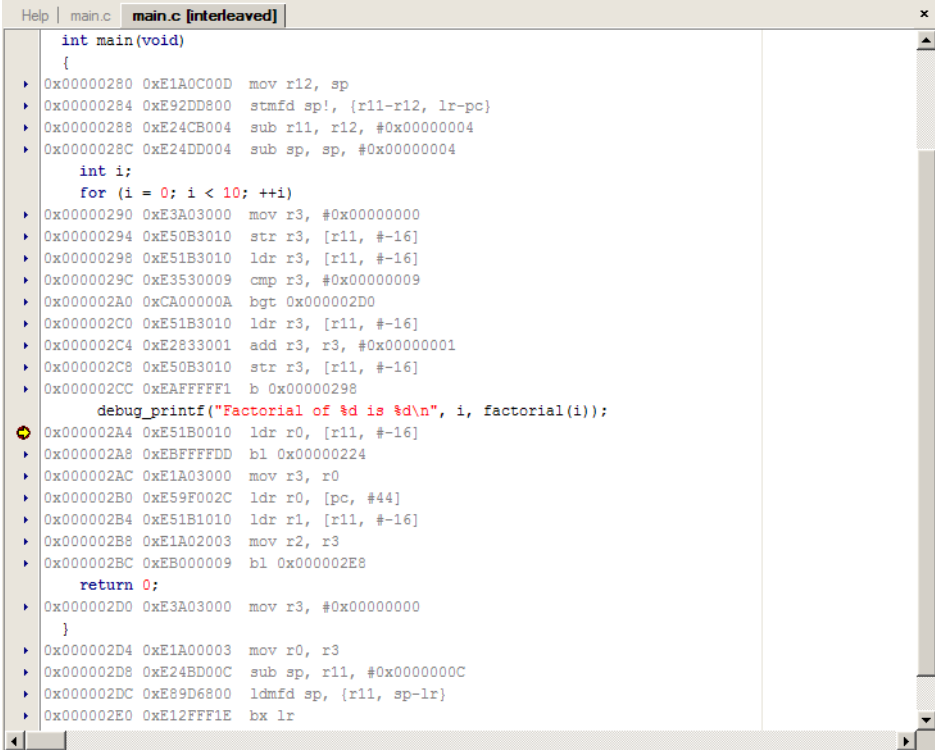
You can configure which registers and peripherals to display in the Registers windows individually. As you single step the program, the contents of the Registers window updates automatically and any change in a register value is highlighted in red.

## Debugging modes

The debugger supports three modes of debug

- **Source mode** where the source code is displayed in a code editor.
- **Interleaved mode** where the editor displays an interleaved listing of the currently located source code. All single stepping is done an instruction at a time.
- **Assembly mode** where a disassembly of the instructions around the currently located instruction is shown in the editor. All single stepping is done an instruction at a time.

You have already seen debugging at the source level. To single step at the assembly level, from the **Debug** menu click **Control** then **Interleaved Mode**. The editor window now interleaves the source code of the application with the assembly language generated by the compiler:



```
Help | main.c | main.c [interleaved]
int main(void)
{
  0x00000280 0xE1A0C00D mov r12, sp
  0x00000284 0xE92DD800 stmfid sp!, {r11-r12, lr-pc}
  0x00000288 0xE24CB004 sub r11, r12, #0x00000004
  0x0000028C 0xE24DD004 sub sp, sp, #0x00000004
  int i;
  for (i = 0; i < 10; ++i)
  0x00000290 0xE3A03000 mov r3, #0x00000000
  0x00000294 0xE50B3010 str r3, [r11, #-16]
  0x00000298 0xE51B3010 ldr r3, [r11, #-16]
  0x0000029C 0xE3530009 cmp r3, #0x00000009
  0x000002A0 0xCA00000A bgt 0x000002D0
  0x000002C0 0xE51B3010 ldr r3, [r11, #-16]
  0x000002C4 0xE2833001 add r3, r3, #0x00000001
  0x000002C8 0xE50B3010 str r3, [r11, #-16]
  0x000002CC 0xEAFFFFF1 b 0x00000298
  debug_printf("Factorial of %d is %d\n", i, factorial(i));
  0x000002A4 0xE51B0010 ldr r0, [r11, #-16]
  0x000002A8 0xEBFFFDD bl 0x00000224
  0x000002AC 0xE1A03000 mov r3, r0
  0x000002B0 0xE59F002C ldr r0, [pc, #44]
  0x000002B4 0xE51B1010 ldr r1, [r11, #-16]
  0x000002B8 0xE1A02003 mov r2, r3
  0x000002BC 0xEB000009 bl 0x000002E8
  return 0;
  0x000002D0 0xE3A03000 mov r3, #0x00000000
}
  0x000002D4 0xE1A00003 mov r0, r3
  0x000002D8 0xE24BD00C sub sp, r11, #0x0000000C
  0x000002DC 0xE89D6800 ldmfd sp, {r11, sp-lr}
  0x000002E0 0xE12FFF1E bx lr
```

In interleaved mode, debugging controls such as single step, step into, and step out work at the instruction level, not the source level. To return to high-level source debugging, select **Debug > Control > Source Mode**.

There are other windows that help you with debugging, such as the memory view and the watch windows, and the CrossStudio Window Reference describes these.



## Stopping and starting debugging

You can stop debugging using **Debug > Stop**. If you wish to restart debugging without reloading the program then you can use **Debug > Debug From Reset**. Note that when you debug from reset no loading takes place so it is expected that your program is built in a way such that any resetting of data values is done as part of the program startup. You can also attach the debugger to a running target using the **Debug > Attach Debugger**.



## Overview

This section introduces the overall layout and operation of the CrossStudio integrated development environment.

### CrossStudio standard layout

CrossStudio's main window is divided into the following areas:

- **Title bar** Displays the name of the current file being edited and the active workspace.
- **Menu bar** Dropdown menus for editing, building, and debugging your program.
- **Toolbars** Frequently used actions are quickly accessible on toolbars below the menu bar.
- **Editing area** A tabbed or MDI view of multiple editors and the HTML viewer.
- **Docked windows** CrossStudio has many windows which can be docked to the left of, to the right of, or below the editing area. You can configure which windows are visible when editing and debugging. The figure shows the project explorer, targets window, and output window.
- **Status bar** At the bottom of the window, the status bar contains useful information about the current editor, build status, and debugging environment.

# The title bar

CrossStudio's title bar displays the name of the active editor tab if in **Tabbed Document Workspace** mode or the active MDI window if in **Multiple Document Workspace** mode.

## Title bar format

The first item shown in the title bar is CrossStudio's name. Because CrossStudio targets different processors, the name of the target processor family is also shown so you can distinguish between instances of CrossStudio when debugging multi-processor or multi-core systems.

The file name of the active editor follows CrossStudio's name; you can configure the exact presentation of the file name this as described below.

After the file name, the title bar displays status information on CrossStudio's state:

- **[building]**. CrossStudio is building a solution, building a project, or compiling a file.
- **[run]**. An application is running under control of the CrossStudio's inbuilt debugger
- **[break]**. The debugger is stopped at a breakpoint.
- **[autostep]**. The debugger is single stepping the application without user interaction—this is called *autostepping*.

The **Target Status** panel in the status bar also shows CrossStudio's state—see **The status bar** (page 38).

## Configuring the title bar

You can configure whether the full path of the file or just its file name is shown in the title bar.

### Displaying the full file path in the title bar

To display the full file path in the title bar, do the following:

- From the **Tools** menu, click **Options**.
- In the **Appearance** group, check **Show full path in title bar**.

### Displaying only the file name in the title bar

To display only the file name in the title bar, do the following:

- From the **Tools** menu, click **Options**.
- In the **Appearance** group, uncheck **Show full path in title bar**.

# The menu bar

The menu bar contains dropdown menus for editing, building, and debugging your program. You can navigate menu items using the keyboard or using the mouse. You'll find a complete description of each menu and its contents in **CrossStudio menu summary** (page 173).

## Navigating menus using the mouse

To navigate menus using the mouse, do the following:

- Click the required menu title in the menu bar; the menu appears.
- Click the required menu item in the dropdown menu.

—or—

- Click and hold the mouse on the required menu title in the menu bar; the menu appears.
- Drag the mouse to the required menu item on the dropdown menu.
- Release the mouse.

## Navigating menus using the keyboard

To navigate menus using the keyboard, do the following:

- Tap the **Alt** key which focuses the menu bar.
- Use the **Left** and **Right** keys to navigate to the required menu.
- Use the **Up** or **Down** key to activate the required menu
- Type **Alt** or **Esc** to cancel menu selection at any time.

Each menu on the menu bar has one letter underlined, its shortcut, so to activate the menu using the keyboard:

- Whilst holding down the **Alt** key, type the menu's shortcut.

Once the menu has dropped down you can navigate it using the cursor keys:

- Use **Up** and **Down** to move up and down the menu.
- Use **Esc** to cancel a dropdown menu.
- Use **Right** or **Enter** to open a submenu.
- Use **Left** or **Esc** to close a submenu and return to the parent menu.
- Type the underlined letter in a menu item to activate that menu item.
- Type **Enter** to activate the selected menu item.

## The status bar

At the bottom of the window, the status bar contains useful information about the current editor, build status, and debugging environment. The status bar is divided into two regions, one that contains a set of fixed panels and the other that is used for messages.

### The message area

The leftmost part of the status bar is a message area that is used for things such as status tips, progress information, warnings, errors, and other notifications.

### The status bar panels

You can show or hide the following panels on the status bar:

Panel	Description
Target device status	Displays the connected target interface. When connected, this panel contains the selected target interface name and, if applicable, the processor that the target interface is connected to. The LED icon flashes green when programs are running, is solid red when stopped at a breakpoint, and is yellow when connected but not running a program. Double clicking this panel displays the Targets window and right clicking it brings up the Target menu.
Cycle count panel	Displays the number of processor cycles run by the executing program. This panel is only visible if the currently connected target supports performance counters which can report the total number of cycles executed. Double clicking this panel resets the cycle counter to zero, and right clicking this panel brings up the Cycle Count menu.
Insert/overwrite status	Indicates whether the current editor is in insert or overwrite mode. If the editor is in overwrite mode the OVR panel is highlighted otherwise it is dimmed.
Read only status	Indicates whether the editor is in read only mode. If the editor is editing a read only file or is in read only mode, the READ panel is highlighted otherwise it is dimmed.
Build status	Indicates the success or failure of the last build. If the last build completed without errors or warnings, the build status pane contains "Build OK" otherwise it contains the number of errors and warnings reported. Right clicking this panel displays the Build Log in the Output window.

Panel	Description
Caret position	Indicates the cursor position of the current editor. For text editors, the caret position pane displays the line number and column number of the cursor; for binary editors it displays the address where the
Caps lock status	Indicates the Caps Lock state. If the Caps Lock is on, CAPS is highlighted, otherwise it is dimmed.
Num lock status	Indicates the Num Lock state. If the Num Lock is on, NUM is highlighted, otherwise it is dimmed.
Scroll lock status	Indicates the Scroll Lock state. If the Scroll Lock is on, SCR is highlighted, otherwise it is dimmed.
Time panel	Displays the current time.

### Configuring the status bar panels

To configure which panels are shown on the status bar, do the following:

- From the **View** menu, click **Status Bar**.
- From the status bar menu, check the panels that you want displayed and uncheck the ones you want hidden.

—or—

- Right click on the status bar.
- From the status bar menu, check the panels that you want displayed and uncheck the ones you want hidden.

You can also select the panels to display from the **Tools > Options** dialog in the **Environment > More...** folder.

- From the **Tools** menu, click **Options**.
- In the tree view **Environment** folder, click **More...**
- In the **Status bar** group, check the panels that you want displayed and uncheck the ones you want hidden.

### Hiding the status bar

To hide the status bar, do the following:

- From the **View** menu, click **Status Bar**.
- From the status bar menu, uncheck the **Status Bar** menu item.

—or—

- Right click on the status bar.

- From the status bar menu, uncheck the **Status Bar** menu item.

### Showing the status bar

To show the status bar, do the following:

- From the **Tools** menu, click **Options**.
- In the tree view **Environment** folder, click **More...**
- In the **Status bar** group, check **(visible)**.

### Showing or hiding the size grip

You can choose to hide or display the size grip when the CrossStudio main window is not maximized—the size grip is never shown in full screen mode or when maximized. To do this:

- From the **View** menu, click **Status Bar**.
- From the status bar menu, uncheck the **Size Grip** menu item.

—or—

- Right click on the status bar.
- From the status bar menu, uncheck the **Size Grip** menu item.

You can also choose to hide or display the size grip from the **Tools > Options** dialog in the **Environment > More...** folder.

- From the **Tools** menu, click **Options**.
- In the tree view **Environment** folder, click **More...**
- In the **Status bar** group, check or uncheck the **Size grip** item.

## The editing workspace

The main area of CrossStudio is the editing workspace. This area contains files that are being edited by any of the editors in CrossStudio, and also the online help system's HTML browser.

You can organize the windows in the editing area either into tabs or as separate windows. In **Tabbed Document Workspace** mode, only one window is visible at any one time, and each of the tabs displays the files name. In **Multiple Document Workspace** mode, many overlapping windows are displayed in the editing area.

By default, CrossStudio starts in **Tabbed Document Workspace** mode, but you can change at any time between the two.



### **Changing to Multiple Document Workspace mode**

To change to Multiple Document Workspace mode, do the following:

- From the **Window** menu, click **Multiple Document Workspace**.

### **Changing to Tabbed Document Workspace mode**

To change to Tabbed Document Workspace mode, do the following:

- From the **Window** menu, click **Tabbed Document Workspace**.

The document mode is remembered between invocations of CrossStudio.



# Project management

CrossWorks has a project system that enables you to manage the source files and build instructions of your solution. The **Project Explorer** and the **Properties Window** are the standard ways to edit and view your solution. You can also edit and view the project file which contains your solution using the text editor—this can be used for making large changes to the solution.

## In this section

- **Project system (page 44).** A summary of the features of the CrossStudio project system.
- **Creating a project (page 46).** Describes how to create a project and add it to a solution.
- **Adding existing files to a project (page 47).** Describes how to add existing files to a project, how filters work, and what folders are for.
- **Adding new files to a project (page 48).** Describes how create and add new files to a project.
- **Removing a file, folder, project, or project link (page 48).** Describes how to remove items from a project.
- **Project properties (page 49).** Describes what properties are, how they relate to a project, and how to change them.
- **Project configurations (page 52).** Describes what project build configurations are, to to create them, and how to use them**Project configurations** (page 52).
- **Project dependencies and build order (page 53).** Describes project dependencies, how to edit them, and how they are used to define the order projects build in.

- **Project macros (page 54).** Describes what project macros are and what they are used for.

#### Related sections

- **Project explorer (page 128).** Describes the project explorer and how to use it.
- **Project property reference.** A complete reference to the properties used in the project system.
- **Project file format (page 340).** Describes the XML format CrossStudio uses for project files.

## Project system

A solution is a collection of projects, and all projects are contained in solutions. Organizing your projects into a solution allows you to build all the projects in a solution with a single keystroke, load them onto the target ready for debugging with another.

Projects in a solution can reside in the same or different directories. Project directories are always relative to the directory of the solution file which enables you to move or share project file hierarchies on different computers.

The **Project Explorer** organizes your projects and files and provides quick access to the commands that operate on them. A tool bar at the top of the window offers quick access to commonly used commands for the item selected in the **Project Explorer**.

## Projects

The projects you create within a solution have a project type which CrossStudio uses to determine how to build the project. The project type is selected when you use the **New Project** dialog. The particular set of project types can vary depending upon the variant of CrossWorks you are using, however the following project types are standard to most CrossWorks variants:

- **Executable** — a program that can be loaded and executed.
- **Externally Built Executable** — an executable that is not built by CrossWorks.
- **Library** — a group of object files that collected into a single file (sometimes called an archive).
- **Object File** — the result of a single compilation.
- **Staging** — a project that can be used to apply a user defined command (for example cp) to each file in a project.

- **Combining** — a project that can be used to apply a user defined command when any files in a project have changed.

## Properties and configurations

Properties are data that are attached to project nodes. They are usually used in the build process for example to define C preprocessor symbols. You can have different values of the same property based on a configuration, for example you can change the value of a C preprocessor symbol for a release or a debug build.

## Folders

Projects can contain folders which are used to group related files together. This grouping can be done using the file extension of the file or it can be done by explicitly creating a file within a folder. Note that folders do not map onto directories in the file store they are solely used to structure the project explorer display.

## Files

The source files of your project can be placed either in folders or directly in the project. Ideally files placed in project should be relative to the project directory, however there are cases when you might want to refer to a file in an absolute location and this is supported by the project system. The project system will allow (with a warning) duplicate files to be put into a project.

The project system uses the extension of the file to determine the appropriate build action to perform on the file. So

- a file with the extension **.c** will be compiled by a C compiler.
- a file with the extension **.s** or **.asm** will be compiled an assembler.
- a file with the extension **.cpp** or **.cxx** will be compiled by a C++ compiler.
- a file with the object file extension **.o** or **.hzo** will be linked.
- a file with the library file extension **.a** or **.hza** will be linked.
- a file with the extension **.xml** will be opened and its file type determined by the XML document type.
- other file extensions will not be compiled/linked with.

You can modify this behaviour by setting the **File Type** property of the file with the **Common** configuration selected in the properties window which enables files with non-standard extensions to be compiled by the project system.

## Solution links

You can create links to existing project files from a solution which enables you to create hierarchical builds. For example you could have a solution that builds a library together with a stub test driver executable. You can then link to this solution (by right clicking on the solution node of the project explorer and selecting **Add Existing Project**) to be able to use the library from a project in the current solution.

## Project and session files

When you have created a solution it is stored in a project file. Project files are text files with the file extension **hzp** that contain an XML description of your project. When you exit CrossWorks, details of your current session are stored in a *session file*. Session files are text files with the file extension **hzs** that contain details such as files you have opened in the editor and breakpoints you set in the breakpoint window.

## Creating a project

You can create a new solution for each project or alternatively create projects in an existing solution.

To create a new project in an existing solution, do the following:

- From the **Project** menu, click **New** then **New Project...** to display the **New Project** wizard.
- In the **New Project** wizard, select the type of project you wish to create and where it will be placed.
- Ensure that the "Add the project to current solution" radio button is checked.
- Click **OK** to go to next stage of project creation or **Cancel** to cancel the creation.

The project name must be unique to the solution and ideally the project directory should be relative to the solution directory. The project directory is where the project system will use as the current directory when it builds your project. Once complete, the project explorer displays the new solution, project, and files of the project. To add another project to the solution, repeat the above steps.

### Creating a new project in a new solution

To create a new project in a new solution, do the following:

- From the **File** menu, click **New** then **New Project...** to display the **New Project** dialog.

- In the **New Project** dialog, select the type of project you wish to create and where it will be placed.
- Click **OK**.

## Adding existing files to a project

You can add existing files to a project in a number of ways.

### Adding existing files to the active project

You can add one or more files to the active project quickly using the standard **Open File** dialog.

To add existing files to the active project do one of the following:

- From the **Project** menu, select **Add Existing File...**

—or—

- On the **Project Explorer** tool bar, click the **Add Existing File** button.

—or—

- Type **Ctrl+D**.

Using the **Open File** dialog, navigate to the directory containing the existing files, select the ones to add to the project, then click **OK**. The selected files are added to the folders whose filter matches the extension of the each of the files. If no filter matches a files extension, the file is placed underneath the project node.

### Adding existing files to any project

To add existing files a project without making it active:

- In the **Project Explorer**, right click on the project to add a new file to.
- From the popup menu, select **Add Existing File...**

The procedure for adding existing files is the same as above.

### Adding existing files to a specific folder

To add existing files directly to a folder bypassing the file filter do the following:

- In the **Project Explorer**, right click on the folder to add a new file to.
- From the popup menu, select **Add Existing File...**

The files are added to the folder without using any filter matching.

## Adding new files to a project

You can add new files to a project in a number of ways.

### Adding a new file to the active project

To add new files to the active project, do one of the following:

- From the **Project** menu, click **Add New File...**

—or—

- On the **Project Explorer** tool bar, click the **Add New File** button.

—or—

- Type **Ctrl+N**.

### Adding a new file to any project

To add a new file to a project without making it active, do one of the following:

- In the **Project Explorer**, right click on the project to add a new file to.
- From the popup menu, select **Add New File...**

When adding a new file, CrossStudio displays the **New File** dialog from which you can choose the type of file to add, its file name, and where it will be stored. Once created, the new file is added to the folder whose filter matches the extension of the newly added file. If no filter matches the newly added file extension, the new file is placed underneath the project node.

### Adding a new file to a specific folder

To add new files directly to a folder bypassing the file filter do the following:

- In the **Project Explorer**, right click on the folder to add a new file to.
- From the popup menu, select **Add New File...**

The new file is added to the folder without using any filter matching.

## Removing a file, folder, project, or project link

You can remove whole projects, folders, or files from a project, or you can remove a project from a solution using the **Remove** tool button on the project explorer's toolbar. Removing a source file from a project does not remove it from disk.

### Removing an item

To remove an item from the solution do one of the following:



- Click on the project item to remove from the **Project Explorer** tree view.
  - On the **Project Explorer** toolbar, click the **Remove** button (or type **Delete**).
- or—
- Right click on the the project item to remove from the **Project Explorer** tree view.
  - From the popup menu, click **Remove**.

## Project properties

For solutions, projects, folders and files - properties can be defined that are used by the project system in the build process. These property values can be viewed and modified using the properties window in conjunction with the project explorer. As you select an item in the project explorer the properties window will list the set of properties that are applicable.

Some properties are only applicable to a given item type. For example linker properties are only applicable to a project that builds an executable file. However other properties can be applied either at the file, project or solution project node. For example a compiler property can be applied to the solution, project or individual file. By setting properties at the solution level you enable all files of the solution to use this property value.

### Unique properties

A unique property has *one* value. When a build is done the value of a unique property is the first one defined in the project hierarchy. For example the **Treat Warnings As Errors** property could be set to **Yes** at the solution level which would then be applicable to every file in the solution that is compiled, assembled and linked. You can then selectively define property values for other project items. For a example particular source file may have warnings that you decide are allowable so you set the **Treat Warnings As Errors** to **No** for this particular file.

Note that when the properties window displays a project property it will be shown in **bold** if it has been defined for unique properties. The inherited or default value will be shown if it hasnt been defined.

```
solution — Treat Warnings As Errors = Yes
  project1 — Treat Warnings As Errors = Yes
    file1 — Treat Warnings As Errors = Yes
    file2 — Treat Warnings As Errors = No
```

project2 — **Treat Warnings As Errors = No**  
 file1 — **Treat Warnings As Errors = No**  
 file2 — **Treat Warnings As Errors = Yes**

In the above example the files will be compiled with these values for **Treat Warnings As Errors**

project1/file1	Yes
project1/file2	No
project2/file1	No
project2/file2	Yes

## Aggregating properties

An aggregating property collects all of the values that are defined for it in the project hierarchy. For example when a C file is compiled the **Preprocessor Definitions** property will take all of the values defined at the file, project and solution level. Note that the properties window *will not* show the inherited values of an aggregating property.

solution — **Preprocessor Definitions = SolutionDef**  
 project1 — **Preprocessor Definitions =**  
 file1 — **Preprocessor Definitions =**  
 file2 — **Preprocessor Definitions = File1Def**  
 project2 — **Preprocessor Definitions = ProjectDef**  
 file1 — **Preprocessor Definitions =**  
 file2 — **Preprocessor Definitions = File2Def**

In the above example the files will be compiled with these **Preprocessor Definitions**

project1/file1	SolutionDef
project1/file2	SolutionDef, File1Def
project2/file1	SolutionDef, ProjectDef
project2/file2	SolutionDef, ProjectDef, File2Def

## Configurations and property values

Property values are defined for a configuration so you can have different values for a property for different builds. A given configuration can inherit the property values of other configurations. When the project system requires a property value it checks for the existence of the property value in current configuration and then in the set of inherited configurations. You can specify the set of inherited configurations using the **Configurations** dialog.

There is a special configuration named **Common** that is always inherited by a configuration. The **Common** configuration enables property values to be set that will apply to all configurations that you create. You can select the **Common** configuration using the **Configurations** combo box of the properties window. If you are modifying a property value of your project its almost certain that you want each configuration to inherit these values - so ensure that the **Common** configuration has been selected.

If the property is unique then it will use the one defined for the particular configuration. If the property isnt defined for this configuration then it uses an arbitrary one from the set of inherited configurations. If the property still isnt defined it uses the value for the **Common** configuration. If it still isnt defined then it tries the to find the value in the next level of the project hierarchy.

```
solution [Common] — Preprocessor Definitions = CommonSolutionDef
solution [Debug] — Preprocessor Definitions = DebugSolutionDef
solution [Release] — Preprocessor Definitions = ReleaseSolutionDef
  project1 - Preprocessor Definitions =
    file1 - Preprocessor Definitions =
      file2 [Common] — Preprocessor Definitions = CommonFile1Def
      file2 [Debug] — Preprocessor Definitions = DebugFile1Def
  project2 [Common] — Preprocessor Definitions = ProjectDef
    file1 — Preprocessor Definitions =
      file2 [Common] - Preprocessor Definitions = File2Def
```

In the above example the files will be compiled with these **Preprocessor Definitions** when in **Debug** configuration

project1/file1	CommonSolutionDef, DebugSolutionDef
project1/file2	CommonSolutionDef, DebugSolutionDef,CommonFile1Def, DebugFile1Def
project2/file1	CommonSolutionDef, DebugSolutionDef, ProjectDef
project2/file2	ComonSolutionDef, DebugSolutionDef, ProjectDef, File2Def

and the files will be compiled with these **Preprocessor Definitions** when in **Release** configuration

project1/file1	CommonSolutionDef, ReleaseSolutionDef
project1/file2	CommonSolutionDef, ReleaseSolutionDef, CommonFile1Def

project1/file1	CommonSolutionDef, ReleaseSolutionDef
project2/file1	CommonSolutionDef, ReleaseSolutionDef, ProjectDef
project2/file2	ComonSolutionDef, ReleaseSolutionDef, ProjectDef, File2Def

## Project configurations

Project configurations are used to create different software builds for your projects. A configuration is used to define different project property values, for example the output directory of a compilation can be put into different directories which are dependent upon the configuration. By default when you create a solution you'll get some default project configurations created.

### Selecting a configuration

You can set the configuration that you are building and debugging with using the combo box of the **Build** tool bar or the **Build > Set Active Build Configuration** menu option.

### Creating a configuration

You can create your own configurations using **Build > Build Configurations** which will show the **Configurations** dialog. The **New** button will produce a dialog that allows you name your configuration. You can now specify which existing configurations your new configuration will inherit values from.

### Deleteing a configuration

You can delete a configuration by selecting it and pressing the **Remove** button. Note that this operation cannot be undone or cancelled so beware.

### Hidden configurations

There are some configurations that are defined purely for inheriting and as such should not appear in the build combo box. When you select a configuration in the configuration dialog you can specify if you want that configuration to be hidden.

## Project dependencies and build order

You can set up dependency relationships between projects using the **Project Dependencies** dialog. Project dependencies make it possible to build solutions in the correct order and where the target permits, to manage loading and deleting applications and libraries in the correct order. A typical usage of project dependencies is to make an executable project dependent upon a library executable. When you elect to build the executable then the build system will ensure that the library it is dependent upon is up to date. In the case of a dependent library then the output file of the library build is supplied as an input to the executable build so you don't have to worry about this.

Project dependencies are stored as project properties and as such can be defined differently based upon the selected configuration. You almost always want project dependencies to be independent of the configuration so the Project Dependencies dialog selects the **Common** configuration by default.

### Making a project dependent upon another

To make one project dependent upon another, do the following:

- From the **Project** menu, click **Dependencies** to display the **Project Dependencies** dialog.
- From the **Project** dropdown, select the target project which depends upon other projects.
- In the **Depends Upon** list box, check the projects that the target project depends upon and uncheck the projects that it does not depend upon.

Some items in the **Depends Upon** list box may be disabled, which indicates that if the project were checked, a circular dependency would result. Studio prevents you from constructing circular dependencies using the **Project Dependencies** dialog.

### Finding the project build order

To display the project build order, do the following:

- From the **Project** menu, click **Build Order** to display the **Project Dependencies** dialog with the **Build Order** tab selected.
- The projects build in order from top to bottom.

If your target supports loading of multiple projects, then the **Build Order** also reflects the order in which projects are loaded onto the target. Projects will load, in order, from top to bottom. Generally, libraries need to be loaded before applications that use them, and you can ensure that this happens by making the application dependent upon the library. With this a dependency set, the library gets built before the application and loaded before the application.

Applications are deleted from a target in reverse build order, and as such applications are removed before the libraries that they depend upon.

## Project macros

You can use macros to modify the way that the project system refers to files. Macros are divided into four classes:

- **System Macros.** These are provided by the Studio application and are used to relay information from the environment, such as paths to common directories.
- **Global Macros.** These macros are saved in the environment and are shared across all solutions and projects. Typically, you would set up paths to library or external items here.
- **Project Macros.** These macros are saved in the project file as project properties and can define macro values specific to the solution/project they are defined in.
- **Build Macros.** These macros are generated by the project system whenever a build occurs.

## System macros

The following macro values are defined by CrossStudio itself:

Macro	Description
StudioDir	The install directory of the CrossStudio application.

System macros can be used in build properties and also for environment settings.

## Global macros

To define a global macro

- Select **Macros** from the **Project** menu.
- Click on the the **Global** tab.
- Set the macro using the syntax *name = replacement text*.

## Project macros

To define a project macro

- Select **Macros** from the **Project** menu.

- Click on the **Project** tab.
- Select the solution or project the macro should apply to.
- Set the macro using the syntax *name = replacement text*.

Alternatively you can set the project macros from the properties window:

- Select the appropriate solution/project in the Project Explorer.
- In the properties window, select the **Macros** property in the **General Options** group.
- Click on the the ellipsis button on the right.
- Set the macro using the syntax *name = replacement text*.

## Build macros

The following macro values are defined by the project system for a build of a given project node.

Macro	Description
ProjectDir	The project directory.
ProjectName	The project name.
Configuration	The selected build configuration.
SolutionDir	The directory containing the solution file.
SolutionName	The solution name.
InputFileName	The name of an input file relative to its project directory.
InputName	The name of an input file relative to its project directory without its extension.
InputExt	The extension of an input file.
IntDir	The macro-expanded value of the <b>Intermediate Directory</b> property.
OutDir	The macro-expanded value of the <b>Output Directory</b> property.
EXE	The default file extension for an executable file including the dot.
LIB	The default file extension for a library file including the dot.
OBJ	The default file extension for an object file including the dot.

Macro	Description
LibExt	A platform specific library extension that is generated based on project property values.

## Using macros

You can use a macro in a project property or an environment setting using the `$(macro)` syntax. For example the **Object File Name** property has a default value of `$(IntDir)/$(InputName)$(OBJ)`.

To enable debugging of builds you should use the **Build Information...** dialog that is on the context menu of the project explorer. This dialog will give a full list of the macros that are specified for the project node selected together with the macro expanded property values.



## Building projects

CrossStudio provides a facility to build projects in various configurations.

### Build configurations and their uses

Configurations are typically used to differentiate debug builds from release builds. For example, debug builds will have different compiler options to a release build: a debug build will set the options so that the project can be debugged easily, whereas a release build will enable optimization to reduce program size or increase its speed. Configurations have other uses; for example, you can use configurations to produce variants of software such as a library for several different hardware variants.

Configurations inherit properties from other configurations. This provides a single point of change for definitions that are common to configurations. A particular property can be overridden in a particular configuration to provide configuration-specific settings.

When a solution is created two configurations are generated, **Debug** and **Release**, and you can create additional configurations using **Build > Build Configurations**. Before you build, ensure that the appropriate configuration is set using **Project > Set Active Build Configuration** or alternatively the configuration box in the build tool bar. You should also ensure that the appropriate build properties are set in the properties window.

## Building your applications

When CrossStudio builds your application, it tries to avoid building files that have not changed since they were last built. It does this by comparing the modification dates of the generated files with the modification dates of the dependent files together with the modification dates of the properties that pertain to the build. If you are copying files then sometimes the modification dates may not be updated when the file is copied—in this instance it is wise to use the **Rebuild** command rather than the **Build** command.

You can see the build rationale CrossStudio is using by setting the **Environment Properties > Build Settings > Show Build Information** property and the build commands themselves by setting the **Environment Properties > Build Settings > Echo Build Command** property.

You may have a solution that contains several projects that are dependent upon each. Typically you might have several executable project and some library projects. The **Project > Dependencies** dialog specifies the dependencies between projects and to see the affect those dependencies have on the solution build order. Note that dependencies can be set on a per configuration basis but the default is for dependencies to be defined in the **Common** configuration.

You will also notice that new folders titled Dependencies has appeared in the project explorer. These folder contains the list of newly generated files and the files that they where generated from. These files can be decoded and displayed in the editor by right clicking on the file and seeing if it supports the **View** operation.

If you have the symbols window displayed then it will be updated with the symbol and section information of all executable files that have been built in the solution.

When CrossStudio builds projects it uses the values set in the properties window. To generalise your builds you can define macro values that are substituted when the project properties are used. These macro values can be defined globally at the solution and project level and can be defined on a per configuration basis. You can view and update the macro values using **Project > Macros**.

The combination of configurations, properties with inheritance, dependencies and macros provides a very powerful build management system. However, these systems can become complicated. To enable you to understand the implications of changing build settings, right clicking a node in the project explorer and selecting **Properties** brings up a dialog that shows the macros and build steps that apply to that project node.

### Building all projects

To build all projects in the solution, do one of the following:

- On the **Build** toolbar, click the **Build Solution** button.

—or—

- From the **Build** menu, select **Build Solution**.

—or—

- Type **Alt+F7**.

—or—

- Right click the solution in the **Project Explorer** window.
- From the menu, click **Build**.

## Building a single project

To build a single project only, do one of the following:

- Select the required project in the **Project Explorer**.
- On the **Build** tool bar, click the **Build** tool button.

—or—

- Select the required project in the **Project Explorer**.
- From the the **Project** menu, click **Build**.

—or—

- Right-click on the required project in the **Project Explorer** window.
- From the menu, click **Build**.

## Compiling a single file

To compile a single file, do one of the following:

- In the **Project Explorer**, right click the source file to compile.
- From the menu, click **Compile**.

—or—

- In the **Project Explorer**, click the source file to compile.
- From the **Build** menu, click **Compile**.

—or—

- In the **Project Explorer**, click the source file to compile.
- Type **Ctrl+F7**.

## Correcting errors after building

The results of a build are displayed in the **Build Log** in the **Output** window. Errors are highlighted in red, and warnings are highlighted in yellow. Double-clicking an error, warning, or note will move the cursor to the appropriate source line.

You can move forward and backward through errors using **Search > Next Location** and **Search > Previous Location**.

## Source code control

CrossWorks supports team development of applications using *source code control*. At present CrossWorks integrates with Microsoft Visual SourceSafe, SourceGear SourceOffSite 3.5.1, and CVS. The source code control integration capability provides:

- Connecting to the source control database (sometimes called a repository).
- Mapping files in the project system to those in the source code control system
- Showing the source control status of files and projects
- Adding and removing files and projects from source control
- Typical source control operations such as Add to source control, Remove from source control, and so on.

### Configuring source control

You need to configure CrossStudio to use source control in your projects. This section describes how to configure CrossStudio and your projects for source control.

#### Connecting to the source control system

Before you can check files in and out of source code control, you must connect to the source control system. To connect to the source control system, do the following:

- From the **Project** menu, click **Source Control** then **Connect...**

This displays a source control system specific dialog that enables you specify which source control database to connect to and to enter passwords etc. This dialog will reappear each time you load the solution to provide you with the opportunity to cancel source control connection.

### Mapping files

In order to map local files to those in the source control database, the project file is taken to be the root of the project hierarchy. The first time CrossWorks tries to check the source control status of the project file it will prompt you to specify the location of this file in the source control database. This mapping will be stored in the session file so you won't need to specify the mapping each time the project is loaded. If you cancel at the prompt to specify the location of a project file in the source control database, use **Project | Source Control | Add To Source Control** to make CrossWorks prompt again.

If a project directory is defined for a project file then this will be prepended to the filename in the project when mapping to files in the source control system. Note that only relative project directories (and filenames) are supported.

## Using source control

Once you have configured source control in CrossStudio, you can use the CrossStudio features to manipulate files in the source control system.

### Adding files to source control

To add a file to the source control system so that it can be controlled, checked in, checked out, and so on, do the following:

- In the **Project Explorer**, right click the file to add to source control.
- From the menu, click **Source Control** then **Add To Source Control**.

### Checking files out

To check a file out of the source control system, do the following:

- In the **Project Explorer**, right click the file to check out.
- From the menu, click **Source Control** then **Check Out**.

—or—

- In the **Project Explorer**, click the file to check out.
- From the **Project** menu, click **Source Control** then **Check Out**.

—or—

- In the **Project Explorer**, click the file to check out.
- On the **Source Control** tool bar, click the **Check Out** button.

## Checking files in

To check a file into the source control system, do the following:

- In the **Project Explorer**, right click the file to check in.
- From the menu, click **Source Control** then **Check In**.

—or—

- In the **Project Explorer**, click the file to check out.
- From the **Project** menu, click **Source Control** then **Check In**.

—or—

- In the **Project Explorer**, click the file to check out.
- On the **Source Control** tool bar, click the **Check In** button.

## Undoing check outs

To undo a check out and return a file on disk to its previous checked in state, do the following:

- In the **Project Explorer**, right click the file to undo the check out of.
- From the menu, click **Source Control** then **Undo Check Out**.

## Getting the latest version of a file

To retrieve the latest version of a file from source control, do the following:

- In the **Project Explorer**, right click the file to check out.
- From the menu, click **Source Control** then **Get Latest Version**.

## Showing the differences between files

To show the differences between the file on disk and the version checked into source control, do the following:

- In the **Project Explorer**, right click the file to show the differences of.
- From the menu, click **Source Control** then **Show Differences**.

## Removing a file from source control

To remove a file from being managed by the source control system, do the following:

- In the **Project Explorer**, right click the file to remove from source control.
- From the menu, click **Source Control** then **Remove From Source Control**.

Note that this deletes the file from the source code control system but does not touch the working file on disk and does not remove the file from the project.

## Source control properties

When a file is controlled, the **Properties** window shows the following properties in the **Source Control Options** group:

- **Checked Out.** If **Yes**, the file is checked out by you to the project location; if **No**, the file is not checked out.
- **Different.** If **Yes**, the checked out file differs from the one held in the source control system; if **No**, they are identical.
- **File Path.** The file path of the file in the source control system.
- **Old Version.** If **Yes**, the file in the project location is an old version compared to the latest version in the source control system.
- **Status. Controlled** indicates that the file is controlled by the source code control system.

## Source control status

By selecting **Project > Source Control > Show Status** a window is displayed that shows the current source control state of each file in the project. If a local file has been changed then this file is displayed in red. You can use this window to do multiple source control operations e.g. add several files to the source control. You can restrict the file list to a node in the project hierarchy e.g. all files of a folder, and supply a filter which enables the file list to be restricted to the source control status e.g. all files that are different.

When a given file or solution is selected in the project explorer, the source control properties appear in the properties window—these properties reflect the local checkout status of the file and whether or not it has been modified.



## Debug expressions

The debugger can evaluate simple expressions that can be subsequently displayed in the watch window or as a tool-tip in the code editor.

The simplest expression is an identifier which the debugger tries to interpret in the following order:

- an identifier that exists in the scope of the current context.
- the name of a global identifier in the program of the current context.

Numbers can be used in expressions, hexadecimal numbers must be prefixed with 0x.

Registers can be referenced by prefixing the register name with @.

The standard C and C++ operators `!`, `~`, `*`, `/`, `%`, `+`, `-`, `>>`, `<<`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `&`, `^`, `|`, `&&`, `||` are supported on numeric types.

The standard assignment operators `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `>>=`, `<<=`, `&=`, `|=`, `^=` are supported on number types.

The array subscript `[]` operator is supported on array and pointer types.

The structure access operator `.` is supported on structured types (this also works on pointers to structures) and the `->` works similarly.

The dereference operator (prefix `*`) is supported on pointers, the `sizeof` (prefix `&`) and `sizeof` operators are supported.

Casting to basic pointer types is supported. For example `(unsigned char *)0x300` can be used to display the memory at a given location.

Operators have the precedence and associativity that one would expect of a C like programming language.



## Basic editing

CrossStudio has a built-in editor which allows you to edit text, but has features that make it particularly well suited to editing code and is referred to as either the Text Editor or the Code Editor, based on its content.

You can open multiple code editors to view or edit source code in projects and copy and paste among them. The **Windows** menu contains a list of all open code editors.

The code editor supports the language of the source file that it is editing, showing code with syntax highlighting and offering smart indenting.

You can open a code editor in several ways, some of which are:

- By double clicking on a file in the **Project Explorer** or by right clicking on a file and selecting **Open** from the context menu.
- Using the **File > New** or **File > Open** commands.
- Right clicking in a source file and selecting a file from the **Open Include File** menu.

### Elements of the code editor

The code editor is divided into several elements which are described here.

- **Code Pane** The area where you edit your code. You can set options that affect the behavior of text in the code pane as it relates to indenting, tabbing, dragging and dropping of text, and so forth. For more information, see **General, All Languages, Text Editor, Options Dialog Box**.

- **Margin gutter** A gray area on the left side of the code editor where margin indicators such as breakpoints, bookmarks, and shortcuts are displayed. Clicking this area sets a breakpoint on the corresponding line of code. You can hide or display the Margin Indicator bar in General, Tools, Text Editor, Options dialog box.
- **Horizontal and vertical scroll bars** Allows you to scroll the code pane horizontally and vertically so that you can view the code that extends beyond the viewable edges of the code pane. You can hide or display the horizontal and vertical scroll bars using the General, Tools, Text Editor, Options dialog box.

## Navigation

There are several ways to move around code editors:

- Mouse and cursor motion keys
- Bookmarks
- The **Go To Line** command
- The **Navigate Backward** and **Navigate Forward** buttons

### Navigating with the mouse and keyboard

The most common way to navigate text is with the mouse and cursor motion keys:

- Click a location with the mouse.
- Use the arrow keys to move one character at a time, or the arrow keys in combination with the Ctrl key to move one word at a time.
- Use the scroll bars or scroll wheel on the mouse to move through the text.
- Use the **Home**, **End**, **PageUp**, and **PageDown** keys.
- Use **Alt+PageUp** and **Alt+PageDown** to move the insertion point to the top or bottom of the window, respectively.
- Use **Ctrl+Up** and **Ctrl+Down** to scroll the view without moving the insertion point.

The keystrokes most commonly used to navigate around a document are:

Keystroke	Description
Up	Moves the cursor up one line.
Down	Moves the cursor down one line.

Keystroke	Description
Left	Moves the cursor left one character.
Right	Moves the cursor right one character.
Home	Moves the cursor to the first character on the line. Pressing Home a second time moves the cursor to the first column.
End	Moves the cursor to the end of the line.
PageUp	Moves the cursor up one page.
PageDown	Moves the cursor down one page.
Ctrl+Left	Moves the cursor left one word.
Ctrl+Right	Moves the cursor right one word.
Ctrl+Up	Moves the cursor to the previous function.
Ctrl+Down	Moves the cursor to the next function.
Ctrl+Home	Moves the cursor to the start of the document.
Ctrl+End	Moves the cursor to the end of the document.
Alt+Up	Scrolls the document up by one line.
Alt+Down	Scrolls the document down by one line.

## Go To Line

To move the cursor to a particular line number, do one of the following:

- From the **Edit** menu, click **Advanced** then **Go To Line**.
- Enter the line number to move the cursor to.

—or—

- Type **Ctrl+G**, **Ctrl+L**.
- Enter the line number to move the cursor to.

## Selecting Text

### Selecting text with the keyboard

You can select text using the keyboard by using **Shift** with the navigation keys.

- Hold **Shift** key down while using the Navigation Keys.

### Selecting text with the mouse

- Move mouse cursor to the point in the document that you want to start selecting.

- Hold down left mouse button and drag mouse to mark selection.
- Release left mouse button to end selection.

## Matching Delimiters

The editor can find the matching partner for delimiter characters such as (), [], {}, <>.

### To match a delimiter

- Move cursor to the left of the delimiter character to be matched.
- Select **Edit | Advanced | Match Delimiter** menu item or use **Ctrl+] keys**.

### To select a delimited range

- Move cursor to the left of the delimiter character to be matched.
- Use **Ctrl+Shift+] keys**.

## Bookmarks

To edit a document elsewhere and then return to your current location, add a bookmark. The bookmarks presented in this section are *temporary bookmarks* and their positions are not saved when the file is closed nor when the solution is closed.

### Adding a bookmark

To add a temporary bookmark, move to the line you want to bookmark and do one of the following:

- On the **Text Edit** tool bar, click the **Toggle Bookmark** button.

—or—

- From the **Edit** menu, click **Bookmarks** then **Toggle Bookmark**.

—or—

- Type **Ctrl+F2**.

A temporary bookmark symbol appears next to the line in the indicator margin which shows that the bookmark has been set.

### Moving through bookmarks

To navigate forward through temporary bookmarks, do one of the following:

- On the **Text Edit** tool bar, click the **Next Bookmark** button.

—or—

- From the **Edit** menu, click **Bookmarks** then **Next Bookmark**.

—or—

- Type **F2**.

The editor moves the cursor to the next bookmark set in the document. If there is no following bookmark, the cursor is moved to the first bookmark in the document.

To navigate backward through temporary bookmarks, do one of the following:

- On the **Text Edit** tool bar, click the **Previous Bookmark** button.

—or—

- From the **Edit** menu, click **Bookmarks** then **Previous Bookmark**.

—or—

- Type **Shift+F2**.

The editor moves the cursor to the previous bookmark set in the document. If there is no previous bookmark, the cursor is moved to the last bookmark in the document.

### **Moving to the first or last bookmark**

To move to the first bookmark set in a document, do one of the following:

- From the **Edit** menu, click **Bookmarks** then **First Bookmark**.

—or—

- Type **Ctrl+K, F2**.

To move to the last bookmark set in a document, do one of the following:

- From the **Edit** menu, click **Bookmarks** then **Last Bookmark**.

—or—

- Type **Ctrl+K, Shift+F2**.

### **Removing bookmarks**

To remove a temporary bookmark, move to the line you want to remove the bookmark from and do one of the following:

- On the **Text Edit** tool bar, click the **Toggle Bookmark** button.

—or—

- From the **Edit** menu, click **Bookmarks** then **Toggle Bookmark**.

—or—

- Type **Ctrl+F2**.

The temporary bookmark symbol disappears which shows that the bookmark has been removed.

To remove all temporary bookmarks set in a document, do the following:

- From the **Edit** menu, click **Bookmarks** then **Clear All Bookmarks**.

—or—

- Type **Ctrl+Shift+F2**.

## Changing text

Whether you are editing code, HTML, or plain text, the Code Editor is just like many other text editors or word processors. For code that is part of a project, the project's programming language support provides syntax highlighting, colorization, indentation, and so on.

### Adding text

The editor has two text input modes:

- **Insertion mode** As text is entered it is inserted at the current cursor position and any text to the right of the cursor is shifted along. A visual indication of insertion mode is that the cursor is a flashing line.
- **Overstrike mode** As text is entered it replaces any text to the right of the cursor. A visual indication of overstrike mode is that the cursor is a flashing block.

Insert and overstrike modes are common to *all* editors: if one editor is in insert mode, *all* editors are set to insert mode. You can configure the cursor appearance in both insertion and overstrike modes using the **Tools > Options dialog** in the **Text Editor > General** pane.

#### Changing to insertion or overstrike mode

To toggle between insertion and overstrike mode, do the following:

- Press the **Insert** button to toggle between insert and overwrite mode.
- If overstrike mode is enabled, the **OVR** status indicator will be enabled and the overstrike cursor will be visible.

#### Adding or inserting text

To add or insert text, do the following:



- Either click somewhere in the document or move the cursor to the desired location.
- Enter the text.
- If your cursor is between existing characters, the text is inserted between them.

To overwrite characters in an existing line, press the **Insert** key to put the editor in Overstrike mode.

## Deleting text

The text editor supports the following common editing keystrokes:

Key	Description
Backspace	Deletes one character to the left of the cursor
Delete	Deletes one character to the right of the cursor
Ctrl+Backspace	Deletes one word to the left of the cursor
Ctrl+Delete	Deletes one word to the right of the cursor

### Deleting characters

To delete characters or a words in a line, do the following:

- Place the cursor immediately before the word or letter you want to delete.
- Press the **Delete** key as many times as needed to delete the characters or words.

—or—

- Place your cursor at the end of the letter or word you want to delete.
- Press the **Backspace** key as many times as needed to delete the characters or words.

**Note** You can double-click a word and then press **Delete** or **Backspace** to delete it.

### Deleting lines or paragraphs

To delete text which spans more than a few characters, do the following:

- Highlight the text you want to delete by selecting it. You can select text by holding down the left mouse button and dragging over the text, or by using the **Shift** key with the either the arrow keys or the **Home**, **End**, **Page Up**, **Page Down** keys.
- Press **Delete** or **Backspace**.

# Using the clipboard

## Copying text

To copy the selected text to the clipboard, do one of the following:

- From the **Edit** menu, select **Copy**.

—or—

- Type **Ctrl+C**.

—or—

- Type **Ctrl+Ins**.

To append the selected text to the clipboard, do the following:

- From the **Edit** menu, click **Clipboard** then **Copy Append**.

To copy whole lines from the current editor and place them onto the clipboard

- Select **Edit | Clipboard | Copy Lines** menu item.

To copy whole lines from the current editor and append them onto the end of the clipboard

- Select **Edit | Clipboard | Copy Lines Append** menu item.

To copy bookmarked lines from the current editor place them onto the clipboard

- Select **Edit | Clipboard | Copy Marked Lines** menu item.

To copy bookmarked lines from the current editor and append them onto the end of the clipboard

- Select **Edit | Clipboard | Copy Marked Lines Append** menu item.

## Cutting text

To cut the selected text to the clipboard, do one of the following:

- From the **Edit** menu, click **Cut**.

—or—

- Type **Ctrl+X**.

—or—

- Type **Shift+Del**.

To cut selected text from the current editor and append them onto the end of the clipboard

- Select **Edit | Clipboard | Cut Append** menu item.

To cut whole lines from the current editor and place them onto the clipboard

- Select **Edit | Clipboard | Cut Lines** menu item.

To cut whole lines from the current editor and append them onto the end of the clipboard

- Select **Edit | Clipboard | Cut Lines Append** menu item.

To cut bookmarked lines from the current editor and place them onto the clipboard

- Select **Edit | Clipboard | Cut Marked Lines** menu item.

To cut bookmarked lines from the current editor and append them onto the end of the clipboard

- Select **Edit | Clipboard | Cut Marked Lines Append** menu item.

### **Pasting text**

To paste text into current editor from clipboard, do one of the following:

- From the **Edit** menu, click **Paste**.

—or—

- Type **Ctrl+V**.

—or—

- Type **Shift+Ins**.

To paste text into a new editor from clipboard, do the following:

- From the **Edit** menu, click **Clipboard** then **Paste As New Document**.

### **Clearing the clipboard**

To clear the clipboard, do the following:

- From the **Edit** menu, click **Clipboard** then **Clear Clipboard**.

## Drag and drop editing

You can select text and then drag and drop it in another location. You can drag text to a different location in the same text editor or to another text editor.

### **Dragging and dropping text**

To drag and drop text, do the following:

- Select the text you want to move, either with the mouse or with the keyboard.

- Click on the highlighted text and keep the mouse button pressed.
- Move the mouse cursor to where you want to place the text.
- Release the mouse button to drop the text.

Dragging text moves it to the new location. You can copy the text to a new location by holding down the **Ctrl** key while moving the text: the mouse cursor changes to indicate a copy. Pressing the **Esc** key while dragging text will cancel a drag and drop edit.

### **Enabling drag and drop editing**

To enable or disable drag and drop editing, do the following:

- From the **Tools** menu, click **Options**.
- Under **Text Editor**, click **General**.
- In the **Editing** section, check **Drag/drop editing** to enable drag and drop editing or uncheck it to disable drag and drop editing.

## Undo and redo

The editor has an undo facility to undo previous editing actions. The redo feature can be used to re-apply previously undone editing actions.

### **Undoing one edit**

To undo one editing action, do one of the following:

- From the **Edit** menu, click **Undo**.

—or—

- On the **Standard** toolbar, click the **Undo** tool button.

—or—

- Type **Ctrl+Z** or **Alt+Backspace**.

### **Undoing multiple edits**

To undo multiple editing actions, do the following:

- On the **Standard** toolbar, click the arrow next to the **Undo** tool button.
- From the menu, select the editing operations to undo.

### **Undoing all edits**

To undo all edits, do one of the following:

- From the **Edit** menu, click **Advanced** then **Undo All**.

—or—

- Type **Ctrl+K, Ctrl+Z**.

### **Redoing one edit**

To redo one editing action, do one of the following:

- From the **Edit** menu, click **Redo**.

—or—

- On the **Standard** toolbar, click the **Redo** tool button.

—or—

- Type **Ctrl+Y** or **Alt+Shift+Backspace**.

### **Redoing multiple edits**

To redo multiple editing actions, do the following:

- On the **Standard** toolbar, click the arrow next to the **Redo** tool button.
- From the menu, select the editing operations to redo.

### **Redoing all edits**

To redo all edits, do one of the following:

- From the **Edit** menu, click **Advanced** then **Redo All**.

—or—

- Type **Ctrl+K, Ctrl+Y**.

## Indentation

The editor uses the **Tab** key to increase or decrease the indentation level. The indentation size can be altered in the editors **Language Properties** window.

### **Changing indentation size**

To change the indentation size, do the following:

- Select the **Properties Window**.
- Select the **Language Properties** pane.
- Set the **Indent Size** property for the required language.

The editor can optionally use tab characters to fill whitespace when indenting. The use of tabs for filling whitespace can be selected in the editors **Language Properties** window.

### Selecting tab or space fill when indenting

To enable or disable the use of tab characters when indenting, do the following:

- Select the **Properties Window**.
- Select the **Language Properties** pane.
- Set the **Use Tabs** property for the required language. Note that changing this setting does not add or remove existing tabs from files, the change will only effect new indents.

The editor can provide assistance with source code indentation while inserting text. There are three levels of indentation assistance:

- **None** The indentation of the source code is left to the user.
- **Indent** This is the default. The editor maintains the current indentation level. When **Return** or **Enter** is pressed, the editor automatically moves the cursor to the indentation level of the previous line.
- **Smart** The editor analyses the source code to compute the appropriate indentation level for the line. The number of lines before the current cursor position that are analysed for context can be altered. The smart indent mode can be configured to either indent open and closing braces or the lines following the braces.

### Changing indentation options

To change the indentation mode, do the following:

- Select the **Properties Window**.
- Select the **Language Properties** pane.
- Set the **Indent Mode** property for the required language.

To change whether opening braces are indented in smart indent mode, do the following:

- Select the **Properties Window**.
- Select the **Language Properties** pane.
- Set the **Indent Opening Brace** property for the required language.

To change whether closing braces are indented in smart indent mode, do the following:

- Select the **Properties Window**.
- Select the **Language Properties** pane.

- Set the **Indent Closing Brace** property for the required language.

### Changing indentation context

To change number of previous line used for context in smart indent mode, do the following:

- Select the **Properties Window**.
- Select the **Language Properties** pane.
- Set the **Indent Context Lines** property for the required language.

## File management

### To create a file

- Select **File | New | New File** menu item.

### Opening an existing file

To open an existing file, do one of the following:

- From the **File menu**, click **Open...**
- Choose the file to open from the dialog.
- Click **Open**.

—or—

- Type **Ctrl+O**.
- Choose the file to open from the dialog.
- Click **Open**.

### Opening multiple files

To open multiple existing files in the same directory, do one of the following

- Select **File | Open** menu item.
- Choose multiple files to open from the dialog. Hold down **Ctrl** key to add individual files or hold down **Shift** to select a range of files.
- Select **Open** from the dialog.

### Saving a file

To save a file, do one of the following:

- Activate the editor to save.
- From the **File** menu, click **Save**.

—or—

- Activate the editor to save.
- Type **Ctrl+S**.

—or—

- Right click the tab of the editor to save.
- From the popup menu, click **Save**.

### **Saving a file to a different name**

:To save a file, do one of the following:

- Select editor to save.
- From the **File** menu, click **Save As...**
- Enter the new file name and click **Save**.

—or—

- Right click the tab of the editor to save.
- From the popup menu, click **Save As...**
- Enter the new file name and click **Save**.

### **Printing a file**

:To print a file, do one of the following:

- Select editor to print.
- From the **File** menu, click **Print...**
- Select the printer to print to and click **OK**.

—or—

- Right click the tab of the editor to print.
- From the popup menu, click **Print...**
- Select the printer to print to and click **OK**.

### **To insert a file at the current cursor position**

- Select the editor to insert file into.
- Move the cursor to the required insertion point.



- Select **Edit | Insert File** menu item.
- Select file to insert.
- Click **Open** button.

#### **To toggle a files write permission**

- Select the editor containing the file.
- Select **Edit | Advanced | Toggle Read Only**.

## Find and replace

#### **To find text in a single file**

- Select **Edit | Find and Replace | Find...** menu item.
- Enter the string to be found in the **Find what** input.
- If the search will be case sensitive, set the **Match case** option.
- If the search will be for a whole word, i.e. there will be whitespace, the beginning or the end of line on either side of the string being searched for, set the **Match whole word** option.
- If the search string is a **Regular expressions** (page 83), set the **Use regular expression** option.
- If the search should move up the document from the current cursor position rather than down the document, set the **Search up** option.
- Click **Find** button to find next occurrence of the string or click **Mark All** to bookmark all lines in the file containing the string.

#### **To find text within a selection**

- Select text to be searched.
- Select **Edit | Find and Replace | Find...** menu item.
- Enter the string to be found in the **Find what** input.
- If the search will be case sensitive, set the **Match case** option.
- If the search will be for a whole word, i.e. there will be whitespace, the beginning or the end of line on either side of the string being searched for, set the **Match whole word** option.

- If the search string is a **Regular expressions** (page 83), set the **Use regular expression** option.
- If the search should move up the document from the current cursor position rather than down the document, set the **Search up** option.
- Click **Mark All** to bookmark all lines in the selection containing the string.

#### To find and replace text

- Select **Edit | Find and Replace | Replace...** menu item.
- Enter the string to be found in the **Find what** input.
- Enter the string to replace the found string with in the **Replace with** input. If the search string is a **Regular expressions** (page 83) then the `\n` backreference can be used in the replace string to reference captured text.
- If the search will be case sensitive, set the **Match case** option.
- If the search will be for a whole word, i.e. there will be whitespace, the beginning or the end of line on either side of the string being searched for, set the **Match whole word** option.
- If the search string is a **Regular expressions** (page 83), set the **Use regular expression** option.
- If the search should move up the document from the current cursor position rather than down the document, set the **Search up** option.
- Click **Find** button to find next occurrence of string and then **Replace** button to replace the found string with replacement string or click **Replace All** to replace all occurrences of the string without prompting.

#### To find text in multiple files

- Select **Edit | Find and Replace | Find in Files...** menu item.
- Enter the string to be found in the **Find what** input.
- Enter the wildcard to use to filter the files in the **In file types** input.
- Enter the folder to start search in the **In folder** input.
- If the search will be case sensitive, set the **Match case** option.
- If the search will be for a whole word, i.e. there will be whitespace, the beginning or the end of line on either side of the string being searched for, set the **Match whole word** option.
- If the search string is a **Regular expressions** (page 83), set the **Use regular expression** option.

- If the search will be carried out in the root folders sub-folders, set the **Look in subfolders** option.
- The output of the search results can go into two separate panes. If the output should go into the second pane, select **Output to pane 2** option.
- Click **Find** button.

## Regular expressions

The editor can search and replace text using regular expressions. A regular expression is a string that uses special characters to describe and reference patterns of text. The regular expression system used by the editor is modelled on Perl's regex language. For more information on regular expressions, see *Mastering Regular Expressions*, Jeffrey E F Freidl, ISBN 0596002890.

### Summary of special characters

The following table summarizes the special characters that the CrossStudio editor supports.

Characters	Meaning
\d	Match a numeric character.
\D	Match a non-numeric character.
\s	Match a whitespace character.
\S	Match a non-whitespace character.
\w	Match a word character.
\W	Match a non-word character.
[c]	Match set of characters, e.g. [ch] matches characters c or h. A range can be specified using the - character, e.g. [0-27-9] matches if character is 0, 1, 2, 7 8 or 9. A range can be negated using the ^ character, e.g. [^a-z] matches if character is anything other than a lower case alphabetic character.
\c	The literal character c. For example to match the character * you would use \*.
\a	Match ASCII bell character.
\f	Match ASCII form feed character.
\n	Match ASCII line feed character.

Characters	Meaning
<code>\r</code>	Match ASCII carriage return character.
<code>\t</code>	Match ASCII horizontal tab character.
<code>\v</code>	Match ASCII vertical tab character.
<code>\xhhhh</code>	Match Unicode character specified by hexadecimal number <i>hhhh</i> .
<code>.</code>	Match any character.
<code>*</code>	Match zero or more occurrences of the preceding expression.
<code>+</code>	Match one or more occurrences of the preceding expression.
<code>?</code>	Match zero or one occurrences of the preceding expression.
<code>{n}</code>	Match <i>n</i> occurrences of the preceding expression.
<code>{n,}</code>	Match at least <i>n</i> occurrences of the preceding expression.
<code>{,m}</code>	Match at most <i>m</i> occurrences of the preceding expression.
<code>{n,m}</code>	Match at least <i>n</i> and at most <i>m</i> occurrences of the preceding expression.
<code>^</code>	Beginning of line.
<code>\$</code>	End of line.
<code>\b</code>	Word boundary.
<code>\B</code>	Non-word boundary.
<code>(e)</code>	Capture expression <i>e</i> .
<code>\n</code>	Backreference to <i>n</i> th captured text.

## Examples

The following regular expressions can be used with the editor's search and replace operations. To use the regular expression mode the **Use regular expression** check box must be set in the search and replace dialog. Once enabled, the regular expressions can be used in the **Find what** search string. The **Replace with** strings can use the "`\n`" backreference string to reference any captured strings.

"Find what" String	"Replace with" String	Description
<code>u\w.d</code>		Search for any length string containing one or more word characters beginning with the character 'u' and ending in the character 'd'.
<code>^.*;\$</code>		Search for any lines ending in a semicolon.
<code>(typedef.+\\s+)(\\S+);</code>	<code>\\1TEST_\\2;</code>	Find C type definition and insert the string "TEST" onto the beginning of the type name.

# Advanced editor features

## Code Templates

The editor provides the ability to use code templates. A code template is a block of frequently used source code that can be inserted automatically by using a particular key sequence. A `|` character is used in the template to indicate the required position of the cursor after the template has been expanded.

### To view code templates

- Select **Edit > Advanced > View Code Templates** menu item.

Code templates can either be expanded manually or automatically when the **Space** key is pressed.

### To expand a code template manually

- Type a key sequence, for example the keys **c** followed by **b** for the comment block template.
- Select **Edit > Advanced > Expand Template** or type **Ctrl+J** to expand the template.

### To expand the template automatically

- Ensure the **Expand Templates On Space** editor property is enabled.
- Type a key sequence, for example the keys **c** followed by **b** for the comment block template.
- Now type **Space** key to expand the template.

## Editing Macros

The editor has a number of built-in macros for carrying out common editing actions.

### To declare a type

- Select **Edit > Editing Macros > Declare Or Cast To** menu item for required type.

### To cast to a type

- Select text in the editor containing expression to cast.
- Select **Edit > Editing Macros > Declare Or Cast To** menu item for required type cast.

### **To insert a qualifier**

- Select **Edit > Editing Macros > Insert** menu item for required qualifier.

## Tab Characters

The editor can either use tab characters or only use space characters to fill whitespace. The use of tabs or spaces when indenting can be specified in the editors language properties. The editor can also add or remove tabs characters in blocks of selected text.

### **To replace spaces with tab characters in selected text**

- Select text.
- Select **Edit > Advanced > Tabify Selection** menu item

### **To replace tab characters with spaces in selected text**

- Select text.
- Select **Edit > Advanced > Untabify Selection** menu item

## Changing Case

The editor can change the case of selected areas of text.

### **To change case of selected text to uppercase**

- Select text.
- Select **Edit > Advanced > Make Selection Uppercase** menu item.

### **To change case of selected text to lowercase**

- Select text.
- Select **Edit > Advanced > Make Selection Lowercase** menu item.

## Commenting

The editor can add or remove language specific comment characters to areas of text.

### **To comment out an area of selected text**

- Select text to comment out.
- Select **Edit > Advanced > Comment** menu item.

### **To uncomment an area of selected text**

- Select text to remove comment characters from.

- Select **Edit > Advanced > Uncomment** menu item.

## Indentation

The editor can increase or decrease the indentation level of an area of selected text.

### To increase indentation of selected text

- Select text.
- Select **Edit > Advanced > Increase Line Indent** menu item.

### To decrease indentation of selected text

- Select text.
- Select **Edit > Advanced > Decrease Line Indent** menu item.

## Sorting

The editor can sort areas of selected text in ascending or descending ASCII order.

### To sort selected lines into ascending order

- Select text to sort.
- Select **Edit > Advanced > Sort Ascending** menu item.

### To sort selected lines into descending order

- Select text to sort.
- Select **Edit > Advanced > Sort Descending** menu item.

## Text Transposition

The editor can transpose word or line pairs.

### To transpose the word at the current cursor position with the previous word

- Select **Edit > Advanced > Transpose Words** menu item.

### To transpose the current line with the previous line

- Select **Edit > Advanced > Transpose Lines** menu item.

## Whitespace

### To make whitespace visible

- Select **Edit > Advanced > Visible Whitespace** menu item.

## Code templates

The editor provides the ability to use code templates. A code template is a block of frequently used source code that can be inserted automatically by using a particular key sequence. A `|` character is used in the template to indicate the required position of the cursor after the template has been expanded.

### Editing code templates

To edit code templates, do the following:

- From the **Edit** menu, click **Advanced** then **View Code Templates**.

Code templates can either be expanded manually or automatically when the **Space** key is pressed.

### Manually expanding a template

To expand a code template manually, do the following:

- Type a key sequence, for example the keys **c** followed by **b** for the comment block template.
- From the **Edit** menu, click **Advanced** then **Expand Template** or type **Ctrl+J** to expand the template.

### Automatically expanding templates

To expand the template automatically, do the following:

- Ensure the **Expand Templates On Space** editor property is enabled.
- Type a key sequence, for example the keys **c** followed by **b** for the comment block template.
- Now type **Space** key to expand the template.



## Memory map editor

Memory map files are tree structured descriptions of the target memory map. Memory map files are used by the compiler to ensure correct placement of program sections. Memory map files are used by the debugger so that it knows which memory addresses are valid on the target and which program sections to load. You can also use the memory files to direct the debugger to display memory mapped peripherals. Usually you don't need to modify memory map files—they will be set up for the particular targets that CrossStudio supports but it is useful to view them with the memory map editor.

You can open memory map files using **File > Open** and selecting the XML file that contains the memory map or alternatively using the **View Memory Map** option on the context menu of the Project Explorer.

The memory map editor provides a tree structured view of the memory space of a target. The memory map consists of a set of different node types that are arranged in a hierarchy. These nodes have properties that can be modified using the properties window when the node is selected. These properties and the placement of nodes within the memory map are used as input to the program building process so that the linker knows where sections should be placed. Additionally the debugger uses the information in memory map files to enable register display and memory display.

The memory map editor supports the following node types:

- **Root.** The top most node of the memory map.
- **Memory Segment.** A range of addresses that represents a memory region in the target.
- **Program Section.** Represents a program section of your application.

- **Register Group.** Represents an area of memory that contains a group of related registers.
- **Register.** Represents a memory mapped register.
- **Bit Field.** Part of a memory mapped register.

The following statements hold regarding the creation and movement of nodes within a memory

- Memory segments can be within the **Root** segment.
- Program sections must be within a memory segment.
- Register groups can be within the Root or within a memory segment.
- Registers can be within memory segments or register groups.
- Bit Fields can be within registers.

All nodes have mandatory and optional properties associated with them. All nodes have a mandatory Name property. This name should be unique within the memory map.

#### **Memory segment and register group properties**

- **Start Address.** A hexadecimal number stating where the memory begins (lowest address).
- **Start Address Symbol.** The name of a linker symbol to generate that has the value of the Start Address.
- **Size.** A hexadecimal number that defines the size in bytes of the memory segment.
- **Size Symbol.** The name of a linker symbol to generate that has the value of the Size.
- **Access Type.** Specifies if the memory segment is read only or read/write.

#### **Program section properties**

- **Start Address.** An optional hexadecimal value that is the absolute load position of the section. If this isn't set then the relative placement of the program section within the memory segment will determine the load position of the section.
- **Size.** An optional decimal value that is the size in bytes of the program section.
- **Load.** Specifies whether or not the section should be loaded by the debugger.
- **Alignment.** An optional decimal value that specifies the alignment requirements of the section.
- **Section To Run In.** An optional name of another program section that this program section will be copied to.

- **Input Section Names.** The optional names of the files that will be placed into this section.

#### **Register properties**

- **Start Address.** A hexadecimal value specifying where the register is placed.
- **Start Address Symbol.** The name of a linker symbol to generate that has the value of the Start Address.
- **Register Type.** Optional, a C type specifying how you want the register to be displayed. The defaults to the word length of the target processor.
- **Endian.** Optional, specifies the byte order of a multibyte register. This defaults to the byte order of the target processor.

#### **Bitfield properties**

- **Bit Offset.** A decimal value that is the starting bit position of the bit field. Bit 0 is the first bit position.
- **Bit Length.** A decimal value that defines the number of bits in the field.

The editor has many of the attributes of the text editor and the same key-bindings for example cut, copy and paste are all accessible from the **Edit** menu. In addition to the standard editor capabilities the memory map editor supports the movement up and down of nodes within a hierarchy. This enables the sequencing of program sections to be achieved.



## Package management

CrossWorks can have additional target support functionality installed and removed using packages. Packages are similar to libraries (files containing other files) and have CrossWorks specific functionality to support package browsing and new project creation.

### Installing packages

The package you wish to install should be saved in the  $\$(StudioDir)/packages$  directory.

- From the **Tools** menu, click **Install Package...**
- Select the package file from the file chooser dialog.
- If you are updating an existing installed package that has files that have been modified then you will be prompted before these files are updated.
- When the installation is complete the installed packages page will be displayed.

You should leave the package in the  $\$(StudioDir)/packages$  directory in case you wish to reinstall or remove the package.

### Removing installed packages

The package you wish to remove should be in the  $\$(StudioDir)/packages$  directory.

- From the **Tools** menu, click **Remove Package...**
- Select the package file from the file chooser dialog.
- You will be prompted if any files that are to be removed have been modified.

- When the package has been removed the installed packages page will be displayed.

#### **Viewing installed packages**

- From the **Tools** menu, click **View Packages**.
- The set of installed packages will be displayed in the web browser.
- Select one of the html links to see the contents of the package.

## Linking and Section placement

Executable programs consists of a number of program sections. Typically there will be program sections for code, initialised data and zero'd data. There will often be more than one code section and these will require placement at specific addresses in memory.

To describe how the program sections of your program are positioned in memory the CrossWorks project system uses a **Memory Map file format** (page 337) and a **Section Placement file format** (page 339). These files are both xml files and can be edited either with the text editor or with the built-in memory map editor. The memory map file specifies the start address and size of memory segments of the target. The section placement file specifies where to place program sections in the memory segments of the target. Separating the memory map from the section placement scheme enables a single hardware description to be shared across projects and also enables a project to be built for a variety of hardware descriptions.

For example a memory map file representing a device with two memory segments called **FLASH** and **SRAM** would look something like this in the memory map editor.

```
ROOT
  FLASH 0x40000000 0x1000
  SRAM  0x00000000 0x1000
```

A corresponding section placement file will refer to the memory segments of the memory map file and list the sections that are to be placed in those segments. This is done using a memory segment name in the section placement file that matches a memory segment name in the memory map file.

For example a section placement file that places a section called **.stack** in the **SRAM** segment and the **.vectors** and **.text** section into the **FLASH** segment would look like this in the memory map editor.

```

ROOT
  FLASH
    .text
    .vectors
  SRAM
    .stack

```

Note that the order of section placement within a segment is from the bottom up when viewed in the memory map editor, in this example **.vectors** is placed before **.text**.

The memory map file and section placement file to use for linkage can either be included as a part of the project or alternatively they can be specified the **Linker Properties** (page 367) of the project.

You can create a new program section using the either the assembler or the compiler. For the C/C++ compiler this can be achieved using **\_\_attribute\_\_** on declarations. For example:

```
void foobar(void) __attribute__ ((section(".foo")));
```

This will allocate **foobar** in the section called **.foo**. Alternatively you can specify the section names of the code, constant, data and zero'd data for an entire compilation unit using thesection options properties.

You can place the section into the section placement file using the memory map editor. If you are modifying a section placement file that is supplied in the CrossWorks distribution you will need to import it into your project using the project explorer.

With the section placement file in the memory map editor you can right click on the memory segment in which you wish to place the section and select **New Program Section** from the context menu. Right click on the new program section and select **Properties** from the context menu. In the properties window you can specify the **Name** which should correspond to the name used when you created the section. You will also need to specify the **Input Sections** property to match the name. In the simplest case this will be the **\*(Name)**. Using the context menu you can move the section within the segment to specify the order in which it should be placed.

Sections containing code and constant data should have their **Load** property set to be "Yes". There are sections that don't require any loading such as stack sections and zero'd data sections, these sections should have the **Load** property set to "No".

For example initialised data is loaded into the section **.data\_load** and then copied into the **.data\_run** section.

```

ROOT
  FLASH
    .data_load Load=Yes, Section To Run In=.data_run
    .text
    .vectors
  SRAM
    .data_run Load=No
    .stack

```



The startup code will need to copy the contents of the **.data\_load** section to the **.data\_run**. To enable this symbols for each section marking the start and end addresses are generated. Each section will have a start symbol generated called `__section name_start__` and an end symbol generated called `__section name_end__`. These symbols can be used to copy the sections from their load positions to their run positions

For example the **.data\_load** section can be copied to the **data\_run** section using the following call to `memcpy`.

```
memcpy(__data_run_start__,
        __data_load_start__,
        __data_load_end__ - __data_load_start__);
```

If you create a section that needs to be copied then you will need to modify the startup code to do the initialisation.



## CrossStudio Windows

This section is a reference to each of the windows in the CrossStudio environment.

### In this section

- **Breakpoints window (page 102).** Describes how to use the breakpoints window to manage breakpoints in a program.
- **Call stack window (page 107).** Describes how to traverse the call stack to examine data and where function calls came from.
- **Clipboard ring window (page 100).** Describes how to use the clipboard ring to make complex cut-and-pastes easier.
- **Execution counts window (page 111) and Trace window (page 122).** Describes how to gather useful profiling statistics on your application on the simulator and targets that support execution profiling and tracing.
- **Globals window (page 112), Locals window (page 114), and Watch window (page 122).** Describes how to examine your application's local and global variables and how to watch specific variables.
- **Memory window (page 116).** Describes how to look at target memory in raw hexadecimal form.
- **Register windows (page 118).** Describes how to examine processor registers and peripherals defined by the project's memory map file.
- **Threads window (page 120).** Describes how CrossStudio can display thread-local data, tasks and objects when you run your application under a real-time operating system.
- **Help window (page 125).** Describes how the CrossStudio help system works and how to get answers to your questions.

- **Output window (page 127).** Describes the output window and the logs it contains.
- **Project dependencies and build order (page 53).** Describes the project explorer and how to manage your projects.
- **Properties window (page 130).** Describes the property window and how to change environment and project properties using it.
- **Source code control (page 61).** Describes how to use the Source Navigator to easily browse your project's functions, methods, and variable.
- **Symbol browser (page 133).** Describes how you can use the Symbol browser to find out how much code and data your application requires.
- **Targets window (page 139).** Describes how to manage your target connections by creating new ones, editing existing ones, and deleting unused ones.

## Clipboard ring window

The code editor captures all **Cut** and **Copy** operations and stores the the cut or copied item on the **Clipboard Ring**. The clipboard ring stores the last 20 text items that were cut or copied, but you can configure the maximum number of items stored on the clipboard ring using the environment options dialog. The clipboard ring is an excellent place to store scraps of text when you're working with many documents and need to cut and paste between them.

### Showing the clipboard ring

To display the **Clipboard Ring** window if it is hidden, do one of the following:

- From the **View** menu, click **Clipboard Ring**.

—or—

- Type **Ctrl+Alt+C**.

—or—

- Right click the tool bar area to display the **View** menu.
- From the popup menu, click **Clipboard Ring**.

### Pasting an item by cycling the clipboard ring

To paste from the clipboard ring, do the following:

- Cut or copy some text from your code. The last item you cut or copy into the clipboard ring is the current item for pasting.
- Type **Ctrl+Shift+V** to paste the clipboard ring's current item to the current document.

- Repeatedly type **Ctrl+Shift+V** to cycle through the entries in the clipboard ring until you get to the one you want to permanently paste in the document. Each time you press **Ctrl+Shift+V**, the editor replaces the last entry you pasted from the clipboard ring so that you end up with only the last one you selected. The item you stop on then becomes the current item.
- Move to another location or cancel the selection. You can use **Ctrl+Shift+V** to paste the current item again or cycle the clipboard ring to a new item.

Clicking an item in the clipboard ring makes it the current item.

### **Pasting a specific item into a document**

To paste an item on the clipboard ring directly into the current document, do one of the following:

- Move the cursor to the position where you want to paste the item into the document.
- Display the dropdown menu of the item to paste by clicking the arrow to its right.
- From the menu, click **Paste**.

—or—

- Make the item you want to paste the current item by clicking it.
- Move the cursor to the position where you want to paste the item into the document.
- Type **Ctrl+Shift+V**.

### **Pasting all items into a document**

To paste all items on the clipboard ring into the current document, move the cursor to the position where you want to paste the items into the document and do one of the following:

- From the **Edit** menu, click **Clipboard Ring** then **Paste All**.

—or—

- On the **Clipboard Ring** tool bar, click the **Paste All** button.

—or—

- Type **Ctrl+R**, **Ctrl+V**.

### **Removing a specific item from the clipboard ring**

To remove an item from the clipboard ring, do the following:

- Display the dropdown menu of the item to delete by clicking the arrow at the right of the item.
- From the menu, click **Delete**.

#### **Removing all items from the clipboard ring**

To remove all items from the clipboard ring, do one of the following:

- From the **Edit** menu, click **Clipboard Ring** then **Clear Clipboard Ring**.

—or—

- On the **Clipboard Ring** tool bar, click the **Clear Clipboard Ring** button.

—or—

- Type **Ctrl+R, Delete**.

#### **Configuring the clipboard ring**

To configure the clipboard ring, do the following:

- From the **Tools** menu, select **Options**.
- Under **Environment**, select **Even More...**
- Check **Preserve Contents** to save the content of the clipboard ring between runs, or uncheck it to start with an empty clipboard ring.
- Change **Maximum Items** to configure the maximum number of items stored on the clipboard ring.

## Build log window

The Build window contain the results of the last build, it is cleared on each rebuild.

If there are any errors in the build then they are displayed in red. Clicking on such a line will locate the editor to the errant source line.

The command lines used to do the build can be echoed to the build log using the tools/options/build/echo checkbox.

## Breakpoints window

The **Breakpoints** window manages the list of currently set breakpoints on the solution. Using the breakpoint window you can:

- Enable, disable and delete existing breakpoints.
- Add new breakpoints.
- Show the status of existing breakpoints.
- Chain breakpoints together.









Breakpoints are stored in the session file so they will be remembered each time you work on a particular project. When running in the debugger, you can set breakpoints on assembly code addresses. These low-level breakpoints appear in the breakpoint window for the duration of the debug run but are not saved when you stop debugging.

When a breakpoint is hit then the matched breakpoint will be highlighted in the breakpoint window.

## Breakpoints window layout

The **Breakpoints** window is divided into a tool bar and the main breakpoint display.





### The Breakpoint tool bar

Button	Description
	Creates a new breakpoint using the <b>New Breakpoint</b> dialog.
	Toggles the selected breakpoint between enabled and disabled states.
	Removes the selected breakpoint.
	Moves the cursor to the statement that the selected breakpoint is set at.
	Deletes all breakpoints.
	Disables all breakpoints.
	Enables all breakpoints.
	Creates a new breakpoint group and makes it active.

### The Breakpoints window display

The main part of the **Breakpoints** window displays the breakpoints that have been set and what state they are in. You can organize breakpoints into folders, called *breakpoint groups*.

CrossStudio displays these icons to the left of each breakpoint:

Icon	Description
	<b>Enabled breakpoint</b> An enabled breakpoint will stop your program running when the breakpoint condition is met.
	<b>Disabled breakpoint</b> A disabled breakpoint will not stop the program when execution passes through it.
	<b>Invalid breakpoint</b> An invalid breakpoint is one where the breakpoint cannot be set, for example there is no executable code associated with the source code line where the breakpoint is set or the processor does not have enough hardware breakpoints.
	<b>Chained breakpoint</b> The breakpoint is linked to its parent and is enabled when its parent is hit.

### Showing the Breakpoints window

To display the **Breakpoints** window if it is hidden, do one of the following:

- From the **View** menu, click **Other Windows** then **Breakpoints**.
- or—
- From the **Debug** menu, click **Debug Windows** then **Breakpoints**.
- or—
- Type **Ctrl+Alt+B**.
- or—
- On the **Debug** tool bar, click the **Breakpoints** icon.
- or—
- Right click the tool bar area to display the **View** menu.
- From the popup menu, click **Other Windows** then **Breakpoints**.

## Managing single breakpoints

You can manage breakpoints in the **Breakpoint** window.

### Deleting a breakpoint

To delete a breakpoint, do the following:

- In the **Breakpoints** window, click the breakpoint to delete.
- From the **Breakpoints** window tool bar, click the **Delete Breakpoint** button.



### Editing a breakpoint

To edit the properties of a breakpoint, do the following:

- In the **Breakpoints** window, right click the breakpoint to edit.
- From the popup menu, click **Edit Breakpoint**.
- Edit the breakpoint in the Breakpoint dialog.

### Enabling or disabling a breakpoint

To toggle the enable state of a breakpoint, do one of the following:

- In the **Breakpoints** window, right click the breakpoint to enable or disable.
- From the popup menu, click **Enable/Disable Breakpoint**.

—or—

- In the **Breakpoints** window, click the breakpoint to enable or disable.
- Type **Ctrl+F9**.

## Chaining breakpoints

You can chain breakpoints together using the **Chain Breakpoint From** dialog. When a breakpoint is chained from another breakpoint it will not be hit until the breakpoint it has been chained from has been hit. Note that when a breakpoint is chained to another breakpoint then that breakpoint will not stop your application executing it is there simply to activate the breakpoint (actually breakpoints) it is chained to.

Chained breakpoints have the breakpoint they are chained from displayed as child nodes in the tree display you can remove the chain with the right click context menu.

Note that when you delete or disable a breakpoint that other breakpoints are chained from then those breakpoints are always activated. The chain will also remain in case you wish to reset it.

## Managing breakpoint groups

Breakpoints are divided into *breakpoint groups*. You can use breakpoint groups to specify sets of breakpoints that are applicable to a particular project in the solution or for a particular debug scenario. Initially there is a single breakpoint group, named **Default**, to which all new breakpoints are added.

### Creating a new breakpoint group

To create a new breakpoint group, do one of the following:

- From the **Breakpoints** window tool bar, click the **New Breakpoint Group** button.

—or—

- From the **Debug** menu, click **Breakpoints** then **New Breakpoint Group**.

—or—

- Right click anywhere in the **Breakpoints** window.
- From the popup menu, click **New Breakpoint Group**.

In the **New Breakpoint Group Dialog**, enter the name of the breakpoint group.

### **Selecting a new active breakpoint group**

When you create a breakpoint, it is added to the active breakpoint group. To make a group the active group, do the following:

- In the **Breakpoints** window, click the breakpoint group to make active.
- From the popup menu, click **Set as Active Group**.

### **Deleting a breakpoint group**

To delete a breakpoint group, do the following:

- In the **Breakpoints** window, right click the breakpoint group to delete.
- From the popup menu, click the **Delete Breakpointt Group** button.

### **Enabling all breakpoints in a breakpoint group**

You can enable all breakpoints within a group as a whole. To enable all breakpoints in a group, do the following:

- In the **Breakpoints** window, right click the breakpoint group to enable.
- From the popup menu, click **Enable Breakpoint Group**.

### **Disabling all breakpoints in a breakpoint group**

You can disable all breakpoints within a group as a whole. To disable all breakpoints in a group, do the following:

- In the **Breakpoints** window, right click the breakpoint group to disable.
- From the popup menu, click **Disable Breakpoint Group**.

## **Managing all breakpoints**

You can delete, enable, or disable all breakpoints.

### **Deleting all breakpoints**

To delete all breakpoints, do one of the following:

- From the **Debug** menu, click **Breakpoints** then **Delete All Breakpoints**.

—or—

- From the **Breakpoints** window tool bar, click the **Delete All Breakpoints** button.

—or—

- Type **Ctrl+Shift+F9**.

### **Enabling all breakpoints**

To enable all breakpoints, do one of the following:

- From the **Debug** menu, click **Breakpoints** then **Enable All Breakpoints**.

—or—

- From the **Breakpoints** window tool bar, click the **Enable All Breakpoints** button.

### **Disabling all breakpoints**

To disable all breakpoints, do one of the following:

- From the **Debug** menu, click **Breakpoints** then **Disable All Breakpoints**.

—or—

- From the **Breakpoints** window tool bar, click the **Disable All Breakpoints** button.







## Call stack window

The **Call Stack** window displays the list of function calls (stack frames) that are active at the point that program execution halted. When program execution halts, CrossStudio populates the call stack window from the active (currently executing) task. For simple single-threaded applications not using the CrossWorks tasking library there is only a single task, but for multi-tasking programs that do use the CrossWorks Tasking Library there may be any number of tasks. CrossStudio updates the **Call Stack** window when you change the active task in the **Threads window** (page 120).

### Call Stack user interface

The **Call Stack** window is divided into a tool bar and the main breakpoint display.




## Call Stack tool bar

Button	Description
	Moves the cursor to where the call to the selected frame was made.
	Sets the debugger context to the selected stack frame.
	Moves the debugger context down one stack to the called function
	Moves the debugger context up one stack to the calling function
	Selects the fields to display for each entry in the call stack.
	Sets the debugger context to the most recent stack frame and moves the cursor to the currently executing statement.

## Call Stack display

The main part of the **Call Stack** window displays each unfinished function call (active stack frame) at the point that program execution halted. The most recent stack frame is displayed at the bottom of the list and the eldest is displayed at the top of the list.

CrossStudio displays these icons to the left of each function name:

Icon	Description
	Indicates the stack frame of the current task.
	Indicates the stack frame selected for the debugger context.
	Indicates that a breakpoint is active and when the function returns to its caller.

These icons can be overlaid to show, for instance, the debugger context and a breakpoint on the same stack frame.

## Showing the Call Stack window

To display the **Call Stack** window if it is hidden, do one of the following:

- From the **View** menu, click **Other Windows** then **Call Stack**.

—or—

- From the **Debug** menu, click **Debug Windows** then **Call Stack**.

—or—

- Type **Ctrl+Alt+S**.

—or—

- On the **Debug** tool bar, click the **Call Stack** icon.

—or—

- Right click the tool bar area to display the **View** menu.
- From the popup menu, click **Other Windows** then **Breakpoints**.

## Configuring the Call Stack window

Each entry in the **Call Stack** window displays the function name and, additionally, parameter names, types, and values. You can configure the **Call Stack** to display varying amounts of information for each stack frame. By default, CrossStudio displays all information.

### Displaying or hiding parameter names

To display or hide the name of each parameter in the call stack, do the following:

- On the **Call Stack** tool bar, click the **Fields** button.
- From the dropdown menu, check or uncheck **Parameter Names**.

### Displaying or hiding parameter values

To display or hide the value of each parameter in the call stack, do the following:

- On the **Call Stack** tool bar, click the **Fields** button.
- From the dropdown menu, check or uncheck **Parameter Value**.

### Displaying or hiding parameter types

To display or hide the type of each parameter in the call stack, do the following:

- On the **Call Stack** tool bar, click the **Fields** button.
- From the dropdown menu, check or uncheck **Parameter Types**.

### Displaying or hiding file names and source line numbers

To display or hide the file name and source line number columns of each frame in the call stack, do the following:

- On the **Call Stack** tool bar, click the **Fields** button.
- From the dropdown menu, check or uncheck **Call Source Location**.

### Displaying or hiding call addresses

To display or hide the call address of each frame in the call stack, do the following:

- On the **Call Stack** tool bar, click the **Fields** button.
- From the dropdown menu, check or uncheck **Call Address**.

## Changing the debugger context

You can select the stack frame for the debugger context from the **Call Stack** window.

### Selecting a specific stack frame

To move the debugger context to a specific stack frame, do one of the following:

- In the **Call Stack** window, double click the stack frame to move to.

—or—

- In the **Call Stack** window, click the stack frame to move to.
- On the **Call Stack** window's tool bar, click the **Switch To Frame** button.

—or—

- In the **Call Stack** window, right click the stack frame to move to.
- From the popup menu, select **Switch To Frame**.

The debugger moves the cursor to the statement where the call was made. If there is no debug information for the statement at the call location, CrossStudio opens a disassembly window at the instruction.

### Moving up one stack frame

To move the debugger context up one stack frame to the calling function, do one of the following:

- On the **Call Stack** window's tool bar, click the **Up One Stack Frame** button.

—or—

- On the **Debug Location** tool bar, click the **Up One Stack Frame** button.

—or—

- Type **Alt+-**.

The debugger moves the cursor to the statement where the call was made. If there is no debug information for the statement at the call location, CrossStudio opens a disassembly window at the instruction.

### Moving down one stack frame

To move the debugger context down one stack frame to the called function, do one of the following:

- On the **Call Stack** window's tool bar, click the **Down One Stack Frame** button.

—or—

- On the **Debug Location** tool bar, click the **Down One Stack Frame** button.

—or—

- Type **Alt++**.

The debugger moves the cursor to the statement where the call was made. If there is no debug information for the statement at the call location, CrossStudio opens a disassembly window at the instruction.

### **Setting a breakpoint on a return to a function**

To set a breakpoint on return to a function, do one of the following:

- In the **Call Stack** window, click the stack frame on the function to stop at when it is returned to.
- From the **Build** tool bar, click the **Toggle Breakpoint** button.

—or—

- In the **Call Stack** window, click the stack frame on the function to stop at when it is returned to.
- Type **F9**.

—or—

- In the **Call Stack** window, right click the function to stop at when it is returned to.
- From the popup menu, click **Toggle Breakpoint**.

## Execution counts window

The Execution Counts window shows a list of source locations and the number of times those source locations have been executed. This window is only available for targets that support the collection of jump trace information.

The count value displayed is the number of times the first instruction of the source code location has been executed. The source locations displayed are target dependent - they could represent each statement of the program or each jump target of the program. If however the debugger is in intermixed or disassembly mode then the count values will be displayed on a per instruction basis.

The execution counts window is updated each time your program stops and the window is visible so if you have this window displayed then single stepping may be slower than usual.

The counts window can be sorted by any column (counts, source file, or function name) by clicking on the appropriate column header. Double clicking on an entry will locate the source display to the appropriate source code location.

## Globals window










The globals window displays a list of all variables that are global to the program. The operations available on the entries in the globals window are the same as the **Watch window** (page 122) except that variables cannot be added to or deleted from the globals window.

### Globals window user interface

The **Globals** window is divided into a tool bar and the main data display.



## Globals tool bar

Button	Description
	Displays the selected item in binary.
	Displays the selected item in octal.
	Displays the selected item in decimal.
	Displays the selected item in hexadecimal.
	Displays the selected item as a signed decimal.
	Displays the selected item as a character or Unicode character.
	Sets the displayed range in the active memory window to the where the selected item is stored.
	Sorts the global variables alphabetically by name.
	Sorts the global variables numerically by address or register number (default).

## Using the Globals window

The Globals window shows the global variables of the application when the debugger is stopped. When the program stops at a breakpoint or is stepped, the Globals window automatically updates to show the active stack frame and new variable values. Items that have changed since they that were previously displayed are highlighted in red.

### Showing the Globals window

To display the **Globals** window if it is hidden, do one of the following:

- From the **View** menu, click **Other Windows** then **Globals**.  
—or—
- From the **Debug** menu, click **Debug Windows** then **Globals**.  
—or—
- Type **Ctrl+Alt+G**.  
—or—
- Right click the tool bar area to display the **View** menu.
- From the popup menu, click **Other Windows** then **Globals**.

### Changing display format

When you select a variable in the main part of the display, the display format button highlighted on the Globals window tool bar changes to show the item's display format.

To change the display format of a global variable, do one of the following:

- Right click the item to change.
- From the popup menu, select the format to display the item in.

—or—

- Click the item to change.
- On the Globals window tool bar, select the format to display the item in.

### Modifying global variable values

To modify the value of a global variable, do one of the following:

- Click the value of the global variable to modify.
- Enter the new value for the global variable. Prefix hexadecimal numbers with '0x', binary numbers with '0b', and octal numbers with '0'.

—or—

- Right click the value of the global variable to modify.
- From the popup menu, select one of the operations to modify the global variable value.










## Locals window

The locals window displays a list of all variables that are in scope of the selected stack frame in the **Call Stack**.

### Locals window user interface

The **Locals** window is divided into a tool bar and the main data display.

## Locals tool bar

Button	Description
	Displays the selected item in binary.
	Displays the selected item in octal.
	Displays the selected item in decimal.
	Displays the selected item in hexadecimal.
	Displays the selected item as a signed decimal.
	Displays the selected item as a character or Unicode character.
	Sets the displayed range in the active memory window to the where the selected item is stored.
	Sorts the local variables alphabetically by name.
	Sorts the local variables numerically by address or register number (default).

## Using the Locals window

The Locals window shows the local variables of the active function when the debugger is stopped. The contents of the Locals window changes when you use the **Debug Location** tool bar items or select a new frame in the Call Stack window. When the program stops at a breakpoint or is stepped, the Locals window automatically updates to show the active stack frame. Items that have changed since they that were previously displayed are highlighted in red.

### Showing the Locals window

To display the **Locals** window if it is hidden, do one of the following:

- From the **View** menu, click **Other Windows** then **Locals**.

—or—

- From the **Debug** menu, click **Debug Windows** then **Locals**.

—or—

- Type **Ctrl+Alt+L**.

—or—

- Right click the tool bar area to display the **View** menu.

- From the popup menu, click **Other Windows** then **Locals**.

### Changing display format

When you select a variable in the main part of the display, the display format button highlighted on the Locals window tool bar changes to show the item's display format.

To change the display format of a local variable, do one of the following:

- Right click the item to change.
- From the popup menu, select the format to display the item in.

—or—

- Click the item to change.
- On the Locals window tool bar, select the format to display the item in.

### Modifying local variable values

To modify the value of a local variable, do one of the following:

- Click the value of the local variable to modify.
- Enter the new value for the local variable. Prefix hexadecimal numbers with '0x', binary numbers with '0b', and octal numbers with '0'.

—or—

- Right click the value of the local variable to modify.
- From the popup menu, select one of the operations to modify the local variable value.

## Memory window

The memory window shows the contents of the connected targets memory areas. The memory window does not show the complete address space of the target and instead you must enter both the start address and the number of bytes for the memory window to display. You can specify the start address and the size using **Debug expressions** (page 65) which enables you to position the memory display at the start address of a variable or use a value in a register. You can also specify if you want the expressions to be evaluated each time the memory window is updated or you can re-evaluate them yourself with the press of a button.

## Memory window updates

The memory window updates each time the debugger locates to source code. So it will update each time your program stops on a breakpoint or single step and whenever you traverse the call stack. If any values that were previously displayed have changed they will be displayed in red.

## Display formats

You can set the memory window to display 8-bit, 16-bit, and 32-bit values that are formatted as hexadecimal, decimal, unsigned decimal, octal or binary. You can also change the number of columns that are displayed.

You can change a value in the memory window by clicking the value to change and editing it as a text field. Note that when you modify memory values you need to prefix hexadecimal numbers with “0x”, binary numbers with “0b” and octal numbers with “0”.

## Saving memory contents

You can save the displayed contents of the memory window to a file in various formats. Alternatively you can export the contents to a binary editor to work on them.

### Saving memory

You can save the displayed memory values as a binary file, Motorola S-record file, Intel hex file, or a Texas Instruments TXT file..

To save the current state of memory to a file, do the following:

- Selects the start address and number of bytes to save by editing the **Start Address** and **Size** fields in the memory window tool bar.
- Right click the main memory display.
- From the popup menu, select **Save As** then select the format from the submenu.

### Exporting memory

To export the current state of memory to a binary editor, do the following:

- Selects the start address and number of bytes to save by editing the **Start Address** and **Size** fields in the memory window tool bar.
- Right click the main memory display.
- From the popup menu, select **Export to Binary Editor**.

Note that subsequent modifications in the binary editor will not modify memory in the target.











# Register windows

The register windows can show the values of both CPU registers and the processor's special function or peripheral registers. Because microcontrollers are becoming very highly integrated, it's not unusual for them to have hundreds of special function registers or peripheral registers, so CrossStudio provides four register windows. You can configure each register window to display one or more register groups for the processor being debugged.

## Register window user interface

The **Registers** window is divided into a tool bar and the main data display.

### Register tool bar

Button	Description
	Displays the CPU, special function register, and peripheral register groups.
	Displays the selected item in binary.
	Displays the selected item in octal.
	Displays the selected item in decimal.
	Displays the selected item in hexadecimal.
	Displays the selected item as a signed decimal.
	Displays the selected item as a character or Unicode character.
	Sets the displayed range in the active memory window to the where the selected item is stored.
	Sorts the registers alphabetically by name.
	Sorts the registers numerically by address or register number (default).

## Using the register window

Both CPU registers and special function registers are shown in the main part of the Registers window. When the program stops at a breakpoint or is stepped, the Register windows automatically update to show the current values of the registers. Items that have changed since they that were previously displayed are highlighted in red.

### Showing the Registers window

To display register window *n* if it is hidden, do one of the following:

- From the **View** menu, click **Other Windows** then **Registers *n***.

—or—

- From the **Debug** menu, click **Debug Windows** then **Registers *n***.

—or—

- Type **Ctrl+T, R, *n***.

—or—


- Right click the tool bar area to display the **View** menu.
- From the popup menu, click **Other Windows** then **Registers *n***.

### Displaying CPU registers

The values of the CPU registers displayed in the registers window depend up upon the selected context. The selected context can be:

- The register state the CPU stopped in.
- The register state when a function call occurred selected using the Call Stack window.
- The register state of the currently selected thread using the the Threads window.
- The register state that you have supplied using the **Debug > Locate** operation.

To display a group of CPU registers, do the following:


- On the Registers window tool bar, click the Groups button —  .
- From the dropdown menu, check the register groups to display and uncheck the ones to hide.

You can uncheck all CPU register groups to allow more space in the display for special function or peripheral registers. So, for instance, you can have one register window showing the CPU registers and other register windows showing different peripheral registers.

### Displaying special function or peripheral registers

The registers window shows the set of register groups that have been defined in the memory map file that the application was built with. If there is no memory map file associated with a project, the Registers window will show only the CPU registers.

To display a special function or peripheral register, do the following:

- On the Registers window tool bar, click the Groups button —  .
- From the dropdown menu, check the register groups to display and uncheck the ones to hide.

### Changing display format

When you select a register in the main part of the display, the display format button highlighted on the Registers window tool bar changes to show the item's display format.

To change the display format of a register, do one of the following:

- Right click the item to change.
- From the popup menu, select the format to display the item in.

—or—

- Click the item to change.
- On the Registers window tool bar, select the format to display the item in.

### Modifying register values

To modify the value of a register, do one of the following:

- Click the value of the register to modify.
- Enter the new value for the register. Prefix hexadecimal numbers with '0x', binary numbers with '0b', and octal numbers with '0'.

—or—

- Right click the value of the register to modify.
- From the popup menu, select one of the operations to modify the register value.

Modifying the saved register value of a function or thread may not be supported.

## Threads window

The threads window displays the set of executing contexts on the target processor structured as a set of queues. The executing contexts are supplied to the threads window using a JavaScript program called `threads.js` that must be in the current active project. When the JavaScript program executes (when the application stops) it creates entries in the threads window that contain the name, priority and status of the thread together with the saved execution context (register state) of the thread. By double



clicking on one of the entries in the threads window the debugger is located to it's saved execution context - you can put the debugger back into the default execution context using **Show Next Statement**.

## Writing threads.js

The JavaScript program contained in threads.js must be have a named function called **update** which is called when the threads window is refreshed. The threads window is updated using the following JavaScript interface

```
Threads.newqueue("queuename")
Threads.add("threadname", threadpriority, "threadstate", registers)
```

The **Threads.newqueue** function takes a string argument and creates a new top level entry in the threads window. Subsequent threads that are added to this window will go under this.

The **Threads.add** function takes a string argument for the thread name, an integer argument for the thread priority, a string argument for the current state of the thread and finally an array (or null) containing the execution context of the thread (registers). The array containing the registers should contain the entries in the order they are displayed in the CPU registers display—typically this will be in register number order e.g. **r0**, **r1**, and so on.

To generate the thread lists you need to access the debugger from the JavaScript program. To do this you can use the JavaScript interface

```
Debug.evaluate("expression");
```

which will evaluate the string argument as a debug expression and return the result. The returned result will be an object if you evaluate an expression that denotes a structure or an array. If the expression denotes an structure then each field can be accessed using the JavaScript array notation, for example:

```
c = Debug.evaluate("complex");
i = c["i"];
j = c["j"];
```

Because JavaScript is a dynamic language, you can write the above in a more natural fashion:

```
c = Debug.evaluate("complex");
i = c.i;
j = c.j;
```

You can access arbitrary memory locations using C style casts, for example:

```
v = Debug.evaluate("* (unsigned*) 0x200");
```

and similarly you can cast to user-defined types:

```
v = Debug.evaluate("*(Complex*)0x200");
```

Note that you should ensure that the JavaScript program will terminate as if it goes into an endless loop then the debugger, and consequently CrossStudio, will become unresponsive and you will need to kill CrossStudio using a task manager.

## Trace window

The trace window displays historical information on the instructions executed by the target. The type and number of the trace entries depends upon the target that is connected when gathering trace information. Some targets may trace all instructions, others may trace jump instructions, and some may trace modifications to variables. You'll find the trace capabilities of your target on the right click context menu.

Each entry in the trace window has a unique number, and the lower the number the earlier the trace. You can click on the header to show earliest to latest or the latest to earliest trace entries. If a trace entry can have source code located to it then double clicking on the trace entry will show the appropriate source display.

Some targets may provide timing information which will be displayed in the ticks column.

The trace window is updated each time the debugger stops when it is visible. So single stepping is likely to be slower if you have this window displayed.










## Watch window

The watch window provides a means to evaluate expressions and display the values of those expressions. Typically expressions are just the name of the variable to be displayed, but can be considerably more complex see **Debug expressions** (page 65). Note that the expressions are always evaluated when your program stops so the expression you are watching is the one that is in scope of the stopped program position.

### Watch window user interface

The **Watch** window is divided into a tool bar and the main data display.

## Watch tool bar

Button	Description
	Displays the selected item in binary.
	Displays the selected item in octal.
	Displays the selected item in decimal.
	Displays the selected item in hexadecimal.
	Displays the selected item as a signed decimal.
	Displays the selected item as a character or Unicode character.
	Sets the displayed range in the active memory window to the where the selected item is stored.
	Sorts the global variables alphabetically by name.
	Sorts the global variables numerically by address or register number (default).

## Using the Watch window

Each expression appears as a row in the display. Each row contains the expression and its value. If the value of an expression is structured (for example an array) then you can open the structure see its contents.

The display is updated each time the debugger locates to source code. So it will update each time your program stops on a breakpoint or single step and whenever you traverse the call stack. Items that have changed since they that were previously displayed are highlighted in red.

### Showing the Watch window

To display watch window *n* if it is hidden, do one of the following:

- From the **View** menu, click **Other Windows** then **Watch *n***.  
—or—
- From the **Debug** menu, click **Debug Windows** then **Watch *n***.  
—or—
- Type **Ctrl+T, W, *n***.  
—or—

- Right click the tool bar area to display the **View** menu.
- From the popup menu, click **Other Windows** then **Watch n**.

### **Adding or changing items to watch**

You can add a new expression to be watched by clicking and typing into the last entry in the watch window. You can change the expression that is watched by clicking the expression entry and editing its contents.

### **Changing display format**

When you select a variable in the main part of the display, the display format button highlighted on the Watch window tool bar changes to show the item's display format.

To change the display format of a local variable, do one of the following:

- Right click the item to change.
- From the popup menu, select the format to display the item in.

—or—

- Click the item to change.
- On the Watch window tool bar, select the format to display the item in.

The selected display format will then be used for all subsequent displays and will be recorded when the debug session stops.

For C programs the interpretation of pointer types can be changed by right clicking and selecting from the popup menu. A pointer can be interpreted as:

- a null terminated ASCII string.
- an array.
- an integer.
- dereferenced.

### **Modifying watched values**

To modify the value of a local variable, do one of the following:

- Click the value of the local variable to modify.
- Enter the new value for the local variable. Prefix hexadecimal numbers with **'0x'**, binary numbers with **'0b'**, and octal numbers with **'0'**.

—or—

- Right click the value of the local variable to modify.
- From the popup menu, select one of the operations to modify the variable's value

# Help window

The help viewer is located in the HTML viewer in the main tab window. It displays the currently selected help topic.

## Context sensitive help

CrossStudio provides four types of context sensitive help with increasing detail:

- **Tool tips.** When you move your mouse pointer over a tool button and keep it still, a small window appears with a very brief description of the tool button and its keyboard shortcut if it has one.
- **Status tips.** In addition to tool tips, CrossStudio provides a longer description in the status bar when you hover over a tool button or when you move over a menu item.
- **Whats This?** For even more detail, Whats This? help provides a description of tool buttons and menu items in an expanded form.
- **Online Manual.** CrossStudio has links from all windows to the online help system.

### Whats This? help

To quickly find out what a menu item or tool button does, you can use Whats This? help. To do this:

- From the **Help** menu, click **Whats This?** or type **Shift+F1**
- Click on the tool button or menu item of interest.

CrossStudio will then display a small window containing the name and a brief description of the tool button or menu item.

### Help in the online manual

CrossStudio provides an extensive HTML-based help system which is available at all times. To go to the help information for a particular window or user interface element, do the following:

- Focus the appropriate element by clicking it.
- From the **Help** menu, click **CrossStudio Help** or type **F1**.

You can return to the **Welcome** page at any time:

- From the **View** menu, click **HTML Browser** then **Home**

—or—

- Type **Alt+Home**.

### **Help from the text editor**

The text editor is linked to the help system in a special way. If you place the cursor over a word and press **F1**, that word is looked up in the help system index and the most likely page displayed in the HTML browser—its a great way to quickly find the reference help pages for functions provided in the library.

## Using the Contents window

The **Contents** view provides a list of all the topics and sub-topics within the help system.

The highlighted entry indicates the current help topic. Other topics can be selected and the help viewer will update accordingly.

### **To move to the next topic**

- From the **Help** menu, click **Next Topic**

—or—

- Click the **Next Topic** tool button on the **Contents** window toolbar.

### **To move to the previous topic**

- From the **Help** menu, click **Previous Topic**

—or—

- Click the **Previous Topic** tool button on the **Contents** window toolbar.

## Using the Search window

Using the **Search** window you can search for multiple words or phrases. When the search button is pressed the matching pages are listed in order of relevance.

When you select a topic in the **Search** window, the corresponding help topic is shown in the HTML browser.

## Using the Index window

The index view allows single keywords to be located. Keywords can either be typed, or selected from the list. As the selected keyword changes the topic with the highest number of hits is displayed. Other topics can be selected and the help viewer will update accordingly.



# Output window

The **Output** window contains logs and transcripts from various systems within CrossStudio. Most notably, it contains the **Build Log**, **Target Log** and **Find in Files** results.

## Output window user interface

The **Output** window is divided into a tool bar and the log display.

### Output tool bar

Button	Description
	<b>Tree view</b> Shows the log as a tree view.
	<b>Flat view</b> Shows the log as a flat view.

### Showing the Output window

To display the **Output** window if it is hidden, do one of the following:

- From the **View** menu, click **Output**.
- or—
- Type **Ctrl+Alt+O**.
- or—
- Right click the tool bar area to display the **View** menu.
- From the popup menu, click **Output**.

## Using the output window

### Showing a specific log

To display a specific log, do one of the following:

- On the **Output** window tool bar, click the **Output Pane List**.
- From the list, click the log to display.
- or—
- From the **View** menu, click **Logs** and then the log to display.
- or—
- Right click the tool bar area to display the **View** menu

- From the popup menu, click **Logs** and then the log to display.

#### **Showing the Build Log**

To display the build log in the output window, do one of the following:

- From the **Build** menu, click **Show Build Log**.

—or—

- Double click the **Target Status** panel in the status bar.

#### **Showing the Target Log**

To display the target log in the output window, do the following:

- From the **Target** menu, click **Show Target Log**.









## Project explorer

The **Project Explorer** organizes your projects and files and provides quick access to the commands that operate on them. A tool bar at the top of the window offers quick access to commonly used commands for the item selected in the **Project Explorer**.

This section gives a brief overview of the project explorer window and its operation, but for a complete description of how to work with projects and how to manage them, please refer to **Project management** (page 43).



## The Project Explorer tool bar

Button	Description
	Adds a new file to the project using the <b>New File</b> dialog.
	Adds an existing files to the project.
	Removes files, folders, projects, and links from the project.
	Creates a new folder in the project.
	Builds the active project.
	Disassembles the selected project item.
	Sets project explorer options.
	Displays the properties dialog for the selected item.

### Showing the Project Explorer

To activate the **Project Explorer** if it is hidden, do one of the following:

- From the **View** menu, click **Project Explorer**.
- or—
- Type **Ctrl+Alt+P**.
- or—
- On the **Standard** tool bar, click the **Project Explorer** icon.
- or—
- Right click the tool bar area to display the **View** menu.
  - From the popup menu, click **Project Explorer**.

### Setting project properties

When you select an item in the project explorer, the properties window displays the properties that can be set for the item. This allows you to set compilation options for source files, for instance.

### Opening files for editing

Double clicking a source file will load it into the code editor for editing. As you switch between files in the editor, the selection in the project explorer changes to highlight the file that you're currently editing.

### Source code control

Using the the project explorer you can check files into and out of a source code control system. Right clicking on a source file brings up a context menu with the following source code control operations:

- **Check In** checks a file into source code control.
- **Check Out** checks a source file out of the repository and makes it writable.
- **Undo Check Out** undoes a check out and reverts the file on disk to the one in the source code control system.
- **Add To Source Control** adds a file to source control.
- **Remove From Source Control** removes a file from source control and deletes it from the source code control database.

For more information on source code control, see **Source code control** (page 61).

### Related sections

See the **Project management** (page 43) section.

## Properties window

The properties window displays properties of the current focused CrossStudio object. Using the properties window you can set build properties of your project, modify the editor defaults and change target settings.

The properties window is organised as a set of key value pairs. As you select one of the keys then a help display explains the purpose of the property. Because the properties are numerous and can be specific to a particular product build you should consider this help to be the definitive help on the property.

You can organise the property display so that it is divided into categories or alternatively display it as a flat list that is sorted alphabetically.

The combo-box enables you to change the properties yourself and explains which properties you are looking at.

Some properties have actions associated with them - you can find these by right clicking on the property key. Most properties that represent filenames can be opened this way.

When the properties window is displaying project properties youll find that some properties are displayed in **bold**. This means that the property value hasnt been inherited. If you wish to inherit rather than define such a property then on the right click context menu youll find an action that enables you to inherit the property.








# Source navigator window

One of the best ways to find your way around your source code is using the Source Navigator. The source navigator parses the active project's source code and organizes classes, functions, and variables in various ways.

## Source navigator user interface

The **Source Navigator** window is divided into a tool bar and the main breakpoint display.





### Source Navigator tool bar





Button	Description
	Sorts the objects alphabetically.
	Sorts the objects by type.
	Sorts the objects by access (public, protected, private).
	Groups objects by type (functions, classes, structures, variables).
	Move the cursor to the statement where the object is defined.
	Move the cursor to the statement where the object is declared. If more than one declaration exists, an arbitrary one is chosen.
	Manually re-parses any changed files in the project.

### Source navigator display

The main part of the **Source Navigator** window an overview of the functions, classes, and variables of your application.

CrossStudio displays these icons to the left of each object:

Icon	Description
	<b>Structure or namespace</b> A C or C++ structure or a C++ namespace.
	<b>C++ class</b> A C++ class.
	<b>Private function</b> A C++ member function that is declared <b>private</b> or a function that is declared with <b>static</b> linkage.
	<b>Protected function</b> A C++ member function that is declared <b>protected</b> .

Icon	Description
	<b>Public function</b> A C++ member function that is declared <b>public</b> or a function that is declared with <b>extern</b> linkage.
	<b>Private variable</b> A C++ member variable that is declared <b>private</b> or a variable declared with <b>static</b> linkage.
	<b>Protected variable</b> A C++ member variable that is declared <b>protected</b> .
	<b>Public variable</b> A C++ member variable that is declared <b>public</b> or a variable that is declared with <b>extern</b> linkage.

### Showing the Source Navigator window

To display the **Source Navigator** window if it is hidden, do one of the following:

- From the **View** menu, click **Source Navigator**.

—or—

- Type **Ctrl+Alt+N**.

—or—

- Right click the tool bar area to display the **View** menu.
- From the popup menu, click **Source Navigator**.

## Using the source navigator

### Parsing source files manually

To parse source files manually, do one of the following:

- From the **Tools** menu, click **Source Navigator** then **Refresh**.

—or—

- On the **Source Navigator** tool bar, click **Refresh**.

CrossStudio re-parses any changed files and updates the source navigator display with the changes. Progress information and any errors are sent to the **Source Navigator Log** in the Output window when parsing.

### Grouping objects by type

You can group object by their type, that is whether they are classes, functions, namespaces, structures, or variables. Each object is placed into a folder according to its type

To group objects in the source browser by type, do one of the following:

- From the **Tools** menu, click **Source Navigator** then **Group By Type**.

—or—

- On the **Source Navigator** tool bar, click the arrow to the right of the **Cycle Grouping** button.
- From the dropdown menu, click **Group By Type**.






## Symbol browser

The Symbol Browser window shows useful information about your linked application and complements the information displayed in the Project Explorer window. You can select different ways to filter and group the information in the symbol browser to provide an at-a-glance overview of your application as a whole. You can use the symbol browser to *drill down* to see how big each part of your program is and where it's placed. The way that symbols are sorted and grouped is saved between runs. When you rebuild an application, CrossStudio automatically updates the symbol browser so you can see the effect your changes are making to the memory layout of your program.

### Symbol browser user interface








The symbol browser is divided into a tool bar and the main symbol display.

## Symbol Browser tool bar

Button	Description
	Groups symbols by source file name.
	Groups symbols by symbol type (equates, functions, labels, sections, and variables)
	Groups symbols by the section that they are defined in.
	Moves the cursor to the statement that defined the symbol.
	Chooses the columns to display in the symbol browser.

## Symbol Browser display

The main part of the symbol browser displays each symbol (both external and static) that the is linked into an application. CrossStudio displays these icons to the left of each symbol:

Icon	Description
	<b>Private Equate</b> A private symbol that is not defined relative to a section.
	<b>Public Equate</b> A public symbol that is not defined relative to a section.
	<b>Private Function</b> A private function symbol.
	<b>Public Function</b> A public function symbol.
	<b>Private Label</b> A private data symbol, defined relative to a section.
	<b>Public Label</b> A public data symbol, defined relative to a section.
	<b>Section</b> A program section.

## Symbol browser columns

You can choose to display the following fields against each symbol:

- **Value.** The value of the symbol. For labels, code, and data symbols this will be the address of the symbol. For absolute or symbolic equates, this will be the value of the symbol.
- **Range.** The range of addresses the code or data item covers. For code symbols that correspond to high-level functions, the range is the range of addresses used for that function's code. For data addresses that correspond to high-level **static** or **extern**

variables, the range is the range of addresses used to store that data item. These ranges are only available if the corresponding source file was compiled with debugging information turned on: if no debugging information is available, the range will simply be the first address of the function or data item.

- **Size.** The size, in bytes, that the code or data item covers. The **Size** column is derived from the **Range** of the symbol: if the symbol corresponds to a high-level code or data item and has a range, then **Size** is calculated as the difference between the start and end address of the range. If a symbol has no range, the size column is left blank.
- **Section.** The section in which the symbol is defined. If the symbol is not defined within a section, the **Section** column is left blank.
- **Type.** The high-level type for the data or code item. If the source file that defines the symbol is compiled with debugging information turned off, type information is not available and the **Type** column is left blank.

### Showing the Symbol Browser window

To display the **Symbol Browser** window if it is hidden, do one of the following:

- From the **View** menu, click **Symbol Browser**.

—or—

- Type **Ctrl+Alt+Y**.

—or—

- Right click the tool bar area to display the **View** menu.
- From the popup menu, click **Symbol Browser**.

## Configuring the Symbol Browser

### Choosing fields to display

Initially the **Range** and **Size** columns are shown in the symbol browser. You can select which columns to display using the **Field Chooser** on the **Symbol Browser** tool bar.

To select the fields to display in the Symbol Browser, do one of the following:

- Click the **Field Chooser** button on the **Symbol Browser** tool bar.
- Check the fields that you wish to display and uncheck the fields that you wish to hide.

—or—

- From the **Tools** menu, select **Symbol Browser** then **Fields**.

- Check the fields that you wish to display and uncheck the fields that you wish to hide.

### Grouping symbols by section

When you group symbols by section, each symbol is grouped underneath the section that it is defined in. Symbols that are absolute or are not defined within a section are grouped beneath “(No Section)”.

To group symbols by section, do the following:

- On the **Symbol Browser** tool bar, click the arrow next to the **Cycle Grouping** tool button.
- From the popup menu, click **Group By Section**.

—or—

- From the **Tools** menu, click **Symbol Browser** then **Group By Section**.

The **Cycle Grouping** tool button icon will change to indicate that the symbol browser is now grouping symbols by section.

### Grouping symbols by type

When you group symbols by type, each symbol is grouped underneath the type of symbol that it is. Each symbol is classified as one of the following:

- An **Equate** has an absolute value and is not defined as relative to, or inside, a section.
- A **Function** is a symbol that is defined by a high-level code sequence.
- A **Variable** is defined by a high-level data declaration.
- A **Label** is a symbol that is defined by an assembly language module. **Label** is also used when high-level modules are compiled with debugging information turned off.

To group symbols by source type, do the following:

- On the **Symbol Browser** tool bar, click the arrow next to the **Cycle Grouping** tool button.
- From the popup menu, click **Group By Type**.

—or—

- From the **Tools** menu, click **Symbol Browser** then **Group By Type**.

The **Cycle Grouping** tool button icon will change to indicate that the symbol browser is now grouping symbols by type.



### Grouping symbols by source file

When you group symbols by source file, each symbol is grouped underneath the source file that it is defined in. Symbols that are absolute, are not defined within a source file, or are compiled with without debugging information, are grouped beneath “(Unknown)”.

To group symbols by source file, do one of the following:

- On the **Symbol Browser** tool bar, click the arrow next to the **Cycle Grouping** tool button.
- From the popup menu, click **Group By Source File**.

—or—

- From the **Tools** menu, click **Symbol Browser** then **Group By Source File**.

The **Cycle Grouping** tool button icon will change to indicate that the symbol browser is now grouping symbols by source file.

### Sorting symbols alphabetically

When you sort symbols alphabetically, all symbols are displayed in a single list in alphabetical order.

To group symbols alphabetically, do one of the following:

- On the **Symbol Browser** tool bar, click the arrow next to the **Cycle Grouping** tool button.
- From the popup menu, click **Sort Alphabetically**.

—or—

- From the **Tools** menu, click **Symbol Browser** then **Sort Alphabetically**.

The **Cycle Grouping** tool button icon will change to indicate that the symbol browser is now grouping symbols alphabetically.

## Filtering, finding, and watching symbols

When you’re dealing with big projects with hundreds, or even thousands, of symbols, a way to filter the display of those symbols and drill down to the ones you need is very useful. The symbol browser provides an editable combo box in the toolbar which you can use to specify the symbols you’d like displayed. The symbol browser uses “\*” to match a sequence of zero or more characters and “?” to match exactly one character.

The symbols are filtered and redisplayed as you type into the combo box. Typing the first few characters of a symbol name is usually enough to narrow the display to the symbol you need. One thing to note is that the C compiler prefixes all high-level language symbols with an underscore character, so the variable `extern int u` or the

function **void fn(void)** have low-level symbol names `_u` and `_fn`. The symbol browser uses the low-level symbol name when displaying and filtering, so you must type the leading underscore to match high-level symbols.

### Finding symbols with a common prefix

To display symbols that start with a common prefix, do the following:

- Type the required prefix into the combo box, optionally followed by a `**`.

For instance, to display all symbols that start with `"i2c_"`, type `"i2c_"` and all matching symbols are displayed—you don't need to add a trailing `**` in this case as it is implied.

### Finding symbols with a common suffix

To display symbols that end with a common suffix, do the following:

- Type `**` into the combo box followed by the required suffix.

For instance, to display all symbols that end in `"_data"`, type `**_data"` and all matching symbols are displayed—in this case the leading `**` is required.

### Jumping to the definition of a symbol

Once you have found the symbol you're interested in and your source files have been compiled with debugging information turned on, you can jump to the definition of a symbol using the **Go To Definition** tool button.

To go to the definition of a symbol, do one of the following:

- Select the symbol from the list of symbols.
- On the **Symbol Browser** tool bar, click **Go To Definition**.

—or—

- Right click the symbol in the list of symbols.
- From the popup menu, click **Go To Definition**.

### Adding symbol to watch and memory windows

If a symbol's range and type is known, you can add it to the most recently opened watch window or memory window.

To add a symbol to the watch window, do the following:

- In the **Symbol Browser**, right click on the the symbol you wish to add to the watch window.
- From the popup menu, click **Add To Watch**.

To add a symbol to the memory window, do the following:

- In the **Symbol Browser**, right click on the the symbol you wish to add to the memory window.
- From the popup menu, click **Locate Memory**.

## Working with the Symbol Browser

Here are a few common ways to use the symbol browser:

### What function takes up the most code space or what takes the most data space?

- Show the symbol browser by selecting **Symbol Browser** from the **Tools** menu.
- Group symbols by type by choosing **Symbol Browser > Group By Type** from the **Tools** menu.
- Make sure that the **Size** field is checked in **Symbol Browser > Fields** on the **Tools** menu.
- Ensure that the filter on the symbol browser tool bar is empty.
- Click on the **Size** field in the header to sort by data size.
- Read off the the sizes of variables under the **Variable** group and functions under the **Functions** group.

### What's the overall size of my application?

- Show the symbol browser by selecting **Symbol Browser** from the **Tools** menu.
- Group symbols by section by choosing **Symbol Browser > Group By Section** from the **Tools** menu.
- Make sure that the **Range** and **Size** fields are checked in **Symbol Browser > Fields** on the **Tools** menu.
- Read off the section sizes and ranges of each section in the application.






## Targets window

The targets window (and associated menu) displays the set of target interfaces that you can connect to in order to download and debug your programs. Using the targets window in conjunction with the properties window enables you to define new targets based on the specific target types supported by the particular CrossStudio release.

You can connect, disconnect, and reconnect to a target system. You can also reset and load programs using the target window. If you load a program using the target window and you need to debug it then you will have to use the **Debug > Attach Debugger** operation.

## Targets window layout

### Targets tool bar

Button	Description
	Connects the selected target interface.
	Disconnects the connected target interface.
	Reconnects the connected target interface.
	Resets the connected target interface.
	Displays the properties of the selected target interface.

### Showing the Targets window

To display the **Targets** window if it is hidden, do one of the following:

- From the **View** or **Target** menu, click **Targets**.
- or—
- Type **Ctrl+Alt+T**.
- or—
- Right click the tool bar area to display the **View** menu.
  - From the popup menu, click **Targets**.

## Managing target connections

### Connecting a target

To connect a target, do one of the following:

- In the **Targets** window, double click the target to connect.
- or—
- From the **Target** menu, click the target to connect.

—or—

- In the **Targets** window, click the target to connect.
- On the **Targets** window tool bar, click the **Connect** button

—or—

- In the **Targets** window, right click the target to connect.
- From the popup menu, click **Connect**

—or—

- In the **Targets** window, click the target to connect.
- Type **Ctrl+T, C**.

### **Disconnecting a target**

To disconnect a target, do one of the following:

- From the **Target** menu, click **Disconnect**

—or—

- On the **Targets** window tool bar, click the **Disconnect** button

—or—

- Type **Ctrl+T, D**.

—or—

- Right click the connected target in the **Targets** window
- From the popup menu, click **Disconnect**.

Alternatively, connecting a different target will automatically disconnect the current target connection.

### **Reconnecting a target**

You can disconnect and reconnect a target in a single operation using the reconnect feature. This may be useful if the target board has been power cycled or reset manually as it forces CrossStudio to resynchronize with the target.

To reconnect a target, do one of the following:

- From the **Target** menu, click **Reconnect**.

—or—

- On the **Targets** window tool bar, click the **Reconnect** button.

—or—

- Type **Ctrl+T, E**.

—or—

- In the **Targets** window, right click the target to reconnect.
- From the popup menu, click **Reconnect**.

### **Automatic target connection**

You can configure CrossStudio to automatically connect to the last used target interface when loading a solution.

To enable or disable automatic target connection, do the following:

- From the **View** menu, click **Properties Window**.
- In the **Properties Window**, click **Environment Properties** from the combo box.
- In the **Target Settings** section, set the **Enable Auto Connect** property to **Yes** to enable automatic connection or to **No** to disable automatic connection.

### **Creating a new target interface**

To create a new target interface, do the following:

- From the targets window's context menu, click **New Target Interface**. A new menu will be displayed containing the types of target interface that may be created.
- Select the type of target interface to create.

### **Setting target interface properties**

All target interfaces have a set of properties. Some properties are read-only and provide information on the target, others are modifiable and allow the target interface to be configured. Target interface properties can be viewed and edited using CrossStudio's property system.

To view or edit target properties, do the following:

- Select a target.
- Select the **Properties** option from the target's context menu.

### **Restoring default target definitions**

The targets window provides the facility to restore the target definitions to the default set. Restoring the default target definitions will undo any of the changes you have made to the targets and their properties and therefore should be used with care.

To restore the default target definitions, do the following:

- Select **Restore Default Targets** from the targets window context menu.
- Click **Yes** when prompted if you want to restore the default targets.

## Controlling target connections

### Resetting the target

Reset of the target is typically handled automatically by the system when you start debugging. However, the target may be manually reset using the **Targets** window.

To reset the connected target, do one of the following:

- On the **Targets** window tool bar, click the **Reset** button.

—or—

- From the **Target** menu, click **Reset**

—or—

- Type **Ctrl+T, S**.

### Downloading programs

Program download is handled automatically by CrossStudio when you start debugging. However, you can download arbitrary programs to a target using the **Targets** window.

To download a program to the currently selected target, do the following:

- In the **Targets** window, right click the selected target.
- From the popup menu, click **Download File**.
- From the **Download File** menu, select the type of file to download.
- In the **Open File** dialog, select the executable file to download and click **Open** to download the file.

CrossStudio supports the following file formats when downloading a program:

- Binary
- Intel Hex
- Motorola S-record
- CrossWorks native object file
- Texas Instruments text file

### Verifying downloaded programs

You can verify a target's contents against a arbitrary programs held on disk using the **Targets** window.

To verify a target's contents against a program, do the following:

- In the **Targets** window, right click the selected target.

- From the popup menu, click **Verify File**.
- From the **Verify File** menu, select the type of file to verify.
- In the **Open File** dialog, select the executable file to verify and click **Open** to verify the file.

CrossStudio supports the same file types for verification as it does for downloading, described above.

### **Erasing target memory**

Usually, erasing target memory is done automatically. CrossStudio downloads a program, but you can erase a target's memory manually.

To completely erase target memory, do the following:

- In the **Targets** window, right click the target to erase.
- From the popup menu, click **Erase All**.

To erase part of target memory, do the following:

- In the **Targets** window, right click the target to erase.
- From the popup menu, click **Erase Range**.

### **Target definition file**

The target interface information in the targets window is stored in an XML file called the target definition file.

To change the target definition file used by the targets window, do the following:

- From the Tools menu, click **Options**.
- In the **Environment Options**, select the **Target** section.
- Edit the **Target definition file** entry to change the path to the target definition file.
- Click **OK** to apply the change, the targets window should load the new target definition file.



## Target interfaces

A target interface is a mechanism for communicating with and controlling a target. A target may be a physical hardware device or a simulator.

CrossStudio has a targets window for viewing and manipulating target interfaces. For more information on the targets window, see **Targets window** (page 139).

Before a target interface can be used, it must be connected. CrossStudio permits connection to only one target at a time. For more information on connecting to target interfaces, see [Connecting to a target](#).

All target interfaces have a set of properties. The properties provide information on the connected target and allow the target interface to be configured. For more information on viewing and editing target properties, see [Viewing and editing target properties](#).

CrossWorks supports a number of different ARM debug interfaces, each target interface can support one or more of these. The following table summarises the supported target interfaces and the ARM debug interfaces supported:

Target Interface	ARM7 DI	ARM7 TDI	ARM9 TDI	XScal e	ADiv5
<b>CrossConnect Target interface</b> (page 148)	Yes	Yes	Yes	Yes	Yes
<b>Macraigor Wiggler (20 and 14 pin) Target interface</b> (page 146)	Yes	Yes	Yes	Yes	Yes

Target Interface	ARM7 DI	ARM7 TDI	ARM9 TDI	XScale	ADIV5
<b>Segger J-Link</b> Target interface (page 149)	Yes	Yes	Yes	No	No
<b>ARM Simulator</b> Target interface (page 151)	Yes	Yes	Yes	Yes	Yes

The following table summarises the debug facilities available to each ARM debug interface:

ARM Debug Architecture	Software Breakpoints	Hardware Breakpoints	Break on Exception
ARM7DI	Unlimited (1 hardware breakpoint used)	2	No
ARM7TDI	Unlimited (1 hardware breakpoint used)	2	No
ARM9TDI	Unlimited (1 hardware breakpoint used)	2	Yes
XScale	Unlimited	4 (2 instruction and 2 data)	Yes
ADIV5	Unlimited	10 (6 instruction and 4 data)	Yes

## Macraigor Wiggler (20 and 14 pin) Target interface

The Macraigor Wiggler target interface provides access to ARM targets via Macraigor System's Wiggler for ARM (or compatible device). This target interface supports program loading and debugging of both RAM and FLASH based applications. There are two variants of the Wiggler, one with 20 pins and one with 14 pins, and both are supported.

### Connection properties

- **Parallel Port** The parallel port connection to use to connect to the target.

- **Parallel Port Address** The base address of the currently connected parallel port (if available).
- **Parallel Port Sharing** If set to **Yes**, parallel port may be shared with other device drivers and programs. If set to **No**, the target interface will demand exclusive use of the port.

## Interface properties

- **Version** The device driver version information.

## Current device properties

- **Device Type** The JTAG device ID of the currently connected device.

## JTAG properties

- **Identify Target** Specifies whether the target should be identified on connection. Use this setting with caution as setting this to "No" disables the checks carried out on connection that the JTAG interface is functioning correctly and may make an malfunctioning JTAG connection appear as if it is working.
- **Invert nSRST** Specifies whether the **nSRST** signal should be inverted.
- **JTAG Clock Divider** The value to divide the JTAG clock frequency. This feature allows the JTAG clock frequency to be reduced in order to allow CrossStudio to communicate with boards with unreliable target interfaces.

## Loader properties

- **Erase All** If set to **Yes**, all of the target's FLASH memory will be erased prior to downloading the application. This can be used to speed up download of large programs as it generally quicker to erase a whole device rather than individual segments. If set to **No**, only the areas of FLASH containing the program being downloaded will be erased.
- **Erase All Timeout** The timeout period for an erase all operation in milliseconds.

## Target properties

- **Debug Interface Type** The type of debug interface that the target has. Note that the value of this property will be automatically set to a project's **ARM Debug Interface** property on connection.

- **Memory Access Timeout** The timeout period for memory accesses in milliseconds.
- **Processor Endian** Specifies the byte order of the target processor. Note that the value of this property will be automatically set to a project's **Endian** property when a project is downloaded or attached to.
- **Processor Stop Time** The timeout period, in milliseconds, to allow when stopping the processor.

## CrossConnect Target interface

The CrossConnect for ARM target interface provides access to ARM targets via the Rowley Associates CrossConnect for ARM. This target interface supports program loading and debugging of both RAM and FLASH based applications.

### Current device properties

- **Device Type** The JTAG device ID of the currently connected device.

### CrossConnect Interface properties

- **Firmware Variant** The variant of the firmware running on the currently connected CrossConnect.
- **Version** The version information of the firmware running on the currently connected CrossConnect.>
- **Information** Current CrossConnect status information.
- **Serial Number** The serial number of the currently connected CrossConnect device.
- **Use Serial Number** The serial number of the CrossConnect you want to connect to. If multiple CrossConnects are connected to your system, this property allows you to specify which one to use. If no serial number is specified, the first available CrossConnect will be used.

### JTAG properties

- **Adaptive Clocking** Specifies whether JTAG adaptive clocking using the RTCK signal should be used. This option requires the variable speed variant of the CrossConnect firmware.

- **Identify Target** Specifies whether the target should be identified on connection. Use this setting with caution as setting this to "No" disables the checks carried out on connection that the JTAG interface is functioning correctly and may make an malfunctioning JTAG connection appear as if it is working.
- **JTAG Clock Divider** The value to divide the JTAG clock frequency. The variable speed variant of the CrossConnect firmware is required if this value is set greater than 1.

## Loader properties

- **Erase All** If set to **Yes**, all of the target's FLASH memory will be erased prior to downloading the application. This can be used to speed up download of large programs as it generally quicker to erase a whole device rather than individual segments. If set to **No**, only the areas of FLASH containing the program being downloaded will be erased.
- **Erase All Timeout** The timeout period for an erase all operation in milliseconds.

## Target properties

- **Debug Interface Type** The type of debug interface that the target has. Note that the value of this property will be automatically set to a project's **ARM Debug Interface** property on connection.
- **Memory Access Timeout** The timeout period for memory accesses in milliseconds.
- **Processor Endian** Specifies the byte order of the target processor. Note that the value of this property will be automatically set to a project's **Endian** property when a project is downloaded or attached to.
- **Processor Stop Time** The timeout period, in milliseconds, to allow when stopping the processor.

## Segger J-Link Target interface

The Segger J-Link target interface provides access to ARM targets via the Segger USB J-Link ARM JTAG interface. This target interface supports program loading and debugging of both RAM and FLASH based applications.

### Current device properties

- **Device Type** The JTAG device ID of the currently connected device.

## J-Link properties

- **Current Speed** The JTAG clock frequency the J-Link is currently running.
- **DLL Compile Time** The J-Link DLL's compilation date and time.
- **DLL Version** The J-Link DLL's version information.
- **Version** The J-Link's firmware string.
- **Firmware Version** The J-Link's firmware version.
- **J-Link DLL File** The file path locating the the JLinkARM.dll to use. The JLinkARM.dll is part of the *J-Link ARM software and documentation pack* which comes with J-Link and is also available for download from the Segger web site.
- **Speed** The JTAG clock frequency.

## JTAG properties

- **Identify Target** Specifies whether the target should be identified on connection. Use this setting with caution as setting this to "No" disables the checks carried out on connection that the JTAG interface is functioning correctly and may make an malfunctioning JTAG connection appear as if it is working.

## Loader properties

- **Erase All** If set to **Yes**, all of the target's FLASH memory will be erased prior to downloading the application. This can be used to speed up download of large programs as it generally quicker to erase a whole device rather than individual segments. If set to **No**, only the areas of FLASH containing the program being downloaded will be erased.
- **Erase All Timeout** The timeout period for an erase all operation in milliseconds.

## Target properties

- **Debug Interface Type** The type of debug interface that the target has. Note that the value of this property will be automatically set to a project's **ARM Debug Interface** property on connection.
- **Memory Access Timeout** The timeout period for memory accesses in milliseconds.
- **Processor Endian** Specifies the byte order of the target processor. Note that the value of this property will be automatically set to a project's **Endian** property when a project is downloaded or attached to.

- **Processor Stop Time** The timeout period, in milliseconds, to allow when stopping the processor.

## ARM Simulator Target interface

The ARM Simulator target interface provides access to CrossStudio's ARM instruction set simulator (ISS). The ISS simulates the ARM V5TE instruction set as defined in Part A of the ARM Architecture Reference Manual (ARM DDI 0100E). The ISS supports exception vectors at address zero - high vectors are not supported. The ISS implements a 3 word instruction pre-fetch buffer.

The ISS supports MCR and MRC access to the 16 primary registers of the System Control coprocessor (CP15) as defined in Part B of the ARM DDI 0100E, however the cache and MMU functionality is not supported. The ISS supports MCR and MRC access to the Debug Communication Channel (CP14) as defined in ARM7TDMI Technical Reference Manual (ARM DDI 0210B).

The memory system simulated by the ISS is specified by a dynamic link library and associated parameter defined in **ARM Simulator Properties** (page 397).

The ISS supports program loading and debugging with an unlimited number of breakpoints. The ISS supports instruction tracing, execution counts, exception vector trapping and exception vector triggering.





## ARM Target Support

When a target specific executable project is created using the **Creating a project** (page 14), the following default files are added to the project:

- *Target\_Startup.s* - The **Target startup code** (page 154)code.
- *crt0.s* - The CrossWorks standard **Startup code** (page 155) code.
- *Target\_MemoryMap.xml* - The **Memory map files** (page 157) file for the board. Note that for some target's a general linker placement file may not be suitable. In these cases there will be two memory map files, one for a Flash build and one for a RAM build.
- *flash\_placement.xml* - The linker placement file for a Flash build.
- *sram\_placement.xml* - The linker placement file for a RAM build.
- *Target\_Target.js* - The **Target script file** (page 159)

Initially, shared versions of these files are added to the project, if you want to modify any these files you should select the file in the project explorer and then click the **Import** option from the context menu. This will copy a writeable version of the file into your project directory and change the path in the project explorer to be that of the local file. You will then be able to make changes to the local file without effecting the shared copy of the file.

The following list describes the typical flow of a C program created using CrossStudio's project templates:

- The processor starts executing at address 0x0000000 which is the reset exception vector. The exception vector table can be found in the **Target startup code** (page 154) code, it is put into the program section *.vectors* which is positioned at address 0x00000000 by the **Memory map files** (page 157) file.

- The processor jumps to the *reset\_handler* label in the **Target startup code** (page 154) code which configures the target.
- When the target is configured the **Target startup code** (page 154) code jumps to the *\_start* entry point in the **Startup code** (page 155) code which sets up the C runtime environment.
- When the C runtime environment has been set up the **Startup code** (page 155) code jumps to the C entry point function *main*.
- When the program returns from main, it re-enters the **Startup code** (page 155) code, executes the destructors and then finally enters an endless loop.

## Target startup code

The following section describes the role of the target specific startup code.

When you create a new project to produce an executable file using a target specific project template, a file containing the default startup code for the target will be added to the project. Initially a shared version of this file will be added to the project, if you want to modify this file you should select the file in the project explorer and then select **Import** to copy the file to your project directory.

The target startup file typically consists of the following:

- *\_vectors* - This is the exception vector table. It is put into it's own *.vectors* section in order to ensure that it is always placed at address 0x00000000.
- *reset\_handler* - This is the main reset handler function and typically the main entry point of an executable. The reset handler will usually carry out any target specific initialisation and then jump to the *\_start* entry point. In a C system the *\_start* entry point is in the **Startup code** (page 155) file.
- *undef\_handler* - This is the default undefined instruction exception handler. This has been declared as a weak symbol to allow the user to override the implementation.
- *swi\_handler* - This is the default software interrupt exception handler. This has been declared as a weak symbol to allow the user to override the implementation.
- *pabort\_handler* - This is the default prefetch abort exception handler. This has been declared as a weak symbol to allow the user to override the implementation.
- *dabort\_handler* - This is the default data abort exception handler. This has been declared as a weak symbol to allow the user to override the implementation.
- *irq\_handler* - This is the default IRQ exception handler. This has been declared as a weak symbol to allow the user to override the implementation.

- *fiq\_handler* - This is the default FIQ exception handler. This has been declared as a weak symbol to allow the user to override the implementation.

## Startup code

The following section describes the role of the C runtime startup code, `crt0.s`.

When you create a new project to produce an executable file using a target specific project template, the `crt0.s` file will be added to the project. Initially a shared version of this file will be added to the project, if you want to modify this file you should select the file in the project explorer and then select **Import** to copy the file to your project directory.

The entry point of the C runtime startup code is `_start`. In a typical system this will be called by the **Target startup code** (page 154) code after it has initialized the target.

The C runtime carries out the following actions:

- Initialize the stacks.
- Copy the contents of the `.data` (initialized data) section from non-volatile memory should it be required.
- Copy the contents of the `.fast` section from non-volatile memory to SRAM should it be required.
- Initialize the `.bss` section.
- Initialize the heap.
- Call constructors.
- Jump to the *main* entry point.
- Call destructors.
- Wait in exit loop.

## Stacks

The ARM maintains six separate stacks. The position and size of these stacks are specified in the project's section placement or memory map file by the following program sections:

- `.stack` — System and User mode stack.
- `.stack_svc` — Supervisor mode stack
- `.stack_irq` — IRQ mode stack

- `.stack_fiq` — FIQ mode stack
- `.stack_abt` — Abort mode stack.
- `.stack_und` — Undefined mode stack.

The `crt0.s` startup code references these sections and initializes each of the stack pointer registers to point to the appropriate memory location. To change the location in memory of a particular stack, the section should be moved to the required position in the section placement or memory map file.

There is a **Stack Size** linker project property for each stack, you can modify this property in order to alter each stack maximum size. For compatibility with earlier versions of CrossStudio you can also specify the stack size using the stack section's **Size** property in the section placement or memory map file.

Should your application not require one or more of these stacks to be set up you can remove the sections from the memory map file or set the size to 0 and remove the initialization code from the `crt0.s` file.

## `.data` Section

The `.data` section contains the initialized data. If the run address is different from the load address, as it would be in a FLASH based application in order to allow the program to run from reset, the `crt0.s` startup code will copy the `.data` section from the load address to the run address before calling the `main` entry point.

## `.fast` Section

For performance reasons it is a common requirement with embedded systems to have critical code running from fast memory, the `.fast` section can be used to simplify this. If the `.fast` section's run address is different from the load address the `crt0.s` startup code will copy the `.fast` section from the load address to the run address before calling the `main` entry point.

## `.bss` Section

The `.bss` section contains the zero initialized data. The `crt0.s` startup code references the `.bss` section and sets its contents to zero.

## Heap

The position and size of the heap is specified in the project's section placement or memory map file by the `.heap` program section.

The `crt0.s` startup code references this section and initializes the heap. To change the position of the heap, the section should be moved to the required position in the section placement or memory map file.

There is a **Heap Size** linker project property, you can modify this property in order to alter the heap size. For compatibility with earlier versions of CrossStudio you can also specify the heap size using the heap section's **Size** property in the section placement or memory map file.

Should your application not require the heap functions, you can remove the heap section from the memory map file or set the size to 0 and remove the heap initialization code from the crt0.s file.

## Memory map files

CrossStudio's memory map files are XML files that are used for the following purposes:

- **Linking** — Memory map files are used by the linker to describe how to lay out a program in memory.
- **Loading** — Memory map files are used by the loader to check that a program being downloaded will actually fit into the target's memory.
- **Debugging** — Memory map files are used by the debugger to describe the location and types of memory a target has. This information is used to decide how to debug the program, for example whether to set hardware or software breakpoints on particular memory location.

There are two types of memory map files:

- *Board Memory Definition* - This type of memory map file is used to describe a target's memory segments. If no *Linker Placement* file is defined, a *Board Memory Definition* file can also describe how program sections should be laid out within the memory segments.
- *Linker Placement* - This type of memory map file is used to describe how program sections should be laid out in the memory segments described by a *Board Memory Definition* file. As the *Linker Placement* file does not describe memory addresses, only the mapping between memory segments and program sections, it can be used as a general means to describe the layout of a program not tied to a particular target. A *Linker Placement* file does not need to be used if the *Board Memory Definition* file contains all the program section information.

Memory map files can be viewed and edited using CrossStudio's memory map editor, for more information see **Memory map editor** (page 89).

To use a memory map file, simply add the memory file to a project. You may have configuration specific memory map files by excluding memory map files from configurations as you would any other source file.

## Project configurations

The following table describes the default set of **Project configurations** (page 52) when you create a new project:

Configuration Name	Description
ARM Flash Debug	Compile/assemble for ARM instruction set. Link ARM version of libraries. Load into and run from Flash memory. Compile/assemble with debug information and with optimization disabled.
ARM Flash Release	Compile/assemble for ARM instruction set. Link ARM version of libraries. Load into and run from Flash memory. Compile/assemble without debug information and with optimization enabled.
ARM RAM Debug	Compile/assemble for ARM instruction set. Link ARM version of libraries. Load into and run from RAM. Compile/assemble with debug information and with optimization disabled.
ARM RAM Release	Compile/assemble for ARM instruction set. Link ARM version of libraries. Load into and run from RAM. Compile/assemble without debug information and with optimization enabled.
Thumb Flash Debug	Compile/assemble for Thumb instruction set. Link Thumb version of libraries. Load into and run from Flash memory. Compile/assemble with debug information and with optimization disabled.
Thumb Flash Release	Compile/assemble for Thumb instruction set. Link Thumb version of libraries. Load into and run from Flash memory. Compile/assemble without debug information and with optimization enabled.
Thumb RAM Debug	Compile/assemble for Thumb instruction set. Link Thumb version of libraries. Load into and run from RAM. Compile/assemble with debug information and with optimization disabled.
Thumb RAM Release	Compile/assemble for Thumb instruction set. Link Thumb version of libraries. Load into and run from RAM. Compile/assemble without debug information and with optimization enabled.

## Target script file

The target interface system uses CrossStudios JavaScript (ECMAScript) interpreter to support board and target-specific behaviour.

The main use for this is to support non-standard target and board reset schemes and also to configure the target after reset, see Reset Script for more information.

The target script system can also be used to carry out target specific operations when the debugger attaches, stops or starts the target. This can be useful when debugging with caches enabled as it provides a mechanism for the debugger to flush and disable caches when the processor enters debug state and then re-enable the caches when the processor is released into run state. See Attach Script, Stop Script, and Run Script for more information.

In order to reduce script duplication, when the target interface runs a reset, attach, run or stop script it first looks in the current active project for a file marked with a project property **File Type** set to **Reset Script**. If a file of this type is found it will be loaded prior to executing the scripts, each of the scripts can then call functions within this script file.

### Reset Script

The **Reset Script** property held in the **Target** project property group is used to define a script to execute to reset and configure the target.

The aim of the reset script is to get the processor into a known state. When the script has executed the processor should be reset, stopped on the first instruction and configured appropriately.

As an example, the following script demonstrates the reset script for an Evaluator 7T target board with a memory configuration that re-maps SRAM to start from 0x00000000. The **Evaluator7T\_Reset** function carries out the standard ARM reset and stops the processor prior to executing the first instruction. The **Evaluator7T\_ResetWithRamAtZero** function calls this reset function and then configures the target memory by accessing the configuration registers directly. See TargetInterface Object for a description of the TargetInterface object which is used by the reset script to access the target hardware.

```
function Evaluator7T_Reset()
{
    TargetInterface.setNSRST(0);
    TargetInterface.setICEBreakerBreakpoint(0, 0x00000000, 0xFFFFFFFF,
                                             0x00000000, 0xFFFFFFFF, 0x100,
0xF7);
    TargetInterface.setNSRST(1);
    TargetInterface.waitForDebugState(1000);
    TargetInterface.trst();
}
```

```

function Evaluator7T_ResetWithRamAtZero()
{
    Evaluator7T_Reset();

/*****
 * Register settings for the following memory configuration:
 *
 *+-----+
 * | ROMCON0 - 512K FLASH | 0x01800000 - 0x0187FFFF
 * +-----+
 * | ROMCON2 - 256K SRAM | 0x00040000 - 0x0007FFFF
 *+-----+
 * | ROMCON1 - 256K SRAM | 0x00000000 - 0x0003FFFF
 *+-----+
 *
 *****/

    TargetInterface.pokeWord(0x03FF0000, 0x07FFFA0); // SYSCFG
    TargetInterface.pokeWord(0x03FF3000, 0x00000000); // CLKCON
    TargetInterface.pokeWord(0x03FF3008, 0x00000000); // EXTACON0
    TargetInterface.pokeWord(0x03FF300C, 0x00000000); // EXTACON1
    TargetInterface.pokeWord(0x03FF3010, 0x0000003E); // EXTDWIDTH
    TargetInterface.pokeWord(0x03FF3014, 0x18860030); // ROMCON0
    TargetInterface.pokeWord(0x03FF3018, 0x00400010); // ROMCON1
    TargetInterface.pokeWord(0x03FF301C, 0x00801010); // ROMCON2
    TargetInterface.pokeWord(0x03FF3020, 0x08018020); // ROMCON3
    TargetInterface.pokeWord(0x03FF3024, 0x0A020040); // ROMCON4
    TargetInterface.pokeWord(0x03FF3028, 0x0C028040); // ROMCON5
    TargetInterface.pokeWord(0x03FF302C, 0x00000000); // DRAMCON0
    TargetInterface.pokeWord(0x03FF3030, 0x00000000); // DRAMCON1
    TargetInterface.pokeWord(0x03FF3034, 0x00000000); // DRAMCON2
    TargetInterface.pokeWord(0x03FF3038, 0x00000000); // DRAMCON3
    TargetInterface.pokeWord(0x03FF303C, 0x9C218360); // REFEXTCON

```

## Attach Script

The **Attach Script** property held in the **Target** project property group is used to define a script that is executed when the debugger first attaches to an application. This can be after a download or reset before the program is run or after an attach to a running application. The aim of the attach script is to carry out any target specific configuration before the debugger first attaches to the application being debugged.

See TargetInterface Object for a description of the TargetInterface object which is used by the attach script to access the target hardware.

## Stop Script

The **Stop Script** property held in the **Target** project property groups is used to define a script that is executed when the target enters debug/stopped state. This can be after the application hits a breakpoint or when the **Debug | Break** operation has been carried out. The aim of the stop script is to carry out any target specific operations



before the debugger starts accessing target memory. This is particularly useful when debugging applications that have caches enabled as the script can disable and flush the caches enabling the debugger to access the current memory state.

See `TargetInterface` Object for a description of the `TargetInterface` object which is used by the stop script to access the target hardware.

## Run Script

This **Run Script** property held in the **Target** project property group is used to define a script that is executed when the target enters run state. This can be when the application is run for the first time or when the **Debug | Go** operation has been carried out after the application has hit a breakpoint or been stopped using the **Debug | Break** operation. The aim of the run script is to carry out any target specific operations after the debugger has finished accessing target memory. This can be useful to re-enable caches previously disabled by the stop script.

See `TargetInterface` Object for a description of the `TargetInterface` object which is used by the run script to access the target hardware.

## TargetInterface Object

The **TargetInterface** object is used to access the currently connected target interface. The following section describes the **TargetInterface** object's member functions.

### TargetInterface.beginDebugAccess

**Synopsis** `TargetInterface.beginDebugAccess()`

**Description** Put target into debug state if it is not already in order to carry out a number of debug operations. The idea behind **beginDebugAccess** and **endDebugAccess** is to minimize the number of times the target enters and exits debug state when carrying out a number of debug operations. Target interface functions that require the target to be in debug state (such as peek and poke) also use **beginDebugAccess** and **endDebugAccess** to get the target into the correct state. A nesting count is maintained, incremented by **beginDebugAccess** and decremented by **endDebugAccess**. The initial processor state is recorded on the first nested call to **beginDebugAccess** and this state is restored when the final **endDebugAccess** is called causing the count to return to its initial state.

### TargetInterface.delay

**Synopsis** `TargetInterface.delay(milliseconds)`

**Description** **TargetInterface.delay** waits for **milliseconds** milliseconds

## TargetInterface.endDebugAccess

**Synopsis** `TargetInterface.endDebugAccess (alwaysRun)`

**Description** Restore the target run state recorded at the first nested call to **beginDebugAccess**. See **beginDebugAccess** for more information. If **alwaysRun** is non-zero the processor will exit debug state on the last nested call to **endDebugAccess**.

## TargetInterface.eraseBytes

**Synopsis** `TargetInterface.eraseBytes (address, length)`

**Description** **TargetInterface.eraseBytes** erases a block of erasable memory. **address** is the start address of the block to erase and **length** is the number of bytes to erase.

## TargetInterface.executeFunction

**Synopsis** `TargetInterface.executeFunction (address, r0, timeout)`

**Description** **TargetInterface.executeFunction** executes a function on the target. **address** is the address of the function entry point, **r0** is the value to set register r0 on entry to the function (in effect the first parameter to the function), and **timeout** is the timeout value in milliseconds to wait for the function to complete.

## TargetInterface.getRegister

**Synopsis** `TargetInterface.getRegister (register)`

**Description** **TargetInterface.getRegister** gets the value of a CPU register. Note that the set of register values are only updated when the CPU stops. **register** is a string specifying the register to get and must be one of r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15, sp, lr, pc, cpsr, r8\_fiq, r9\_fiq, r10\_fiq, r11\_fiq, r12\_fiq, r13\_fiq, r14\_fiq, spsr\_fiq, r13\_svc, r14\_svc, spsr\_svc, r13\_abt, r14\_abt, spsr\_abt, r13\_irq, r14\_irq, spsr\_irq, r13\_und, r14\_und, spsr\_und. **TargetInterface.getRegister** returns the register's value.

## TargetInterface.peekByte

**Synopsis** `TargetInterface.peekByte (address)`

**Description** **TargetInterface.peekByte** reads a byte of target memory from **address** and returns it.

## TargetInterface.peekBytes

**Synopsis** `TargetInterface.peekBytes (address, length)`

**Description** `TargetInterface.peekBytes` reads a block of bytes from target memory starting at `>address` for `length` bytes and returns the result as an array containing the bytes read.

## TargetInterface.peekUint16

**Synopsis** `TargetInterface.peekUint16(address)`

**Description** `TargetInterface.peekUint16` reads a 16-bit unsigned integer from target memory from `address` and returns it.

## TargetInterface.peekUint32

**Synopsis** `TargetInterface.peekUint32(address)`

**Description** `TargetInterface.peekUint32` reads a 32-bit unsigned integer from target memory from `address` and returns it.

## TargetInterface.peekWord

**Synopsis** `TargetInterface.peekWord(address)`

**Description** `TargetInterface.peekWord` reads a word as an unsigned integer from target memory from `address` and returns it.

## TargetInterface.pokeByte

**Synopsis** `TargetInterface.pokeByte(address, data)`

**Description** `TargetInterface.pokeByte` writes the byte `data` to `address` in target memory.

### Synopsis `TargetInterface.pokeUint16`

`TargetInterface.pokeUint16(address, data)`

**Description** `TargetInterface.pokeUint16` writes `data` as a 16-bit value to `address` in target memory.

### Synopsis `TargetInterface.pokeUint32`

`TargetInterface.pokeUint32(address, data)`

**Description** `TargetInterface.pokeUint32` writes `data` as a 32-bit value to `address` in target memory.

### Synopsis `TargetInterface.pokeWord`

`TargetInterface.pokeWord(address, data)`

**Description** `TargetInterface.pokeWord` writes `data` as a word value to `address` in target memory.

## TargetInterface.pokeBytes

### Synopsis

## TargetInterface.pokeBytes

```
TargetInterface.pokeBytes(address, data)
```

### Description

**TargetInterface.pokeBytes** writes the array **data** containing 8-bit data to target memory at **address**.

## TargetInterface.peekMultUint16

### Synopsis

```
TargetInterface.peekMultUint16(address, length)
```

### Description

**TargetInterface.peekMultUint16** reads **length** unsigned 16-bit integers from target memory starting at **address** and returns them as an array.

## TargetInterface.peekMultUint32

### Synopsis

```
TargetInterface.peekMultUint32(address, length)
```

### Description

**TargetInterface.peekMultUint32** reads **length** unsigned 32-bit integers from target memory starting at **address** and returns them as an array.

## TargetInterface.pokeMultUint16

### Synopsis

```
TargetInterface.pokeMultUint16(address, data)
```

### Description

**TargetInterface.pokeBytes** writes the array **data** containing 16-bit data to target memory at **address**.

## TargetInterface.pokeMultUint32

### Synopsis

```
TargetInterface.pokeMultUint32(address, data)
```

### Description

**TargetInterface.pokeBytes** writes the array **data** containing 32-bit data to target memory at **address**.

## TargetInterface.setICEBreakerBreakpoint

### Synopsis

```
TargetInterface.setICEBreakerBreakpoint(n, addressValue, addressMask,  
dataValue, dataMask controlValue, controlMask)
```

**Description** `TargetInterface.setICEBreakerBreakpoint` sets an ICEBreaker breakpoint. `nis` is the number of the watchpoint unit to use, `addressValue` is the address value, `addressMask` is the address mask, `dataValue` is the data value, `dataMask` is the data mask, `controlValue` is the control value, and `controlMask` is the control mask.

## TargetInterface.resetAndStop

**Synopsis** `TargetInterface.resetAndStop(delay)`

**Description** `TargetInterface.resetAndStop` resets the target by cycling nSRST and then stops the processor. The `delay` parameter specifies the number of milliseconds to hold the nSRST signal. This function allows the processor to run for a period of time before the processor is stopped which may cause unpredictable behaviour as the processor may be left in an undefined state if it has been allowed to start up from reset.

## TargetInterface.stopAndReset

**Synopsis** `TargetInterface.stopAndReset(delay)`

**Description** `TargetInterface.stopAndReset` resets the target by stopping the processor and then cycling nSRST. The `delay` parameter specifies the number of milliseconds to hold the nSRST signal. The difference between this function and `TargetInterface.resetAndStop` is that this function attempts to keep the processor stopped whilst it is reset and therefore allows you to start your program with the processor in a known state. Some ARM targets however can't do this as the reset also resets the debug interface which allows the processor to run when it is released from reset. On these targets this function behaves in the same way as `TargetInterface.resetAndStop`.

## TargetInterface.setNSRST

**Synopsis** `TargetInterface.setNSRST(state)`

**Description** `TargetInterface.setNSRST` sets the level of the target's nSRST reset signal high if state is non-zero.

## TargetInterface.setRegister

**Synopsis** `TargetInterface.setRegister(register, value)`

**Description** `TargetInterface.setRegister` sets the CPU register `register` to `value`.

**register** is a string describing the register to set and must be one of r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13 or sp, r14 or lr, r15 or pc, cpsr, r8\_fiq, r9\_fiq, r10\_fiq, r11\_fiq, r12\_fiq, r13\_fiq, r14\_fiq, spsr\_fiq, r13\_svc, r14\_svc, spsr\_svc, r13\_abt, r14\_abt, spsr\_abt, r13\_irq, r14\_irq, spsr\_irq, r13\_und, r14\_und, spsr\_und.

Note that this function will only change the CPU register state if the CPU is stopped.

## TargetInterface.trst

**Synopsis** `TargetInterface.trst()`

**Description** **TargetInterface.trst** performs a JTAG TAP reset.

## TargetInterface.waitForDebugState

**Synopsis** `TargetInterface.waitForDebugState(timeout)`

**Description** **TargetInterface.waitForDebugState** waits for the target to enter debug state with a timeout of **timeout** milliseconds. If the timeout period expires and exception is thrown which is caught by the debugger.

# Program loading

CrossStudio for ARM supports FLASH programming (and subsequent debugging) by loading a program into the RAM of the target and transmitting it the data to be programmed.

The use of a target loader is determined by the value of the **Loader File Path** project property defined for the appropriate configuration of the project. The **Loader File Path** property specifies the location of the loader executable to use, if this is defined the loader executable will be downloaded onto the target and run prior to download of the main application.

In addition to the **Loader File Path** property, the **Loader File Type** project property must be specified. This tells CrossStudio how to communicate with the loader program. The various communication mechanisms available are explained in more detail later. The **Load File Type** property may be set to one of the following:

- **LIBMEM RPC Loader**— Calls to LIBMEM library functions will be made directly using a remote procedure call mechanism.
- **Comms Channel Loader** — The ARM debug comms channel is used to communicate with the loader.

- **Fast Comms Channel Loader** — The ARM debug comms channel is used to communicate with the loader. This scheme is significantly faster at downloading than **Comms Channel Loader** because it makes the assumption that the loader program is always ready to read data and therefore does not check the ARM comms channel status before transmitting data. This may not be suitable for all targets or loaders. If you experience reliability problems downloading and verifying programs using this setting, you should revert to the **Comms Channel Loader** setting.
  - **RAM Loader** — The targets RAM is used to communicate with the loader.
- To write your own loader programs you should use the LIBMEM loader library.

## ARM Device Specific Target Support





### Debug file search editor

When a program is built with debugging enabled the debugging information contains paths describing where the source files that went into the program are located in order to allow the debugger to find them. If a program or libraries linked into the program are being run on a different machine to the one they were compiled on or if the source files have moved since the program was compiled, the debugger will be unable to find the source files.

In this situation the simplest way to help CrossStudio find the moved source files is to add the directory containing the source file to one of its source file search paths. Alternatively, if CrossStudio cannot find a source file it will prompt you for its location and record its new location in its source file map.

### Debug source file search paths

The debug source file search paths can be used to help the debugger locate source files that are no longer in the same location as they were at compile time. When a source file cannot be found, the search path directories will be checked in turn to see if they contain the source file. CrossStudio maintains two debug source file search paths:

- **Project session search path** This path is set in the current project session and does not apply to all projects.
- **The global search path** This path is set system wide and applies to all projects.

The project session search path is checked before the global search path.

### To view and edit the debug search paths

- From the **Debug** menu, click **Edit Search Paths**

## Debug source file map

If a source file cannot be found whilst debugging and the debugger has to prompt the user for its location, the results are stored in the debug source file map. The debug source file map is simply a mapping between the original file name and its new location. When a file cannot be found at its original location or in the debug search paths the debug source file map is checked to see if a new location for the file has been recorded or if the user has specified that the file does not exist. Each project session maintains its own source file map, the map is not shared by all projects.

### To view the debug source file map

- From the **Debug** menu, click **Edit Search Paths**

### To remove individual entries from the debug source file map

- From the **Debug** menu, click **Edit Search Paths**
- Right click on the mapping you want to delete
- From the context menu, click **Delete Mapping**

### To remove all entries from the debug source file map

- From the **Debug** menu, click **Edit Search Paths**
- Select **Delete All Mappings** from the context menu

## Environment options

### Environment General

- **Window menu contains.** Specifies the maximum number of open windows to be listed in the **Window** menu.
- **Most recently used project list contains.** Specifies the maximum number of project files to be listed in the **File > Recent Projects** menu.
- **Most recently used files list contains.** Specifies the maximum number of files to be listed in the **File > Recent Files** menu.
- **Use large icons in toolbars.** Enables large icons in tool bars.
- **Show status bar.** Enables display of status bar.

- **Show full path in title bar.** Enables display in title bar of full file path of current file being edited.
- **Show dock window contents when moving.** Enables display of docking window contents when window is being moved.
- **Docking windows show toolbars.** Configures docking window toolbar display.
- **Projects location.** Specifies the default directory location of projects.

### Environment Workspace

The workspaces dialog provides the ability to specify which windows and toolbars are displayed in Full Screen, Normal and Debug run states.

### Text Editor General

- **Vertical scroll bar.** Enables display of vertical scroll bar.
- **Horizontal scroll bar.** Enables display of horizontal scroll bar.
- **Indicator margin.** Enables display of indicator margin.
- **Margins enabled.** Enables margins.
- **Top margin.** Size in lines of top margin.
- **Left margin.** Size in columns of left margin.
- **Bottom margin.** Size in lines of bottom margin.
- **Right margin.** Size in columns of right margin.
- **Hide mouse cursor when typing.** Enables hiding of mouse cursor when typing.
- **Use I-beam text cursor.** Enables I-beam cursor when mouse is moved over editor.
- **Allow editing or read only files.** Enables the editing of read only files.
- **Insert mode style.** Specifies the caret style when the editor is in insert mode.
- **Overtyping mode style.** Specifies the caret style when the editor is in overwrite mode.

### Text Editor Indent

- **File type.** The type of file to configure the indent options for.
- **Insert spaces.** Enables insertion of spaces only.
- **Tab size.** Specifies the tab size in characters.
- **Keep tabs.** Enables use of tab characters.
- **Indent size.** Specifies the default indentation size in characters.
- **Auto indent.** Specifies indent mode.

- **Indent open brace.** Enables indentation of open braces when in smart indent mode.
- **Indent closing brace.** Enables indentation of closing braces when in smart indent mode.
- **Previous lines used for context.** Specifies the maximum number of lines prior to the current line to start parsing when in smart indent mode.

### **Build General**

- **Echo command lines to log.** Enables echoing of build command lines to output window.
- **Show build information in log.** Enables build avoidance logic to be displayed.

### **Debugger General**

- **Default data display.** Specifies the default data display format.
- **Source file search path.** Comma separated list of directories to use to locate source files.

### **Debugger Data Tips**

- **Limit data tip array display to  $n$  elements.** Specifies the maximum number of array elements to display when showing array data tips.
- **Display extended data tips.** Enables display of extended data tips. Extended data tips display the data in a number of formats.
- **Extended data tip formats.** Specifies the formats to display when displaying extended data tips is enabled.

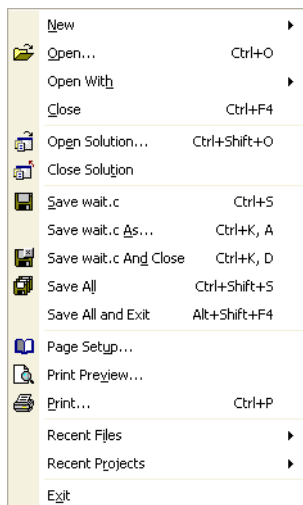
## CrossStudio menu summary

The following sections describe each menu and each menu item.

### File menu

The **File** menu provides commands to create, open, and close files, and to print them.

#### The File menu



## File commands

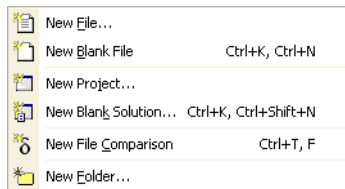
Menu command	Keystroke	Description
New		Displays the <b>New</b> menu.
Open	Ctrl+O	Opens an existing file for editing.
Open With		Displays the <b>Open With</b> menu.
Close	Ctrl+F4	Closes the active editor. If you have made changes to the file, CrossStudio prompts you to save the file.
Open Solution	Ctrl+Shift+O	Opens an existing solution for editing. If you already have an open solution, CrossStudio will close it before opening the new solution and, if you have made changes to any of the files in your solution, you are prompted to save each of them.
Close Solution		Closes the current solution. If you have made changes to any of the files in your solution, you are prompted to save each of them.
Save <i>file</i>	Ctrl+S	Saves the contents of the active editor to disk. If it is a new file without a name, CrossStudio opens a file browser for you to choose where to save the file and what to call it.
Save <i>file</i> As...	Ctrl+K, A	Saves the contents of the active editor to disk using a different name. CrossStudio opens a file browser for you to choose where to save the file and what to call it. After saving, the editor is set to edit the newly saved file, not the previous file.
Save <i>file</i> And Close	Ctrl+K, D	Saves the contents of the active editor to disk and then closes the editor. If it is a new file without a name, CrossStudio opens a file browser for you to choose where to save the file and what to call it.
Save All	Ctrl+Shift+S	Saves all edited files to disk. For each new file without a name, CrossStudio opens a file browser for you to choose where to save the file and what to call it. Cancelling a save at any time will return you to CrossStudio without saving the remainder of the files.
Save All And Exit	Alt+Shift+F4	Saves all edited files to disk and then exits CrossStudio. For each new file without a name, CrossStudio opens a file browser for you to choose where to save the file and what to call it. Cancelling a save at any time will return you to CrossStudio without exiting.
Page Setup...		Steps into the next statement or instruction and enters C functions and assembly language subroutines. If a breakpoint is hit when stepping, the debugger immediately stops at that breakpoint.
Print Preview...		Opens the <b>Print Preview</b> dialog and shows the document as it will appear when it is printed.

Menu command	Keystroke	Description
Recent Files		Opens the <b>Recent Files</b> menu which contains a list of files that have been recently opened, with the most recently opened file first in the list. You can configure the number of files retained in the <b>Recent Files</b> menu in the <b>Environment Options</b> dialog. You can clear the list of recent files by selecting <b>Clear Recent Files List</b> from the <b>Recent Files</b> menu.
Recent Projects		Opens the <b>Recent Projects</b> menu which contains a list of projects that have been recently opened, with the most recently opened project first in the list. You can configure the number of projects retained in the <b>Recent Projects</b> menu in the <b>Environment Options</b> dialog. You can clear the list of recent projects by selecting <b>Clear Recent Projects List</b> from the <b>Recent Projects</b> menu.
Exit	Alt+F4	Saves all edited files, closes the solution, and exits CrossStudio. For each new file without a name, CrossStudio opens a file browser for you to choose where to save the file and what to call it. Cancelling a save at any time will return you to CrossStudio without exiting.

## New menu

The **New** menu provides commands to create files and folders.

### The New menu



### New menu commands

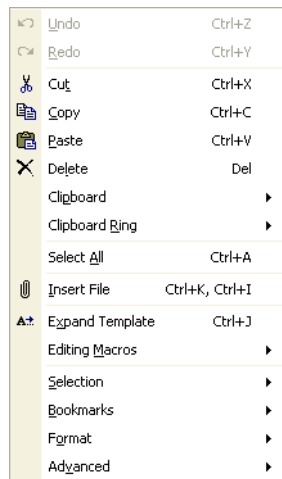
Menu command	Keystroke	Description
New File...		Creates a new file using the <b>New File</b> dialog
New Blank File	Ctrl+K, Ctrl+N	Creates a new, unnamed document.
New Project...		Creates a new project using the <b>New Project</b> dialog.
New Blank Solution	Ctrl+K, Ctrl+Shift+N	Creates a new solution containing no projects.

Menu command	Keystroke	Description
New File Comparison	Ctrl+T, F	Creates a new file comparison window.
New Folder...		Creates a new folder underneath the currently selected item in the <b>Project Explorer</b> .

## Edit menu

The **Edit** menu provides commands to edit files.

### The Edit menu



### Edit menu commands

Menu command	Keystroke	Description
Undo	Ctrl+Z —or— Alt+Backspace	Undoes the last editing action.
Redo	Ctrl+Y —or— Alt+Shift+Backspace	Redoes the last undone editing action.
Cut	Ctrl+X —or— Shift+Delete	Cuts the selected text to the clipboard. If no text is selected, cuts the current line to the clipboard.

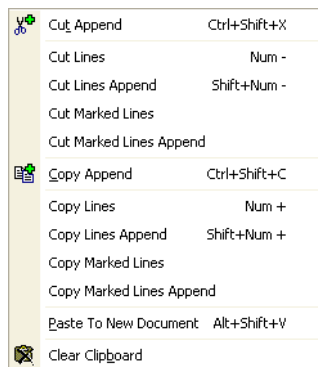


Menu command	Keystroke	Description
Copy	Ctrl+C —or— Ctrl+Insert	Cuts the selected text to the clipboard. If no text is selected, copies the current line to the clipboard.
Paste	Ctrl+V —or— Shift+Insert	Pastes the clipboard into the document.
Delete	Delete	Deletes the selection. If no text is selected, deletes the character to the right of the cursor.
Clipboard		Displays the <b>Clipboard</b> menu.
Clipboard Ring		Displays the <b>Clipboard Ring</b> menu.
Select All	Ctrl+A	Selects all text or items in the document.
Insert File	Ctrl+K, Ctrl+I	Inserts a file into the document at the cursor position.
Expand Template	Ctrl+J	Forces expansion of a template.
Editing Macros		Displays the <b>Editing Macros</b> menu.
Selection		Displays the <b>Edit Selection</b> menu. See <b>Edit Selection menu</b> (page 180).
Bookmarks		Displays the <b>Bookmarks</b> menu.
Format		Displays the <b>Formatting</b> menu.
Advanced		Displays the <b>Advanced Editing</b> menu.

## Clipboard menu

The **Clipboard** menu provides commands to edit files using the clipboard.

## The Clipboard menu



### Clipboard menu commands





Menu command	Keystroke	Description
Cut Append	Ctrl+Shift+X	Cuts the selected text and appends it to the clipboard. If no text is selected, cuts and appends the current line to the clipboard.
Cut Lines	Num -	Converts the selection to complete lines then cuts the selected text lines them to the clipboard. If no text is selected, cuts and appends the current line to the clipboard.
Cut Lines Append	Shift+Num -	Converts the selection to complete lines then cuts the selected text lines and appends them to the clipboard. If no text is selected, cuts and appends the current line to the clipboard.
Cut Marked Lines		Cuts all bookmarked lines in the current document to the clipboard.
Cut Marked Lines Append		Cuts all bookmarked lines in the current document and appends them to the clipboard.
Copy Append		Copies the selected text and appends it to the clipboard. If no text is selected, copies and appends the current line to the clipboard.
Copy Lines		Converts the selection to complete lines then copies the selected text lines them to the clipboard. If no text is selected, copies and appends the current line to the clipboard.
Copy Lines Append		Converts the selection to complete lines then copies the selected text lines and appends them to the clipboard. If no text is selected, copies and appends the current line to the clipboard.
Copy Marked Lines		Copies all bookmarked lines in the current document to the clipboard.
Copy Marked Lines Append		Copies all bookmarked lines in the current document and appends them to the clipboard.

Menu command	Keystroke	Description
Paste to New Document	Alt+Shift+V	Creates a new, unnamed document and pastes the clipboard into it.
Clear Clipboard		Clears the contents of the clipboard.

## Clipboard Ring menu

The **Clipboard Ring** menu provides commands to edit files using the clipboard ring.

### The Clipboard Ring menu

 Paste All	Ctrl+R, Ctrl+W
 Cycle Clipboard Ring	Ctrl+Shift+W
 Clear Clipboard Ring	Ctrl+R, Del
 Clipboard Ring	Ctrl+Alt+C






### Clipboard Ring menu commands

Menu command	Keystroke	Description
Paste All	Ctrl+Shift+X	Pastes the contents of the clipboard ring to the current document.
Cycle Clipboard Ring	Num -	Cycles the clipboard ring.
Clear Clipboard Ring	Ctrl+R, Del	Clears the contents of the clipboard ring.
Clipboard Ring	Ctrl+Alt+C	Displays the Clipboard Ring window. See <b>Clipboard ring window</b> (page 100).

## Macros menu

The **Macros** menu provides additional commands to record and play key sequences as well as provide some fixed macros.

## The Macros menu

 Play Recording	Ctrl+Shift+P
 Start Recording	Ctrl+Shift+R
 Pause/Resume Recording	
 Stop Recording	
 Cancel Recording	
<hr/>	
Insert Hard Tab	Ctrl+Q, Tab
Declare Or Cast To "char"	Ctrl+Q, Ctrl+C
Declare Or Cast To "short"	Ctrl+Q, Ctrl+S
Declare Or Cast To "int"	Ctrl+Q, Ctrl+I
Declare Or Cast To "long"	Ctrl+Q, Ctrl+L
Declare Or Cast To "void"	Ctrl+Q, Ctrl+V
<hr/>	
Insert "const"	Ctrl+Q, Ctrl+K
Insert "volatile"	Ctrl+Q, Ctrl+O
Insert "extern"	Ctrl+Q, Ctrl+X















## Macros menu commands

Menu command	Keystroke	Description
Play Recording	Ctrl+Shift+P	Plays the last recorded keyboard macro.
Start Recording	Ctrl+Shift+R	Starts recording a keyboard macro.
Pause/Resume Recording		Temporarily pauses a recording a keyboard macro. If already paused, recommences recording of the keyboard macro.
Stop Recording		Stops recording a keyboard macro and saves it. Note that when recording has commenced, the keystroke to stop recording the keyboard macro is Ctrl+Shift+R.
Cancel Recording		Cancels recording without changing the current keyboard macro.
Insert Hard Tab	Ctrl+Q, Tab	Inserts a tab character into the document even if the documents language settings inserts tabs as spaces.
Declare Or Cast to <i>type</i>		If there is a selection, parentheses are placed around the selection and that expression is cast to <i>type</i> . If there is no selection, <i>type</i> is inserted into the document.
Insert <i>keyword</i>		Inserts <i>keyword</i> into the document, followed by a space.

## Edit Selection menu

The **Edit > Selection** menu provides commands to operate on the selection.

## The Edit Selection menu

	Tabify	Ctrl+K, Tab
	Untabify	Ctrl+K, Space
	Make Uppercase	Ctrl+Shift+U
	Make Lowercase	Ctrl+U
	Switch Case	Alt+Shift+U
	Comment	Ctrl+/ /
	Uncomment	Ctrl+Shift+/ /
	Increase Line Indent	Tab
	Decrease Line Indent	Shift+Tab
	Align Left	Ctrl+K, Ctrl+J, L
	Center	Ctrl+K, Ctrl+J, C
	Align Right	Ctrl+K, Ctrl+J, R
	Sort Ascending	
	Sort Descending	

### Edit Selection menu commands







Menu command	Keystroke	Description
Tabify	Ctrl+K, Tab	Convert space characters in the selection to tabs according to the tab settings for the language.
Untabify	Ctrl+K, Space	Convert tab characters in the selection to spaces according to the tab settings for the language.
Make Uppercase	Ctrl+Shift+U	Convert the letters in the selection to uppercase. If there is no selection, CrossStudio converts the character to the right of the cursor to uppercase and moves the cursor right one character.
Switch Case	Ctrl+U	Switches the letter case of letters in the selection; that is, uppercase characters become lowercase, and lowercase become uppercase. If there is no selection, CrossStudio switches the letter case of the character to the right of the cursor and moves the cursor right one character.
Comment	Ctrl+/ /	Prefixes lines in the selection with language-specific comment characters. If there is no selection, CrossStudio comments the cursor line.
Uncomment	Ctrl+Shift+/ /	Removes the prefixed from lines in the selection that contains language-specific comment characters. If there is no selection, CrossStudio uncomments the cursor line.
Increase Line Indent	Tab	Increases the line indent of the selection. If there is no selection, the cursor is moved to the next tab stop by inserting spaces or a tab character according to the tab settings for the document.
Decrease Line Indent	Shift+Tab	Decreases the line indent of the selection. If there is no selection, the cursor is moved to the previous tab stop.

Menu command	Keystroke	Description
Align Left	Ctrl+K, Ctrl+J, L	Aligns all text in the selection to the leftmost non-blank character in the selection.
Align Center	Ctrl+K, Ctrl+J, C	Centers all text in the selection between the leftmost and rightmost non-blank characters in the selection.
Align Right	Ctrl+K, Ctrl+J, R	Aligns all text in the selection to the rightmost non-blank character in the selection.
Sort Ascending		Sorts the selection into ascending lexicographic order.
Sort Descending		Sorts the selection into descending lexicographic order.

## Bookmarks menu

The **Bookmarks** menu provides commands to drop and find temporary bookmarks.

### The Bookmarks menu

	Toggle Bookmark	Ctrl+F2
	Next Bookmark	F2
	Previous Bookmark	Shift+F2
	First Bookmark	Ctrl+K, F2
	Last Bookmark	Ctrl+K, Shift+F2
	Clear All Bookmarks	Ctrl+Shift+F2










### Bookmarks menu commands

Menu command	Keystroke	Description
Toggle Bookmark	Ctrl+F2	Inserts or removes a bookmark on the cursor line.
Next Bookmark	F2	Moves the cursor to the next bookmark in the document. If there is no following bookmark, the cursor is placed at the first bookmark in the document.
Previous Bookmark	Shift+F2	Moves the cursor to the previous bookmark in the document. If there is no previous bookmark, the cursor is placed at the last bookmark in the document.
First Bookmark	Ctrl+K, F2	Moves the cursor to the first bookmark in the document.
Last Bookmark	Ctrl+K, Shift+F2	Moves the cursor to the last bookmark in the document.
Clear All Bookmarks	Ctrl+Shift+F2	Removes all bookmarks from the document.

## Advanced menu

The **Advanced** menu provides additional commands to edit your document.

### The Advanced menu

	Undo All	Ctrl+K, Ctrl+Z
	Redo All	Ctrl+K, Ctrl+Y
	Transpose Words	Ctrl+K, Ctrl+T
	Transpose Lines	Ctrl+K, T
	Scroll To Top	Ctrl+G, Ctrl+T
	Scroll To Middle	Ctrl+G, Ctrl+M
	Scroll To Bottom	Ctrl+G, Ctrl+B
	Toggle Read Only	Ctrl+K, Ctrl+R
	Visible Whitespace	Ctrl+Shift+8

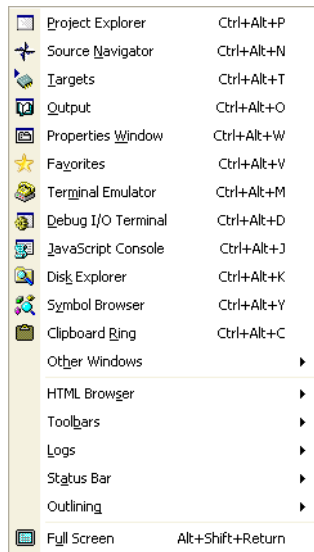
### Advanced menu commands

Menu command	Keystroke	Description
Undo All	Ctrl+K, Ctrl+Z	Undoes all editing actions in the document.
Redo All	Ctrl+K, Ctrl+Y	Redoes all editing actions in the document.
Transpose Words	Ctrl+K, Ctrl+T	Swaps the word at the cursor position with the preceding word.
Transpose Lines	Ctrl+K, T	Swaps the cursor line with the preceding line.
Scroll To Top	Ctrl+G, Ctrl+T	Moves the cursor line to the top of the window.
Scroll To Middle	Ctrl+G, Ctrl+M	Moves the cursor line to the middle of the window.
Scroll To Bottom	Ctrl+G, Ctrl+B	Moves the cursor line to the bottom of the window.
Toggle Read Only	Ctrl+K, Ctrl+R	Toggles the read only bit of the document.
Visible Whitespace	Ctrl+Shift+8	Toggles the document display between non-visible whitespace and visible whitespace where tabs and spaces are shown with special characters.

## View menu

The **View** menu provides commands to control the way that windows and their contents are seen within CrossStudio.

## The View menu



## View menu commands

Menu command	Keystroke	Description
Project Explorer	Ctrl+Alt+P	Activates the <b>Project Explorer</b> . See <b>Project explorer</b> (page 128).
Source Navigator	Ctrl+Alt+N	Activate the <b>Source Navigator</b> . See <b>Source Navigator</b> .
Targets	Ctrl+Alt+T	Activates the <b>Targets</b> window. See <b>Targets window</b> (page 139).
Output	Ctrl+Alt+O	Activates the <b>Output</b> window. See <b>Output window</b> (page 127).
Properties Window	Ctrl+Alt+W	Activates the <b>Properties</b> window. See <b>Properties window</b> (page 130).
Favorites	Ctrl+Alt+V	Activates the <b>Favorites</b> window. See <b>Favorites Window</b> .
Terminal Emulator	Ctrl+Alt+M	Activates the <b>Terminal Emulator</b> window. See <b>Terminal Emulator Window</b> .
Debug Console	Ctrl+Alt+D	Activates the <b>Debug Console</b> window.
JavaScript Console	Ctrl+Alt+J	Activates the <b>JavaScript Console</b> window.
Symbol Browser	Ctrl+Alt+Y	Activates the <b>Symbol Browser</b> window. See <b>Symbol browser</b> (page 133).
Clipboard Ring	Ctrl+Alt+C	Activates the <b>Clipboard Ring</b> window.
Other Windows		Displays the <b>Other Windows</b> menu.
HTML Browser		Displays the <b>HTML Browser</b> menu.




















Menu command	Keystroke	Description
Logs		Displays the <b>Logs</b> menu.
Status Bar		Displays the <b>Status Bar</b> menu.
Outlining		Displays the <b>Outlining</b> menu.
Full Screen	Alt+Shift+Return	Activates the <b>Full Screen</b> workspace.

## Other Windows menu

The **Other Windows** menu provides commands to activate additional windows in CrossStudio.

### The Other Windows menu

 Breakpoints	Ctrl+Alt+B
 Call Stack	Ctrl+Alt+S
 Locals	Ctrl+Alt+L
 Globals	Ctrl+Alt+G
 Threads	Ctrl+Alt+D
 Registers 1	Ctrl+T, R, 1
 Registers 2	Ctrl+T, R, 2
 Registers 3	Ctrl+T, R, 3
 Registers 4	Ctrl+T, R, 4
 Watch 1	Ctrl+T, W, 1
 Watch 2	Ctrl+T, W, 2
 Watch 3	Ctrl+T, W, 3
 Watch 4	Ctrl+T, W, 4
 Memory 1	Ctrl+T, M, 1
 Memory 2	Ctrl+T, M, 2
 Memory 3	Ctrl+T, M, 3
 Memory 4	Ctrl+T, M, 4
 Execution Trace	
 Execution Counts	

### Other Windows commands

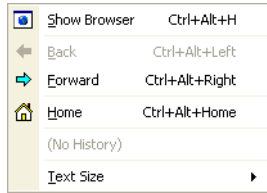
Menu command	Keystroke	Description
Breakpoints	Ctrl+Alt+B	Activates the <b>Breakpoints</b> window. See <b>Breakpoints window</b> (page 102).
Call Stack	Ctrl+Alt+S	Activates the <b>Call Stack</b> window. See <b>Call stack window</b> (page 107).

Menu command	Keystroke	Description
Locals	Ctrl+Alt+L	Activates the <b>Locals</b> window. See <b>Locals window</b> (page 114).
Globals	Ctrl+Alt+G	Activates the <b>Globals</b> window. See <b>Globals window</b> (page 112).
Threads	Ctrl+Alt+D	Activates the <b>Threads</b> window. See <b>Threads window</b> (page 120).
Registers 1	Ctrl+T, R, 1	Activates the first <b>Register</b> window. See <b>Register windows</b> (page 118).
Registers 2	Ctrl+T, R, 2	Activates the second <b>Register</b> window. See <b>Register windows</b> (page 118).
Registers 3	Ctrl+T, R, 3	Activates the third <b>Register</b> window. See <b>Register windows</b> (page 118).
Registers 4	Ctrl+T, R, 4	Activates the fourth <b>Register</b> window. See <b>Register windows</b> (page 118).
Watch 1	Ctrl+T, W, 1	Activates the first <b>Watch</b> window. See <b>Watch window</b> (page 122).
Watch 2	Ctrl+T, W, 2	Activates the second <b>Watch</b> window. See <b>Watch window</b> (page 122).
Watch 3	Ctrl+T, W, 3	Activates the third <b>Watch</b> window. See <b>Watch window</b> (page 122).
Watch 4	Ctrl+T, W, 4	Activates the fourth <b>Watch</b> window. See <b>Watch window</b> (page 122).
Memory 1	Ctrl+T, M, 1	Activates the first <b>Memory</b> window. See <b>Memory window</b> (page 116).
Memory 2	Ctrl+T, M, 2	Activates the second <b>Memory</b> window. See <b>Memory window</b> (page 116).
Memory 3	Ctrl+T, M, 3	Activates the third <b>Memory</b> window. See <b>Memory window</b> (page 116).
Memory 4	Ctrl+T, M, 4	Activates the fourth <b>Memory</b> window. See <b>Memory window</b> (page 116).
Execution Trace		Activates the <b>Execution Trace</b> window. See <b>Trace window</b> (page 122).
Execution Counts		Activates the <b>Execution Counts</b> window. See <b>Execution counts window</b> (page 111).

## Browser menu

The **Browser** menu provides commands navigate through the browser history.

## The Browser menu



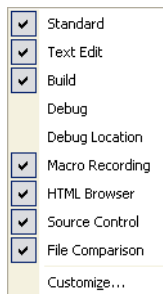
### Browser commands

Menu command	Keystroke	Description
Show Browser	Ctrl+Alt+H	Activates the <b>Browser</b> window.
Back	Ctrl+Alt+Left	Displays the previous page in the browser history.
Forward	Ctrl+Alt+Right	Displays the following page in the browser history.
Home	Ctrl+Alt+Home	Displays the home page.
Text Size		Displays the <b>Browser Text Size</b> menu.

## Toolbars menu

The **Toolbars** menu provides commands to display or hide CrossStudio tool bars.

### The Toolbars menu



### Toolbar menu commands


















Menu command	Keystroke	Description
Standard		Displays the <b>Standard</b> tool bar.

Menu command	Keystroke	Description
Text Edit		Displays the <b>Text Edit</b> tool bar.
Build		Displays the <b>Build</b> tool bar.
Debug		Displays the <b>Debug</b> tool bar.
Debug Location		Displays the <b>Debug Location</b> tool bar.
Macro Recording		Displays the <b>Macro Recording</b> tool bar.
HTML Browser		Displays the <b>HTML Browser</b> tool bar.
Source Control		Displays the <b>Source Control</b> tool bar.
File Comparison		Displays the <b>File Comparison</b> tool bar.
Customize...		Displays the <b>Toolbar Configuration</b> dialog.

## Search menu

The **Search** menu provides commands to search in files.

### The Search menu

	Find...	Ctrl+F
	Find in Files...	Ctrl+Shift+F
	Replace...	Ctrl+H
	Replace in Files...	Ctrl+Shift+H
	Find Next	F3
	Find Previous	Shift+F3
	Find Selected Text	Ctrl+F3
	Find and Mark All	Alt+Shift+F3
	Go To Line...	Ctrl+G, Ctrl+L
	Go To Mate	Ctrl+] ]
	Next Location	F4
	Previous Location	Shift+F4
	Next Function	Ctrl+PgDown
	Previous Function	Ctrl+PgUp
	Case Sensitive Matching	Ctrl+K, Ctrl+F, C
	Whole Word Matching	Ctrl+K, Ctrl+F, W
	Regular Expression Matching	Ctrl+K, Ctrl+F, X

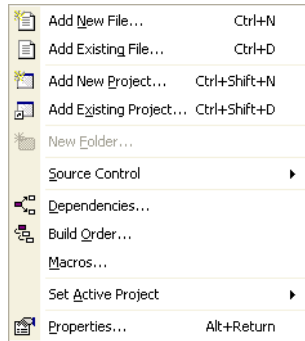
## Search menu commands

Menu command	Keystroke	Description
Find	Ctrl+F	Searches documents for strings.
Find in Files	Ctrl+Shift+F	Searches for a string in multiple files.
Replace	Replace	Replaces text with different text.
Replace in Files	Ctrl+Shift+H	Replaces text with different text in multiple files.
Find Next	F3	Searches for the next occurrence of the specified text.
Find Previous	Shift+F3	Searches for the previous occurrence of the specified text.
Find Selected Text	Ctrl+F3	Searches for the next occurrence of the selection.
Find and Mark All	Alt+Shift+F3	Searches the document for all occurrences of the specified text and marks them with bookmarks.
Go To Line	Ctrl+G, Ctrl+L	Moves the cursor to a specified line in the document.
Go To Mate	Ctrl+] ]	Moves the cursor to the bracket, parenthesis, or brace that matches the one at the cursor.
Next Location	F4	Moves the cursor to the line containing the next error or tag.
Previous Location	Shift+F4	Moves the cursor to the line containing the previous error or tag.
Next Function	Ctrl+PgDn	Moves the cursor to the declaration of the next function.
Previous Function	Ctrl+PgUp	Moves the cursor to the declaration of the previous function.
Case Sensitive Matching	Ctrl+K, Ctrl+F, C	Enables or disables the case sensitivity of letters when searching.
Whole Word Matching	Ctrl+K, Ctrl+F, W	Enables or disables whole word matching when searching.
Regular Expression Matching	Ctrl+K, Ctrl+F, X	Enables or disables expression matching rather than plain text matching.

## Project menu

The **Project** menu provides commands to manipulate the project.

## The Project menu



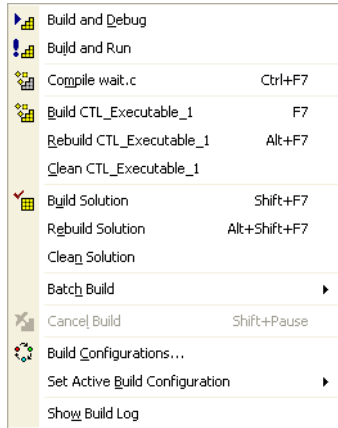
### Project menu commands

Menu command	Keystroke	Description
Add New File...	Ctrl+N	Adds a new file to the active project.
Add Existing File...	Ctrl+D	Adds an existing file to the active project.
Add New Project...	Ctrl+Shift+N	Adds a new project to the solution.
Add Existing Project	Ctrl+Shift+D	Adds a link to an existing project to the solution.
New Folder...		Adds a new folder to the current project or folder.
Source Control		Displays the <b>Source Control</b> menu.
Dependencies...		Displays the <b>Project Dependencies</b> dialog to alter project dependencies.
Build Order...		Displays the <b>Build Order</b> tab of the <b>Project Dependencies</b> dialog.
Macros...		Displays the <b>Project Macros</b> dialog to edit the macros defined in a project.
Set Active Project		Displays a menu which allows you to select the active project.
Properties	Alt+Return	Displays the <b>Project Properties</b> dialog for the current project item.

## Build menu

The **Build** menu provides commands to build projects and solutions.

## The Build menu



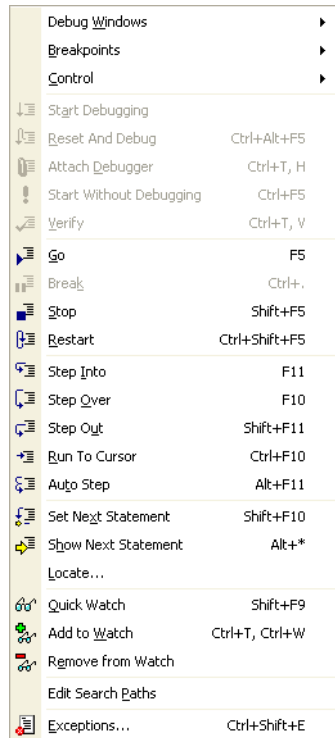
### Build menu commands

Menu command	Keystroke	Description
Build and Debug		Builds the active project and starts debugging it.
Build and Run		Builds the active project and runs it without debugging.
Compile <i>file</i>	Ctrl+F7	Compiles the selected project file.
Build <i>project</i>	F7	Builds the active project.
Rebuild <i>project</i>	Alt+F7	Rebuilds the active project.
Clean <i>project</i>		Removes all output and temporary files generated by the active project.
Build Solution	Shift+F7	Builds all projects in the solution.
Rebuild Solution	Alt+Shift+F7	Rebuilds all projects in the solution.
Clean Solution		Removes all output and temporary files generated by all projects in the solution.
Batch Build		Displays the <b>Batch Build</b> menu.
Cancel Build	Shift+Pause	Stops any build in progress.
Build Configurations...		Displays the <b>Build Configurations</b> dialog.
Set Active Build Configuration		Displays a menu which allows you to select the active build configuration.
Show Build Log		Displays the <b>Build Log</b> in the <b>Output</b> window.

# Debug menu

The **Debug** menu provides commands to download, run, and debug your application. You can find common debug actions as tool buttons on the **Debug** toolbar.

## The Debug menu



## The Debug toolbar



## Debug commands

Menu command	Keystroke	Description
Debug Windows		Displays the <b>Debug Windows</b> menu. See <b>Debug Windows menu</b> (page 196).
Breakpoints		Displays the <b>Breakpoints</b> menu. See <b>Breakpoint menu</b> (page 195).
Control		Displays the <b>Debug Control</b> menu. See <b>Debug Control menu</b> (page 194).












Menu command	Keystroke	Description
Start Debugging	F5	Downloads the program to the selected target interface and starts running the program under control of the debugger.
Reset and Debug	Ctrl+Alt+F5	Resets the selected target interface without downloading the project and starts running the program.
Attach Debugger	Ctrl+T, H	Attaches the debugger to the program running on the selected target interface.
Start Without Debugging	Ctrl+F5	Downloads the program to the selected target interface and starts running the program without the debugger.
Go	F5	Continues running the program until a breakpoint is hit or a hardware exception is raised.
Break	Ctrl+.	Stops the program running and returns control to the debugger.
Stop	Shift+F5	Stops debugging the program and returns to the editing workspace.
Restart	Ctrl+Shift+F5	Resets the selected target interface and starts debugging the program.
Step Into	F11	Steps into the next statement or instruction and enters C functions and assembly language subroutines. If a breakpoint is hit when stepping, the debugger immediately stops at that breakpoint.
Step Over	F10	Steps over the next statement or instruction without entering C functions and assembly language subroutines. If a breakpoint is hit when stepping, the debugger immediately stops at that breakpoint.
Step Out	Shift+F11	Steps out of the function or subroutine by executing up to the instruction following the call to the current function or subroutine. If a breakpoint is hit when stepping, the debugger immediately stops at that breakpoint.
Run To Cursor	Ctrl+F10	Runs the program to the statement or instruction the cursor is at. If a breakpoint is hit when stepping, the debugger immediately stops at that breakpoint.
Auto Step	Alt+F11	Animates program execution by running the program and updates all debugger windows after each statement or instruction executed.
Set Next Statement	Shift+F10	Sets the program counter to the statement or instruction that the cursor is on. Note that doing this may lead to unpredictable or incorrect execution of your program.
Show Next Statement	Alt+*	Displays the source line or instruction associated with the program counter. You can use this to show the execution point after navigating through files.
Locate...		

Menu command	Keystroke	Description
Quick Watch	Shift+F9	Opens a viewer on the variable or expression at the cursor position. If no text is selected, CrossStudio opens a viewer using the word at the cursor position as the expression. If some text is selected, CrossStudio opens a viewer using the selected text as the expression.
Add To Watch	Ctrl+T, Ctrl+W	Adds the variable or expression at the cursor position to the last activated watch window. If no text is selected, CrossStudio adds the word at the cursor position to the watch window. If some text is selected, CrossStudio adds the selected text as the expression to the watch window.
Remove From Watch		Removes the variable or expression at the cursor position to the last activated watch window. If no text is selected, CrossStudio removes any expression matching the word at the cursor position from the watch window. If some text is selected, CrossStudio removes any expression matching the selected text from the watch window.
Edit Search Paths		Opens the <b>Debug Search File</b> dialog. See <b>Debug file search editor</b> (page 169).
Exceptions		Opens the <b>Exceptions</b> dialog.

## Debug Control menu

The **Debug Control** menu provides commands to control how you debug your program. The **Debug Control** menu is a submenu of the **Debug** menu.

### The Debug Control menu

	Source Mode	Ctrl+T, S
	Interleaved Mode	Ctrl+T, I
	Assembly Mode	Ctrl+T, A
	Toggle Debug Mode	Ctrl+F11
	Enable Interrupt Processing	Ctrl+T, N
	Disable Interrupt Processing	Ctrl+T, X
	Start Cycle Counter	
	Pause Cycle Counter	
	Reset Cycle Counter	

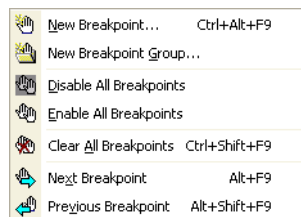
## Debug Control commands







Menu command	Keystroke	Description
Source Mode	Ctrl+T, S	Switches the debugger into <b>Source Debugging</b> mode where code is stepped a statement at a time.
Interleaved Mode	Ctrl+T, I	Switches the debugger into <b>Interleaved Debugging</b> mode where code is stepped one instruction at a time and source code is intermixed with the generated assembly code.
Assembly Mode	Ctrl+T, A	Switches the debugger into <b>Assembly Debugging</b> mode where code is stepped one instruction at a time with a simple disassembly of memory.
Toggle Debug Mode	Ctrl+F11	Toggles between <b>Source Debugging</b> and <b>Interleaved Debugging</b> modes.
Enable Interrupt Processing	Ctrl+T, N	Enables global interrupts in the processor by writing to the appropriate register.
Disable Interrupt Processing	Ctrl+T, X	Disables global interrupts in the processor by writing to the appropriate register.
Start Cycle Counter		Restarts the cycle counter after it has been paused.
Pause Cycle Counter		Pauses the cycle counter so that it does not increment and count cycles even though code executes.
Reset Cycle Counter		Resets the cycle counter to zero.

## Breakpoint menu

The **Breakpoint** menu provides commands to create, modify, and remove breakpoints. The **Breakpoint** menu is a submenu of the **Debug** menu.

### The Breakpoint menu



	New Breakpoint...	Ctrl+Alt+F9
	New Breakpoint Group...	
	Disable All Breakpoints	
	Enable All Breakpoints	
	Clear All Breakpoints	Ctrl+Shift+F9
	Next Breakpoint	Alt+F9
	Previous Breakpoint	Alt+Shift+F9

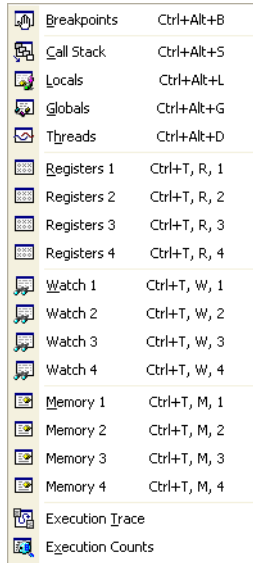
## Breakpoint commands




















Menu command	Keystroke	Description
New Breakpoint...	Ctrl+Alt+F9	Activates the <b>New Breakpoint</b> menu which allows you to create complex breakpoints on code or data. See <b>Breakpoints window</b> (page 102).
New Breakpoint Group...		Creates a new breakpoint group in the <b>Breakpoints</b> window. You can manage breakpoints individually or as a group.
Disable All Breakpoints		Disables all breakpoints so that they are never hit.
Enable All Breakpoints		Enables all breakpoints so that they can be hit.
Clear All Breakpoints	Ctrl+Shift+F9	Removes all breakpoints set in the <b>Breakpoints</b> window.
Next Breakpoint	Alt+F9	Selects the next breakpoint in the <b>Breakpoint</b> window and moves the cursor to the statement or instruction associated with that breakpoint.
Previous Breakpoint	Alt+Shift+F9	Selects the previous breakpoint in the <b>Breakpoint</b> window and moves the cursor to the statement or instruction associated with that breakpoint.

## Debug Windows menu

The **Debug Windows** menu provides commands to activate debugging windows. The **Debug Windows** menu is a submenu of the **Debug** menu.

## The Debug Windows menu



 Breakpoints	Ctrl+Alt+B
 Call Stack	Ctrl+Alt+S
 Locals	Ctrl+Alt+L
 Globals	Ctrl+Alt+G
 Threads	Ctrl+Alt+D
 Registers 1	Ctrl+T, R, 1
 Registers 2	Ctrl+T, R, 2
 Registers 3	Ctrl+T, R, 3
 Registers 4	Ctrl+T, R, 4
 Watch 1	Ctrl+T, W, 1
 Watch 2	Ctrl+T, W, 2
 Watch 3	Ctrl+T, W, 3
 Watch 4	Ctrl+T, W, 4
 Memory 1	Ctrl+T, M, 1
 Memory 2	Ctrl+T, M, 2
 Memory 3	Ctrl+T, M, 3
 Memory 4	Ctrl+T, M, 4
 Execution Trace	
 Execution Counts	

## Debug Windows commands

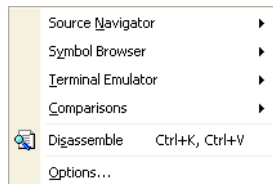
Menu command	Keystroke	Description
Breakpoints	Ctrl+Alt+B	Activates the <b>Breakpoints</b> window. See <b>Breakpoints window</b> (page 102).
Call Stack	Ctrl+Alt+S	Activates the <b>Call Stack</b> window. See <b>Call stack window</b> (page 107).
Locals	Ctrl+Alt+L	Activates the <b>Locals</b> window. See <b>Locals window</b> (page 114).
Globals	Ctrl+Alt+G	Activates the <b>Globals</b> window. See <b>Globals window</b> (page 112).
Threads	Ctrl+Alt+D	Activates the <b>Threads</b> window. See <b>Threads window</b> (page 120).
Registers 1	Ctrl+T, R, 1	Activates the first <b>Register</b> window. See <b>Register windows</b> (page 118).
Registers 2	Ctrl+T, R, 2	Activates the second <b>Register</b> window. See <b>Register windows</b> (page 118).
Registers 3	Ctrl+T, R, 3	Activates the third <b>Register</b> window. See <b>Register windows</b> (page 118).
Registers 4	Ctrl+T, R, 4	Activates the fourth <b>Register</b> window. See <b>Register windows</b> (page 118).
Watch 1	Ctrl+T, W, 1	Activates the first <b>Watch</b> window. See <b>Watch window</b> (page 122).
Watch 2	Ctrl+T, W, 2	Activates the second <b>Watch</b> window. See <b>Watch window</b> (page 122).

Menu command	Keystroke	Description
Watch 3	Ctrl+T, W, 3	Activates the third <b>Watch</b> window. See <b>Watch window</b> (page 122).
Watch 4	Ctrl+T, W, 4	Activates the fourth <b>Watch</b> window. See <b>Watch window</b> (page 122).
Memory 1	Ctrl+T, M, 1	Activates the first <b>Memory</b> window. See <b>Memory window</b> (page 116).
Memory 2	Ctrl+T, M, 2	Activates the second <b>Memory</b> window. See <b>Memory window</b> (page 116).
Memory 3	Ctrl+T, M, 3	Activates the third <b>Memory</b> window. See <b>Memory window</b> (page 116).
Memory 4	Ctrl+T, M, 4	Activates the fourth <b>Memory</b> window. See <b>Memory window</b> (page 116).
Execution Trace		Activates the <b>Execution Trace</b> window. See <b>Trace window</b> (page 122).
Execution Counts		Activates the <b>Execution Counts</b> window. See <b>Execution counts window</b> (page 111).

## Tools menu

The **Tools** menu provides setup and configuration of CrossStudio.

### The Tools menu



### Tools menu commands

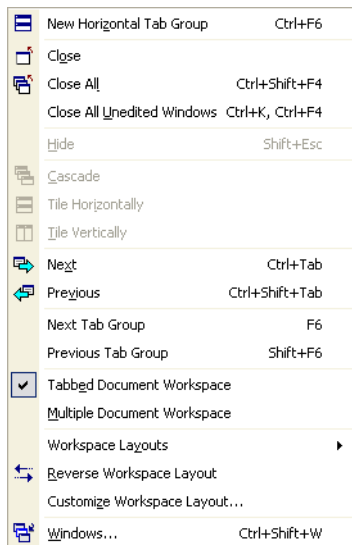
Menu command	Keystroke	Description
Source Navigator		Displays the <b>Source Navigator</b> configuration menu.
Symbol Browser		Displays the <b>Symbol Browser</b> configuration menu.
Terminal Emulator		Displays the <b>Terminal Emulator</b> configuration menu.
Comparisons		Displays the <b>Comparisons</b> menu.

Menu command	Keystroke	Description
Disassemble	Ctrl+K, Ctrl+V	Disassembles the selected project item.
Options...		Displays the <b>Environment Options</b> dialog.

## Window menu

The **Window** menu provides commands to control windows within CrossStudio.

### The Window menu



### Window menu commands

Menu command	Keystroke	Description
New Horizontal Tab Group	Ctrl+F6	Splits the tab group in two at the active tab and creates two tab groups in tabbed document workspace mode.
Close	Ctrl+F4	Closes the active document window.
Close All	Ctrl+Shift+F4	Closes all document windows.
Close All Unedited Windows	Ctrl+K, Ctrl+F4	Closes all document windows that have not been changed.
Hide <i>window</i>		Hides the focused dock window.

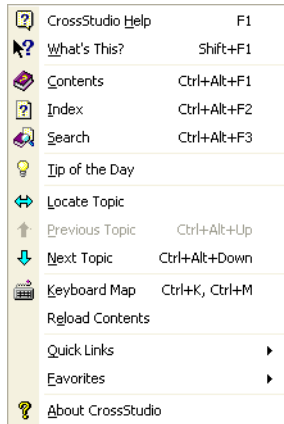
Menu command	Keystroke	Description
Cascade		Cascades windows in multiple document interface mode.
Tile Horizontally		Tiles windows horizontally in multiple document interface mode.
Next	Ctrl+Tab	Activates the next window in the tab group or window stack.
Previous	Ctrl+Shift+Tab	Activates the previous window in the tab group or window stack.
Next Tab Group	F6	Activates the next tab group in tabbed document interface mode.
Previous Tab Group	Shift+F6	Activates the next tab group in tabbed document interface mode.
Tabbed Document Workspace		Enables the tabbed document workspace.
Multiple Document Workspace		Enables the multiple document workspace.
Workspace Layouts		Displays the <b>Workspace Layout</b> menu which allows selection of various workspace layouts.
Reverse Workspace Layout		Reverses the left and right dock areas.
Customize Workspace Layout...		Displays the <b>Document Workspace Layout</b> dialog.
Windows...		Displays the <b>Windows</b> dialog.

## Help menu

The **Help** menu provides access to online help for CrossStudio.



## The Help menu



## Help menu commands

Menu command	Keystroke	Description
CrossStudio Help	F1	Displays online help for the focused GUI element.
Whats This	Shift+F1	Enters Whats This? mode which provides a short description of each GUI element.
Contents	Ctrl+Alt+F1	Activates the <b>Contents</b> window.
Index	Ctrl+Alt+F2	Activates the <b>Index</b> window.
Search	Ctrl+Alt+F3	Activates the <b>Search</b> window.
Tip of the Day		Activates the <b>Tip of the Day</b> window.
Locate Topic		Locates the help page displayed by the browser in the <b>Contents</b> window.
Previous Topic	Ctrl+Alt+Up	Moves to the previous topic in the Contents window and updates the browser.
Next Topic	Ctrl+Alt+Down	Moves to the next topic in the Contents window and updates the browser.
Keyboard Map	Ctrl+K, Ctrl+M	Displays the <b>Keyboard Map</b> dialog.
Quick Links		Displays the <b>Quick Links</b> menu which contains useful shortcuts to the online manual.
Favorites		Displays the web pages from the <b>Favorites</b> window.
About CrossStudio		Displays information on CrossStudio, the license agreement, and activation status.



# Part III

## Tasking Library Tutorial

# Tasking Library Tutorial

This section describes the CrossWorks Tasking Library which will be subsequently referred to as the CTL. The CTL provides a multi-priority, preemptive, task switching and synchronisation facility. Additionally the library provides timer, interrupt service routine and memory block allocation support.

### In this section

- **Overview (page 205).** Describes the principles behind the CTL.
- **Tasks (page 209).** Describes how to create CTL tasks, turn the main program into a task and manage tasks.
- **Event sets (page 213).** Describes what a CTL event set is and how it can be used.
- **Semaphores (page 217).** Describes what a CTL semaphore is and how it can be used.
- **Message queues (page 221).** Describes what a CTL message queue is and how it can be used.
- **Byte queues (page 225).** Describes what a CTL byte queue is and how it can be used.
- **Global interrupts control (page 229).** Describes how you can use CTL functions to enable and disable global interrupts.
- **Timer support (page 231).** Describes the timer facilities that the CTL provides.

- **Programmable interrupt handling (page 233).** Describes how you can use CTL functions on systems that have programmable interrupt controller hardware.
- **Low-level interrupt handling (page 235).** Describes how to write interrupt service routines that co-exist with the CTL.
- **Memory areas (page 237).** Describes how you can use the CTL to allocate fixed sized memory blocks.

#### **Related sections**

- **<ctl\_api.h> - CTL functions.** The reference for each of the functions and variables defined by the CTL.
- **Threads window (page 120).** A scriptable debugger window that displays the threads of a running program together with their state.

## Overview

The CTL enables your application to have multiple tasks. You will typically use a task when you have some algorithmic or protocol processing that suspend its execution whilst other activities occur. For example you may have a protocol processing task and a user interface task.

### Tasks

A task (sometimes called a thread) is a CPU execution context which is typically a subset of the CPU register state. When a task switch occurs the CPU execution context is saved on to the stack of the current task, a new task is selected to run and its saved CPU execution context is restored. The process of selecting a new task to run is called task switching or (re)scheduling.

A task has a priority associated with it, the lowest priority is 0 the highest is 255. A task is either executing (the current task) or it is queued in the task list. The task list is kept in priority order with the highest priority task at the head of the list. The current task will always have a priority that is greater than or equal to the first runnable task in the task list.

Task switching can be *cooperative* or *preemptive*.

Cooperative task switching occurs when the current task calls a CTL function which checks for rescheduling and the next task ready to run is of the same or higher priority than the current task.

Preemptive task switching occurs when an interrupt service routine calls a CTL function which checks for rescheduling and the next task ready to run is of a higher priority than the current task.

Preemptive task switching can also occur when an interrupt service routine calls a CTL function which checks for rescheduling, time slicing is enabled, the time slice period has been exceeded and the next task ready to run is of the same priority as the current task.

There is one executing task and there must always be a task ready to execute i.e. the task list must have a runnable task queued on it. Typically there will always be an idle task that loops and perhaps puts the CPU into a power save mode. A task on the task list is either runnable or waiting for something (e.g. timeout).

When a task switch occurs global interrupts will be enabled. So you can safely call the tasking library functions with interrupts disabled.

## Task synchronization and resource allocation

The CrossWorks tasking library provides several mechanisms to synchronize execution of tasks and to serialise resource allocation.

- **Event Sets** — An event set is a word sized variable which tasks can wait for specific bits (events) to be set to 1. You can wait for any specified events in an event set or for all of the specified events. You can also specify that the events the task are waiting on are automatically cleared (set to 0) when the task has completed its wait.
- **Counting Semaphores** — A counting semaphore is a word size variable which tasks can wait for to be non-zero. Counting semaphores are useful when serialising access to fixed sized buffers i.e. the count value can represent the number of free or used elements in the buffer.
- **Message Queues** — A message queue is a structure that enables tasks to post and receive data. Message queues are used to provide a buffered communication mechanism between tasks.
- **Byte Queues** — A byte queue is a specialisation of a message queue i.e. it is a message queue where the messages are one byte in size.
- **Interrupt enable/disable** — The tasking library provides functions that enable and disable the global interrupt enables state of the processor. These functions can be used to provide a time critical mutual exclusion facility.

Note that all waits on task synchronization objects are priority based i.e. the highest priority task waiting will be scheduled first.

## Timer support

If your application can provide a periodic timer interrupt (for example one that keeps a watch dog alive) then you can use the timer wait facility of the library. This is a simple software counter that is incremented by your timer interrupt. You can use this to specify a wakeup time and to prevent your program waiting forever for something to happen.

## Interrupt service routine support

On systems that have programmable interrupt controllers the CTL provides functions that enable you to install interrupt service routines as C functions and associate the required hardware priority to their execution. On systems that have fixed interrupt schemes functions are provided that enable you to create interrupt service routines that co-operate with the CTL.

Tasks can synchronize with interrupt service routines using either event sets, semaphores or message queues. Interrupt service routines are allowed to set (and clear) events in an event set, to signal a semaphore and to do a non blocking post to a message queue. Interrupt service routines cannot wait for events, wait for a semaphore or use blocking message queue functions.

## Memory block allocation support

The CTL provides a simple memory block allocator that can be used in situations where the standard C malloc and free functions are either too slow or may block the calling task.

## C library support

The CTL provides a task specific errno as well as exclusion mechanisms to enable usage of malloc/free functions in a multi-tasking environment.





# 18

## Tasks

Each task has a corresponding task structure that holds information such as the priority and the saved register state. You allocate task structures by declaring them as C variables.

```
CTL_TASK_t mainTask;
```

You create the first task using the `ctl_task_init` function which turns the main program into a task. This function takes a pointer to the task structure that represents the main task, its priority and a name as parameters.

```
ctl_task_init(&mainTask, 255, "main");
```

This function must be called before any other CrossWorks tasking library calls are made. The priority (second parameter) must be between 0 (the lowest priority) and 255 (the highest priority). It is advisable to create the first task with the highest priority which enables the main task to create other tasks without being descheduled. The name should point to a zero terminated ASCII string for debug purposes. When this function has been called global interrupts will be enabled, so you must ensure that any interrupt sources are disabled before calling this function.

You can create other tasks using the function `ctl_task_run` which initialises a task structure and may cause a context switch. You supply the same arguments as `task_init` together with the function that the task will run and the memory that the task will use as its stack.

The function that a task will run should take a `void *` parameter and not return any value.

```
void task1Fn(void *parameter)
{
```

```
    // task code goes in here
    ...
}
```

The `parameter` value is supplied to the function by the `ctl_task_run` call. Note when a task function returns the `ctl_task_die` function is called which terminates the task.

You have to allocate the stack for the task as an C array of unsigned.

```
unsigned task1Stack[64];
```

The size of the stack you need depends on the CPU type (the number of registers that have to be saved), the function calls that the task will make and (depending upon the CPU) the stack used for interrupt service routines. Running out of stack space is common problem with multi-tasking systems and the error behaviour is often misleading. It is recommended that you initialise the stack to known values so that you can check the stack with the CrossWorks debugger if problems occur.

```
memset(task1Stack, 0xba, sizeof(task1Stack));
```

Your `ctl_task_run` function call should look something like this.

```
ctl_task_run(&task1Task,
            12,
            task1Fn,
            0,
            "task1",
            sizeof(task1Stack) / sizeof(unsigned),
            task1Stack,
            0);
```

The first parameter is a pointer to the task structure. The second parameter is the priority (in this case 12) the task will start executing at. The third parameter is a pointer to the function to execute (in this case `task1Fn`). The fourth parameter is the value that is supplied to the task function (in this case zero). The fifth parameter is a null terminated string that names the task for debug purposes. The sixth parameter is the size of the stack in words. The seventh parameter is the pointer to the stack. The last parameter is for systems that have a separate call stack and is the number of words to reserve for the call stack.

You can change the priority of a task using the `ctl_task_set_priority` function call which takes a pointer to a task structure and the new priority as parameters.

```
ctl_task_set_priority(&mainTask, 0);
```

### Example

The following example turns main into a task and creates another task. The main task ultimately will be the lowest priority task that switches the CPU into a power save mode when it is scheduled - this satisfies the requirement of always having a task to execute and enables a simple power saving system to be implemented.

```

#include <ctl_api.h>

void task1(void *p)
{
    // task code, on return task will be terminated
}

static CTL_TASK_t mainTask, task1Task;
static unsigned task1Stack[64];

int
main(void)
{
    // Turn myself into a task running at the highest priority.
    ctl_task_init(&mainTask, 255, "main");

    // Initialise the stack of task1.
    memset(task1Stack, 0xba, sizeof(task1Stack)/sizeof(unsigned));

    // Make another task ready to run.
    ctl_task_run(&task1Task, 1, task1, 0, "task1", sizeof(task1Stack) /
sizeof(unsigned), task1Stack, 0);

    // Now all the tasks have been created go to lowest priority.
    ctl_task_set_priority(&mainTask, 0);

    // Main task, if activated because task1 is suspended, just
    // enters low power mode and waits for task1 to run again
    // (for example, because an interrupt wakes it).
    for (;;)
    {
        // Go into low power mode
        sleep();
    }
}

```

Note that initially the main task is created at the highest priority whilst it creates the other tasks, it then changes its priority to the lowest task. This technique can be used when multiple tasks are created to enable all of the tasks to be created before they start to execute.

Note the usage of **sizeof** when passing the stack size to **ctl\_task\_run**.



# 19

## Event sets

An event set is a means to synchronise tasks with other tasks and interrupt service routines. An event set contains a set of events (one per bit) which tasks can wait to become set (value 1). When a task waits on an event set the events it is waiting for are matched against the current values—if they match then the task can still execute. If they don't match, the task is put on the task list together with details of the event set and the events that the task is waiting for.

You allocate an event set by declaring it as C variable

```
CTL_EVENT_SET_t e1;
```

An `CTL_EVENT_SET_t` is a synonym for an **unsigned** type. Thus, when an **unsigned** is naturally 16 bits an event set will contain 16 events and when it is naturally 32 bits an event set will contain 32 events.

You can initialise an event set using the `ctl_events_init` function.

```
ctl_events_init(&e1, 0);
```

Note that initialisation should be done before any tasks can use an event set.

You can set and clear events of an event set using the `ctl_events_set_clear` function.

```
ctl_events_set_clear(&e1, 1, 0x80);
```

This example will set the bit zero event and clear the bit 15 event. If any tasks are waiting on this event set the events they are waiting on will be matched against the new event set value which could cause the task to become runnable.

You can wait for events to be set using the `ctl_events_wait` function. You can wait for any of the events in an event set to be set (`CTL_EVENT_WAIT_ANY_EVENTS`) or all of the events to be set (`CTL_EVENT_WAIT_ALL_EVENTS`). You can also specify that when events have been set and have been matched that they should be automatically reset (`CTL_EVENT_WAIT_ANY_EVENTS_WITH_AUTO_CLEAR` and `CTL_EVENT_WAIT_ALL_EVENTS_WITH_AUTO_CLEAR`). You can associate a timeout with a wait for an event set to stop your application blocking indefinitely.

```
ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS, &e1, 0x80, 0, 0);
```

This example waits for bit 15 of the event set pointed to by `e1` to become set.

```
if (ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS, &e1, 0x80, 1,
    ctl_get_current_time()+1000)==0)
    {
        // timeout occurred
    }
```

This example uses a timeout and tests the return result to see if the timeout occurred.

### Task synchronisation in an interrupt service routine

The following example illustrates synchronising a task with a function called from an interrupt service routine.

```
CTL_EVENT_SET_t e1;
CTL_TASK_s t1;

void ISRfn()
{
    // do work, and then...
    ctl_events_set_clear(&e1, 1, 0);
}

void task1(void *p)
{
    while (1)
    {
        ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS, &e1, 1, 0, 0);
        ...
        ctl_events_set_clear(&e1, 0, 1);
    }
}
```

### Task synchronisation with more than one interrupt service routine

The following example illustrates synchronising a task with functions called from two interrupt service routines.

```
CTL_EVENT_SET_t e1;
CTL_TASK_s t1;

void ISRfn1(void)
{
    // do work, and then...
```

```

    ctl_events_set_clear(&e1, 1, 0);
}
void ISRfn2(void)
{
    // do work, and then...
    ctl_events_set_clear(&e1, 2, 0);
}
void task1(void *p)
{
    for (;;)
    {
        unsigned e;
        e = ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS_AUTO_CLEAR,
                           &e1,
                           1 | 2,
                           0, 0);
        if (e & 1)
        {
            // ISRfn1 completed
        }
        else if (e & 2)
        {
            // ISRfn2 completed
        }
        else
        {
            // error
        }
    }
}

```

### Resource serialisation

The following example illustrates resource serialisation of two tasks.

```

CTL_EVENT_SET_t e1;

void task1(void)
{
    for (;;)
    {
        ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS_AUTO_CLEAR, &e1, 1, 0, 0);
        // resource has now been acquired
        ctl_events_set_clear(&e1, 1, 0);
        // resource has now been released
    }
}

void task2(void)
{
    for (;;)
    {
        ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS_AUTO_CLEAR, &e1, 1, 0, 0);
        // resource has now been acquired
        ctl_events_set_clear(&e1, 1, 0);
        // resource has now been released
    }
}
...
void main(void)
{

```

```
    ...  
    ctl_events_init(&e1, 1);  
    ...  
}
```

Note that **e1** is initialised with the event set—without this neither task would acquire the resource.



## Semaphores

A semaphore is a counter which tasks can wait for to be non-zero. When a semaphore is non-zero and a task waits on it then the semaphore value is decremented and the task continues execution. When a semaphore is zero and a task waits on it then the task will be suspended until the semaphore is signalled. When a semaphore is signalled and no tasks are waiting for it then the semaphore value is incremented. When a semaphore is signalled and tasks are waiting then one of the tasks is made runnable.

You allocate a semaphore by declaring it as a C variable

```
CTL_SEMAPHORE_t s1;
```

An **CTL\_SEMAPHORE\_t** is a synonym for an **unsigned** type, so the maximum value of the counter is dependent upon the word size of the processor (either 16 or 32 bits).

You can initialise a semaphore using the `ctl_semaphore_init` function.

```
ctl_semaphore_init(&s1, 1);
```

Note that initialisation should be done before any tasks can use a semaphore.

You can signal a semaphore using the `ctl_semaphore_signal` function.

```
ctl_semaphore_signal(&s1);
```

The highest priority task waiting on the semaphore pointed at by **s1** will be made runnable by this call. If no tasks are waiting on the semaphore then the semaphore value is incremented.

You can wait for a semaphore with an optional timeout using the `ctl_semaphore_wait` function.

```
ctl_semaphore_wait(&s1, 0, 0);
```

This example will block the task if the semaphore is zero, otherwise it will decrement the semaphore and continue execution.

```
if (ctl_semaphore_wait(&s1, 1, ctl_get_current_time()+1000)==0)
{
    // timeout occurred
}
```

This example uses a timeout and tests the return result to see if the timeout occurred.

### Task synchronisation in an interrupt service routine

The following example illustrates synchronising a task with a function called from an interrupt service routine.

```
CTL_SEMAPHORE_t s1;

void ISRfn()
{
    // do work
    ctl_semaphore_signal(&s1);
}

void task1(void *p)
{
    while (1)
    {
        ctl_semaphore_wait(&s1, 0, 0);
        ...
    }
}
```

### Resource serialisation

The following example illustrates resource serialisation of two tasks.

```
CTL_SEMAPHORE_t s1=1;

void task1(void)
{
    for (;;)
    {
        ctl_semaphore_wait(&s1, 0, 0);
        /* resource has now been acquired */
        ...
        ctl_semaphore_signal(&s1);
        /* resource has now been released */
    }
}

void task2(void)
{
    for (;;)
    {
        ctl_semaphore_wait(&s1);
        /* resource has now been acquired */
        ...
    }
}
```

```
        ctl_semaphore_signal(&s1);
        /* resource has now been released */
    }
}

int
main(void)
{
    ...
    ctl_semaphore_init(&s1, 1);
    ...
}
```

Note that **s1** is initialised to one, without this neither task would acquire the resource.



## Message queues

A message queue is a structure that enables tasks to post and receive messages. A message is a generic (void) pointer and as such can be used to send data that will fit into a pointer type (2 or 4 bytes depending upon processor word size) or can be used to pass a pointer to a block of memory. The message queue has a buffer that enables a number of posts to be completed without receives occurring. The buffer keeps the posted messages in a fifo order so the oldest message is received first. When the buffer isnt full a post will put the message at the back of the queue and the calling task continues execution. When the buffer is full a post will block the calling task until there is room for the message. When the buffer isnt empty a receive will return the message from the front of the queue and continue execution of the calling task. When the buffer is empty a receive will block the calling task until a message is posted.

You allocate a message queue by declaring it as a C variable

```
CTL_MESSAGE_QUEUE_t m1;
```

A message queue is initialised using the `ctl_message_queue_init` function.

```
void *queue[20];  
...  
ctl_message_queue_init(&m1, queue, 20);
```

This example uses an 20 element array for the message queue. Note that the array is a `void *` which enables pointers to memory or (cast) integers to be communicated via a message queue.

You can post a message to a message queue with an optional timeout using the `ctl_message_queue_post` function.

```
ctl_message_queue_post(&m1, (void *)45, 0, 0);
```

This example posts the integer 45 onto the message queue.

```
if (ctl_message_queue_post(&m1, (void *)45, 1, ctl_get_current_time()+1000)
== 0)
{
    // timeout occurred
}
```

This example uses a timeout and tests the return result to see if the timeout occurred.

If you want to post a message and you don't want to block (e.g. from an interrupt service routine) you can use the `ctl_message_queue_post_nb` function.

```
if (ctl_message_queue_post_nb(&m1, (void *)45)==0)
{
    // queue is full
}
```

This example tests the return result to see if the post failed.

You can receive a message with an optional timeout using the `ctl_message_queue_receive` function.

```
void *msg;
ctl_message_queue_receive(&m1, &msg, 0, 0);
```

This example receives the oldest message in the message queue.

```
if (ctl_message_queue_receive(&m1, &msg, 1, ctl_get_current_time()+1000) ==
0)
{
    // timeout occurred
}
```

This example uses a timeout and tests the return result to see if the timeout occurred.

If you want to receive a message and you don't want to block (e.g. from an interrupt service routine) you can use the `ctl_message_queue_receive_nb` function.

```
if (ctl_message_queue_receive_nb(&m1, &msg)==0)
{
    // queue is empty
}
```

#### Example

The following example illustrates usage of a message queue to implement the producer-consumer problem.

```
CTL_MESSAGE_QUEUE_t m1;
void *queue[20];

void task1(void)
{
    ...
    ctl_message_queue_post(&m1, (void *)i, 0, 0);
    ...
}
```

```
void task2(void)
{
    void *msg;
    ...
    ctl_message_queue_receive(&m1, &msg, 0, 0);
    ...
}

int
main(void)
{
    ...
    ctl_message_queue_init(&m1, queue, 20);
    ...
}
```





## Byte queues

A byte queue is a structure that enables tasks to post and receive data bytes. The byte queue has a buffer that enables a number of posts to be completed without receives occurring. The buffer keeps the posted bytes in a fifo order so the oldest byte is received first. When the buffer isn't full a post will put the byte at the back of the queue and the calling task continues execution. When the buffer is full a post will block the calling task until there is room for the byte. When the buffer isn't empty a receive will return the byte from the front of the queue and continue execution of the calling task. When the buffer is empty a receive will block the calling task until a byte is posted.

You allocate a byte queue by declaring it as a C variable

```
CTL_BYTE_QUEUE_t m1;
```

A byte queue is initialised using the `ctl_byte_queue_init` function.

```
unsigned char queue[20];  
...  
ctl_byte_queue_init(&m1, queue, 20);
```

This example uses an 20 element array for the byte queue.

You can post a byte to a byte queue with an optional timeout using the `ctl_byte_queue_post` function.

```
ctl_byte_queue_post(&m1, 45, 0, 0);
```

This example posts the byte 45 onto the byte queue.

```
if (ctl_byte_queue_post(&m1, 45, 1, ctl_get_current_time()+1000) == 0)  
{  
    // timeout occurred  
}
```

This example uses a timeout and tests the return result to see if the timeout occurred. If you want to post a byte and you don't want to block (e.g. from an interrupt service routine) you can use the `ctl_byte_queue_post_nb` function

```
if (ctl_byte_queue_post_nb(&m1, 45)==0)
{
    // queue is full
}
```

This example tests the return result to see if the post failed.

You can receive a byte with an optional timeout using the `ctl_byte_queue_receive` function.

```
void *msg;
ctl_byte_queue_receive(&m1, &msg, 0, 0);
```

This example receives the oldest byte in the byte queue.

```
if (ctl_byte_queue_receive(&m1, &msg, 1, ctl_get_current_time()+1000) == 0)
{
    // timeout occurred
}
```

This example uses a timeout and tests the return result to see if the timeout occurred.

If you want to receive a byte and you don't want to block (e.g. from an interrupt service routine) you can use the `ctl_byte_queue_receive_nb` function.

```
if (ctl_byte_queue_receive_nb(&m1, &msg)==0)
{
    // queue is empty
}
```

#### Example

The following example illustrates usage of a byte queue to implement the producer-consumer problem.

```
CTL_BYTE_QUEUE_t m1;
void *queue[20];

void task1(void)
{
    ...
    ctl_byte_queue_post(&m1, (void *)i, 0, 0);
    ...
}

void task2(void)
{
    void *msg;
    ...
    ctl_byte_queue_receive(&m1, &msg, 0, 0);
    ...
}

int
main(void)
```

```
{  
  ...  
  ctl_byte_queue_init(&m1, queue, 20);  
  ...  
}
```



## Global interrupts control

The CrossWorks tasking library provides functions that lock and unlock the global interrupt enables. These functions can be used (sparingly) to provide a fast mutual exclusion facility for time critical uses.

You can disable interrupts using the `ctl_global_interrupts_disable` function call.

```
int en=ctl_global_interrupts_disable();
```

This function returns the previous global interrupts enabled state.

You can enable interrupts using the `ctl_global_interrupts_enable` function call.

```
int en=ctl_global_interrupts_enable();
```

This function returns the previous global interrupts enabled state.

You can restore the previous global interrupts enabled state you the `ctl_global_interrupts_set` function call.

```
int en = ctl_global_interrupts_disable();  
...  
ctl_set_interrupts(en);
```

Note that you can call a tasking library function that causes a task switch with global interrupts disabled. The tasking library will ensure that when a task is scheduled that global interrupts are enabled.

You can re-enable global interrupt enables from within an interrupt service routine using the `ctl_global_interrupts_re_enable_from_isr` function call in order to permit higher priority interrupts to occur. A call to this function must be matched with a call to the `ctl_global_interrupts_un_re_enable_from_isr` function.

```
// code of interrupt service routine
...
ctl_global_interrupts_re_enable_from_isr();
...
// global interrupts are now enabled so another interrupt can be handled.
...
ctl_global_interrupts_un_re_enable_from_isr();
...
```

## Timer support

The current time is held as a 32 bit value in the `ctl_current_time` variable. This variable is incremented by a periodic interrupt that is started using the `ctl_start_timer` function. When you start the timer you must pass it a function to call when the periodic interrupt occurs. The interrupt function can be a user defined function that calls `ctl_increment_tick_from_isr`.

```
void myfn(void)
{
    ..
    ctl_increment_tick_from_isr();
    ..
}
void main(...)
..
    ctl_start_timer(myfn);
..
```

Alternatively you can pass the `ctl_increment_tick_from_isr` function as the parameter

```
void main(...)
{
    ..
    ctl_start_timer(ctl_increment_tick_from_isr);
    ..
}
```

You can atomically read `ctl_current_time` using the `ctl_get_current_time` function on systems whose word size is not 32 bit.

You can find out the resolution of the timer using the `ctl_get_ticks_per_second` function.

You can suspend execution of a task for a fixed period using the `ctl_timeout_wait` function.

Note that this function takes the timeout not the duration as a parameter, so you should always call this function with **ctl\_get\_current\_time()+duration**.

```
ctl_timeout_wait(ctl_get_current_time()+100);
```

This example suspends execution of the calling task for 100 increments of the **ctl\_current\_time** variable.



## Programmable interrupt handling

The CTL provides an optional set of functions for establishing C functions as interrupt service routines. These functions are available on systems that have programmable interrupt controller hardware. On systems that have fixed interrupt schemes you should use the facilities described in **Low-level interrupt handling** (page 235) when you create your interrupt service routines.

The function `ctl_set_isr` is used to establish a C function as an interrupt service routine.

You must enable an interrupt source using `ctl_unmask_isr` and you can disable an interrupt source using `ctl_mask_isr`.

The C function you have established is called when the interrupt occurs. On entry to this function interrupts will still be disabled. To allow interrupts of a higher priority to occur you should enable interrupts on entry by calling `ctl_global_interrupts_re_enable_from_isr` and disable interrupts on exit by calling `ctl_global_interrupts_un_re_enable_from_isr`. Note that the pending interrupt flag in the interrupt controller hardware will be cleared by the CTL when your interrupt service routine returns.

### Interrupt service routine example

```
void isr(void)
{
    ctl_global_interrupts_re_enable_from_isr();
    ...
    // do interrupt handling stuff in here
    // including clearing the source of the interrupt
    ...
    ctl_global_interrupts_un_re_enable_from_isr();
}
```

```
int main(void)
{
    ...
    ctl_set_isr(11, 11, CTL_ISR_TRIGGER_FIXED, isr, 0);
    ctl_unmask_isr(11);
    ...
}
```

The `isr` function is triggered from interrupt vector 11 and will run at priority 11. When the function is run it enables interrupts which will allow higher priority interrupts to trigger whilst it is executing.

## Low-level interrupt handling

If your system doesn't support a programmable interrupt controller and you want tasks to be rescheduled when interrupts occur, you must save the register state of the CPU on entry to an interrupt service routine and increment the global variable `ctl_interrupt_count`.

When you are executing an interrupt service routine you must not call the tasking library functions that may block (**`task_wait_events`**, **`task_wait_semaphore`**, **`task_post_message`**, **`task_receive_message`**, **`task_wait_timeout`**) — you can call other tasking library functions, but a task switch will only occur when the last interrupt handler has completed execution.

Whilst you are executing an interrupt service routine you can allow interrupts of a higher priority to occur by calling `ctl_global_interrupts_re_enable_from_isr`. You must also disable interrupts before exit from the interrupt service routine by calling `ctl_global_interrupts_un_re_enable_from_isr`.

In order to achieve a task switch from an interrupt service routine the `ctl_exit_isr` function must be jumped to as the last action of an interrupt service routine. This function must be passed a pointer to the saved registers.

### Interrupt service routine (ARM example)

This example declares an ISR using the GCC syntax for declaring naked functions and accessing assembly code instructions.

```
void irq_handler(void) __attribute__((naked));

void
irq_handler(void)
{
    asm("stmfd sp!, {r0-r12, lr}");
    asm("mrs r0, spsr");
}
```

```
asm("stmfd sp!, {r0}");
ctl_interrupt_count++;
...
// do interrupt handling stuff in here
...
asm("mov r0, sp");
asm("b ctl_exit_isr");
}
```

Note that the registers **SPSR**, **R0** through **R12** and **R14** (user mode program counter) must be saved on the stack. The user mode **R13** and **R14** registers don't need to be saved because they are held in banked registers.

Note that FIQ handlers are not supported on the ARM.

## Memory areas

Memory areas provide your application with dynamic allocation of fixed sized memory blocks. Memory areas should be used in preference to the standard C library malloc and free functions if the calling task (or interrupt service routine) cannot block.

You allocate a memory area by declaring it as a C variable

```
CTL_MEMORY_AREA_t m1;
```

A message queue is initialised using the `ctl_memory_area_init` function.

```
unsigned mem[20];  
...  
ctl_message_queue_init(&m1, mem, 2, 10);
```

This example uses an 20 element array for the memory. The array is split into 10 blocks of each of which two words in size.

You can allocate a memory block from a memory area using the `ctl_memory_area_allocate` function. If the memory block cannot be allocated then zero is returned.

```
unsigned *block = ctl_memory_area_allocate(&m1);  
if (block)  
    // block has been allocated  
else  
    // no block has been allocated
```

When you have finished with a memory block you should return it to the memory area from which it was allocated using `ctl_memory_area_free`:

```
ctl_memory_area_free(&m1, block);
```



# Part IV

---

## Library Reference

# Library Reference





## Standard C Library Reference

CrossWorks C provides a library that conforms to the ANSI and ISO standards for C.

### In this section

- **<assert.h> - Diagnostics (page 242).** Describes the diagnostic facilities which you can build into your application.
- **<ctype.h> - Character handling (page 243).** Describes the character classification and manipulation functions.
- **<errno.h> - Errors (page 247).** Describes the macros and error values returned by the C library.
- **<limits.h> - Integer numerical limits (page 248).** Describes the macros that define the extreme values of underlying C types.
- **<math.h> - Mathematics (page 252).** Describes the mathematical functions provided by the C library.
- **<setjmp.h> - Non-local jumps (page 276).** Describes the non-local goto capabilities of the C library.
- **<stdarg.h> - Variable arguments (page 278).** Describes the way in which variable parameter lists are accessed.
- **<stdio.h> - Input/output functions (page 280).** Describes the formatted input and output functions.
- **<stdio.h> - Input/output functions (page 280).** Describes the general utility functions provided by the C library.
- **<string.h> - String handling (page 310).** Describes the string handling functions provided by the C library.

## <assert.h> - Diagnostics

The header file <assert.h> defines the **assert** macro under control of the **NDEBUG** macro, which the library *does not define*.

### Macros

---

**assert** (page 242)    Assert that a condition is true

---

### assert

**Synopsis**

```
#include <assert.h>
void assert(expression);
```

**Description**    **assert** allows you to place assertions and diagnostic tests into programs. If **NDEBUG** is defined as a macro name at the point in the source file where <assert.h> is included, the **assert** macro is defined as:

```
#define assert(ignore) ((void)0)
```

If **NDEBUG** is not defined as a macro name at the point in the source file where <assert.h> is included, the **assert** macro expands to a **void** expression that calls **\_\_assert**. When such an **assert** is executed and **expression** is false, **assert** calls the **\_\_assert** function with information about the particular call that failed: the text of the argument, the name of the source file, and the source line number. These are the stringized expression and the values of the preprocessing macros **\_\_FILE\_\_** and **\_\_LINE\_\_**.

The prototype for **\_\_assert** is:

```
extern void __assert(const char *, const char *, int);
```

There is no default implementation of **\_\_assert**. Keeping **\_\_assert** out of the library means that you can customize its behaviour without rebuilding the library.

**Important notes**    The **assert** macro is redefined according to the current state of **NDEBUG** each time that <assert.h> is included.

**Portability**    **assert** conforms to ISO/IEC 9899:1990 (C90).

## <ctype.h> - Character handling

The header <ctype.h> declares several functions useful for classifying and mapping characters.

The character argument to all functions is an **int**, the value of which is representable as an unsigned char or is the value of the macro **EOF**. If the argument has any other value, the behavior is undefined.

Only the "C" locale is supported by CrossWorks C, and thus the functions in this header are not affected by locales.

The term printing character refers to a member of a set of characters, each of which occupies one printing position on a display device; the term control character refers to a member of a set of characters that are not printing characters. All letters and digits are printing characters.

### Classification functions

<b>isalnum</b>	Is character alphanumeric?
<b>isalpha</b>	Is character alphabetic?
<b>isblank</b>	Is character a space or horizontal tab?
<b>isctrl</b>	Is character a control character?
<b>isdigit</b>	Is character a decimal digit?
<b>isgraph</b>	Is character any printing character except space?
<b>islower</b>	Is character a lowercase letter?
<b>isprint</b>	Is character printable?
<b>ispunct</b>	Is character a punctuation mark?
<b>isspace</b>	Is character a whitespace character?
<b>isupper</b>	Is character an uppercase letter?
<b>isxdigit</b>	Is character a hexadecimal letter?

### Conversion functions

<b>tolower</b>	Convert uppercase character to lowercase
<b>toupper</b>	Convert lowercase character to uppercase

---

## isalnum

- Synopsis** `#include <ctype.h>`  
`int isalnum(int c);`
- Description** **isalnum** returns nonzero (true) if and only if **isalpha** or **isdigit** return true for value of the argument **c**.
- Portability** **isalnum** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
- See also** **isalpha** (page 244) **isdigit** (page 245)

---

## isalpha

- Synopsis** `#include <ctype.h>`  
`int isalpha(int c);`
- Description** **isalpha** returns nonzero (true) if and only if **isupper** or **islower** return true for value of the argument **c**.
- Portability** **isalpha** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
- See also** **isupper** (page 246) **islower** (page 245)

---

## isblank

- Synopsis** `#include <ctype.h>`  
`int isblank(int c);`
- Description** **isblank** returns nonzero (true) if and only if the value of the argument **c** is either a space character ( ' ') or the horizontal tab character ( '\t ').
- Portability** **isblank** ISO/IEC 9899:1999 (C99).
- See also** **isspace** (page 246)

---

## iscntrl

- Synopsis** `#include <ctype.h>`  
`int iscntrl(int c);`
- Description** **iscntrl** returns nonzero (true) if and only if the value of the argument **c** is a control character. Control characters have values 0 through 31 and the single value 127.
- Portability** **iscntrl** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## isdigit

**Synopsis**

```
#include <ctype.h>
int isdigit(int c);
```

**Description**   **isdigit** returns nonzero (true) if and only if the value of the argument **c** is a decimal digit 0 through 9.

**Portability**   **isdigit** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## isgraph

**Synopsis**

```
#include <ctype.h>
int isgraph(int c);
```

**Description**   **isgraph** returns nonzero (true) if and only if the value of the argument **c** is any printing character except space ( ' ').

**Portability**   **isgraph** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## isupper

**Synopsis**

```
#include <ctype.h>
int islower(int c);
```

**Description**   **islower** returns nonzero (true) if and only if the value of the argument **c** is an uppercase letter a through z.

**Portability**   **islower** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## isprint

**Synopsis**

```
#include <ctype.h>
int isprint(int c);
```

**Description**   **isprint** returns nonzero (true) if and only if the value of the argument **c** is any printing character including space ( ' ').

**Portability**   **isprint** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## ispunct

**Synopsis**

```
#include <ctype.h>
int ispunct(int c);
```

- Description** **ispunct** returns nonzero (true) for every printing character for which neither **isspace** nor **isalnum** is true.
- Portability** **ispunct** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
- See also** **isspace** (page 246) **isalnum** (page 244)
- 

## **isspace**

- Synopsis**

```
#include <ctype.h>
int isspace(int c);
```
- Description** **isspace** returns nonzero (true) if and only if the value of the argument **c** is a standard white-space character. The standard white-space characters are space ( ' ' ), form feed ( '\f' ), new-line ( '\n' ), carriage return ( '\r' ), horizontal tab ( '\t' ), and vertical tab ( '\v' ).
- Portability** **isspace** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
- See also** **isblank** (page 244)
- 

## **isupper**

- Synopsis**

```
#include <ctype.h>
int isupper(int c);
```
- Description** **isupper** returns nonzero (true) if and only if the value of the argument **c** is an uppercase letter A through Z.
- Portability** **isupper** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
- 

## **isxdigit**

- Synopsis**

```
#include <ctype.h>
int isxdigit(int c);
```
- Description** **isxdigit** returns nonzero (true) if and only if the value of the argument **c** is a hexadecimal digit 0 through 9, a through f, or A through F.
- Portability** **isxdigit** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
- 

## **tolower**

- Synopsis**

```
#include <ctype.h>
int tolower(int c);
```

**Description** **tolower** converts an uppercase letter to a corresponding lowercase letter.

If the argument **c** is a character for which **isupper** is true, **tolower** returns the corresponding lowercase letter; otherwise, the argument is returned unchanged.

**Portability** **tolower** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## **toupper**

**Synopsis**

```
#include <ctype.h>
int toupper(int c);
```

**Description** **toupper** converts a lowercase letter to a corresponding uppercase letter.

If the argument **c** is a character for which **islower** is true, **toupper** returns the corresponding uppercase letter; otherwise, the argument is returned unchanged.

**Portability** **toupper** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

## <errno.h> - Errors

The header file **<errno.h>** defines macros defines several macros, all relating to the reporting of error conditions.

### Macros

---

<b>errno</b>	Error number
--------------	--------------

---

## **errno**

**Synopsis**

```
#include <errno.h>
int errno;
```

**Description** **errno** expands to a modifiable lvalue of type **int**, the value of which is set to a positive error number by several library functions.

The ISO standard does not specify whether **errno** is a macro or an identifier declared with external linkage. Portable programs must not make assumptions about the implementation of **errno**.

The value of **errno** is zero at program startup, but is never set to zero by any library function. The value of **errno** may be set to a nonzero value by a library function, and this effect is documented in each function that does so.

The header file `<errno.h>` defines the macros **EDOM**, **EILSEQ**, and **ERANGE** which expand to integer constant expressions with type **int**, distinct positive values, and which are suitable for use in **#if** preprocessing directives.

Portability

**errno** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

## `<limits.h>` - Integer numerical limits

The header file `<limits.h>` defines macros that expand to various limits and parameters of the standard integer types.

### Type sizes

---

<b>CHAR_BIT</b>	Number of bits in a <b>char</b>
-----------------	---------------------------------

---

### Character minimum and maximum values

---

<b>CHAR_MIN</b>	Minimum value of a <b>char</b>
-----------------	--------------------------------

---

<b>CHAR_MAX</b>	Maximum value of a <b>char</b>
-----------------	--------------------------------

---

<b>SCHAR_MIN</b>	Minimum value of a <b>signed char</b>
------------------	---------------------------------------

---

<b>SCHAR_MAX</b>	Maximum value of a <b>signed char</b>
------------------	---------------------------------------

---

<b>UCHAR_MAX</b>	Maximum value of an <b>unsigned char</b>
------------------	--

---

### Short minimum and maximum values

---

<b>SHRT_MIN</b>	Minimum value of a <b>short</b>
-----------------	---------------------------------

---

<b>SHRT_MAX</b>	Maximum value of a <b>short</b>
-----------------	---------------------------------

---

<b>USHRT_MAX</b>	Maximum value of an <b>unsigned short</b>
------------------	---

---

### Integer minimum and maximum values

---

<b>INT_MIN</b>	Minimum value of an <b>int</b>
----------------	--------------------------------

---

<b>INT_MAX</b>	Maximum value of an <b>int</b>
----------------	--------------------------------

---

<b>UINT_MAX</b>	Maximum value of an <b>unsigned int</b>
-----------------	---

---

### Long integer minimum and maximum values

---

<b>LONG_MIN</b>	Minimum value of a <b>long</b>
-----------------	--------------------------------

---

<b>LONG_MAX</b>	Maximum value of a <b>long</b>
-----------------	--------------------------------

---

<b>ULONG_MAX</b>	Maximum value of an <b>unsigned long</b>
------------------	--

---



## Long long integer minimum and maximum values

---

**LLONG\_MIN** Minimum value of a **long long**

---

**LLONG\_MAX** Maximum value of a **long long**

---

**ULLONG\_MAX** Maximum value of an **unsigned long long**

---

## CHAR\_BIT

**Synopsis**

```
#include <limits.h>
#define CHAR_BIT 8
```

**Description** **CHAR\_BIT** is the number of bits for smallest object that is not a bit-field (byte).

**Portability** **CHAR\_BIT** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## CHAR\_MIN

**Synopsis**

```
#include <limits.h>
#define CHAR_MIN 0
```

**Description** **CHAR\_MIN** is the minimum value for an object of type **char**.

**Portability** **CHAR\_MIN** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## CHAR\_MAX

**Synopsis**

```
#include <limits.h>
#define CHAR_MAX 255
```

**Description** **CHAR\_MAX** is the maximum value for an object of type **char**.

**Portability** **CHAR\_MAX** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## INT\_MIN

**Synopsis**

```
#include <limits.h>
#define INT_MIN processor-dependent-value
```

**Description** **INT\_MIN** is the minimum value for an object of type **int**.  
For processors where an integer is held in 16 bits, **INT\_MIN** is -32768, and for processors where an integer is held in 32 bits, **INT\_MIN** is -2147483648.

**Portability** **INT\_MIN** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

---

## INT\_MAX

- Synopsis** `#include <limits.h>`  
`#define INT_MAX processor-dependent-value`
- Description** **INT\_MAX** is the maximum value for an object of type **int**.  
For processors where an integer is held in 16 bits, **INT\_MAX** is 32767, and for processors where an integer is held in 32 bits, **INT\_MAX** is 2147483647.
- Portability** **INT\_MAX** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
- 

## LLONG\_MIN

- Synopsis** `#include <limits.h>`  
`#define LLONG_MIN (-9223372036854775807-1)`
- Description** **LLONG\_MIN** is the minimum value for an object of type **long long int**.
- Portability** **LLONG\_MIN** conforms to ISO/IEC 9899:1999 (C99).
- 

## LLONG\_MAX

- Synopsis** `#include <limits.h>`  
`#define LLONG_MAX (-9223372036854775807-1)`
- Description** **LLONG\_MAX** is the maximum value for an object of type **long long int**.
- Portability** **LLONG\_MAX** conforms to ISO/IEC 9899:1999 (C99).
- 

## LONG\_MIN

- Synopsis** `#include <limits.h>`  
`#define LONG_MIN (-2147483647-1)`
- Description** **LONG\_MIN** is the minimum value for an object of type **long int**.
- Portability** **LONG\_MIN** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
- 

## LONG\_MAX

- Synopsis** `#include <limits.h>`  
`#define LONG_MAX 2147483647`
- Description** **LONG\_MAX** is the maximum value for an object of type **long int**.
- Portability** **LONG\_MAX** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
-

---

## SCHAR\_MIN

- Synopsis**     `#include <limits.h>`  
                  `#define SCHAR_MIN -127`
- Description**   **SCHAR\_MIN** is the minimum value for an object of type **signed char**.
- Portability**   **SCHAR\_MIN** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
- 

## SCHAR\_MAX

- Synopsis**     `#include <limits.h>`  
                  `#define SCHAR_MAX 127`
- Description**   **SCHAR\_MAX** is the maximum value for an object of type **signed char**.
- Portability**   **SCHAR\_MAX** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
- 

## SHRT\_MIN

- Synopsis**     `#include <limits.h>`  
                  `#define SHRT_MIN (-32767-1)`
- Description**   **SHRT\_MIN** is the minimum value for an object of type **short int**.
- Portability**   **SHRT\_MIN** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
- 

## SHRT\_MAX

- Synopsis**     `#include <limits.h>`  
                  `#define SHRT_MAX 32767`
- Description**   **SHRT\_MAX** is the maximum value for an object of type **short int**.
- Portability**   **SHRT\_MAX** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
- 

## UCHAR\_MAX

- Synopsis**     `#include <limits.h>`  
                  `#define UCHAR_MAX 255`
- Description**   **UCHAR\_MAX** is the maximum value for an object of type **unsigned char**.
- Portability**   **UCHAR\_MAX** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
-

---

## UINT\_MAX

- Synopsis** `#include <limits.h>`  
`#define UINT_MAX processor-dependent-value`
- Description** **UINT\_MAX** is the maximum value for an object of type **unsigned int**.  
For processors where an unsigned integer is held in 16 bits, **UINT\_MAX** is 65535, and for processors where an unsigned integer is held in 32 bits, **UINT\_MAX** is 4294967295.
- Portability** **UINT\_MAX** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
- 

## ULLONG\_MAX

- Synopsis** `#include <limits.h>`  
`#define ULLONG_MAX 18446744073709551615`
- Description** **ULLONG\_MAX** is the maximum value for an object of type **unsigned long long int**.
- Portability** **ULLONG\_MAX** conforms to ISO/IEC 9899:1999 (C99).
- 

## ULONG\_MAX

- Synopsis** `#include <limits.h>`  
`#define ULONG_MAX 2147483647`
- Description** **ULONG\_MAX** is the maximum value for an object of type **unsigned long int**.
- Portability** **ULONG\_MAX** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
- 

## USHRT\_MAX

- Synopsis** `#include <limits.h>`  
`#define USHRT_MAX 65535`
- Description** **USHRT\_MAX** is the maximum value for an object of type **unsigned short int**.
- Portability** **USHRT\_MAX** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
- 

## <math.h> - Mathematics

The header file **<math.h>** defines a number of types, macros, and mathematical functions.

### Classification functions

---

<code>isfinite</code>	Is floating value finite?
<code>isinf</code>	Is floating value an infinity?
<code>isnan</code>	Is floating value a NaN?

---

### Trigonometric functions

---

<code>sin</code>	Compute sine of a <b>double</b>
<code>sinf</code>	Compute sine of a <b>float</b>
<code>cos</code>	Compute cosine of a <b>double</b>
<code>cosf</code>	Compute cosine of a <b>float</b>
<code>tan</code>	Compute tangent of a <b>double</b>
<code>tanf</code>	Compute tangent of a <b>float</b>

---

### Inverse trigonometric functions

---

<code>asin</code>	Compute inverse sine of a <b>double</b>
<code>asinf</code>	Compute inverse sine of a <b>float</b>
<code>acos</code>	Compute inverse cosine of a <b>double</b>
<code>acosf</code>	Compute inverse cosine of a <b>float</b>
<code>atan</code>	Compute inverse tangent of a <b>double</b>
<code>atanf</code>	Compute inverse tangent of a <b>float</b>
<code>atan2</code>	Compute inverse tangent of a ratio of <b>doubles</b>
<code>atan2f</code>	Compute inverse tangent of a ratio of <b>floats</b>

---

### Inverse hyperbolic functions

---

<code>acosh</code>	Compute inverse hyperbolic cosine of a <b>double</b>
<code>acoshf</code>	Compute inverse hyperbolic cosine of a <b>float</b>
<code>asinh</code>	Compute inverse hyperbolic sine of a <b>double</b>
<code>asinhf</code>	Compute inverse hyperbolic sine of a <b>float</b>
<code>atanh</code>	Compute inverse hyperbolic tangent of a <b>double</b>
<code>atanhf</code>	Compute inverse hyperbolic tangent of a <b>float</b>

---

### Hyperbolic functions

---

<code>cosh</code>	Compute hyperbolic cosine of a <b>double</b>
-------------------	--

---

<code>coshf</code>	Compute hyperbolic cosine of a <b>float</b>
<code>sinh</code>	Compute hyperbolic sine of a <b>double</b>
<code>sinhf</code>	Compute hyperbolic sine of a <b>float</b>
<code>tanh</code>	Compute hyperbolic tangent of a <b>double</b>
<code>tanhf</code>	Compute hyperbolic tangent of a <b>float</b>

### Exponential and logarithmic functions

<code>exp</code>	Compute exponential of a <b>double</b>
<code>expf</code>	Compute exponential of a <b>float</b>
<code>frexp</code>	Set exponent of a <b>double</b>
<code>frexpf</code>	Set exponent of a <b>float</b>
<code>ldexp</code>	Adjust exponent of a <b>double</b>
<code>ldexpf</code>	Adjust exponent of a <b>float</b>
<code>log</code>	Compute natural logarithm of a <b>double</b>
<code>logf</code>	Compute natural logarithm of a <b>float</b>
<code>log10</code>	Compute common logarithm of a <b>double</b>
<code>log10f</code>	Compute common logarithm of a <b>float</b>

### Power functions

<code>sqrt</code>	Compute square root of a <b>double</b>
<code>sqrtf</code>	Compute square root of a <b>float</b>
<code>cbrt</code>	Compute cube root of a <b>double</b>
<code>cbrtf</code>	Compute cube root of a <b>float</b>
<code>pow</code>	Raise a <b>double</b> to a power
<code>powf</code>	Raise a <b>float</b> to a power

### Absolute value functions

<code>fabs</code>	Compute absolute value of a <b>double</b>
<code>fabsf</code>	Compute absolute value of a <b>float</b>
<code>hypot</code>	Compute complex magnitude of two <b>doubles</b>
<code>hypotf</code>	Compute complex magnitude of two <b>floats</b>

## Remainder functions

---

<b>fmod</b>	Compute remainder after division of two <b>doubles</b>
<b>fmodf</b>	Compute remainder after division of two <b>floats</b>
<b>modf</b>	Break a <b>double</b> to integer and fractional parts
<b>modff</b>	Break a <b>float</b> to integer and fractional parts

---

## Maximum, minimum, and positive difference functions

---

<b>fmax</b>	Compute maximum of two <b>doubles</b>
<b>fmaxf</b>	Compute maximum of two <b>floats</b>
<b>fmin</b>	Compute minimum of two <b>doubles</b>
<b>fminf</b>	Compute minimum of two <b>floats</b>

---

## Nearest integer functions

---

<b>ceil</b>	Compute smallest integer not greater than a <b>double</b>
<b>ceilf</b>	Compute smallest integer not greater than a <b>float</b>
<b>floor</b>	Compute largest integer not greater than a <b>double</b>
<b>floorf</b>	Compute largest integer not greater than a <b>float</b>

---

---

## acos

**Synopsis** `#include <math.h>`  
`double acos(double x);`

**Description** **acos** returns the principal value, in radians, of the inverse circular cosine of **x**. The principal value lies in the interval  $[0, \text{PI}]$  radians.

**Fast math library behavior** If  $|x| > 1$ , **errno** is set to **EDOM** and **acos** returns **HUGE\_VAL**.

**IEC 60559 math library behavior** If **x** is NaN, **acos** returns **x**.  
If  $|x| > 1$ , **acos** returns NaN with invalid signal.

**Portability** **acos** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## acosf

**Synopsis** `#include <math.h>`  
`float acosf(float x);`

**Description** `acosf` returns the principal value, in radians, of the inverse circular cosine of `x`. The principal value lies in the interval  $[0, \text{PI}]$  radians.

**Fast math library behavior** If  $|x| > 1$ , `errno` is set to `EDOM` and `acosf` returns `HUGE_VAL`.

**IEC 60559 math behavior** If `x` is NaN, `acosf` returns `x`.  
If  $|x| > 1$ , `acosf` returns NaN with invalid signal.

**Portability** `acosf` conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## acosh

**Synopsis**  

```
#include <math.h>
double acosh(double x);
```

**Description** `acosh` returns the non-negative inverse hyperbolic cosine of `x`.  
`acosh(x)` is defined as  $\log(x + \sqrt{x^2 - 1})$ , assuming completely accurate computation.

**Fast math library behavior** If  $x < 1$ , `errno` is set to `EDOM` and `acosh` returns `HUGE_VAL`.

**IEC 60559 math library behavior** If  $x < 1$ , `acosh` returns NaN with signal.  
If `x` is NaN, `acosh` returns NaN without signal.

**Portability** `acosh` conforms to ISO/IEC 9899:1999 (C99).

---

## acoshf

**Synopsis**  

```
#include <math.h>
float acoshf(float x);
```

**Description** `acoshf` returns the non-negative inverse hyperbolic cosine of `x`.

**Fast math library behavior** If  $x < 1$ , `errno` is set to `EDOM` and `acoshf` returns `HUGE_VALF`.

**IEC 60559 math library behavior** If  $x < 1$ , `acoshf` returns NaN with signal.  
If `x` is NaN, `acoshf` returns NaN without signal.

**Portability** `acoshf` conforms to ISO/IEC 9899:1999 (C99).

---

## asin

**Synopsis**  

```
#include <math.h>
double asin(double x);
```



**Description** **asin** returns the principal value, in radians, of the inverse circular sine of **x**. The principal value lies in the interval  $[-\pi/2, +\pi/2]$  radians.

**Fast math library behavior** If  $|x| > 1$ , **errno** is set to **EDOM** and **asin** returns **HUGE\_VAL**.

**IEC 60559 math library behavior** If **x** is NaN, **asin** returns **x**.  
If  $|x| > 1$ , **asin** returns NaN with invalid signal.

**Portability** **asin** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## asinf

**Synopsis**

```
#include <math.h>
float asinf(float val);
```

**Description** **asinf** returns the principal value, in radians, of the inverse circular sine of **val**. The principal value lies in the interval  $[-\pi/2, +\pi/2]$  radians.

**Fast math library behavior** If  $|x| > 1$ , **errno** is set to **EDOM** and **asinf** returns **HUGE\_VALF**.

**IEC 60559 math library behavior** If **x** is NaN, **asinf** returns **x**.  
If  $|x| > 1$ , **asinf** returns NaN with invalid signal.

**Portability** **asinf** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## asinh

**Synopsis**

```
#include <math.h>
double asinh(double x);
```

**Description** **asinh** returns the inverse hyperbolic sine of **x**.

**Portability** **asinh** conforms to ISO/IEC 9899:1999 (C99).

---

## asinhf

**Synopsis**

```
#include <math.h>
float asinhf(float x);
```

**Description** **asinhf** returns the inverse hyperbolic sine of **x**.

**Portability** **asinhf** conforms to ISO/IEC 9899:1999 (C99).

---

## atan

- Synopsis** `#include <math.h>`  
`double atan(double x);`
- Description** **atan** returns the principal value, in radians, of the inverse circular tangent of **x**. The principal value lies in the interval  $[-\frac{3}{4}\pi, +\frac{3}{4}\pi]$  radians.
- Portability** **atan** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
- See Also** **atan2** (page 258)

---

## atan2

- Synopsis** `#include <math.h>`  
`double atan2(double y, double x);`
- Description** **atan2** returns the value, in radians, of the inverse circular tangent of **y** divided by **x** using the signs of **x** and **y** to compute the quadrant of the return value. The principal value lies in the interval  $[-\pi/2, +\pi/2]$  radians.
- Fast math library behavior** If **x** = **y** = 0, **errno** (page 247) is set to **EDOM** and **atan2** returns **HUGE\_VAL**.
- IEC 60559 math library behavior** **atan2f**(*x*, NaN) is NaN  
**atan2f**(NaN, *x*) is NaN  
**atan2f**(0, +(anything but NaN)) is 0  
**atan2f**(0, -(anything but NaN)) is ?  
**atan2f**((anything but 0 and NaN), 0) is  $\pi/2$   
**atan2f**((anything but Infinity and NaN), +Infinity) is 0  
**atan2f**((anything but Infinity and NaN), -Infinity) is ?  
**atan2f**(Infinity, +Infinity) is  $\pi/4$   
**atan2f**(Infinity, -Infinity) is  $3\pi/4$   
**atan2f**(Infinity, (anything but 0, NaN, and Infinity)) is  $\pi/2$
- Portability** **atan2** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
- See Also** **atan** (page 258)

---

## atan2f

- Synopsis** `#include <math.h>`  
`float atan2f(float y, float x);`
- Description** **atan2f** returns the value, in radians, of the inverse circular tangent of **y** divided by **x** using the signs of **x** and **y** to compute the quadrant of the return value. The principal value lies in the interval  $[-\pi/2, +\pi/2]$  radians.

Fast math library behavior

If  $x = y = 0$ , **errno** (page 247) is set to **EDOM** and **atan2f** returns **HUGE\_VALF**.

IEC 60559 math library behavior

**atan2f**( $x$ , NaN) is NaN

**atan2f**(NaN,  $x$ ) is NaN

**atan2f**(0, +(anything but NaN)) is 0

**atan2f**(0, -(anything but NaN)) is ?

**atan2f**((anything but 0 and NaN), 0) is  $?/2$

**atan2f**((anything but Infinity and NaN), +Infinity) is 0

**atan2f**((anything but Infinity and NaN), -Infinity) is ?

**atan2f**(Infinity, +Infinity) is  $?/4$

**atan2f**(Infinity, -Infinity) is  $3?/4$

**atan2f**(Infinity, (anything but 0, NaN, and Infinity)) is  $?/2$

Portability

**atan2f** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

See Also

**atanf** (page 259)

---

## atanf

Synopsis

```
#include <math.h>
float atanf(float x);
```

Description

**atanf** returns the principal value, in radians, of the inverse circular tangent of  $x$ . The principal value lies in the interval  $[-\frac{3}{4}\pi, +\frac{3}{4}\pi]$  radians.

Portability

**atanf** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## atanh

Synopsis

```
#include <math.h>
double atanh(double x)
```

Description

**atanh** returns the inverse hyperbolic tangent of  $x$ .

Fast math library

If  $|x| \geq 1$ , **errno** is set to **EDOM** and **atanh** returns **HUGE\_VAL**.

IEC 60559 math library behavior

If  $|x| > 1$  **atanh** returns NaN with signal.

If  $x$  is NaN, **atanh** returns that NaN with no signal.

If  $x$  is 1, **atanh** returns Infinity with signal.

If  $x$  is -1, **atanh** returns -Infinity with signal.

Portability

**atanh** conforms to ISO/IEC 9899:1999 (C99).

---

## atanhf

Synopsis	<pre>#include &lt;math.h&gt; float atanhf(float val)</pre>
Description	<b>atanhf</b> returns the inverse hyperbolic tangent of <b>val</b> .
Fast math library behavior	If $ x  > 1$ , <b>errno</b> is set to <b>EDOM</b> and <b>atanhf</b> returns <b>HUGE_VALF</b> .
IEC 60559 math library behavior	If $ val  > 1$ <b>atanhf</b> returns NaN with signal. If <b>val</b> is NaN, <b>atanhf</b> returns that NaN with no signal. If <b>val</b> is 1, <b>atanhf</b> returns Infinity with signal. If <b>val</b> is -1, <b>atanhf</b> returns -Infinity with signal.
Portability	<b>atanhf</b> conforms to ISO/IEC 9899:1999 (C99).

---

## cbrt

Synopsis	<pre>#include &lt;math.h&gt; double cbrt(double x);</pre>
Description	<b>cbrt</b> computes the cube root of <b>x</b> .
Portability	<b>cbrt</b> conforms to ISO/IEC 9899:1999 (C99).

---

## cbrtf

Synopsis	<pre>#include &lt;math.h&gt; float cbrt(float x);</pre>
Description	<b>cbrtf</b> computes the cube root of <b>x</b> .
Portability	<b>cbrtf</b> conforms to ISO/IEC 9899:1999 (C99).

---

## ceil

Synopsis	<pre>#include &lt;math.h&gt; double ceil(double x);</pre>
Description	<b>ceil</b> computes the smallest integer value not less than <b>x</b> .
IEC 60559 math library behavior	<b>ceil(0)</b> is 0. <b>ceil(Infinity)</b> is Infinity.
Portability	<b>ceil</b> conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## ceilf

Synopsis	<pre>#include &lt;math.h&gt; float ceilf(float x);</pre>
Description	<b>ceilf</b> computes the smallest integer value not less than <i>x</i> .
IEC 60559 math library behavior	<b>ceilf</b> (0) is 0. <b>ceilf</b> (Infinity) is Infinity.
Portability	<b>ceilf</b> conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## cos

Synopsis	<pre>#include &lt;math.h&gt; double cos(double x);</pre>
Description	<b>cos</b> returns the radian circular cosine of <i>x</i> .
Fast math library behavior	If $ x  > 10^9$ , <b>errno</b> is set to <b>EDOM</b> and <b>cos</b> returns <b>HUGE_VAL</b> .
IEC 60559 math library behavior	If <i>x</i> is NaN, <b>cos</b> returns <i>x</i> . If $ x $ is Infinity, <b>cos</b> returns NaN with invalid signal.
Portability	<b>cos</b> conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## cosf

Synopsis	<pre>#include &lt;math.h&gt; float cosf(float x);</pre>
Description	<b>cosf</b> returns the radian circular cosine of <i>x</i> .
Fast math library behavior	If $ x  > 10^9$ , <b>errno</b> is set to <b>EDOM</b> and <b>cosf</b> returns <b>HUGE_VALF</b> .
IEC 60559 math library behavior	If <i>x</i> is NaN, <b>cosf</b> returns <i>x</i> . If $ x $ is Infinity, <b>cosf</b> returns NaN with invalid signal.
Portability	<b>cosf</b> conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## cosh

Synopsis	<pre>#include &lt;math.h&gt; double cosh(double x);</pre>
Description	<b>cosh</b> calculates the hyperbolic cosine of <i>x</i> .

Fast math library behavior

If  $|x| > \sim 709.782$ , **errno** is set to **EDOM** and **cosh** returns **HUGE\_VAL**.

IEC 60559 math library behavior

If  $x$  is **+Infinity**, **-Infinity**, or **NaN**, **cosh** returns  $|x|$ .

If  $|x| > \sim 709.782$ , **cosh** returns **+Infinity** or **-Infinity** depending upon the sign of  $x$ .

Portability

**cosh** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## coshf

Synopsis

```
#include <math.h>
float coshf(float x);
```

Description

**coshf** calculates the hyperbolic sine of  $x$ .

Fast math library behavior

If  $|x| > \sim 88.7228$ , **errno** is set to **EDOM** and **coshf** returns **HUGE\_VALF**.

IEC 60559 math library behavior

If  $x$  is **+Infinity**, **-Infinity**, or **NaN**, **coshf** returns  $|x|$ .

If  $|x| > \sim 88.7228$ , **coshf** returns **+Infinity** or **-Infinity** depending upon the sign of  $x$ .

Portability

**coshf** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## exp

Synopsis

```
#include <math.h>
double exp(double x);
```

Description

**exp** computes the base- $e$  exponential of  $x$ .

Fast math library behavior

If  $|x| > \sim 709.782$ , **errno** is set to **EDOM** and **exp** returns **HUGE\_VAL**.

IEC 60559 math library behavior

If  $x$  is **NaN**, **exp** returns **NaN**.

If  $x$  is **Infinity**, **exp** returns **Infinity**.

If  $x$  is **-Infinity**, **exp** returns 0.

Portability

**exp** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## expf

Synopsis

```
#include <math.h>
float expf(float x);
```

Description

**expf** computes the base- $e$  exponential of  $x$ .

Fast math library behavior

If  $|x| > \sim 88.722$ , **errno** is set to **EDOM** and **expf** returns **HUGE\_VALF**.

IEC 60559 math  
library behavior

If  $x$  is NaN, **expf** returns NaN.  
If  $x$  is Infinity, **expf** returns Infinity  
If  $x$  is -Infinity, **expf** returns 0.

Portability **expf** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## fabs

Synopsis 

```
#include <math.h>
double fabs(double x);
```

Description **fabs** computes the absolute value of the floating-point number  $x$ .

Portability **fabs** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## fabsf

Synopsis 

```
#include <math.h>
float fabsf(float x);
```

Description **fabsf** computes the absolute value of the floating-point number  $x$ .

Portability **fabsf** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## floor

Synopsis 

```
#include <math.h>
double floor(double x);
```

Description **floor** computes the largest integer value not greater than  $x$ .

IEC 60559 math  
library behavior **floor**(0) is 0.  
**floor**(Infinity) is Infinity.

Portability **floor** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## floorf

Synopsis 

```
#include <math.h>
double floorf(double x);
```

Description **floorf** computes the largest integer value not greater than  $x$ .

IEC 60559 math  
library behavior **floorf**(0) is 0.  
**floorf**(Infinity) is Infinity.

Portability **floorf** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## **fmax**

Synopsis	<pre>#include &lt;math.h&gt; double fmax(double x, double y);</pre>
Description	<b>fmax</b> determines the minimum of <b>x</b> and <b>y</b> .
IEC 60559 math library behavior	<b>fmax</b> (NaN, <b>y</b> ) is <b>y</b> . <b>fmax</b> ( <b>x</b> , NaN) is <b>x</b> .
Portability	<b>fmax</b> conforms to ISO/IEC 9899:1999 (C99).

---

## **fmaxf**

Synopsis	<pre>#include &lt;math.h&gt; float fmaxf(float x, float y);</pre>
Description	<b>fmaxf</b> determines the minimum of <b>x</b> and <b>y</b> .
IEC 60559 math library behavior	<b>fmaxf</b> (NaN, <b>y</b> ) is <b>y</b> . <b>fmaxf</b> ( <b>x</b> , NaN) is <b>x</b> .
Portability	<b>fmaxf</b> conforms to ISO/IEC 9899:1999 (C99).

---

## **fmin**

Synopsis	<pre>#include &lt;math.h&gt; double fmin(double x, double y);</pre>
Description	<b>fmin</b> determines the minimum of <b>x</b> and <b>y</b> .
IEC 60559 math library behavior	<b>fmin</b> (NaN, <b>y</b> ) is <b>y</b> . <b>fmin</b> ( <b>x</b> , NaN) is <b>x</b> .
Portability	<b>fmin</b> conforms to ISO/IEC 9899:1999 (C99).

---

## **fminf**

Synopsis	<pre>#include &lt;math.h&gt; float fminf(float x, float y);</pre>
Description	<b>fminf</b> determines the minimum of <b>x</b> and <b>y</b> .
IEC 60559 math library behavior	<b>fminf</b> (NaN, <b>y</b> ) is <b>y</b> . <b>fminf</b> ( <b>x</b> , NaN) is <b>x</b> .
Portability	<b>fminf</b> conforms to ISO/IEC 9899:1999 (C99).



---

## fmod

**Synopsis**

```
#include <math.h>
double fmod(double x, double y);
```

**Description** **fmod** computes the floating-point remainder of **x** divided by **y**. **fmod** returns the value  $x - ny$ , for some integer  $n$  such that, if **y** is nonzero, the result has the same sign as **x** and magnitude less than the magnitude of **y**.

**Fast math library behavior** If **y** = 0, **fmod** returns zero and **errno** is set to **EDOM**.

**IEC 60559 math library behavior** **fmod**(0, **y**) is 0 for **y** not zero.  
**fmod**(Infinity, **y**) is NaN and raises the “invalid” floating-point exception.  
**fmod**(**x**, 0) is NaN and raises the “invalid” floating-point exception.  
**fmod**(**x**, Infinity) is **x** for **x** not infinite.

**Portability** **fmod** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## fmodf

**Synopsis**

```
#include <math.h>
float fmodf(float x, float y);
```

**Description** **fmodf** computes the floating-point remainder of **x** divided by **y**. **fmodf** returns the value  $x - ny$ , for some integer  $n$  such that, if **y** is nonzero, the result has the same sign as **x** and magnitude less than the magnitude of **y**.

**Fast math library behavior** If **y** = 0, **fmodf** returns zero and **errno** is set to **EDOM**.

**IEC 60559 math library behavior** **fmodf**(0, **y**) is 0 for **y** not zero.  
**fmodf**(Infinity, **y**) is NaN and raises the “invalid” floating-point exception.  
**fmodf**(**x**, 0) is NaN and raises the “invalid” floating-point exception.  
**fmodf**(**x**, Infinity) is **x** for **x** not infinite.

**Portability** **fmodf** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## frexp

**Synopsis**

```
#include <math.h>
double frexp(double x, int *exp);
```

**Description** **frexp** breaks a floating-point number into a normalized fraction and an integral power of 2.

**frexp** stores power of two in the **int** object pointed to by **exp** and returns the value **x**, such that **x** has a magnitude in the interval  $[1/2, 1)$  or zero, and value equals  $x * 2^{\text{exp}}$ .  
If **x** is zero, both parts of the result are zero.

IEC 60559 math  
library behavior

If **x** is Infinity or NaN, **frexp** returns **x** and stores zero into the **int** object pointed to by **exp**.

Portability

**frexp** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## frexpf

Synopsis 

```
#include <math.h>
float frexpf(float x, int *exp);
```

Description **frexpf** breaks a floating-point number into a normalized fraction and an integral power of 2.

**frexpf** stores power of two in the **int** object pointed to by **exp** and returns the value **x**, such that **x** has a magnitude in the interval  $[1/2, 1)$  or zero, and value equals  $x * 2^{\text{exp}}$ .  
If **x** is zero, both parts of the result are zero.

IEC 60559 math  
library behavior

If **x** is Infinity or NaN, **frexpf** returns **x** and stores zero into the **int** object pointed to by **exp**.

Portability

**frexpf** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## hypot

Synopsis 

```
#include <math.h>
double hypot(double x, double y);
```

Description **hypot** compute the square root of the sum of the squares of **x** and **y**,  $\text{sqrt}(x^2 + y^2)$ , without undue overflow or underflow. If **x** and **y** are the lengths of the sides of a right-angled triangle, then **hypot** computes the length of the hypotenuse..

IEC 60559 math  
library behavior

If **x** or **y** is +Infinity or -Infinity, **hypot** returns Infinity.  
If **x** or **y** is NaN, **hypot** returns NaN.

Portability

**hypot** conforms to ISO/IEC 9899:1999 (C99).

---

## hypotf

Synopsis 

```
#include <math.h>
float hypotf(float x, float y);
```

**Description** **hypotf** compute the square root of the sum of the squares of **x** and **y**, **sqrtf(x\*x + y\*y)**, without undue overflow or underflow. If **x** and **y** are the lengths of the sides of a right-angled triangle, then **hypotf** computes the length of the hypotenuse..

**IEC 60559 math library behavior** If **x** or **y** is +Infinity or -Infinity, **hypotf** returns Infinity. If **x** or **y** is NaN, **hypotf** returns NaN.

**Portability** **hypotf** conforms to ISO/IEC 9899:1999 (C99).

---

## isfinite

**Synopsis**

```
#include <math.h>
int isfinite(floating-type x);
```

**Description** **isfinite** determines whether **x** is a finite value (zero, subnormal, or normal, and not infinite or NaN). The **isfinite** macro returns a non-zero value if and only if its argument has a finite value.

**Fast math library behavior** As the fast math library does not support NaN and infinite values, **isfinite** always returns a non-zero value.

**Portability** **isfinite** conforms to ISO/IEC 9899:1999 (C99).

---

## isinf

**Synopsis**

```
#include <math.h>
int isinf(floating-type x);
```

**Description** **isinf** determines whether its argument value is an infinity (positive or negative). The determination is based on the type of the argument.

**Portability** **isinf** confirms to ISO/IEC 9899:1999 (C99).

---

## isnan

**Synopsis**

```
#include <math.h>
int isnan(floating-type x);
```

**Description** **isnan** determines whether its argument value is a NaN. The determination is based on the type of the argument.

**Portability** **isnan** confirms to ISO/IEC 9899:1999 (C99).

---

---

## ldexp

Synopsis	<pre>#include &lt;math.h&gt; double ldexp(double x, int exp);</pre>
Description	<b>ldexp</b> multiplies a floating-point number by an integral power of 2. <b>ldexp</b> returns $x * 2^{\text{exp}}$ .
Fast math library behavior	If the result overflows, <b>errno</b> is set to <b>ERANGE</b> and <b>ldexp</b> returns <b>HUGE_VAL</b> .
IEC 60559 math library behavior	If <b>x</b> is Infinity or NaN, <b>ldexp</b> returns <b>x</b> . If the result overflows, <b>ldexp</b> returns Infinity.
Portability	<b>ldexp</b> conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## ldexpf

Synopsis	<pre>#include &lt;math.h&gt; float ldexpf(float x, int exp);</pre>
Description	<b>ldexpf</b> multiplies a floating-point number by an integral power of 2. <b>ldexpf</b> returns $x * 2^{\text{exp}}$ .
Fast math library behavior	If the result overflows, <b>errno</b> is set to <b>ERANGE</b> and <b>ldexpf</b> returns <b>HUGE_VALF</b> .
IEC 60559 math library behavior	If <b>x</b> is Infinity or NaN, <b>ldexpf</b> returns <b>x</b> . If the result overflows, <b>ldexpf</b> returns Infinity.
Portability	<b>ldexpf</b> conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## log

Synopsis	<pre>#include &lt;math.h&gt; double log(double x);</pre>
Description	<b>log</b> computes the base- <i>e</i> logarithm of <b>x</b> .
Fast math library behavior	If <b>x</b> = 0, <b>errno</b> is set to <b>ERANGE</b> and <b>log</b> returns <b>-HUGE_VAL</b> . If <b>x</b> < 0, <b>errno</b> is set to <b>EDOM</b> and <b>log</b> returns <b>-HUGE_VAL</b> .
IEC 60559 math library behavior	If <b>x</b> < 0 or <b>x</b> = -Infinity, <b>log</b> returns NaN with signal. If <b>x</b> = 0, <b>log</b> returns -Infinity with signal. If <b>x</b> = Infinity, <b>log</b> returns Infinity. If <b>x</b> = NaN, <b>log</b> returns <b>x</b> with no signal.
Portability	<b>log</b> conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## log10

Synopsis	<pre>#include &lt;math.h&gt; double log10(double x);</pre>
Description	<b>log10</b> computes the base-10 logarithm of $x$ .
Fast math library behavior	If $x = 0$ , <b>errno</b> is set to <b>ERANGE</b> and <b>log10</b> returns <b>-HUGE_VAL</b> . If $x < 0$ , <b>errno</b> is set to <b>EDOM</b> and <b>log10</b> returns <b>-HUGE_VAL</b> .
IEC 60559 math library behavior	If $x < 0$ or $x = -\text{Infinity}$ , <b>log10</b> returns NaN with signal. If $x = 0$ , <b>log10</b> returns $-\text{Infinity}$ with signal. If $x = \text{Infinity}$ , <b>log10</b> returns $\text{Infinity}$ . If $x = \text{NaN}$ , <b>log10</b> returns $x$ with no signal.
Portability	<b>log10</b> conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## log10f

Synopsis	<pre>#include &lt;math.h&gt; float log10f(float x);</pre>
Description	<b>log10f</b> computes the base-10 logarithm of $x$ .
Fast math library behavior	If $x = 0$ , <b>errno</b> is set to <b>ERANGE</b> and <b>log10f</b> returns <b>-HUGE_VALF</b> . If $x < 0$ , <b>errno</b> is set to <b>EDOM</b> and <b>log10f</b> returns <b>-HUGE_VALF</b> .
IEC 60559 math library behavior	If $x < 0$ or $x = -\text{Infinity}$ , <b>log10f</b> returns NaN with signal. If $x = 0$ , <b>log10f</b> returns $-\text{Infinity}$ with signal. If $x = \text{Infinity}$ , <b>log10f</b> returns $\text{Infinity}$ . If $x = \text{NaN}$ , <b>log10f</b> returns $x$ with no signal.
Portability	<b>log10f</b> conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## logf

Synopsis	<pre>#include &lt;math.h&gt; float logf(float x);</pre>
Description	<b>logf</b> computes the base- $e$ logarithm of $x$ .
Fast math library behavior	If $x = 0$ , <b>errno</b> is set to <b>ERANGE</b> and <b>logf</b> returns <b>-HUGE_VALF</b> . If $x < 0$ , <b>errno</b> is set to <b>EDOM</b> and <b>logf</b> returns <b>-HUGE_VALF</b> .

IEC 60559 math  
library behavior

If  $x < 0$  or  $x = -\text{Infinity}$ , **logf** returns NaN with signal.

If  $x = 0$ , **logf** returns  $-\text{Infinity}$  with signal.

If  $x = \text{Infinity}$ , **logf** returns  $\text{Infinity}$ .

If  $x = \text{NaN}$ , **logf** returns  $x$  with no signal.

Portability

**logf** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## modf

Synopsis

```
#include <math.h>
double modf(double x, double *iptr);
```

Description

**modf** breaks  $x$  into integral and fractional parts, each of which has the same type and sign as  $x$ .

The integral part (in floating-point format) is stored in the object pointed to by **iptr** and **modf** returns the signed fractional part of  $x$ .

Portability

**modf** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## modff

Synopsis

```
#include <math.h>
float modff(float x, double *iptr);
```

Description

**modff** breaks  $x$  into integral and fractional parts, each of which has the same type and sign as  $x$ .

The integral part (in floating-point format) is stored in the object pointed to by **iptr** and **modff** returns the signed fractional part of  $x$ .

Portability

**modff** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## pow

Synopsis

```
#include <math.h>
double pow(double x, double y);
```

Description

**pow** computes  $x$  raised to the power  $y$ .

Fast math library  
behavior

If  $x < 0$  and  $y \leq 0$ , **errno** is set to **EDOM** and **pow** returns **-HUGE\_VAL**.

If  $x \leq 0$  and  $y$  is not an integer value, **errno** is set to **EDOM** and **pow** returns **-HUGE\_VAL**.

IEC 60559 math  
library behavior

If  $y = 0$ , **pow** returns 1.

If  $y = 1$ , **pow** returns  $x$ .

If  $y = \text{NaN}$ , **pow** returns NaN.

If  $x = \text{NaN}$  and  $y$  is anything other than 0, **pow** returns NaN.

If  $x < -1$  or  $1 < x$ , and  $y = +\text{Infinity}$ , **pow** returns  $+\text{Infinity}$ .  
 If  $x < -1$  or  $1 < x$ , and  $y = -\text{Infinity}$ , **pow** returns 0.  
 If  $-1 < x < 1$  and  $y = +\text{Infinity}$ , **pow** returns +0.  
 If  $-1 < x < 1$  and  $y = -\text{Infinity}$ , **pow** returns  $+\text{Infinity}$ .  
 If  $x = +1$  or  $x = -1$  and  $y = +\text{Infinity}$  or  $y = -\text{Infinity}$ , **pow** returns NaN.  
 If  $x = +0$  and  $y > 0$  and  $y \neq \text{NaN}$ , **pow** returns +0.  
 If  $x = -0$  and  $y > 0$  and  $y \neq \text{NaN}$  or  $y$  not an odd integer, **pow** returns +0.  
 If  $x = +0$  and  $y < 0$  and  $y \neq \text{NaN}$ , **pow** returns  $+\text{Infinity}$ .  
 If  $x = -0$  and  $y > 0$  and  $y \neq \text{NaN}$  or  $y$  not an odd integer, **pow** returns  $+\text{Infinity}$ .  
 If  $x = -0$  and  $y$  is an odd integer, **pow** returns -0.  
 If  $x = +\text{Infinity}$  and  $y > 0$  and  $y \neq \text{NaN}$ , **pow** returns  $+\text{Infinity}$ .  
 If  $x = +\text{Infinity}$  and  $y < 0$  and  $y \neq \text{NaN}$ , **pow** returns +0.  
 If  $x = -\text{Infinity}$ , **pow** returns **pow**(-0,  $y$ )  
 If  $x < 0$  and  $x \neq \text{Infinity}$  and  $y$  is a non-integer, **pow** returns NaN.

Portability **pow** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

## powf

Synopsis 

```
#include <math.h>
float powf(float x, float y);
```

Description **powf** computes  $x$  raised to the power  $y$ .

Fast math library behavior If  $x < 0$  and  $y \leq 0$ , **errno** (page 247) is set to **EDOM** and **powf** returns **-HUGE\_VALF**. If  $x \leq 0$  and  $y$  is not an integer value, **errno** (page 247) is set to **EDOM** and **powf** returns **-HUGE\_VALF**.

IEC 60559 math library behavior

If  $y = 0$ , **powf** returns 1.  
 If  $y = 1$ , **powf** returns  $x$ .  
 If  $y = \text{NaN}$ , **powf** returns NaN.  
 If  $x = \text{NaN}$  and  $y$  is anything other than 0, **powf** returns NaN.  
 If  $x < -1$  or  $1 < x$ , and  $y = +\text{Infinity}$ , **powf** returns  $+\text{Infinity}$ .  
 If  $x < -1$  or  $1 < x$ , and  $y = -\text{Infinity}$ , **powf** returns 0.  
 If  $-1 < x < 1$  and  $y = +\text{Infinity}$ , **powf** returns +0.  
 If  $-1 < x < 1$  and  $y = -\text{Infinity}$ , **powf** returns  $+\text{Infinity}$ .  
 If  $x = +1$  or  $x = -1$  and  $y = +\text{Infinity}$  or  $y = -\text{Infinity}$ , **powf** returns NaN.  
 If  $x = +0$  and  $y > 0$  and  $y \neq \text{NaN}$ , **powf** returns +0.  
 If  $x = -0$  and  $y > 0$  and  $y \neq \text{NaN}$  or  $y$  not an odd integer, **powf** returns +0.  
 If  $x = +0$  and  $y < 0$  and  $y \neq \text{NaN}$ , **powf** returns  $+\text{Infinity}$ .  
 If  $x = -0$  and  $y > 0$  and  $y \neq \text{NaN}$  or  $y$  not an odd integer, **powf** returns  $+\text{Infinity}$ .  
 If  $x = -0$  and  $y$  is an odd integer, **powf** returns -0.  
 If  $x = +\text{Infinity}$  and  $y > 0$  and  $y \neq \text{NaN}$ , **powf** returns  $+\text{Infinity}$ .

If  $x = +\text{Infinity}$  and  $y < 0$  and  $y \neq \text{NaN}$ , **powf** returns  $+0$ .  
If  $x = -\text{Infinity}$ , **powf** returns **powf**( $-0$ ,  $y$ )  
If  $x < 0$  and  $x \neq \text{Infinity}$  and  $y$  is a non-integer, **powf** returns NaN.

**Portability** **powf** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## scalbn

**Synopsis**

```
#include <math.h>
double scalbn(double x, int exp);
```

**Description** **scalbn** multiplies a floating-point number by an integral power of **FLT\_RADIX**. As floating-point arithmetic conforms to IEC 60559, **FLT\_RADIX** is 2 and **scalbn** is (in this implementation) identical to **ldexp**.  
**scalbn** returns  $x * \text{FLT\_RADIX}^{\text{exp}}$ .

**Fast math library behavior** If the result overflows, **errno** is set to **ERANGE** and **scalbn** returns **HUGE\_VAL**.

**IEC 60559 math library behavior** If  $x$  is Infinity or NaN, **scalbn** returns  $x$ .  
If the result overflows, **scalbn** returns Infinity.

**Portability** **scalbn** conforms to ISO/IEC 9899:1999 (C99).

**See Also** **ldexp** (page 268)

---

## scalbnf

**Synopsis**

```
#include <math.h>
float scalbnf(float x, int exp);
```

**Description** **scalbnf** multiplies a floating-point number by an integral power of **FLT\_RADIX**. As floating-point arithmetic conforms to IEC 60559, **FLT\_RADIX** is 2 and **scalbnf** is (in this implementation) identical to **ldexpf**.  
**scalbnf** returns  $x * \text{FLT\_RADIX}^{\text{exp}}$ .

**Fast math library behavior** If the result overflows, **errno** (page 247) is set to **ERANGE** and **scalbnf** returns **HUGE\_VALF**.

**IEC 60559 math library behavior** If  $x$  is Infinity or NaN, **scalbnf** returns  $x$ .  
If the result overflows, **scalbnf** returns Infinity.

**Portability** **scalbnf** conforms to ISO/IEC 9899:1999 (C99).

**See Also** **ldexpf** (page 268)

---



---

## sin

Synopsis	<pre>#include &lt;math.h&gt; double sin(double x);</pre>
Description	<b>sin</b> returns the radian circular sine of <b>x</b> .
Fast math library behavior	If $ x  > 10^9$ , <b>errno</b> (page 247) is set to <b>EDOM</b> and <b>sin</b> returns <b>HUGE_VAL</b> .
IEC 60559 math library behavior	<b>sin</b> returns <b>x</b> if <b>x</b> is NaN. <b>sin</b> returns NaN with invalid signal if $ x $ is Infinity.
Portability	<b>sin</b> conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## sinf

Synopsis	<pre>#include &lt;math.h&gt; float sinf(float x);</pre>
Description	<b>sinf</b> returns the radian circular sine of <b>x</b> .
Fast math library special cases	If $ x  > 10^9$ , <b>errno</b> (page 247) is set to <b>EDOM</b> and <b>sin</b> returns <b>HUGE_VALF</b> .
IEC 60559 math library special cases	<b>sinf</b> returns <b>x</b> if <b>x</b> is NaN. <b>sinf</b> returns NaN with invalid signal if $ x $ is Infinity.
Portability	<b>sinf</b> conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## sinh

Synopsis	<pre>#include &lt;math.h&gt; double sinh(double x);</pre>
Description	<b>sinh</b> calculates the hyperbolic sine of <b>x</b> .
Fast math library behavior	If $ x  > \sim 709.782$ , <b>errno</b> (page 247) is set to <b>EDOM</b> and <b>sinh</b> returns <b>HUGE_VAL</b> .
IEC 60559 math library behavior	If <b>x</b> is +Infinity, -Infinity, or NaN, <b>sinh</b> returns $ x $ . If $ x  > \sim 709.782$ , <b>sinh</b> returns +Infinity or -Infinity depending upon the sign of <b>x</b> .
Portability	<b>sinh</b> conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## sinhf

Synopsis	<pre>#include &lt;math.h&gt; float sinh(float x);</pre>
Description	<b>sinhf</b> calculates the hyperbolic sine of <b>x</b> .
Fast math library behavior	If $ x  > \sim 88.7228$ , <b>errno</b> (page 247) is set to <b>EDOM</b> and <b>sinhf</b> returns <b>HUGE_VALF</b> .
IEC 60559 math library behavior	If <b>x</b> is <b>+Infinity</b> , <b>-Infinity</b> , or <b>NaN</b> , <b>sinhf</b> returns $ x $ . If $ x  > \sim 88.7228$ , <b>sinhf</b> returns <b>+Infinity</b> or <b>-Infinity</b> depending upon the sign of <b>x</b> .
Portability	<b>sinhf</b> conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## sqrt

Synopsis	<pre>#include &lt;math.h&gt; double sqrt(double val);</pre>
Description	<b>sqrt</b> computes the nonnegative square root of <b>val</b> . C90 and C99 require that a domain error occurs if the argument is less than zero. CrossWorks C deviates and always uses IEC 60559 semantics.
Special cases	If <b>val</b> is <b>+0</b> , <b>sqrt</b> returns <b>+0</b> . If <b>val</b> is <b>-0</b> , <b>sqrt</b> returns <b>-0</b> . If <b>val</b> is <b>Infinity</b> , <b>sqrt</b> returns <b>Infinity</b> . If <b>val</b> $< 0$ , <b>sqrt</b> returns <b>NaN</b> with invalid signal. If <b>val</b> is <b>NaN</b> , <b>sqrt</b> returns that <b>NaN</b> with invalid signal for signaling <b>NaN</b> .
Portability	<b>sqrt</b> conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99) except in the case of domain errors.

---

## sqrtf

Synopsis	<pre>#include &lt;math.h&gt; float sqrtf(float val);</pre>
Description	<b>sqrtf</b> computes the nonnegative square root of <b>val</b> . C90 and C99 require that a domain error occurs if the argument is less than zero. CrossWorks C deviates and always uses IEC 60559 semantics.
Special cases	If <b>val</b> is <b>+0</b> , <b>sqrtf</b> returns <b>+0</b> . If <b>val</b> is <b>-0</b> , <b>sqrtf</b> returns <b>-0</b> . If <b>val</b> is <b>Infinity</b> , <b>sqrtf</b> returns <b>Infinity</b> . If <b>val</b> $< 0$ , <b>sqrtf</b> returns <b>NaN</b> with invalid signal. If <b>val</b> is <b>NaN</b> , <b>sqrtf</b> returns that <b>NaN</b> with invalid signal for signaling <b>NaN</b> .

**Portability** `sqrtf` conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99) except in the case of domain errors.

---

## **tan**

**Synopsis**

```
#include <math.h>
double tan(double x);
```

**Description** `tan` returns the radian circular tangent of `x`.

**Fast math library behaviour** If  $|x| > 10^9$ , `errno` (page 247) is set to `EDOM` and `tan` returns `HUGE_VAL`.

**IEC 60559 math library behaviour** If `x` is NaN, `tan` returns `x`.  
If  $|x|$  is Infinity, `tan` returns NaN with invalid signal.

**Portability** `tan` conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## **tanf**

**Synopsis**

```
#include <math.h>
float tanf(float x);
```

**Description** `tanf` returns the radian circular tangent of `x`.

**Fast math library special cases** If  $|x| > 10^9$ , `errno` is set to `EDOM` and `tanf` returns `HUGE_VALF`.

**IEC 60559 math library special cases** If `x` is NaN, `tanf` returns `x`.  
If  $|x|$  is Infinity, `tanf` returns NaN with invalid signal.

**Portability** `tanf` conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## **tanh**

**Synopsis**

```
#include <math.h>
double tanh(double x);
```

**Description** `tanh` calculates the hyperbolic tangent of `x`.

**IEC 60559 math library behavior** If `x` is NaN, `tanh` returns NaN.

**Portability** `tanh` conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## **tanhf**

**Synopsis** `#include <math.h>`  
`float tanhf(float x);`

**Description** **tanhf** calculates the hyperbolic tangent of *x*.

**IEC 60559 math library behavior** If *x* is NaN, **tanhf** returns NaN.

**Portability** **tanhf** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

## <setjmp.h> - Non-local jumps

The header file **<setjmp.h>** defines macros and functions for non-local flow of control, commonly used to implement exception handling in a C program.

### Types

---

<b>jmp_buf</b>	Structure to hold processor state
----------------	-----------------------------------

---

### Functions

---

<b>longjmp</b>	Non-local jump to saved state
<b>setjmp</b>	Save state for non-local jump

---

---

## **jmp\_buf**

**Synopsis** `#include <setjmp.h>`  
`typedef implementation-defined-type jmp_buf[];`

**Description** The type **jmp\_buf** is an array type suitable for holding the information needed to restore a calling environment. The environment of a call to **setjmp** consists of information sufficient for a call to the **longjmp** function to return execution to the correct block and invocation of that block, were it called recursively. It does not include the state of the floating-point status flags, of open files, or of any other component of the machine.

**Portability** **jmp\_buf** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

**See also** **longjmp** (page 277), **setjmp** (page 277)

---

## longjmp

**Synopsis**

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val);
```

**Description** **longjmp** restores the environment saved by the most recent invocation of **setjmp** with the corresponding **jmp\_buf** argument. If there has been no such invocation, or if the function containing the invocation of **setjmp** has terminated execution in the interim, the behavior of **longjmp** is undefined.

When the environment is restored, all accessible objects have values and state as of the time the **longjmp** function was called.

After **longjmp** is completed, program execution continues as if the corresponding invocation of **setjmp** had just returned the value specified by **val**. Note that **longjmp** cannot cause **setjmp** to return the value 0; if **val** is 0, **setjmp** returns the value 1.

**Important notes** Objects of automatic storage duration that are local to the function containing the invocation of the corresponding **setjmp** that do not have **volatile**-qualified type and have been changed between the **setjmp** invocation and **longjmp** call are indeterminate.

**Portability** **longjmp** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

**See also** **setjmp** (page 277)

---

## setjmp

**Synopsis**

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

**Description** **setjmp** saves its calling environment in the **jmp\_buf** argument **env** for later use by the **longjmp** function.

On return from a direct invocation, **setjmp** returns the value zero. If the return is from a call to the **longjmp** function, the **setjmp** macro returns a nonzero value determined by the call to **longjmp**.

The ISO standard does not specify whether **setjmp** is a macro or an identifier declared with external linkage. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with the name **setjmp**, the behavior of **setjmp** is undefined.

**Portability** **setjmp** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

**See also** **longjmp** (page 277)

## <stdarg.h> - Variable arguments

The header file <stdarg.h> defines a number of macros to access variable parameter lists.

### Functions

---

<b>va_end</b>	Start access to variable arguments
<b>va_arg</b>	Get variable argument value
<b>va_end</b>	Finish access to variable arguments
<b>va_copy</b>	Copy <b>va_arg</b> structure

---

---

### va\_arg

**Synopsis**

```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

**Description** **va\_arg** expands to an expression that has the specified type and the value of the **type** argument. The **ap** parameter must have been initialized by **va\_start** or **va\_copy**, without an intervening invocation of **va\_end**. You can create a pointer to a **va\_list** and pass that pointer to another function, in which case the original function may make further use of the original list after the other function returns.

Each invocation of the **va\_arg** macro modifies **ap** so that the values of successive arguments are returned in turn. The parameter type must be a type name such that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a '\*' to **type**.

If there is no actual next argument, or if **type** is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior of **va\_arg** is undefined, except for the following cases:

- one type is a signed integer type, the other type is the corresponding unsigned integer type, and the value is representable in both types;
- one type is pointer to **void** and the other is a pointer to a character type.

The first invocation of the **va\_arg** macro after that of the **va\_start** macro returns the value of the argument after that specified by **parmN**. Successive invocations return the values of the remaining arguments in succession.

**Examples** When calling **va\_arg**, you must ensure that **type** is the *promoted type* of the argument, not the argument type. The following will not work as you expect:

```
char x = va_arg(ap, char);
```

Because characters are promoted to integers, the above must be written:

```
char ch = (char)va_arg(ap, int);
```

**Portability** **va\_arg** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

**See also** **va\_copy** (page 279), **va\_end** (page 279), **va\_end** (page 279)

---

## **va\_copy**

**Synopsis**

```
#include <stdarg.h>
void va_copy(va_list dest, va_list src);
```

**Description** **va\_copy** initializes **dest** as a copy of **src**, as if the **va\_start** macro had been applied to **dest** followed by the same sequence of uses of the **va\_arg** macro as had previously been used to reach the present state of **src**. Neither the **va\_copy** nor **va\_start** macro shall be invoked to reinitialize **dest** without an intervening invocation of the **va\_end** macro for the same **dest**.

**Portability** **va\_copy** conforms to ISO/IEC 9899:1999 (C99).

**See also** **va\_arg** (page 278), **va\_end** (page 279), **va\_end** (page 279)

---

## **va\_end**

**Synopsis**

```
#include <stdarg.h>
void va_end(va_list ap);
```

**Description** **va\_end** indicates a normal return from the function whose variable argument list **ap** was initialised by **va\_start** or **va\_copy**. The **va\_end** macro may modify **ap** so that it is no longer usable without being reinitialized by **va\_start** or **va\_copy**. If there is no corresponding invocation of **va\_start** or **va\_copy**, or if **va\_end** is not invoked before the return, the behavior is undefined.

**Portability** **va\_end** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

**See also** **va\_arg** (page 278), **va\_copy** (page 279), **va\_end** (page 279)

---

## **va\_end**

**Synopsis**

```
#include <stdarg.h>
void va_start(va_list ap, parmN);
```

**Description** **va\_start** initializes **ap** for subsequent use by the **va\_arg** and **va\_end** macros. The parameter **parmN** is the identifier of the last fixed parameter in the variable parameter list in the function definition (the one just before the ', ...').

The behaviour of `va_start` and `va_arg` is undefined if the parameter `parmN` is declared with the `register` storage class, with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions.

`va_start` must be invoked before any access to the unnamed arguments.

`va_start` and `va_copy` must not be invoked to reinitialize `ap` without an intervening invocation of the `va_end` macro for the same `ap`.

**Portability** `va_start` conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

**See also** `va_arg` (page 278), `va_copy` (page 279), `va_end` (page 279)

## <stdio.h> - Input/output functions

The header file `<stdio.h>` defines a number of functions to format and output values. The format-control directives that for the formatted input and output function are described in **Formatted input control strings** (page 285) and **Formatted output control strings** (page 281).

### Character and string I/O functions

<code>getchar</code>	Read a character from standard input
<code>gets</code>	Read a string from standard input
<code>putchar</code>	Write a character to standard output
<code>puts</code>	Write a string to standard output

### Formatted input functions

<code>scanf</code>	Read formatted text from standard input
<code>sscanf</code>	Read formatted text from a string
<code>vscanf</code>	Read formatted text from standard input using a <code>va_list</code> argument
<code>vsscanf</code>	Read formatted text from a string using a <code>va_list</code> argument

### Formatted output functions

<code>printf</code>	Write formatted text to standard output
<code>snprintf</code>	Write formatted text to a string with truncation
<code>sprintf</code>	Write formatted text to a string



<b>vprintf</b>	Write formatted text to standard output using a <b>va_list</b> argument
<b>vsprintf</b>	Write formatted text to a string with truncation using a <b>va_list</b> argument
<b>vsnprintf</b>	Write formatted text to a string using a <b>va_list</b> argument

## Formatted output control strings

The format is composed of zero or more directives: ordinary characters (not '%'), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments, converting them, if applicable, according to the corresponding conversion specifier, and then writing the result to the output stream.

### Overview

Each conversion specification is introduced by the character '%'. After the '%', the following appear in sequence:

- Zero or more *flags* (in any order) that modify the meaning of the conversion specification.
- An optional *minimum field width*. If the converted value has fewer characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag has been given) to the field width. The field width takes the form of an asterisk '\*' or a decimal integer.
- An optional precision that gives the minimum number of digits to appear for the 'd', 'i', 'o', 'u', 'x', and 'X' conversions, the number of digits to appear after the decimal-point character for 'e', 'E', 'f', and 'F' conversions, the maximum number of significant digits for the 'g' and 'G' conversions, or the maximum number of bytes to be written for 's' conversions. The precision takes the form of a period '.' followed either by an asterisk '\*' or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.
- An optional length modifier that specifies the size of the argument.
- A conversion specifier character that specifies the type of conversion to be applied.

As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an int argument supplies the field width or precision. The arguments specifying field width, or precision, or both, must appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a '-' flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

Some CrossWorks library variants do not support width and precision specifiers in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Width/Precision Support** property of the project if you use these.

### Flag characters

The flag characters and their meanings are:

- **'-'**. The result of the conversion is left-justified within the field. The default, if this flag is not specified, is that the result of the conversion is left-justified within the field.
- **'+'**. The result of a signed conversion *always* begins with a plus or minus sign. The default, if this flag is not specified, is that it begins with a sign only when a negative value is converted.
- **space**. If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space is prefixed to the result. If the space and **'+'** flags both appear, the space flag is ignored.
- **'#'**. The result is converted to an *alternative form*. For **'o'** conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both zero, a single **'0'** is printed). For **'x'** or **'X'** conversion, a nonzero result has **'0x'** or **'0X'** prefixed to it. For **'e'**, **'E'**, **'f'**, **'F'**, **'g'**, and **'G'** conversions, the result of converting a floating-point number always contains a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if a digit follows it.) For **'g'** and **'F'** conversions, trailing zeros are not removed from the result. As an extension, when used in **'p'** conversion, the results has **'#'** prefixed to it. For other conversions, the behavior is undefined.
- **'0'**. For **'d'**, **'i'**, **'o'**, **'u'**, **'x'**, **'X'**, **'e'**, **'E'**, **'f'**, **'F'**, **'g'**, and **'G'** conversions, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing space padding, except when converting an infinity or NaN. If the **'0'** and **'-'** flags both appear, the **'0'** flag is ignored. For **'d'**, **'i'**, **'o'**, **'u'**, **'x'**, and **'X'** conversions, if a precision is specified, the **'0'** flag is ignored. For other conversions, the behavior is undefined.

### Length modifiers

The length modifiers and their meanings are:

- **'hh'**. Specifies that a following **'d'**, **'i'**, **'o'**, **'u'**, **'x'**, or **'X'** conversion specifier applies to a **signed char** or **unsigned char** argument (the argument will have been promoted according to the integer promotions, but its value will be converted to **signed char** or **unsigned char** before printing); or that a following **'n'** conversion specifier applies to a pointer to a **signed char** argument.
- **'h'**. Specifies that a following **'d'**, **'i'**, **'o'**, **'u'**, **'x'**, or **'X'** conversion specifier applies to a **short int** or **unsigned short int** argument (the argument will

have been promoted according to the integer promotions, but its value is converted to **short int** or **unsigned short int** before printing); or that a following **'n'** conversion specifier applies to a pointer to a **short int** argument.

- **'l'**. Specifies that a following **'d'**, **'i'**, **'o'**, **'u'**, **'x'**, or **'X'** conversion specifier applies to a **long int** or **unsigned long int** argument; that a following **'n'** conversion specifier applies to a pointer to a **long int** argument; or has no effect on a following **'e'**, **'E'**, **'f'**, **'F'**, **'g'**, or **'G'** conversion specifier. Some CrossWorks library variants do not support the **'l'** length modifier in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Integer Support** property of the project if you use this length modifier.
- **'ll'**. Specifies that a following **'d'**, **'i'**, **'o'**, **'u'**, **'x'**, or **'X'** conversion specifier applies to a **long long int** or **unsigned long long int** argument; that a following **'n'** conversion specifier applies to a pointer to a **long long int** argument. Some CrossWorks library variants do not support the **'ll'** length modifier in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Integer Support** property of the project if you use this length modifier.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined. Note that the C99 length modifiers **'j'**, **'z'**, **'t'**, and **'L'** are not supported.

### Conversion specifiers

The conversion specifiers and their meanings are:

- **'d', 'i'**. The argument is converted to signed decimal in the style `[-]dddd`. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading spaces. The default precision is one. The result of converting a zero value with a precision of zero is no characters.
- **'o', 'u', 'x', 'X'**. The unsigned argument is converted to unsigned octal for **'o'**, unsigned decimal for **'u'**, or unsigned hexadecimal notation for **'x'** or **'X'** in the style `dddd`; the letters **'abcdef'** are used for **'x'** conversion and the letters **'ABCDEF'** for **'X'** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading spaces. The default precision is one. The result of converting a zero value with a precision of zero is no characters.
- **'f', 'F'**. A double argument representing a floating-point number is converted to decimal notation in the style `[-]ddd.ddd`, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the **'#'** flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits. A double argument representing an infinity is converted to **'inf'**. A double argument representing a NaN is converted to **'nan'**. The **'F'** conversion specifier

produces **'INF'** or **'NAN'** instead of **'inf'** or **'nan'**, respectively. Some CrossWorks library variants do not support the **'f'** and **'F'** conversion specifiers in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Floating Point Support** property of the project if you use these conversion specifiers.

- **'e', 'E'**. A double argument representing a floating-point number is converted in the style `[-]d.dddedd`, where there is one digit (which is nonzero if the argument is nonzero) before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the **'#'** flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The **'E'** conversion specifier produces a number with **'E'** instead of **'e'** introducing the exponent. The exponent always contains at least two digits, and only as many more digits as necessary to represent the exponent. If the value is zero, the exponent is zero. A double argument representing an infinity is converted to **'inf'**. A double argument representing a NaN is converted to **'nan'**. The **'E'** conversion specifier produces **'INF'** or **'NAN'** instead of **'inf'** or **'nan'**, respectively. Some CrossWorks library variants do not support the **'f'** and **'F'** conversion specifiers in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Floating Point Support** property of the project if you use these conversion specifiers.
- **'g', 'G'**. A double argument representing a floating-point number is converted in style **'f'** or **'e'** (or in style **'F'** or **'E'** in the case of a **'G'** conversion specifier), with the precision specifying the number of significant digits. If the precision is zero, it is taken as one. The style used depends on the value converted; style **'e'** (or **'E'**) is used only if the exponent resulting from such a conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result unless the **'#'** flag is specified; a decimal-point character appears only if it is followed by a digit. A double argument representing an infinity is converted to **'inf'**. A double argument representing a NaN is converted to **'nan'**. The **'G'** conversion specifier produces **'INF'** or **'NAN'** instead of **'inf'** or **'nan'**, respectively. Some CrossWorks library variants do not support the **'f'** and **'F'** conversion specifiers in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Floating Point Support** property of the project if you use these conversion specifiers.
- **'c'**. The argument is converted to an **unsigned char**, and the resulting character is written.
- **'s'**. The argument is be a pointer to the initial element of an array of character type. Characters from the array are written up to (but not including) the terminating null character. If the precision is specified, no more than that many characters are written. If the precision is not specified or is greater than the size of the array, the array must contain a null character.

- **'p'**. The argument is a pointer to **void**. The value of the pointer is converted in the same format as the **'x'** conversion specifier with a fixed precision of  $2 * \text{sizeof}(\text{void} *)$ .
- **'n'**. The argument is a pointer to signed integer into which is *written* the number of characters written to the output stream so far by the call to the formatting function. No argument is converted, but one is consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.
- **'%'**. A **'%'** character is written. No argument is converted.

Note that the C99 width modifier **'l'** used in conjunction with the **'c'** and **'s'** conversion specifiers is not supported and nor are the conversion specifiers **'a'** and **'A'**.

## Formatted input control strings

The format is composed of zero or more directives: one or more white-space characters, an ordinary character (neither **'%'** nor a white-space character), or a conversion specification.

### Overview

Each conversion specification is introduced by the character **'%'**. After the **'%'**, the following appear in sequence:

- An optional assignment-suppressing character **'\*'**.
- An optional nonzero decimal integer that specifies the maximum field width (in characters).
- An optional length modifier that specifies the size of the receiving object.
- A conversion specifier character that specifies the type of conversion to be applied.

The formatted input function executes each directive of the format in turn. If a directive fails, the function returns. Failures are described as input failures (because of the occurrence of an encoding error or the unavailability of input characters), or matching failures (because of inappropriate input).

A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read.

A directive that is an ordinary character is executed by reading the next characters of the stream. If any of those characters differ from the ones composing the directive, the directive fails and the differing and subsequent characters remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a character from being read, the directive fails.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

- Input white-space characters (as specified by the `isspace` function) are skipped, unless the specification includes a `'l'`, `'c'`, or `'n'` specifier.
- An input item is read from the stream, unless the specification includes an `n` specifier. An input item is defined as the longest sequence of input characters which does not exceed any specified field width and which is, or is a prefix of, a matching input sequence. The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails; this condition is a matching failure unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.
- Except in the case of a `'%'` specifier, the input item (or, in the case of a `%n` directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a `'*'`, the result of the conversion is placed in the object pointed to by the first argument following the format argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the object, the behavior is undefined.

### Length modifiers

The length modifiers and their meanings are:

- `'hh'`. Specifies that a following `'d'`, `'i'`, `'o'`, `'u'`, `'x'`, `'X'`, or `'n'` conversion specifier applies to an argument with type pointer to **signed char** or pointer to **unsigned char**.
- `'h'`. Specifies that a following `'d'`, `'i'`, `'o'`, `'u'`, `'x'`, `'X'`, or `'n'` conversion specifier applies to an argument with type pointer to **short int** or **unsigned short int**.
- `'l'`. Specifies that a following `'d'`, `'i'`, `'o'`, `'u'`, `'x'`, `'X'`, or `'n'` conversion specifier applies to an argument with type pointer to **long int** or **unsigned long int**; that a following `'e'`, `'E'`, `'f'`, `'F'`, `'g'`, or `'G'` conversion specifier applies to an argument with type pointer to **double**. Some CrossWorks library variants do not support the `'l'` length modifier in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Printf Integer Support** property of the project if you use this length modifier.
- `'ll'`. Specifies that a following `'d'`, `'i'`, `'o'`, `'u'`, `'x'`, `'X'`, or `'n'` conversion specifier applies to an argument with type pointer to **long long int** or **unsigned long long int**. Some CrossWorks library variants do not support the `'ll'` length modifier in order to reduce code and data space requirements; please ensure that you have

selected the correct library in the **Printf Integer Support** property of the project if you use this length modifier.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined. Note that the C99 length modifiers `'j'`, `'z'`, `'t'`, and `'L'` are not supported.

### Conversion specifiers

- `'d'`. Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the `strtol` function with the value 10 for the `base` argument. The corresponding argument must be a pointer to signed integer.
- `'i'`. Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the `strtol` function with the value zero for the `base` argument. The corresponding argument must be a pointer to signed integer.
- `'o'`. Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the `strtol` function with the value 18 for the `base` argument. The corresponding argument must be a pointer to signed integer.
- `'u'`. Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the `strtoul` function with the value 10 for the `base` argument. The corresponding argument must be a pointer to unsigned integer.
- `'x'`. Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the `strtoul` function with the value 16 for the `base` argument. The corresponding argument must be a pointer to unsigned integer.
- `'e'`, `'f'`, `'g'`. Matches an optionally signed floating-point number whose format is the same as expected for the subject sequence of the `strtod` function. The corresponding argument shall be a pointer to floating. Some CrossWorks library variants do not support the `'e'`, `'f'` and `'F'` conversion specifiers in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Scanf Floating Point Support** property of the project if you use these conversion specifiers.
- `'c'`. Matches a sequence of characters of exactly the number specified by the field width (one if no field width is present in the directive). The corresponding argument must be a pointer to the initial element of a character array large enough to accept the sequence. No null character is added.
- `'s'`. Matches a sequence of non-white-space characters. The corresponding argument must be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

- **'[']**. Matches a nonempty sequence of characters from a set of expected characters (the *scanset*). The corresponding argument must be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically. The conversion specifier includes all subsequent characters in the format string, up to and including the matching right bracket **']'**. The characters between the brackets (the *scanlist*) compose the scanset, unless the character after the left bracket is a circumflex **'^'**, in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with **'[]'** or **'[^]'**, the right bracket character is in the scanlist and the next following right bracket character is the matching right bracket that ends the specification; otherwise the first following right bracket character is the one that ends the specification. If a **'-'** character is in the scanlist and is not the first, nor the second where the first character is a **'^'**, nor the last character, it is treated as a member of the scanset. Some CrossWorks library variants do not support the **'[']** conversion specifier in order to reduce code and data space requirements; please ensure that you have selected the correct library in the **Scanf Classes Supported** property of the project if you use this conversion specifier.
- **'p'**. Reads a sequence output by the corresponding **'%p'** formatted output conversion. The corresponding argument must be a pointer to a pointer to **void**.
- **'n'**. No input is consumed. The corresponding argument shall be a pointer to signed integer into which is to be written the number of characters read from the input stream so far by this call to the formatted input function. Execution of a **'%n'** directive does not increment the assignment count returned at the completion of execution of the `fscanf` function. No argument is converted, but one is consumed. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.
- **'%'**. Matches a single **'%'** character; no conversion or assignment occurs.

Note that the C99 width modifier **'l'** used in conjunction with the **'c'**, **'s'**, and **'[']** conversion specifiers is not supported and nor are the conversion specifiers **'a'** and **'A'**.

---

## getchar

**Synopsis** `#include <stdio.h>  
int getchar(void);`

**Description** `getchar` reads a single character from the standard input stream. If the stream is at end-of-file or a read error occurs, `getc` returns **EOF**.

**Portability** `getchar` conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).



---

## gets

**Synopsis**

```
#include <stdio.h>
char *gets(char *s);
```

**Description** **gets** reads characters from standard input into the array pointed to by **s** until end-of-file is encountered or a new-line character is read. Any new-line character is discarded, and a null character is written immediately after the last character read into the array.

**gets** returns **s** if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and **gets** returns a null pointer. If a read error occurs during the operation, the array contents are indeterminate and **gets** returns a null pointer.

**Portability**

**gets** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

**See also** **getchar** (page 288)

---

## printf

**Synopsis**

```
#include <stdio.h>
int printf(const char *format, ...);
```

**Description** **printf** writes to the standard output stream using **putchar**, under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

**printf** returns number of characters transmitted, or a negative value if an output or encoding error occurred.

**Portability** **printf** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

**See Also** **Formatted output control strings** (page 281)

---

## putchar

**Synopsis**

```
#include <stdio.h>
int putchar(int c);
```

**Description** **putchar** writes the character **c** to the standard output stream.

**putchar** returns the character written. If a write error occurs, **putchar** returns EOF.

**Portability** **putchar** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

**See also** Customizing putchar, **puts** (page 290)

---

## puts

**Synopsis**

```
#include <stdio.h>
int puts(const char *s);
```

**Description** **puts** writes the string pointed to by **s** to the standard output stream using **putchar** and appends a new-line character to the output. The terminating null character is not written.

**puts** returns **EOF** if a write error occurs; otherwise it returns a nonnegative value.

**Portability** **puts** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

**See also** **putchar** (page 289)

---

## scanf

**Synopsis**

```
#include <stdio.h>
int scanf(const char *format, ...);
```

**Description** **scanf** reads input from the standard input stream under control of the string pointed to by **format** that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

**scanf** returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, **scanf** returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

**Portability** **scanf** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

**See Also** **Formatted input control strings** (page 285)

---

## snprintf

**Synopsis**

```
#include <stdio.h>
int snprintf(char *s, size_t n, const char *format, ...);
```

**Description** **snprintf** writes to the string pointed to by **s** under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output.

If **n** is zero, nothing is written, and **s** can be a null pointer. Otherwise, output characters beyond the **n**-1st are discarded rather than being written to the array, and a null character is written at the end of the characters actually written into the array. A null character is written at the end of the conversion; it is not counted as part of the returned value.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

If copying takes place between objects that overlap, the behavior is undefined.

**snprintf** returns the number of characters that would have been written had **n** been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than **n**.

**Portability** **snprintf** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

**See Also** **Formatted output control strings** (page 281)

---

## sprintf

**Synopsis**

```
#include <stdio.h>
int sprintf(char *s, const char *format, ...);
```

**Description** **sprintf** writes to the string pointed to by **s** under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output. A null character is written at the end of the characters written; it is not counted as part of the returned value.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

If copying takes place between objects that overlap, the behavior is undefined.

**sprintf** returns number of characters transmitted (not counting the terminating null), or a negative value if an output or encoding error occurred.

**Portability** **sprintf** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

**See Also** **Formatted output control strings** (page 281)

---

## sscanf

**Synopsis**

```
#include <stdio.h>
int sscanf(const char *s, const char *format, ...);
```

Description	<p><b>sscanf</b> reads input from the string <b>s</b> under control of the string pointed to by <b>format</b> that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input.</p> <p>If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.</p> <p><b>sscanf</b> returns the value of the macro <b>EOF</b> if an input failure occurs before any conversion. Otherwise, <b>sscanf</b> returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.</p>
Portability	<b>sscanf</b> conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
See Also	<b>Formatted input control strings</b> (page 285)

## vprintf

Synopsis	<pre>#include &lt;stdio.h&gt; int vprintf(const char *format, va_list arg);</pre>
Description	<p><b>vprintf</b> writes to the standard output stream using <b>putchar</b>, under control of the string pointed to by <b>format</b> that specifies how subsequent arguments are converted for output. Before calling <b>vprintf</b>, <b>arg</b> must be initialized by the <b>va_start</b> macro (and possibly subsequent <b>va_arg</b> calls). <b>vprintf</b> does not invoke the <b>va_end</b> macro.</p> <p><b>vprintf</b> returns number of characters transmitted, or a negative value if an output or encoding error occurred.</p>
Notes	<b>vprintf</b> is equivalent to <b>printf</b> with the variable argument list replaced by <b>arg</b> .
Portability	<b>vprintf</b> conforms to ISO/IEC 9899:1999 (C99).
See Also	<b>Formatted output control strings</b> (page 281)

## vscanf

Synopsis	<pre>#include &lt;stdio.h&gt; int vscanf(const char *format, va_list arg);</pre>
Description	<p><b>vscanf</b> reads input from the standard input stream under control of the string pointed to by <b>format</b> that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. Before calling <b>vscanf</b>, <b>arg</b> must be initialized by the <b>va_start</b> macro (and possibly subsequent <b>va_arg</b> calls). <b>vscanf</b> does not invoke the <b>va_end</b> macro.</p> <p>If there are insufficient arguments for the format, the behavior is undefined.</p>

**vscanf** returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, **vscanf** returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

**Notes** **vscanf** is equivalent to **scanf** with the variable argument list replaced by **arg**.

**Portability** **vscanf** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

**See Also** **Formatted input control strings** (page 285)

---

## **vsnprintf**

**Synopsis**

```
#include <stdio.h>
int vsnprintf(char *s, size_t n, const char *format, va_list arg);
```

**Description** **vsnprintf** writes to the string pointed to by **s** under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output. Before calling **vsnprintf**, **arg** must be initialized by the **va\_start** macro (and possibly subsequent **va\_arg** calls). **vsnprintf** does not invoke the **va\_end** macro.

If **n** is zero, nothing is written, and **s** can be a null pointer. Otherwise, output characters beyond the **n**-1st are discarded rather than being written to the array, and a null character is written at the end of the characters actually written into the array. A null character is written at the end of the conversion; it is not counted as part of the returned value.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

If copying takes place between objects that overlap, the behavior is undefined.

**vsnprintf** returns the number of characters that would have been written had **n** been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than **n**.

**Notes** **vsnprintf** is equivalent to **snprintf** with the variable argument list replaced by **arg**.

**Portability** **vsnprintf** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

**See Also** **Formatted output control strings** (page 281)

---

## **vsprintf**

**Synopsis**

```
#include <stdio.h>
int vsprintf(char *s, const char *format, va_list arg);
```

Description	<p><b>vsprintf</b> writes to the string pointed to by <b>s</b> under control of the string pointed to by <b>format</b> that specifies how subsequent arguments are converted for output. Before calling <b>vsprintf</b>, <b>arg</b> must be initialized by the <b>va_start</b> macro (and possibly subsequent <b>va_arg</b> calls). <b>vsprintf</b> does not invoke the <b>va_end</b> macro.</p> <p>A null character is written at the end of the characters written; it is not counted as part of the returned value.</p> <p>If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.</p> <p>If copying takes place between objects that overlap, the behavior is undefined.</p> <p><b>vsprintf</b> returns number of characters transmitted (not counting the terminating null), or a negative value if an output or encoding error occurred.</p>
Notes	<b>vsprintf</b> is equivalent to <b>sprintf</b> with the variable argument list replaced by <b>arg</b> ,
Portability	<b>vsprintf</b> conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
See Also	<b>Formatted output control strings</b> (page 281)

## vsscanf

Synopsis	<pre>#include &lt;stdio.h&gt; int vsscanf(const char *s, const char *format, va_list arg);</pre>
Description	<p><b>vsscanf</b> reads input from the string <b>s</b> under control of the string pointed to by <b>format</b> that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. Before calling <b>vsscanf</b>, <b>arg</b> must be initialized by the <b>va_start</b> macro (and possibly subsequent <b>va_arg</b> calls). <b>vsscanf</b> does not invoke the <b>va_end</b> macro.</p> <p>If there are insufficient arguments for the format, the behavior is undefined.</p> <p><b>vsscanf</b> returns the value of the macro <b>EOF</b> if an input failure occurs before any conversion. Otherwise, <b>vsscanf</b> returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.</p>
Notes	<b>vsscanf</b> is equivalent to <b>sscanf</b> with the variable argument list replaced by <b>arg</b> .
Portability	<b>vsscanf</b> conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
See Also	<b>Formatted input control strings</b> (page 285)

## <stdlib.h> - General utilities

The header file <stdlib.h> defines a number of types, macros, and functions of general utility.

### Types

<b>div_t</b>	Structure containing quotient and remainder after division of <b>ints</b>
<b>ldiv_t</b>	Structure containing quotient and remainder after division of <b>longs</b>
<b>lldiv_t</b>	Structure containing quotient and remainder after division of <b>long longs</b>

### String to number conversions

<b>atoi</b>	Convert string to <b>int</b>
<b>atol</b>	Convert string to <b>long</b>
<b>atoll</b>	Convert string to <b>long long</b>
<b>strtol</b>	Convert string to <b>long</b>
<b>strtoll</b>	Convert string to <b>long long</b>
<b>strtoul</b>	Convert string to <b>unsigned long</b>
<b>strtoull</b>	Convert string to <b>unsigned long long</b>

### Number to string conversions

<b>itoa</b>	Convert <b>int</b> to string
<b>ltoa</b>	Convert <b>long</b> to string
<b>lltoa</b>	Convert <b>long long</b> to string
<b>utoa</b>	Convert <b>unsigned</b> to string
<b>ultoa</b>	Convert <b>unsigned long</b> to string
<b>ulltoa</b>	Convert <b>unsigned long long</b> to string

### Integer arithmetic functions

<b>div</b>	Divide two <b>ints</b> returning quotient and remainder
<b>ldiv</b>	Divide two <b>longs</b> returning quotient and remainder
<b>lldiv</b>	Divide two <b>long longs</b> returning quotient and remainder

## Pseudo-random sequence generation functions

---

<b>RAND_MAX</b>	Maximum value returned by <b>rand</b>
<b>rand</b>	Return next random number in sequence
<b>srand</b>	Set seed of random number sequence

---

## Memory allocation functions

---

<b>calloc</b>	Allocate space for an array of objects and initialize them to zero
<b>free</b>	Frees allocated memory for reuse
<b>malloc</b>	Allocate space for a single object
<b>realloc</b>	Resizes allocated memory space or allocates memory space

---

---

## atof

**Synopsis** `#include <stdlib.h>`  
`int atof(const char *nptr);`

**Description** **atof** converts the initial portion of the string pointed to by **nptr** to an **int** representation. **atof** does not affect the value of **errno** on an error. If the value of the result cannot be represented, the behavior is undefined.

Except for the behavior on error, **atof** is equivalent to `strtod(nptr, (char **)NULL)`.

**atoi** returns the converted value.

**Portability** **atoi** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

**See Also** **strtol** (page 303)

---

## atoi

**Synopsis** `#include <stdlib.h>`  
`int atoi(const char *nptr);`

**Description** **atoi** converts the initial portion of the string pointed to by **nptr** to an **int** representation. **atoi** does not affect the value of **errno** on an error. If the value of the result cannot be represented, the behavior is undefined.

Except for the behavior on error, **atoi** is equivalent to `(int)strtol(nptr, (char **)NULL, 10)`.

**atoi** returns the converted value.



**Portability** **atoi** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

**See Also** **strtol** (page 304)

---

## atoi

**Synopsis**

```
#include <stdlib.h>
long int atoi(const char *nptr);
```

**Description** **atoi** converts the initial portion of the string pointed to by **nptr** to a **long int** representation.

**atoi** does not affect the value of **errno** on an error. If the value of the result cannot be represented, the behavior is undefined.

Except for the behavior on error, **atoi** is equivalent to `strtol(nptr, (char **)NULL, 10)`.

**atoi** returns the converted value.

**Portability** **atoi** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

**See Also** **strtol** (page 304)

---

## atoll

**Synopsis**

```
#include <stdlib.h>
long int atoll(const char *nptr);
```

**Description** **atoll** converts the initial portion of the string pointed to by **nptr** to a **long long int** representation.

**atoll** does not affect the value of **errno** on an error. If the value of the result cannot be represented, the behavior is undefined.

Except for the behavior on error, **atoll** is equivalent to `strtoll(nptr, (char **)NULL, 10)`.

**atoll** returns the converted value.

**Portability** **atoll** conforms to ISO/IEC 9899:1999 (C99).

**See Also** **strtoll** (page 306)

---

## calloc

**Synopsis**

```
#include <stdlib.h>
void *calloc(size_t nmem, size_t size);
```

Description	<p><b>calloc</b> allocates space for an array of <b>nmemb</b> objects, each of whose size is <b>size</b>. The space is initialized to all zero bits.</p> <p><b>calloc</b> returns a null pointer if the space for the array of object cannot be allocated from free memory; if space for the array can be allocated, <b>calloc</b> returns a pointer to the start of the allocated space.</p>
Portability	<b>calloc</b> conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## div

Synopsis	<pre>#include &lt;stdlib.h&gt; div_t div(int numer, int denom);</pre>
Description	<p><b>div</b> computes <b>numer</b> / <b>denom</b> and <b>numer</b> % <b>denom</b> in a single operation.</p> <p><b>div</b> returns a structure of type <b>div_t</b> (page 298) comprising both the quotient and the remainder. The structures contain the members <b>quot</b> (the quotient) and <b>rem</b> (the remainder), each of which has the same type as the arguments <b>numer</b> and <b>denom</b>. If either part of the result cannot be represented, the behavior is undefined.</p>
Portability	<b>div</b> conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
See also	<b>div_t</b> (page 298)

---

## div\_t

Synopsis	<pre>#include &lt;stdlib.h&gt; typedef struct {     int quot;     int rem; } div_t;</pre>
Description	<b>div_t</b> stores the quotient and remainder returned by <b>div</b> (page 298).
Portability	<b>div_t</b> conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
See also	<b>div</b> (page 298)

---

## free

Synopsis	<pre>#include &lt;stdlib.h&gt; void free(void *ptr);</pre>
Description	<b>free</b> causes the space pointed to by <b>ptr</b> to be deallocated, that is, made available for further allocation. If <b>ptr</b> is a null pointer, no action occurs.

**Notes** If **ptr** does not match a pointer earlier returned by **calloc**, **malloc**, or **realloc**, or if the space has been deallocated by a call to **free** or **realloc**, the behaviour is undefined.

**Portability** **free** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## itoa

**Synopsis**

```
#include <stdlib.h>
char *itoa(int val, char *buf, int radix);
```

**Description** **itoa** converts **val** to a string in base **radix** and places the result in **buf**.  
**itoa** returns **buf** as the result.

If **radix** is greater than 36, the result is undefined.

If **val** is negative and **radix** is 10, the string has a leading minus sign (-); for all other values of **radix**, **value** is considered unsigned and never has a leading minus sign.

**Portability** **itoa** is an extension to the standard C library provided by *CrossWorks C*.

**See Also** **ltoa** (page 301), **lltoa** (page 300), **ultoa** (page 309), **ultoa** (page 309), **utoa** (page 310)

---

## ldiv

**Synopsis**

```
#include <stdlib.h>
ldiv_t ldiv(long int numer, long int denom);
```

**Description** **ldiv** computes **numer / denom** and **numer % denom** in a single operation.

**ldiv** returns a structure of type **ldiv\_t** (page 299) comprising both the quotient and the remainder. The structures contain the members **quot** (the quotient) and **rem** (the remainder), each of which has the same type as the arguments **numer** and **denom**. If either part of the result cannot be represented, the behavior is undefined.

**Portability** **ldiv** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

**See also** **ldiv\_t** (page 299)

---

## ldiv\_t

**Synopsis**

```
#include <stdlib.h>
typedef struct
{
    long int quot;
    long int rem;
} ldiv_t;
```

Description	<b>ldiv_t</b> stores the quotient and remainder returned by <b>ldiv</b> (page 299).
Portability	<b>ldiv_t</b> conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
See also	<b>ldiv</b> (page 299)

---

## lldiv

Synopsis	<pre>#include &lt;stdlib.h&gt; lldiv_t lldiv(long long int numer, long long int denom);</pre>
Description	<b>lldiv</b> computes <b>numer</b> / <b>denom</b> and <b>numer</b> % <b>denom</b> in a single operation.  <b>lldiv</b> returns a structure of type <b>lldiv_t</b> (page 300) comprising both the quotient and the remainder. The structures contain the members <b>quot</b> (the quotient) and <b>rem</b> (the remainder), each of which has the same type as the arguments <b>numer</b> and <b>denom</b> . If either part of the result cannot be represented, the behavior is undefined.
Portability	<b>lldiv</b> conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
See also	<b>lldiv_t</b> (page 300)

---

## lldiv\_t

Synopsis	<pre>#include &lt;stdlib.h&gt; typedef struct {     long long int quot;     long long int rem; } lldiv_t;</pre>
Description	<b>lldiv_t</b> stores the quotient and remainder returned by <b>lldiv</b> (page 300).
Portability	<b>lldiv_t</b> conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
See also	<b>lldiv</b> (page 300)

---

## lltoa

Synopsis	<pre>#include &lt;stdlib.h&gt; char *lltoa(long long val, char *buf, int radix);</pre>
Description	<b>lltoa</b> converts <b>val</b> to a string in base <b>radix</b> and places the result in <b>buf</b> . <b>lltoa</b> returns <b>buf</b> as the result. If <b>radix</b> is greater than 36, the result is undefined.

If **val** is negative and radix is 10, the string has a leading minus sign (-); for all other values of **radix**, **value** is considered unsigned and never has a leading minus sign.

**Portability** **ltoa** is an extension to the standard C library provided by *CrossWorks C*.

**See Also** **itoa** (page 299) **ltoa** (page 301) **ultoa** (page 309) **ultoa** (page 309) **utoa** (page 310)

---

## ltoa

**Synopsis**

```
#include <stdlib.h>
char *ltoa(long val, char *buf, int radix);
```

**Description** **ltoa** converts **val** to a string in base **radix** and places the result in **buf**.

**ltoa** returns **buf** as the result.

If **radix** is greater than 36, the result is undefined.

If **val** is negative and radix is 10, the string has a leading minus sign (-); for all other values of **radix**, **value** is considered unsigned and never has a leading minus sign.

**Portability** **ltoa** is an extension to the standard C library provided by *CrossWorks C*.

**See Also** **itoa** (page 299) **lltoa** (page 300) **ultoa** (page 309) **ultoa** (page 309) **utoa** (page 310)

---

## malloc

**Synopsis**

```
#include <stdlib.h>
void *malloc(size_t size);
```

**Description** **malloc** allocates space for an object whose size is specified by **size** and whose value is indeterminate.

**malloc** returns a null pointer if the space for the object cannot be allocated from free memory; if space for the object can be allocated, **malloc** returns a pointer to the start of the allocated space.

**Portability** **malloc** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## rand

**Synopsis**

```
#include <stdlib.h>
int rand(void);
```

**Description** **rand** computes a sequence of pseudo-random integers in the range 0 to **RAND\_MAX**. **rand** returns the computed pseudo-random integer.

**Portability** **rand** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

See Also **srand** (page 302) **RAND\_MAX** (page 302)

---

## **RAND\_MAX**

**Synopsis**

```
#include <stdlib.h>
#define RAND_MAX 32767
```

**Description** **RAND\_MAX** expands to an integer constant expression that is the maximum value returned by **rand**.

**Portability** **RAND\_MAX** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

See Also **rand** (page 301) **srand** (page 302)

---

## **realloc**

**Synopsis**

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

**Description** **realloc** deallocates the old object pointed to by **ptr** and returns a pointer to a new object that has the size specified by **size**. The contents of the new object is identical to that of the old object prior to deallocation, up to the lesser of the new and old sizes. Any bytes in the new object beyond the size of the old object have indeterminate values.

If **ptr** is a null pointer, **realloc** behaves like **malloc** for the specified size. If memory for the new object cannot be allocated, the old object is not deallocated and its value is unchanged.

**realloc** function returns a pointer to the new object (which may have the same value as a pointer to the old object), or a null pointer if the new object could not be allocated.

**Notes** If **ptr** does not match a pointer earlier returned by **calloc**, **malloc**, or **realloc**, or if the space has been deallocated by a call to **free** or **realloc**, the behaviour is undefined.

**Portability** **realloc** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## **srand**

**Synopsis**

```
#include <stdlib.h>
void srand(unsigned int seed);
```

**Description** **srand** uses the argument **seed** as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to **rand**. If **srand** is called with the same seed value, the same sequence of pseudo-random numbers is generated.

If **rand** is called before any calls to **srand** have been made, a sequence is generated as if **srand** is first called with a seed value of 1.

**Portability** **strand** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

**See Also** **rand** (page 301) **RAND\_MAX** (page 302)

---

## strtol

**Synopsis**

```
#include <stdlib.h>
double strtod(const char *nptr, char **endptr);
```

**Description** **strtod** converts the initial portion of the string pointed to by **nptr** to a **double** representation.

First, **strtod** decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by **isspace** (page 246)), a subject sequence resembling a floating-point constant, and a final string of one or more unrecognized characters, including the terminating null character of the input string. **strtod** then attempts to convert the subject sequence to a floating-point number, and return the result.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

The expected form of the subject sequence is an optional plus or minus sign followed by a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

**strtod** returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **HUGE\_VAL** is returned according to the sign of the value, if any, and the value of the macro **errno** (page 247) is stored in **errno** (page 247).

**Portability** **strtod** conforms to ISO/IEC 9899:1990 (C90).

---

## strtof

**Synopsis**

```
#include <stdlib.h>
float strtof(const char *nptr, char **endptr);
```

**Description** **strtof** converts the initial portion of the string pointed to by **nptr** to a **double** representation.

First, **strtof** decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by **isspace** (page 246)), a subject sequence resembling a floating-point constant, and a final string of one or more unrecognized characters, including the terminating null character of the input string. **strtof** then attempts to convert the subject sequence to a floating-point number, and return the result.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

The expected form of the subject sequence is an optional plus or minus sign followed by a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

**strtof** returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **HUGE\_VALF** is returned according to the sign of the value, if any, and the value of the macro **errno** (page 247) is stored in **errno** (page 247).

**Portability** **strtof** conforms to ISO/IEC 9899:1990 (C90).

---

## strtol

**Synopsis**

```
#include <stdlib.h>
long int strtol(const char *nptr, char **endptr, int base);
```

**Description** **strtol** converts the initial portion of the string pointed to by **nptr** to a **long int** representation.



First, **strtol** decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by **isspace** (page 246)), a subject sequence resembling an integer represented in some radix determined by the value of **base**, and a final string of one or more unrecognized characters, including the terminating null character of the input string. **strtol** then attempts to convert the subject sequence to an integer, and return the result.

When converting, no integer suffix (such as U, L, UL, LL, ULL) is allowed.

If the value of **base** is zero, the expected form of the subject sequence is an optional plus or minus sign followed by an integer constant.

If the value of **base** is between 2 and 36 (inclusive), the expected form of the subject sequence is an optional plus or minus sign followed by a sequence of letters and digits representing an integer with the radix specified by **base**. The letters from a (or A) through z (or Z) represent the values 10 through 35; only letters and digits whose ascribed values are less than that of **base** are permitted.

If the value of **base** is 16, the characters '0x' or '0X' may optionally precede the sequence of letters and digits, following the optional sign.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of **base** is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of **base** is between 2 and 36, it is used as the base for conversion.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

**strtol** returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **LONG\_MIN** (page 250) or **LONG\_MAX** (page 250) is returned according to the sign of the value, if any, and the value of the macro **errno** (page 247) is stored in **errno** (page 247).

#### Portability

**strtol** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## strtoll

**Synopsis**

```
#include <stdlib.h>
long long int strtoll(const char *nptr, char **endptr, int base);
```

**Description** **strtoll** converts the initial portion of the string pointed to by **nptr** to a **long int** representation.

First, **strtoll** decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by **isspace** (page 246)), a subject sequence resembling an integer represented in some radix determined by the value of **base**, and a final string of one or more unrecognized characters, including the terminating null character of the input string. **strtoll** then attempts to convert the subject sequence to an integer, and return the result.

When converting, no integer suffix (such as U, L, UL, LL, ULL) is allowed.

If the value of **base** is zero, the expected form of the subject sequence is an optional plus or minus sign followed by an integer constant.

If the value of **base** is between 2 and 36 (inclusive), the expected form of the subject sequence is an optional plus or minus sign followed by a sequence of letters and digits representing an integer with the radix specified by **base**. The letters from a (or A) through z (or Z) represent the values 10 through 35; only letters and digits whose ascribed values are less than that of **base** are permitted.

If the value of **base** is 16, the characters '0x' or '0X' may optionally precede the sequence of letters and digits, following the optional sign.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of **base** is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of **base** is between 2 and 36, it is used as the base for conversion.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

**strtoll** returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **LLONG\_MIN** (page 250) or **LLONG\_MAX** (page 250) is returned according to the sign of the value, if any, and the value of the macro **ERANGE** is stored in **errno** (page 247).

Portability **strtoll** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## strtoul

Synopsis 

```
#include <stdlib.h>
unsigned long int strtoul(const char *nptr, char **endptr, int base);
```

Description **strtoul** converts the initial portion of the string pointed to by **nptr** to a **long int** representation.

First, **strtoul** decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by **isspace** (page 246)), a subject sequence resembling an integer represented in some radix determined by the value of **base**, and a final string of one or more unrecognized characters, including the terminating null character of the input string. **strtoul** then attempts to convert the subject sequence to an integer, and return the result.

When converting, no integer suffix (such as U, L, UL, LL, ULL) is allowed.

If the value of **base** is zero, the expected form of the subject sequence is an optional plus or minus sign followed by an integer constant.

If the value of **base** is between 2 and 36 (inclusive), the expected form of the subject sequence is an optional plus or minus sign followed by a sequence of letters and digits representing an integer with the radix specified by **base**. The letters from a (or A) through z (or Z) represent the values 10 through 35; only letters and digits whose ascribed values are less than that of **base** are permitted.

If the value of **base** is 16, the characters '0x' or '0X' may optionally precede the sequence of letters and digits, following the optional sign.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of **base** is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of **base** is between 2 and 36, it is used as the base for conversion.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

**strtoul** returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **LONG\_MAX** (page 250) or **ULONG\_MAX** (page 252) is returned according to the sign of the value, if any, and the value of the macro **ERANGE** is stored in **errno** (page 247).

Portability

**strtoul** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## strtoull

Synopsis

```
#include <stdlib.h>
unsigned long long int strtoull(const char *nptr,
                               char **endptr,
                               int base);
```

Description

**strtoull** converts the initial portion of the string pointed to by **nptr** to a **long int** representation.

First, **strtoull** decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by **isspace** (page 246)), a subject sequence resembling an integer represented in some radix determined by the value of **base**, and a final string of one or more unrecognized characters, including the terminating null character of the input string. **strtoull** then attempts to convert the subject sequence to an integer, and return the result.

When converting, no integer suffix (such as U, L, UL, LL, ULL) is allowed.

If the value of **base** is zero, the expected form of the subject sequence is an optional plus or minus sign followed by an integer constant.

If the value of **base** is between 2 and 36 (inclusive), the expected form of the subject sequence is an optional plus or minus sign followed by a sequence of letters and digits representing an integer with the radix specified by **base**. The letters from a (or A) through z (or Z) represent the values 10 through 35; only letters and digits whose ascribed values are less than that of **base** are permitted.

If the value of **base** is 16, the characters '0x' or '0X' may optionally precede the sequence of letters and digits, following the optional sign.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of **base** is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of **base** is between 2 and 36, it is used as the base for conversion.

If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.

A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed, the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

**strtoull** returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **LLONG\_MAX** (page 250) or **ULLONG\_MAX** (page 252) is returned according to the sign of the value, if any, and the value of the macro **ERANGE** is stored in **errno** (page 247).

**Portability** **strtoull** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## ulltoa

**Synopsis**

```
#include <stdlib.h>
char *ulltoa(unsigned long long val, char *buf, int radix);
```

**Description** **ulltoa** converts **val** to a string in base **radix** and places the result in **buf**. **ulltoa** returns **buf** as the result.

If **radix** is greater than 36, the result is undefined.

**Portability** **ulltoa** is an extension to the standard C library provided by *CrossWorks C*.

**See Also** **itoa** (page 299) **ltoa** (page 301) **lltoa** (page 300) **ultra** (page 309) **utoa** (page 310)

---

## ultra

**Synopsis**

```
#include <stdlib.h>
char *ultra(unsigned long val, char *buf, int radix);
```

Description	<b>ultoa</b> converts <b>val</b> to a string in base <b>radix</b> and places the result in <b>buf</b> . <b>ultoa</b> returns <b>buf</b> as the result. If <b>radix</b> is greater than 36, the result is undefined.
Portability	<b>ultoa</b> is an extension to the standard C library provided by <i>CrossWorks C</i> .
See Also	<b>itoa</b> (page 299) <b>Itoa</b> (page 301) <b>ltoa</b> (page 300) <b>ultoa</b> (page 309) <b>utoa</b> (page 310)

## utoa

Synopsis	<pre>#include &lt;stdlib.h&gt; char *utoa(unsigned val, char *buf, int radix);</pre>
Description	<b>utoa</b> converts <b>val</b> to a string in base <b>radix</b> and places the result in <b>buf</b> . <b>utoa</b> returns <b>buf</b> as the result. If <b>radix</b> is greater than 36, the result is undefined.
Portability	<b>utoa</b> is an extension to the standard C library provided by <i>CrossWorks C</i> .
See Also	<b>itoa</b> (page 299) <b>Itoa</b> (page 301) <b>ltoa</b> (page 300) <b>ultoa</b> (page 309) <b>ultra</b> (page 309)

## <string.h> - String handling

The header file **<string.h>** defines functions that operate on arrays that are interpreted as null-terminated strings.

Various methods are used for determining the lengths of the arrays, but in all cases a **char \*** or **void \*** argument points to the initial (lowest addressed) character of the array. If an array is accessed beyond the end of an object, the behavior is undefined.

Where an argument declared as **size\_t n** specifies the length of an array for a function, **n** can have the value zero on a call to that function. Unless explicitly stated otherwise in the description of a particular function, pointer arguments must have valid values on a call with a zero size. On such a call, a function that locates a character finds no occurrence, a function that compares two character sequences returns zero, and a function that copies characters copies zero characters.

### Copying functions

<b>memcpy</b>	Copy memory
<b>memmove</b>	Safely copy overlapping memory
<b>strcpy</b>	Copy string

**strcpy** Copy string up to a maximum length

---

### Concatenation functions

---

**strcat** Convert string to **int**

---

**strncat** Convert string to **long**

---

### Comparison functions

---

**memcmp** Compare memory

---

**strcmp** Compare strings

---

**strncmp** Compare strings up to a maximum length

---

**strcoll** Collate strings

---

### Search functions

---

**memchr** Search memory for a character

---

**strchr** Find first occurrence of character within string

---

**strcspn** Compute size of string not prefixed by a set of characters

---

**strpbrk** Find first occurrence of characters within string

---

**strrchr** Find last occurrence of character within string

---

**strspn** Compute size of string prefixed by a set of characters

---

**strstr** Find first occurrence of a string within a string

---

**strtok** Break string into tokens

---

### Miscellaneous functions

---

**memset** Set memory to character

---

**strerror** Return string from error code

---

**strlen** Calculate length of string

---

---

## memchr

**Synopsis**

```
#include <string.h>
void *memchr(const void *s, int c, size_t n);
```

**Description** **memchr** locates the first occurrence of **c** (converted to an **unsigned char**) in the initial **n** characters (each interpreted as **unsigned char**) of the object pointed to by **s**. Unlike **strchr**, **memchr** does *not* terminate a search when a null character is found in the object pointed to by **s**.

**memchr** returns a pointer to the located character, or a null pointer if **c** does not occur in the object.

Portability **memchr** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

See also **strchr** (page 313)

---

## memcmp

Synopsis 

```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```

Description **memcmp** compares the first **n** characters of the object pointed to by **s1** to the first **n** characters of the object pointed to by **s2**. **memcmp** returns an integer greater than, equal to, or less than zero as the object pointed to by **s1** is greater than, equal to, or less than the object pointed to by **s2**.

Portability **memcmp** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## memcpy

Synopsis 

```
#include <string.h>
void *memcpy(void *s1, const void *s2, size_t n);
```

Description **memcpy** copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. The behaviour of **memcpy** is undefined if copying takes place between objects that overlap.

**memcpy** returns the value of **s1**.

Portability **memcpy** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## memmove

Synopsis 

```
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);
```

Description **memmove** copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1** ensuring that if **s1** and **s2** overlap, the copy works correctly. Copying takes place as if the **n** characters from the object pointed to by **s2** are first copied into a temporary array of **n** characters that does not overlap the objects pointed to by **s1** and **s2**, and then the **n** characters from the temporary array are copied into the object pointed to by **s1**.

**memmove** returns the value of **s1**.



Portability **memmove** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## memset

Synopsis 

```
#include <string.h>
void *memset(void *s, int c, size_t n);
```

Description **memset** copies the value of **c** (converted to an **unsigned char**) into each of the first **n** characters of the object pointed to by **s**.  
**memset** returns the value of **s**.

Portability **memset** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## strcat

Synopsis 

```
#include <string.h>
char *strcat(char *s1, const char *s2);
```

Description **strcat** appends a copy of the string pointed to by **s2** (including the terminating null character) to the end of the string pointed to by **s1**. The initial character of **s2** overwrites the null character at the end of **s1**. The behaviour of **strcat** is undefined if copying takes place between objects that overlap.  
**strcat** returns the value of **s1**.

Portability **strcat** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## strchr

Synopsis 

```
#include <string.h>
char *strchr(const char *s, int c);
```

Description **strchr** locates the first occurrence of **c** (converted to a **char**) in the string pointed to by **s**. The terminating null character is considered to be part of the string.  
**strchr** returns a pointer to the located character, or a null pointer if **c** does not occur in the string.

Portability **strchr** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

See also **memchr** (page 311)

---

## strcmp

Synopsis 

```
#include <string.h>
int strcmp(const char *s1, const char *s2);
```

- Description** **strcmp** compares the string pointed to by **s1** to the string pointed to by **s2**. **strcmp** returns an integer greater than, equal to, or less than zero if the string pointed to by **s1** is greater than, equal to, or less than the string pointed to by **s2**.
- Portability** **strcmp** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
- 

## strcoll

- Synopsis**

```
#include <string.h>
int strcoll(const char *s1, const char *s2);
```
- Description** **strcoll** compares the string pointed to by **s1** to the string pointed to by **s2**. **strcoll** returns an integer greater than, equal to, or less than zero if the string pointed to by **s1** is greater than, equal to, or less than the string pointed to by **s2**.
- strcoll** is not affected by the locale as *CrossWorks C* provides no locale capability.
- Portability** **strcoll** is provided for compaibility only and is not required in a freestanding implementation according to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
- 

## strcpy

- Synopsis**

```
#include <string.h>
char *strcpy(char *s1, const char *s2);
```
- Description** **strcpy** copies the string pointed to by **s2** (including the terminating null character) into the array pointed to by **s1**. The behaviour of **strcpy** is undefined if copying takes place between objects that overlap.
- strcpy** returns the value of **s1**.
- Portability** **strcpy** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).
- 

## strcspn

- Synopsis**

```
#include <string.h>
size_t strcspn(const char *s1, const char *s2);
```
- Description** **strcspn** computes the length of the maximum initial segment of the string pointed to by **s1** which consists entirely of characters not from the string pointed to by **s2**.
- strcspn** returns the length of the segment.
- Portability** **strcspn** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## strerror

**Synopsis**

```
#include <string.h>
char *strerror(int errnum);
```

**Description** **strerror** maps the number in **errnum** to a message string. Typically, the values for **errnum** come from **errno**, but **strerror** can map any value of type **int** to a message. **strerror** returns a pointer to the message string.

The program must not modify the returned message string. The message may be overwritten by a subsequent call to **strerror**.

**Portability** **strerror** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## strlen

**Synopsis**

```
#include <string.h>
size_t strlen(const char *s);
```

**Description** **strlen** returns the length of the string pointed to by **s**, that is the number of characters that precede the terminating null character.

**Portability** **strlen** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## strncat

**Synopsis**

```
#include <string.h>
char *strncat(char *s1, const char *s2, size_t n);
```

**Description** **strncat** appends not more than **n** characters from the array pointed to by **s2** to the end of the string pointed to by **s1**. A null character in **s1** and characters that follow it are not appended. The initial character of **s2** overwrites the null character at the end of **s1**. A terminating null character is always appended to the result. The behaviour of **strncat** is undefined if copying takes place between objects that overlap.

**strncat** returns the value of **s1**.

**Portability** **strncat** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## strncmp

**Synopsis**

```
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t n);
```

**Description** **strncmp** compares not more than **n** characters from the array pointed to by **s1** to the array pointed to by **s2**. Characters that follow a null character are not compared.

**strncmp** returns an integer greater than, equal to, or less than zero, if the possibly null-terminated array pointed to by **s1** is greater than, equal to, or less than the possibly null-terminated array pointed to by **s2**.

**Portability** **strncmp** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## strncpy

**Synopsis**

```
#include <string.h>
char *strncpy(char *s1, const char *s2, size_t n);
```

**Description** **strncpy** copies not more than **n** characters from the array pointed to by **s2** to the array pointed to by **s1**. Characters that follow a null character in **s1** are not copied. The behaviour of **strncpy** is undefined if copying takes place between objects that overlap. If the array pointed to by **s2** is a string that is shorter than **n** characters, null characters are appended to the copy in the array pointed to by **s1**, until **n** characters in all have been written.

**strncpy** returns the value of **s1**.

**Portability** **strncpy** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## strpbrk

**Synopsis**

```
#include <string.h>
char *strpbrk(const char *s1, const char *s2);
```

**Description** **strpbrk** locates the first occurrence in the string pointed to by **s1** of any character from the string pointed to by **s2**.

**strpbrk** returns a pointer to the character, or a null pointer if no character from **s2** occurs in **s1**.

**Portability** **strpbrk** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## strrchr

**Synopsis**

```
#include <string.h>
char *strrchr(const char *s, int c);
```

**Description** **strrchr** locates the last occurrence of **c** (converted to a **char**) in the string pointed to by **s**. The terminating null character is considered to be part of the string.

**strrchr** returns a pointer to the character, or a null pointer if **c** does not occur in the string.

**Portability** **strchr** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

**See also** **strchr** (page 313)

---

## strspn

**Synopsis**

```
#include <string.h>
size_t strspn(const char *s1, const char *s2);
```

**Description** **strspn** computes the length of the maximum initial segment of the string pointed to by **s1** which consists entirely of characters from the string pointed to by **s2**.

**strspn** returns the length of the segment.

**Portability** **strspn** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## strstr

**Synopsis**

```
#include <string.h>
char *strstr(const char *s1, const char *s2);
```

**Description** **strstr** locates the first occurrence in the string pointed to by **s1** of the sequence of characters (excluding the terminating null character) in the string pointed to by **s2**.

**strstr** returns a pointer to the located string, or a null pointer if the string is not found. If **s2** points to a string with zero length, **strstr** returns **s1**.

**Portability** **strstr** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

---

## strtok

**Synopsis**

```
#include <string.h>
char *strtok(char *s1, const char *s2);
```

**Description** A sequence of calls to **strtok** breaks the string pointed to by **s1** into a sequence of tokens, each of which is delimited by a character from the string pointed to by **s2**. The first call in the sequence has a non-null first argument; subsequent calls in the sequence have a null first argument. The separator string pointed to by **s2** may be different from call to call.

The first call in the sequence searches the string pointed to by **s1** for the first character that is not contained in the current separator string pointed to by **s2**. If no such character is found, then there are no tokens in the string pointed to by **s1** and **strtok** returns a null pointer. If such a character is found, it is the start of the first token.

**strtok** then searches from there for a character that is contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by **s1**, and subsequent searches for a token will return a null pointer. If such a character is found, it is overwritten by a null character, which terminates the current token. **strtok** saves a pointer to the following character, from which the next search for a token will start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

Portability

**strtok** conforms to ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99).

# Part V

## Command Line Tools

# Command Line Tools





The command line program **crossbuild** enables your software to be built without using **CrossStudio**. This tool can be used for production build purposes but isn't designed to be used for development. The tool works from a **crossstudio** project file (.hzip) and options that specify what is to be built.

```
crossbuild [options] project.hzip
```

You must specify a configuration to build in using the **-config** option.

```
crossbuild -config "V5T Thumb LE Release" arm.hzip
```

This example will build all projects in the solution contained in **arm.hzip** in the configuration **"V5T Thumb LE Release"**.

If you want to build a specific project in the solution then you can specify it using the **-project** option.

```
crossbuild -config "V5T Thumb LE Release" -project "libm" libc.hzip
```

This example will build the project **libm** contained in **libc.hzip** in the configuration **"V5T Thumb LE Release"**.

If your project file imports other project files (using the <import.> mechanism) then denoting projects requires you to specify the solution names as a comma separated list in brackets after the project name.

```
crossbuild -config "V5T Thumb LE Release" -project "libc(C Library)" arm.hzip
```

With this example **libc(C Library)** specifies the **libc** project in the **C Library** solution that has been imported by the project file **arm.hzip**.

If you want to build a specific solution that has been imported from other project files you can use the **-solution** option. This option takes the solution names as a comma separated list.

```
crossbuild -config "ARM Debug" -solution "ARM Targets,EB55" arm.hzp
```

With this example **ARM Targets,EB55** specifies the **EB55** solution imported by the **ARM Targets** solution which in turn was imported by the project file **arm.hzp**.

You can do a batch build using the **-batch** option.

```
crossbuild -config "ARM Debug" -batch libc.hzp
```

With this example the projects in **libc.hzp** which are marked to batch build in the configuration **"ARM Debug"** will be built.

By default a **make** style build will be done i.e. the dates of input files are checked against the dates of output files and the build is avoided if the output file is up to date. You can force a complete build by using the **-rebuild** option. Alternatively you can remove all output files using the **-clean** option.

You can see the commands that are being used in the build if you use the **-echo** option and you can also see why commands are being executed using the **-verbose** option. You can see what commands will be executed without executing them using the **-show** option.

### CrossBuild Options

-batch	Do a batch build.
-config 'name'	Specify the configuration to build in. If the 'name' configuration can't be found crossbuild will list the set of configurations that are available.
-clean	Remove all the output files of the build process.
-D macro=value	Define a macro value for the build process.
-echo	Show the command lines as they are executed.
-project 'name'	Specify the name of the project to build. If crossbuild can't find the specified project then a list of project names is shown.
-rebuild	Always execute the build commands.
-show	Show the command lines but don't execute them.
-solution 'name'	Specify the name of the solution to build. If crossbuild can't find the specified solution then a list of solution names is shown.
-verbose	Show build information.

## CrossLoad

**CrossLoad** is a command line program that allows you to download and verify applications without using **CrossStudio**. This tool can be used for production purposes but is not designed to be used for development.

### Usage

crossload [options]  
[files...]

#### Options

<b>-target</b> <i>target</i>	Specify the target interface to use. Use the <b>-listtargets</b> option to display the list of supported target interfaces.
<b>-listtargets</b>	List all of the supported target interfaces.
<b>-solution</b> <i>file</i>	Specify the CrossWorks solution file to use.
<b>-project</b> <i>name</i>	Specify the name of the project to use.
<b>-config</b> <i>configuration</i>	Specify the build configuration to use.
<b>-filetype</b> <i>filetype</i>	Specify the type of the file to download. By default <b>CrossLoad</b> will attempt to detect the file type, you should use this option if <b>CrossLoad</b> cannot determine the file type or to override the detection and force the type. Use the <b>-listfiletypes</b> option to display the list of supported file types.
<b>-listfiletypes</b>	List all of the supported file types.
<b>-setprop</b> <i>property=value</i>	Set the target property <i>property</i> to <i>value</i> .
<b>-listprops</b>	List the target properties of the target specified by the <b>-target</b> option.

## Usage

-noverify	Do not carry out verification of download.
-nodownload	Do not carry out download, just verify.
-nodisconnect	Do not disconnect the target interface when finished.
-help	Display the command line options.
-verbose	Produce verbose output.
-quiet	Do not output any progress messages.

In order to carry out a download or verify **CrossLoad** needs to know what target interface to use. The supported target interfaces vary between systems, to produce a list of the currently supported target interfaces use the *-listtargets* option.

```
crossload -listtargets
```

This command will produce a list of target interface names and descriptions:

```
usb          USB CrossConnect
parport     Parallel Port Interface
sim         Simulator
```

Use the *-target* option followed by the target interface name to specify which target interface to use:

```
crossload -target usb ...
```

**CrossLoad** is normally used to download and/or verify projects created and built with **CrossStudio**. To do this you need to specify the target interface you want to use, the **CrossStudio** solution file, the project name and the build configuration. The following command line will download and verify the debug version of the project *MyProject* contained within the *MySolution.hzp* solution file using a USB CrossConnect:

```
crossload -target usb -solution MySolution.hzp -project MyProject -config Debug
```

In some cases it is useful to download a program that might not have been created using **CrossStudio** using the settings from an existing **CrossStudio** project. You might want to do this if your existing project describes specific loaders or scripts that are required in order to download the application. To do this you simply need to add the name of the file you want to download to the command line. For example the following command line will download the HEX file *ExternalApp.hex* using the release settings of the project *MyProject* using a USB CrossConnect:

```
crossload -target usb -solution MySolution.hzp -project MyProject -config Release
ExternalApp.hex
```

CrossLoad is able to download and verify a range of file types. The supported file types vary between systems, to display a list of the file types supported by **CrossLoad** use the *-listfiletypes* option:

```
crossload -listfiletypes
```

This command will produce a list of the supported file types, for example:

```
hzz                CrossStudio Executable File
bin                Binary File
ihex               Intel Hex File
hex               Hex File
tihex             TI Hex File
srec               Motorola S-Record File
```

**CrossLoad** will attempt to determine the type of any load file given to it, if it cannot do this you may specify the file type using the **-filetype** option:

```
crossload -target usb -solution MySolution.hzp -project MyProject -config Release
ExternalApp.txt -filetype tihex
```

It is possible with some targets to carry out a download without the need to specify a **CrossStudio** project. In this case all you need to specify is the target interface and the load file. For example the following command line will download **myapp.s19** using a USB CrossConnect:

```
crossload -target usb myapp.s19
```

Each target interface has a range of configurable properties that allow you to customize the default behaviour. To produce a list of the target properties and their current value use the **-listprops** option:

```
crossload -target parport -listprops
```

This command will produce a list of the *parport* target interfaces properties, a description of what the properties are and their current value:

```
Name:           JTAG Clock Divider
Description:    The amount to divide the JTAG clock frequency.
Value          : 1

Name:           Parallel Port
Description:    The parallel port connection to use to connect to target.
Value          : Lpt1

Name:           Parallel Port Sharing
Description:    Specifies whether sharing of the parallel port with other
device drivers or programs is permitted.
Value          : No
```

You can modify a target property using the **-setprop** option. For example the following command line would set the parallel port interfaced used to **lpt2**:

```
crossload -target parport -setprop "Parallel Port"="Ltp2" ...
```



## mkhdr - C/C++ header file generator

The command line program **mkhdr** can generate a C/C++ header file from a crossworks memory map file.

For each register definition in the memory map file a corresponding `#define` is generated in the header file. The `#define` is named the same as the register name and is defined as a volatile pointer to the address. The type of the pointer is derived from the size of the register. A 4 byte register will generate an unsigned long pointer. A 2 byte register will generate an unsigned short pointer. A 1 byte register will generate an unsigned char pointer.

If a register definition in the memory map file has bitfields then `#defines` are generated for each bitfield. Each bitfield will have two `#defines` generated, one representing the mask and one defining the start bit position. The bitfield `#define` names are formed by prepending the register name to the bitfield name. The mask definition has `_MASK` appended to it and the start definition has `_BIT` appended to it.

For example given the following definitions in the the file `memorymap.xml`.

```
<RegisterGroup start="0xFFFFF000" name="AIC" >
  <Register start="+0x00" size="4" name="AIC_SMR0">
    <BitField size="3" name="PRIOR" start="0" />
    <BitField size="2" name="SRCTYPE" start="5" />
  </Register>
  ...
```

The command line

```
mkhdr memorymap.xml memorymap.h
```

Will generate the following definitions in the file `memorymap.h`.

```
#define AIC_SMR0 (*(volatile unsigned long *)0xFFFFF000)
#define AIC_SMR0_PRIOR_MASK 0x7
#define AIC_SMR0_PRIOR_BIT 0
#define AIC_SMR0_SRCTYPE_MASK 0x60
#define AIC_SMR0_SRCTYPE_BIT 5
```

These definitions can be used in the following way in a C/C++ program.

Reading from a register.

```
unsigned r = AIC_SMR0;
```

Writing to a register.

```
AIC_SMR0 = (priority << AIC_SMR0_PRIOR_BIT) | (srctype <<
AIC_SMR0_SRCTYPE_BIT);
```

Reading a bitfield.

```
unsigned srctype = (AIC_SMR0 & AIC_SMR0_SRCTYPE_MASK) >>
AIC_SMR0_SRCTYPE_BIT;
```

Writing a bitfield.

```
AIC_SMR0 = (AIC_SMR0 & ~AIC_SMR0_SRCTYPE_MASK) | ((srctype &
AIC_SMR0_SRCTYPE_MASK) << AIC_SMR0_SRCTYPE_BIT);
```

Usage:

```
mkhdr inputfile outputfile [targetname] [option]
```

Element	Description
inputfile	The name of the source CrossWorks <b>Memory map editor</b> (page 89) file.
outputfile	The name of the file to generated.
-regbaseoffsets	Include offsets of registers from the peripheral base.
-nobitfields	Do not generate any definitions for bitfields.



## mkpkg - package creator

To create a package the program **mkpkg** can be used. The set of files to put into the package should be in the desired location in the  $\$(StudioDir)$  directory. The **mkpkg** command should be run with  $\$(StudioDir)$  as the working directory and all files to go into the package must be referred to using relative paths. A package must have a package description file that is placed in the  $\$(StudioDir)/packages$  directory. The package description file name must end with **\_package.xml**. If a package is to create entries in the new project wizard then it must have a file name **project\_templates.xml**.

For example a package **MyNewTarget** would supply the following files.

- A **Project Templates file format** (page 341) file called `targets/MyNewTarget/project_templates.xml`.
- The  $\$(StudioDir)$  relative files that define the functionality of the package.
- A package description file called `packages/MyNewTarget_package.xml`.

The package file **MyNewTarget.hzq** would be created using the following command line

```
mkpkg -c packages/MyNewTarget.hzq targets/MyNewTarget/project_templates.xml
.. packages/MyNewTarget_package.xml
```

You can list the contents of the package using the **-t** option

```
mkpkg -t packages/MyNewTarget.hzq
```

You can remove an entry from a package using the **-d** option

```
mkpkg -d packages/MyNewTarget.hzq -d fileToRemove
```

You can add or replace a file into an existing package using the **-r** option

```
mkpkt -r packages/MyNewTarget.hzq -r fileToAddOrReplace
```

You can automate the package creation process using a **Combining** project type.

- Using the new project wizard create a combining project in the directory `$(StudioDir)`.
- Set the **Output File Path** property to be `$(StudioDir)/packages/mypackage.hzq`
- Set the **Combine command** property to `$(StudioDir)/bin/mkpkg -c $(CombiningOutputFilePath) $(CombiningRelInputPaths)`
- Add the files you want to go into the package into the project using the project explorer.
- Right click on the project node in the project explorer and build.

When a package is installed the files in the package are copied into the desired `$(StudioDir)` relative locations. When a file is copied into the `$(StudioDir)/packages` directory and it's filename ends with **\_package.xml** the file `$(StudioDir)/packages/installed_packages.xml` is updated with an entry

```
<include filename="MyNewTarget_package.xml">
```

During development of a package you can manually edit this file. The same applies to the file `$(StudioDir)/targets/project_templates.xml` which will contain a reference to your `project_templates.xml` file.

Usage:

```
mkpkg [options] package file1 file2 ..
```

### Options

-c	Create a new package.
-d	Remove files from a package.
-r	Replace files in a package.
-t	List the contents of a package.
-v	verbose.
-V	show verbose.









# Part VI

---

## Appendices

# Appendices





## Memory Map file format

CrossStudio memory map files are structured using XML syntax to enable simple construction and parsing.

The first entry of the project file defines the XML document type which is used to validate the file format.

```
<!DOCTYPE Board_Memory_Definition_File>
```

The next entry is the Root element; there can only be one Root element in a memory map file.

```
<Root name="My Board" >
```

A Root element has a **name** attribute - every element in a memory map file has a name attributes. Names should be unique within a hierarchy level. Within a Root element there are MemorySegment elements which represent memory regions within the memory map.

```
<Root name="My Board" >  
  <MemorySegment name="Flash" start="0x1000" size="0x200" access="ReadOnly"  
>
```

MemorySegment elements have the following attributes.

- **start** The start address of the memory segment. A 0x prefixed hexadecimal number.
- **size** The size of the memory segment. A 0x prefixed hexadecimal number.

- **access** The permissible access types of the memory segment. One of **ReadOnly**, **Read/Write**, **WriteOnly**, **None**.
- **address\_symbol** A symbolic name for the start address.
- **size\_symbol** A symbolic name for the size.

A MemorySegment can contain Register and RegisterGroup elements. RegisterGroup elements are used to group sets of registers. Register elements are used to define peripheral registers.

```
<Root name="My Board" >
  <MemorySegment name="System" start="0x2000" size="0x200" >
    <RegisterGroup name="Peripheral1" start="0x2100" size="0x10" >
      <Register name="Register1" start="+0x8" size="4" >
```

Register group elements have the same attributes as MemorySegment elements. Register elements have the following attributes.

- **name** Register names should be valid C/C++ identifier names i.e. alphanumeric and underscores but not starting with a number.
- **start** The start address of the memory segment. Either a 0x prefixed hexadecimal number or a + prefixed offset from the enclosing element's start address.
- **size** The size of the register in bytes - either 1, 2 or 4.
- **access** The same as the MemorySegment element.
- **address\_symbol** The same as the MemorySegment element.

A Register element can contain BitField elements that represent bits within a peripheral register.

```
<Root name="My Board" >
  <MemorySegment name="System" start="0x2000" size="0x200" >
    <RegisterGroup name="Peripheral1" start="0x2100" size="0x10" >
      <Register name="Register1" start="+0x8" size="4" >
        <BitField name="Bits_0_to_3" start="0" size="4" />
```

BitField elements have the following attributes.

- **name** BitFields names should be valid C/C++ identifier names i.e. alphanumeric and underscores but not starting with a number.
- **start** The starting bit position 0-31.
- **size** The number of bits 1-31.

You can refer to other memory map files using the Import element.

```
<Root name="My Board" >
  <MemorySegment name="External SRAM" start="0x3000" size="0x200"
```

```
access="ReadOnly" >
  <Import filename="$(StudioDir)/targets/Manufacturer1/Processor1.xml" />
```

The **filename** attribute is an absolute filename which is macro expanded using CrossWorks system macros. An imported memory map file must be an XML file (without a DOCTYPE). The Processor element can be used to group several MemorySegment elements for this purpose.

```
<Processor name="Processor1" >
  <MemorySegment name="Internal SRAM" start="0x1000" size="0x200"
access="ReadOnly" >
  <MemorySegment name="System" start="0x2000" size="0x200" >
```

## Section Placement file format

CrossStudio section placement files are structured using XML syntax to enable simple construction and parsing.

The first entry of the project file defines the XML document type which is used to validate the file format.

```
<!DOCTYPE Linker_Placement_File>
```

The next entry is the Root element; there can only be one Root element in a memory map file.

```
<Root name="Flash Placement" >
```

A Root element has a **name** attribute - every element in a section placement file has a name attribute. Names should be unique within a hierarchy level. Within a Root element there are MemorySegment elements. These correspond to memory regions that are defined in a memory map file that will be used in conjunction with the section placement file when linking a program.

```
<Root name="Flash Placement" >
  <MemorySegment name="FLASH" >
```

A MemorySegment contains ProgramSection elements which represent program sections created by the C/C++ compiler and assembler. The order of ProgramSection elements within a MemorySegment element represents the order in which the sections will be placed when linking a program. The first ProgramSection will be placed first and the last ProgramSection will be placed last.

```
<Root name="My Board" >
  <MemorySegment name="FLASH" >
    <ProgramSection name=".text" ... >
```

ProgramSection elements have the following attributes.

- **alignment** The required alignment of the program section - a decimal number.
- **inputsections** An expression describing the input sections to be placed in this section. This is typically the name of the section enclosed in `*(.name.name.*)`
- **load** If "Yes" then the section is loaded. If "No" then the section isn't loaded.
- **runin** This species the section to copy this section to.
- **start** The optional start address of the program section - a 0x prefixed hexadecimal number.
- **size** The optional size of the program section in bytes - a 0x prefixed hexadecimal number.

## Project file format

CrossStudio project files are held in text files with the .hzip extension. We anticipate that you may want to edit project files and perhaps generate them so they are structured using XML syntax to enable simple construction and parsing.

The first entry of the project file defines the XML document type which is used to validate the file format.

```
<!DOCTYPE CrossStudio_Project_File>
```

The next entry is the solution element; there can only be one solution element in a project file. This specifies the name of solution displayed in the project explorer and also has a version attribute which defines the file format version of the project file. Solutions can contain projects, projects can contain folder and /files, and folders can contain folder and files. This hierarchy is reflected in the XML nesting, for example:

```
<solution version="1" Name="solutionname">
  <project Name="projectname">
    <file Name="filename"/>
    <folder Name="foldername">
      <file Name="filename2"/>
    </folder>
  </project>
</solution>
```

Note that each entry has a Name attribute. Names of project elements must be unique to the solution, names of folder elements must be unique to the project however names of files do not need to be unique.

Each file element must have a file\_name attribute that is unique to the project. Ideally the file\_name is a file path relative to the project (or solution directory) but you can also specify a full file path if you want to. File paths are case sensitive and use / as the directory separator. They may contain macro instantiations so you cannot have file paths containing the \$ character. For example

```
<file file_name="$(StudioDir)/source/crt0.s" Name="crt0.s" />
```

will be expanded using the value of the \$(StudioDir) when the file is referenced from CrossStudio.

Project properties are held in configuration elements with the Name attribute of the configuration element corresponding to the configuration name e.g. "Debug". At a given project level (solution, project, folder) there can only be one named configuration element i.e. all properties defined for a configuration are in single configuration element.

```
<project Name="projectname">
  ...
  <configuration project_type="Library" Name="Common" />
  <configuration Name="Release" build_debug_information="No" />
  ...
</project>
```

You can link projects together using the import element.

```
<import file_name="target/libc.hzp" />
```

## Project Templates file format

The CrossStudio New Project Dialog works from a file called project\_templates.xml which is held in the targets subdirectory of the CrossStudio installation directory. We anticipate that you may want to add your own new project types so they are structured using XML syntax to enable simple construction and parsing.

The first entry of the project file defines the XML document type which is used to validate the file format.

```
<!DOCTYPE Project_Templates_File>
```

The next entry is the `projects` element; which is used to group a set of new project entries into an XML hierarchy.

```
<projects>
  <project...
</projects>
```

Each project entry has a `project` element that contains the class of the project (attribute `caption`), the name of the project (attribute `name`), its type (attribute `type`) and a description (attribute `description`).

```
<project caption="ARM Evaluator7T" name="Executable" description="An
executable for an ARM Evaluator7T." type="Executable"/>
```

The project type can be one of

- "Executable" - a fully linked executable.
- "Library" - a static library.
- "Object file" - an object file.
- "Staging" - a staging project.
- "Combining" - a combining project.
- "Externally Built Executable" - an externally built executable.

The configurations that are to be created for the project are defined using the `configuration` element. The configuration element must have a `name` attribute.

```
<configuration name="ARM RAM Release"/>
```

The property values to be created for the project are defined using the `property` element. If you have a defined value then you can specify this using the `value` attribute and optionally set the property in a defined `configuration`.

```
<property name="target_reset_script" configuration="RAM"
value="Evaluator7T_ResetWithRamAtZero()"/>
```

Alternatively you can include a property that will be shown to the user who can supply a value as part of the new project process.

```
<property name="linker_output_format"/>
```

The folders to be created are defined using the `folder` element. The folder element must have a `name` attribute and can also have a `filter` attribute.

```
<folder name="Source Files" filter="c;cpp;cxx;cc;h;s;asm;inc"/>
```

The files to be in the project are specified using the `file` element. You can use build system **Project macros** (page 54) to specify files that are in the CrossStudio installation directory. Files will be copied to the project directory or just left as references based on the value of the `expand` attribute.

```
<file name="$(StudioDir)/source/crt0.s" expand="no"/>
```

You can define the set of configurations that can be referred to in the the top-level `configurations` element.

```
<configurations>
  <configuration...
</configurations>
```

This contains the set of all configurations that can be created when a project is created. Each configuration is defined using a `configuration` element which can define the property values for that configuration.

```
<configuration name="Debug">
  <property name="build_debug_information" value="Yes">
```





# Property Reference

## Build Properties

### Build Options

These properties are applicable to a range of project types.

Property	Type	Save Name	Description
Build Quietly	Boolean	build_quietly	Suppress the display of the startup banners and information messages.
Enable Unused Symbol Removal	Boolean	build_remove_unused_symbols	If this option is set then any unreferenced symbols will be removed from your program.

Property	Type	Save Name	Description
Exclude From Build	Boolean	build_exclude_from_build	Specifies whether or not to exclude the project/file from the build.

---

Property	Type	Save Name	Description
----------	------	-----------	-------------

File Type	Enumeration	file_type	Use this property to change the file type of the selected file. If this property isn't set then the file extension is used to determine the file type. In the case of an XML file the document type is used. The options are <b>C</b> (compile using a C compiler), <b>C++</b> (compile using a C++ compiler), <b>Basic</b> (compile using a BASIC compiler), <b>Assembly</b> (assemble using an assembler), <b>Section Placement</b> (an XML linker section placement file), <b>Memory Map</b> (an XML memory map file), <b>Target</b>
-----------	-------------	-----------	---

Property	Type	Save Name	Description
Include Debug Information	Boolean	build_debug_information	Specifies whether symbolic debug information is generated.
Macros	String List	macros	Defines macro values that are used for filename generation. Each macro is defined as <b>name=value</b> .
Intermediate Directory	String	build_intermediate_directory	Specifies a relative path from the project directory to the intermediate file directory. This property will have macro expansion applied to it.

Property	Type	Save Name	Description
Output Directory	String	build_output_directory	Specifies a relative path from the project directory to the output file directory. This property will have macro expansion applied to it.
Project Directory	String	project_directory	Specifies the project directory. This can be either relative to the solution directory (recommended) or can be an absolute directory.
Project Type	Enumeration	project_type	Specifies the type of project to build. The options are <b>Executable</b> , <b>Library</b> , <b>Object file</b> , <b>Staging</b> , <b>Combining</b> , and <b>Externally Built Executable</b> .

Property	Type	Save Name	Description
Suppress Warnings	Boolean	build_suppress_warnings	Specifies whether the display of warning messages should be suppressed.
Treat Warnings as Errors	Boolean	build_treat_warnings_as_errors	Specifies whether warning messages should be treated as errors.

## Compilation Properties

These properties are applicable to C and assembly code source files that are in a **Executable**, **Library** or **Object File** project type.

## Assembler and Compiler Options

Property Name	Type	Save Name	Description
Additional Assembler Options	String List	asm_additional_options	Additional command line options to be supplied to the assembler.
Additional Compiler Options	String List	c_additional_options	Additional command line options to be supplied to the C/C++ compiler.

Property Name	Type	Save Name	Description
ARM Architecture	Enumeration	arm_architecture	Specifies the version of the ARM or Thumb instruction set to generate code for. The options are <b>v4T</b> , <b>v5TE</b> and <b>v7M</b> . Use <b>v4T</b> for ARM7TDMI, ARM720T and ARM920T based processors. Use <b>v5TE</b> for XScale and ARM9E based processors. Use <b>v7M</b> for Cortex-M3 based processors.



Property Name	Type	Save Name	Description
ARM Core Type	Enumeration	arm_core_type	Specifies the version of the ARM processor core to generate code for. The options are <b>ARM7TDMI</b> , <b>ARM7TDMI-S</b> , <b>ARM720T</b> , <b>ARM920T</b> , <b>ARM926EJ-S</b> , <b>Cortex-M3</b> and <b>XScale</b> . This option is currently not used for code generation.

Property Name	Type	Save Name	Description
ARM Floating Point Format	Enumeration	arm_fp_format	Specifies the ARM floating point format to generate code for. The options are <b>Software VFP</b> and <b>Software FPA</b> . The C libraries supplied with CrossWorks for ARM are compiled for <b>Software VFP</b> .
ARM/Thumb interworking	Boolean	arm_interwork	Specifies that the ARM code generated can be called either from ARM or Thumb code. Setting this property to "No" may result in smaller code sizes when compiling for architecture v4T.

Property Name	Type	Save Name	Description
Endian	Enumeration	arm_endian	Specifies the endianness to build for. The options are <b>Little</b> and <b>Big</b> . Note that the value of this property at project level will be used to automatically set the <b>Endian</b> target property when a project is downloaded or attached to.
Enforce ANSI Checking	Boolean	c_enforce_ansi_checking	Enable additional checking to ensure C programs conform to the ISO C99 standard and C++ programs conform to the 1998 ISO C++ standard.

Property Name	Type	Save Name	Description
Generate Static call_via_rX	Boolean	arm_module_call_vias	Works around a problem in the C/C++ compiler of the GCC when using long calls from Thumb code.
Instruction Set	Enumeration	arm_instruction_set	Specifies the instruction set the compiler should generate code for. The options are <b>ARM</b> or <b>Thumb</b> .
Keep Assembly	Boolean	arm_keep_assembly	When set the assembly code generated by the C/C++ compiler of the GCC are kept.

Property Name	Type	Save Name	Description
Long Calls	Boolean	arm_long_calls	Specifies whether function calls are made using 32-bit absolute addresses. Set this if you get relocation errors from the linker.
Object File Name	String	build_object_file_name	Specifies the name of the object file produced by the compiler/assembler. This property will have macro expansion applied to it.

Property Name	Type	Save Name	Description
Optimization Level	Enumeration	gcc_optimization_level	Specifies the optimization level to use for compilation. The options are <b>None</b> , <b>Level x</b> and <b>Optimize For Size</b> . Use <b>None</b> when developing and debugging code. Use <b>Level 1</b> for release builds. Don't use any of the other options.

## Preprocessor Options

Property	Type	Save Name	Description
Ignore Includes	Boolean	c_ignore_includes	If set to <b>Yes</b> , the <b>System Include Directories</b> and <b>User Include Directories</b> properties are ignored.
Preprocessor Definitions	String List	c_preprocessor_definitions	Specifies one or more preprocessor definitions.
Preprocessor Undefinitions	String List	c_preprocessor_undefinitions	Specifies one or more preprocessor undefinitions.
System Include Directories	String List	c_system_include_directories	Specifies the system include path. This property will have macro expansion applied to it.

Property	Type	Save Name	Description
Undefine All Preprocessor Definitions	Boolean	c_undefine_all_preprocessor_definitions	If set to <b>Yes</b> , no standard preprocessor or definitions will be defined.
User Include Directories	String List	c_user_include_directories	Specifies the user include path. This property will have macro expansion applied to it.



## Section Options

Property	Type	Save Name	Description
Code Section Name	String	default_code_section	Specifies the default section name to use for the program code section.
Constant Section Name	String	default_const_section	Specifies the default section name to use for the read-only constant section.
Data Section Name	String	default_data_section	Specifies the default section name to use for the initialised, writable data section.
Vector Section Name	String	default_vector_section	Specifies the default section name to use for the interrupt vector section.

Property	Type	Save Name	Description
Zeroed Section Name	String	default_zeroed_section	Specifies the default section name to use for the zero-initialised, writable data section.

---

## External Build Project Properties

These properties are applicable to **Externally Built Executable** project types.

## External Build Options

Property	Type	Save Name	Description
Build Command	String	external_build_command	The command line that will build the executable. The first entry must be the full path to an executable program.
Clean Command	String	external_clean_command	The command line that will clean the executable. The first entry must be the full path to an executable program.
Executable File	String	external_build_file_name	The name of the externally built executable file. This property will have macro expansion applied to it.

Property	Type	Save Name	Description
Load Address	String	external_load_address	The address to load the file at. This is required if the load address isn't contained in the executable file - for example a binary file.
Load File Type	Enumeration	external_load_file_type	The type of the executable file. The options are <b>detect</b> (based upon the file extension of the load file), <b>elf</b> (an ELF executable), <b>bin</b> (a plain binary file), <b>ihex</b> (an Intel hex file), <b>hex</b> (a hex file), <b>tihex</b> (a TI hex file), and <b>srec</b> (a Motorola S-Record file).

Property	Type	Save Name	Description
Start Address	String	external_start_address	The address to start execution from. This defaults to be the <b>Load Address</b> if it isn't defined.

## Folder Properties

### Folder Options

These properties are applicable to project folders.

Property	Type	Save Name	Description
Filter	String List	filter	A list of file extensions that are matched when a file is added to the project.

# Library Properties

## Library Options

These properties are applicable to **Library** project types.

Property	Type	Save Name	Description
Library File Name	String	build_output_file_name	Specifies the name of the output file produced by the librarian. This property will have macro expansion applied to it. Note that the save name is the same as the Executable File Name property. If this property is used at the solution node level it will be applied to both Library and Executable project types within the solution.

# Linker Properties

These properties are applicable to **Executable** project types.

## Printf/Scanf Options

Property	Type	Save Name	Description
Printf Floating Point Supported	Boolean	linker_printf_f_p_enabled	Specifies whether the version of the printf that supports floating point numbers should be linked into the application.
Printf Integer Support	Enumeration	linker_printf_mt_level	Specifies the largest integer type supported by the printf function group. Options are <b>int</b> (char, short and integer support), <b>long</b> (char, short, integer and long support), <b>long long</b> (char, short, integer, long and long long support).



Property	Type	Save Name	Description
Printf Width/Precision Supported	Boolean	linker_printf_width_precision_supported	Enables support for width and precision in the printf function group.
Scanf Classes Supported	Boolean	linker_scanf_character_group_matching_enabled	Enables support for %[...] and %^[...] character class matching in the scanf functions.
Scanf Floating Point Supported	Boolean	linker_scanf_fp_enabled	Specifies whether the version of the scanf that supports floating point numbers should be linked into the application.

Property	Type	Save Name	Description
Scanf Integer Support	Enumeration	linker_scanf_fmt_level	Specifies the largest integer type supported by the scanf function group. Options are <b>int</b> (char, short and integer support), <b>long</b> (char, short, integer and long support), <b>long long</b> (char, short, integer, long and long long support)

## Linker Options

Property	Type	Save Name	Description
Additional Input Files	String List	linker_additional_files	Additional object and library files to be supplied to the linker. This property will have macro expansion applied to it.
Additional Linker Options	String List	linker_additional_options	Additional command line options to be supplied to the linker.

Property	Type	Save Name	Description
Additional Output Format	Enumeration	linker_output_format	Specifies the format of an additional output file to be generated. Options are <b>None</b> (no additional output file), <b>bin</b> (a binary output file), <b>hex</b> (an Intel hex output file), <b>srec</b> (a Motorola S-Record output file).
Check For Memory Segment Overflow	Boolean	arm_library_check_memory_segment_overflow	Specifies that the linker should check whether program sections fit into the memory segments they have been placed in.

Property	Type	Save Name	Description
Entry Point	String	gcc_entry_point	Specifies the entry point of the program. This may be a symbol or an absolute address.
Executable File Name	String	build_output_file_name	Specifies the name of the output file produced by the linker. This property will have macro expansion applied to it. Note that the save name is the same as the Library File Name property. If this property is used at the solution node level it will be applied to both Library and Executable project types within the solution.

Property	Type	Save Name	Description
Generate Map File	Boolean	linker_map_file	Specifies whether or not a linker map file is generated.
Heap Size	Integer	arm_linker_heap_size	Specifies the heap size in bytes to be used by the application. You must define this if your program uses malloc.
Include Standard Libraries	Boolean	link_include_standard_libraries	Specifies whether the standard libraries should be linked into the application.
Include Startup Code	Boolean	link_include_startup_code	Specifies whether the standard C startup code is linked into the application.

Property	Type	Save Name	Description
Keep Symbols	String List	linker_keep_symbols	Specifies a list of symbols that should always be included by the linker in the output file even if those symbols aren't directly used by the application.
Library Instruction Set	Enumeration	arm_library_instruction_set	Specifies the library variant the linker should use. The options are <i>ARM</i> or <i>Thumb</i> .

Property	Type	Save Name	Description
Memory Map File	String	linker_memory_map_file	The name of the file containing the memory map description. This property will have macro expansion applied to it. Note that a memory map file in the project will be used in preference to this setting.
Post Build Command	String	linker_post_build_command	Specifies a command to run after the link command has executed. This must be the full path to an executable program. This property will have macro expansion applied to it.



Property	Type	Save Name	Description
Section Placement File	String	linker_section_placement_file	The name of the file containing the section placement description. This property will have macro expansion applied to it. Note that a section placement file in the project will be used in preference to this setting.
Stack Size (Abort Mode)	Integer	arm_linker_stack_abt_size	Specifies the size of the <b>Abort</b> mode stack in bytes.
Stack Size (FIQ Mode)	Integer	arm_linker_stack_fiq_size	Specifies the size of the <b>FIQ</b> mode stack in bytes.
Stack Size (IRQ Mode)	Integer	arm_linker_stack_irq_size	Specifies the size of the <b>IRQ</b> mode stack in bytes.

Property	Type	Save Name	Description
Stack Size (Supervisor Mode)	Integer	arm_linker_stack_svc_size	Specifies the size of the <b>Supervisor</b> mode stack in bytes.
Stack Size (Undefined Mode)	Integer	arm_linker_stack_und_size	Specifies the size of the <b>Undefined</b> mode stack in bytes.
Stack Size (User/System Mode)	Integer	arm_linker_stack_size	Specifies the size of the <b>User/System</b> mode stack in bytes.
Treat Linker Warnings as Errors	Boolean	arm_linker_treat_warnings_as_errors	If set then warnings produced by the linker are treated as errors.
Use GCC Libraries	Boolean	arm_use_gcc_libraries	Use GCC floating point, exception and rtti libraries.
Use Multi Threaded Libraries	Boolean	link_use_multithreaded_libraries	Specifies that multi-threaded (re-entrant) versions of the libraries should be linked in.

# Staging And Combining Project Properties

## Staging Options

These properties are applicable to **Staging** project types.

Property	Type	Save Name	Description
Output File Path	String	stage_output_filepath	Specifies the name the file will be copied to. This property will have macro expansion applied to it.
Package File Path	String	stage_package_filepath	Specifies the package file that the file will be added to. This property will have macro expansion applied to it.

Property	Type	Save Name	Description
Package Member Name	String	stage_package_member_name	Specifies the name that the file will have in the package. This property will have macro expansion applied to it. The dynamic macros \$(StageOutputFilePath) and \$(StudioDirRelativeOutputFilePath) are defined.
Set Readonly	Boolean	stage_set_readonly	Specifies that the output file will have its permissions set to read only.

Property	Type	Save Name	Description
Stage Command	String	stage_command	Specifies the command be used to do the staging operation. This property will have macro expansion applied to it. The dynamic macro <code>\$(StageOutputFilePath)</code> is defined to the value of the output file path property. The dynamic macro <code>\$(StudioDirRelativeOutputFilePath)</code> is defined to value of the output file path property relative to the <code>\$(StudioDir)</code> .

## Combining Options

These properties are applicable to **Combining** project types.

Property	Type	Save Name	Description
Output File Path	String	combine_output_filepath	Specifies the name of the file the combine command will create. This property will have macro expansion applied to it.
Set Readonly	Boolean	combine_set_readonly	Specifies that the output file will have its permissions set to read only.

Property	Type	Save Name	Description
Combine Command	String	combine_command	Specifies the command be used to do the combining operation. This property will have macro expansion applied to it. The dynamic macro <code>\$(CombiningRelInputPaths)</code> is defined to be the list of files in the project relative to the project directory.

## Target Interface Properties

These properties are applicable to **Executable** and **Externally Built Executable** project types.

## Target Options

Property	Type	Save Name	Description
Attach Script	String	target_attach_script	The script that is executed when the debugger attaches to the target.
Reset Script	String	target_reset_script	The script that is executed when the target is reset. This script is typically responsible for resetting the target and configuring memory.
Run Script	String	target_run_script	The script that is executed when the target is released into run state. This script is typically responsible for re-enabling caches previously disabled by the stop script.



Property	Type	Save Name	Description
Stop Script	String	target_stop_script	The script that is executed when the target enters debug state. This script is typically responsible for disabling or flushing caches.

---

Property	Type	Save Name	Description
ARM Debug Interface.	Enumeration	arm_target_debug_interface_type	Specifies the variant of debug interface that the target has. The options are <b>ARM7TDI</b> (for ARM7TD MI/ARM7 TDMI-S and ARM720T cores), <b>ARM9TDI</b> (for ARM920T and ARM9E cores), <b>XScale</b> (for XScale XSC1 cores), <b>XScale7BitIR</b> (for XScale XSC2 cores), and <b>ADiv5</b> - (for Cortex-M3 cores).

Property	Type	Save Name	Description
Debug Handler Load Address	String	arm_target_debug_handler_load_address	Some debug interfaces require a debug handler or monitor to be downloaded along with the application in order to debug the target. This property specifies the address to load the debug handler.

---

Property	Type	Save Name	Description
Debug Handler File Path	Filename	arm_target_debug_handler_file_path	Some debug interfaces require a debug handler or monitor to be downloaded along with the application in order to debug the target. This property specifies the file path to the debug handler to use, it should be left blank if no debug handler is required.

Property	Type	Save Name	Description
JTAG Data Bits After	Integer	arm_linker_jtag_pad_post_dr	Specifies the number of bits to pad the JTAG data register after the data for the ARM processor being targeted. As the width of the BYPASS register is normally 1 bit this value is usually equal to the number of devices in the scan chain after the device being targeted.

Property	Type	Save Name	Description
JTAG Data Bits Before	Integer	arm_linker_jtag_pad_pre_dr	Specifies the number of bits to pad the JTAG data register before the data for the ARM processor being targeted. As the width of the the BYPASS register is normally 1 bit this value is usually equal to the number of devices in the scan chain before the device being targeted.

Property	Type	Save Name	Description
JTAG Instruction Bits After	Integer	arm_linker_jtag_pad_post_ir	Specifies the number of bits to pad the JTAG instruction register with the BYPASS instruction (all bits set) after the instruction for the ARM processor being targeted. This value should be the combined length of the instruction registers for all devices in the scan chain after the ARM processor being targeted.

Property	Type	Save Name	Description
JTAG Instruction Bits Before	Integer	arm_linker_jtag_pad_pre_ir	Specifies the number of bits to pad the JTAG instruction register with the BYPASS instruction (all bits set) before the instruction for the ARM processor being targeted. This value should be the combined length of the instruction registers for all devices in the scan chain before the ARM processor being targeted.



Property	Type	Save Name	Description
First Loader Program Section	String	arm_target_loader_first_program_section	The name of the loader's first program section. This value is used to tell CrossStudio the area of memory occupied by the loader in order to prevent it from being overwritten during download. This parameter is only required if the program being downloaded overwrites the loader.

Property	Type	Save Name	Description
Last Loader Program Section	String	arm_target_loader_last_program_section	The name of the loader's last program section. This parameter is only required if the program section specified by <i>First Loader Program Section</i> is not the loader's only program section.

---

Property	Type	Save Name	Description
Loader File Path	Filename	arm_target_flash_loader_file_path	Specifies the file path of the loader program to use. This is typically used by targets that support FLASH download. It is not possible to download programs to FLASH using only the ARM's debug interface. A loader program therefore has to be downloaded and run from RAM prior to the download of the main application.

Property	Type	Save Name	Description
			Specifies the communication mechanism used to communicate with the loader. Options are <b>Comms Channel Loader</b> (a loader that communicates using the ARM DCC), <b>Fast Comms Channel Loader</b> (a loader that communicates using the ARM DCC), <b>RAM Loader</b> (a loader that communicates using RAM and breakpoints), and <b>LIBMEM RPC Loader</b> (a loader that communicates using remote procedure calls).
Loader File Type	Enumeration	arm_target_flash_loader_type	<b>Loader</b> (a loader that communicates using the ARM DCC), <b>RAM Loader</b> (a loader that communicates using RAM and breakpoints), and <b>LIBMEM RPC Loader</b> (a loader that communicates using remote procedure calls).

Property	Type	Save Name	Description
Loader Parameter	String	arm_target_loader_parameter	This field allows a parameter to be passed to the loader. The parameter is loader specific.
Reset After Download	Boolean	arm_target_loader_reset_after_download	Specifies whether the target should be reset after a download using a loader.
Stop CPU Using DBGRQ	Boolean	arm_target_stop_cpu_using_dbgq	Specifies whether the CPU should be stopped by asserting <b>DBGRQ</b> rather than by using breakpoints.

## ARM Simulator Properties

These properties are applicable to **Executable** and **Externally Built Executable** project types.

## Simulator Options

Property	Type	Save Name	Description
Memory Simulation Filename	Filename	arm_simulator_memory_simulation_filename	The path to the dynamic link library that will implement the memory for the ARM simulator. This property will have macro expansion applied to it.
Memory Simulation Parameter	String	arm_simulator_memory_simulation_parameter	A string to pass to the memory simulation file when it is connected to.

## Code editor command summary

The following table summarizes the keystrokes and corresponding menu items for code editor commands:

Keystrokes	Menu	Description
Up		Move the caret one line up.
Down		Move the caret one line down.
Left		Move the caret one character to the left.
Right		Move the caret one character to the right.
Home		Move the caret to the start of the current line.
End		Move the caret to the end of the current line.
PageUp		Move the caret on page up.
PageDown		Move the caret one page down.
Ctrl+Up		Scroll the document down one line.
Ctrl+Down		Scroll the document up one line.
Ctrl+Left		Move the caret to the start of the previous word.
Ctrl+Right		Move the caret to the start of the next word.
Ctrl+Home		Move the caret to the start of the document.
Ctrl+End		Move the caret to the end of the document.
Ctrl+PageUp		Move the caret to the top of the window.

Keystrokes	Menu	Description
Ctrl+PageDown		Move the caret to the bottom of the window.
Enter Return		Insert a new line and move the caret to an appropriate position on the next line dependant on the indent settings.
Shift+Up		Extend the current selection up by one line.
Shift+Down		Extend the current selection down by one line.
Shift+Left		Extend the current selection left by one character.
Shift+Right		Extend the current selection right by one character.
Shift+Home		Extend the current selection to the beginning of the current line.
Shift+End		Extend the current selection to the end of the current line.
Shift+PageUp		Extend the current selection up by one page.
Shift+PageDown		Extend the current selection down by one page.
Ctrl+Shift+Left		Extend the current selection to the beginning of the previous word.
Ctrl+Shift+Right		Extend the current selection to the end of the next word.
Ctrl+Shift+Home		Extend the current selection to the beginning of the file.
Ctrl+Shift+End		Extend the current selection to the end of the file.
Ctrl+Shift+PageUp		Extend the current selection to the top of the window.
Ctrl+Shift+PageDown		Extend the current selection to the end of the window.
Ctrl+Shift+] ]		Select the text contained within the nearest delimiter pair.
Ctrl+A		Select the entire document.
Ctrl+F8		Select the current line.
	Edit   Advanced   Sort Ascending	Sort the lines contained within the current selection into ascending order.



Keystrokes	Menu	Description
	Edit   Advanced   Sort Descending	Sort the lines contained within the current selection into descending order.
Ctrl+C Ctrl+Insert	Edit   Copy	Copy the current selection into the clipboard.
Ctrl+X Shift+Delete	Edit   Cut	Copy the current selection into the clipboard and remove the selected text from the document.
Ctrl+V Shift+Insert	Edit   Paste	Insert the contents of the clipboard into the document at the current caret position.
Ctrl+L		Cut the current line <i>or selection</i> .
Ctrl+Shift+L		Delete the current line <i>or selection</i> .
	Edit   Clipboard   Clear Clipboard	Empty the current contents of the clipboard.
Ctrl+F2	Edit   Bookmarks   Toggle Bookmark	Add or remove a bookmark to the current line.
F2	Edit   Bookmarks   Next Bookmark	Move the caret to the next bookmark.
Shift+F2	Edit   Bookmarks   Previous Bookmark	Move the caret to the previous bookmark.
	Edit   Bookmarks   First Bookmark	Move the caret to the first bookmark in the document.
	Edit   Bookmarks   Last Bookmark	Move the caret to the last bookmark in the document.
Ctrl+Shift+F2	Edit   Bookmarks   Clear All Bookmarks	Remove all bookmarks from the document.
Alt+F2		Add a permanent bookmark on the current line.

Keystrokes	Menu	Description
Ctrl+F	Edit   Find	Display the find dialog.
Ctrl+H	Edit   Replace	Display the replace dialog.
F3		Find the next occurrence of the previous search ahead of the current caret position.
Shift+F3		Find the next occurrence of the previous search behind the current caret position.
Ctrl+] ]		Find the matching delimiter character for the nearest delimiter character on the current line.
Ctrl+F3		Search up the document for currently selected text.
Ctrl+Shift+F3		Search down the document for the currently selected text.
Ctrl+G, Ctrl+L		Display the goto line dialog.
Backspace		Delete the character to the left of the caret position.
Delete		Delete the character to the right of the caret position.
Ctrl+Backspace		Delete from the caret position to the start of the current word.
Ctrl+Delete		Delete from the caret position to the end of the current word.
Ctrl+L		Delete current line.
Ctrl+Alt+L		Delete from the caret position to the end of the line.
Alt+Shift+L		Delete from the caret position to the next blank line.
Tab	Edit   Advanced   Increase Line Indent	Either advance the caret to the next indent position or, if there is selected text, indent each line of the selection.
Shift+Tab	Edit   Advanced   Decrease Line Indent	Either move the caret to the previous indent position or, if there is selected text, unindent each line of the selection.
Alt+Right		Indent the current line.
Alt+Left		Unindent the current line.
Ctrl+S	File   Save	Save the current file.

Keystrokes	Menu	Description
	File   Save As	Save the current file under a different file name.
Ctrl+Shift+S	File   Save All	Save all the files.
Ctrl+P	File   Print	Print the current file.
Ctrl+U	Edit   Advanced   Make Selection Lowercase	Either change the current character to lowercase or, if there is selected text, change all characters within the selection to lowercase.
Ctrl+Shift+U	Edit   Advanced   Make Selection Uppercase	Either change the current character to uppercase or, if there is selected text, change all characters within the selection to uppercase.
Ctrl+ /	Edit   Advanced   Comment	If there is a selection, adds a comment to the start of each selected line. If there is no selection, adds a comment to the start of the line the caret is on.
Ctrl+Shift+ /	Edit   Advanced   Uncomment	If there is a selection, removes any comment from the start of each selected line. If there is no selection, removes any comment from the start of the line the caret is on.
Ctrl+Z or Alt+Backspace	Edit   Undo	Undoes the last operation.
Ctrl+Y	Edit   Redo	Redoes the last operation.
Insert		Enable or disable overwrite mode.
Ctrl+Shift+T		Swap the current word with the previous word or, if there is no previous word, the next word.
Alt+Shift+T		Swap the current line with the previous line or, if there is no previous line, the next line.
Ctrl+Alt+J		Appends the line below the caret onto the end of the current line.
	Edit   Advanced   Tabify Selection	Replace whitespace with appropriate tabs within the current selection.
	Edit   Advanced   Untabify Selection	Remove tabs from within the current selection.

Keystrokes	Menu	Description
	Edit   Advanced   Visible Whitespace	Enable or disable visible whitespace.
	Edit   Advanced   Toggle Read Only	Toggle the write permissions of the current file.

## Binary editor command summary

The following table summarizes the keystrokes and corresponding menu items for binary editor commands:

Keystrokes	Menu	Action
Up		Move the caret 16 bytes back.
Down		Move the caret 16 bytes forward.
Left		Move the caret one byte back.
Right		Move the caret one byte forward.
Home		Move the caret to the start of the current line of bytes.
End		Move the caret to the end of the current line of bytes.
Page Up		Move the caret one page up.
Page Down		Move the caret one page down.
Ctrl+Home		Move the caret to address 0.
Ctrl+End		Move the caret to the address of the last byte in the file.
Ctrl+Up		Move the view up one line.
Down		Move the view down one line.
Ctrl+Left		Move the caret 4 bytes back.
Ctrl+Right		Move the caret 4 bytes forward.
Ctrl+F	Edit   Find	Display the find dialog.

Keystrokes	Menu	Action
F3		Find the next occurrence of the value most recently searched for.
Backspace		Removes the byte in the address before the caret position.
Delete		Removes the currently selected byte.
Ctrl+S	File   Save	Save the current file.
	File   Save As	Save the current file under a different file name. <b>WARNING</b> the current version of the binary editor continues to display the original file name after a "Save as" operation.
Ctrl+Z	Edit   Undo	Undoes the last operation.
Ctrl+Y	Edit   Redo	Redoes the last undone operation.
Insert		Enable or disable overwrite mode.
Ctrl+T		When in text input mode the currently selected byte will be replaced with the ASCII character code for the input key. e.g. 0F would become 66 when f is pressed. When not in text input mode the currently selected byte is replaced with the HEX value of the input keys. e.g. 00 would become 0F when f is pressed. 00 would become FA if the 'f' then 'a' keys were pressed.
	Edit   Advanced   Toggle Read Only	Toggle the write permissions of the current file.
Ctrl+E		Allows the file to be a fixed size or a variable size
Down or Right		Extends the file when the last address is selected and the file is write enabled. the new bytes will be initialised to 00.

The following terms are in common use and are used throughout the CrossWorks documentation:

- **Active project.** The project that is currently selected in the **Project Explorer**. The **Build** tool bar contains a dropdown and the **Project > Set Active Project** menu contains an item that display the active project. You can change the active project using either of these elements.
- **Active configuration.** The configuration that is currently selected for building. The **Build** too bar contains a dropdown and the **Build > Set Active Build Configuration** menu display the active configuration. You can change the active configuration using either of these elements.
- **Assembler.** A program that translates low-level assembly language statements into executable machine code. See *Assembler Reference*.
- **Compiler.** A program that translates high-level statement into executable machine code. See *C Compiler Reference*.
- **Integrated development environment.** A program that supports editing, managing, building, and debugging your programs within a single environment.
- **Linker.** A program that combines multiple relocatable object modules and resolves inter-module references to produce an executable program. See *Linker Reference*.
- **Project explorer.** A docking indow that contains a visual representation of the project. See **Project explorer** (page 128).

