IAR Embedded Workbench®

IDE Project Management and Building Guide

for Advanced RISC Machines Ltd's
ARM Cores

UIDEARM-1

IAR SYSTEMS

EDITION NOTICE

First edition: October 2010

Part number: UIDEARM-1

This guide applies to version 6.1x of IAR Embedded Workbench® for Advanced RISC Machines Ltd's ARM core family. Together with the *C-SPY® Debugging Guide for ARM®*, the *IDE Project Management and Building Guide for ARM®* replaces the *IAR Embedded Workbench IDE User Guide for ARM®*.

Internal reference: M7, Too6.1, ISUD.

Brief contents

Contents

Tables

Figures

Preface

Welcome to the *IDE Project Management and Building Guide for ARM*. The purpose of this guide is to help you fully use the features in IAR Embedded Workbench with its integrated Windows development tools for the ARM core. The IAR Embedded Workbench IDE is a very powerful Integrated Development Environment that allows you to develop and manage a complete embedded application project.

This guide describes the processes of editing, project managing, and building, and provides related reference information

Who should read this guide

Read this guide if you want to get the most out of the features and tools available in the IDE. In addition, you should have working knowledge of:

- The C or C++ programming language
- Application development for embedded systems
- The architecture and instruction set of the ARM core (refer to the chip manufacturer's documentation)
- The operating system of your host computer.

For more information about the other development tools incorporated in the IDE, refer to their respective documentation, see *Other documentation*, page 17.

How to use this guide

If you are new to using this product, we suggest that you first read the guide *Getting Started with IAR Embedded Workbench®* for an overview of the tools and the features that the IDE offers. The tutorials which you can find in IAR Information Center will help you get started using IAR Embedded Workbench.

The process of managing projects and building, as well as editing, is described in this guide, whereas information about how to use C-SPY for debugging is described in the *C-SPY® Debugging Guide for ARM®*.

If you are an experienced user and need this guide only for reference information, see the reference chapters in *Part 2. Reference information* and the online help system available from the IAR Embedded Workbench IDE **Help** menu.

Finally, we recommend the *Glossary* if you should encounter any unfamiliar terms in the IAR Systems user documentation.

What this guide contains

This is a brief outline and summary of the chapters in this guide.

Part 1. Project management and building

This section describes the process of editing and building your application:

- *The development environment* introduces you to the IAR Embedded Workbench development environment. The chapter also demonstrates the facilities available for customizing the environment to meet your requirements.
- *Managing projects* describes how you can create workspaces with multiple projects, build configurations, groups, source files, and options that help you handle different versions of your applications.
- *Building* discusses the process of building your application.
- *Editing* contains detailed descriptions of the IAR Embedded Workbench editor, how to use it, and the facilities related to its usage. The final section also contains information about how to integrate an external editor of your choice.

Part 2. Reference information

- *Installed files* describes the directory structure and the types of files it contains.
- *IAR Embedded Workbench IDE reference* contains detailed reference information about the development environment, such as details about the graphical user interface.
- *General options* specifies the target, output, library, and MISRA C options.
- *Compiler options* specifies compiler options for language, optimizations, code, output, list file, preprocessor, diagnostics, and MISRA C.
- *Assembler options* describes the assembler options for language, output, list, preprocessor, and diagnostics.
- *Output converter options* describes the options available for converting linker output files from the ELF format.
- *Custom build options* describes the options available for custom tool configuration.

- *Build actions options* describes the options available for pre-build and post-build actions.
- *Linker options* describes the options for setting up for linking.
- *Library builder options* describes the options for building a library.

Other documentation

User documentation is available as hypertext PDFs and as a context-sensitive online help system in HTML format. You can access the documentation from the Information Center or from the **Help** menu in the IAR Embedded Workbench IDE. The online help system is also available via the F1 key.

USER AND REFERENCE GUIDES

The complete set of IAR Systems development tools is described in a series of guides. For information about:

- System requirements and information about how to install and register the IAR Systems products, refer to the booklet *Quick Reference* (available in the product box) and the *Installation and Licensing Guide*.
- Getting started using IAR Embedded Workbench and the tools it provides, refer to the guide *Getting Started with IAR Embedded Workbench®* .
- Using the IAR C-SPY® Debugger, refer to the *C-SPY® Debugging Guide for ARM®*
- Programming for the IAR C/C++ Compiler for ARM and linking using the IAR ILINK Linker, refer to the *IAR C/C++ Development Guide for ARM®*
- Programming for the IAR C/C++ Compiler, refer to the *IAR C/C++ Compiler Reference Guide* if your product package includes the IAR XLINK Linker, and the *IAR C/C++ Development Guide, Compiling and Linking* if your product package includes the IAR ILINK Linker.
- Programming for the IAR Assembler for ARM, refer to the *ARM® IAR Assembler Reference Guide*.
- Using the IAR DLIB Library, refer to the *DLIB Library Reference information*, available in the online help system.
- Porting application code and projects created with a previous version of the IAR Embedded Workbench for ARM, refer to *IAR Embedded Workbench® Migration Guide for ARM®*.
- Developing safety-critical applications using the MISRA C guidelines, refer to the *IAR Embedded Workbench® MISRA C:2004 Reference Guide* or the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*.

Note: Additional documentation might be available depending on your product installation.

THE ONLINE HELP SYSTEM

The context-sensitive online help contains:

- Comprehensive information about debugging using the IAR C-SPY® Debugger
- Reference information about the menus, windows, and dialog boxes in the IDE
- Compiler reference information
- Keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1.

WEB SITES

Recommended web sites:

- The Advanced RISC Machines Ltd web site, **www.arm.com**, contains information and news about the ARM cores.
- The IAR Systems web site, **www.iar.com**, holds application notes and other product information.
- Finally, the Embedded C++ Technical Committee web site, **www.caravan.net/ec2plus**, contains information about the Embedded C++ standard.

Document conventions

When, in this text, we refer to the programming language C, the text also applies to C++, unless otherwise stated.

When referring to a directory in your product installation, for example `arm\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench 6.`*`n`*`\arm\doc`.

TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:

Style	Used for
`computer`	• Source code examples and file paths. • Text on the command line. • Binary, hexadecimal, and octal numbers.

Table 1: Typographic conventions used in this guide

Style	Used for
`parameter`	A placeholder for an actual value used as a parameter, for example *`filename`*`.h` where *`filename`* represents the name of the file.
`[option]`	An optional part of a command.
`[a|b|c]`	An optional part of a command with alternatives.
`{a|b|c}`	A mandatory part of a command with alternatives.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
italic	• A cross-reference within this guide or to another guide. • Emphasis.
…	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
	Identifies instructions specific to the IAR Embedded Workbench® IDE interface.
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.
	Identifies warnings.

Table 1: Typographic conventions used in this guide (Continued)

NAMING CONVENTIONS

The following naming conventions are used for the products and tools from IAR Systems® referred to in this guide:

Brand name	Generic term
IAR Embedded Workbench® for ARM	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for ARM	the IDE
IAR C-SPY® Debugger for ARM	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for ARM	the compiler
IAR Assembler™ for ARM	the assembler
IAR ILINK Linker™	ILINK, the linker
IAR DLIB Library™	the DLIB library

Table 2: Naming conventions used in this guide

Part 1. Project management and building

This part of the IDE Project Management and Building Guide contains these chapters:

- The development environment
- Managing projects
- Building
- Editing.

The development environment

This chapter introduces you to the IAR Embedded Workbench® integrated development environment (IDE). The chapter also demonstrates how you can customize the environment to suit your requirements.

The IAR Embedded Workbench IDE—an overview

THE TOOLCHAIN

The IDE is the environment where all necessary tools—the *toolchain*—are integrated: a C/C++ compiler, an assembler, a linker, an editor, a project manager with Make utility, and the IAR C-SPY® Debugger. The tools used specifically for building your source code are referred to as the *build tools*.

The compiler, assembler, and linker can also be run from a command line environment, if you want to use them as external tools in an already established project environment.

CONTINUOUS WORKFLOW

You have the same user interface regardless of which microcontroller you have chosen to work with—coupled with general and target-specific support for each device.

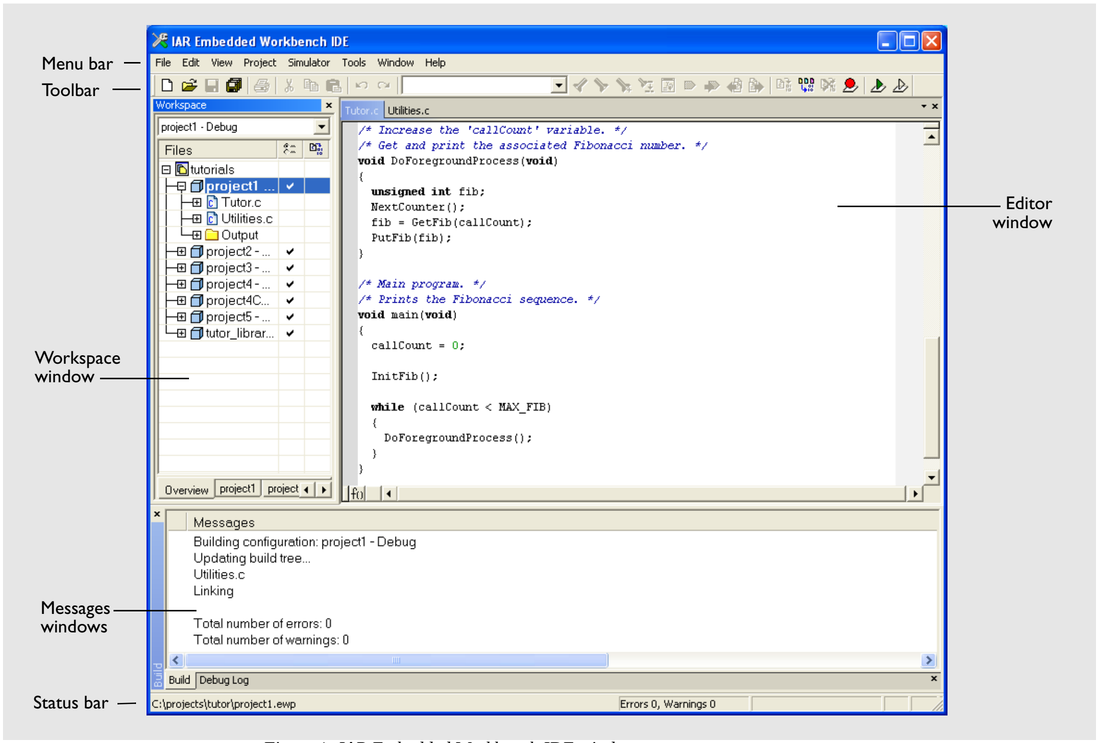
AN EXTENSIBLE AND MODULAR ENVIRONMENT

Although the IDE provides all the features required for your project, you can also integrate with other tools. For example, you can add IAR visualSTATE to the toolchain, which means that you can add state machine diagrams directly to your project in the IDE. Source code control (SCC)—or revision control—is useful for keeping track of different versions of your source code. IAR Embedded Workbench can identify and access any third-party source code control system that conforms to the SCC interface published by Microsoft. You can use the Custom Build mechanism to incorporate also other tools to the toolchain, see *Extending the toolchain*, page 46.

WINDOW MANAGEMENT

To give you full and convenient control of the placement of the windows, each window is *dockable* and you can optionally organize the windows in *tab groups*.

This illustration shows the IAR Embedded Workbench IDE window with various components.



```
/* Increase the 'callCount' variable. */
/* Get and print the associated Fibonacci number. */
void DoForegroundProcess(void)
{
  unsigned int fib;
  NextCounter();
  fib = GetFib(callCount);
  PutFib(fib);
}

/* Main program. */
/* Prints the Fibonacci sequence. */
void main(void)
{
  callCount = 0;

  InitFib();

  while (callCount < MAX_FIB)
  {
    DoForegroundProcess();
  }
}
```

```
Building configuration: project1 - Debug
Updating build tree...
Utilities.c
Linking

Total number of errors: 0
Total number of warnings: 0
```

Figure 1: IAR Embedded Workbench IDE window

The window might look different depending on what additional tools you are using.

RUNNING THE IDE

Click the **Start** button on the Windows taskbar and choose **All Programs>IAR Systems>IAR Embedded Workbench for ARM>IAR Embedded Workbench**.

The file `IarIdePm.exe` is located in the `common\bin` directory under your IAR Systems installation, in case you want to start the program from the command line or from within Windows Explorer.

Double-clicking the workspace filename

The workspace file has the filename extension `eww`. If you double-click a workspace filename, the IDE starts. If you have several versions of IAR Embedded Workbench installed, the workspace file is opened by the most recently used version of your IAR Embedded Workbench that uses that file type.

EXITING

To exit the IDE, choose **File>Exit**. You will be asked whether you want to save any changes to editor windows, the projects, and the workspace before closing them.

Customizing the environment

The IDE is a highly customizable environment. This section demonstrates how you can work with and organize the windows on the screen, the possibilities for customizing the IDE, and how you can set up the environment to communicate with external tools.

ORGANIZING THE WINDOWS ON THE SCREEN

In the IDE, you can position the windows and arrange a layout according to your preferences. You can *dock* windows at specific places, and organize them in tab groups. You can also make a window *floating,* which means it is always on top of other windows. If you change the size or position of a floating window, other currently open windows are not affected.

Each time you open a previously saved workspace, the same windows are open, and they have the same sizes and positions.

For every project that is executed in the C-SPY environment, a separate layout is saved. In addition to the information saved for the workspace, information about all open debugger-specific windows is also saved.

Using docked versus floating windows

Each window that you open has a default location, which depends on other currently open windows. To give you full and convenient control of window placement, each window can either be docked or floating.

A docked window is locked to a specific area in the Embedded Workbench main window, which you can decide. To keep many windows open at the same time, you can organize the windows in tab groups. This means one area of the screen is used for several concurrently open windows. The system also makes it easy to rearrange the size of the windows. If you rearrange the size of one docked window, the sizes of any other docked windows are adjusted accordingly.

A floating window is always on top of other windows. Its location and size does not affect other currently open windows. You can move a floating window to any place on your screen, also outside of the IAR Embedded Workbench IDE main window.

Note: The editor window is always docked. When you open the editor window, its placement is decided automatically depending on other currently open windows. For more information about how to work with the editor window, see *Using the IAR Embedded Workbench editor*, page 49.

Organizing windows

To place a window as a *separate* window, drag it next to another open window.

To place a window in the same tab group as another open window, drag the window you want to locate and drop it in the middle of the other window.

To make a window floating, double-click on the window's title bar.

The status bar, located at the bottom of the IAR Embedded Workbench IDE main window, contains useful help about how to arrange windows.

CUSTOMIZING THE IDE

To customize the IDE, choose **Tools>Options** to get access to a wide variety of commands for:

- Configuring the editor
- Configuring the editor colors and fonts
- Configuring the project build command
- Organizing the windows in C-SPY
- Using an external editor
- Changing common fonts
- Changing key bindings
- Configuring the amount of output to the Messages window.

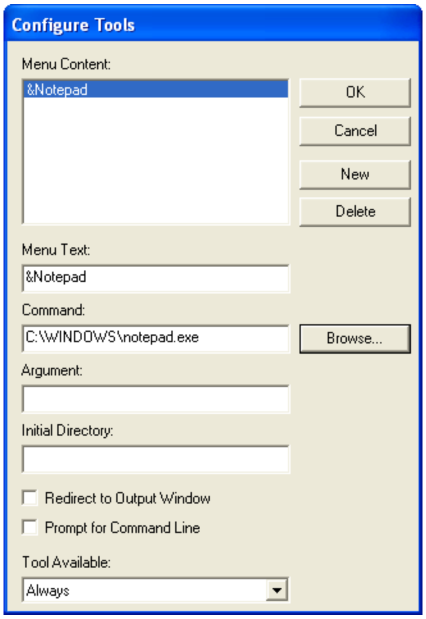
In addition, you can increase the number of recognized filename extensions. By default, each tool in the build toolchain accepts a set of standard filename extensions. If you have source files with a different filename extension, you can modify the set of accepted filename extensions. Choose **Tools>Filename Extensions** to get access to the necessary commands.

For reference information about the commands for customizing the IDE, see *Tools menu*, page 121. You can also find further information related to customizing the editor in the section *Customizing the editor environment*, page 57. For further information about customizations related to C-SPY, see the *C-SPY® Debugging Guide for ARM®*.

INVOKING EXTERNAL TOOLS

The **Tools** menu is a configurable menu to which you can add external tools for convenient access to these tools from within the IDE. For this reason, the menu might look different depending on which tools you have preconfigured to appear as menu commands.

To add an external tool to the menu, choose **Tools>Configure Tools** to open the **Configure Tools** dialog box.



Menu Content: &Notepad

Menu Text: &Notepad

Command: C:\WINDOWS\notepad.exe

Argument:

Initial Directory:

Redirect to Output Window

Prompt for Command Line

Tool Available: Always

Figure 2: Configure Tools dialog box

For reference information about this dialog box, see *Configure Tools dialog box*, page 143.

Note: You cannot use the **Configure Tools** dialog box to extend the toolchain in the IDE, see *The toolchain*, page 23.

After you have entered the appropriate information and clicked **OK**, the menu command you have specified is displayed on the **Tools** menu.

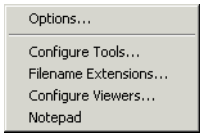


Figure 3: Customized Tools menu

Note: If you intend to add an external tool to the standard build toolchain, see *Extending the toolchain*, page 46.

Adding command line commands

Command line commands and calls to batch files must be run from a command shell. You can add command line commands to the **Tools** menu and execute them from there.

1 To add commands to the **Tools** menu, you must specify an appropriate command shell. Type a command shell in the **Command** text box, for example `cmd.exe`.

2 Specify the command line command or batch file name in the **Argument** text box.

The **Argument** text should be specified as:

```
/C name
```

where *`name`* is the name of the command or batch file you want to run.

The `/C` option terminates the shell after execution, to allow the IDE to detect when the tool has finished.

Example

To add the command **Backup** to the **Tools** menu to make a copy of the entire `project` directory to a network drive, you would specify **Command** as `cmd.exe` (or `command.cmd` depending on your host environment), and **Argument** as:

```
/C copy c:\project\*.* F:
```

Alternatively, to use a variable for the argument to allow relocatable paths:

```
/C copy $PROJ_DIR$\*.* F:
```

Managing projects

This chapter discusses the project model used by the IAR Embedded Workbench IDE. It covers how projects are organized and how you can specify workspaces with multiple projects, build configurations, groups, source files, and options that help you handle different versions of your applications. The chapter also describes the steps involved in interacting with an external third-party source code control system.

The project model

In a large-scale development project, with hundreds of files, you must be able to organize the files in a structure that is easily navigated and maintained by perhaps several engineers involved.

MANAGING PROJECTS—AN OVERVIEW

The IDE comes with functions that will help you to stay in control of all project modules, for example, C or C++ source code files, assembler files, include files, and other related modules. You create workspaces and let them contain one or several projects. Files can be grouped, and options can be set on all levels—project, group, or file. Changes are tracked so that a request for rebuild will retranslate all required modules, making sure that no executable files contain out-of-date modules. This list shows some additional features:

- Project templates to create a project that can be built and executed for a smooth development startup
- Hierarchical project representation
- Source browser with an hierarchical symbol presentation
- Options can be set globally, on groups of source files, or on individual source files
- The Make command automatically detects changes and performs only the required operations
- Text-based project files
- Custom Build utility to expand the standard toolchain in an easy way
- Command line build with the project file as input.

HOW PROJECTS ARE ORGANIZED

The IDE has been designed to suit the way that software development projects are typically organized. For example, perhaps you need to develop related versions of an application for different versions of the target hardware, and you might also want to include debugging routines into the early versions, but not in the final application.

Versions of your applications for different target hardware will often have source files in common, and you might want to be able to maintain only one unique copy of these files, so that improvements are automatically carried through to each version of the application. Perhaps you also have source files that differ between different versions of the application, such as those dealing with hardware-dependent aspects of the application.

The IDE allows you to organize projects in an hierarchical tree structure showing the logical structure at a glance. In the following sections the various levels of the hierarchy are described.

Projects and workspaces

Typically you create a *project* which contains the source files needed for your embedded systems application. If you have several related projects, you can access and work with them simultaneously. To achieve this, you can organize related projects in *workspaces*.

Each workspace you define can contain one or more projects, and each project must be part of at least one workspace.

Consider this example: two related applications—for instance A and B—are developed, requiring one development team each (team A and B). Because the two applications are related, they can share parts of the source code between them. The following project model can be applied:

- Three projects—one for each application, and one for the common source code
- Two workspaces—one for team A and one for team B.

Collecting the common sources in a library project (compiled but not linked object code) is both convenient and efficient, to avoid having to compile it unnecessarily.

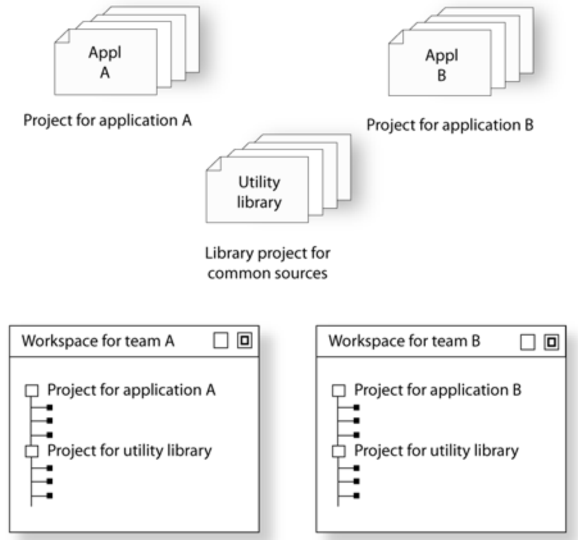

Appl
A
Project for application A
Appl
B
Project for application B
Utility
library
Library project for common sources
Workspace for team A
Project for application A
Project for utility library
Workspace for team B
Project for application B
Project for utility library

Figure 4: Examples of workspaces and projects

For an example where a library project has been combined with an application project, see *Creating and using libraries* in the tutorials.

Projects and build configurations

Often, you need to build several versions of your project. The Embedded Workbench lets you define multiple build configurations for each project. In a simple case, you might need just two, called **Debug** and **Release**, where the only differences are the options used for optimization, debug information, and output format. In the Release configuration, the preprocessor symbol `NDEBUG` is defined, which means the application will not contain any asserts.

Additional build configurations might be useful, for instance, if you intend to use the application on different target devices. The application is the same, but hardware-related parts of the code differ. Thus, depending on which target device you intend to build for, you can exclude some source files from the build configuration. These build configurations might fulfil these requirements for Project A:

- Project A - Device 1:Release
- Project A - Device 1:Debug

- Project A - Device 2:Release
- Project A - Device 2:Debug

Groups

Normally, projects contain hundreds of files that are logically related. You can define each project to contain one or more groups, in which you can collect related source files. You can also define multiple levels of subgroups to achieve a logical hierarchy. By default, each group is present in all build configurations of the project, but you can also specify a group to be excluded from a particular build configuration.

Source files

Source files can be located directly under the project node or in a hierarchy of groups. The latter is convenient if the amount of files makes the project difficult to survey. By default, each file is present in all build configurations of the project, but you can also specify a file to be excluded from a particular build configuration.

Only the files that are part of a build configuration will actually be built and linked into the output code.

Once a project has been successfully built, all include files and output files are displayed in the structure below the source file that included or generated them.

Note: The settings for a build configuration can affect which include files that are used during the compilation of a source file. This means that the set of include files associated with the source file after compilation can differ between the build configurations.

CREATING AND MANAGING WORKSPACES

This section describes the overall procedure for creating the workspace, projects, groups, files, and build configurations. The **File** menu provides the commands for creating workspaces. The **Project** menu provides commands for creating projects, adding files to a project, creating groups, specifying project options, and running the IAR Systems development tools on the current projects.

For reference information about these menus, menu commands, and dialog boxes, see *IAR Embedded Workbench IDE reference*, page 69.

The steps involved for creating and managing a workspace and its contents are:

- Creating a workspace.

  An empty Workspace window appears, which is the place where you can view your projects, groups, and files.

- Adding new or existing projects to the workspace.

  When creating a new project, you can base it on a *template project* with preconfigured project settings. Template projects are available for C applications, C++ applications, assembler applications, and library projects.
- Creating groups.

  A group can be added either to the project's top node or to another group within the project.
- Adding files to the project.

  A file can be added either to the project's top node or to a group within the project.
- Creating new build configurations.

  By default, each project you add to a workspace will have two build configurations called **Debug** and **Release**.

  You can base a *new* configuration on an already existing configuration. Alternatively, you can choose to create a default build configuration.

  Note that you do not have to use the same toolchain for the new build configuration as for other build configurations in the same project.
- Excluding groups and files from a build configuration.

  Note that the icon indicating the excluded group or file will change to white in the Workspace window.
- Removing items from a project.

For a detailed example, see *Creating an application project* in the tutorials.

Note: It might not be necessary for you to perform all of these steps.

Drag and drop

You can easily drag individual source files and project files from the Windows file explorer to the Workspace window. Source files dropped on a *group* are added to that group. Source files dropped outside the project tree—on the Workspace window background—are added to the active project.

Source file paths

The IDE supports relative source file paths to a certain degree, for:

- Project file

  Paths to files part of the project file is relative if they are located on the same drive. The path is relative either to `$PROJ_DIR$` or `$EW_DIR$`. The argument variable `$EW_DIR$` is only used if the path refers to a file located in subdirectory to `$EW_DIR$` and the distance from `$EW_DIR$` is shorter than the distance from `$PROJ_DIR$`.

Paths to files that are part of the project file are absolute if the files are located on different drives.

- Workspace file

  For files located on the same drive as the workspace file, the path is relative to `$PROJ_DIR$`.

  For files located on another drive as the workspace file, the path is absolute.

- Debug files

  The path is absolute if the file is built with IAR Systems compilation tools.

Starting the IAR C-SPY® Debugger

When you start the IAR C-SPY Debugger, the current project is loaded. It is also possible to load C-SPY with a project that was built outside IAR Embedded Workbench, for example projects built on the command line. For more information, see the *C-SPY® Debugging Guide for ARM®*.

Navigating project files

There are two main different ways to navigate your project files: using the Workspace window or the Source Browser window. The Workspace window displays an hierarchical view of the source files, dependency files, and output files and how they are logically grouped. The Source Browser window, on the other hand, displays information about the build configuration that is currently active in the Workspace window. For that configuration, the Source Browser window displays a hierarchical view of all globally defined symbols, such as variables, functions, and type definitions. For classes, information about any base classes is also displayed.

VIEWING THE WORKSPACE

The Workspace window is where you access your projects and files during the application development.

1 To choose which project you want to view, click its tab at the bottom of the Workspace window.

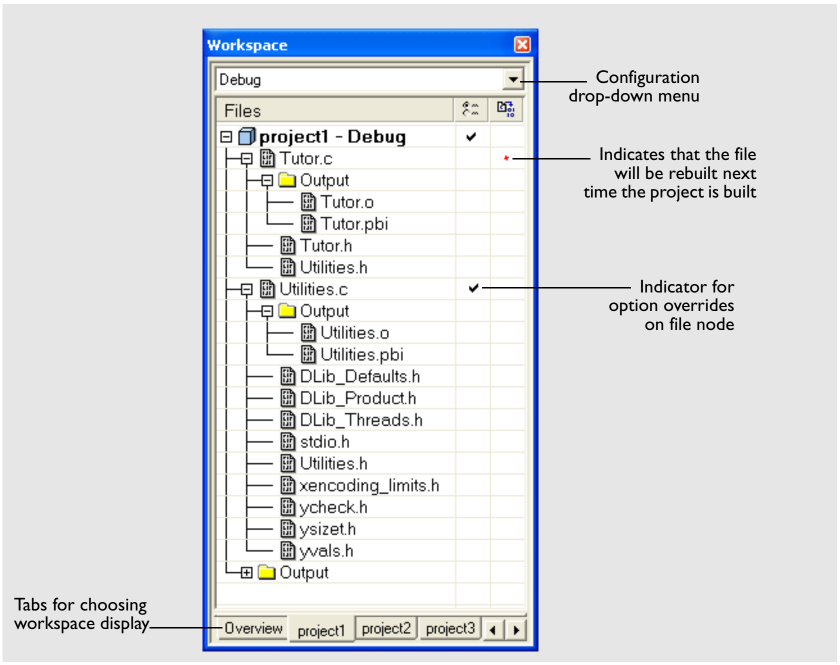


Figure 5: Displaying a project in the Workspace window

For each file that has been built, an `Output` folder icon appears, containing generated files, such as object files and list files. The latter is generated only if the list file option is enabled. There is also an `Output` folder related to the project node that contains generated files related to the whole project, such as the executable file and the linker map file (if the list file option is enabled).

Also, any included header files will appear, showing dependencies at a glance.

2 To display the project with a different build configuration, choose that build configuration from the drop-down list at the top of the Workspace window.

The project and build configuration you have selected are displayed highlighted in the Workspace window. It is the project and build configuration that you select from the drop-down list that is built when you build your application.

3 To display an overview of all projects in the workspace, click the **Overview** tab at the bottom of the Workspace window.

An overview of all project members is displayed.

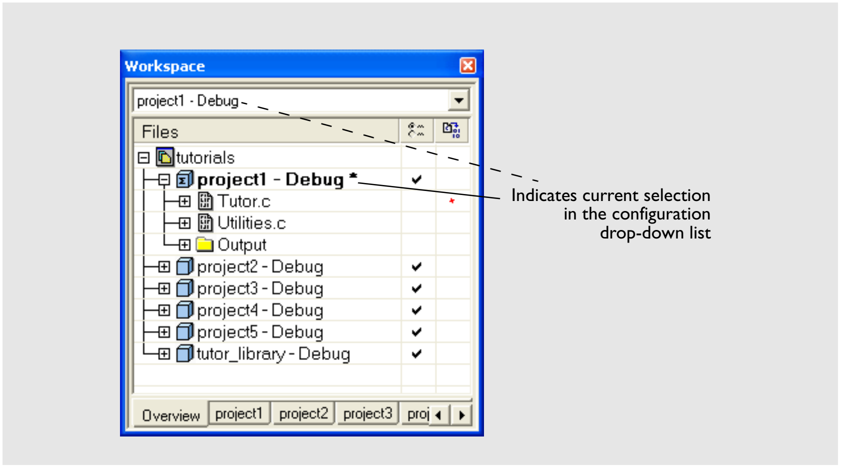


Figure 6: Workspace window—an overview

The current selection in the **Build Configuration** drop-down list is also highlighted when an overview of the workspace is displayed.

DISPLAYING BROWSE INFORMATION

To display browse information in the Source Browser window, choose **Tools>Options>Project** and select the option **Generate browse information**.

To open the Source Browser window, choose **View>Source Browser**. The Source Browser window is, by default, docked with the Workspace window. Source browse information is displayed for the active build configuration. For reference information, see *Source Browser window*, page 87.

Note that you can choose a file filter and a type filter from the context menu that appears when you right-click in the top pane of the window.

To see the definition of a global symbol or a function, you can use three alternative methods:

- In the Source Browser window, right-click on a symbol, or function, and choose the **Go to definition** command from the context menu that appears

- In the Source Browser window, double-click on a row
- In the editor window, right-click on a symbol, or function, and choose the **Go to definition** command from the context menu that appears.

The definition of the symbol or function is displayed in the editor window.

The source browse information is continuously updated in the background. While you are editing source files, or when you open a new project, there will be a short delay before the information is up-to-date.

Source code control

IAR Embedded Workbench can identify and access any installed third-party source code control (SCC) system that conforms to the SCC interface published by Microsoft corporation. From within the IDE you can connect an IAR Embedded Workbench project to an external SCC project, and perform some of the most commonly used operations.

To connect your IAR Embedded Workbench project to a source code control system you should be familiar with the source code control *client application* you are using. Note that some of the windows and dialog boxes that appear when you work with source code control in the IAR Embedded Workbench IDE originate from the SCC system and are not described in the documentation from IAR Systems. For information about details in the client application, refer to the documentation supplied with that application.

Note: Different SCC systems use very different terminology even for some of the most basic concepts involved. You must keep this in mind when you read the following description.

INTERACTING WITH SOURCE CODE CONTROL SYSTEMS

In any SCC system, you use a client application to maintain a central archive. In this archive you keep the working copies of the files of your project. The SCC integration in IAR Embedded Workbench allows you to conveniently perform a few of the most common SCC operations directly from within the IDE. However, several tasks must still be performed in the client application.

To connect an IAR Embedded Workbench project to a source code control system, you should:

- In the SCC client application, set up an SCC project
- In IAR Embedded Workbench, connect your project to the SCC project.

Setting up an SCC project in the SCC client application

Use your SCC client tools to set up a working directory for the files in your IAR Embedded Workbench project that you want to control using your SCC system. The files can be placed in one or more nested subdirectories, all located under a common root. Specifically, all the source files must reside in the same directory as the `ewp` project file, or nested in subdirectories of this directory.

For information about the steps involved, refer to the documentation supplied with the SCC client application.

Connecting projects in IAR Embedded Workbench

In IAR Embedded Workbench, connect your application project to the SCC project.

1. In the Workspace window, select the project for which you have created an SCC project. From the **Project** menu, choose **Source Code Control>Add Project To Source Control**. This command is also available from the context menu that appears when you right-click in the Workspace window.

   Note: The commands on the **Source Code Control** submenu are available when at least one SCC client application is available.

2. If you have source code control systems from different vendors installed, a dialog box will appear to let you choose which system you want to connect to.

3. An SCC-specific dialog box will appear where you can navigate to the proper SCC project that you have set up.

Viewing the SCC states

When your IAR Embedded Workbench project has been connected to the SCC project, a column that contains status information for source code control will appear in the Workspace window. Different icons are displayed depending on whether:

- a file is checked out to you
- a file is checked out to someone else
- a file is checked in
- a file has been modified
- a new version of a file is in the archive.

There are also icons for some combinations of these states. Note that the interpretation of these states depends on the SCC client application you are using. For reference information about the icons and the different states they represent, see *Source code control states*, page 77.

For reference information about the commands available for accessing the SCC system, see *Source Code Control menu*, page 76.

Configuring the source code control system

To customize the source code control system, choose **Tools>Options** and click the **Source Code Control** tab. For reference information about the available commands, see *Source Code Control options*, page 136.

Building

This chapter briefly discusses the process of building your application, and describes how you can extend the chain of build tools with tools from third-party suppliers.

Building your application

The build process consists of these steps:

- Setting project options
- Building the project
- Correcting any errors detected during the build procedure.

To make the build process more efficient, you can use the **Batch Build** command. This gives you the possibility to perform several builds in one operation. If necessary, you can also specify pre-build and post-build actions.

In addition to using the IAR Embedded Workbench IDE to build projects, you can also use the command line utility `iarbuild.exe`.

For examples of building application and library projects, see the tutorials in the Information Center. For further information about building library projects, see the *IAR C/C++ Development Guide for ARM®* .

SETTING OPTIONS

To specify how your application should be built, you must define one or several build configurations. Every build configuration has its own settings, which are independent of the other configurations. All settings are indicated in a separate column in the Workspace window.

For example, a configuration that is used for debugging would not be highly optimized, and would produce output that suits the debugging. Conversely, a configuration for building the final application would be highly optimized, and produce output that suits a flash or PROM programmer.

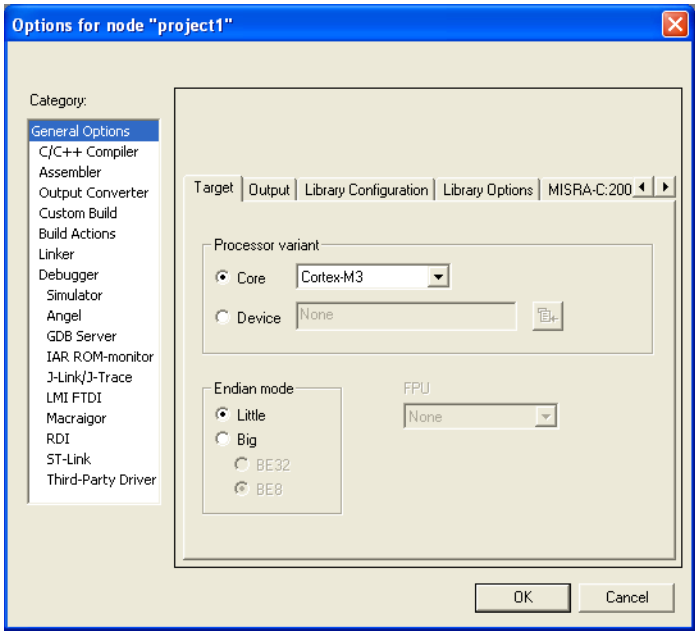
For each build configuration, you can set options on the project level, group level, and file level. Many options can only be set on the project level because they affect the entire build configuration. Examples of such options are General Options (for example, processor variant and library object file), linker settings, and debug settings. Other options, such as compiler and assembler options, that you set on project level are default for the entire build configuration.

To override project level settings, select the required item—for instance a specific group of files—and then select the option **Override inherited settings**. The new settings will affect all members of that group, that is, files and any groups of files. To restore all settings to the default factory settings, click the **Factory Settings** button.

Note: There is one important restriction on setting options. If you set an option on group or file level (group or file level override), no options on higher levels that operate on files will affect that group or file.

Using the Options dialog box

The **Options** dialog box—available by choosing **Project>Options**—provides options for the build tools. You set these options for the selected item in the Workspace window. Options in the **General Options**, **Linker**, and **Debugger** categories can only be set for the entire build configuration, and not for individual groups and files. However, the options in the other categories can be set for the entire build configuration, a group of files, or an individual file.


Options for node "project1"

Category:

- General Options
- C/C++ Compiler
- Assembler
- Output Converter
- Custom Build
- Build Actions
- Linker
- Debugger
  - Simulator
  - Angel
  - GDB Server
  - IAR ROM-monitor
  - J-Link/J-Trace
  - LMI FTDI
  - Macraigor
  - RDI
  - ST-Link
  - Third-Party Driver

Target | Output | Library Configuration | Library Options | MISRA-C:200

Processor variant: Core [✓] Cortex-M3; Device: None

Endian mode: Little [✓]; Big; BE32; BE8

FPU: None

OK | Cancel

Figure 7: General options

The **Category** list allows you to select which building tool to set options for. Which tools that are available in the **Category** list depends on which tools are included in your product. If you select **Library** as output file on the **Output** page, **Linker** is replaced by

Library Builder in the category list. When you select a category, one or more pages containing options for that component are displayed.

Click the tab corresponding to the type of options you want to view or change. To restore all settings to the default factory settings, click the **Factory Settings** button, which is available for all categories except **General Options** and **Custom Build**. Note that two sets of factory settings are available: Debug and Release. Which one that is used depends on your build configuration; see *New Configuration dialog box*, page 114.

For information about each option and how to set options, see the chapters:

- *General options*
- *Compiler options*
- *Assembler options*
- *Output converter options*
- *Custom build options*
- *Build actions options*
- *Linker options*
- *Library builder options*
- *C-SPY driver options* (see the *C-SPY® Debugging Guide for ARM®).*

Note: If you add to your project a source file with a non-recognized filename extension, you cannot set options on that source file. However, you can add support for additional filename extensions. For reference information, see *Filename Extensions dialog box*, page 146.

BUILDING A PROJECT

You have access to the build commands both from the **Project** menu and from the context menu that appears if you right-click an item in the Workspace window.

The three build commands **Make**, **Compile**, and **Rebuild All** run in the background, so you can continue editing or working with the IDE while your project is being built.

For further reference information, see *Project menu*, page 109.

BUILDING MULTIPLE CONFIGURATIONS IN A BATCH

Use the batch build feature when you want to build more than one configuration at once. A batch is an ordered list of build configurations. The **Batch Build** dialog box—available from the **Project** menu—lets you create, modify, and build batches of configurations.

For workspaces that contain several configurations, it is convenient to define one or more different batches. Instead of building the entire workspace, you can build only the appropriate build configurations, for instance Release or Debug configurations.

For detailed information about the **Batch Build** dialog box, see *Batch Build dialog box*, page 119.

USING PRE- AND POST-BUILD ACTIONS

If necessary, you can specify pre-build and post-build actions that you want to occur before or after the build. The **Build Actions** dialog box—available from the **Project** menu—lets you specify the actions required.

For detailed information about the **Build Actions** dialog box, see *Build actions options*, page 189.

Using pre-build actions for time stamping

You can use pre-build actions to embed a time stamp for the build in the resulting binary file. Follow these steps:

1. Create a dedicated time stamp file, for example, `timestamp.c` and add it to your project.
2. In this source file, use the preprocessor macros `__TIME__` and `__DATE__` to initialize a string variable.
3. Choose **Project>Options>Build Actions** to open the **Build Actions** dialog box.
4. In the **Pre-build command line** text field, specify for example this pre-build action:

   ```
   "touch $PROJ_DIR$\timestamp.c"
   ```

   You can use the open source command line utility `touch` for this purpose or any other suitable utility which updates the modification time of the source file.
5. If the project is not entirely up-to-date, the next time you use the **Make** command, the pre-build action will be invoked before the regular build process. Then the regular build process must always recompile `timestamp.c` and the correct timestamp will end up in the binary file.

   If the project already is up-to-date, the pre-build action will not be invoked. This means that nothing is built, and the binary file still contains the timestamp for when it was last built.

CORRECTING ERRORS FOUND DURING BUILD

The compiler, assembler, and debugger are fully integrated with the development environment. If your source code contains errors, you can jump directly to the correct position in the appropriate source file by double-clicking the error message in the error listing in the Build message window, or selecting the error and pressing Enter.

After you have resolved any problems reported during the build process and rebuilt the project, you can directly start debugging the resulting code at the source level.

To specify the level of output to the Build message window, choose **Tools>Options** to open the **IDE Options** dialog box. Click the **Messages** tab and select the level of output in the **Show build messages** drop-down list. Alternatively, you can right-click in the **Build Messages** window and select **Options** from the context menu.

For reference information about the Build messages window, see *Build window*, page 91.

BUILDING FROM THE COMMAND LINE

To build the project from the command line, use the IAR Command Line Build Utility (`iarbuild.exe`) located in the `common\bin` directory. As input you use the project file, and the invocation syntax is:

```
iarbuild project.ewp [-clean|-build|-make] <configuration>
[-log errors|warnings|info|all]
```

Parameter	Description
`project``.ewp`	Your IAR Embedded Workbench project file.
`-clean`	Removes any intermediate and output files.
`-build`	Rebuilds and relinks all files in the current build configuration.
`-make`	Brings the current build configuration up to date by compiling, assembling, and linking only the files that have changed since the last build.
`configuration`	The name of the configuration you want to build, which can either be one of the predefined configurations Debug or Release, or a name that you define yourself. For more information about build configurations, see *Projects and build configurations*, page 31.
`-log errors`	Displays build error messages.
`-log warnings`	Displays build warning and error messages.
`-log info`	Displays build warning and error messages, and messages issued by the `#pragma message` preprocessor directive.
`-log all`	Displays all messages generated from the build, for example compiler sign-on information and the full command line.

Table 3: iarbuild.exe command line options

If you run the application from a command shell without specifying a project file, you will get a sign-on message describing available parameters and their syntax.

Extending the toolchain

IAR Embedded Workbench provides a feature—Custom Build—which lets you extend the standard toolchain. This feature is used for executing external tools (not provided by IAR Systems). You can make these tools execute each time specific files in your project have changed.

If you specify custom build options on the **Custom tool configuration** page, the build commands treat the external tool and its associated files in the same way as the standard tools within the IAR Embedded Workbench IDE and their associated files. The relation between the external tool and its input files and generated output files is similar to the relation between the C/C++ Compiler, `c` files, `h` files, and `o` files. See *Custom build options*, page 187, for details about available custom build options.

You specify filename extensions of the files used as input to the external tool. If the input file has changed since you last built your project, the external tool is executed; just as the compiler executes if a `c` file has changed. In the same way, any changes in additional input files (for instance include files) are detected.

You must specify the name of the external tool. You can also specify any necessary command line options needed by the external tool, and the name of the output files generated by the external tool. Note that you can use argument variables for substituting file paths.

For some of the file information, you can use argument variables.

You can specify custom build options to any level in the project tree. The options you specify are inherited by any sublevel in the project tree.

TOOLS THAT CAN BE ADDED TO THE TOOLCHAIN

Some examples of external tools, or types of tools, that you can add to the IAR Embedded Workbench toolchain are:

- Tools that generate files from a specification, such as Lex and YACC
- Tools that convert binary files—for example files that contain bitmap images or audio data—to a table of data in an assembler or C source file. This data can then be compiled and linked together with the rest of your application.

ADDING AN EXTERNAL TOOL

The following example demonstrates how to add the tool *Flex* to the toolchain. The same procedure can be used also for other tools.

In the example, Flex takes the file `myFile.lex` as input. The two files `myFile.c` and `myFile.h` are generated as output.

1 Add the file you want to work with to your project, for example `myFile.lex`.

2 Select this file in the workspace window and choose **Project>Options**. Select **Custom Build** from the list of categories.

3 In the **Filename extensions** field, type the filename extension `.lex`. Remember to specify the leading period (`.`).

4 In the **Command line** field, type the command line for executing the external tool, for example

```
flex $FILE_PATH$ -o$FILE_BNAME$.c
```

During the build process, this command line is expanded to:

```
flex myFile.lex -omyFile.c
```

Note the usage of *argument variables*. Note specifically the use of `$FILE_BNAME$` which gives the base name of the input file, in this example appended with the `c` extension to provide a C source file in the same directory as the input file `foo.lex`. For further details of these variables, see *Argument variables*, page 117.

5 In the **Output files** field, describe the output files that are relevant for the build. In this example, the tool Flex would generate two files—one source file and one header file. The text in the **Output files** text box for these two files would look like this:

```
$FILE_BPATH$.c
$FILE_BPATH$.h
```

6 If the external tool uses any additional files during the build, these should be added in the **Additional input files** field, for instance:

```
$TOOLKIT_DIR$\inc\stdio.h
```

This is important, because if the dependency files change, the conditions will no longer be the same and the need for a rebuild is detected.

7 Click **OK**.

8 To build your application, choose **Project>Make**.

Editing

This chapter describes in detail how to use the IAR Embedded Workbench editor. The final section describes how to customize the editor and how to use an external editor of your choice.

Using the IAR Embedded Workbench editor

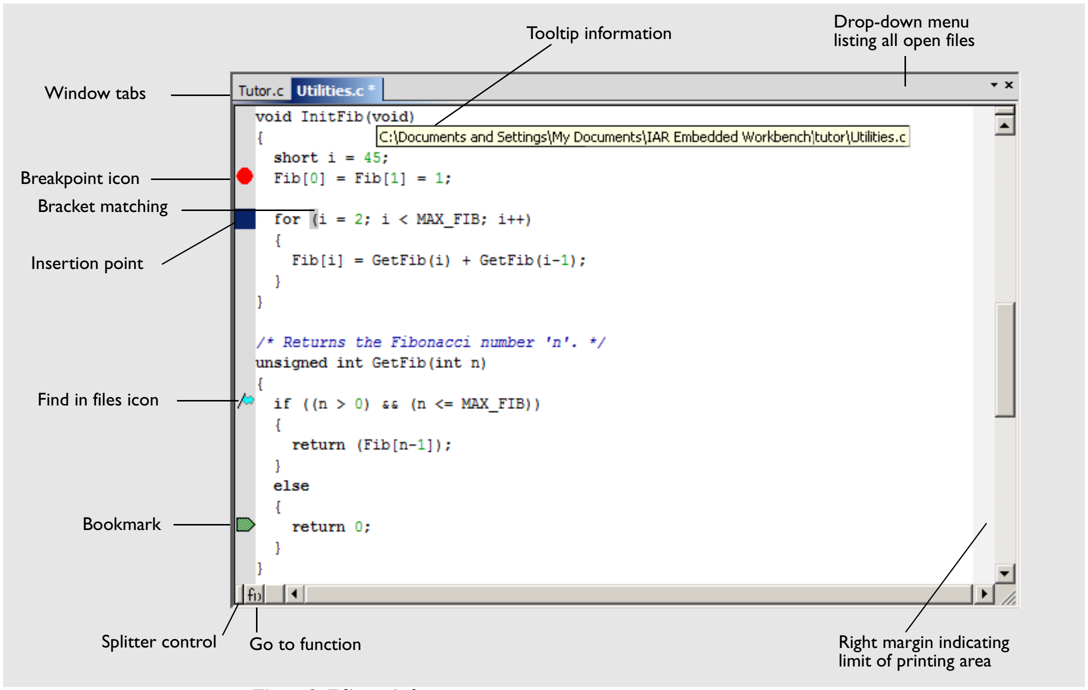
The integrated text editor allows editing of multiple files in parallel, and provides all basic editing features expected from a modern editor. In addition, it provides functions specific to software development, like coloring of keywords (C/C++, assembler, and user-defined), block indent, and function navigation within source files. It also recognizes C language elements like matching brackets. This list shows some additional features:

- Context-sensitive help system that can display reference information for DLIB library functions
- Syntax of C or C++ programs and assembler directives shown using text styles and colors
- Powerful search and replace commands, including multi-file search
- Direct jump to context from error listing
- Multibyte character support
- Parenthesis matching
- Automatic completion and indentation
- Bookmarks
- Unlimited undo and redo for each window.

EDITING A FILE

The editor window is where you write, view, and modify your source code. You can open one or several text files, either from the **File** menu, or by double-clicking a file in the Workspace window. If you open several files, they are organized in a *tab group*. Several editor windows can be open at the same time.

Click the tab for the file that you want to display. All open files are also available from the drop-down menu at the upper right corner of the editor window.



```
void InitFib(void)
{
  short i = 45;
  Fib[0] = Fib[1] = 1;

  for (i = 2; i < MAX_FIB; i++)
  {
    Fib[i] = GetFib(i) + GetFib(i-1);
  }
}

/* Returns the Fibonacci number 'n'. */
unsigned int GetFib(int n)
{
  if ((n > 0) && (n <= MAX_FIB))
  {
    return (Fib[n-1]);
  }
  else
  {
    return 0;
  }
}
```

C:\Documents and Settings\My Documents\IAR Embedded Workbench\tutor\Utilities.c

Figure 8: Editor window

The name of the open source file is displayed on the tab. If a file is read-only, a padlock is visible at the bottom left corner of the editor window. If a file has been modified after it was last saved, an asterisk appears on the tab after the filename, for example `Utilities.c *`.

The commands on the **Window** menu allow you to split the editor window into panes. On the **Window** menu you also find commands for opening multiple editor windows, and commands for moving files between editor windows. For reference information about each command on the menu, see *Window menu*, page 150. For reference information about the editor window, see *Editor window*, page 81.

Note: When you want to print a source file, it can be useful to enable the option **Show line numbers**—available by choosing **Tools>Options>Editor**.

Accessing reference information for DLIB library functions

When you need to know the syntax for any library function, select the function name in the editor window and press F1. The library documentation for the selected function appears in a help window.

Using and customizing editor commands and shortcut keys

The **Edit** menu provides commands for editing and searching in editor windows, for instance, unlimited undo/redo (the **Edit>Undo** and **Edit>Redo** commands, respectively). You can also find some of these commands on the context menu that appears when you right-click in the editor window. For reference information about each command, see *Edit menu*, page 97.

There are also editor shortcut keys for:

- moving the insertion point
- scrolling text
- selecting text.

For detailed information about these shortcut keys, see *Editor shortcut key summary*, page 86.

To change the default shortcut key bindings, choose **Tools>Options**, and click the **Key Bindings** tab. For further details, see *Key Bindings options*, page 123.

Splitting the editor window into panes

You can split the editor window horizontally or vertically into multiple panes, to look at different parts of the same source file at once, or to move text between two different panes.

To split the window, double-click the appropriate splitter bar, or drag it to the middle of the window. Alternatively, you can split a window into panes using the **Window>Split** command.

To revert to a single pane, double-click the splitter control or drag it back to the end of the scroll bar.

Dragging and dropping of text

You can easily move text within an editor window or between editor windows. Select the text and drag it to the new location.

Syntax coloring

If the **Tools>Options>Editor>Syntax highlighting** option is enabled, the IAR Embedded Workbench editor automatically recognizes the syntax of:

- C and C++ keywords
- C and C++ comments
- Assembler directives and comments
- Preprocessor directives
- Strings.

The different parts of source code are displayed in different text styles.

To change these styles, choose **Tools>Options**, and use the **Editor>Colors and Fonts** options. For additional information, see *Editor Colors and Fonts options*, page 131.

In addition, you can define your own set of keywords that should be syntax-colored automatically:

1. In a text file, list all the keywords that you want to be automatically syntax-colored. Separate each keyword with either a space or a new line.
2. Choose **Tools>Options** and select **Editor>Setup Files**.
3. Select the **Use Custom Keyword File** option and specify your newly created text file. A browse button is available for your convenience.
4. Select **Editor>Colors and Fonts** and choose **User Keyword** from the **Syntax Coloring** list. Specify the font, color, and type style of your choice. For additional information, see *Editor Colors and Fonts options*, page 131.
5. In the editor window, type any of the keywords you listed in your keyword file; see how the keyword is syntax-colored according to your specification.

Automatic text indentation

The text editor can perform various kinds of indentation. For assembler source files and normal text files, the editor automatically indents a line to match the previous line. If you want to indent several lines, select the lines and press the Tab key. Press Shift+Tab to move a whole block of lines to the left.

For C/C++ source files, the editor indents lines according to the syntax of the C/C++ source code. This is performed whenever you:

- Press the Return key
- Type any of the special characters `{`, `}`, `:`, and `#`
- Have selected one or several lines, and choose the **Edit>Auto Indent** command.

To enable or disable the indentation:

1 Choose **Tools>Options** and select **Editor**.

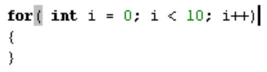
2 Select or deselect the **Auto indent** option.

To customize the C/C++ automatic indentation, click the **Configure** button.

For additional information, see *Configure Auto Indent dialog box*, page 127.

Matching brackets and parentheses

When the insertion point is located next to a parenthesis, the matching parenthesis is highlighted with a light gray color:



```
for( int i = 0; i < 10; i++)
{
}
```

Figure 9: Parenthesis matching in editor window

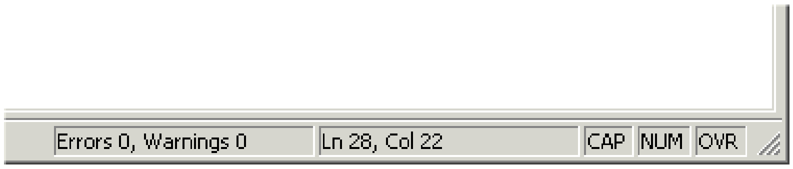
The highlight remains in place as long as the insertion point is located next to the parenthesis.

To select all text between the brackets surrounding the insertion point, choose **Edit>Match Brackets**. Every time you choose **Match Brackets** after that, the selection will increase to the next hierarchic pair of brackets.

Note: Both of these functions—automatic matching of corresponding parentheses and selection of text between brackets—apply to `()`, `[]`, and `{}`.

Displaying status information

As you are editing, the status bar—available by choosing **View>Status Bar**— shows the current line and column number containing the insertion point, and the Caps Lock, Num Lock, and Overwrite status:



Errors 0, Warnings 0 | Ln 28, Col 22 | CAP | NUM | OVR

Figure 10: Editor window status bar

USING AND ADDING CODE TEMPLATES

Code templates is a method for conveniently inserting frequently used source code sequences, for example `for` loops and `if` statements. The code templates are defined in a normal text file. By default, a few example templates are provided. In addition, you can easily add your own code templates.

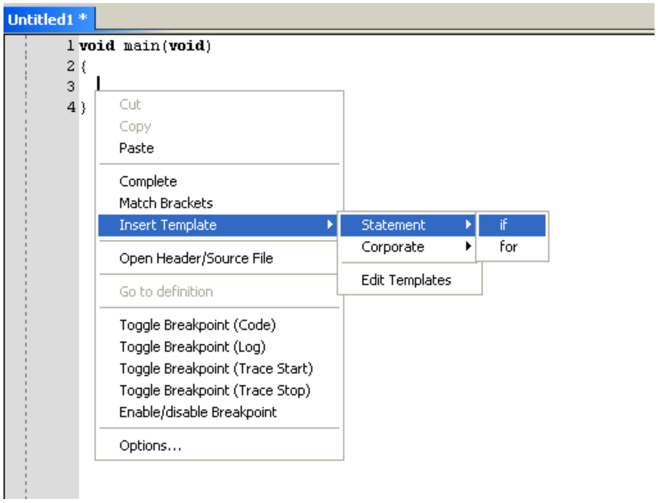
Enabling code templates

By default, code templates are enabled. To enable and disable the use of code templates:

1. Choose **Tools>Options**.
2. Go to the **Editor Setup Files** page.
3. Select or deselect the **Use Code Templates** option.
4. In the text field, specify which template file you want to use; either the default file or one of your own template files. A browse button is available for your convenience.

Inserting a code template into your source code

To insert a code template into your source code, place the insertion point at the location where you want the template to be inserted, right-click, and choose **Insert Template** and the appropriate code template from the menu that appears.



```
void main(void)
{

}
```

Figure 11: Inserting a code template

If the code template you choose requires any type of field input, as in the `for` loop example which needs an end value and a count variable, an input dialog box appears.

Adding your own code templates

The source code templates are defined in a normal text file. The original template file `CodeTemplates.txt` is located in the `common\config` installation directory. The first time you use IAR Embedded Workbench, the original template file is copied to a directory for local settings, and this is the file that is used by default if code templates are enabled. To use your own template file, follow the procedure described in *Enabling code templates*, page 54.

To open the template file and define your own code templates, choose **Edit>Code Templates>Edit Templates**.

The syntax for defining templates is described in the default template file.

Selecting the correct language version of the code template file

When you start the IAR Embedded Workbench IDE for the very first time, you are asked to select a language version. This only applies if you are using an IDE that is available in other languages than English.

Selecting a language creates a corresponding language version of the default code template file in the `Application Data\IAR Embedded Workbench` subdirectory of the current Windows user (for example `CodeTemplates.ENU.txt` for English and `CodeTemplates.JPN.txt` for Japanese). The default code template file does not change automatically if you change the language version of the IDE afterwards.

To change the code template:

1. Choose **Tools>Options>IDE Options>Editor>Setup Files**.
2. Click the browse button of the **Use Code Templates** option and select a different template file.

   If the code template file you want to select is not in the browsed directory, you must:
3. Delete the file name in the **Use Code Templates** text box.
4. Deselect the **Use Code Templates** option and click OK.
5. Restart the IAR Embedded Workbench IDE.
6. Then choose **Tools>Options>IDE Options>Editor>Setup Files** again.

   The default code template file for the selected language version of the IDE should now be displayed in the **Use Code Templates** text box. Select the check box to enable the template.

NAVIGATING IN AND BETWEEN FILES

The editor provides several functions for easy navigation within the files and between files:

- Switching between source and header files

  If the insertion point is located on an `#include` line, you can choose the **Open "header.h"** command from the context menu, which opens the header file in an editor window. You can also choose the command **Open Header/Source File**, which opens the header or source file that corresponds to the current file, or activates it if it is already open. This command is available if the insertion point is located on any line except an `#include` line.
- Function navigation

  Click the **Go to function** button in the bottom left corner in an editor window to list all functions defined in the source file displayed in the window. You can then choose to go directly to one of the functions by double-clicking it in the list.
- Adding bookmarks

  Use the **Edit>Navigate>Toggle Bookmark** command to add and remove bookmarks. To switch between the marked locations, choose **Edit>Navigate>Go to Bookmark**.

SEARCHING

There are several standard search functions available in the editor:

- **Quick search** text box
- **Find** dialog box
- **Replace** dialog box
- **Find in files** dialog box
- **Incremental Search** dialog box.

To use the Quick search text box on the toolbar:

1 Type the text you want to search for and press Enter.

2 Press Esc to stop the search. This is a quick method for searching for text in the active editor window.

To use the Find, Replace, Find in Files, and Incremental Search functions:

1 Before you use the search commands, choose **Tools>Options>Editor** and make sure the **Show bookmarks** option is selected.

2 Choose the appropriate search command from the **Edit** menu. For reference information about each search function, see *Edit menu*, page 97.

3 To remove the blue flag icons that have appeared in the left-hand margin, right-click in the Find in Files window and choose **Clear All** from the context menu.

Customizing the editor environment

The IDE editor can be configured on the IDE Options pages **Editor** and **Editor Colors and Fonts**. Choose **Tools>Options** to access the pages.

For details about these pages, see *Tools menu*, page 121.

USING AN EXTERNAL EDITOR

The **External Editor** options—available by choosing **Tools>Options>Editor**—let you specify an external editor of your choice.

Note: While debugging using C-SPY, C-SPY will not use the external editor for displaying the current debug state. Instead, the built-in editor will be used.

To specify an external editor of your choice, follow this procedure:

1 Select the option **Use External Editor**.

2 An external editor can be called in one of two ways, using the **Type** drop-down menu.

Command Line calls the external editor by passing command line parameters.

DDE calls the external editor by using DDE (Windows Dynamic Data Exchange).

3 If you use the command line, specify the command line to pass to the editor, that is, the name of the editor and its path, for instance:

```
C:\Windows\NOTEPAD.EXE.
```

To send an argument to the external editor, type the argument in the **Arguments** field. For example, type `$FILE_PATH$` to start the editor with the active file (in editor, project, or Messages window).

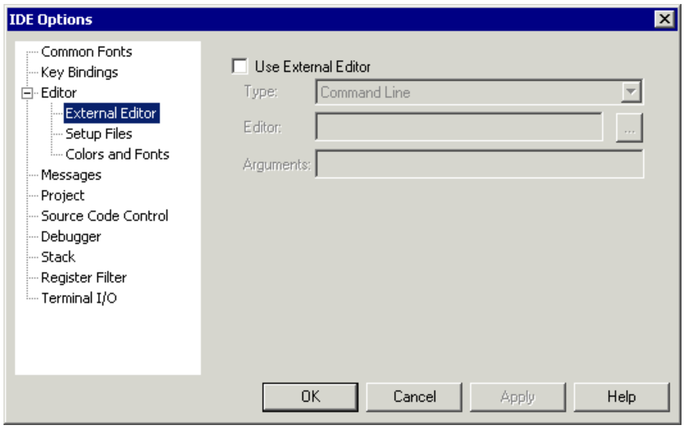


Figure 12: Specifying an external command line editor

Note: Options for Register Filter and Terminal I/O are only available when the C-SPY debugger is running.

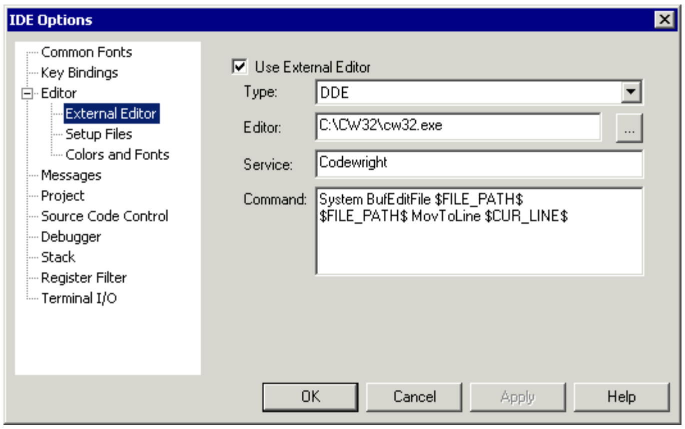
4 If you use DDE, specify the editor's DDE service name in the **Service** field. In the **Command** field, specify a sequence of command strings to send to the editor.

The service name and command strings depend on the external editor that you are using. Refer to the user documentation of your external editor to find the appropriate settings.

The command strings should be entered as:

```
DDE-Topic CommandString1
DDE-Topic CommandString2
```

as in this example, which applies to Codewright®:



☑ Use External Editor

Type: DDE

Editor: C:\CW32\cw32.exe

Service: Codewright

Command:
```
System BufEditFile $FILE_PATH$
$FILE_PATH$ MovToLine $CUR_LINE$
```

Figure 13: External editor DDE settings

The command strings used in this example will open the external editor with a dedicated file activated. The cursor will be located on the current line as defined in the context from where the file is open, for instance when searching for a string in a file, or when double-clicking an error message in the Message window.

5 Click **OK**.

When you double-click a file in the Workspace window, the file is opened by the external editor.

Variables can be used in the arguments. For more information about the argument variables that are available, see *Argument variables*, page 117.

Part 2. Reference information

This part of the IDE Project Management and Building Guide contains these chapters:

- Installed files
- IAR Embedded Workbench IDE reference
- General options
- Compiler options
- Assembler options
- Output converter options
- Custom build options
- Build actions options
- Linker options
- Library builder options.

Installed files

This chapter describes which directories that are created during installation and which file types that are used.

Directory structure

The installation procedure creates several directories to contain the various types of files used with the IAR Systems development tools. The following sections give a description of the files contained by default in each directory.

ROOT DIRECTORY

The root directory created by the default installation procedure is the `x:\Program Files\IAR Systems\Embedded Workbench 6.n\` directory where *x* is the drive where Microsoft Windows is installed and `6.n` is the version number of the IDE.

THE ARM DIRECTORY

The `arm` directory contains all product-specific subdirectories.

Directory	Description
`arm\bin`	The `arm\bin` subdirectory contains executable files for ARM-specific components, such as the compiler, the assembler, the linker and the library tools, and the C-SPY® drivers.
`arm\config`	The `arm\config` subdirectory contains files used for configuring the development environment and projects, for example: • Linker configuration files (`*.icf`) • C-SPY device description files (`*.ddf`) • Device selection files (`*.i79`, `*.menu`) • Flash loader applications for various devices (`*.out`) • Syntax coloring configuration files (`*.cfg`) • Project templates for both application and library projects (`*.ewp`), and for the library projects, the corresponding library configuration files.
`arm\doc`	The `arm\doc` subdirectory contains release notes with recent additional information about the ARM tools. We recommend that you read all of these files. The directory also contains online versions in hypertext PDF format of this user guide, and of the ARM reference guides, as well as online help files (`*.chm`).

Table 4: The ARM directory

Directory	Description
`arm\drivers`	The `arm\drivers` subdirectory contains low-level device drivers, typically USB drivers required by the C-SPY drivers.
`arm\examples`	The `arm\examples` subdirectory contains files related to example projects, which can be opened from the Information Center.
`arm\inc`	The `arm\inc` subdirectory holds include files, such as the header files for the standard C or C++ library. There are also specific header files that define special function registers (SFRs); these files are used by both the compiler and the assembler.
`arm\lib`	The `arm\lib` subdirectory holds prebuilt libraries and the corresponding library configuration files, used by the compiler.
`arm\plugins`	The `arm\plugins` subdirectory contains executable files and description files for components that can be loaded as plugin modules.
`arm\powerpac`	The `arm\powerpac` subdirectory contains files related to the add-on product IAR PowerPac.
`arm\src`	The `arm\src` subdirectory holds source files for some configurable library functions. This directory also holds the library source code and the source code for ELF utilities.
`arm\tutor`	The `arm\tutor` subdirectory contains the files used for the tutorials in this guide.

Table 4: The ARM directory (Continued)

THE COMMON DIRECTORY

The `common` directory contains subdirectories for components shared by all IAR Embedded Workbench products.

Directory	Description
`common\bin`	The `common\bin` subdirectory contains executable files for components common to all IAR Embedded Workbench products, such asthe editor and the graphical user interface components. The executable file for the IDE is also located here.
`common\config`	The `common\config` subdirectory contains files used by the IDE for settings in the development environment.
`common\doc`	The `common\doc` subdirectory contains release notes with recent additional information about the components common to all IAR Embedded Workbench products, such as the linker and library tools. We recommend that you read these files.The directory also contains documentation related to installation and licensing, and getting started using IAR Embedded Workbench.

Table 5: The common directory

Directory	Description
`common\plugins`	The `common\plugins` subdirectory contains executable files and description files for components that can be loaded as plugin modules, for example modules for Code coverage and Profiling.

Table 5: The common directory (Continued)

THE INSTALL-INFO DIRECTORY

The `install-info` directory contains metadata (version number, name, etc.) about the installed product components. Do not modify these files.

File types

The ARM versions of the IAR Systems development tools use the following default filename extensions to identify the produced files and other recognized file types:

Ext.	Type of file	Output from	Input to
`asm`	Assembler source code	Text editor	Assembler
`bat`	Windows command batch file	C-SPY	Windows
`c`	C source code	Text editor	Compiler
`cfg`	Syntax coloring configuration	--	IDE
`chm`	Online help system file	--	IDE
`cpp`	C++ source code	Text editor	Compiler
`dat`	Macros for formatting of STL containers	IDE	IDE
`dbgdt`	Debugger desktop settings	C-SPY	C-SPY
`ddf`	Device description file	--	C-SPY
`dep`	Dependency information	IDE	IDE
`dni`	Debugger initialization file	C-SPY	C-SPY
`ewd`	Project settings for C-SPY	IDE	IDE
`ewp`	IAR Embedded Workbench project (current version)	IDE	IDE
`ewplugin`	IDE description file for plugin modules	--	IDE
`eww`	Workspace file	IDE	IDE
`flash`	Flash memory description file	Text editor	C-SPY
`flashdict`	Container for `flash` files	Text editor	C-SPY
`fmt`	Formatting information for the Locals and Watch windows	IDE	IDE

Table 6: File types

Ext.	Type of file	Output from	Input to
h	C/C++ or assembler header source	Text editor	Compiler or assembler #include
helpfiles	Help menu configuration file	--	IDE
html, htm	HTML document	Text editor	IDE
i	Preprocessed source	Compiler	Compiler
i79	Device selection file	--	IDE
icf	Linker configuration file	Text editor	ILINK linker
inc	Assembler header source	Text editor	Assembler #include
ini	Project configuration	IDE	–
log	Log information	IDE	–
lst	List output	Compiler and assembler	–
mac	C-SPY macro definition	Text editor	C-SPY
menu	Device selection file	Text editor	IDE
o	Object module	Compiler and assembler	ILINK
o	Library	iarchive	ILINK
out	Target application	ILINK	EPROM, C-SPY, etc.
out	Target application with debug information	ILINK	C-SPY and other symbolic debuggers
pbd	Source browse information	IDE	IDE
pbi	Source browse information	IDE	IDE
pew	IAR Embedded Workbench project (old project format)	IDE	IDE
prj	IAR Embedded Workbench project (old project format)	IDE	IDE
s	ARM assembler source code	Text editor	Assembler
svd	CMSIS System View Description	--	C-SPY
vsp	visualSTATE project files	IAR visualSTATE Designer	IAR visualSTATE Designer and IAR Embedded Workbench IDE

Table 6: File types (Continued)

Ext.	Type of file	Output from	Input to
`wsdt`	Workspace desktop settings	IDE	IDE
`xcl`	Extended command line	Text editor	Assembler, compiler, linker

Table 6: File types (Continued)

When you run the IDE, some files are created and located in dedicated directories under your project directory, by default `$PROJ_DIR$\Debug`, `$PROJ_DIR$\Release`, `$PROJ_DIR$\settings`, and the file `*.dep` under the installation directory. None of these directories or files affect the execution of the IDE, which means you can safely remove them if required.

FILES WITH NON-DEFAULT FILENAME EXTENSIONS

In the IDE you can increase the number of recognized filename extensions using the **Filename Extensions** dialog box, available from the **Tools** menu. You can also connect your filename extension to a specific tool in the toolchain. See *Filename Extensions dialog box*, page 146.

To override the default filename extension from the command line, include an explicit extension when you specify a filename.

IAR Embedded Workbench IDE reference

This chapter contains reference information about the windows, menus, menu commands, and the corresponding components that are found in the IDE. This chapter contains the following sections:

- *Windows*, page 69
- *Menus*, page 95.

The IDE is a modular application. Which menus are available depends on which components are installed.

Windows

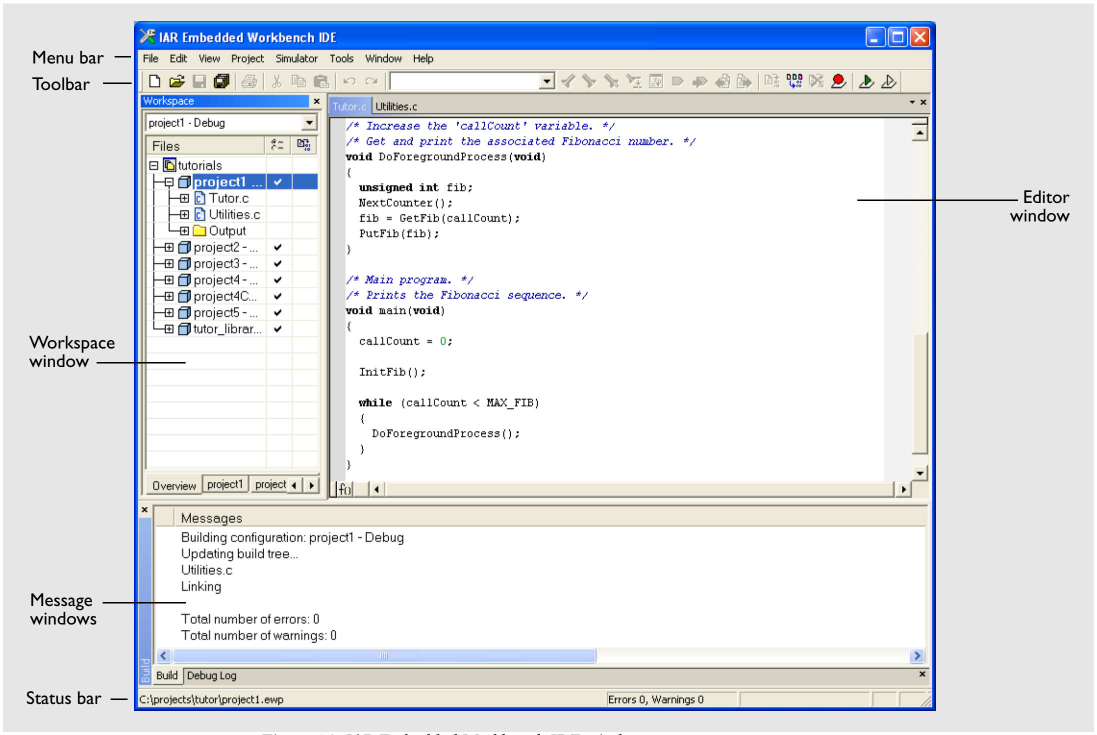
The available windows are:

- IAR Embedded Workbench IDE window
- Workspace window
- Editor window
- Source Browser window
- Message windows.

In addition, a set of C-SPY®-specific windows becomes available when you start the debugger. For reference information about these windows, see the *C-SPY® Debugging Guide for ARM®*.

IAR Embedded Workbench IDE window

The main window of the IDE is displayed when you launch the IDE.



```
/* Increase the 'callCount' variable. */
/* Get and print the associated Fibonacci number. */
void DoForegroundProcess(void)
{
  unsigned int fib;
  NextCounter();
  fib = GetFib(callCount);
  PutFib(fib);
}

/* Main program. */
/* Prints the Fibonacci sequence. */
void main(void)
{
  callCount = 0;

  InitFib();

  while (callCount < MAX_FIB)
  {
    DoForegroundProcess();
  }
}
```

```
Messages
Building configuration: project1 - Debug
Updating build tree...
Utilities.c
Linking

Total number of errors: 0
Total number of warnings: 0
```

```
C:\projects\tutor\project1.ewp          Errors 0, Warnings 0
```

Figure 14: IAR Embedded Workbench IDE window

The figure shows the window and its various components. The window might look different depending on which plugin modules you are using.

Menu bar

The menu bar contains:

File	Commands for opening source and project files, saving and printing, and exiting from the IDE.
Edit	Commands for editing and searching in editor windows and for enabling and disabling breakpoints in C-SPY.

View	Commands for opening windows and controlling which toolbars to display.
Project	Commands for adding files to a project, creating groups, and running the IAR Systems tools on the current project.
Tools	User-configurable menu to which you can add tools for use with the IDE.
Window	Commands for manipulating the IDE windows and changing their arrangement on the screen.
Help	Commands that provide help about the IDE.

For reference information about each menu, see *Menus*, page 95.

Toolbar

The IDE toolbar—available from the **View** menu—provides buttons for the most useful commands on the IDE menus, and a text box for typing a string to do a quick search.

For a description of any button, point to it with the mouse button. When a command is not available, the corresponding toolbar button is dimmed, and you will not be able to click it.

This figure shows the menu commands corresponding to each of the toolbar buttons:

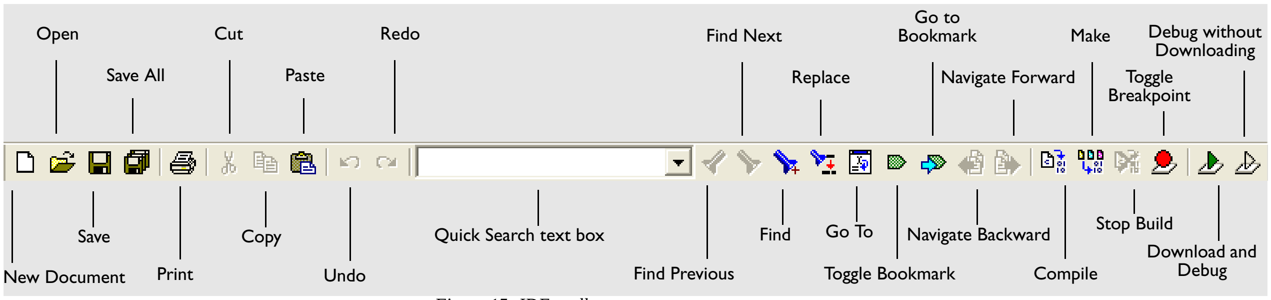


Figure 15: IDE toolbar

Note: When you start C-SPY, the **Download and Debug** button will change to a **Make and Debug** button and the **Debug without Downloading** will change to a **Restart Debugger** button.

Status bar

The status bar at the bottom of the window displays the number of errors and warnings generated during a build, the position of the insertion point in the editor window, and the state of the modifier keys. The status bar can be enabled from the **View** menu.

As you are editing, the status bar shows the current line and column number containing the insertion point, and the Caps Lock, Num Lock, and Overwrite status. A flag in the corner shows the language version you are using. Click the flag to change the language the next time you launch the IDE.

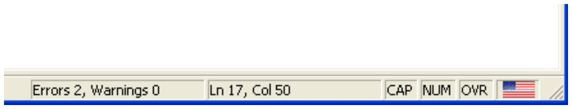


Figure 16: IAR Embedded Workbench IDE window status bar

Workspace window

The Workspace window is available from the View menu.

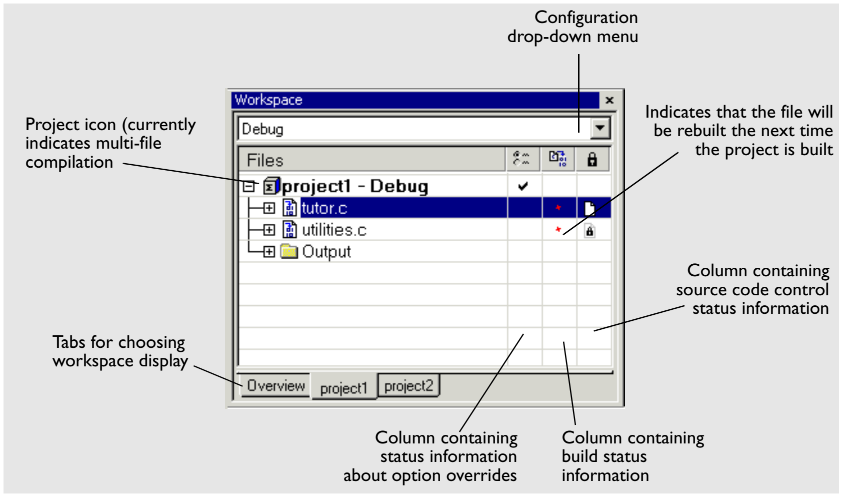

Configuration drop-down menu
Workspace
Debug
Files
project1 - Debug
tutor.c
utilities.c
Output
Overview
project1
project2
Project icon (currently indicates multi-file compilation
Indicates that the file will be rebuilt the next time the project is built
Column containing source code control status information
Tabs for choosing workspace display
Column containing status information about option overrides
Column containing build status information

Figure 17: Workspace window

Use the Workspace window to access your projects and files during the application development.

Drop-down list

At the top of the window there is a drop-down list where you can choose a build configuration to display in the window for a specific project.

The display area

This area contains four columns.

The **Files** column displays the name of the current workspace and a tree representation of the projects, groups and files included in the workspace. One or more of these icons are displayed:

Workspace

Project

Project with multi-file compilation

Group of files

Group excluded from the build

Group of files, part of multi-file compilation

Group of files, part of multi-file compilation, but excluded from the build

Object file or library

Assembler source file

C source file

C++ source file

Source file excluded from the build

Header file

Text file

HTML text file

Control file, for example the linker configuration file

IDE internal file

Other file

The column that contains status information about option overrides can have one of three icons for each level in the project:

Blank	There are no settings/overrides for this file/group.
Black check mark	There are local settings/overrides for this file/group.

Red check mark	There are local settings/overrides for this file/group, but they are either identical to the inherited settings or they are ignored because you use multi-file compilation, which means that the overrides are not needed.

The column that contains build status information can have one of three icons for each file in the project:

Blank	The file will not be rebuilt next time the project is built.
Red star	The file will be rebuilt next time the project is built.
Gearwheel	The file is being rebuilt.

For details about the various source code control icons, see *Source code control states*, page 77.

Use the tabs at the bottom of the window to choose which project to display. Alternatively, you can choose to display an overview of the entire workspace.

For more information about project management and using the Workspace window, see the chapter *Managing projects*.

Context menu

This context menu is available:

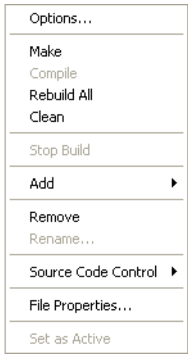


Figure 18: Workspace window context menu

These commands are available:

Options	Displays a dialog box where you can set options for each build tool, for the selected item in the Workspace window. You can set options for the entire project, for a group of files, for on an individual file. See *Setting options*, page 41.
Make	Brings the current target up to date by compiling, assembling, and linking only the files that have changed since the last build.
Compile	Compiles or assembles the currently active file as appropriate. You can choose the file either by selecting it in the Workspace window, or by selecting the editor window containing the file you want to compile.
Rebuild All	Recompiles and relinks all files in the selected build configuration.
Clean	Deletes intermediate files.
Stop Build	Stops the current build operation.
Add>Add Files	Displays a dialog box where you can add files to the project.
Add>Add *filename*	Adds the indicated file to the project. This command is only available if there is an open file in the editor.
Add>Add Group	Displays the **Add Group** dialog box where you can add new groups to the project. For more information about groups, see *Groups*, page 32.
Remove	Removes selected items from the Workspace window.
Rename	Displays the **Rename Group** dialog box where you can rename a group. For more information about groups, see *Groups*, page 32.
Source Code Control	Opens a submenu with commands for source code control, see *Source Code Control menu*, page 76.
File Properties	Displays a standard **File Properties** dialog box for the selected file.
Set as Active	Sets the selected project in the overview display to be the active project. It is the active project that will be built when the **Make** command is executed.

Source Code Control menu

The **Source Code Control** menu is available from the **Project** menu and from the context menu in the Workspace window.

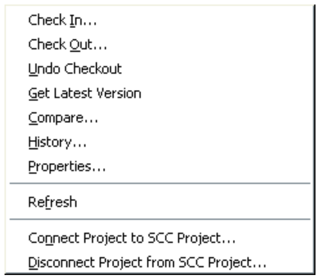


Figure 19: Source Code Control menu

These commands are available:

Check In	Displays the **Check In Files** dialog box where you can check in the selected files; see *Check In Files dialog box*, page 79. Any changes you have made in the files will be stored in the archive. This command is enabled when currently checked-out files are selected in the Workspace window.
Check Out	Checks out the selected file or files. Depending on the SCC (Source Code Control) system you are using, a dialog box might appear; see *Check Out Files dialog box*, page 80. This means you get a local copy of the file(s), which you can edit. This command is enabled when currently checked-in files are selected in the Workspace window.
Undo Checkout	Reverts the selected files to the latest archived version; the files are no longer checked out. Any changes you have made to the files will be lost. This command is enabled when currently checked-out files are selected in the Workspace window.
Get Latest Version	Replaces the selected files with the latest archived version.
Compare	Displays—in an SCC-specific window—the differences between the local version and the most recent archived version.
History	Displays SCC-specific information about the revision history of the selected file.

Properties	Displays information available in the SCC system for the selected file.
Refresh	Updates the SCC display status for all the files that are part of the project. This command is always enabled for all projects under SCC.
Connect Project to SCC Project	Displays a dialog box, which originates from the SCC client application, to let you create a connection between the selected IAR Embedded Workbench project and an SCC project; the IAR Embedded Workbench project will then be an SCC-controlled project. After creating this connection, a special column that contains status information will appear in the Workspace window.
Disconnect Project from SCC Project	Removes the connection between the selected IAR Embedded Workbench project and an SCC project; your project will no longer be a SCC-controlled project. The column in the Workspace window that contains SCC status information will no longer be visible for that project.

For more information about interacting with an external source code control system, see *Source code control*, page 37.

Source code control states

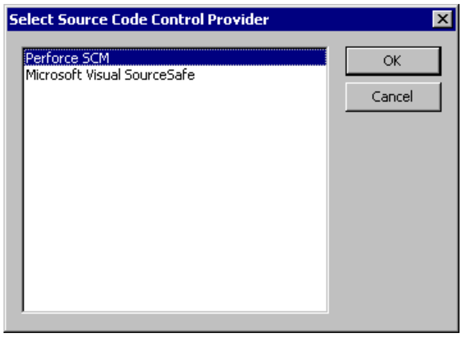
Each source code-controlled file can be in one of several states.

(blank)	Checked out to you. The file is editable.
(checkmark)	Checked out to you. The file is editable and you have modified the file.
(gray padlock)	Checked in. In many SCC systems this means that the file is write-protected.
(gray padlock)	Checked in. A new version is available in the archive.
(red padlock)	Checked out exclusively to another user. In many SCC systems this means that you cannot check out the file.
(red padlock)	Checked out exclusively to another user. A new version is available in the archive. In many SCC systems this means that you cannot check out the file.

Note: The source code control in IAR Embedded Workbench depends on the information provided by the SCC system. If the SCC system provides incorrect or incomplete information about the states, IAR Embedded Workbench might display incorrect symbols.

Select Source Code Control Provider dialog box

The **Select Source Code Control Provider** dialog box is displayed if several SCC systems from different vendors are available.



Perforce SCM
Microsoft Visual SourceSafe

Figure 20: Select Source Code Control Provider dialog box

Use this dialog box to choose the SCC system you want to use.

Check In Files dialog box

The **Check In Files** dialog box is available by choosing the **Project>Source Code Control>Check In** command, alternatively available from the Workspace window context menu.

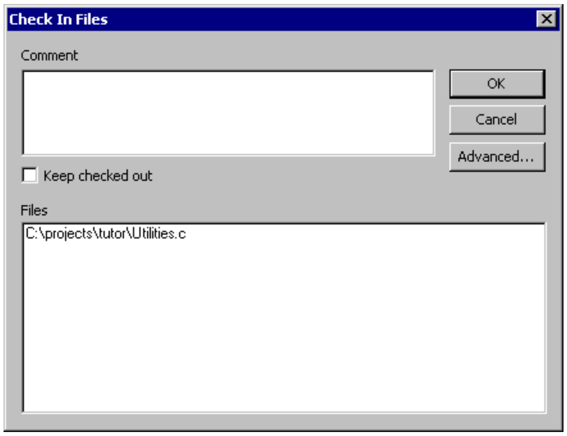


Figure 21: Check In Files dialog box

Comment

Specify a comment—typically a description of your changes—that will be stored in the archive together with the file revision. This text box is only enabled if the SCC system supports adding comments at check in.

Keep checked out

Specifies that the files will continue to be checked out after they have been checked in. Typically, this is useful if you want to make your modifications available to other members in your project team, without stopping your own work with the file.

Advanced

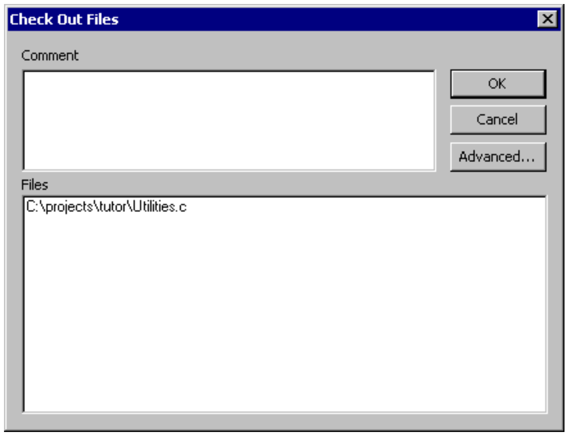
Displays a dialog box, originating from the SCC client application, that contains advanced options. This button is only available if the SCC system supports setting advanced options at check in.

Files

Lists the files that will be checked in. The list will contain all files that were selected in the Workspace window when the **Check In Files** dialog box was opened.

Check Out Files dialog box

The **Check Out Files** dialog box is available by choosing the **Project>Source Code Control>Check Out** command, alternatively available from the Workspace window context menu. However, this dialog box is only available if the SCC system supports adding comments at check out or advanced options.



C:\projects\tutor\Utilities.c

Figure 22: Check Out Files dialog box

Comment

Specify a comment—typically the reason why the file is checked out—that will be placed in the archive together with the file revision. This text box is only enabled if the SCC system supports the adding of comments at check out.

Advanced

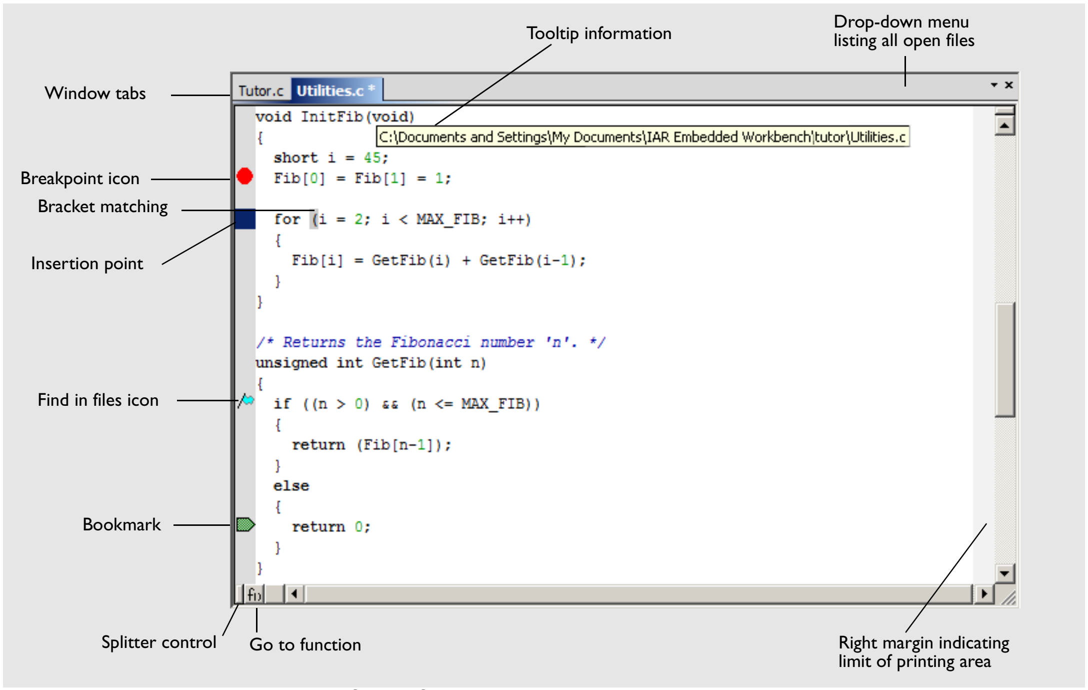
Displays a dialog box, originating from the SCC client application, that contains advanced options. This button is only available if the SCC system supports setting advanced options at check out.

Files

Lists files that will be checked out. The list will contain all files that were selected in the Workspace window when the **Check Out Files** dialog box was opened.

Editor window

The editor window is opened when you open or create a text file in the IDE.



```
void InitFib(void)
{
  short i = 45;
  Fib[0] = Fib[1] = 1;

  for (i = 2; i < MAX_FIB; i++)
  {
    Fib[i] = GetFib(i) + GetFib(i-1);
  }
}

/* Returns the Fibonacci number 'n'. */
unsigned int GetFib(int n)
{
  if ((n > 0) && (n <= MAX_FIB))
  {
    return (Fib[n-1]);
  }
  else
  {
    return 0;
  }
}
```

C:\Documents and Settings\My Documents\IAR Embedded Workbench\tutor\Utilities.c

Figure 23: Editor window

Source code files and HTML files are displayed in editor windows. From an open HTML document, hyperlinks to HTML files work like in normal web browsing. A link to an `eww` workspace file opens the workspace in the IDE, and closes any currently open workspace and the open HTML document.

You can have one or several editor windows open at the same time. On the **Window** menu you find commands for opening multiple editor windows, and commands for moving files between the editor windows.

The editor window is always docked, and its size and position depend on other currently open windows. If a file is read-only, a padlock icon is visible at the bottom left corner of the editor window.

For more information about using the editor, see *Edit menu*, page 97 and the chapter *Editing*.

Source file paths

The IDE supports relative source file paths to a certain degree.

If a source file is located in the project file directory or in any subdirectory of the project file directory, the IDE will use a path relative to the project file when accessing the source file.

Window tabs

The name of the open file is displayed on the tab. If a file has been modified after it was last saved, an asterisk appears after the filename on the tab, for example `Utilities.c *`.

A context menu appears if you right-click on a tab in the editor window.

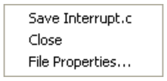


Figure 24: Editor window tab context menu

These commands are available:

Save *file*	Saves the file.
Close	Closes the file.
File Properties	Displays a standard file properties dialog box.

All open files are available from the drop-down menu in the upper right corner of the editor window.

Splitter controls

Use the **Window>Split** command—or the Splitter controls—to split the editor window horizontally or vertically into multiple panes.

Go to function

Click the **Go to function** button in the bottom left-hand corner of the editor window to list all functions of the C or C++ editor window.

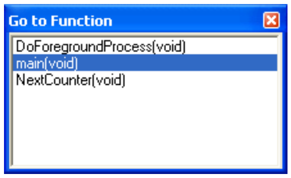


Figure 25: Go to Function window

Double-click the function that you want to show in the editor window.

Context menu

This context menu is available:

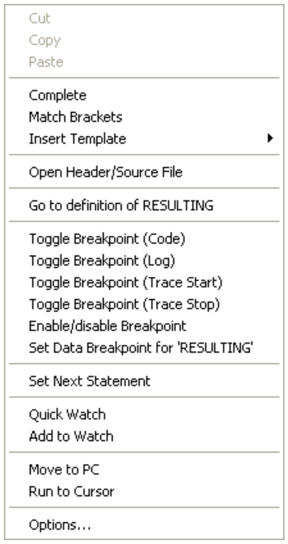


Figure 26: Editor window context menu

The contents of this menu depends on whether the debugger is started or not, and on the C-SPY driver you are using. Typically, additional breakpoint types might be available on this menu. For information about available breakpoints, see the *C-SPY® Debugging Guide for ARM®*.

These commands are available:

Cut, Copy, Paste	Standard window commands.
Complete	Attempts to complete the word you have begun to type, basing the guess on the contents of the rest of the editor document.
Match Brackets	Selects all text between the brackets immediately surrounding the insertion point, increases the selection to the next hierarchic pair of brackets, or beeps if there is no higher bracket hierarchy.
Insert Template	Displays a list in the editor window from which you can choose a code template to be inserted at the location of the insertion point. If the code template you choose requires any field input, the **Template** dialog box appears; for information about this dialog box, see *Template dialog box*, page 106. For information about using code templates, see *Using and adding code templates*, page 53.
Open "*header.h*"	Opens the header file `header.h` in an editor window. This menu command is only available if the insertion point is located on an `#include` line when you open the context menu.
Open Header/Source File	Jumps from the current file to the corresponding header or source file. If the destination file is not open when performing the command, the file will first be opened. This menu command is only available if the insertion point is located on any line except an `#include` line when you open the context menu. This command is also available from the **File>Open** menu.
Go to definition of *symbol*	Shows the declaration of the symbol where the insertion point is placed.
Check In **Check Out** **Undo Checkout**	Commands for source code control; for more details, see *Source Code Control menu*, page 76. These menu commands are only available if the current source file in the editor window is SCC-controlled. The file must also be a member of the current project.
Toggle Breakpoint (Code)	Toggles a code breakpoint at the statement or instruction containing or close to the cursor in the source window. For information about code breakpoints, see the *C-SPY® Debugging Guide for ARM®*.

Toggle Breakpoint (Log)	Toggles a log breakpoint at the statement or instruction containing or close to the cursor in the source window. For information about log breakpoints, see the *C-SPY® Debugging Guide for ARM®*.
Toggle Breakpoint (Trace Start)	Toggles a Trace Start breakpoint. When the breakpoint is triggered, trace data collection starts. For information about Trace Start breakpoints, see the *C-SPY® Debugging Guide for ARM®*. Note that this menu command is only available if the C-SPY driver you are using supports trace.
Toggle Breakpoint (Trace Stop)	Toggles a Trace Stop breakpoint. When the breakpoint is triggered, trace data collection stops. For information about Trace Stop breakpoints, see the *C-SPY® Debugging Guide for ARM®*. Note that this menu command is only available if the C-SPY driver you are using supports trace.
Enable/disable Breakpoint	Toggles a breakpoint between being disabled, but not actually removed—making it available for future use—and being enabled again.
Set Data Breakpoint for '*variable*'	Toggles a data breakpoint on variables with static storage duration. Requires support in the C-SPY driver you are using.
Edit Breakpoint	Displays the **Edit Breakpoint** dialog box to let you edit the breakpoint available on the source code line where the insertion point is located. If there is more than one breakpoint on the line, a submenu is displayed that lists all available breakpoints on that line.
Set Next Statement	Sets the PC directly to the selected statement or instruction without executing any code. Use this menu command with care. This menu command is only available when you are using the debugger.
Quick Watch	Opens the Quick Watch window, see the *C-SPY® Debugging Guide for ARM®*. This menu command is only available when you are using the debugger.
Add to Watch	Adds the selected symbol to the Watch window. This menu command is only available when you are using the debugger.
Move to PC	Moves the insertion point to the current PC position in the editor window. This menu command is only available when you are using the debugger.

Run to Cursor	Executes from the current statement or instruction up to a selected statement or instruction. This menu command is only available when you are using the debugger.
Options	Displays the **IDE Options** dialog box, see *Tools menu*, page 121.

Editor shortcut key summary

The following tables summarize the editor's shortcut keys.

Moving the insertion point:

To move the insertion point	Press
One character left	Arrow left
One character right	Arrow right
One word left	Ctrl+Arrow left
One word right	Ctrl+Arrow right
One line up	Arrow up
One line down	Arrow down
To the start of the line	Home
To the end of the line	End
To the first line in the file	Ctrl+Home
To the last line in the file	Ctrl+End

Table 7: Editor keyboard commands for insertion point navigation

Scrolling text:

To scroll	Press
Up one line	Ctrl+Arrow up
Down one line	Ctrl+Arrow down
Up one page	Page Up
Down one page	Page Down

Table 8: Editor keyboard commands for scrolling

Selecting text:

To select	Press
The character to the left	Shift+Arrow left
The character to the right	Shift+Arrow right

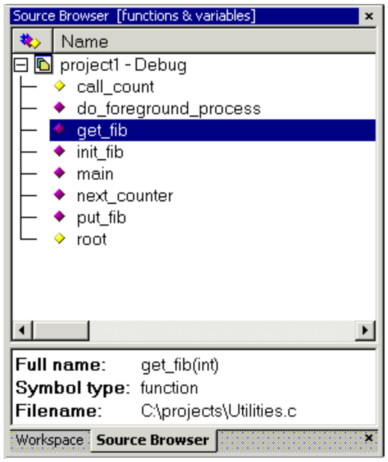
Table 9: Editor keyboard commands for selecting text

To select	Press
One word to the left	Shift+Ctrl+Arrow left
One word to the right	Shift+Ctrl+Arrow right
To the same position on the previous line	Shift+Arrow up
To the same position on the next line	Shift+Arrow down
To the start of the line	Shift+Home
To the end of the line	Shift+End
One screen up	Shift+Page Up
One screen down	Shift+Page Down
To the beginning of the file	Shift+Ctrl+Home
To the end of the file	Shift+Ctrl+End

Table 9: Editor keyboard commands for selecting text (Continued)

Source Browser window

The Source Browser window is available from the **View** menu.



Full name: get_fib(int)
Symbol type: function
Filename: C:\projects\Utilities.c

Figure 27: Source Browser window

The Source Browser window displays an hierarchical view in alphabetical order of all symbols defined in the active build configuration. This means that source browse information is available for symbols in source files and include files part of that configuration. Source browse information is not available for symbols in linked libraries. The window consists of two separate display areas.

For information about how to use the Source Browser window, see *Displaying browse information*, page 36.

The upper display area

The upper display area contains two columns:

	An icon that corresponds to the Symbol type classification, see *Icons used for the symbol types*, page 89.
Name	The names of global symbols and functions defined in the project.

If you click in the window header, you can sort the symbols either by name or by symbol type.

In the upper display area you can also access a context menu; see *Context menu*, page 89.

The lower display area

For a symbol selected in the upper display area, the lower area displays its properties:

Full name	Displays the unique name of each element, for instance `classname`::`membername`.
Symbol type	Displays the symbol type for each element, see *Icons used for the symbol types*, page 89.
Filename	Specifies the path to the file in which the element is defined.

Icons used for the symbol types

These are the icons used:

		Base class
		Class
		Configuration
		Enumeration
		Enumeration constant
◆	(Yellow rhomb)	Field of a struct
◆	(Purple rhomb)	Function
#		Macro
{ }		Namespace
		Template class
		Template function
		Type definition
		Union
◆	(Yellow rhomb)	Variable

Context menu

This context menu is available in the upper display area:

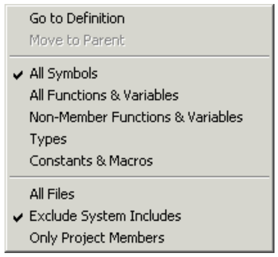


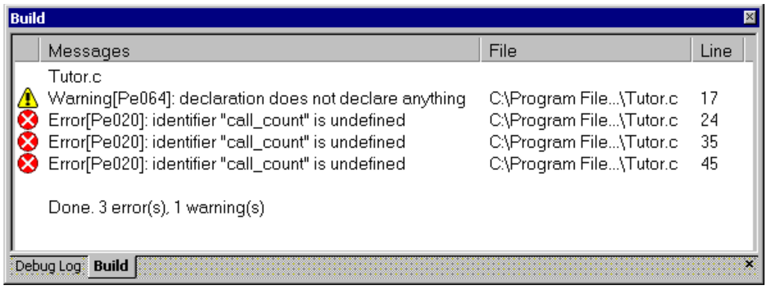
Figure 28: Source Browser window context menu

These commands are available on the context menu:

Go to Definition	The editor window will display the definition of the selected item.
Move to Parent	If the selected element is a member of a class, struct, union, enumeration, or namespace, this menu command can be used for moving to its enclosing element.
All Symbols	Type filter; displays all global symbols and functions defined in the project.
All Functions & Variables	Type filter; displays all functions and variables defined in the project.
Non-Member Functions & Variables	Type filter; displays all functions and variables that are not members of a class.
Types	Type filter; displays all types such as structures and classes defined in the project.
Constants & Macros	Type filter; displays all constants and macros defined in the project.
All Files	File filter; displays symbols from all files that you have explicitly added to your project and all files included by them.
Exclude System Includes	File filter; displays symbols from all files that you have explicitly added to your project and all files included by them, except the include files in the IAR Embedded Workbench installation directory.
Only Project Members	File filter; displays symbols from all files that you have explicitly added to your project, but no include files.

Build window

The Build window is available by choosing **View>Messages**.



	Messages	File	Line
	Tutor.c		
⚠	Warning[Pe064]: declaration does not declare anything	C:\Program File...\Tutor.c	17
⊗	Error[Pe020]: identifier "call_count" is undefined	C:\Program File...\Tutor.c	24
⊗	Error[Pe020]: identifier "call_count" is undefined	C:\Program File...\Tutor.c	35
⊗	Error[Pe020]: identifier "call_count" is undefined	C:\Program File...\Tutor.c	45
	Done. 3 error(s), 1 warning(s)		

Figure 29: Build window (message window)

The Build window displays the messages generated when building a build configuration. When opened, this window is, by default, grouped together with the other message windows, see *Windows*, page 69. Double-click a message in the Build window to open the appropriate file for editing, with the insertion point at the correct position.

Context menu

This context menu is available:

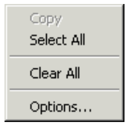


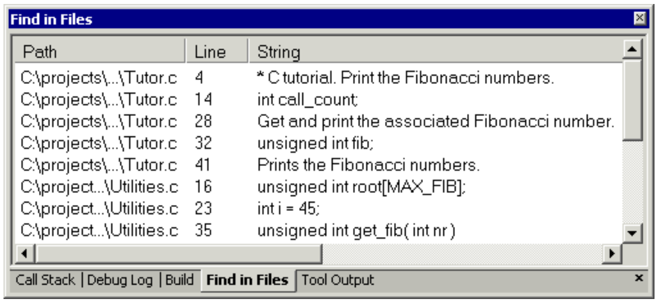
Figure 30: Build window context menu

These commands are available:

Copy	Copies the contents of the window.
Select All	Selects the contents of the window.
Clear All	Deletes the contents of the window.
Options	Opens the **Messages** page of the **IDE options** dialog box. On this page you can set options related to messages; see *Messages options*, page 132.

Find in Files window

The Find in Files window is available by choosing **View>Messages**.



Path	Line	String
C:\projects\...\Tutor.c	4	* C tutorial. Print the Fibonacci numbers.
C:\projects\...\Tutor.c	14	int call_count;
C:\projects\...\Tutor.c	28	Get and print the associated Fibonacci number.
C:\projects\...\Tutor.c	32	unsigned int fib;
C:\projects\...\Tutor.c	41	Prints the Fibonacci numbers.
C:\project...\Utilities.c	16	unsigned int root[MAX_FIB];
C:\project...\Utilities.c	23	int i = 45;
C:\project...\Utilities.c	35	unsigned int get_fib(int nr)

Figure 31: Find in Files window (message window)

The Find in Files window displays the output from the **Edit>Find and Replace>Find in Files** command. When opened, this window is, by default, grouped together with the other message windows, see *Windows*, page 69.

Double-click an entry in the window to open the appropriate file with the insertion point positioned at the correct location. That source location is highlighted with a blue flag icon.

Context menu

This context menu is available:

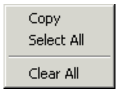


Figure 32: Find in Files window context menu

These commands are available:

Copy	Copies the contents of the window.
Select All	Selects the contents of the window.
Clear All	Deletes the contents of the window and any blue flag icons in the left-side margin of the editor window.

Tool Output window

The Tool Output window is available by choosing **View>Messages>Tool Output**.

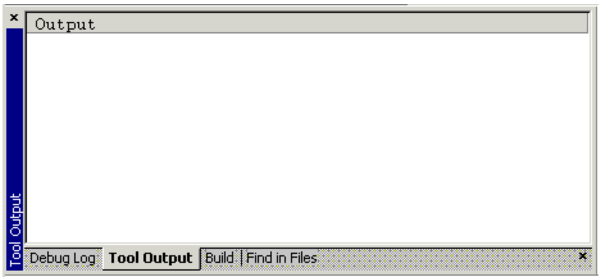


Figure 33: Tool Output window (message window)

The Tool Output window displays any messages output by user-defined tools in the **Tools** menu, provided that you have selected the option **Redirect to Output Window** in the **Configure Tools** dialog box; see *Configure Tools dialog box*, page 143. When opened, this window is, by default, grouped together with the other message windows, see *Windows*, page 69.

Context menu

This context menu is available:

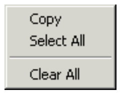


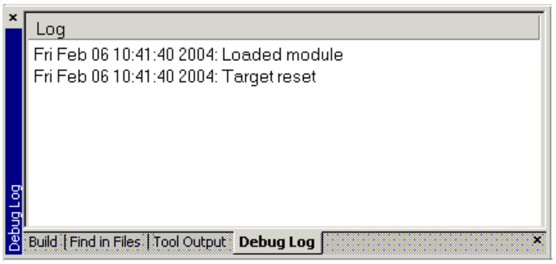
Figure 34: Tool Output window context menu

These commands are available:

Copy	Copies the contents of the window.
Select All	Selects the contents of the window.
Clear All	Deletes the contents of the window.

Debug Log window

The Debug Log window is available by choosing **View>Messages>Debug Log**.



```
Fri Feb 06 10:41:40 2004: Loaded module
Fri Feb 06 10:41:40 2004: Target reset
```

Figure 35: Debug Log window (message window)

The Debug Log window displays debugger output, such as diagnostic messages and trace information. When opened, this window is, by default, grouped together with the other message windows, see *Windows*, page 69.

Double-click any rows in one of the following formats to display the corresponding source code in the editor window:

```
<path> (<row>):<message>
<path> (<row>,<column>):<message>
```

Context menu

This context menu is available:

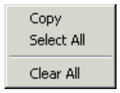


Figure 36: Debug Log window context menu

These commands are available:

Copy	Copies the contents of the window.
Select All	Selects the contents of the window.
Clear All	Deletes the contents of the window.

Menus

The available menus are:

- File menu
- Edit menu
- View menu
- Project menu
- Tools menu
- Window menu
- Help menu.

In addition, a set of C-SPY-specific menus become available when you start the debugger. For reference information about these menus, see the *C-SPY® Debugging Guide for ARM®*.

File menu

The **File** menu provides commands for opening workspaces and source files, saving and printing, and exiting from the IDE.

The menu also includes a numbered list of the most recently opened files and workspaces. To open one of them, choose it from the menu.

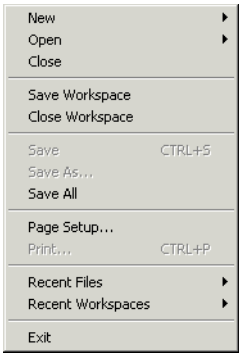


Figure 37: File menu

These commands are available:

🗋	**New** Ctrl+N	Displays a submenu with commands for creating a new workspace, or a new text file.

Open>File Ctrl+O	Displays a submenu from which you can select a text file or an HTML document to open. See *Editor window*, page 81.
Open>Workspace	Displays a submenu from which you can select a workspace file to open. Before a new workspace is opened you will be prompted to save and close any currently open workspaces.
Open>Header/Source File Ctrl+Shift+H	Opens the header file or source file that corresponds to the current file, and jumps from the current file to the newly opened file. This command is also available from the context menu available from the editor window.
Close	Closes the active window. You will be given the opportunity to save any files that have been modified before closing.
Open Workspace	Displays a dialog box where you can open a workspace file. You will be given the opportunity to save and close any currently open workspace file that has been modified before opening a new workspace.
Save Workspace	Saves the current workspace file.
Close Workspace	Closes the current workspace file.
Save Ctrl+S	Saves the current text file or workspace file.
Save As	Displays a dialog box where you can save the current file with a new name.
Save All	Saves all open text documents and workspace files.
Page Setup	Displays a dialog box where you can set printer options.
Print Ctrl+P	Displays a dialog box where you can print a text document.
Recent Files	Displays a submenu where you can quickly open the most recently opened text documents.
Recent Workspaces	Displays a submenu where you can quickly open the most recently opened workspace files.

Exit	Exits from the IDE. You will be asked whether to save any changes to text files before closing them. Changes to the project are saved automatically.

Edit menu

The **Edit** menu provides commands for editing and searching.

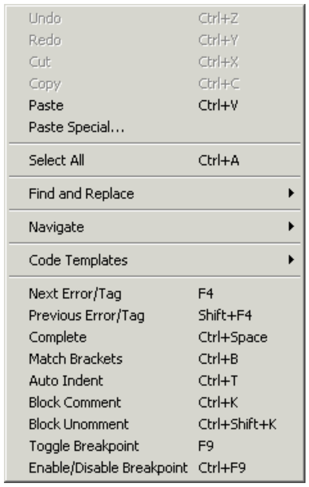


Figure 38: Edit menu

These commands are available:

Undo Ctrl+Z	Undoes the last edit made to the current editor window.
Redo Ctrl+Y	Redoes the last **Undo** in the current editor window. You can undo and redo an unlimited number of edits independently in each editor window.
Cut Ctrl+X	The standard Windows command for cutting text in editor windows and text boxes.
Copy Ctrl+C	The standard Windows command for copying text in editor windows and text boxes.

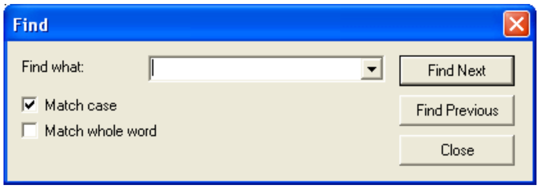
Paste Ctrl+V	The standard Windows command for pasting text in editor windows and text boxes.
Paste Special	Provides you with a choice of the most recent contents of the clipboard to choose from when pasting in editor documents.
Select All Ctrl+A	Selects all text in the active editor window.
Find and Replace>Find Ctrl+F	Displays the **Find** dialog box where you can search for text within the current editor window; see *Find dialog box*, page 101. Note that if the insertion point is located in the Memory window when you choose the **Find** command, the dialog box will contain a different set of options than otherwise. If the insertion point is located in the Trace window when you choose the **Find** command, the **Find in Trace** dialog box is opened; the contents of this dialog box depend on the C-SPY driver you are using, see the *C-SPY® Debugging Guide for ARM®* for more information.
Find and Replace>Find Next F3	Finds the next occurrence of the specified string.
Find and Replace>Find Previous Shift+F3	Finds the previous occurrence of the specified string.
Find and Replace>Find Next (Selected) Ctrl+F3	Searches for the next occurrence of the currently selected text or the word currently surrounding the insertion point.
Find and Replace>Find Previous (Selected) Ctrl+Shift+F3	Searches for the previous occurrence of the currently selected text or the word currently surrounding the insertion point.
Find and Replace>Replace Ctrl+H	Displays a dialog box where you can search for a specified string and replace each occurrence with another string; see *Replace dialog box*, page 102. Note that if the insertion point is located in the Memory window when you choose the **Replace** command, the dialog box will contain a different set of options than otherwise.

Find and Replace>Find in Files	Displays a dialog box where you can search for a specified string in multiple text files; see *Find in Files dialog box*, page 103.
Find and Replace>Incremental Search Ctrl+I	Displays a dialog box where you can gradually fine-tune or expand the search by continuously changing the search string; see *Incremental Search dialog box*, page 105.
Navigate>Go To Ctrl+G	Displays the **Go to Line** dialog box where you can move the insertion point to a specified line and column in the current editor window.
Navigate>Toggle Bookmark Ctrl+F2	Toggles a bookmark at the line where the insertion point is located in the active editor window.
Navigate>Go to Bookmark F2	Moves the insertion point to the next bookmark that has been defined with the Toggle Bookmark command.
Navigate>Navigate Backward Alt+Left Arrow	Navigates backward in the insertion point history. The current position of the insertion point is added to the history by actions like **Go to definition** and clicking on a result from the **Find in Files** command.
Navigate>Navigate Forward Alt+Right Arrow	Navigates forward in the insertion point history. The current position of the insertion point is added to the history by actions like **Go to definition** and clicking on a result from the **Find in Files** command.
Navigate>Go to Definition F12	Shows the declaration of the selected symbol or the symbol where the insertion point is placed. This menu command is available when browse information has been enabled, see *Project options*, page 134.
Code Templates>Insert Template Ctrl+Shift+Space	Displays a list in the editor window from which you can choose a code template to be inserted at the location of the insertion point. If the code template you choose requires any field input, the **Template** dialog box appears; see *Template dialog box*, page 106. For information about using code templates, see *Using and adding code templates*, page 53.
Code Templates>Edit Templates	Opens the current code template file, where you can modify existing code templates and add your own code templates. For information about using code templates, see *Using and adding code templates*, page 53.

Next Error/Tag F4	If the message window contains a list of error messages or the results from a **Find in Files** search, this command displays the next item from that list in the editor window.
Previous Error/Tag Shift+F4	If the message window contains a list of error messages or the results from a **Find in Files** search, this command displays the previous item from that list in the editor window.
Complete Ctrl+Space	Attempts to complete the word you have begun to type, basing the guess on the contents of the rest of the editor document.
Match Brackets	Selects all text between the brackets immediately surrounding the insertion point, increases the selection to the next hierarchic pair of brackets, or beeps if there is no higher bracket hierarchy.
Auto Indent Ctrl+T	Indents one or several lines you have selected in a C/C++ source file. To configure the indentation, see *Configure Auto Indent dialog box*, page 127.
Block Comment Ctrl+K	Places the C++ comment character sequence `//` at the beginning of the selected lines.
Block Uncomment Ctrl+K	Removes the C++ comment character sequence `//` from the beginning of the selected lines.
Toggle Breakpoint F9	Toggles a breakpoint at the statement or instruction that contains or is located near the cursor in the source window. This command is also available as an icon button in the debug bar.
Enable/Disable Breakpoint Ctrl+F9	Toggles a breakpoint between being disabled, but not actually removed—making it available for future use—and being enabled again.

Find dialog box

The **Find** dialog box is available from the **Edit** menu.



[✓] Match case
[ ] Match whole word

Figure 39: Find dialog box

Note that the contents look different if you search in an editor window compared to if you search in the Memory window.

Find what	Specify the text to search for.
Match case	Searches only for occurrences that exactly match the case of the specified text. Otherwise, specifying `int` will also find `INT` and `Int`. This option is only available when you search in an editor window.
Match whole word	Searches for the specified text only if it occurs as a separate word. Otherwise, specifying `int` will also find `print`, `sprintf` etc. This option is only available when you search in an editor window.
Search as hex	Searches for the specified hexadecimal value. This option is only available when you search in the Memory window.
Find next	Searches for the next occurrence of the selected text.
Find previous	Searches for the previous occurrence of the selected text.
Stop	Stops an ongoing search. This button is only available during a search in the Memory window.

Replace dialog box

The **Replace** dialog box is available from the **Edit** menu.

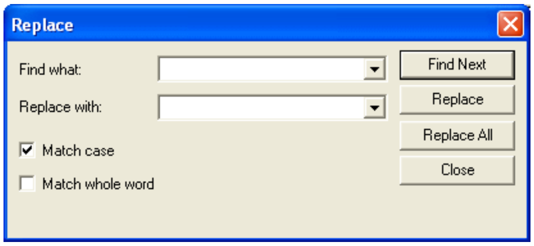


Figure 40: Replace dialog box

Note that the contents look different if you search in an editor window compared to if you search in the Memory window.

Find what	Specify the text to search for.
Replace with	Specify the text to replace each found occurrence with.
Match case	Searches only for occurrences that exactly match the case of the specified text. Otherwise, specifying `int` will also find `INT` and `Int`. This option is only available when you search in an editor window.
Match whole word	Searches for the specified text only if it occurs as a separate word. Otherwise, `int` will also find `print`, `sprintf` etc. This option is only available when you search in an editor window.
Search as hex	Searches for the specified hexadecimal value. This option is only available when you perform the search in the Memory window.
Find next	Searches for the next occurrence of the text you have specified.
Replace	Replaces the searched text with the specified text.
Replace all	Replaces all occurrences of the searched text in the current editor window.

Find in Files dialog box

The **Find in Files** dialog box is available from the **Edit** menu.

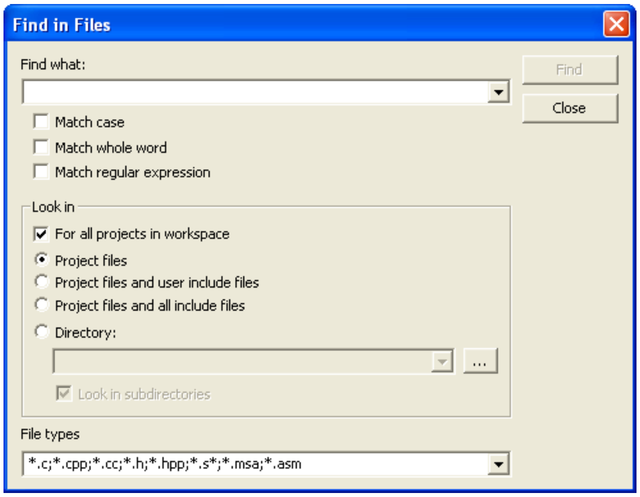


Figure 41: Find in Files dialog box

Use this dialog box to search for a string in files.

The result of the search appears in the Find in Files message window—available from the **View** menu. You can then go to each occurrence by choosing the **Edit>Next Error/Tag** command, alternatively by double-clicking the messages in the Find in Files message window. This opens the corresponding file in an editor window with the insertion point positioned at the start of the specified text. A blue flag in the left-hand margin indicates the line.

Find what

Specify the string you want to search for or a regular expression. You can narrow the search down with one or more of these conditions:

Match case	Searches only for occurrences that exactly match the case of the specified text. Otherwise, specifying `int` will also find `INT` and `Int`.
Match whole word	Searches only for the string when it occurs as a separate word (short cut `&w`). Otherwise, `int` will also find `print`, `sprintf` and so on.

Match regular expression	Searches only for the regular expression, which must follow the standard for the Perl programming language.

Look in

Specify which files you want to search in. Choose between:

For all projects in workspace	Searches all projects in the workspace, not just the active project.
Project files	Searches all files that you have explicitly added to your project.
Project files and user include files	Searches all files that you have explicitly added to your project and all files that they include, except the include files in the IAR Embedded Workbench installation directory.
Project files and all include files	Searches all project files that you have explicitly added to your project and all files that they include.
Directory	Searches the directory that you specify. Recent search locations are saved in the drop-down list. Locate the directory using the browse button.
Look in subdirectories	Searches the directory that you have specified and all its subdirectories.

File types

A filter for choosing which type of files to search; the filter applies to all **Look in** settings. Choose the appropriate filter from the drop-down list. The text field is editable, to let you add your own filters. Use the `*` character to indicate zero or more unknown characters of the filters, and the `?` character to indicate one unknown character.

Stop

Stops an ongoing search. This button is only available during an ongoing search.

Incremental Search dialog box

The **Incremental Search** dialog box is available from the **Edit** menu.

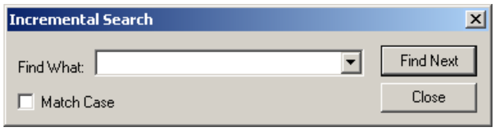


Figure 42: Incremental Search dialog box

Use this dialog box to gradually fine-tune or expand the search string.

Find What

Type the string to search for. The search is performed from the location of the insertion point—the *start point*. Every character you add to or remove from the search string instantly changes the search accordingly. If you remove a character, the search starts over again from the start point.

If a word in the editor window is selected when you open the **Incremental Search** dialog box, this word will be displayed in the **Find What** text box.

Match Case

Searches for occurrences that exactly match the case of the specified text. Otherwise, searching for `int` will also find `INT` and `Int`.

Find Next

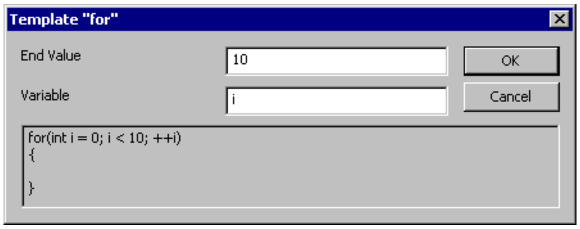
Searches for the next occurrence of the current search string. If the **Find What** text box is empty when you click the **Find Next** button, a string to search for will automatically be selected from the drop-down list. To search for this string, click **Find Next**.

Close

Closes the dialog box.

Template dialog box

The **Template** dialog box appears when you insert a code template that requires any field input.



End Value: 10

Variable: i

```
for(int i = 0; i < 10; ++i)
{

}
```

Figure 43: Template dialog box

Use this dialog box to specify any field input that is required by the source code template you insert.

Note: The figure reflects the default code template that can be used for automatically inserting code for a `for` loop.

Text fields

Specify the required input in the text fields. Which fields that appear depends on how the code template is defined.

Display area

The display area shows the code that would result from the code template, using the values you submit.

For more information about using code templates, see *Using and adding code templates*, page 53.

View menu

The **View** menu provides several commands for opening windows and displaying toolbars in the IDE. When the debugger is running you can also open debugger-specific windows from this menu. See the *C-SPY® Debugging Guide for ARM®* for information about these.

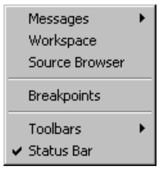


Figure 44: View menu

These commands are available:

Messages	Displays a submenu which gives access to the message windows—Build, Find in Files, Tool Output, Debug Log—that display messages and text output from the IAR Embedded Workbench commands. If the window you choose from the menu is already open, it becomes the active window.
Workspace	Opens the current Workspace window, see *Workspace window*, page 72.
Source Browser	Opens the Source Browser window, see *Source Browser window*, page 87.
Breakpoints	Opens the Breakpoints window, see the *C-SPY® Debugging Guide for ARM®*.
Disassembly window	Opens the Disassembly window. Only available when the debugger is running.
Memory window	Opens the Memory window. Only available when the debugger is running.
Symbolic Memory window	Opens the Symbolic Memory window. Only available when the debugger is running.
Register window	Opens the Register window. Only available when the debugger is running.
Watch window	Opens the Watch window. Only available when the debugger is running.
Locals window	Opens the Locals window. Only available when the debugger is running.

Statics window	Opens the Statics window. Only available when the debugger is running.
Auto window	Opens the Auto window. Only available when the debugger is running.
Live Watch window	Opens the Live Watch window. Only available when the debugger is running.
Quick Watch window	Opens the Quick Watch window. Only available when the debugger is running.
Call Stack window	Opens the Call Stack window. Only available when the debugger is running.
Terminal I/O window	Opens the Terminal I/O window. Only available when the debugger is running.
Code Coverage window	Opens the Code Coverage window. Only available when the debugger is running.
Profiling window	Opens the Profiling window. Only available when the debugger is running.
Stack window	Opens the Stack window. Only available when the debugger is running.
Toolbars	The options **Main** and **Debug** toggle the two toolbars on or off.
Status bar	Toggles the status bar on or off.

Project menu

The **Project** menu provides commands for working with workspaces, projects, groups, and files, and for specifying options for the build tools, and running the tools on the current project.

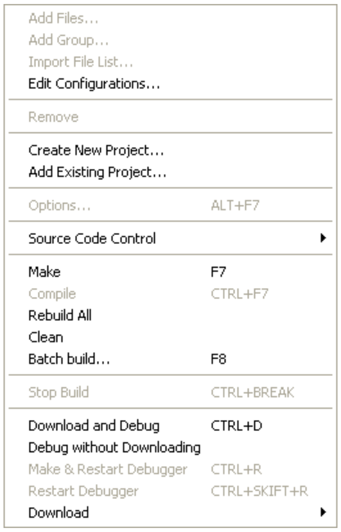


Figure 45: Project menu

These commands are available:

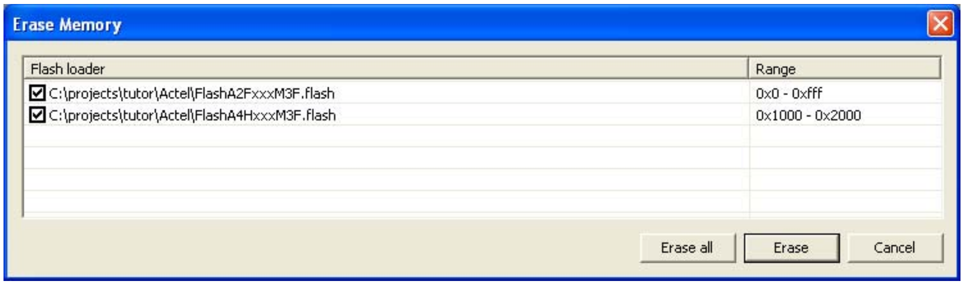
Add Files	Displays a dialog box where you can select which files to include in the current project.
Add Group	Displays a dialog box where you can create a new group. In the **Group Name** text box, specify the name of the new group. For more information about groups, see *Groups*, page 32.
Import File List	Displays a standard **Open** dialog box where you can import information about files and groups from projects created using another IAR Systems toolchain. To import information from project files which have one of the older filename extensions `pew` or `prj` you must first have exported the information using the context menu command **Export File List** available in your current IAR Embedded Workbench.

Edit Configurations	Displays the **Configurations for project** dialog box, where you can define new or remove existing build configurations. See *Configurations for project dialog box*, page 113.
Remove	In the Workspace window, removes the selected item from the workspace.
Create New Project	Displays the **Create New Project** dialog box where you can create a new project and add it to the workspace; see *Create New Project dialog box*, page 115.
Add Existing Project	Displays a standard **Open** dialog box where you can add an existing project to the workspace.
Options Alt+F7	Displays the **Options** dialog box, where you can set options for the build tools, for the selected item in the Workspace window; see *Options dialog box*, page 116. You can set options for the entire project, for a group of files, or for an individual file.
Source Code Control	Displays a submenu with commands for source code control, see *Source Code Control menu*, page 76.
Make F7	Brings the current build configuration up to date by compiling, assembling, and linking only the files that have changed since the last build.
Compile Ctrl+F7	Compiles or assembles the currently selected file, files, or group. One or more files can be selected in the Workspace window—all files in the same project, but not necessarily in the same group. You can also select the editor window containing the file you want to compile. The **Compile** command is only enabled if *all* files in the selection can be compiled or assembled. You can also select a *group*, in which case the command is applied to each file in the group (including inside nested groups) that can be compiled, even if the group contains files that cannot be compiled, such as header files. If the selected file is part of a multi-file compilation group, the command will still only affect the selected file.
Rebuild All	Rebuilds and relinks all files in the current target.
Clean	Removes any intermediate files.

Batch Build F8	Displays the **Batch Build** dialog box where you can configure named batch build configurations, and build a named batch. See *Batch Build dialog box*, page 119.
Stop Build Ctrl+Break	Stops the current build operation.
Download and Debug Ctrl+D	Downloads the application and starts C-SPY so that you can debug the project object file. If necessary, a make will be performed before running C-SPY to ensure the project is up to date. This command is not available during debugging.
Debug without Downloading	Starts C-SPY so that you can debug the project object file. This menu command is a shortcut for the **Suppress Download** option available on the **Download** page. The **Debug without Downloading** command is not available during debugging.
Make & Restart Debugger	Stops C-SPY, makes the active build configuration, and starts the debugger again; all in a single command. This command is only available during debugging.
Restart Debugger	Stops C-SPY and starts the debugger again; all in a single command. This command is only available during debugging.
Download	Commands for flash download and erase. Choose between these commands: **Download active application** downloads the active application to the target without launching a full debug session. The result is roughly equivalent to launching a debug session but exiting it again before the execution starts. **Download file** opens a standard **Open** dialog box where you can specify a file to be downloaded to the target system without launching a full debug session. **Erase memory** erases all parts of the flash memory. If your `.board` file specifies only one flash memory, a simple confirmation dialog box is displayed where you confirm the erasure. However, if your `.board` file specifies two or more flash memories, the **Erase Memory** dialog box is displayed. See *Erase Memory dialog box*, page 112.

Erase Memory dialog box

The **Erase Memory** dialog box is displayed when you have chosen **Project>Download>Erase Memory** and your flash memory system configuration file (filename extension `.board`) specifies two or more flash memories.



Flash loader	Range
☑ C:\projects\tutor\Actel\FlashA2FxxxM3F.flash	0x0 - 0xfff
☑ C:\projects\tutor\Actel\FlashA4HxxxM3F.flash	0x1000 - 0x2000

Figure 46: Erase Memory dialog box

Use this dialog box to erase one or more of the flash memories.

Display area

Each line lists the path to the flash memory device configuration file (filename extension `.flash`) and the associated memory range. Select the memory you want to erase.

Buttons

These buttons are available:

Erase all	All memories listed in the dialog box are erased, regardless of individually selected lines.
Erase	Erases the selected memories.
Cancel	Closes the dialog box.

Configurations for project dialog box

The **Configurations for project** dialog box is available by choosing **Project>Edit Configurations**.

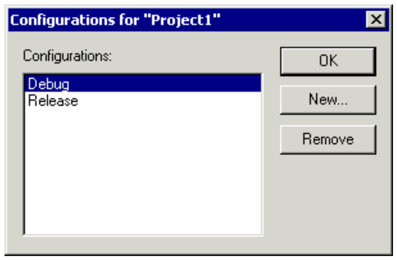


Figure 47: Configurations for project dialog box

Use this dialog box to define new build configurations for the selected project; either entirely new, or based on a previous project.

Configurations

Lists existing configurations, which can be used as templates for new configurations.

New

Displays a dialog box where you can define new build configurations, see *New Configuration dialog box*, page 114.

Remove

Removes the configuration that is selected in the **Configurations** list.

New Configuration dialog box

The **New Configuration** dialog box is available by clicking **New** in the **Configurations for project** dialog box.

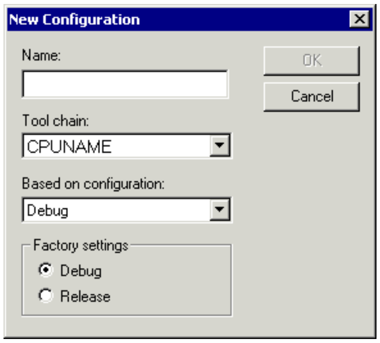


Figure 48: New Configuration dialog box

Use this dialog box to define new build configurations; either entirely new, or based on any currently defined configuration.

Name

Type the name of the build configuration.

Tool chain

Specify the target to build for. If you have several versions of IAR Embedded Workbench for different targets installed on your host computer, the drop-down list might contain some or all of these targets.

Based on configuration

Selects a currently defined build configuration to base the new configuration on. The new configuration will inherit the project settings and information about the factory settings from the old configuration. If you select **None**, the new configuration will be based strictly on the factory settings.

Factory settings

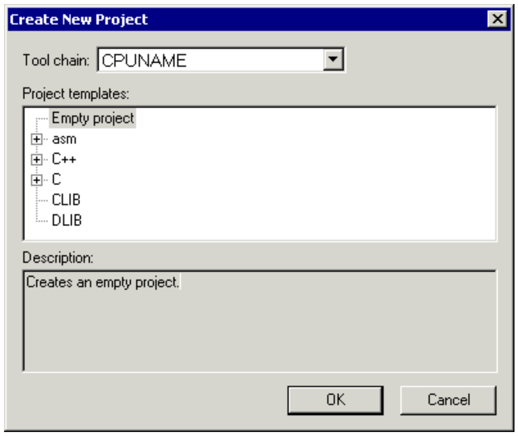
Select the default factory settings that you want to apply to your new build configuration. These factory settings will be used by your project if you click the **Factory Settings** button in the **Options** dialog box.

Choose between:

Debug	Factory settings suitable for a debug build configuration.
Release	Factory settings suitable for a release build configuration.

Create New Project dialog box

The **Create New Project** dialog box is available from the **Project** menu.



Creates an empty project.

Figure 49: Create New Project dialog box

Use this dialog box to create a new project based on a template project. Template projects are available for C/C++ applications, assembler applications, and library projects. You can also create your own template projects.

Tool chain

Selects the target to build for. If you have several versions of IAR Embedded Workbench for different targets installed on your host computer, the drop-down list might contain some or all of these targets.

Project templates

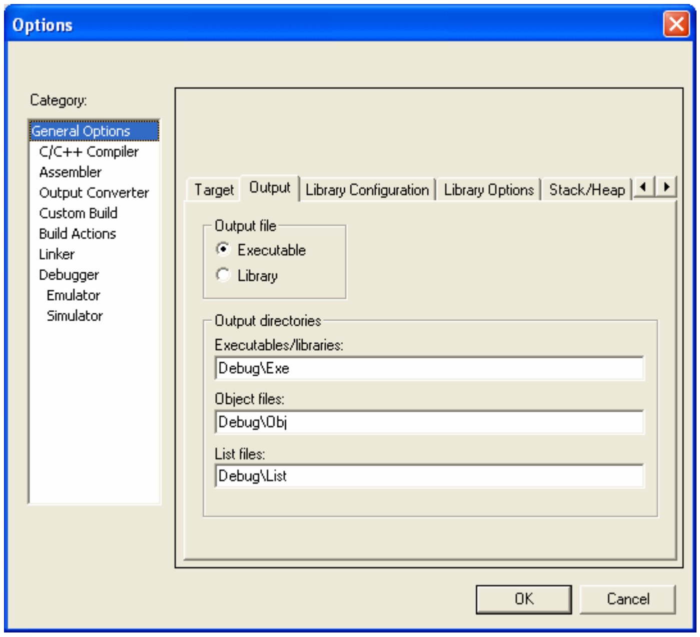
Select a template to base the new project on, from this list of available template projects.

Description

A description of the currently selected template.

Options dialog box

The **Options** dialog box is available from the **Project** menu.



Output file
(● Executable) ○ Library

Output directories
Executables/libraries: Debug\Exe
Object files: Debug\Obj
List files: Debug\List

Figure 50: Options dialog box

Use this dialog box to specify your project settings.

For detailed information about the options in each category, see the option reference chapters:

- *General options*
- *Compiler options*
- *Assembler options*
- *Output converter options*
- *Custom build options*
- *Build actions options*
- *Linker options*
- *Library builder options*
- *Debugger options* (in the *C-SPY® Debugging Guide for ARM®).*

Category

Selects the build tool you want to set options for. The available categories will depend on the tools installed in your IAR Embedded Workbench IDE, and will typically include:

General Options	General options
C/C++ Compiler	IAR C/C++ Compiler options
Assembler	IAR Assembler options
Output Converter	Options for converting ELF output to Motorola, Intel-standard, or other simple formats.
Custom Build	Options for extending the toolchain
Build Actions	Options for pre-build and post-build actions
Linker	Linker options. This category is available for application projects.
Library Builder	Library builder options. This category is available for library projects.
Debugger	IAR C-SPY Debugger options
Simulator	Simulator-specific options
C-SPY hardware drivers	Options specific to additional hardware debuggers might be available depending on the installed drivers.

Selecting a category displays one or more pages of options for that component of the IDE.

Factory Settings

Restores all settings to the default factory settings.

Argument variables

On many of the pages in the **Options** dialog box, you can use argument variables for paths and arguments:

Variable	Description
`$CONFIG_NAME$`	The name of the current build configuration, for example Debug or Release.

Table 10: Argument variables

Variable	Description
`$CUR_DIR$`	Current directory
`$CUR_LINE$`	Current line
`$DATE$`	Today's date
`$EW_DIR$`	Top directory of IAR Embedded Workbench, for example `c:\program files\iar systems\embedded workbench 6.`*`n`*
`$EXE_DIR$`	Directory for executable output
`$FILE_BNAME$`	Filename without extension
`$FILE_BPATH$`	Full path without extension
`$FILE_DIR$`	Directory of active file, no filename
`$FILE_FNAME$`	Filename of active file without path
`$FILE_PATH$`	Full path of active file (in Editor, Project, or Message window)
`$LIST_DIR$`	Directory for list output
`$OBJ_DIR$`	Directory for object output
`$PROJ_DIR$`	Project directory
`$PROJ_FNAME$`	Project filename without path
`$PROJ_PATH$`	Full path of project file
`$TARGET_DIR$`	Directory of primary output file
`$TARGET_BNAME$`	Filename without path of primary output file and without extension
`$TARGET_BPATH$`	Full path of primary output file without extension
`$TARGET_FNAME$`	Filename without path of primary output file
`$TARGET_PATH$`	Full path of primary output file
`$TOOLKIT_DIR$`	Directory of the active product, for example `c:\program files\iar systems\embedded workbench 6.`*`n`*`\arm`
`$USER_NAME$`	Your host login name
`$_`*`ENVVAR`*`_$`	The environment variable *`ENVVAR`*. Any name within `$_` and `_$` will be expanded to that system environment variable.

Table 10: Argument variables (Continued)

Argument variables can also be used on some pages in the **IDE Options** dialog box, see *Tools menu*, page 121.

Batch Build dialog box

The **Batch Build** dialog box is available by choosing **Project>Batch build**.

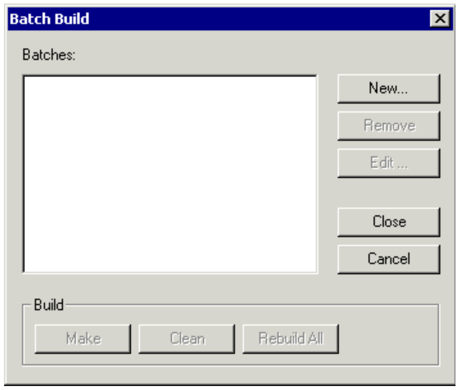


Figure 51: Batch Build dialog box

This dialog box lists all defined batches of build configurations. For more information, see *Building multiple configurations in a batch*, page 43.

Batches

Select the batch you want to build from this list of currently defined batches of build configurations.

Build

Give the build command you want to execute:

- **Make**
- **Clean**
- **Rebuild All**.

New

Displays the **Edit Batch Build** dialog box, where you can define new batches of build configurations.

Remove

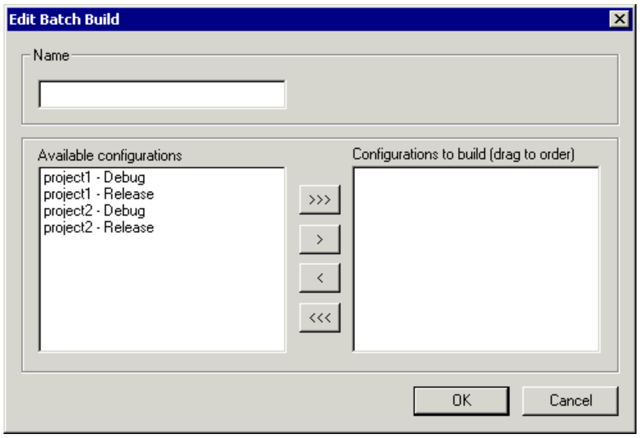
Removes the selected batch.

Edit

Displays the **Edit Batch Build** dialog box, where you can edit existing batches of build configurations.

Edit Batch Build dialog box

The **Edit Batch Build** dialog box is available from the **Batch Build** dialog box.



project1 - Debug
project1 - Release
project2 - Debug
project2 - Release

Figure 52: Edit Batch Build dialog box

Use this dialog box to create new batches of build configurations, and edit already existing batches.

Name

Type a name for a batch that you are creating, or change the existing name (if you wish) for a batch that you are editing.

Available configurations

Select the configurations you want to move to be included in the batch you are creating or editing, from this list of all build configurations that belong to the workspace.

To move a build configuration from the **Available configurations** list to the **Configurations to build** list, use the arrow buttons.

Configurations to build

Lists the build configurations that will be included in the batch you are creating or editing. Drag the build configurations up and down to set the order between the configurations.

Tools menu

The **Tools** menu provides commands for customizing the environment, such as changing common fonts and shortcut keys.

It is a user-configurable menu to which you can add tools for use with IAR Embedded Workbench. Thus, it might look different depending on which tools you have preconfigured to appear as menu items.

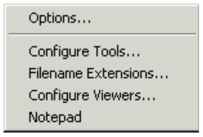


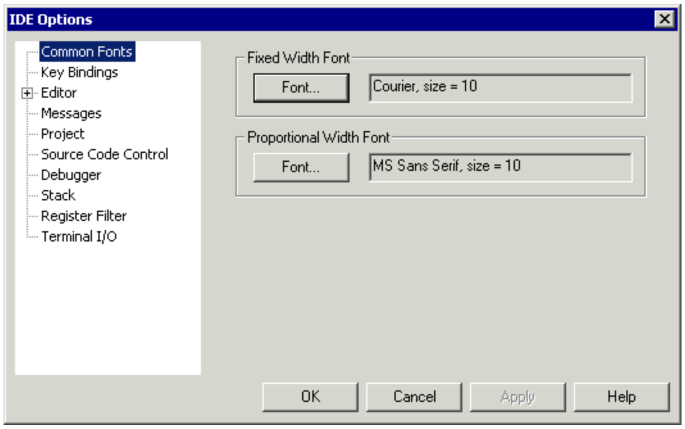
Figure 53: Tools menu

These commands are available:

Options	Displays the **IDE Options** dialog box where you can customize the IDE.
Configure Tools	Displays the **Configure Tools** dialog box where you can set up the interface to use external tools; see *Configure Tools dialog box*, page 143.
Filename Extensions	Displays the **Filename Extensions** dialog box where you can define the filename extensions to be accepted by the build tools; see *Filename Extensions dialog box*, page 146.
Configure Viewers	Displays the **Configure Viewers** dialog box where you can configure viewer applications to open documents with; see *Configure Viewers dialog box*, page 148.
Notepad	User-configured. This is an example of a user-configured addition to the Tools menu.

Common Fonts options

The **Common Fonts** options are available by choosing **Tools>Options**.



Fixed Width Font: Courier, size = 10

Proportional Width Font: MS Sans Serif, size = 10

Figure 54: Common Fonts options

Use this page to configure the fonts used for all project windows except the editor windows.

For information about how to change the font in the editor windows, see *Editor Colors and Fonts options*, page 131.

Fixed Width Font

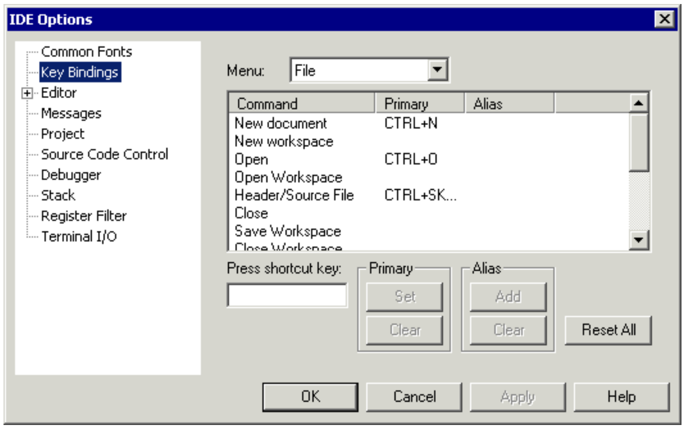
Selects which font to use in the Disassembly, Register, and Memory windows.

Proportional Width Font

Selects which font to use in all windows except the Disassembly, Register, Memory, and editor windows.

Key Bindings options

The **Key Bindings** options are available by choosing **Tools>Options**.



Command	Primary	Alias
New document	CTRL+N	
New workspace		
Open	CTRL+O	
Open Workspace		
Header/Source File	CTRL+SK...	
Close		
Save Workspace		

Figure 55: Key Bindings options

Use this page to customize the shortcut keys used for the IDE menu commands.

Menu

Selects the menu to be edited. Any currently defined shortcut keys for the selected menu are listed below the **Menu** drop-down list.

List of commands

Selects the menu command you want to configure your own shortcut keys for, from this list of all commands available on the selected menu.

Press shortcut key

Type the key combination you want to use as shortcut key for the selected command. You cannot set or add a shortcut if it is already used by another command.

Primary

Choose to:

Set	Saves the key combination in the **Press shortcut key** field as a shortcut for the selected command in the list.
Clear	Removes the listed primary key combination as a shortcut for the selected command in the list.

The new shortcut will be displayed next to the command on the menu.

Alias

Choose to:

Add	Saves the key combination in the **Press shortcut key** field as an alias—a hidden shortcut—for the selected command in the list.
Clear	Removes the listed alias key combination as a shortcut for the selected command in the list.

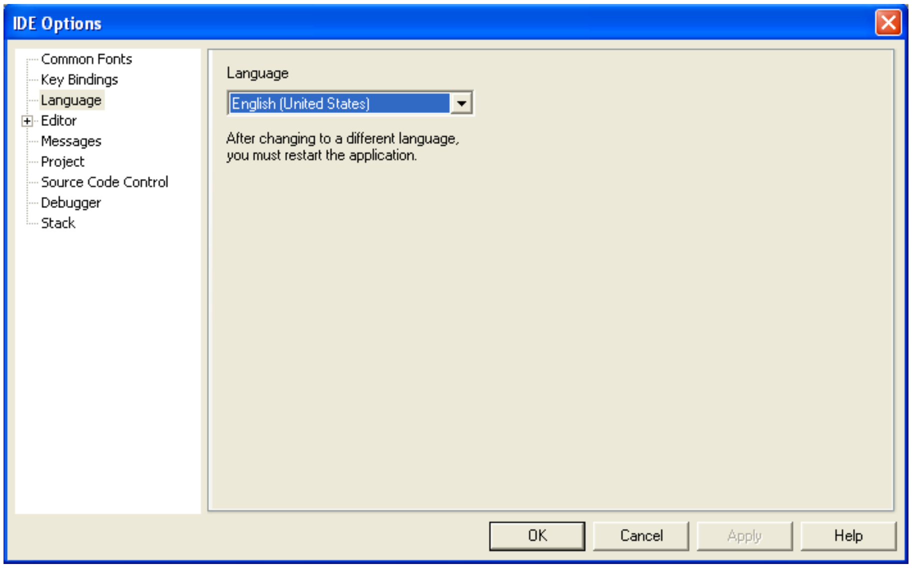
The new shortcut will be not displayed next to the command on the menu.

Reset All

Reverts the shortcuts for all commands to the factory settings.

Language options

The **Language** options are available by choosing **Tools>Options**.



Language

English (United States)

After changing to a different language, you must restart the application.

Figure 56: Language options

Use this page to specify the language to be used in windows, menus, dialog boxes, etc.

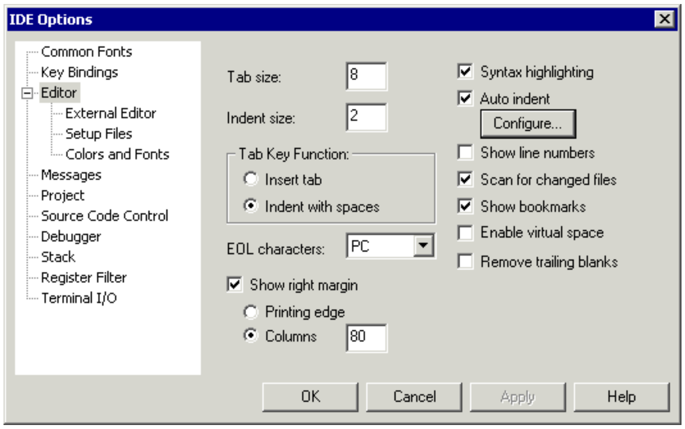
Language

Selects the language to be used. In the IDE, **English (United States)** and **Japanese** are available.

Note: If you have installed IAR Embedded Workbench for several different toolchains in the same directory, the IDE might be in mixed languages if the toolchains are available in different languages.

Editor options

The **Editor** options are available by choosing **Tools>Options**.


IDE Options

- Common Fonts
- Key Bindings
- Editor
  - External Editor
  - Setup Files
  - Colors and Fonts
- Messages
- Project
- Source Code Control
- Debugger
- Stack
- Register Filter
- Terminal I/O

Tab size: 8

Indent size: 2

Tab Key Function: Insert tab [✓ Indent with spaces]

EOL characters: PC

[✓] Show right margin

Printing edge [✓ Columns] 80

[✓] Syntax highlighting

[✓] Auto indent

[ ] Show line numbers

[✓] Scan for changed files

[✓] Show bookmarks

[ ] Enable virtual space

[ ] Remove trailing blanks

Figure 57: Editor options

Use this page to configure the editor.

For more information about the editor, see *Editing*, page 49.

Tab size

Specify how wide a tab character is, in terms of character spaces.

Indent size

Specify the number of spaces to be used when tabulating with an indentation.

Tab Key Function

Controls what happens when you press the Tab key. Choose between:

Insert tab	Inserts a tab character when the Tab key is pressed.
Indent with spaces	Inserts an indentation (space characters) when the Tab key is pressed.

EOL character

Selects which line break character to be used when editor documents are saved. Choose between:

PC (default)	Windows and DOS end of line characters.
Unix	UNIX end of line characters.
Preserve	The same end of line character as the file had when it was opened, either PC or UNIX. If both types or neither type are present in the opened file, PC end of line characters are used.

Show right margin

Displays the area of the editor window outside the right margin as a light gray field. If this option is selected, you can set the width of the text area between the left margin and the right margin. Choose to set the width based on:

Printing edge	Bases the width on the printable area, which is taken from the general printer settings.
Columns	Bases the width based on the number of columns.

Syntax highlighting

Makes the editor display the syntax of C or C++ applications in different text styles.

To read more about syntax highlighting, see *Editor Colors and Fonts options*, page 131, and *Syntax coloring*, page 52.

Auto indent

Makes the editor indent the new line automatically when you press Return. For C/C++ source files, click the **Configure** button to configure the automatic indentation; see *Configure Auto Indent dialog box*, page 127. For all other text files, the new line will have the same indentation as the previous line.

Show line numbers

Makes the editor display line numbers in the editor window.

Scan for changed files

Makes the editor reload files that have been modified by another tool.

If a file is open in the IDE, and the same file has concurrently been modified by another tool, the file will be automatically reloaded in the IDE. However, if you already started to edit the file, you will be prompted before the file is reloaded.

Show bookmarks

Makes the editor display a column on the left side in the editor window, with icons for compiler errors and warnings, **Find in Files** results, user bookmarks, and breakpoints.

Enable virtual space

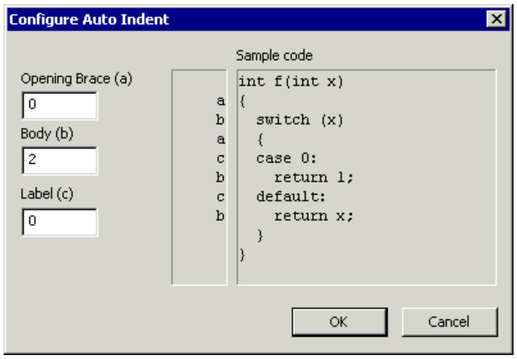
Allows the insertion point to move outside the text area.

Remove trailing blanks

Removes trailing blanks from files when they are saved to disk. Trailing blanks are blank spaces between the last non-blank character and the end of line character.

Configure Auto Indent dialog box

The **Configure Auto Indent** dialog box is available from the **IDE Options** dialog box.



Opening Brace (a): 0
Body (b): 2
Label (c): 0

```
  int f(int x)
a {
b   switch (x)
a   {
c   case 0:
b     return 1;
c   default:
b     return x;
    }
  }
```

Figure 58: Configure Auto Indent dialog box

Use this dialog box to configure the editor's automatic indentation of C/C++ source code.

To read more about indentation, see *Automatic text indentation*, page 52.

To open the Configure Auto Indent dialog box:

1 Choose **Tools>Options**.

2 Open the **Editor** page.

3 Select the **Auto indent** option and click the **Configure** button.

Opening Brace (a)

Specify the number of spaces used for indenting an opening brace.

Body (b)

Specify the number of additional spaces used for indenting code after an opening brace, or a statement that continues onto a second line.

Label (c)

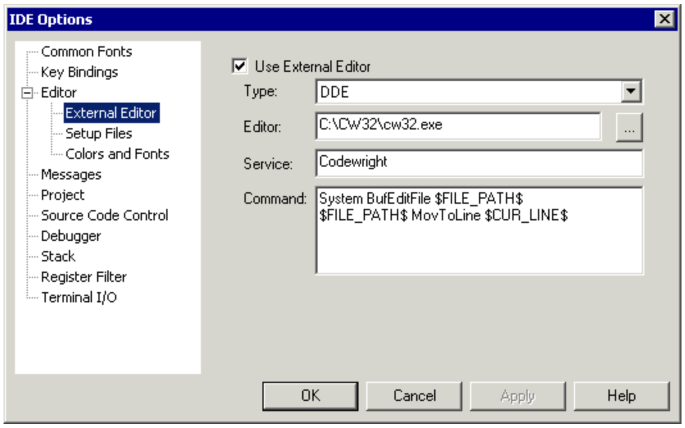
Specify the number of additional spaces used for indenting a label, including case labels.

Sample code

This area reflects the settings made in the text boxes for indentation. All indentations are relative to the preceding line, statement, or other syntactic structures.

External Editor options

The **External Editor** options are available by choosing **Tools>Options**.



☑ Use External Editor

Type: DDE

Editor: C:\CW32\cw32.exe

Service: Codewright

Command: System BufEditFile $FILE_PATH$ $FILE_PATH$ MovToLine CUR_LINE

Figure 59: External Editor options

Use this page to specify an external editor of your choice.

Note: The contents of this dialog box depends on the setting of the **Type** option.

See also *Using an external editor*, page 57.

Use External Editor

Enables the use of an external editor.

Type

Selects the type of interface. Choose between:

- **Command Line**
- **DDE** (Windows Dynamic Data Exchange).

Editor

Specify the filename and path of your external editor. A browse button is available for your convenience.

Arguments

Specify any arguments to be passed to the editor. This is only applicable if you have selected **Command Line** as the interface type, see *Type*, page 129.

Service

Specify the DDE service name used by the editor. This is only applicable if you have selected **DDE** as the interface type, see *Type*, page 129.

The service name depends on the external editor that you are using. Refer to the user documentation of your external editor to find the appropriate settings.

Command

Specify a sequence of command strings to be passed to the editor. The command strings should be typed as:

```
DDE-Topic CommandString1
DDE-Topic CommandString2
```

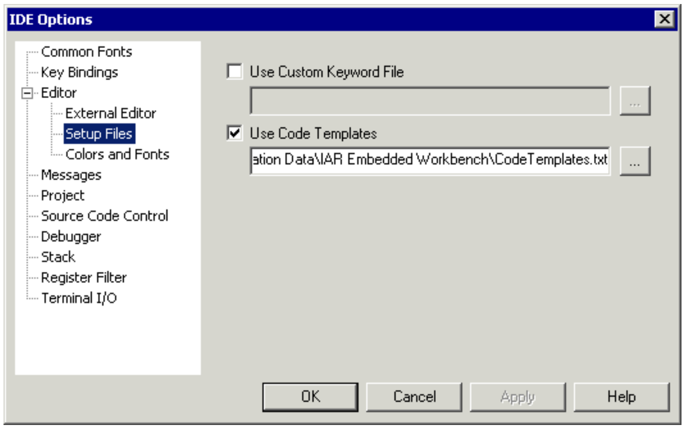
This is only applicable if you have selected **DDE** as the interface type, see *Type*, page 129.

The command strings depend on the external editor that you are using. Refer to the user documentation of your external editor to find the appropriate settings.

Note: You can use variables in arguments. See *Argument variables*, page 117, for information about available argument variables.

Editor Setup Files options

The **Editor Setup Files** options are available by choosing **Tools>Options**.



☐ Use Custom Keyword File

☑ Use Code Templates

ation Data\IAR Embedded Workbench\CodeTemplates.txt

Figure 60: Editor Setup Files options

Use this page to specify setup files for the editor.

Use Custom Keyword File

Specify a text file containing keywords that you want the editor to highlight. For information about syntax coloring, see *Syntax coloring*, page 52.

Use Code Templates

Specify a text file with code templates that you can use for inserting frequently used code in your source file. For information about using code templates, see *Using and adding code templates*, page 53.

Editor Colors and Fonts options

The **Editor Colors and Fonts** options are available by choosing **Tools>Options**.

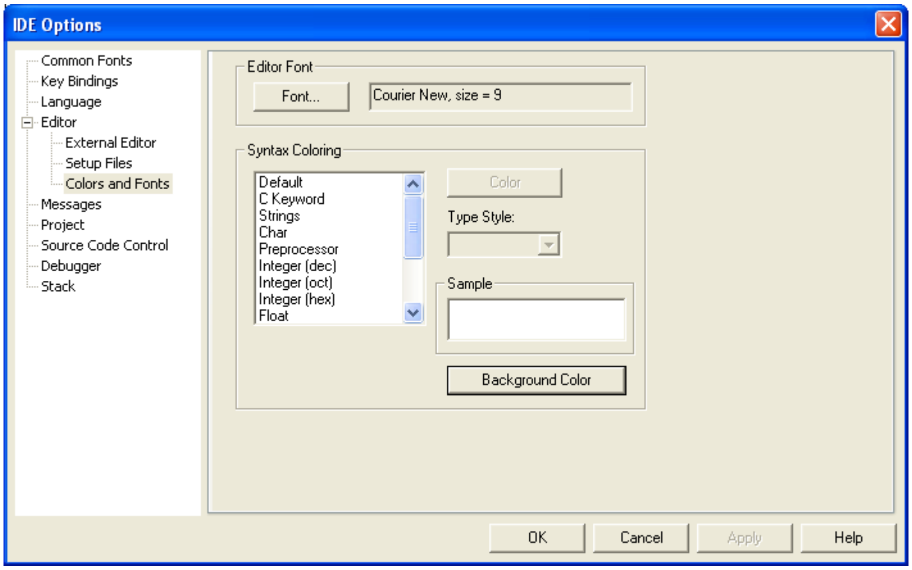


Figure 61: Editor Colors and Fonts options

Use this page to specify the colors and fonts used for text in the editor windows. The keywords controlling syntax highlighting for assembler and C or C++ source code are specified in the files `syntax_icc.cfg` and `syntax_asm.cfg`, respectively. These files are located in the `arm\config` directory.

Editor Font

Click the **Font** button to open the standard **Font** dialog box where you can choose the font and its size to be used in editor windows.

Syntax Coloring

Selects a syntax element in the list and sets the color and style for it:

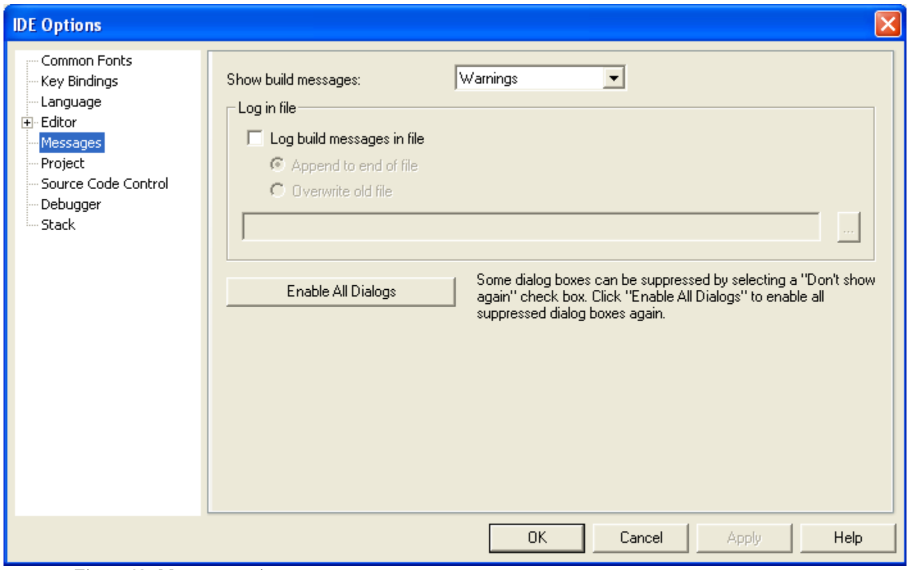
Color	Lists colors to choose from. Choose **Custom** from the list to define your own color.
Type Style	Select **Normal**, **Bold**, or **Italic** style for the selected element.
Sample	Displays the current appearance of the selected element.

Background Color Click to set the background color of the editor window.

Note: The **User keyword** syntax element refers to the keywords that you have listed in the custom keyword file; see *Use Custom Keyword File*, page 130.

Messages options

The **Messages** options are available by choosing **Tools>Options**.



Show build messages: Warnings

Log in file

[ ] Log build messages in file

(●) Append to end of file

( ) Overwrite old file

Some dialog boxes can be suppressed by selecting a "Don't show again" check box. Click "Enable All Dialogs" to enable all suppressed dialog boxes again.

Figure 62: Messages options

Use this page to choose the amount of output in the Build messages window.

Show build messages

Selects the amount of output to display in the Build messages window. Choose between:

All	Shows all messages, including compiler and linker information.
Messages	Shows messages, warnings, and errors.
Warnings	Shows warnings and errors.
Errors	Shows errors only.

Log in file

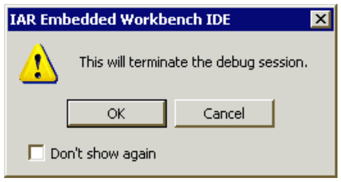
Select the **Log build messages in file** option to write build messages to a log file. Choose between:

Append to end of file	Appends the messages at the end of the specified file.
Overwrite old file	Replaces the contents in the file you specify.

Type the filename you want to use in the text box. A browse button is available for your convenience.

Enable All Dialogs

Enables all dialog boxes you have suppressed by selecting a **Don't show again** check box, for example:

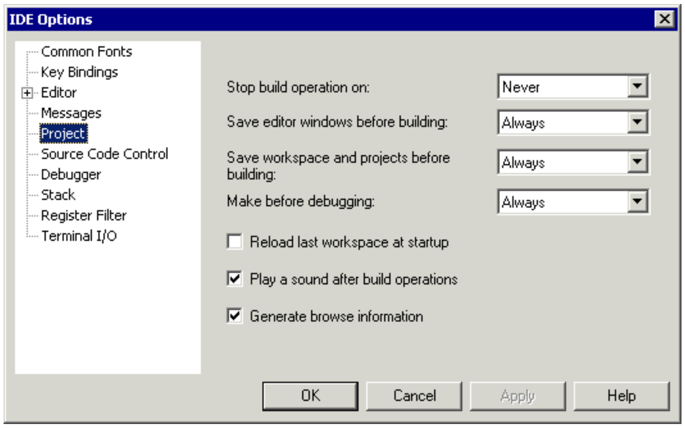


This will terminate the debug session.

Figure 63: Message dialog box containing a Don't show again option

Project options

The **Project** options are available by choosing **Tools>Options**.



Stop build operation on: Never

Save editor windows before building: Always

Save workspace and projects before building: Always

Make before debugging: Always

☐ Reload last workspace at startup

☑ Play a sound after build operations

☑ Generate browse information

Figure 64: Project options

Use this page to set options for the **Make** and **Build** commands.

Stop build operation on

Selects when the build operation should stop. Choose between:

Never	Never stops.
Warnings	Stops on warnings and errors.
Errors	Stops on errors.

Save editor windows before building

Selects when the editor windows should be saved before a build operation. Choose between:

Never	Never saves.
Ask	Prompts before saving.
Always	Always saves before Make or Build.

Save workspace and projects before building

Selects when a workspace and included projects should be saved before a build operation. Choose between:

Never	Never saves.
Ask	Prompts before saving.
Always	Always saves before Make or Build.

Make before debugging

Selects when a Make operation should be performed as you start a debug session. Choose between:

Never	Never performs a Make operation before debugging.
Ask	Prompts before performing a Make operation.
Always	Always performs a Make operation before debugging.

Reload last workspace at startup

Loads the last active workspace automatically the next time you start the IAR Embedded Workbench IDE.

Play a sound after build operations

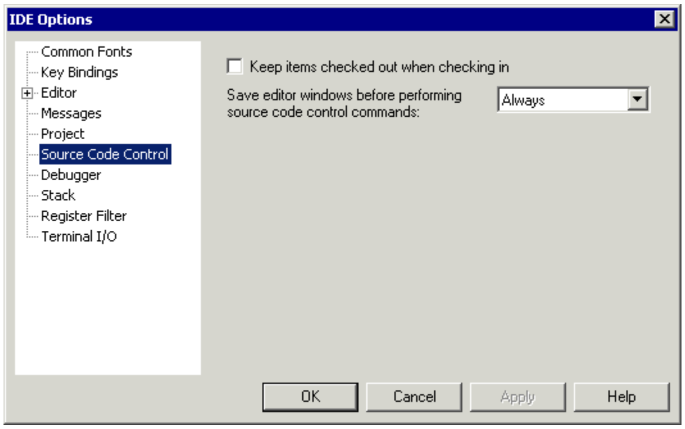
Plays a sound when the build operations are finished.

Generate browse information

Enables the use of the Source Browser window, see *Source Browser window*, page 87.

Source Code Control options

The **Source Code Control** options are available by choosing **Tools>Options**.



Keep items checked out when checking in

Save editor windows before performing source code control commands: Always

Figure 65: Source Code Control options

Use this page to configure the interaction between an IAR Embedded Workbench project and an SCC project.

Keep items checked out when checking in

Determines the default setting for the option **Keep Checked Out** in the **Check In Files** dialog box; see *Check In Files dialog box*, page 79.

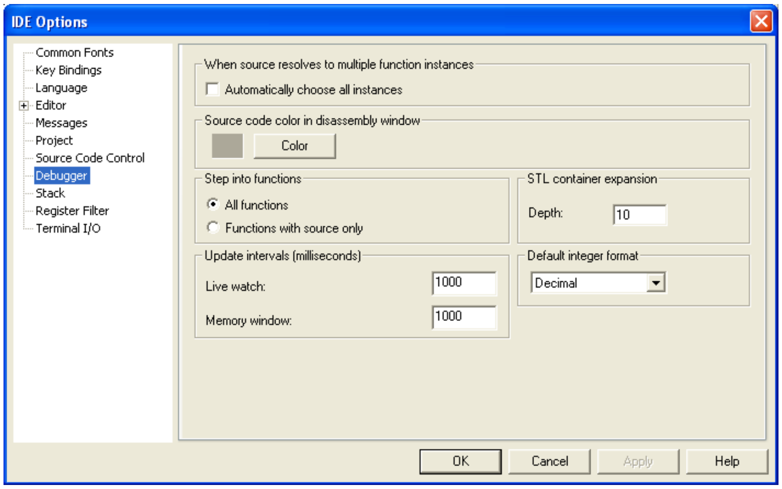
Save editor windows before performing source code control commands

Determines whether editor windows should be saved before you perform any source code control commands. Choose between:

Never	Never saves editor windows before performing any source code control commands.
Ask	Prompts before performing any source code control commands.
Always	Always saves editor windows before performing any source code control commands.

Debugger options

The **Debugger** options are available by choosing **Tools>Options**.



When source resolves to multiple function instances

- [ ] Automatically choose all instances

Source code color in disassembly window

Color

Step into functions

- [✓ All functions]
- Functions with source only

STL container expansion

Depth: 10

Update intervals (milliseconds)

Live watch: 1000

Memory window: 1000

Default integer format

Decimal

Figure 66: Debugger options

Use this page to configure the debugger environment.

When source resolves to multiple function instances

Some source code corresponds to multiple code instances, for example template code. When specifying a source location in such code, for example when setting a source breakpoint, you can make C-SPY act on all instances or a subset of instances. Use the **Automatically choose all instances** option to let C-SPY act on all instances without asking first.

Source code color in disassembly window

Click the **Color** button to select the color of for source code in the Disassembly window. To define your own color, choose **Custom** from the list.

Step into functions

Controls the behavior of the **Step Into** command. Choose between:

All functions	Makes the debugger step into all functions.

Functions with source only	Makes the debugger step only into functions for which the source code is known. This helps you avoid stepping into library functions or entering disassembly mode debugging.

STL container expansion

Specify how many elements that are shown initially when a container value is expanded in, for example, the Watch window.

Update intervals

Specify how often the contents of the Live Watch window and the Memory window are updated.

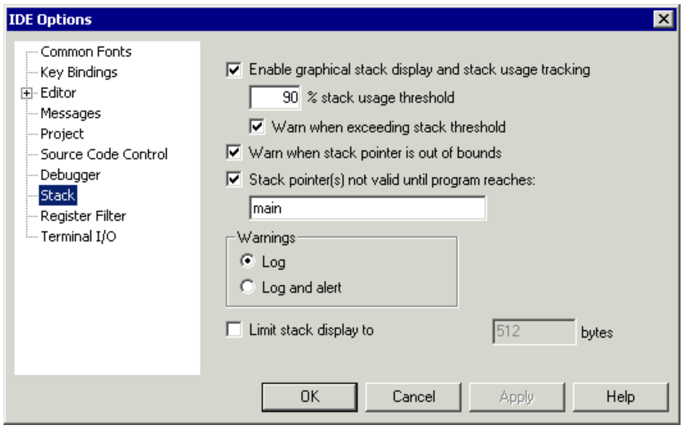
These text boxes are only available if the C-SPY driver you are using has access to the target system memory while executing your application.

Default integer format

Selects the default integer format in the Watch, Locals, and related windows.

Stack options

The **Stack** options are available by choosing **Tools>Options** or from the context menu in the Memory window.



[✓] Enable graphical stack display and stack usage tracking

90 % stack usage threshold

[✓] Warn when exceeding stack threshold

[✓] Warn when stack pointer is out of bounds

[✓] Stack pointer(s) not valid until program reaches:

main

Warnings

[✓ Log] Log and alert

[ ] Limit stack display to 512 bytes

Figure 67: Stack options

Use this page to set options specific to the Stack window.

Enable graphical stack display and stack usage tracking

Enables the graphical stack bar available at the top of the Stack window. It also enables detection of stack overflows. To read more about the stack bar and the information it provides, see the *C-SPY® Debugging Guide for ARM®*.

% stack usage threshold

Specify the percentage of stack usage above which C-SPY should issue a warning for stack overflow.

Warn when exceeding stack threshold

Makes C-SPY issue a warning when the stack usage exceeds the threshold specified in the **% stack usage threshold** option.

Warn when stack pointer is out of bounds

Makes C-SPY issue a warning when the stack pointer is outside the stack memory range.

Stack pointer(s) not valid until program reaches

Specify a *location* in your application code from where you want the stack display and verification to occur. The Stack window will not display any information about stack usage until execution has reached this location.

By default, C-SPY will not track the stack usage before the `main` function. If your application does not have a `main` function, for example, if it is an assembler-only project, you should specify your own start label. If this option is selected, after each reset C-SPY keeps a breakpoint on the given location until it is reached.

Typically, the stack pointer is set up in the system initialization code `cstartup`, but not necessarily from the very first instruction. Select this option to avoid incorrect warnings or misleading stack display for this part of the application.

Warnings

Selects where warnings should be issued. Choose between:

Log	Warnings are issued in the Debug Log window.
Log and alert	Warnings are issued in the Debug Log window and as alert dialog boxes.

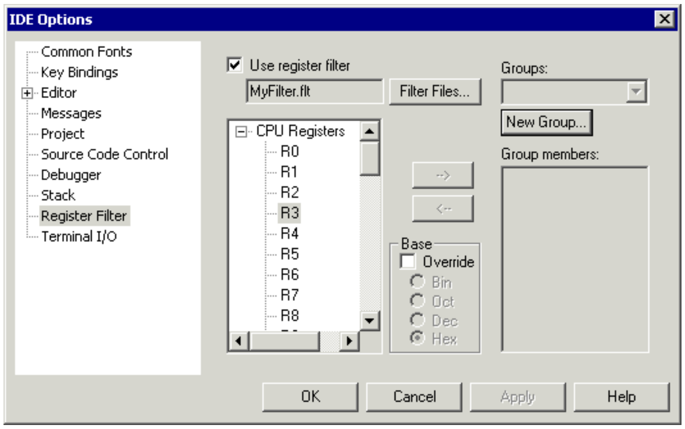
Limit stack display to

Limits the amount of memory displayed in the Stack window by specifying a number of bytes, counting from the stack pointer. This can be useful if you have a big stack or if you are only interested in the topmost part of the stack. Using this option can improve the Stack window performance, especially if reading memory from the target system is slow. By default, the Stack window shows the whole stack, or in other words, from the stack pointer to the bottom of the stack. If the debugger cannot determine the memory range for the stack, the byte limit is used even if the option is not selected.

Note: The Stack window does not affect the execution performance of your application, but it might read a large amount of data to update the displayed information when the execution stops.

Register Filter options

The **Register Filter** options are available by choosing **Tools>Options** when the debugger is running.



[✓] Use register filter: MyFilter.flt

Figure 68: Register Filter options

Use this page to display registers in the Register window in groups you have created yourself.

For more information about register groups, see the *C-SPY® Debugging Guide for ARM®*.

To define application-specific register groups:

1 Choose **Tools>Options>Register Filter**.

2 Specify the filename for your new group.

3 Click **New Group** and specify the name of the group.

4 Select the registers to be included using the arrow buttons.

5 Optionally, you can override the default integer base.

6 Your new group is now available in the Register window.

Use register filter

Enables the use of register filters.

Filter Files

Displays a dialog box where you can select or create a new filter file.

Groups

Lists all available register groups in the filter file, alternatively displays the new register group.

New Group

Click to create a new register group.

Group members

Shows the registers in the group currently selected in the **Groups** drop-down list.

To add registers to the group, select the registers you want to add in the list of all available registers to the left and move them using the arrow button.

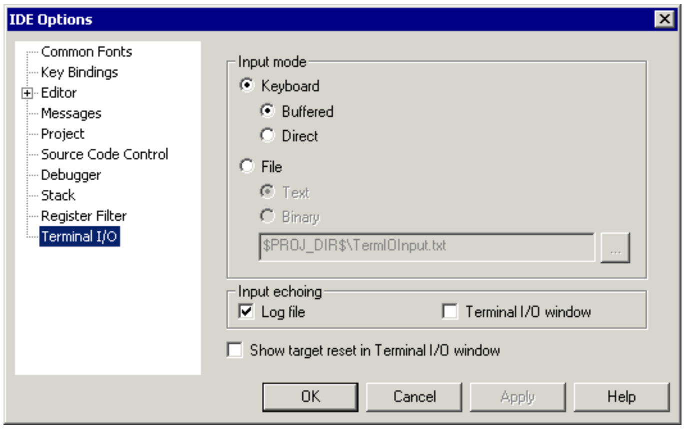
To remove registers from the group, select the registers you want to remove and move them using the arrow button.

Base

Overrides the default integer base.

Terminal I/O options

The **Terminal I/O** options are available by choosing **Tools>Options** when the debugger is running.



Input mode
[● Keyboard]
[● Buffered]
○ Direct
○ File
[● Text]
○ Binary
$PROJ_DIR$\TermIOInput.txt

Input echoing
[✓ Log file]
☐ Terminal I/O window

☐ Show target reset in Terminal I/O window

Figure 69: Terminal I/O options

Use this page to configure the C-SPY terminal I/O functionality.

Input mode

Controls how the terminal I/O input is read.

Keyboard	Makes the input characters be read from the keyboard. Choose between: **Buffered**: Buffers input characters. **Direct**: Does not buffer input characters.
File	Makes the input characters be read from a file. Choose between: **Text**: Reads input characters from a text file. **Binary**: Reads input characters from a binary file. A browse button is available for locating the input file.

Input echoing

Determines whether to echo the input characters and where to echo them. The choices are:

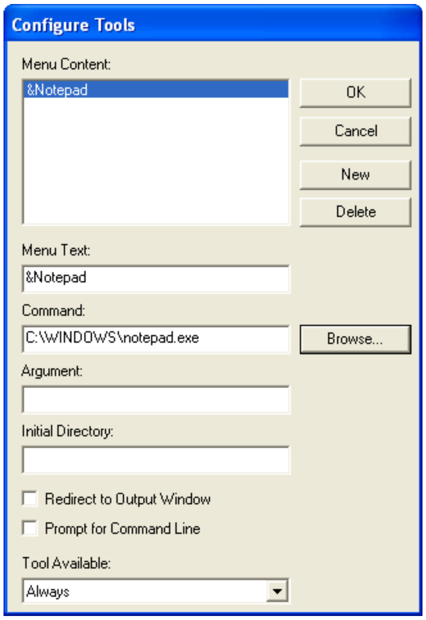
- **Log file**. Requires that you have enabled the option **Debug>Logging>Enable log file**.
- **Terminal I/O window**.

Show target reset in Terminal I/O window

Displays a message in the C-SPY Terminal I/O window when the target resets.

Configure Tools dialog box

The **Configure Tools** dialog box is available from the **Tools** menu.



Menu Content: &Notepad

Menu Text: &Notepad

Command: C:\WINDOWS\notepad.exe

Tool Available: Always

Figure 70: Configure Tools dialog box

Use this dialog box to specify a tool of your choice to add to the **Tools** menu, like this:

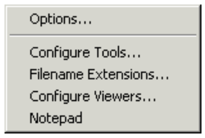


Figure 71: Customized Tools menu

Note: If you intend to add an external tool to the standard build toolchain, see *Extending the toolchain*, page 46.

You can use variables in the arguments, which allows you to set up useful tools such as interfacing to a command line revision control system, or running an external tool on the selected file.

Adding a command line command or batch file to the Tools menu:

1 Specify or browse to the `cmd.exe` command shell in the **Command** text box.

2 Specify the command line command or batch file name in the **Argument** text box.

The **Argument** text should be specified as:

```
/C name
```

where *`name`* is the name of the command or batch file you want to run.

The `/C` option terminates the shell after execution, to allow the IDE to detect when the tool has finished.

For an example, see *Adding command line commands*, page 28.

New

Creates a stub for a new menu command for you to configure using this dialog box.

Delete

Removes the command selected in the **Menu Content** list.

Menu Content

Lists all menu commands that you have defined.

Menu Text

Specify the name of the menu command. If you add the `&` sign anywhere in the name, the following letter, `N` in this example, will appear as the mnemonic key for this command. The text you specify will be reflected in the **Menu Content** list.

Command

Specify the tool and its path, to be run when you choose the command from the menu. A browse button is available for your convenience.

Argument

Optional: Specify an argument for the command.

Initial Directory

Specify an initial working directory for the tool.

Redirect to Output window

Makes the IDE send any console output from the tool to the **Tool Output** page in the message window. Tools that are launched with this option cannot receive any user input, for instance input from the keyboard.

Tools that *require* user input or make special assumptions regarding the console that they execute in, will *not* work at all if launched with this option.

Prompt for Command Line

Makes the IDE prompt for the command line argument when the command is chosen from the **Tools** menu.

Tool Available

Specifies in which context the tool should be available. Choose between:

- **Always**
- **When debugging**
- **When not debugging**.

Filename Extensions dialog box

The **Filename Extensions** dialog box is available from the **Tools** menu.

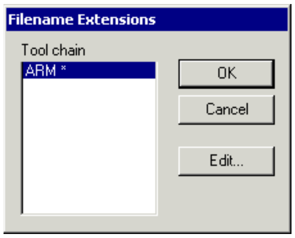


Figure 72: Filename Extensions dialog box

Use this dialog box to customize the filename extensions recognized by the build tools. This is useful if you have many source files with different filename extensions.

Toolchain

Lists the toolchains for which you have an IAR Embedded Workbench installed on your host computer. Select the toolchain you want to customize filename extensions for.

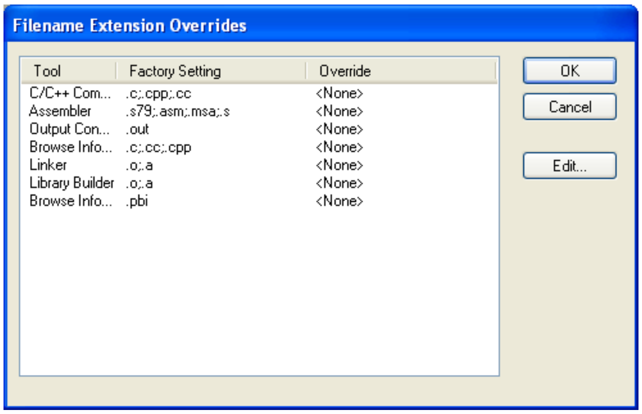
Note the `*` character which indicates user-defined overrides. If there is no `*` character, factory settings are used.

Edit

Displays the **Filename Extension Overrides** dialog box; see *Filename Extension Overrides dialog box*, page 147.

Filename Extension Overrides dialog box

The **Filename Extension Overrides** dialog box is available from the **Filename Extensions** dialog box.



Tool	Factory Setting	Override
C/C++ Com...	.c;.cpp;.cc	<None>
Assembler	.s79;.asm;.msa;.s	<None>
Output Con...	.out	<None>
Browse Info...	.c;.cc;.cpp	<None>
Linker	.o;.a	<None>
Library Builder	.o;.a	<None>
Browse Info...	.pbi	<None>

Figure 73: Filename Extension Overrides dialog box

This dialog box lists filename extensions recognized by the build tools.

Display area

This area contains these columns:

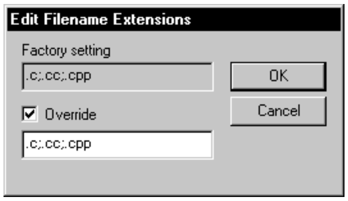
Tool	The available tools in the build chain.
Factory Setting	The filename extensions recognized by default by the build tool.
Override	The filename extensions recognized by the build tool if there are overrides to the default setting.

Edit

Displays the **Edit Filename Extensions** dialog box for the selected tool.

Edit Filename Extensions dialog box

The **Edit File Extensions** dialog box is available from the **Filename Extension Overrides** dialog box.



Factory setting: .c;.cc;.cpp
☑ Override: .c;.cc;.cpp

Figure 74: Edit Filename Extensions dialog box

This dialog box lists the filename extensions recognized by the IDE and lets you add new filename extensions.

Factory setting

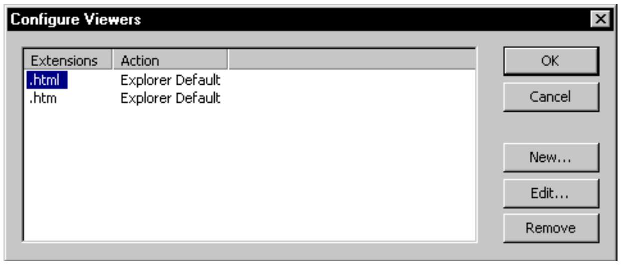
Lists the filename extensions recognized by default.

Override

Specify the filename extensions you want to be recognized. Extensions can be separated by commas or semicolons, and should include the leading period.

Configure Viewers dialog box

The **Configure Viewers** dialog box is available from the **Tools** menu.



Extensions	Action
.html	Explorer Default
.htm	Explorer Default

Figure 75: Configure Viewers dialog box

This dialog box lists overrides to the default associations between the document formats that IAR Embedded Workbench can handle and viewer applications.

Display area

This area contains these columns:

Extensions	Explicitly defined filename extensions of document formats that IAR Embedded Workbench can handle.
Action	The viewer application that is used for opening the document type. `Explorer Default` means that the default application associated with the specified type in Windows Explorer is used.

New

Displays the **Edit Viewer Extensions** dialog box.

Edit

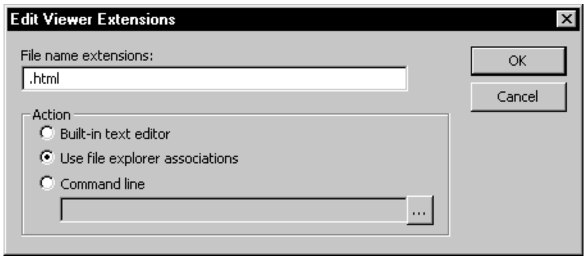
Displays the **Edit Viewer Extensions** dialog box.

Delete

Removes the association between the selected filename extensions and the viewer application.

Edit Viewer Extensions dialog box

The **Edit Viewer Extensions** dialog box is available from the **Configure Viewers** dialog box.


Edit Viewer Extensions

File name extensions: .html

Action
- ( ) Built-in text editor
- (●) Use file explorer associations
- ( ) Command line

Figure 76: Edit Viewer Extensions dialog box

Use this dialog box to specify how to open a new document type or edit the setting for an existing document type.

File name extensions

Specify the filename extension for the document type—including the separating period (.).

Action

Selects how to open documents with the filename extension specified in the **Filename extensions** text box. Choose between:

Built-in text editor	Opens all documents of the specified type with the IAR Embedded Workbench text editor.
Use file explorer associations	Opens all documents of the specified type with the default application associated with the specified type in Windows Explorer.
Command line	Opens all documents of the specified type with the viewer application you type or browse your way to. You can give any command line options you would like to the tool.

Window menu

The **Window** menu provides commands for manipulating the IDE windows and changing their arrangement on the screen.

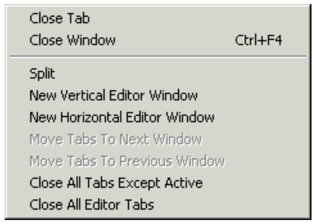

Close Tab
Close Window Ctrl+F4
Split
New Vertical Editor Window
New Horizontal Editor Window
Move Tabs To Next Window
Move Tabs To Previous Window
Close All Tabs Except Active
Close All Editor Tabs

Figure 77: Window menu

The last section of the **Window** menu lists the currently open windows. Choose the window you want to switch to.

These commands are available:

Close Tab	Closes the active tab.
Close Window Ctrl+F4	Closes the active editor window.
Split	Splits an editor window horizontally or vertically into two or four panes, which means that you can see more parts of a file simultaneously.
New Vertical Editor Window	Opens a new empty window next to the current editor window.
New Horizontal Editor Window	Opens a new empty window under the current editor window.
Move Tabs To Next Window	Moves all tabs in the current window to the next window.
Move Tabs To Previous Window	Moves all tabs in the current window to the previous window.
Close All Tabs Except Active	Closes all the tabs except the active tab.
Close All Editor Tabs	Closes all tabs currently available in editor windows.

Help menu

The **Help** menu provides help about IAR Embedded Workbench and displays the version numbers of the user interface and of the IDE.

You can also access the Information Center from the **Help** menu. The Information Center is an integrated navigation system that gives easy access to the information resources you need to get started and during your project development: tutorials, example projects, user guides, support information, and release notes. It also provides shortcuts to useful sections on the IAR Systems web site.

General options

This chapter describes the general options in the IAR Embedded Workbench® IDE.

For information about how to set options, see *Setting options*, page 41.

Description of general options

This section gives detailed information about the options in the **General Options** category.

To set general options in the IDE:

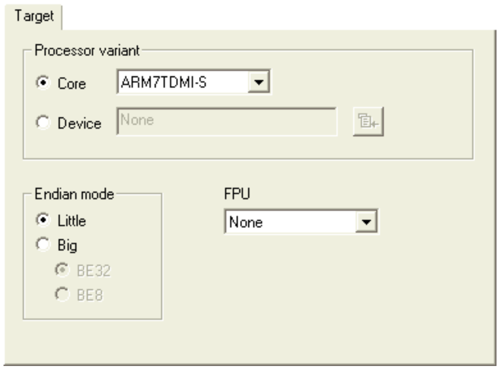
1 Choose **Project>Options** to display the **Options** dialog box.

2 Select **General Options** in the **Category** list.

3 To restore all settings to the default factory settings, click the **Factory Settings** button.

Target options

The **Target** options specify target-specific features for the IAR C/C++ Compiler and Assembler.



[✓ Core] ARM7TDMI-S

Endian mode: [✓ Little] Big — [✓ BE32] BE8

FPU: None

Figure 78: General target options

Processor variant

Selects the processor variant:

Core	The processor core you are using. For a description of the available variants, see the *IAR C/C++ Development Guide for ARM®* .
Device	The device your are using. The choice of device will automatically determine the default C-SPY® device description file. For information about how to override the default files, see the *C-SPY® Debugging Guide for ARM®*.

Endian mode

Selects the byte order for your project:

Little	The lowest byte is stored at the lowest address in memory. The highest byte is the most significant; it is stored at the highest address.
Big	The lowest address holds the most significant byte, while the highest address holds the least significant byte. Choose between two variants of the big-endian mode: **BE8** to make data big-endian and code little-endian **BE32** to make both data and code big-endian.

FPU

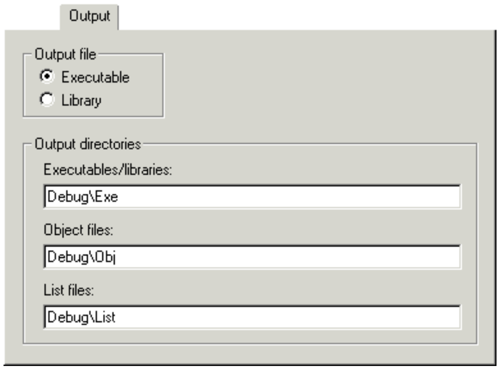
Selects the floating-point unit:

None (default)	The software floating-point library is used.
VFPv1	A VFP unit conforming to architecture VFPv1.
VFPv2	A VFP unit conforming to architecture VFPv2.
VFPv3	A VFP unit conforming to architecture VFPv3.
VFPv4	A VFP unit conforming to architecture VFPv4.
VFP9-S	A VFPv2 architecture that can be used with the ARM9E family of CPU cores. Selecting this coprocessor is therefore identical to selecting the VFPv2 architecture.

By selecting a VFP coprocessor, you will override the use of the software floating-point library for all supported floating-point operations.

Output

The **Output** options determine the type of output file. You can also specify the destination directories for executable files, object files, and list files.



Executables/libraries: Debug\Exe

Object files: Debug\Obj

List files: Debug\List

Figure 79: Output options

Output file

Selects the type of the output file:

Executable (default)	As a result of the build process, the linker will create an *application* (an executable output file). When this setting is used, linker options will be available in the **Options** dialog box. Before you create the output you should set the appropriate linker options.
Library	As a result of the build process, the library builder will create a *library file*. When this setting is used, library builder options will be available in the **Options** dialog box, and **Linker** will disappear from the list of categories. Before you create the library you can set the options.

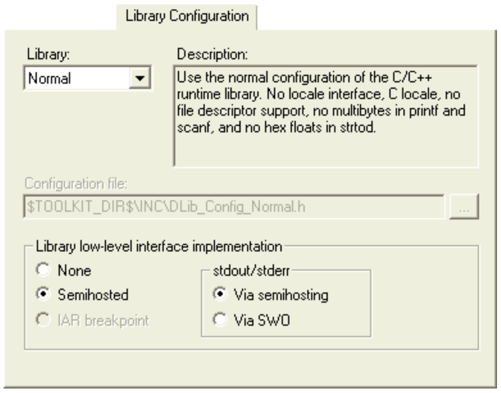
Output directories

Specify the paths to the destination directories. Note that incomplete paths are relative to your project directory. You can specify:

Executables/libraries	Overrides the default directory for executable or library files. Type the name of the directory where you want to save executable files for the project.
Object files	Overrides the default directory for object files. Type the name of the directory where you want to save object files for the project.
List files	Overrides the default directory for list files. Type the name of the directory where you want to save list files for the project.

Library Configuration

The **Library Configuration** options determine which library to use.



Library: Normal

Description: Use the normal configuration of the C/C++ runtime library. No locale interface, C locale, no file descriptor support, no multibytes in printf and scanf, and no hex floats in strtod.

Configuration file: $TOOLKIT_DIR$\INC\DLib_Config_Normal.h

Library low-level interface implementation: [✓ Semihosted]

stdout/stderr: [✓ Via semihosting]

Figure 80: Library Configuration options

For information about the runtime library, library configurations, the runtime environment they provide, and the possible customizations, see *IAR C/C++ Development Guide for ARM®* .

Library

Selects which runtime library to use. For information about available libraries, see the *IAR C/C++ Development Guide for ARM®* .

The names of the library object file and library configuration file that actually will be used are displayed in the **Library file** and **Configuration file** text boxes, respectively.

Configuration file

Displays the library configuration file that will be used. A library configuration file is chosen automatically depending on the project settings. If you have chosen **Custom** in the **Library** drop-down list, you must specify your own library configuration file.

Library low-level interface implementation

Selects what type of low-level interface for I/O to be included in the library.

For Cortex-M, choose between:

None	No low-level support for I/O available in the libraries. You must provide your own `__write` function to use the I/O functions part of the library.
Semihosted and **stdout/stderr via semihosting**	Semihosted I/O which uses the `BKPT` instruction.
Semihosted and **stdout/stderr via SWO**	Semihosted I/O which uses the `BKPT` instruction for all functions except for the `stdout` and `stderr` output where the SWO interface—available on some J-Link debug probes—is used. This means a much faster mechanism where the application does not need to halt execution to transfer data.
IAR breakpoint	Not available.

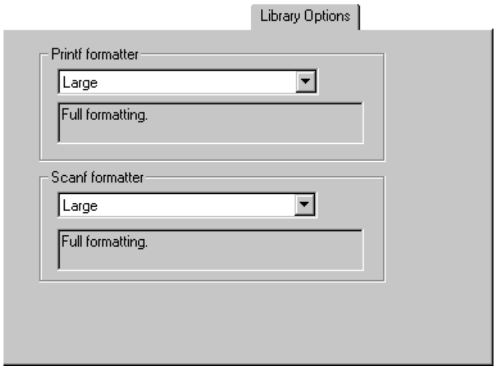
For other cores, choose between:

None	No low-level support for I/O available in the libraries. You must provide your own `__write` function to use the I/O functions part of the library.
Semihosted	Semihosted I/O which uses the `SVC` instruction (earlier `SWI`).

IAR breakpoint The IAR proprietary variant of semihosting, which does not use the `SVC` instruction and thus does not need to set a breakpoint on the `SVC` vector. This is an advantage for applications which require the `SVC` vector for their own use, for example an RTOS. This method can also lead to performance improvements. However, note that this method does not work with applications, libraries, and object files that are built using tools from other vendors.

Library Options

The **Library Options** select the `printf` and `scanf` formatters.



Printf formatter: Large

Full formatting.

Scanf formatter: Large

Full formatting.

Figure 81: Library Options

See the *IAR C/C++ Development Guide for ARM®* for more information about the formatting capabilities.

Printf formatter

The full formatter version is memory-consuming, and provides facilities that are not required in many embedded applications. To reduce the memory consumption, alternative versions are also provided:

Printf formatters in the library are: **Full**, **Large**, **Small**, and **Tiny**.

Scanf formatter

The full formatter version is memory-consuming, and provides facilities that are not required in many embedded applications. To reduce the memory consumption, alternative versions are also provided:

Scanf formatters in the library are: **Full**, **Large**, and **Small**.

MISRA C

The **MISRA-C:1998** and **MISRA-C:2004** options control how the IDE checks the source code for deviations from the MISRA C rules. The settings are used for both the compiler and the linker.

For details about specific options, see the *IAR Embedded Workbench® MISRA C:2004 Reference Guide* or the *IAR Embedded Workbench® MISRA C:1998 Reference Guide* available from the **Help** menu.

Compiler options

This chapter describes the compiler options available in the IAR Embedded Workbench® IDE.

For information about how to set options, see *Setting options*, page 41.

Description of compiler options

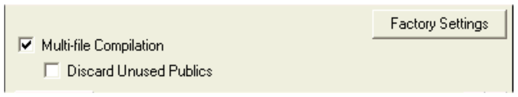
This section gives detailed information about the options in the **C/C++ Compiler** category.

To set compiler options in the IDE:

1. Choose **Project>Options** to display the **Options** dialog box.
2. Select **C/C++ Compiler** in the **Category** list.
3. To restore all settings to the default factory settings, click the **Factory Settings** button.

Multi-file Compilation

Before you set specific compiler options, you can decide whether you want to use multi-file compilation, which is an optimization technique. **Multi-file compilation** enables multi-file compilation from the group of project files that you have selected in the workspace window.



[✓] Multi-file Compilation

[ ] Discard Unused Publics

Figure 82: Multi-file Compilation

You can use this option for the entire project or for individual groups of files. All C/C++ source files in such a group are compiled together using one invocation of the compiler.

Discard Unused Publics

Discards any unused public functions and variables from the compilation unit.

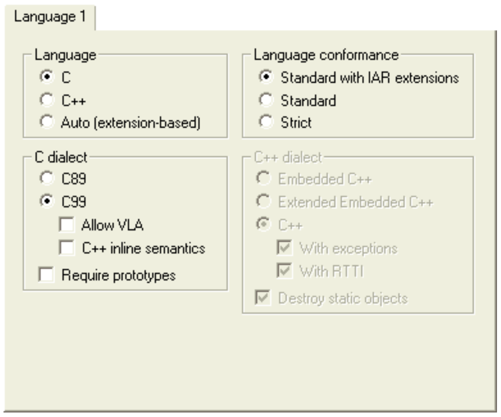
This means that all files included in the selected group are compiled using the compiler options which have been set on the group or nearest higher enclosing node which has any options set. Any overriding compiler options on one or more files are ignored when building, because a group compilation must use exactly one set of options.

For information about how multi-file compilation is displayed in the workspace window, see *Workspace window*, page 72.

For more information about multi-file compilation and discarding unused public functions, see the *IAR C/C++ Development Guide for ARM®* .

Language 1

The **Language 1** options determine which programming language to use and which extensions to enable.



Language 1

Language
- (●) C
- ( ) C++
- ( ) Auto (extension-based)

Language conformance
- (●) Standard with IAR extensions
- ( ) Standard
- ( ) Strict

C dialect
- ( ) C89
- (●) C99
  - [ ] Allow VLA
  - [ ] C++ inline semantics
- [ ] Require prototypes

C++ dialect
- ( ) Embedded C++
- ( ) Extended Embedded C++
- (●) C++
  - [✓] With exceptions
  - [✓] With RTTI
- [✓] Destroy static objects

Figure 83: Compiler language options

For more information about these options, see the *IAR C/C++ Development Guide for ARM®* .

Language

Determines the compiler support for either C or C++:

C (default)	Makes the compiler treat the source code as C, which means that features specific to C++ cannot be used.
C++	Makes the compiler treat the source code as C++.

Auto	Language support is decided automatically depending on the filename extension of the file being compiled: c, files with this filename extension are treated as C source files. cpp, files with this filename extension will be treated as C++ source files.

C dialect

Selects the dialect if **C** is the supported language:

C89	Enables the C89 standard instead of Standard C. Note that this setting is mandatory when the MISRA C checking is enabled.
C99	Enables the C99 standard, also known as Standard C. This is the default standard used in the compiler, and it is stricter than C89. Features specific to C89 cannot be used. Choose between: **Allow VLA**, allows the use of C99 variable length arrays. **C++ inline semantics**, enables C++ inline semantics when compiling a Standard C source code file.
Require prototypes	Forces the compiler to verify that all functions have proper prototypes, which means that source code containing any of the following will generate an error: ● A function call of a function with no declaration, or with a Kernighan & Ritchie C declaration. ● A function definition of a public function with no previous prototype declaration. ● An indirect function call through a function pointer with a type that does not include a prototype.

C++ dialect

Selects the dialect if **C++** is the supported language:

Embedded C++	Makes the compiler treat the source code as Embedded C++. This means that features specific to C++, such as classes and overloading, can be used.

Extended Embedded C++	Enables features like namespaces or the standard template library in your source code.
C++	Makes the compiler treat the source code as Standard C++. Choose between: **With exceptions**, enables exception support in the C++ language. **With RTTI**, enables runtime type information (RTTI) support in the C++ language.
Destroy static objects	Makes the compiler generate code to destroy C++ static variables that require destruction at program exit.

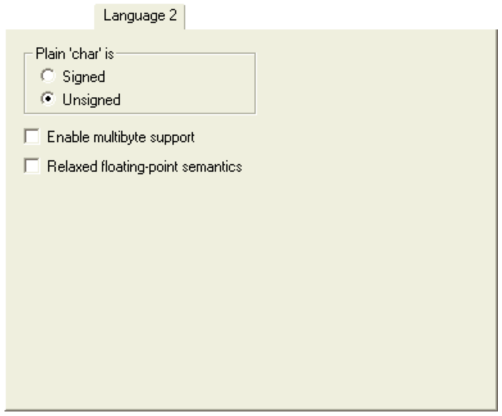
Language conformance

Controls how strictly the compiler adheres to the standard C or C++ language:

Standard with IAR extensions	Accepts ARM-specific keywords as extensions to the standard C or C++ language. In the IDE, this setting is enabled by default.
Standard	Disables IAR Systems extensions, but does not adhere strictly to the C or C++ dialect you have selected. Some very useful relaxations to C or C++ are still available.
Strict	Adheres strictly to the C or C++ dialect you have selected. This setting disables a great number of useful extensions and relaxations to C or C++.

Language 2

The **Language 2** options determine which programming language to use and which extensions to enable.



Plain 'char' is
- Signed
- [✓ Unsigned]

- [ ] Enable multibyte support
- [ ] Relaxed floating-point semantics

Figure 84: Compiler language2 options

Plain 'char' is

Normally, the compiler interprets the plain `char` type as `unsigned char`. **Plain 'char' is Signed** makes the compiler interpret the `char` type as `signed char` instead, for example for compatibility with another compiler.

Note: The runtime library is compiled with unsigned plain characters. If you select the **Signed** option, references to library functionality that uses plain unsigned characters will not work.

Enable multibyte support

By default, multibyte characters cannot be used in C or Embedded C++ source code. **Enable multibyte support** makes the compiler interpret multibyte characters in the source code according to the host computer's default setting for multibyte support.

Multibyte characters are allowed in C and C++ style comments, in string literals, and in character constants. They are transferred untouched to the generated code.

Relaxed floating-point semantics

Makes the compiler relax the language rules and perform more aggressive optimization of floating-point expressions. This option improves performance for floating-point expressions that fulfill these conditions:

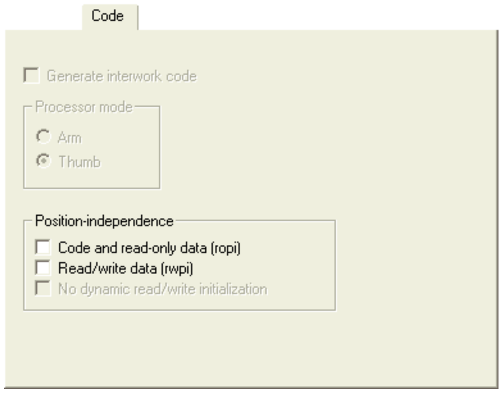
- The expression consists of both single- and double-precision values

- The double-precision values can be converted to single precision without loss of accuracy
- The result of the expression is converted to single precision.

Note that performing the calculation in single precision instead of double precision might cause a loss of accuracy.

Code

The **Code** options control the code generation of the compiler.



Generate interwork code

Processor mode
Arm
[✓ Thumb]

Position-independence
Code and read-only data (ropi)
Read/write data (rwpi)
No dynamic read/write initialization

Figure 85: C ode options

For detailed reference information about these compiler options, see the *IAR C/C++ Development Guide for ARM®* .

Generate interwork code

Makes the compiler be able to mix ARM and Thumb code. This option is enabled by default.

Processor mode

Selects the processor mode for your project:

Arm	Generates code that uses the full 32-bit instruction set.
Thumb	Generates code that uses the reduced 16-bit instruction set. Thumb code minimizes memory usage and provides higher performance in 8/16-bit bus environments.

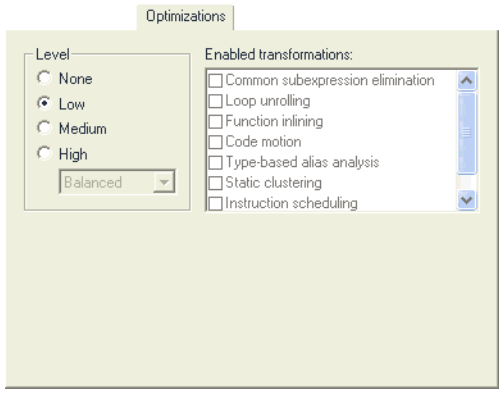
Position-independence

Determines how the compiler should handle position-independent code and data:

Code and read-only data (ropi)	Generates code that uses PC-relative references to address code and read-only data.
Read/write data (rwpi)	Generates code that uses an offset from the static base register to address-writable data.
No dynamic read/write initialization	Disables runtime initialization of static C variables.

Optimizations

The **Optimizations** options determine the type and level of optimization for the generation of object code.



Level: None [✓ Low] Medium High

Figure 86: Optimizations options

Level

Selects the optimization level:

None	No optimization; provides best debug support.
Low	The lowest level of optimization.
Medium	The medium level of optimization.
High, balanced	The highest level of optimization, balancing between speed and size.
High, speed	The highest level of optimization, favors speed.
High, size	The highest level of optimization, favors size.

By default, a debug project will have a size optimization that is fully debuggable, while a release project will have a high balanced optimization that generates small code without sacrificing speed.

For a list of optimizations performed at each optimization level, see the *IAR C/C++ Development Guide for ARM®* .

Enabled transformations

Selects which transformations that are available at different optimization levels. When a transformation is available, you can enable or disable it by selecting its check box. Choose between:

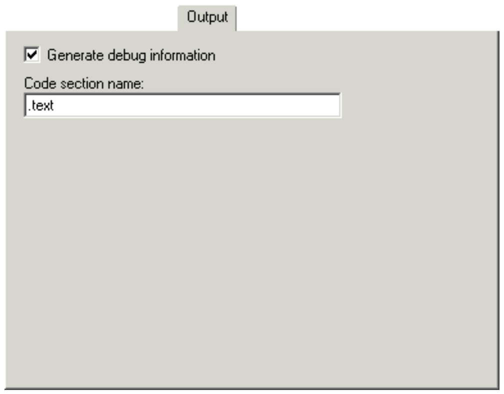
- Common subexpression elimination
- Loop unrolling
- Function inlining
- Code motion
- Type-based alias analysis
- Static variable clustering
- Instruction scheduling.

In a *debug* project the transformations are, by default, disabled. In a *release* project the transformations are, by default, enabled.

For a brief description of the transformations that can be individually disabled, see the *IAR C/C++ Development Guide for ARM®* .

Output

The **Output** options determine the generated compiler output.



☑ Generate debug information

Code section name:
.text

Figure 87: Compiler output options

Generate debug information

Makes the compiler include additional information in the object modules that is required by C-SPY® and other symbolic debuggers.

Generate debug information is selected by default. Deselect it if you do not want the compiler to generate debug information.

Note: The included debug information increases the size of the object files.

Code section name

The compiler places functions into named sections which are referred to by the IAR ILINK Linker. **Code section name** specifies a different name than the default name to place any part of your application source code into separate non-default sections. This is useful if you want to control placement of your code to different address ranges and you find the `@` notation, alternatively the `#pragma location` directive, insufficient.

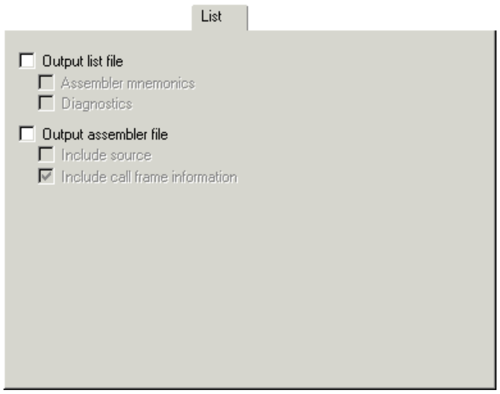
Note: Take care when you explicitly place a function in a predefined section other than the one used by default. This is useful in some situations, but incorrect placement can result in anything from error messages during compilation and linking to a malfunctioning application. Carefully consider the circumstances; there might be strict requirements on the declaration and use of the function or variable.

Note that any changes to the section names require a corresponding modification in the linker configuration file.

For detailed information about sections and the various methods for controlling the placement of code, see the *IAR C/C++ Development Guide for ARM®* .

List

The **List** options make the compiler generate a list file and determine its contents.



☐ Output list file
☐ Assembler mnemonics
☐ Diagnostics
☐ Output assembler file
☐ Include source
☑ Include call frame information

Figure 88: Compiler list file options

By default, the compiler does not generate a list file. Select any of the following options to generate a list file or an assembler file. The list file will be saved in the `List` directory, and its filename will consist of the source filename, plus the filename extension `lst`.

If you want to save the list file in another directory than the default directory for list files, use the **Output Directories** option in the **General Options** category; see *Output*, page 155, for additional information.

You can open the output files directly from the **Output** folder which is available in the Workspace window.

Output list file

Makes the compiler generate a list file. You can open the output files directly from the **Output** folder which is available in the Workspace window. By default, the assembler does not generate a list file. For the list file content, choose between:

Assembler mnemonics	Includes assembler mnemonics in the list file.
Diagnostics	Includes diagnostic information in the list file.

Output assembler file

Makes the compiler generate an assembler list file. For the list file content, choose between:

Include source	Includes source code in the assembler file.
Include call frame information	Includes compiler-generated information for runtime model attributes, call frame information, and frame size information.

Preprocessor

The **Preprocessor** options allow you to define symbols and include paths for use by the compiler.

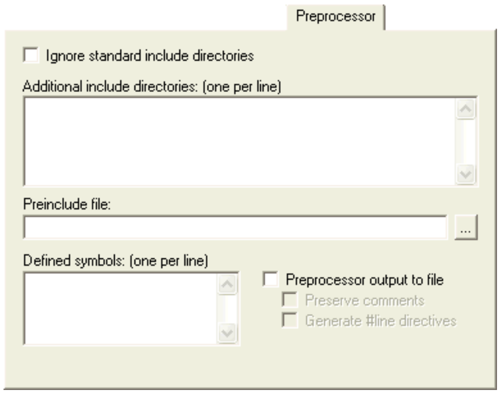

Preprocessor
Ignore standard include directories
Additional include directories: (one per line)
Preinclude file:
...
Defined symbols: (one per line)
Preprocessor output to file
Preserve comments
Generate #line directives

Figure 89: Compiler preprocessor options

Ignore standard include directories

Ignores the standard include files, which will not be used when the project is built.

Additional include directories

Specify the full file path to the list of `#include` file paths. The paths required by the product are specified automatically based on your choice of runtime library.

Note: Any directories you specify are searched before the standard include directories.

To make your project more portable, use the argument variable `$TOOLKIT_DIR$` for the subdirectories of the active product and `$PROJ_DIR$` for the directory of the current project. For an overview of the argument variables, see *Argument variables*, page 117.

Preinclude file

Makes the compiler include the include file you specify before the compiler starts to read the source file. This is useful if you want to change something in the source code for the entire application, for instance if you want to define a new symbol.

Defined symbols

Define symbols, which you otherwise must define in the source file. Type the symbols you want to define, one per line, and specify their values.

For example:

```
TESTVER=1
```

Note that there should be no space around the equal sign.

You might also want to arrange your source to produce either the test or production version of your application depending on whether the symbol `TESTVER` was defined. To do this you would use include sections such as:

```
#ifdef  TESTVER
  ...  ; additional code lines for test version only
#endif
```

You would then define the symbol `TESTVER` in the Debug target but not in the Release target.

Defined symbols has the same effect as a `#define` statement at the top of the source file.

Preprocessor output to file

Generates preprocessor output to a file with the filename extension `i` and located in the `lst` directory. By default, the compiler does not generate preprocessor output. Choose between:

Preserve comments	Preserves comments.
Generate #line directives	Generates #line directives.

Diagnostics

The **Diagnostics** options determine how diagnostic messages are classified and displayed. Use the diagnostics options to override the default classification of the specified diagnostics.

Note: The diagnostic messages cannot be suppressed for fatal errors, and fatal errors cannot be reclassified.

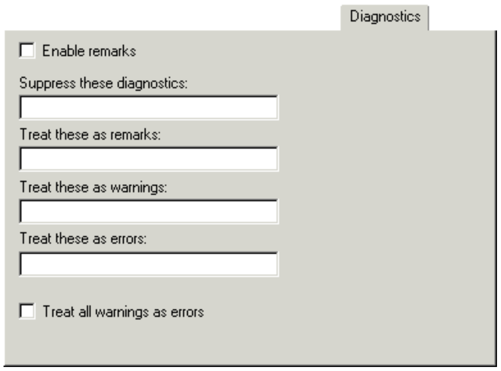


Figure 90: Compiler diagnostics options

Enable remarks

Makes the compiler generate remarks. By default, remarks are not issued.

The least severe diagnostic messages are called *remarks*. A remark indicates a source code construct that might cause strange behavior in the generated code.

Suppress these diagnostics

Suppresses the output of diagnostic messages for the tags that you specify.

For example, to suppress the warnings `Pe117` and `Pe177`, type:

```
Pe117,Pe177
```

Treat these as remarks

Classifies diagnostic messages as remarks. A remark is the least severe type of diagnostic message. It indicates a source code construct that might cause strange behavior in the generated code.

For example, to classify the warning `Pe177` as a remark, type:

```
Pe177
```

Treat these as warnings

Classifies diagnostic messages as warnings. A *warning* indicates an error or omission that is of concern, but which will not cause the compiler to stop before compilation is completed.

For example, to classify the remark `Pe826` as a warning, type:

```
Pe826
```

Treat these as errors

Classifies diagnostic messages as errors. An *error* indicates a violation of the C or C++ language rules, of such severity that object code will not be generated, and the exit code will be non-zero.

For example, to classify the warning `Pe117` as an error, type:

```
Pe117
```

Treat all warnings as errors

Classifies all warnings as errors. If the compiler encounters an error, object code is not generated.

MISRA C

The **MISRA-C:1998** and **MISRA-C:2004** options override the corresponding options in the **General Options** category.

For details about specific option, see the *IAR Embedded Workbench® MISRA C:2004 Reference Guide* or the *IAR Embedded Workbench® MISRA C:1998 Reference Guide* available from the **Help** menu.

Extra Options

The **Extra Options** page provides you with a command line interface to the compiler.

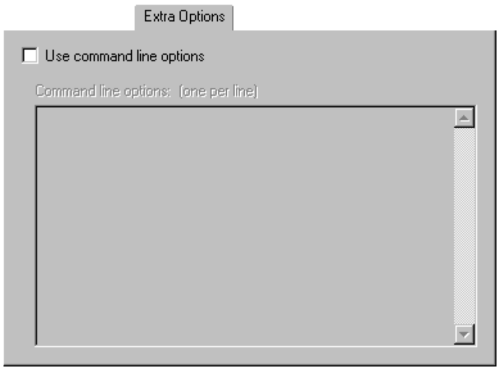


Figure 91: Compiler extra options

Use command line options

Specify additional command line arguments to be passed to the compiler (not supported by the GUI).

Assembler options

This chapter describes the assembler options available in the IAR Embedded Workbench® IDE.

For information about how to set options, see *Setting options*, page 41.

Description of assembler options

This section gives detailed information about the options in the **Assembler** category.

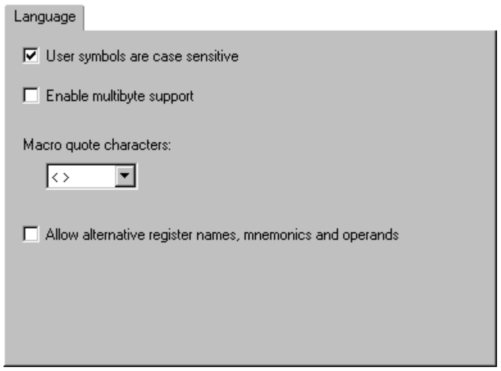
To set assembler options in the IDE:

1 Choose **Project>Options** to display the **Options** dialog box.

2 Select **Assembler** in the **Category** list.

Language

The **Language** options control certain behavior of the assembler language.



[✓] User symbols are case sensitive

[ ] Enable multibyte support

Macro quote characters: < >

[ ] Allow alternative register names, mnemonics and operands

Figure 92: Assembler language options

User symbols are case sensitive

Toggles case sensitivity on and off. By default, case sensitivity is on. This means that, for example, `LABEL` and `label` refer to different symbols. When case sensitivity is off, `LABEL` and `label` will refer to the same symbol.

Enable multibyte support

Makes the assembler interpret multibyte characters in the source code according to the host computer's default setting for multibyte support. By default, multibyte characters cannot be used in assembler source code.

Multibyte characters are allowed in comments, in string literals, and in character constants. They are transferred untouched to the generated code.

Macro quote characters

Selects the characters used for the left and right quotes of each macro argument. By default, the characters are < and >.

Macro quote characters changes the quote characters to suit an alternative convention or simply to allow a macro argument to contain < or >.

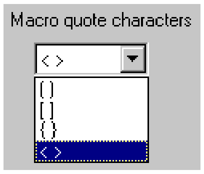


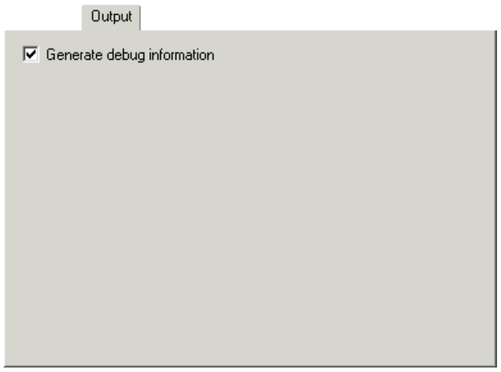
Figure 93: Choosing macro quote characters

Allow alternative register names, mnemonics and operands

To enable migration from an existing application to the IAR Assembler for ARM, alternative register names, mnemonics, and operands can be allowed. This is controlled by the assembler command line option `-j`. Use this option for assembler source code written for the ARM ADS/RVCT Assembler. For more information, see the *ARM® IAR Assembler Reference Guide*.

Output

The **Output** options determine the generated assembler output.



☑ Generate debug information

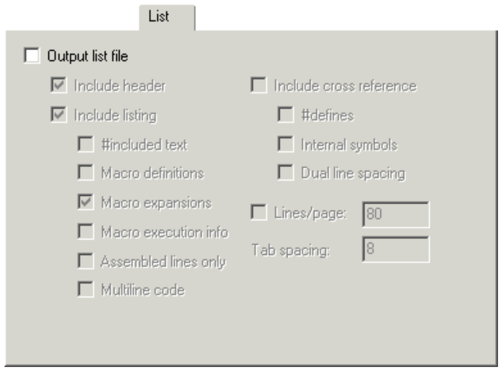
Figure 94: Assembler output options

Generate debug information

Makes the assembler generate debug information. Use this option if you want to use a debugger with your application. By default, this option is selected in a Debug project, but not in a Release project.

List

The **List** options make the assembler generate a list file and determine its contents.



☐ Output list file
☑ Include header
☑ Include listing
☐ #included text
☐ Macro definitions
☑ Macro expansions
☐ Macro execution info
☐ Assembled lines only
☐ Multiline code
☐ Include cross reference
☐ #defines
☐ Internal symbols
☐ Dual line spacing
☐ Lines/page: 80
Tab spacing: 8

Figure 95: Assembler list file options

Output list file

Makes the assembler generate a list file and send it to the file `sourcename.lst`. By default, the assembler does not generate a list file.

If you want to save the list file in another directory than the default directory for list files, use the **Output Directories** option in the **General Options** category; see *Output*, page 155, for additional information. You can open the output files directly from the **Output** folder which is available in the Workspace window.

Include header

Includes the header. The header of the assembler list file contains information about the product version, date and time of assembly, and the command line equivalents of the assembler options that were used.

Include listing

Selects which type of information to include in the list file:

#included text	Includes `#include` files in the list file.
Macro definitions	Includes macro definitions in the list file.
Macro expansion	Excludes macro expansions from the list file.
Macro execution info	Prints macro execution information on every call of a macro.
Assembled lines only	Excludes lines in false conditional assembler sections from the list file.
Multiline code	Lists the code generated by directives on several lines if necessary.

Include cross-reference

Includes a cross-reference table at the end of the list file:

#define	Includes preprocessor #defines.
Internal symbols	Includes all symbols, user-defined as well as assembler-internal.
Dual line spacing	Uses dual-line spacing.

Lines/page

Specify the number of lines per page, within the range 10 to 150. The default number of lines per page is 80 for the assembler list file.

Tab spacing

Specify the number of character positions per tab stop, within the range 2 to 9. By default, the assembler sets eight character positions per tab stop.

Preprocessor

The **Preprocessor** options allow you to define symbols and include paths for use by the assembler.

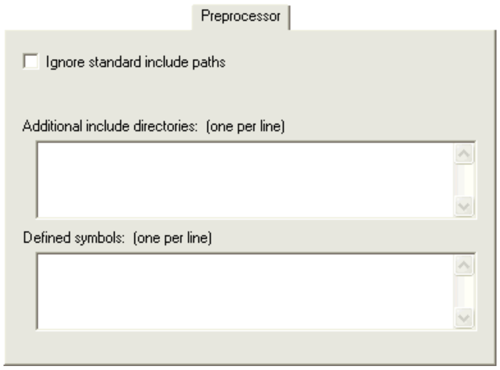

Preprocessor
Ignore standard include paths
Additional include directories: (one per line)
Defined symbols: (one per line)

Figure 96: Assembler preprocessor options

Ignore standard include directories

Ignores the standard include files, which will not be used when the project is built.

Additional include directories

Specify the full path to the list of `#include` file paths. The path required by the product is specified automatically.

To make your project more portable, use the argument variable `$TOOLKIT_DIR$` for the subdirectories of the active product and `$PROJ_DIR$` for the directory of the current project. For an overview of the argument variables, see *Argument variables*, page 117.

See the *ARM® IAR Assembler Reference Guide* for information about the `#include` directive.

Note: By default, the assembler also searches for `#include` files in the paths specified in the `IASMARM_INC` environment variable. We do not, however, recommend that you use environment variables in the IDE.

Defined symbols

Define symbols, which you otherwise must define in the source file. Specify the symbols you want to define, one per line, and specify their values.

For example, you might arrange your source to produce either the test or production version of your application depending on whether the symbol `TESTVER` was defined. To do this you would use include sections such as:

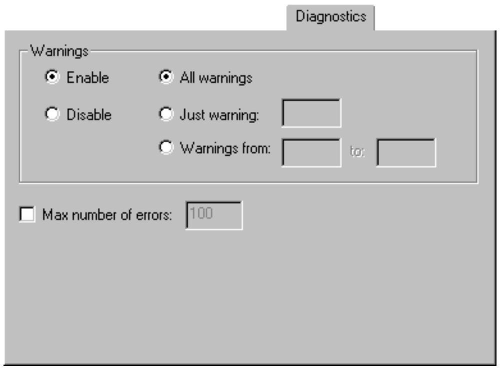
```
#ifdef  TESTVER
...     ; additional code lines for test version only
#endif
```

You would then define the symbol `TESTVER` in the Debug target but not in the Release target.

Alternatively, your source might use a variable that you need to change often, for example `FRAMERATE`. You would leave the variable undefined in the source and use this option to specify a value for the project, for example `FRAMERATE=3`.

Diagnostics

The **Diagnostics** options control individual warnings or ranges of warnings.



Warnings: [✓ Enable] Disable; [✓ All warnings] Just warning: / Warnings from: to:
☐ Max number of errors: 100

Figure 97: Assembler diagnostics options

Warnings

Controls the assembler warnings. The assembler displays a warning message when it finds an element of the source code that is legal, but probably the result of a programming error. By default, all warnings are enabled. To control the generation of warnings, choose between:

Enable	Enables warnings.
Disable	Disables warnings.
All warnings	Enables/disables all warnings.
Just warning	Enables/disables the warning you specify.
Warnings from to	Enables/disables all warnings in the range you specify.

For additional information about assembler warnings, see the *ARM® IAR Assembler Reference Guide*.

Max number of errors

Specify the maximum number of errors. This means that you can increase or decrease the number of reported errors, for example, to see more errors in a single assembly. By default, the maximum number of errors reported by the assembler is 100.

Extra Options

The **Extra Options** page provides you with a command line interface to the assembler.

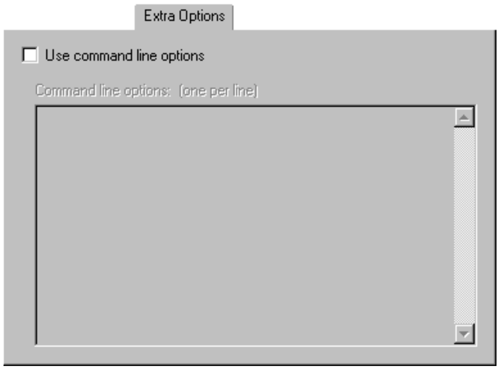


Figure 98: Extra Options for the assembler

Use command line options

Specify the additional command line arguments to be passed to the assembler (not supported by the GUI).

Output converter options

This chapter describes the options available in the IAR Embedded Workbench® IDE for converting output files from the ELF format.

For information about how to set options, see *Setting options*, page 41.

Description of output converter options

This section gives detailed information about the options in the **Output Converter** category.

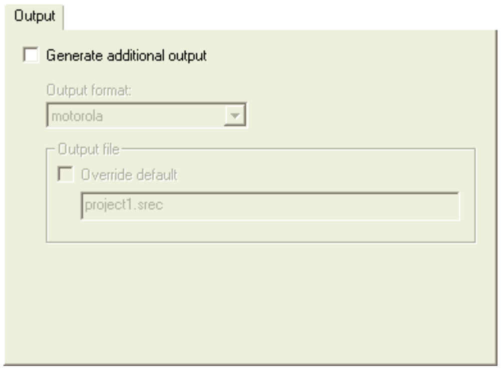
To set output converter options in the IDE:

1 Choose **Project>Options** to display the **Options** dialog box.

2 Select **Output Converter** in the **Category** list.

Output

The **Output** options determine details about the promable output format.



☐ Generate additional output

Output format: motorola

Output file

☐ Override default

project1.srec

Figure 99: Converter output options

Generate additional output

The ILINK linker generates ELF as output, optionally including DWARF for debug information. **Generate additional output** makes the converter `ielftool` convert the

ELF output to the format you specify, for example Motorola or Intel-extended. For more information about the converter, see the *IAR C/C++ Development Guide for ARM®* .

Output format

Selects the format for the output from `ielftool`. Choose between: Motorola, Intel-extended, binary, and simple-code. For more information about the converter, see the *IAR C/C++ Development Guide for ARM®* .

Output file

Specifies the name of the `iarelf` converted output file. By default, the linker will use the project name with a filename extension. The filename extension depends on which output format you choose; for example, either `srec` or `hex`. To override the default name, select **Override default** and specify the alternative filename or filename extension.

Custom build options

This chapter describes the Custom Build options available in the IAR Embedded Workbench® IDE.

For information about how to set options, see *Setting options*, page 41.

Description of custom build options

This section gives detailed information about the options in the **Custom Build** category.To set custom build options in the IDE:

1 Choose **Project>Options** to display the **Options** dialog box.

2 Select **Custom Build** in the **Category** list.

Custom Tool Configuration

The **Custom Tool Configuration** options control the invocation of the tools you want to add to the tool chain.

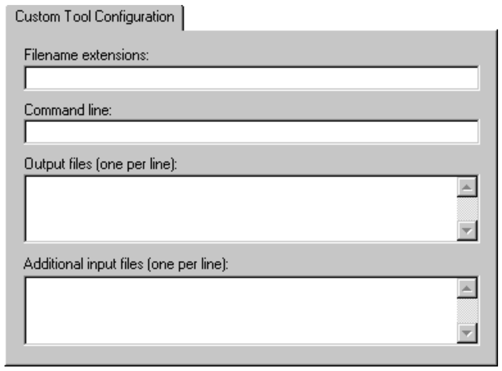

Custom Tool Configuration
Filename extensions:
Command line:
Output files (one per line):
Additional input files (one per line):

Figure 100: Custom tool configuration options

For an example, see *Extending the toolchain*, page 46.

Filename extensions

Specify the filename extensions for the types of files that are to be processed by the custom tool. You can type several filename extensions. Use commas, semicolons, or blank spaces as separators. For example:

```
.htm; .html
```

Command line

Specify the command line for executing the external tool.

Output file

Specify the name for the output files from the external tool.

Additional input files

Specify any additional files to be used by the external tool during the build process. If these additional input files, *dependency* files, are modified, the need for a rebuild is detected.

Build actions options

This chapter describes the options for pre-build and post-build actions available in the IAR Embedded Workbench® IDE.

For information about how to set options, see *Setting options*, page 41.

Description of build actions options

This section gives detailed information about the options in the **Build Actions** category.

To set build action options in the IDE:

1 Choose **Project>Options** to display the **Options** dialog box.

2 Select **Build Actions** in the **Category** list.

Build Actions Configuration

The **Build Actions Configuration** options specify pre-build and post-build actions in the IDE. These options apply to the whole build configuration, and cannot be set on groups or files.

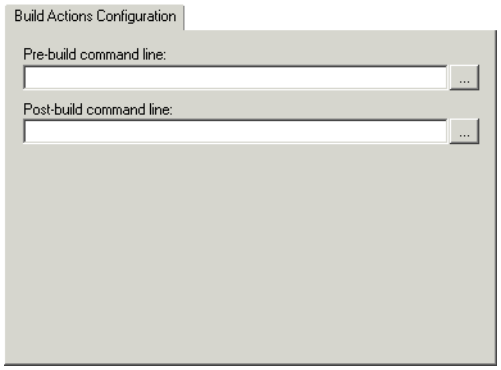


Figure 101: Build actions configuration options

Pre-build command line

Specify the command line to be executed directly before a build; a browse button for locating an extended command line file is available for your convenience. The commands will not be executed if the configuration is already up-to-date.

Post-build command line

Specify the command line to be executed directly after each successful build; a browse button is available for your convenience. The commands will not be executed if the configuration was up-to-date. This is useful for copying or post-processing the output file.

Linker options

This chapter describes the linker options available in the IAR Embedded Workbench® IDE.

For information about how to set options, see *Setting options*, page 41.

Description of linker options

This section gives detailed information about the options in the **Linker** category.

To set linker options in the IDE:

1 Choose **Project>Options** to display the **Options** dialog box.

2 Select **Linker** in the **Category** list.

Config

The **Config** options specify the path and name of the linker configuration file and define symbols for the configuration file.

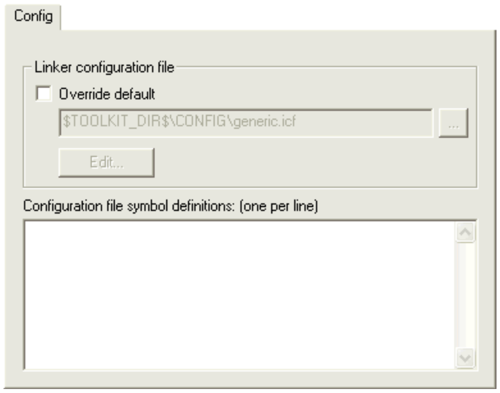

Config
Linker configuration file
Override default
$TOOLKIT_DIR$\CONFIG\generic.icf
...
Edit...
Configuration file symbol definitions: (one per line)

Figure 102: Linker configuration options

Linker configuration file

A default linker configuration file is selected automatically based on your project settings. To override the default file, select **Override default** and specify an alternative file.

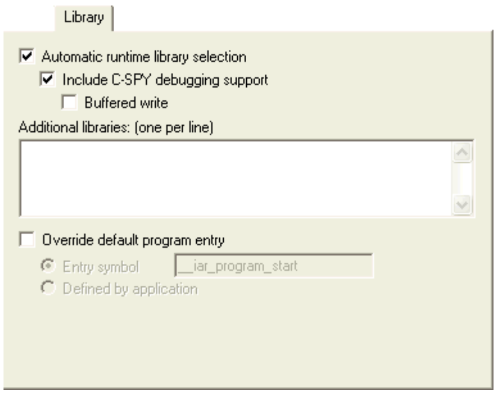
The argument variables `$TOOLKIT_DIR$` or `$PROJ_DIR$` can be used for specifying a project-specific or predefined configuration file.

Configuration file symbol definitions

Define constant configuration symbols to be used in the configuration file. Such a symbol has the same effect as a symbol defined using the `define symbol` directive in the linker configuration file.

Library

The **Library** options select the set of used libraries.



[✓] Automatic runtime library selection
[✓] Include C-SPY debugging support
[ ] Buffered write
Additional libraries: (one per line)
[ ] Override default program entry
(●) Entry symbol __iar_program_start
( ) Defined by application

Figure 103: Linker library options

See the *IAR C/C++ Development Guide for ARM®* for more information about available libraries.

Automatic runtime library selection

Makes the linker automatically choose the appropriate library based on your project settings.

Include C-SPY debugging support

Includes a debug library for communication between the application you debug and the debugger itself.

Buffered write

Buffers terminal output during program execution, instead of instantly printing each new character to the C-SPY Terminal I/O window. This option is useful when you use debugger systems that have slow communication.

Additional libraries

Specify additional libraries that you want the linker to include during the link process. You can only specify one library per line and you must specify the full path to the library. The argument variables `$PROJ_DIR$` and `$TOOLKIT_DIR$` can be used.

Alternatively, you can add an additional library directly to your project in the workspace window. You can find an example of this in the tutorial for creating and using libraries.

Override default program entry

By default, the program entry is the label `__iar_program_start`. The linker makes sure that a module containing the program entry label is included, and that the section containing that label is not discarded.

Override default program entry overrides the default entry label; choose between:

Entry symbol	Specify an entry symbol other than default.
Defined by application	Uses an entry symbol defined in the linked object code. The linker will, as always, include all program modules, and enough library modules to satisfy all symbol references, keeping all sections that are marked with the `root` attribute or that are referenced, directly or indirectly, from such a section.

Input

The **Input** options specify how to handle input to the linker.

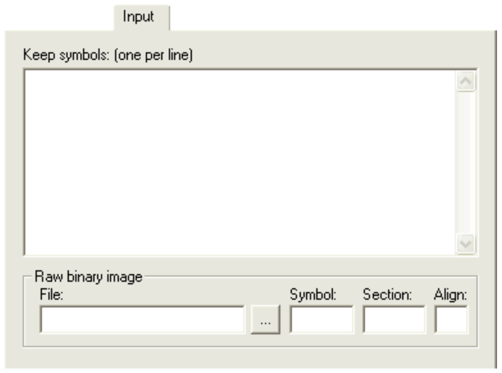


Figure 104: Linker input options

Keep symbols

Define the symbol, or several symbols one per line, that shall always be included in the final application.

By default, the linker keeps a symbol only if your application needs it.

Raw binary image

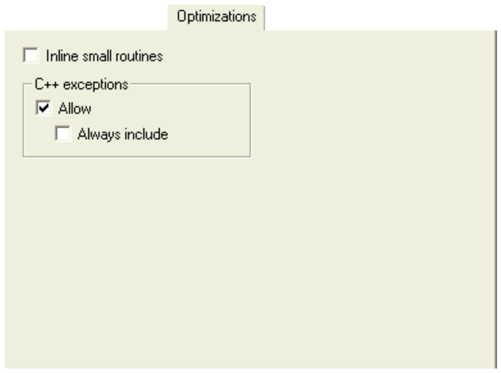
Links pure binary files in addition to the ordinary input files. Specify these parameters:

File	The pure binary file you want to link.
Symbol	The symbol defined by the section where the binary data is placed.
Section	The section where the binary data is placed.
Align	The alignment of the section where the binary data is placed.

The entire contents of the file are placed in the section you specify, which means it can only contain pure binary data, for example, the raw binary output format. The section where the contents of the specified file are placed, is only included if the specified symbol is required by your application. Use the `--keep` linker option if you want to force a reference to the symbol. Read more about single output files and the `--keep` option in the *IAR C/C++ Development Guide for ARM®* .

Optimizations

The **Optimizations** options specify optimizations performed at link time.



[ ] Inline small routines

C++ exceptions

[✓] Allow

[ ] Always include

Figure 105: Linker optimizations options

Inline small routines

Some routines are so small that they can fit in the space of the instruction that calls the routine. Use this option to make the linker replace the call of a routine with the body of the routine, where applicable.

C++ exceptions

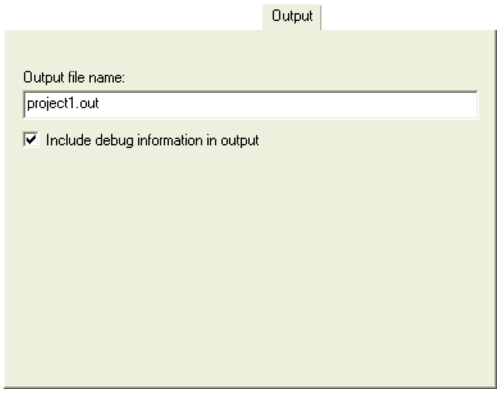
Controls the use of exceptions. Choose to:

Allow	Makes the linker not generate an error if there is a throw in the included code. By not using this option, you can make sure that your application does not use exceptions.
Always include	Makes the linker include exception tables and exception code even when the linker heuristics indicate that exceptions are not used.

The linker considers exceptions to be used if there is a `throw` expression that is not a rethrow in the included code. If there is no such `throw` expression in the rest of the code, the linker arranges for `operator new`, `dynamic_cast`, and `typeid` to call `abort` instead of throwing an exception on failure. If you need to catch exceptions from these constructs but your code contains no other throws, you might need to use this option.

Output

The **Output** options determine the generated linker output.



Output file name:
project1.out
[✓] Include debug information in output

Figure 106: Linker output options

Output filename

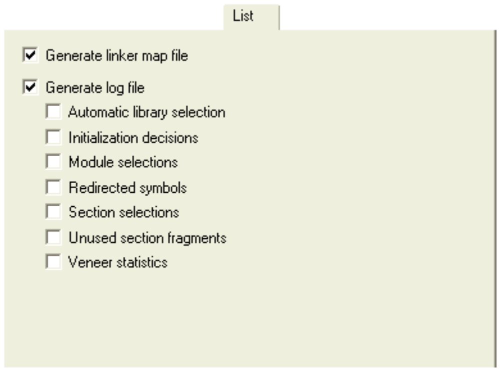
Sets the name of the ILINK output file. By default, the linker will use the project name with the filename extension `out`. To override the default name, specify an alternative name of the output file.

Include debug information in output

Makes the linker generate an ELF output file including DWARF for debug information.

List

The **List** options control the generation of linker listings.


List

- [✓] Generate linker map file
- [✓] Generate log file
  - [ ] Automatic library selection
  - [ ] Initialization decisions
  - [ ] Module selections
  - [ ] Redirected symbols
  - [ ] Section selections
  - [ ] Unused section fragments
  - [ ] Veneer statistics

Figure 107: Linker list options

Generate linker map file

Makes the linker generate a linker memory map file and send it to the `*projectname*.map` file located in the `list` directory. For detailed information about the map file and its contents, see the *IAR C/C++ Development Guide for ARM®* .

Generate log file

Makes the linker save log information to the `*projectname*.log` file located in the `list` directory. The log information can be useful for understanding why an executable image became the way it is. You can log:

- Automatic library selection
- Initialization decisions
- Module selections
- Redirected symbols
- Section selections
- Unused section fragments
- Veneer statistics.

#define

The **#define** options define absolute symbols at link time.

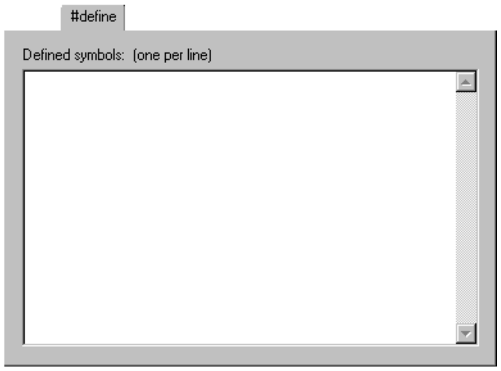


Figure 108: Linker #define options

Defined symbols

Define absolute symbols to be used at link time. This is especially useful for configuration purposes. Type the symbols that you want to define for the project, one per line, and specify their value. For example:

```
TESTVER=1
```

Note that there should be no space around the equal sign.

Any number of symbols can be defined in a linker configuration file. The symbol(s) defined in this manner will be located in a special module called `?ABS_ENTRY_MOD`, which is generated by the linker.

The linker will display an error message if you attempt to redefine an existing symbol.

Diagnostics

The **Diagnostics** options determine how diagnostics are classified and displayed. Use the diagnostics options to override the default classification of the specified diagnostics.

Note: The diagnostics cannot be suppressed for fatal errors, and fatal errors cannot be reclassified.

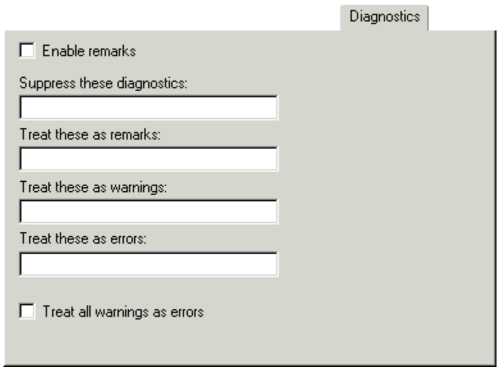


Figure 109: Linker diagnostics options

Enable remarks

Makes the linker generate remarks. By default, remarks are not issued.

The least severe diagnostic messages are called *remarks*. A remark indicates a source code construct that might cause strange behavior in the generated code.

Suppress these diagnostics

Suppresses the output of diagnostic messages for the tags that you specify.

For example, to suppress the warnings `Pe117` and `Pe177`, type:

```
Pe117,Pe177
```

Treat these as remarks

Classifies diagnostic messages as remarks.

For example, to classify the warning `Pe177` as a remark, type:

```
Pe177
```

Treat these as warnings

Classifies diagnostic messages as warnings. A *warning* indicates an error or omission that is of concern, but which will not cause the linker to stop before linking is completed.

For example, to classify the remark `Pe826` as a warning, type:

```
Pe826
```

Treat these as errors

Classifies diagnostic messages as errors. An *error* indicates a violation of the linking rules, of such severity that an executable image will not be generated, and the exit code will be non-zero.

For example, to classify the warning `Pe117` as an error, type:

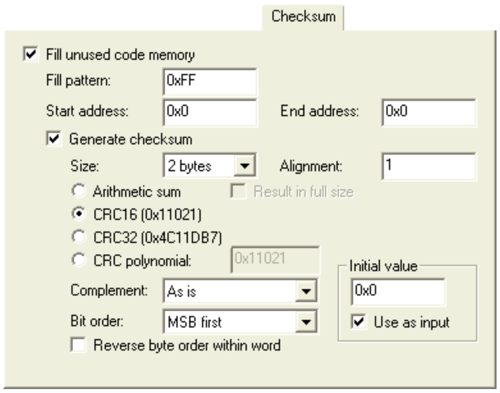
```
Pe117
```

Treat all warnings as errors

Classifies all warnings as errors. If the linker encounters an error, an executable image is not generated.

Checksum

The **Checksum** options control filling and checksumming.



[✓] Fill unused code memory
Fill pattern: 0xFF
Start address: 0x0 End address: 0x0
[✓] Generate checksum
Size: 2 bytes Alignment: 1
○ Arithmetic sum [ ] Result in full size
● CRC16 (0x11021)
○ CRC32 (0x4C11DB7)
○ CRC polynomial: 0x11021
Complement: As is
Bit order: MSB first
[ ] Reverse byte order within word
Initial value: 0x0
[✓] Use as input

Figure 110: Linker checksum and fill options

For more information about filling and checksumming, see the *IAR C/C++ Development Guide for ARM®* .

Fill unused code memory

Fills unused memory in the range you specify:

Fill pattern	Specify a size, in hexadecimal notation, of the filler to be used in gaps between segment parts.
Start address	Specify the start address for the range to be filled.

End address	Specify the end address for the range to be filled.

Generate checksum

Checksums the specified range.

Choose between:

Size	Selects the size of the checksum, which can be 1, 2, or 4 bytes.
Alignment	Specify an optional alignment for the checksum. If you do not specify an alignment explicitly, an alignment of 2 is used.
Arithmetic sum	Selects the simple arithmetic sum algorithm. The result is truncated to one byte.
Result in full size	Generates the result of the arithmetic sum algoritm in the size you specify instead of truncating it to one byte.
CRC16 (default)	Selects the CRC16 algorithm (generating polynomial 0x11021).
CRC32	Selects the CRC32 algorithm (generating polynomial 0x104C11DB7).
Crc polynomial	Selects the Crc polynomial algorithm, a generating polynomial of the value you specify.
Complement	Selects the complement variant, either the one's complement or two's complement.
Bit order	Selects the bit order of the result to be output. Choose between: **MSB first**, which ouputs the most significant bit first for each byte. **LSB first**, which reverses the bit order for each byte and outputs the least significant bit first.
Reverse byte order within word	Reverses the byte order of the input data within each word of the size specified in **Size**.
Initial value	Specify an initial value for the checksum. This is useful if the core you are using has its own checksum calculation and you want that calculation to correspond to the calculation performed by the linker.

Use as input	Prefixes the input data with a word of size **Size** that contains the value specified in **Initial value**.

Extra Options

The **Extra Options** page provides you with a command line interface to the linker.

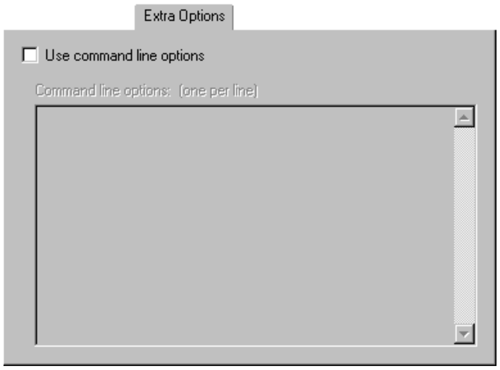


Figure 111: Linker extra options

Use command line options

Specify additional command line arguments to be passed to the linker (not supported by the GUI).

Library builder options

This chapter describes the library builder options available in the IAR Embedded Workbench® IDE.

For information about how to set options, see *Setting options*, page 41.

Description of library builder options

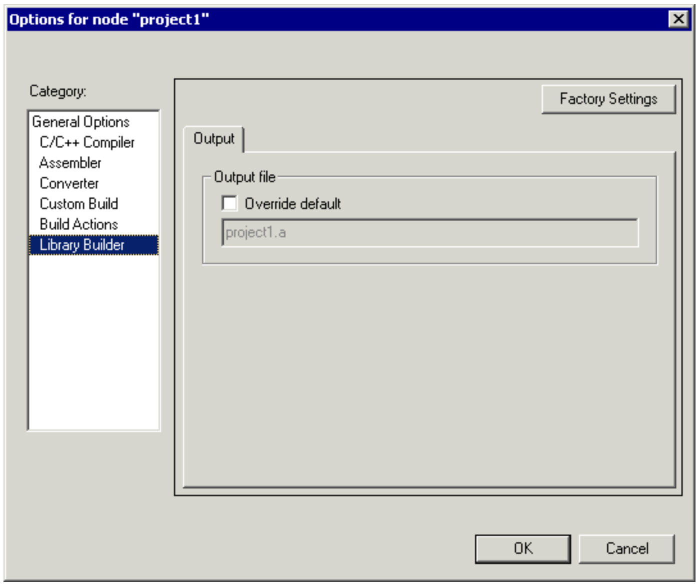
This section gives detailed information about the options in the **Library Builder** category.

Options for the library builder are not available by default. Before you can set these options in the IDE, you must add the library builder tool to the list of categories.

1. Choose **Project>Options>General Options>Output**.
2. Select the **Library** option, which means that **Library Builder** appears as a category in the **Options** dialog box.
3. Select **Library Builder** in the **Category** list.

Output

The **Output** options control the library builder and as a result of the build process, the library builder will create a library output file.



Output file

☐ Override default

project1.a

Figure 112: Library builder output options

Factory Settings

Restores all settings to the default factory settings.

Output file

Specifies the name of the output file from the library builder. By default, the linker will use the project name with a filename extension. To override the default name, select **Override default** and specify an alternative name of the output file.

Glossary

This is a general glossary for terms relevant to embedded systems programming. Some of the terms do not apply to the IAR Embedded Workbench® version that you are using.

A

Absolute location
A specific memory address for an object specified in the source code, as opposed to the object being assigned a location by the linker.

Address expression
An expression which has an address as its value.

AEABI
Embedded Application Binary Interface for ARM, defined by ARM Limited.

Application
The program developed by the user of the IAR Systems toolkit and which will be run as an embedded application on a target processor.

Ar
The GNU binary utility for creating, modifying, and extracting from archives, that is, libraries. See also *Iarchive*.

Architecture
A term used by computer designers to designate the structure of complex information-processing systems. It includes the kinds of instructions and data used, the memory organization and addressing, and the methods by which the system is implemented. The two main architecture types used in processor design are *Harvard architecture* and *von Neumann architecture*.

Archive
See *Library*.

Assembler directives
The set of commands that control how the assembler operates.

Assembler language
A machine-specific set of mnemonics used to specify operations to the target processor and input or output registers or data areas. Assembler language might sometimes be preferred over C/C++ to save memory or to enhance the execution speed of the application.

Assembler options
Parameters you can specify to change the default behavior of the assembler.

Attributes
See *Section attributes*.

Auto variables
The term refers to the fact that each time the function in which the variable is declared is called, a new instance of the variable is created automatically. This can be compared with the behavior of local variables in systems using static overlay, where a local variable only exists in one instance, even if the function is called recursively. Also called local variables. Compare *Register variables*.

B

Backtrace
Information for keeping call frame information up to date so that the IAR C-SPY® Debugger can return from a function correctly. See also *Call frame information*.

Bank
See *Memory bank*.

Bank switching
Switching between different sets of memory banks. This software technique increases a computer's usable memory by allowing different pieces of memory to occupy the same address space.

Banked code
Code that is distributed over several banks of memory. Each function must reside in only one bank.

Banked data
Data that is distributed over several banks of memory. Each data object must fit inside one memory bank.

Banked memory
Has multiple storage locations for the same address. See also *Memory bank*.

Bank-switching routines
Code that selects a memory bank.

Batch files
A text file containing operating system commands which are executed by the command line interpreter. In Unix, this is called a "shell script" because it is the Unix shell which includes the command line interpreter. Batch files can be used as a simple way to combine existing commands into new commands.

Bitfield
A group of bits considered as a unit.

Block, in linker configuration file
A continuous piece of code or data. It is either built up of blocks, overlays, and sections or it is empty. A block has a name, and the start and end address of the block can be referred to from the application. It can have attributes such as a maximum size, a specific size, or a minimum alignment. The contents can have a specific order or not.

Breakpoint
1. Code breakpoint. A point in a program that, when reached, triggers some special behavior useful to the process of debugging. Generally, breakpoints are used for stopping program execution or dumping the values of some or all of the program variables. Breakpoints can be part of the program itself, or they can be set by the programmer as part of an interactive session with a debugging tool for scrutinizing the program's execution.

2. Data breakpoint. A point in memory that, when accessed, triggers some special behavior useful to the process of debugging. Generally, data breakpoints are used to stop program execution when an address location is accessed either by a read operation or a write operation.

3. Immediate breakpoint. A point in memory that, when accessed, trigger some special behavior useful in the process of debugging. Immediate breakpoints are generally used for halting the program execution in the middle of a memory access instruction (before or after the actual memory access depending on the access type) while performing some user-specified action. The execution is then resumed. This feature is only available in the simulator version of C-SPY.

C

Call frame information
Information that allows the IAR C-SPY® Debugger to show, without any runtime penalty, the complete stack of function calls—*call stack*—wherever the program counter is, provided that the code comes from compiled C functions. See also *Backtrace*.

Calling convention
A calling convention describes the way one function in a program calls another function. This includes how register parameters are handled, how the return value is returned, and which registers that will be preserved by the called function. The compiler handles this automatically for all C and C++ functions. All code written in assembler language must conform to the rules in the calling convention to be callable from C or C++, or to be able to call C and C++ functions. The C calling convention and the C++ calling conventions are not necessarily the same.

Cheap
As in *cheap memory access*. A cheap memory access either requires few cycles to perform, or few bytes of code to implement. A cheap memory access is said to have a low cost. See *Memory access cost*.

Checksum
A computed value which depends on the ROM content of the whole or parts of the application, and which is stored along with the application to detect corruption of the data. The checksum is produced by the linker to be verified with the application. Several algorithms are supported. Compare *CRC (cyclic redundancy checking)*.

Code banking
See *Banked code*.

Code model
The code model controls how code is generated for an application. Typically, the code model controls behavior such as how functions are called and in which code section functions will be located. All object files of an application must be compiled using the same code model.

Code pointers
A code pointer is a function pointer. As many cores allow several different methods of calling a function, compilers for embedded systems usually provide the users with the ability to use all these methods.

Do not confuse code pointers with data pointers.

Code sections
Read-only sections that contain code. See also *Section*.

Compilation unit
See *Translation unit*.

Compiler options
Parameters you can specify to change the default behavior of the compiler.

Cost
See *Memory access cost*.

CRC (cyclic redundancy checking)
A number derived from, and stored with, a block of data to detect corruption. A CRC is based on polynomials and is a more advanced way of detecting errors than a simple arithmetic checksum. Compare *Checksum*.

C-SPY options
Parameters you can specify to change the default behavior of the IAR C-SPY Debugger.

Cstartup
Code that sets up the system before the application starts executing.

C-style preprocessor
A preprocessor is either a stand-alone application or an integrated part of a compiler, that performs preprocessing of the input stream before the actual compilation occurs. A C-style preprocessor follows the rules set up in Standard C and implements commands like `#define`, `#if`, and `#include`, which are used to handle textual macro substitution, conditional compilation, and inclusion of other files.

D

Data banking
See *Banked data*.

Data model
The data model specifies the default memory type. This means that the data model typically controls one or more of the following: The method used and the code generated to access static and global variables, dynamically allocated data, and the runtime stack. It also controls the default pointer type and in which data sections static and global variables will be located. A project can only use one data model at a time, and the same model must be used by all user modules and all library modules in the project.

Data pointers
Many cores have different addressing modes to access different memory types or address spaces. Compilers for embedded systems usually have a set of different data pointer types so they can access the available memory efficiently.

Data representation
How different data types are laid out in memory and what value ranges they represent.

Declaration
A specification to the compiler that an object, a variable or function, exists. The object itself must be defined in exactly one translation unit (source file). An object must either be declared or defined before it is used. Normally an object that is used in many files is defined in one source file. A declaration is normally placed in a header file that is included by the files that use the object.

For example:

```
/* Variable "a" exists somewhere. Function
 "b" takes two int parameters and returns an
 int. */

extern int a;
int b(int, int);
```

Definition
The variable or function itself. Only one definition can exist for each variable or function in an application. See also *Tentative definition*.

For example:

```
int a;
int b(int x, int y)
{
   return x + y;
}
```

Demangling
To restore a mangled name to the more common C/C++ name. See also *Mangling*.

Device description file
A file used by C-SPY that contains various device-specific information such as I/O registers (SFR) definitions, interrupt vectors, and control register definitions.

Device driver
Software that provides a high-level programming interface to a particular peripheral device.

Digital signal processor (DSP)
A device that is similar to a microprocessor, except that the internal CPU is optimized for use in applications involving discrete-time signal processing. In addition to standard microprocessor instructions, digital signal processors usually support a set of complex instructions to perform common signal-processing computations quickly.

Disassembly window
A C-SPY window that shows the memory contents disassembled as machine instructions, interspersed with the corresponding C source code (if available).

DWARF
An industry-standard debugging format which supports source level debugging. This is the format used by the IAR ILINK Linker for representing debug information in an object.

Dynamic initialization
Variables in a program written in C are initialized during the initial phase of execution, before the main function is called. These variables are always initialized with a static value, which is determined either at compile time or at link time. This is called static initialization. In C++, variables might require initialization to be performed by executing code, for example, running the constructor of global objects, or performing dynamic memory allocation.

Dynamic memory allocation
There are two main strategies for storing variables: statically at link time, or dynamically at runtime. Dynamic memory allocation is often performed from the heap and it is the size of the heap that determines how much memory that can be used for dynamic objects and variables. The advantage of dynamic memory allocation is that several variables or objects that are not active at the same time can be stored in the same memory, thus reducing the memory requirements of an application. See also *Heap memory*.

Dynamic object
An object that is allocated, created, destroyed, and released at runtime. Dynamic objects are almost always stored in memory that is dynamically allocated. Compare *Static object*.

E

EEPROM
Electrically Erasable, Programmable Read-Only Memory. A type of ROM that can be erased electronically, and then be re-programmed.

ELF
Executable and Linking Format, an industry-standard object file format. This is the format used by the IAR ILINK Linker. The debug information is formatted using DWARF.

Embedded C++
A subset of the C++ programming language, which is intended for embedded systems programming. The fact that performance and portability are particularly important in embedded systems development was considered when defining the language.

Embedded system
A combination of hardware and software, designed for a specific purpose. Embedded systems are often part of a larger system or product.

Emulator
An emulator is a hardware device that performs emulation of one or more derivatives of a processor family. An emulator can often be used instead of the actual core and connects directly to the printed circuit board—where the core would have been connected—via a connecting device. An emulator always behaves exactly as the processor it emulates, and is used when debugging requires all systems actuators, or when debugging device drivers.

Enea OSE Load module format
A specific ELF format that is loadable by the OSE operating system. See also *ELF*.

Enumeration
A type which includes in its definition an exhaustive list of possible values for variables of that type. Common examples include Boolean, which takes values from the list [true, false], and day-of-week which takes values [Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday]. Enumerated types are a feature of typed languages, including C and Ada.

Characters, (fixed-size) integers, and even floating-point types might be (but are not usually) considered to be (large) enumerated types.

EPROM
Erasable, Programmable Read-Only Memory. A type of ROM that can be erased by exposing it to ultraviolet light, and then be re-programmed.

Executable image
Contains the executable image; the result of linking several relocatable object files and libraries. The file format used for an object file is ELF with embedded DWARF for debug information.

Exceptions
An exception is an interrupt initiated by the processor hardware, or hardware that is tightly coupled with the processor, for instance, a memory management unit (MMU). The exception signals a violation of the rules of the architecture (access to protected memory), or an extreme error condition (division by zero).

Do not confuse this use of the word exception with the term *exception* used in the C++ language (but not in Embedded C++).

Expensive
As in *expensive memory access*. An expensive memory access either requires many cycles to perform, or many bytes of code to implement. An expensive memory access is said to have a high cost. See *Memory access cost*.

Extended keywords
Non-standard keywords in C and C++. These usually control the definition and declaration of objects (that is, data and functions). See also *Keywords*.

F

Filling
How to fill up bytes—with a specific fill pattern—that exists between the sections in an executable image. These bytes exist because of the alignment demands on the sections.

Format specifiers
Used to specify the format of strings sent by library functions such as `printf`. In the following example, the function call contains one format string with one format specifier, `%c`, that prints the value of `a` as a single ASCII character:

```
printf("a = %c", a);
```

G

General options
Parameters you can specify to change the default behavior of all tools that are included in the IDE.

Generic pointers
Pointers that have the ability to point to all different memory types in, for example, a core based on the Harvard architecture.

H

Harvard architecture
A core based on the Harvard architecture has separate data and instruction buses. This allows execution to occur in parallel. As an instruction is being fetched, the current instruction is executing on the data bus. Once the current instruction is complete, the next instruction is ready to go. This theoretically allows for much faster execution than a von Neumann architecture, but adds some silicon complexity. Compare *von Neumann architecture*.

Heap memory
The heap is a pool of memory in a system that is reserved for dynamic memory allocation. An application can request parts of the heap for its own use; once memory is allocated from the heap it remains valid until it is explicitly released back to the heap by the application. This type of memory is useful when the number of objects is not known until the application executes. Note that this type of memory is risky to use in systems with a limited amount of memory or systems that are expected to run for a very long time.

Heap size
Total size of memory that can be dynamically allocated.

Host
The computer that communicates with the target processor. The term is used to distinguish the computer on which the debugger is running from the core the embedded application you develop runs on.

I

Iarchive
The IAR Systems utility for creating archives, that is, libraries. Iarchive is delivered with IAR Embedded Workbench.

IDE (integrated development environment)
A programming environment with all necessary tools integrated into one single application.

Ielfdumparm
The IAR Systems utility for creating a text representation of the contents of ELF relocatable or executable image.

Ielftool
The IAR Systems utility for performing various transformations on an ELF executable image, such as fill, checksum, and format conversion.

ILINK
The IAR ILINK Linker which produces absolute output in the ELF/DWARF format.

ILINK configuration
The definition of available physical memories and the placement of sections—pieces of code and data—into those memories. ILINK requires a configuration to build an executable image.

Image
See *Executable image*.

Include file
A text file which is included into a source file. This is often done by the preprocessor.

Initialization setup in linker configuration file
Defines how to initialize RAM sections with their initializers. Normally, only non-constant non-noinit variables are initialized but, for example, pieces of code can be initialized as well.

Initialized sections
Read-write sections that should be initialized with specific values at startup. See also *Section*.

Inline assembler
Assembler language code that is inserted directly between C statements.

Inlining
An optimization that replaces function calls with the body of the called function. This optimization increases the execution speed and can even reduce the size of the generated code.

Instruction mnemonics
A word or acronym used in assembler language to represent a machine instruction. Different processors have different instruction sets and therefore use a different set of mnemonics to represent them, such as, `ADD`, `BR` (branch), `BLT` (branch if less than), `MOVE`, `LDR` (load register).

Interrupt vector
A small piece of code that will be executed, or a pointer that points to code that will be executed when an interrupt occurs.

Interrupt vector table
A table containing interrupt vectors, indexed by interrupt type. This table contains the processor's mapping between interrupts and interrupt service routines and must be initialized by the programmer.

Interrupts
In embedded systems, the use of interrupts is a method of detecting external events immediately, for example a timer overflow or the pressing of a button.

Interrupts are asynchronous events that suspend normal processing and temporarily divert the flow of control through an “interrupt handler” routine. Interrupts can be caused by both hardware (I/O, timer, machine check) and software (supervisor, system call or trap instruction). Compare *Trap*.

Intrinsic
An adjective describing native compiler objects, properties, events, and methods.

Intrinsic functions
1. Function calls that are directly expanded into specific sequences of machine code. 2. Functions called by the compiler for internal purposes (that is, floating-point arithmetic etc.).

Iobjmanip
The IAR Systems utility for performing low-level manipulation of ELF object files.

K

Key bindings
Key shortcuts for menu commands used in the IDE.

Keywords
A fixed set of symbols built into the syntax of a programming language. All keywords used in a language are reserved—they cannot be used as identifiers (in other words, user-defined objects such as variables or procedures). See also *Extended keywords*.

L

L-value
A value that can be found on the left side of an assignment and thus be changed. This includes plain variables and de-referenced pointers. Expressions like `(x + 10)` cannot be assigned a new value and are therefore not L-values.

Language extensions
Target-specific extensions to the C language.

Library
See *Runtime library*.

Library configuration file
A file that contains a configuration of the runtime library. The file contains information about what functionality is part of the runtime environment. The file is used for tailoring a build of a runtime library. See also *Runtime library*.

Linker configuration file
A file that contains a configuration used by the IAR ILINK Linker when building an executable image. See also *ILINK configuration*.

Local variable
See *Auto variables*.

Location counter
See *Program location counter (PLC)*.

Logical address
See *Virtual address (logical address)*.

M

MAC (Multiply and accumulate)
A special instruction, or on-chip device, that performs a multiplication together with an addition. This is very useful when performing signal processing where many filters and transforms have the form:

$$y_j = \sum_{i=0}^{N} c_i \cdot x_{i+j}$$

The accumulator of the MAC usually has a higher precision (more bits) than normal registers. See also *Digital signal processor (DSP)*.

Macro
1. Assembler macros are user-defined sets of assembler lines that can be expanded later in the source file by referring to the given macro name. Parameters will be substituted if referred to.

2. C macro. A text substitution mechanism used during preprocessing of source files. Macros are defined using the `#define` preprocessing directive. The replacement text of each macro is then substituted for any occurrences of the macro name in the rest of the translation unit.

3. C-SPY macros are programs that you can write to enhance the functionality of C-SPY. A typical application of C-SPY macros is to associate them with breakpoints; when such a breakpoint is hit, the macro is run and can for example be used to simulate peripheral devices, to evaluate complex conditions, or to output a trace.

The C-SPY macro language is like a simple dialect of C, but is less strict with types.

Mailbox
A mailbox in an RTOS is a point of communication between two or more tasks. One task can send messages to another task by placing the message in the mailbox of the other task. Mailboxes are also known as message queues or message ports.

Mangling
Mangling is a technique used for mapping a complex C/C++ name into a simple name. Both mangled and unmangled names can be produced for C/C++ symbols in ILINK messages.

Memory, in linker configuration file
A physical memory. The number of units it contains and how many bits a unit consists of, are defined in the linker configuration file. The memory is always addressable from `0x0` to size -1.

Memory access cost
The cost of a memory access can be in clock cycles, or in the number of bytes of code needed to perform the access. A memory which requires large instructions or many instructions is said to have a higher access cost than a memory which can be accessed with few, or small instructions.

Memory area
A region of the memory.

Memory bank
The smallest unit of continuous memory in banked memory. One memory bank at a time is visible in a core's physical address space.

Memory map
A map of the different memory areas available to the core.

Memory model
Specifies the memory hierarchy and how much memory the system can handle. Your application must use only one memory model at a time, and the same model must be used by all user modules and all library modules.

Microcontroller
A microprocessor on a single integrated circuit intended to operate as an embedded system. In addition to a CPU, a microcontroller typically includes small amounts of RAM, PROM, timers, and I/O ports.

Microprocessor
A CPU contained on one (or a few) integrated circuits. A single-chip microprocessor can include other components such as memory, memory management, caches, floating-point unit, I/O ports and timers. Such devices are also known as microcontrollers.

Module
An object. An object file contains a module and library contains one or more objects. The basic unit of linking. A module contains definitions for symbols (exports) and references to external symbols (imports). When you compile C/C++, each translation unit produces one module.

Multi-file compilation
A technique which means that the compiler compiles several source files as one compilation unit, which enables for interprocedural optimizations such as inlining, cross call, and cross jump on multiple source files in a compilation unit.

N

Nested interrupts
A system where an interrupt can be interrupted by another interrupt is said to have nested interrupts.

Non-banked memory
Has a single storage location for each memory address in a core's physical address space.

Non-initialized memory
Memory that can contain any value at reset, or in the case of a soft reset, can remember the value it had before the reset.

No-init sections
Read-write sections that should not be initialized at startup. See also *Section*.

Non-volatile storage
Memory devices such as battery-backed RAM, ROM, magnetic tape and magnetic disks that can retain data when electric power is shut off. Compare *Volatile storage*.

NOP
No operation. This is an instruction that does not do anything, but is used to create a delay. In pipelined architectures, the `NOP` instruction can be used for synchronizing the pipeline. See also *Pipeline*.

O

Objcopy
A GNU binary utility for converting an absolute object file in ELF format into an absolute object file, for example the format Motorola-std or Intel-std. See also *Ielftool*.

Object
An object file or a library member.

Object file, absolute
See *Executable image*.

Object file, relocatable
The result of compiling or assembling a source file. The file format used for an object file is ELF with embedded DWARF for debug information.

Operator
A symbol used as a function, with infix syntax if it has two arguments (`+`, for example) or prefix syntax if it has only one (for instance, bitwise negation, `~`). Many languages use operators for built-in functions such as arithmetic and logic.

Operator precedence
Each operator has a precedence number assigned to it that determines the order in which the operator and its operands are evaluated. The highest precedence operators are evaluated first. Use parentheses to group operators and operands to control the order in which the expressions are evaluated.

Options
A set of commands that control the behavior of a tool, for example the compiler or linker. The options can be specified on the command line or via the IDE.

Output image
The resulting application after linking. This term is equivalent to *executable image*, which is the term used in the IAR Systems user documentation.

Overlay, in linker configuration file
Like a block, but it contains several overlaid entities, each built up of blocks, overlays, and sections. The size of an overlay is determined by its largest constituent.

P

Parameter passing
See *Calling convention*.

Peripheral unit
A hardware component other than the processor, for example memory or an I/O device.

Pipeline
A structure that consists of a sequence of stages through which a computation flows. New operations can be initiated at the start of the pipeline even though other operations are already in progress through the pipeline.

Placement, in linker configuration file
How to place blocks, overlays, and sections into a region. It determines how pieces of code and data are actually placed in the available physical memory.

Pointer
An object that contains an address to another object of a specified type.

#pragma
During compilation of a C/C++ program, the #pragma preprocessing directive causes the compiler to behave in an implementation-defined manner. This can include, for example, producing output on the console, changing the declaration of a subsequent object, changing the optimization level, or enabling/disabling language extensions.

Pre-emptive multitasking
An RTOS task is allowed to run until a higher priority process is activated. The higher priority task might become active as the result of an interrupt. The term preemptive indicates that although a task is allotted to run a given length of time (a timeslice), it might lose the processor at any time. Each time an interrupt occurs, the task scheduler looks for the highest priority task that is active and switches to that task. If the located task is different from the task that was executing before the interrupt, the previous task is suspended at the point of interruption.

Compare *Round Robin*.

Preprocessing directives
A set of directives that are executed before the parsing of the actual code is started.

Preprocessor
See *C-style preprocessor*.

Processor variant
The different chip setups that the compiler supports.

Program counter (PC)
A special processor register that is used to address instructions. Compare *Program location counter (PLC)*.

Program location counter (PLC)
Used in the IAR Assembler to denote the code address of the current instruction. The `PLC` is represented by a special symbol (typically `$`) that can be used in arithmetic expressions. Also called simply location counter (`LC`).

Project
The user application development project.

Project options
General options that apply to an entire project, for example the target processor that the application will run on.

PROM
Programmable Read-Only Memory. A type of ROM that can be programmed only once.

Q

Qualifiers
See *Type qualifiers*.

R

Range, in linker configuration file
A range of consecutive addresses in a memory. A region is built up of ranges.

Read-only sections
Sections that contain code or constants. See also *Section*.

Real-time operating system (RTOS)
An operating system which guarantees the latency between an interrupt being triggered and the interrupt handler starting, and how tasks are scheduled. An RTOS is typically much smaller than a normal desktop operating system. Compare *Real-time system*.

Real-time system
A computer system whose processes are time-sensitive. Compare *Real-time operating system (RTOS)*.

Region, in linker configuration file
A set of non-overlapping ranges. The ranges can lie in one or more memories. For ILINK, blocks, overlays, and sections are placed into regions in the linker configuration file.

Region expression, in linker configuration file
A region built up from region literals, regions, and the common set operations possible in the linker configuration file.

Region literal, in linker configuration file
A literal that defines a set of one or more non-overlapping ranges in a memory.

Register
A small on-chip memory unit, usually just one or a few bytes in size, which is particularly efficient to access and therefore often reserved as a temporary storage area during program execution.

Register constant
A register constant is a value that is loaded into a dedicated processor register when the system is initialized. The compiler can then generate code that assumes that the constants are present in the dedicated registers.

Register locking
Register locking means that the compiler can be instructed that some processor registers shall not be used during normal code generation. This is useful in many situations. For example, some parts of a system might be written in assembler language to gain speed. These parts might be given dedicated processor registers. Or the register might be used by an operating system, or by other third-party software.

Register variables
Typically, register variables are local variables that are placed in registers instead of on the (stack) frame of the function. Register variables are much more efficient than other variables because they do not require memory accesses, so the compiler can use shorter/faster instructions when working with them. See also *Auto variables*.

Relay
A synonym to veneer, see *Veneer*.

Relocatable sections
Sections that have no fixed location in memory before linking.

Reset
A reset is a restart from the initial state of a system. A reset can originate from hardware (hard reset), or from software (soft reset). A hard reset can usually not be distinguished from the power-on condition, which a soft reset can be.

ROM-monitor
A piece of embedded software designed specifically for use as a debugging tool. It resides in the ROM of the evaluation board chip and communicates with a debugger via a serial port or network connection. The ROM-monitor provides a set of primitive commands to view and modify memory locations and registers, create and remove breakpoints, and execute your application. The debugger combines these primitives to fulfill higher-level requests like program download and single-step.

Round Robin
Task scheduling in an operating system, where all tasks have the same priority level and are executed in turn, one after the other. Compare *Pre-emptive multitasking*.

RTOS
See *Real-time operating system (RTOS)*.

Runtime library
A collection of relocatable object files that will be included in the executable image only if referred to from an object file, in other words conditionally linked.

Runtime model attributes
A mechanism that is designed to prevent modules that are not compatible to be linked into an application. A runtime attribute is a pair constituted of a named key and its corresponding value.

ILINK uses the runtime model attributes when automatically choosing a library, to verify that the correct one is used.

R-value
A value that can be found on the right side of an assignment. This is just a plain value. See also *L-value*.

S

Saturation arithmetics
Most, if not all, C and C++ implementations use mod–2^N 2-complement-based arithmetics where an overflow wraps the value in the value domain, that is, (127 + 1) = -128. Saturation arithmetics, on the other hand, does *not* allow wrapping in the value domain, for instance, (127 + 1) = 127, if 127 is the upper limit. Saturation arithmetics is often used in signal processing, where an overflow condition would have been fatal if value wrapping had been allowed.

Scheduler
The part of an RTOS that performs task-switching. It is also responsible for selecting which task that should be allowed to run. Many scheduling algorithms exist, but most of them are either based on static scheduling (performed at compile-time), or on dynamic scheduling (where the actual choice of which task to run next is taken at runtime, depending on the state of the system at the time of the task-switch). Most real-time systems use static scheduling, because it makes it possible to prove that the system will not violate the real-time requirements.

Scope
The section of an application where a function or a variable can be referenced by name. The scope of an item can be limited to file, function, or block.

Section
An entity that either contains data or text. Typically, one or more variables, or functions. A section is the smallest linkable unit.

Section attributes
Each section has a name and an attribute. The attribute defines what a section contains, that is, if the section content is read-only, read/write, code, data, etc.

Section fragment
A part of a section, typically a variable or a function.

Section selection
In the linker configuration file, defining a set of sections by using section selectors. A section belongs to the most restrictive section selector if it can be part of more than one selection. Three different selectors can be used individually or in conjunction to select the set of sections: *section attribute* (selecting by the section content), *section name* (selecting by the section name), and *object name* (selecting from a specific object).

Semaphore
A semaphore is a type of flag that is used for guaranteeing exclusive access to resources. The resource can be a hardware port, a configuration memory, or a set of variables. If several tasks must access the same resource, the parts of the code (the critical sections) that access the resource must be made exclusive for every task. This is done by obtaining the semaphore that protects that resource, thus blocking all other tasks from it. If another task wishes to use the resource, it also must obtain the semaphore. If the semaphore is already in use, the second task must wait until the semaphore is released. After the semaphore is released, the second task is allowed to execute and can obtain the semaphore for its own exclusive access.

Severity level
The level of seriousness of the diagnostic response from the assembler, compiler, or debugger, when it notices that something is wrong. Typical severity levels are remarks, warnings, errors, and fatal errors. A remark just points to a possible problem, while a fatal error means that the programming tool exits without finishing.

Sharing
A physical memory that can be addressed in several ways. For ILINK, defined in the linker configuration file.

Short addressing
Many cores have special addressing modes for efficient access to internal RAM and memory mapped I/O. Short addressing is therefore provided as an extended feature by many compilers for embedded systems. See also *Data pointers*.

Side effect
An expression in C or C++ is said to have a side-effect if it changes the state of the system. Examples are assignments to a variable, or using a variable with the post-increment operator. The C and C++ standards state that a variable that is subject to a side-effect should not be used more that once in an expression. As an example, this statement violates that rule:

```
*d++ = *d;
```

Signal
Signals provide event-based communication between tasks. A task can wait for one or more signals from other tasks. Once a task receives a signal it waits for, execution continues. A task in an RTOS that waits for a signal does not use any processing time, which allows other tasks to execute.

Simulator
A debugging tool that runs on the host and behaves as similar to the target processor as possible. A simulator is used to debug the application when the hardware is unavailable, or not needed for proper debugging. A simulator is usually not connected to any physical peripheral devices. A simulated processor is often slower, or even much slower, than the real hardware.

Single stepping
Executing one instruction or one C statement at a time in the debugger.

Skeleton code
An incomplete code framework that allows the user to specialize the code.

Special function register (SFR)
A register that is used to read and write to the hardware components of the core.

Stack frames
Data structures containing data objects like preserved registers, local variables, and other data objects that must be stored temporary for a particular scope (usually a function).

Earlier compilers usually had a fixed size and layout on a stack frame throughout a complete function, while modern compilers might have a very dynamic layout and size that can change anywhere and anytime in a function.

Stack sections
The section or sections that reserve space for the stack(s). Most processors use the same stack for calls and parameters, but some have separate stacks.

Standard libraries
The C and C++ library functions as specified by the C and C++ standard, and support routines for the compiler, like floating-point routines.

Static object
An object whose memory is allocated at link-time and is created during system startup (or at first use). Compare *Dynamic object*.

Static overlay
Instead of using a dynamic allocation scheme for parameters and auto variables, the linker allocates space for parameters and auto variables at link time. This generates a worst-case scenario of stack usage, but might be preferable for small chips with expensive stack access or no stack access at all.

Statically allocated memory
This kind of memory is allocated once and for all at link-time, and remains valid all through the execution of the application. Variables that are either global or declared `static` are allocated this way.

Structure value
A collecting names for structs and unions. A struct is a collection of data object placed sequentially in memory (possibly with pad bytes between them). A union is a collection of data sharing the same memory location.

Symbolic location
A location that uses a symbolic name because the exact address is unknown.

T

Target
1. An architecture. 2. A piece of hardware. The particular embedded system you are developing the application for. The term is usually used to distinguish the system from the host system.

Task (thread)
A task is an execution thread in a system. Systems that contain many tasks that execute in parallel are called multitasking systems. Because a processor only executes one instruction stream at the time, most systems implement some sort of task-switch mechanism (often called context switch) so that all tasks get their share of processing time. The process of determining which task that should be allowed to run next is called scheduling. Two common scheduling methods are *Pre-emptive multitasking* and *Round Robin*.

Tentative definition
A variable that can be defined in multiple files, provided that the definition is identical and that it is an absolute variable.

Terminal I/O
A simulated terminal window in C-SPY.

Timer
A peripheral that counts independent of the program execution.

Timeslice
The (longest) time an RTOS allows a task to run without running the task-scheduling algorithm. A task might be allowed to execute during several consecutive timeslices before being switched out. A task might also not be allowed to use its entire time slice, for example if, in a preemptive system, a higher priority task is activated by an interrupt.

Translation unit
A source file together with all the header files and source files included via the preprocessor directive `#include`, except for the lines skipped by conditional preprocessor directives such as `#if` and `#ifdef`.

Trap
A trap is an interrupt initiated by inserting a special instruction into the instruction stream. Many systems use traps to call operating system functions. Another name for trap is software interrupt.

Type qualifiers
In Standard C/C++, `const` or `volatile`. IAR Systems compilers usually add target-specific type qualifiers for memory and other type attributes.

U

UBROF (Universal Binary Relocatable Object Format)
File format produced by some of the IAR Systems programming tools, if your product package includes the XLINK linker.

V

Value expressions, in linker configuration file
A constant number that can be built up out of expressions that has a syntax similar to C expressions.

Veneer
A small piece of code that is inserted as a springboard between caller and callee when:

• There is a mismatch in mode, for example ARM and Thumb

• The call instruction does not reach its destination.

Virtual address (logical address)
An address that must be translated by the compiler, linker or the runtime system into a physical memory address before it is used. The virtual address is the address seen by the application, which can be different from the address seen by other parts of the system.

Virtual space
An IAR Embedded Workbench Editor feature which allows you to place the insertion point outside of the area where there are actual characters.

Volatile storage
Data stored in a volatile storage device is not retained when the power to the device is turned off. To preserve data during a power-down cycle, you should store it in non-volatile storage. This should not be confused with the C keyword `volatile`. Compare *Non-volatile storage*.

von Neumann architecture
A computer architecture where both instructions and data are transferred over a common data channel. Compare *Harvard architecture*.

W

Watchpoints
Watchpoints keep track of the values of C variables or expressions in the C-SPY Watch window as the application is being executed.

X

XLINK
The IAR XLINK Linker which uses the UBROF output format.

Z

Zero-initialized sections
Sections that should be initialized to zero at startup. See also *Section*.

Zero-overhead loop

A loop in which the loop condition, including branching back to the beginning of the loop, does not take any time at all. This is usually implemented as a special hardware feature of the processor and is not available in all architectures.

Zone

Different processors have widely differing memory architectures. *Zone* is the term C-SPY uses for a named memory area. For example, on processors with separately addressable code and data memory there would be at least two zones. A processor with an intricate banked memory scheme might have several zones.

Symbols