AVR® IAR EMBEDDED WORKBENCH™

User Guide

for Atmel® Corporation's

AVR® Microcontroller

COPYRIGHT NOTICE

DISCLAIMER

The information in this document is subject to change without notice and does not represent a commitment on any part of IAR Systems. While the information contained herein is assumed to be accurate, IAR Systems assumes no responsibility for any errors or omissions.

In no event shall IAR Systems, its employees, its contractors, or the authors of this document be liable for special, direct, indirect, or consequential damage, losses, costs, charges, claims, demands, claim for lost profits, fees, or expenses of any nature or kind.

TRADEMARKS

IAR and C-SPY are registered trademarks of IAR Systems. IAR Embedded Workbench, IAR XLINK Linker, and IAR XLIB Librarian are trademarks of IAR Systems. AVR and Atmel are registered trademarks of Atmel Corporation. Microsoft is a registered trademark, and Windows is a trademark of Microsoft Corporation. Pentium® is a registered trademark of Intel Corporation. Codewright is a registered trademark of Premia Corporation. Adobe and Acrobat Reader are registered trademarks of Adobe Systems Incorporated.

All other product names are trademarks or registered trademarks of their respective owners.

First edition: March 2000

Part number: UAVR-1

WELCOME

Welcome to the AVR IAR Embedded Workbench™ User Guide. This guide describes how to use the IAR Embedded Workbench™ with its integrated Windows development tools for the AVR microcontroller.

Before starting to use the tools, we recommend you to read the initial chapters of this guide. Here you will find information about installing the tools, product overviews, and tutorials that will help you get started.

This guide also includes complete reference information about the IAR Embedded Workbench with the simulator version of the IAR C-SPY® Debugger for the AVR microcontroller.

Refer to the *AVR IAR Compiler Reference Guide* and *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide* for more information about the development tools incorporated in the IAR Embedded Workbench.

Refer to the chip manufacturer's documentation for information about the AVR architecture and instruction set.

If you want to know more about IAR Systems, visit the website **www.iar.com** where your will find company information, product news, technical support, and much more.

ABOUT THIS GUIDE

This guide consists of the following parts:

- ***Part 1: The IAR development tools***

  The IAR Embedded Workbench provides a brief summary of the features of the IAR Systems development tools for the AVR: the IAR Embedded Workbench™, IAR Compiler, IAR Assembler, IAR XLINK Linker™, IAR XLIB Librarian™, and IAR C-SPY® Debugger.

  Installation and documentation describes the system requirements and explains how to run the IAR Embedded Workbench with the IAR C-SPY Debugger. It also describes the directory structure and file types, and gives an overview of the documentation supplied with the IAR development tools.

  The project model explains how the IAR Embedded Workbench organizes your work into a project, to help you keep track of the source files involved in a typical application. It explains how the configuration options relate to your project, and how you use the IAR development tools to operate on the files within a project.

- ***Part 2: Tutorials***

  IAR Embedded Workbench tutorial describes a typical development cycle using the IAR Embedded Workbench, the AVR IAR Compiler, and the IAR XLINK Linker™. It also introduces you to the IAR C-SPY Debugger.

  Compiler tutorials illustrates how you might use the IAR Embedded Workbench and the IAR C-SPY Debugger to develop a series of typical programs for the AVR IAR Compiler, using some of the compiler's most important features.

  Assembler tutorials illustrates how you might use the IAR Embedded Workbench and the IAR C-SPY Debugger to develop machine-code programs, using some of the most important features of the AVR IAR Assembler. It also introduces you to the IAR XLIB Librarian™.

  Advanced tutorials illustrates how you might use both code written for the AVR IAR Compiler and code written for the AVR IAR Assembler in the same project. It also explores the functionality of the IAR C-SPY Debugger.

- ***Part 3: The IAR Embedded Workbench***

  General options describes how to set general project options in the IAR Embedded Workbench.

  Compiler options explains how to set compiler options from the IAR Embedded Workbench, and describes each option.

  Assembler options explains how to set assembler options in the IAR Embedded Workbench, and describes each option.

  XLINK options explains how to set linker options in the IAR Embedded Workbench, and describes each option.

  C-SPY options explains how to set C-SPY options in the IAR Embedded Workbench, and describes each option.

  IAR Embedded Workbench reference provides reference information about the IAR Embedded Workbench, and the commands on each of the IAR Embedded Workbench menus.

- ***Part 4: The C-SPY simulator***

  Introduction to C-SPY describes the general functionality of the IAR C-SPY Debugger.

C-SPY expressions defines the syntax of the expressions and variables used in C-SPY macros, and gives examples to show how to use macros in debugging.

C-SPY macros lists the built-in system macros supplied with the IAR C-SPY Debugger.

C-SPY reference provides complete reference information about the C-SPY windows, menu commands, and their associated dialog boxes.

C-SPY command line options gives information about customizing the IAR C-SPY Debugger using command line options or setup macros.

ASSUMPTIONS AND CONVENTIONS

ASSUMPTIONS

This guide assumes that you have a working knowledge of the following:

- The C or Embedded C++ programming language and the IAR AVR assembly language.
- The architecture and instruction set of the AVR microcontroller.
- The procedures for using menus, windows, and dialog boxes in a Windows environment.

Note: The illustrations in this guide show the IAR Embedded Workbench running in a Windows 95-style environment, and their appearance will be slightly different if you are using another platform.

CONVENTIONS

This user guide uses the following typographical conventions:

Style	*Used for*
`computer`	Text that you type in, or that appears on the screen.
`parameter`	A label representing the actual value you should type as part of a command.
`[option]`	An optional part of a command.
`{a | b | c}`	Alternatives in a command.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.

Style	*Used for*
reference	A cross-reference to another part of this guide, or to another guide.
	Identifies instructions specific to the IAR Embedded Workbench versions of the IAR development tools.
	Identifies instructions specific to the command line versions of the IAR development tools.

FURTHER READING

The following books may be of interest to you when using the IAR Systems development tools for the AVR microcontroller:

- Michael Barr, Andy Oram (editor): *Programming Embedded Systems in C and C++* (O'Reilly & Associates)
- Brian W. Kernighan, Dennis M. Ritchie: *The C Programming Language* (Prentice Hall)

  The later editions describe the ANSI C standard.
- Claus Kühnel: *AVR RISC Microcontroller Handbook* (Newnes)
- Jean J. Labrosse: *Embedded Systems Building Blocks: Complete and Ready-To-Use Modules in C* (R&D Books)
- Bernhard Mann: *C für Mikrocontroller* (Franzis-Verlag)
- Bjarne Stroustrup: *The C++ Programming Language* (Addison-Wesley)

We recommend that you visit the websites of Atmel Corporation and IAR Systems:

- At the Atmel website, **www.atmel.com**, you can find information and news about the AVR microcontrollers.
- At the IAR website, **www.iar.com**, you will find AVR application notes and other product information.

CONTENTS

PART 1: THE IAR DEVELOPMENT TOOLS

This part of the AVR IAR Embedded Workbench™ User Guide includes the following chapters:

- *The IAR Embedded Workbench*
- *Installation and documentation*
- *The project model*.

THE IAR EMBEDDED WORKBENCH

The IAR Embedded Workbench™ is a very powerful Integrated Development Environment (IDE), allowing you to develop and manage your complete embedded application project. It is a true 32-bit Windows environment, with all the features you would expect to find in your everyday working place.

THE FRAMEWORK

The IAR Embedded Workbench is the framework, where all necessary tools are seamlessly integrated. Support for a large number of target processors can be added into the IAR Embedded Workbench, allowing you to stay within a well-known development environment also for your next project.

The IAR Embedded Workbench also promotes a useful working methodology, and thus a significant reduction of the development time can be achieved by using the IAR tools. We call this concept: “Different Architectures. One Solution”. The IAR Embedded Workbench is available for a large number of microprocessors and microcontrollers in the 8-, 16-, and 32-bit segments. It provides an easy-to-learn and highly efficient development environment with maximum code inheritance capabilities, comprehensive and specific target support.

INTEGRATED TOOLS

The IAR Embedded Workbench integrates a highly optimized C/EC++ compiler, an assembler, the versatile IAR XLINK Linker, the IAR XLIB Librarian, a powerful editor, a project manager with Make utility, and C-SPY®, a state-of-the-art high-level-language debugger.

Although the IAR Embedded Workbench provides all the features required for a successful project, we also recognize the need to integrate other tools. Therefore the IAR Embedded Workbench can be easily adapted to work with your editor of choice, your preferred revision control system, etc. Project files can be saved as text files, to support your own Make facility. The IAR XLINK Linker can produce a large number of output formats, allowing for debugging on most third-party emulators.

The command line version of the compiler is also included in the product package, if you want to use the compiler and linker as external tools in an already established project environment.

If you want more information about supported target processors, contact your software distributor or your IAR representative, or visit the IAR website **www.iar.com** for information about recent product releases.

IAR EMBEDDED WORKBENCH

The IAR Embedded Workbench™ is a flexible integrated development environment, allowing you to develop applications for a variety of different target processors. It provides a convenient Windows interface for rapid development and debugging.

FEATURES

Below follows a brief overview of the features of the IAR Embedded Workbench.

General features

The IAR Embedded Workbench provides the following general features:

- Runs under Windows 95/98, or Windows NT 4 or later.
- Intuitive user interface, taking advantage of Windows 95/98 features.
- Hierarchical project representation.
- Full integration between the IAR Embedded Workbench tools and editor.
- Binary File Editor with multi-level undo and redo.

The IAR Embedded Workbench editor

The IAR Embedded Workbench Editor provides the following features:

- Syntax of C or Embedded C++ programs shown using text styles and colors.
- Powerful search and replace commands, including multi-file search.
- Direct jump to context from error listing.
- Parenthesis matching.
- Automatic indentation.

- Multi-level undo and redo for each window.

Compiler and assembler projects
The IAR Embedded Workbench provides the following features for the IAR Compiler and the IAR Assembler:

- Projects build in the background, allowing simultaneous editing.
- Options can be set globally, on groups of source files, or on individual source files.
- The Make utility recompiles, reassembles, and links files only when necessary.
- Generic and AVR-specific optimization techniques produce very efficient machine code.

Documentation
The AVR IAR Embedded Workbench is documented in the *AVR IAR Embedded Workbench™ User Guide* (this guide). There is also context-sensitive help and hypertext versions of the user documentation available online.

IAR COMPILER

The IAR Compiler for the AVR microcontroller offers the standard features of the C or Embedded C++ language, plus many extensions designed to take advantage of the AVR-specific facilities.

The AVR IAR Compiler is integrated with other IAR Systems software for the AVR microcontroller. It is supplied with the IAR AVR Assembler, with which it shares linker and librarian manager tools.

FEATURES

The following section describes the features of the AVR IAR Compiler.

Language facilities
- Conformance to the ISO/ANSI standard for a free-standing environment.
- Standard library of functions applicable to embedded systems, with source code optionally available.
- IEEE-compatible floating-point arithmetic.
- Embedded C++.

- Object code can be linked with assembly routines.
- Interrupt functions can be written in C or Embedded C++.

Type checking

- External references are type-checked at link time.
- Extensive type checking at compile time.
- Link-time inter-module consistency checking of the run-time module.

Code generation

- Selectable optimization for code size or execution speed.
- Comprehensive output options, including relocatable object code, assembler source code, and C or Embedded C++ list files with optional assembler mnemonics.
- Easy-to-understand error and warning messages.
- Compatibility with the C-SPY® high-level debugger.

Target support

- Flexible variable allocation.
- `#pragma` directives to maintain portability while using processor-specific extensions.
- Supports both the standard instruction set and the enhanced instruction set.
- Inline assembler statements.
- Intrinsic functions.

Documentation

The AVR IAR Compiler is documented in the *AVR IAR Compiler Reference Guide.*

IAR ASSEMBLER

The AVR IAR Assembler is a powerful relocating macro assembler with a versatile set of directives.

The AVR IAR Assembler uses the same mnemonics as the Atmel AVR Assembler, which makes the migration of existing code quite easy. For detailed information, see the *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide*.

FEATURES

The AVR IAR Assembler provides the following features:

- Integration with other IAR Systems software for the AVR microcontroller.
- Built-in C language preprocessor.
- Extensive set of assembler directives and expression operators.
- Conditional assembly.
- Powerful recursive macro facilities supporting the Intel/Motorola style.
- List file with augmented cross-reference output.
- Number of symbols and program size limited only by available memory.
- Support for complex expressions with external references.
- Up to 65536 relocatable segments per module.
- 255 significant characters in symbol names.
- 32-bit arithmetic.

Documentation

The AVR IAR Assembler is documented in the *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide*.

IAR XLINK LINKER

The IAR XLINK Linker converts one or more relocatable object files produced by the IAR Systems assembler or compiler to machine code for a specified target processor. It supports a wide range of industry-standard loader formats, in addition to the IAR Systems debug format used by the IAR C-SPY Debugger.

The IAR XLINK Linker supports user libraries, and will load only those modules that are actually needed by the program you are linking.

The final output produced by the IAR XLINK Linker is an absolute, target-executable object file that can be downloaded to the AVR microcontroller or to a hardware emulator.

FEATURES

The IAR XLINK Linker offers the following important features:

- Full C-level type checking across all modules.
- Full dependency resolution of all symbols in all input files, independent of input order.
- Simple override of library modules.
- Supports 255 character symbol names.
- Checks for compatible compiler settings for all modules.
- Checks that the correct version and variant of the C or Embedded C++ run-time library is used.
- Flexible segment commands allow detailed control of code and data placement.
- Link-time symbol definition enables flexible configuration control.
- Support for over 30 output formats.
- Can generate checksum of code for run-time checking.

Documentation

The IAR XLINK Linker is documented in the *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide*.

IAR XLIB LIBRARIAN

The IAR XLIB Librarian enables you to manipulate the relocatable object files produced by the IAR Systems assembler and compiler.

FEATURES

The IAR XLIB Librarian provides the following features:

- Support for modular programming.
- Modules can be listed, added, inserted, replaced, deleted, or renamed.
- Modules can be changed between program and library type.
- Segments can be listed and renamed.
- Symbols can be listed and renamed.
- Interactive or batch mode operation.
- A full set of library listing operations.

Documentation
The IAR XLIB Librarian is documented in the *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide.*

IAR C-SPY DEBUGGER

The IAR C-SPY Debugger is a high-level-language debugger for embedded applications. It is designed for use with the IAR compilers, assemblers, IAR XLINK Linker, and IAR XLIB Librarian. The IAR C-SPY Debugger allows you to switch between source mode and disassembly mode debugging as required, for both C or Embedded C++ and assembler code.

Source mode debugging provides the quickest and easiest way of verifying the less critical parts of your application, without needing to worry about how the compiler has implemented your C or Embedded C++ code in assembler. During C or Embedded C++ level debugging you can execute the program a C or Embedded C++ statement at a time, and monitor the values of C or Embedded C++ variables and data structures.

You can choose between disassembled code and original assembler source code. Disassembly mode debugging lets you focus on the critical sections of your application, and provides you with precise control over the hardware. You can execute the program an assembler instruction at a time, and display the registers and memory or change their contents.

FEATURES

The IAR C-SPY Debugger offers a unique combination of features. These are described in the following sections.

General

The IAR C-SPY Debugger offers the following general features:

- Intuitive user interface, taking advantage of Windows 95/98 features.
- Source and disassembly mode debugging.
- Fast simulator.
- Log file option.
- Powerful macro language.
- Complex code and data breakpoints.
- Memory validation.
- Interrupt simulation.
- UBROF, INTEL-EXTENDED, and Motorola input formats supported.

High-level-language debugging

- Expression analyzer.
- Extensive type recognition of variables.
- Configurable register window and multiple memory windows.
- Function trace.
- C or Embedded C++ call stack with parameters.
- Watchpoints on expressions.
- Code coverage.
- Function-level profiling.
- Watch, Locals, and QuickWatch windows allow you to expand `arrays` and `structs`.
- Optional terminal I/O emulation.

Assembler-level debugging

- Full support for auto and register variables.

- Built-in assembler/disassembler.

Documentation
The IAR C-SPY Debugger is documented in the *AVR IAR Embedded Workbench™ User Guide* (this guide). There is also context-sensitive help available online.

VERSIONS

The IAR C-SPY Debugger for the AVR microcontroller is currently available in a simulator version and a ROM-monitor version for the AT90SCC Crypto Controller, configured for the Smart Card Development Kit from Atmel-ES2.

Contact your software distributor or IAR representative for information about other versions of C-SPY.

Below are general descriptions of the different C-SPY versions.

Simulator version
The simulator version simulates the functions of the target processor entirely in software. With this C-SPY version, the program logic can be debugged long before any hardware is available. Since no hardware is required, it is also the most cost-effective solution for many applications.

For additional information about the simulator version of the IAR C-SPY Debugger, refer to *Part 4: The C-SPY simulator* in this guide.

Emulator version
The emulator version of the IAR C-SPY Debugger provides control over an in-circuit emulator, which is connected to the host computer.

The IAR C-SPY Debugger uses the hardware features of the emulator, such as breakpoint logic and memory inspection, to allow an application to be executed in real time and in the proper target environment.

ROM-monitor version
A ROM-monitor is a software component that runs on e.g. an evaluation board which is connected to the host. C-SPY uses this software to access memory and register information.

INSTALLATION AND DOCUMENTATION

This chapter contains information about system requirements, explains how to run the IAR Embedded Workbench™, describes the directory structure and file types, and gives an overview of the available documentation.

Refer to the *QuickStart Card*, which is delivered with the product, for information about how to install and register the IAR products.

INCLUDED IN THIS PACKAGE

The IAR Systems development tools package for the AVR microcontroller contains the following items:

- Installation media.
- *QuickStart Card*.
- User documentation:

  AVR IAR Embedded Workbench™ User Guide (this guide).

  AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide.

  AVR IAR Compiler Reference Guide.

SYSTEM REQUIREMENTS

The IAR Systems development tools for the AVR microcontroller run under Windows 95/98, or Windows NT 4 or later.

We recommend a Pentium® processor with at least 64 Mbytes of RAM allowing you to fully utilize and take advantage of the product features, and 100 Mbytes of free disk space for the IAR development tools.

To access all the product documentation, you also need a web browser and the Adobe Acrobat® Reader.

RUNNING THE PROGRAM

RUNNING THE IAR EMBEDDED WORKBENCH

Select the **Start** button in the taskbar and select **Programs**. Select **IAR Systems** in the menu. Then select **IAR Embedded Workbench for AVR** and **IAR Embedded Workbench** to run the `ew23.exe` program which is located in the installation root directory.

Note: The product version number may become updated in a future release. Refer to the `ewavr` read-me file for the most recent product information.

RUNNING THE IAR C-SPY DEBUGGER

The most common way to start the IAR C-SPY Debugger is from within the IAR Embedded Workbench, where you select **Debugger** from the **Project** menu or click the **Debugger** icon in the toolbar.

You can also start C-SPY from the **Programs** menu. Select **IAR Systems** in the menu. Then select **IAR Embedded Workbench for AVR** and **IAR C-SPY** to run the `cw23.exe` program which is located in the installation root directory.

It is also possible to start C-SPY by using the Windows **Run...** command, specifying options. See *Setting C-SPY options from the command line*, page 243, for additional information about command line options.

UPGRADING TO A NEW VERSION

When upgrading to a new version of the product, you should first uninstall the previous version.

First make sure to create back-up copies of all files you may have modified, such as linker command files (`*.xcl`). Then use the standard procedure in Windows to uninstall the previous product version (select **Add/Remove Programs** in the **Control Panel** in Windows). Finally install the new version of the product, using the same path as before.

UNINSTALLING THE PRODUCTS

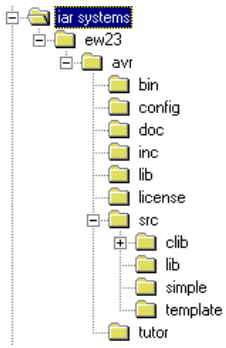
To uninstall the IAR toolkit, use the standard procedure by selecting **Add/Remove Programs** in the **Control Panel** in Windows.

DIRECTORY STRUCTURE

The installation procedure creates several directories to contain the different 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\ew23\ directory where *x* is the drive where Microsoft Windows is installed. The executable files for the IAR Embedded Workbench and the IAR C-SPY Debugger are located here. The root directory also contains the avr directory, where all product-specific subdirectories are located.



If you already have an ew23.exe file installed, the installation program will suggest to use its root directory also for the installation of the AVR IAR development tools.

THE BIN DIRECTORY

The bin subdirectory contains executable files such as exe and dll files, the C-SPY driver, and the AVR help files.

THE CONFIG DIRECTORY

The config subdirectory contains files to be used for configuring the system. A linker command file (*.xcl) for each supported derivative is located here. The C-SPY device description files (*.ddf) are also located in this directory.

THE DOC DIRECTORY

The doc subdirectory contains read-me files (*.htm or *.txt) with recent additional information about the AVR tools. It is recommended that you read all of these files before proceeding. The directory also contains online versions (PDF format) of this user guide, and of the AVR reference guides.

THE INC DIRECTORY

The inc subdirectory holds include files, such as the header files for the standard C or Embedded C++ library, as well as a specific header file defining special function registers (SFRs). These files are used by both the compiler and the assembler, as defined in the iomacro.h file.

THE LIB DIRECTORY

The lib subdirectory holds library modules used by the compiler.

The IAR XLINK Linker™ searches for library files in the directory specified by the XLINK_DFLTDIR environment variable. If you set this environment variable to the path of the lib subdirectory, you can refer to lib library modules simply by their basenames.

THE LICENSE DIRECTORY

The license subdirectory holds the IAR Systems License Manager utility.

THE SRC DIRECTORY

The src\clib subdirectory contains the IAR C library in source format. This library is provided for backward compatibility. The subdirectory src\clib\lib will be created when you run the cl.bat file.

The src\lib subdirectory contains source files that are shared between the standard C or Embedded C++ library and the IAR C library.

The src\simple subdirectory contains the reader software for the XLINK SIMPLE output format.

The src\template subdirectory contains linker command file templates (*.xcl). The *Configuration* chapter in the *AVR IAR Compiler Reference Guide* describes how to use templates to tailor a linker command file for a particular application.

THE TUTOR DIRECTORY

The `tutor` subdirectory contains the files used for the tutorials in this guide.

FILE TYPES

The AVR versions of the IAR Systems development tools use the following default filename extensions to identify the IAR-specific file types:

Ext.	*Type of file*	*Output from*	*Input to*
`a90`	Target program	XLINK	EPROM, C-SPY, etc
`c` `cpp`	C or Embedded C++ program source	Text editor	Compiler
`d90`	Target program with debug information	XLINK	C-SPY and other symbolic debuggers
`ddf`	Device description file	Text editor	C-SPY
`h`	C or Embedded C++ header source	Text editor	Compiler `#include`
`i`	Preprocessed code	Compiler	Compiler
`inc`	Assembler header	Text editor	Assembler `#include` file
`lst`	List	Compiler and assembler	–
`mac`	C-SPY macro definition	Text editor	C-SPY
`prj`	IAR Embedded Workbench project	IAR Embedded Workbench	IAR Embedded Workbench
`r90`	Object module	Compiler and assembler	XLINK and XLIB
`s90`	Assembler program source	Text editor	Assembler
`xcl`	Extended command	Text editor	XLINK
`xlb`	Librarian command	Text editor	XLIB

You can override the default filename extension by including an explicit extension when specifying a filename.

Files with the extensions `ini` and `cfg` are created dynamically when you install and run the IAR Embedded Workbench tools. These files contain information about your configuration and other settings.

Note: If you run the tools from the command line, the XLINK listings (maps) will by default have the extension `lst`, which may overwrite the list file generated by the compiler. Therefore, we recommend that you name XLINK map files explicitly, for example `project1.map`.

DOCUMENTATION

This section briefly describes the information that is available in the AVR user and reference guides, in the online help, and on the Internet.

For information about the C or Embedded C++ programming language, embedded systems programming, and the AVR architecture, see *Further reading*, page vi.

THE USER AND REFERENCE GUIDES

The user and reference guides provided with the IAR Embedded Workbench are as follows:

AVR IAR Embedded Workbench™ User Guide
This guide.

AVR IAR Compiler Reference Guide
This guide provides reference information about the AVR IAR Compiler. You should refer to this guide for information about:

- How to configure the compiler to suit your target processor and application requirements
- How to write efficient code for your target processor
- The available data types
- The run-time libraries
- The IAR language extensions
- How to migrate from the A90 IAR Compiler.

AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide
This guide provides reference information about the AVR IAR Assembler, IAR XLINK Linker, and IAR XLIB Librarian™:

- The assembler reference sections include details of the assembler source format, and reference information about the assembler operators, directives, mnemonics, and diagnostics.
- The IAR XLINK Linker reference sections provide information about XLINK options, output formats, environment variables, and diagnostics.
- The IAR XLIB Librarian reference sections provide information about XLIB commands, environment variables, and diagnostics.

ONLINE HELP AND DOCUMENTATION

From the **Help** menu in the IAR Embedded Workbench and the IAR C-SPY Debugger, you can access the AVR online documentation. Context-sensitive help is also available via the F1 button in the IAR Embedded Workbench and C-SPY windows and dialog boxes.

Online documentation
The following documentation is supplied with the product:

Help menu item	*Description*
Embedded Workbench Guide	This guide.
Compiler Reference Guide	The *AVR IAR Compiler Reference Guide.*
Assembler, Linker, and Librarian Guide	The *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide*.
C Library Reference Guide	The *General C Library Definitions Reference Guide.*
EC++ Library Reference Guide	The *C++ Library Reference.*

Recent information
We recommend that you read the following files for recent information that may not be included in the user guides:

Read-me file	*Description*
`aavr`	Assembler release notes
`clibrary`	Standard C/EC++ library release notes
`csavr`	IAR C-SPY Debugger release notes
`ewavr`	IAR Embedded Workbench release notes
`iccavr`	IAR Compiler release notes
`migrate`	ICCA90 migration information
`xlink`	IAR XLINK Linker release notes
`xman`	IAR XLINK Linker recent updates

The read-me files are located in the `avr\doc` directory.

Note: The `clib` read-me file contains release notes for the IAR C library. This library is provided for backward compatibility.

IAR ON THE WEB

The latest news from IAR Systems is available at the website **www.iar.com**. You can access the IAR site directly from the IAR Embedded Workbench **Help** menu and receive information about:

- Product announcements.
- Updates and news about current versions.
- Special offerings.
- Evaluation copies of the IAR products.
- Technical Support, including frequently asked questions (FAQs).
- Application notes.
- Links to chip manufacturers and other interesting sites.
- Distributors; the names and addresses of distributors in each country.

THE PROJECT MODEL

This chapter gives a brief discussion of the project model used by the IAR Embedded Workbench™, and explains how you use it to develop typical applications.

The concepts discussed in this chapter are illustrated in *Part 2: Tutorials* in this guide. You may find it helpful to return to this chapter while running the tutorials.

DEVELOPING PROJECTS

The IAR Embedded Workbench provides a powerful environment for developing projects with a range of different target processors, and a selection of tools for each target processor.

HOW PROJECTS ARE ORGANIZED

The IAR Embedded Workbench has been specially designed to fit in with the way that software development projects are typically organized. For example, you may need to develop related versions of an application for different versions of the target hardware, and you may also want to include debugging routines into the early versions, but not in the final code.

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

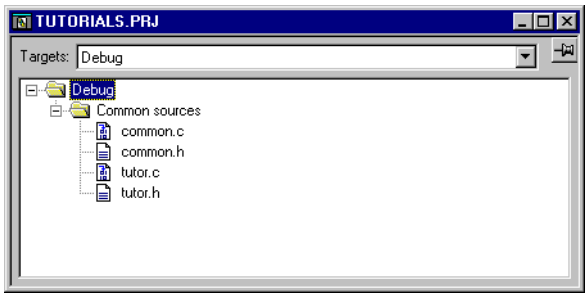
The IAR Embedded Workbench addresses these requirements, and provides a powerful environment for maintaining the source files used for building all versions of an application. It allows you to organize projects in a hierarchical tree structure showing the dependency between files at a glance.

Targets
At the highest level of the structure you specify the different target versions of your application that you want to build. For a simple application you might need just two targets, called **Debug** and **Release**. A more complex project might include additional targets for each of the different processor variants that the application is to run on.

Groups
Each target in turn contains one or more groups, which collect together related sets of source files. A group can be unique to a particular target, or it can be present in two or more targets. For example, you might create a group called **Debugging routines** which would be present only in the **Debug target**, and another group called **Common sources** which would be present in all targets.

Source files
Each group is used for grouping together one or more related source files. For maximum flexibility each group can be included in one or more targets.

When you are working with a project you always have a current target selected, and only the groups that are members of that target, along with their enclosed files, are visible in the Project window. Only these files will actually be built and linked into the output code.



SETTING OPTIONS

For each target you set global assembler and compiler options at the target level, to specify how that target should be built. At this level you typically define which processor configuration and memory model to use.

You can also set local compiler and assembler options on individual groups and source files. These local options are specific to the context of a target and override any corresponding global options set at the target level, and are specific to that target. A group can be included in two different targets and have different options set for it in each target. For example, you might set optimization high for a group containing source files that you have already debugged, but remove optimization from another group containing source files that you are still developing.

For an example where different options are set on file level, see *Tutorial 8*, page 90. For information about how to set options, see the chapters *Compiler options* and *Assembler options* in *Part 3: The IAR Embedded Workbench* in this guide.

BUILDING A PROJECT

The **Compile** command on the IAR Embedded Workbench **Project** menu allows you to compile or assemble the files of a project individually. The IAR Embedded Workbench automatically determines whether a source file should be compiled or assembled depending on the filename extension.

Alternatively, you can build the entire project using the **Make** command. This command identifies the modified files, and only recompiles or assembles those files that have changed before it relinks the project.

A **Build All** option is also provided, which unconditionally regenerates all files.

The **Compile**, **Make**, **Link**, and **Build** commands all run in the background so that you can continue editing or working with the IAR Embedded Workbench while your project is being built.

TESTING THE CODE

The compiler and assembler are fully integrated with the development environment, so that if there are errors in your source code you can jump directly from the error listing to the correct position in the appropriate source file, to allow you to locate and correct the error.

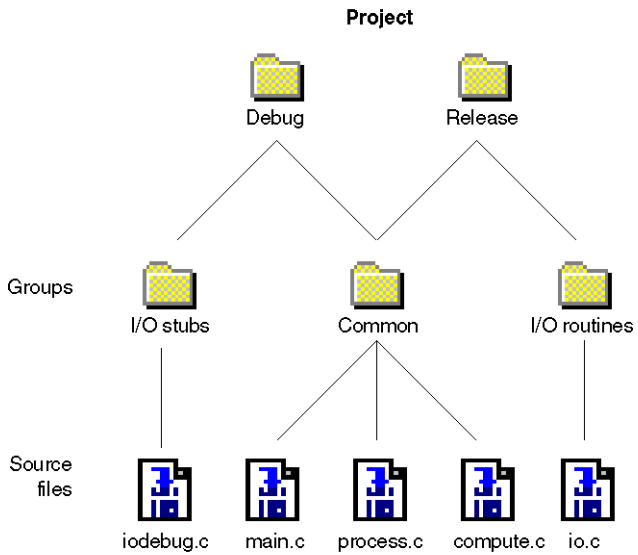
After you have resolved any problems reported during the build process, you can switch directly to C-SPY to test the resulting code at source level. The C-SPY debugger runs in a separate window, so that you can make changes to the original source files to correct problems as you identify them in C-SPY.

SAMPLE APPLICATIONS

The following examples describe two sample applications to illustrate how you would use the IAR Embedded Workbench in typical development projects.

A basic application

The following diagram shows a simple application, developed for one target processor only. Here you would manage with the two default targets, Release and Debug:



Both targets share a common group containing the project's core source files. Each target also contains a group containing the source files specific to that target: **I/O routines**, contains the source files for the input/output routines to be used in the final release code, and **I/O stubs** which contains input/output stubs to allow the I/O to be debugged with a debugger such as C-SPY.

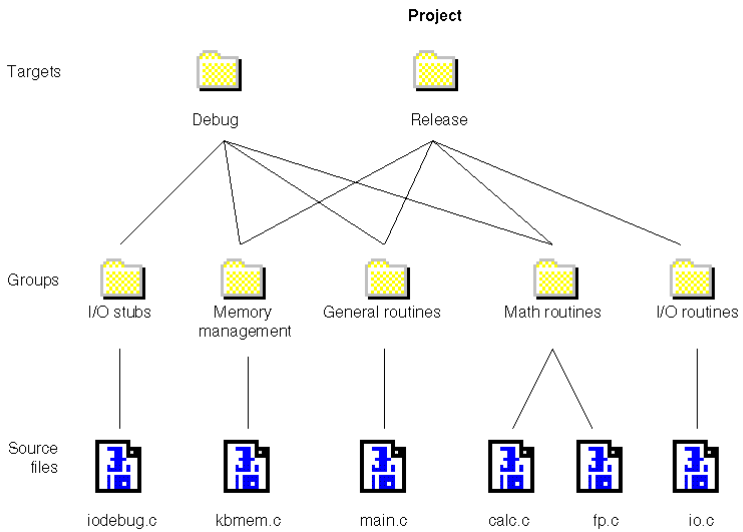
The release and debug targets would typically have different compiler options set for them; for example, you could compile the **Debug** version with trace, assertions, etc, and the **Release** version without it.

A more complex project

In the following more complex project an application is being developed for several different pieces of target hardware, containing different variants of a processor, different I/O ports and memory configurations. The project therefore includes a debug target, and a release target for each of the different sets of target hardware.

The source files that are common to all the targets are collected together, for convenience, into groups which are included in each of the targets. The names of these groups reflect the areas in the application that the source code deals with; for example math routines.

Areas of the application that depend on the target hardware, such as the memory management, are included in a number of separate groups, one per target. Finally, as before, debugging routines are provided for the Debug target.



When you are working on a large project such as this, the IAR Embedded Workbench minimizes your development time by helping you to keep track of the structure of your project, and by optimizing the development cycle by assembling and compiling the minimum set of source files necessary to keep the object code completely up to date after changes.

PART 2: TUTORIALS

This part of the AVR IAR Embedded Workbench™ User Guide contains the following chapters:

- *IAR Embedded Workbench tutorial*
- *Compiler tutorials*
- *Assembler tutorials*
- *Advanced tutorials*.

You should install the IAR development tools before running these tutorials. The installation procedure is described in the chapter *Installation and documentation*.

Notice that it may be helpful to return to the chapter *The project model* while running the tutorials.

IAR EMBEDDED WORKBENCH TUTORIAL

This chapter introduces you to the IAR Embedded Workbench™ and the IAR C-SPY® Debugger. It demonstrates how you might create and debug a small program for the IAR Compiler.

Tutorial 1 describes a typical development cycle:

- We first create a project, add source files to it, and specify target options.
- We then compile the program, examine the list file, and link the program.
- Finally we run the program in the IAR C-SPY Debugger.

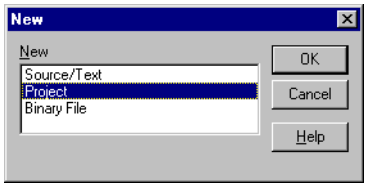
  Alternatively, you may follow this tutorial by examining the list files created. They show which areas of memory to monitor.

TUTORIAL 1

We recommend that you create a specific directory where you can store all your project files, for example the `avr\projects` directory.

CREATING A NEW PROJECT

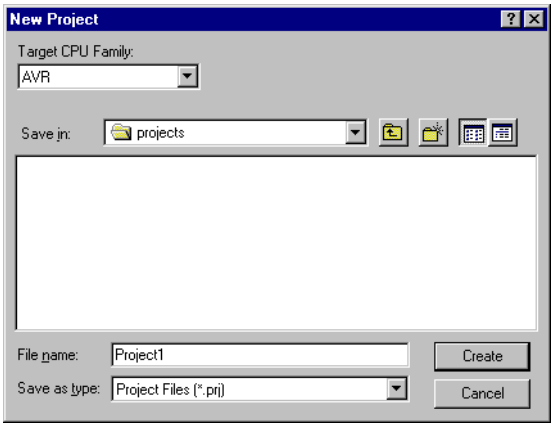
The first step is to create a new project for the tutorial programs. Start the IAR Embedded Workbench, and select **New...** from the **File** menu to display the following dialog box:



The **Help** button provides access to information about the IAR Embedded Workbench. You can at any time press the F1 key to access the online help.

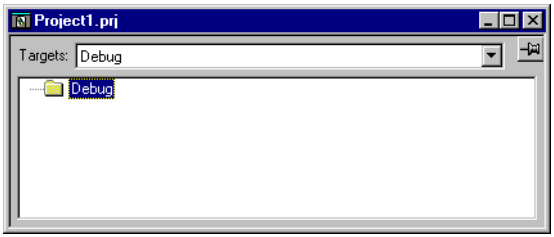
Select **Project** and click **OK** to display the **New Project** dialog box.

Enter `Project1` in the **File name** box, and set the **Target CPU Family** to **AVR**. Specify where you want to place your project files, for example in a `projects` directory:



Then click **Create** to create the new project.

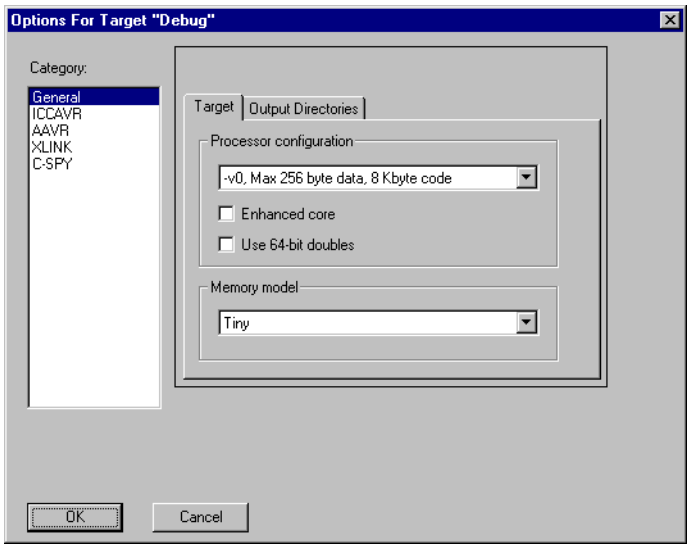
The Project window will be displayed. If necessary, select **Debug** from the **Targets** drop-down list to display the **Debug** target:



Now set up the target options to suit the processor configuration in this tutorial.

Select the **Debug** folder icon in the Project window and choose **Options...** from the **Project** menu. The **Target** options page in the **General** category is displayed.

In this tutorial we use the default settings. Make sure that the **Processor configuration** is set to **-v0, Max 256 byte data, 8 Kbyte code** and that the **Memory model** is set to **Tiny**:



Then click **OK** to save the target options.

THE SOURCE FILES

This tutorial uses the source files `tutor.c` and `common.c`, and the include files `tutor.h` and `common.h`, which are all supplied with the product.

The program initializes an array with the ten first Fibonacci numbers and prints the result in the Terminal I/O window.

The tutor.c program

The `tutor.c` program is a simple program using only standard C or Embedded C++ facilities. It repeatedly calls a function that prints a number series to the Terminal I/O window in C-SPY. A copy of the program is provided with the product.

```
#include "tutor.h"


   /* Global call counter */
int call_count;

   /* Get and print next Fibonacci number. */
void do_foreground_process(void)
{
  unsigned int fib;
  ++call_count;
  fib = get_fibonacci( call_count );
  put_value( fib );
}

   /* Main program. Prints the Fibonacci numbers. */
void main(void)
 {
 call_count = 0;
 init_fibonacci();
 while (  call_count < MAX_FIBONACCI )
   do_foreground_process();
 }
```

The common.c program

The `common.c` program, which is also provided with the product, contains utility routines for the Fibonacci calculations:

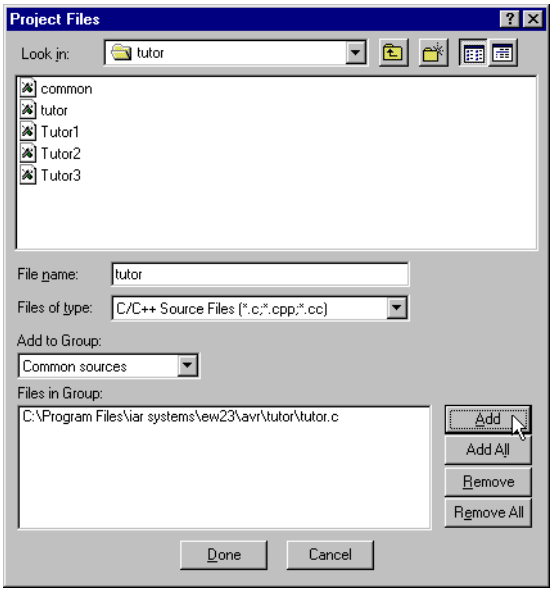
```
#include <stdio.h>
#include "common.h"

static unsigned int fibonacci[MAX_FIBONACCI];

/* Initialize the array above with the first Fibonacci
numbers*/
void init_fibonacci(void)
  {
    char i;

    fibonacci[0] = 1;
    fibonacci[1] = 1;
```

```
    for(i=2; i<MAX_FIBONACCI; ++i)
      fibonacci[i] = fibonacci[i-1] + fibonacci[i-2];
  }

/* Get the n:th fibonacci number, or 0 if the */
/* index is greater than MAX_FIBONACCI.       */
unsigned int get_fibonacci(char index)
  {
    if (index >= MAX_FIBONACCI)
      return 0;

    return fibonacci[index];
  }

/* Print the given number to the standard output */
void put_value(unsigned int value)
  {
    char buf[8], *p, ch;

    p = buf;
    *p++ = 0;
    do
      {
        *p++ = '0' + value % 10;
        value /= 10;
      } while(value != 0);
    *p++ = '\n';

    while((ch = *--p) != 0)
      putchar(ch);
  }
```

ADDING FILES TO THE PROJECT

We will now add the `tutor.c` and `common.c` source files to the `Project1` project.

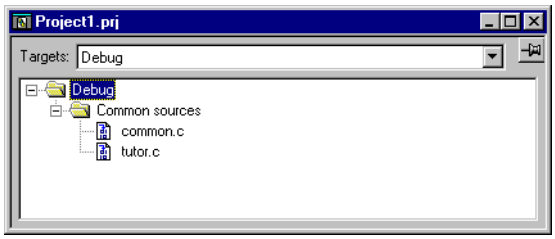
Choose **Files...** from the **Project** menu to display the **Project Files** dialog box. Locate the file `tutor.c` in the file selection list in the upper half of the dialog box, and click **Add** to add it to the **Common Sources** group.



Then locate the file `common.c` and add it to the group.

Finally click **Done** to close the **Project Files** dialog box.

Click on the plus sign icon to display the file in the Project window tree display:

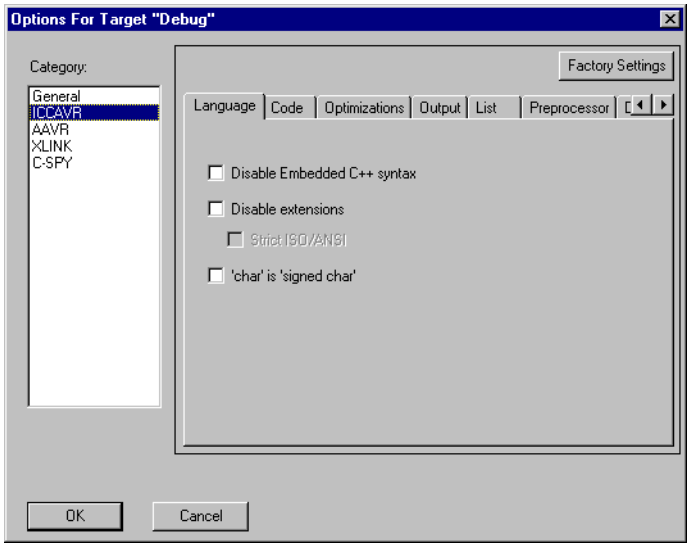


The **Common Sources** group was created by the IAR Embedded Workbench when you created the project. More information about groups is available in the chapter *The project model* in *Part 1: The IAR development tools* in this guide.

SETTING COMPILER OPTIONS

Now you should set up the compiler options for the project.

Select the **Debug** folder icon in the Project window, choose **Options...** from the **Project** menu, and select **ICCAVR** in the **Category** list to display the IAR Compiler options pages:



Make sure that the following options are selected on the appropriate pages of the **Options** dialog box:

Page	*Options*
Language	Enable extensions
Optimizations	Optimizations, Size: Low
Output	Generate debug information
List	C list file Assembler mnemonics

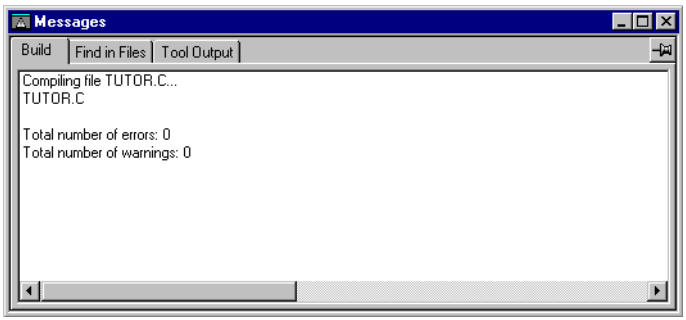
When you have made these selections, click **OK** to set the options you have specified.

COMPILING THE TUTOR.C AND COMMON.C FILES

To compile the `tutor.c` file, select it in the Project window and choose **Compile** from the **Project** menu.

Alternatively, click the **Compile** button in the toolbar or select the **Compile** command from the pop-up menu that is available in the Project window. It appears when you click the right mouse button.

The progress will be displayed in the Messages window.



You can specify the amount of information to be displayed in the Messages window. In the **Options** menu, select **Settings...** and then select the **Make Control** page.

Compile the file `common.c` in the same manner.

The IAR Embedded Workbench has now created new directories in your project directory. Since you have chosen the **Debug** target, a `Debug` directory has been created containing the new directories `List`, `Obj`, and `Exe`:

- In the `list` directory, your list files from the `Debug` target will be placed. The list files have the extension `lst` and will be located here.
- In the `obj` directory, the object files from the compiler and the assembler will be placed. These files have the extension `r90` and will be used as input to the IAR XLINK Linker.
- In the `exe` directory, you will find the executable files. These files have the extension `d90` and will be used as input to the IAR C-SPY Debugger. Notice that this directory will be empty until you have linked the object files.

VIEWING THE LIST FILE

Open the list file `tutor.lst` by selecting **Open...** from the **File** menu, and selecting `tutor.lst` from the `debug\list` directory. Examine the list file, which contains the following information:

The *header* shows the product version, information about when the file was created, and the command line version of the compiler options that were used:

```
#############################################################################
 IAR AVR   AVR C/EC++ Compiler Vx.Xxxx/xxx         dd/Mmm/yyyy  hh:mm:ss
 Copyright 2000 IAR Systems. All rights reserved.

    Source file  =  C:\Program Files\iar systems\ew23\avr\tutor\tutor.c
    Command line =  -v0 -mt -o "C:\Program Files\iar
                    systems\ew23\avr\projects\Debug\Obj\" -I "C:\Program
                    Files\iar systems\ew23\avr\INC\" -lcN "C:\Program
                    Files\iar systems\ew23\avr\projects\Debug\List\"
                    --ec++ -e --initializers_in_flash -z3 --no_cse
                   --no_inline --no_code_motion --no_cross_call --debug
                   "C:\Program Files\iar systems\ew23\avr\tutor\tutor.c"
    List file    =  C:\Program Files\iar systems\ew23\avr\projects\
                    Debug\List\tutor.lst
    Object file  =  C:\Program Files\iar systems\ew23\avr\projects\
                    Debug\Obj\tutor.r90
#############################################################################
```

The *body* of the list file shows the assembler code and binary code generated for each C or Embedded C++ statement. It also shows how the variables are assigned to different segments:

```
    16          void do_foreground_process(void)
    17          {
    18            unsigned int fib;
    19            ++call_count;
  \                       do_foreground_process:
  \   00000000   ....                         LDI     R30,LOW(call_count)
  \   00000002   ....                         LDI     R31,call_count >> 8
  \   00000004   8120                         LD      R18,Z
  \   00000006   8131                         LDD     R19,Z+1
  \   00000008   5F2F                         SUBI    R18,255
  \   0000000A   4F3F                         SBCI    R19,255
  \   0000000C   8331                         STD     Z+1,R19
  \   0000000E   8320                         ST      Z,R18
    20            fib = get_fibonacci( call_count );
  \   00000010   ....                         LDI     R30,LOW(call_count)
  \   00000012   ....                         LDI     R31,call_count >> 8
  \   00000014   8100                         LD      R16,Z
  \   00000016   ....                         RCALL   get_fibonacci
    21            put_value( fib );
  \   00000018   ....                         RCALL   put_value
    22          }
  \   0000001A   9508                         RET
```

The *end* of the list file shows the amount of stack, code, and data memory required, and contains information about error and warning messages that may have been generated:

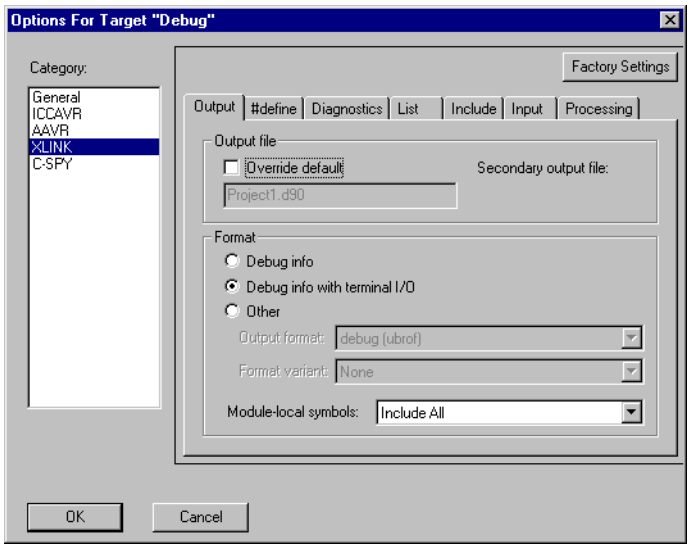
```
Maximum stack usage in bytes:

Function                    CSTACK  RSTACK
    --------                    ------  ------
    do_foreground_process           0       2
      -> get_fibonacci              0       2
      -> put_value                  0       2
    main                            0       2
      -> init_fibonacci             0       2
      -> do_foreground_process      0       2
```

```
     64 bytes in segment CODE
      4 bytes in segment INITTAB
      2 bytes in segment TINY_Z

     64 bytes of CODE memory (+ 4 bytes shared)
      2 bytes of DATA memory

Errors: none
Warnings: none
```

LINKING THE TUTOR.C PROGRAM

First set up the options for the IAR XLINK Linker™:

Select the **Debug** folder icon in the Project window and choose **Options...** from the **Project** menu. Then select **XLINK** in the **Category** list to display the XLINK options pages:



Make sure that the following options are selected on the appropriate pages of the **Options** dialog box:

Page	*Options*
Output	Debug info with terminal I/O

Page	*Options*
List	Generate linker listing Segment map Module map
Include	`lnk0t.xcl` (the linker command file)

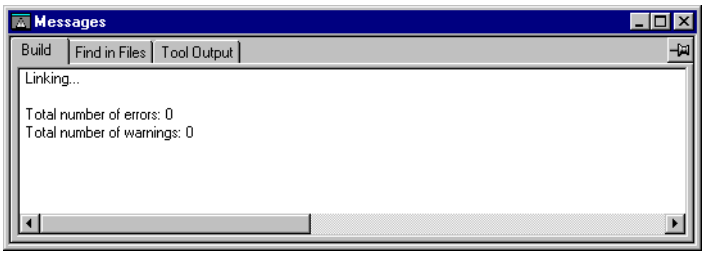
If you want to examine the linker command file, use a suitable text editor, such as the IAR Embedded Workbench editor, or print a copy of the file, and verify that the entries match your requirements.

The definitions in the linker command file are not permanent; they can be altered later on to suit your project if the original choice proves to be incorrect, or less than optimal. For more information about linker command files, see the *Configuration* chapter in the *AVR IAR Compiler Reference Guide.*

Click **OK** to save the XLINK options.

Note: The chapter *XLINK options* in *Part 3: The IAR Embedded Workbench* in this guide contains information about the XLINK options available in the IAR Embedded Workbench. In the linker command file, XLINK command line options such as `-P` and `-Z` are used for segment control. These options are described in the chapters *Introduction to the IAR XLINK Linker* and *XLINK options* in the *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide*.

Now you should link the object file to generate code that can be debugged. Choose **Link** from the **Project** menu. The progress will be displayed in the Messages window:



The result of the linking is a code file `project1.d90` with debug information and a map file `project1.map`.

Viewing the map file

Examine the `project1.map` file to see how the segment definitions and code were placed into their physical addresses. Following are the main points of interest in a map file:

- The header includes the options used for linking.
- The `CROSS REFERENCE` section shows the address of the program entry.
- The `RUNTIME MODEL` section shows the runtime model attributes that are used.
- The `MODULE MAP` shows the files that are linked. For each file, information about the modules that were loaded as part of the program, including segments and global symbols declared within each segment, is displayed.
- The `SEGMENTS IN ADDRESS ORDER` section lists all the segments that constitute the program.

Viewing the build tree

In the Project window, press the right mouse button and select **Save as Text...** from the pop-up menu that appears. This creates a text file that allows you to conveniently examine the options for each level of the project.

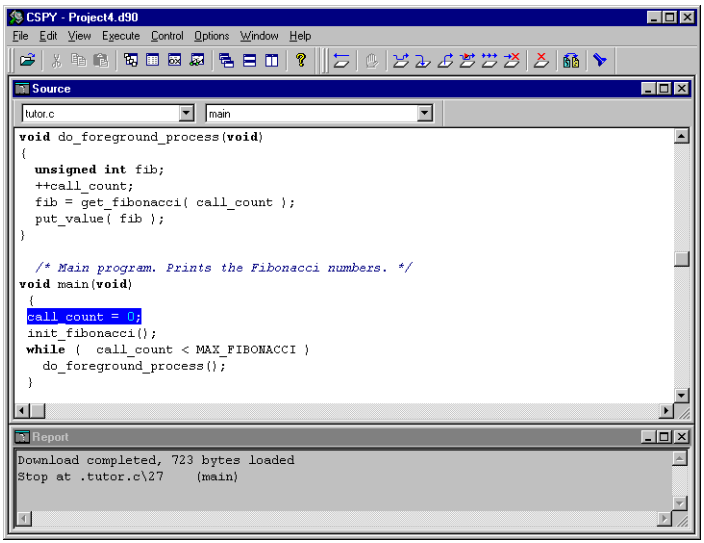
Notice that the text file will contain the command line equivalents to the options that you have specified in the IAR Embedded Workbench. The command line options are described in the *AVR IAR Compiler Reference Guide* and *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide*, respectively.

RUNNING THE PROGRAM

Now we will run the `project1.d90` program using the IAR C-SPY Debugger to watch variables, set a breakpoint, and print the program output in the Terminal I/O window.

Choose **Debugger** from the **Project** menu in the IAR Embedded Workbench. Alternatively, click the C-SPY button in the toolbar.

The following C-SPY window will be opened for this file:

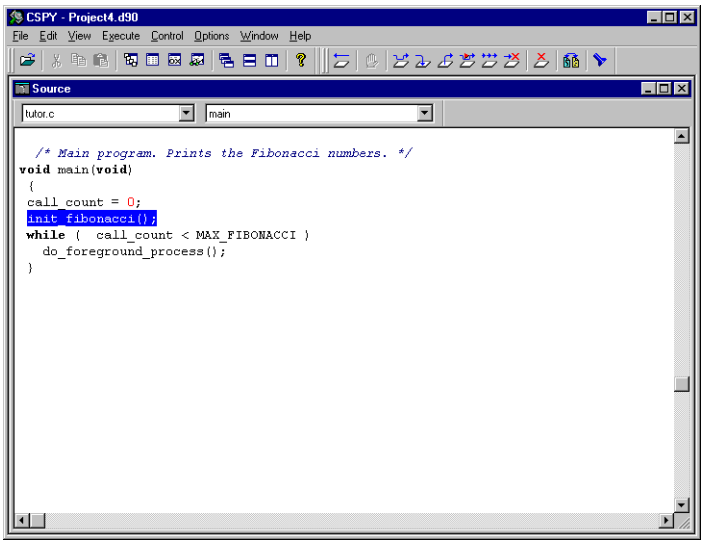


C-SPY starts in source mode, and will stop at the first executable statement in the `main` function. The current position in the program, which is the next C or Embedded C++ statement to be executed, is shown highlighted in the Source window.

The corresponding assembler instructions are always available. To inspect them, select **Toggle Source/Disassembly** from the **View** menu. Alternatively, click the **Toggle Source/Disassembly** button in the toolbar. In disassembly mode stepping is executed one assembler instruction at a time. Return to source mode by selecting **Toggle Source/Disassembly** again.

Execute one step by choosing **Step** from the **Execute** menu. Alternatively, click the **Step** button in the toolbar. At source level **Step** executes one source statement at a time.

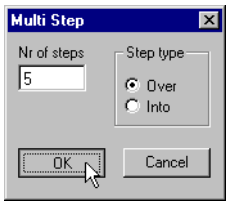
The current position should be the call to the `init_fibonacci` function:



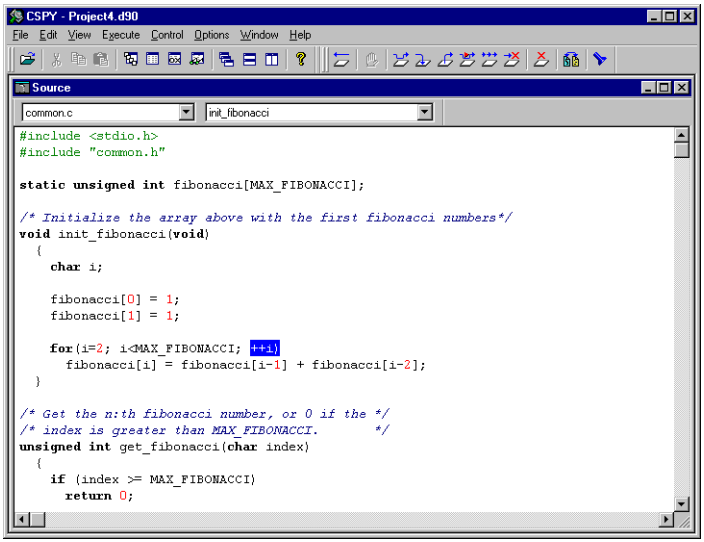
Select **Step Into** from the **Execute** menu to execute `init_fibonacci` one step at the time. Alternatively, click the **Step Into** button in the toolbar.

When **Step Into** is executed you will notice that the file in the **Source file** list box (to the upper left in the Source window) changes to `common.c` since the function `init_fibonacci` is located in this file. The **Function** list box (to the right of the **Source file** list box) shows the name of the function where the current position is.

Step five more times. Choose **Multi Step...** from the **Execute** menu, and enter 5.



You will notice that the three individual parts of a `for` statement are separated, as C-SPY debugs on statement level, not on line level. The current position should now be `++i`:



WATCHING VARIABLES

C-SPY allows you to set watchpoints on C or Embedded C++ variables or expressions, to allow you to keep track of their values as you execute the program. You can watch variables in a number of ways; for example, you can watch a variable by pointing at it in the Source window with the mouse pointer, or by opening the Locals window. Alternatively, you can open the QuickWatch window from the pop-up menu that appears when you press the right mouse button in the Source window.

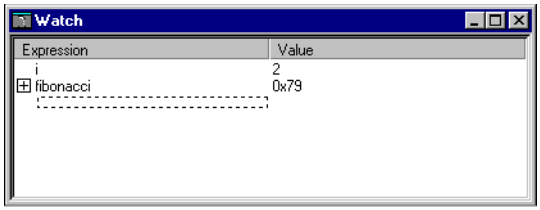
Here we will use the Watch window. Choose **Watch** from the **Window** menu to open the Watch window, or click the **Watch Window** button in the toolbar. If necessary, resize and rearrange the windows so that the Watch window is visible.

Setting a watchpoint

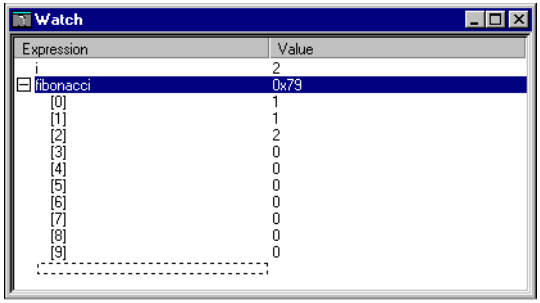
Set a watchpoint on the variable `i` using the following procedure: Select the dotted rectangle, then click and briefly hold the left mouse button. In the entry field that appears when you release the button, type `i` and press the Enter key.

You can also drag and drop a variable in the Watch window. Select the `fibonacci` array in the `init_fibonacci` function. When `fibonacci` is marked, drag and drop it in the Watch window.

The Watch window will show the current value of `i` and `fibonacci`:



`fibonacci` is an array and can be watched in more detail. This is indicated in the Watch window by the plus sign icon to the left of the variable. Click the symbol to display the current contents of `fibonacci`:



Now execute some more steps to see how the values of `i` and `fibonacci` change.

Variables in the Watch window can be specified with module name and function name to separate variables that appear with the same name in different functions or modules. If no module or function name is specified, its value in the current context is shown.

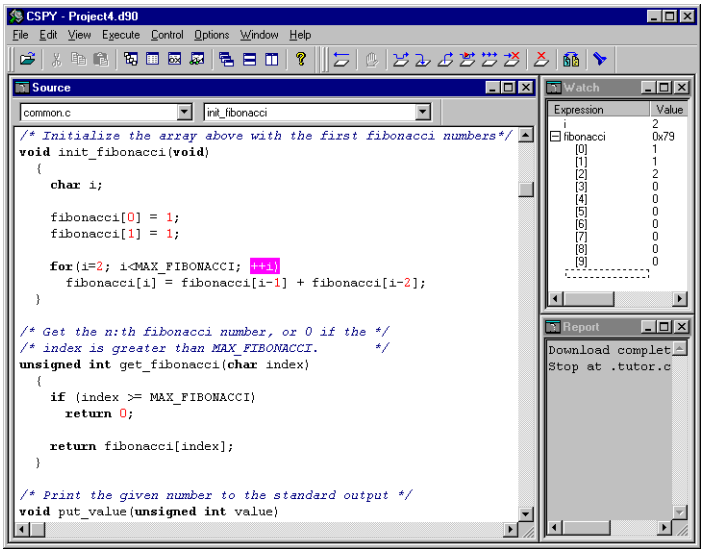
SETTING BREAKPOINTS

You can set breakpoints at C or Embedded C++ function names or line numbers, or at assembler symbols or addresses. The most convenient way is usually to set breakpoints interactively, simply by positioning the cursor in a statement and then choosing the **Toggle Breakpoint** command. For additional information, see *Toggle breakpoint*, page 224.

To display information about breakpoint execution, make sure that the Report window is open by choosing **Report** from the **Window** menu. You should now have the Source, Report, and Watch windows on the screen; position them neatly before proceeding.

Set a breakpoint at the statement `++i` using the following procedure: First click in this statement in the Source window, to position the cursor. Then choose **Toggle Breakpoint** from the **Control** menu, or click the **Toggle Breakpoint** button in the toolbar.

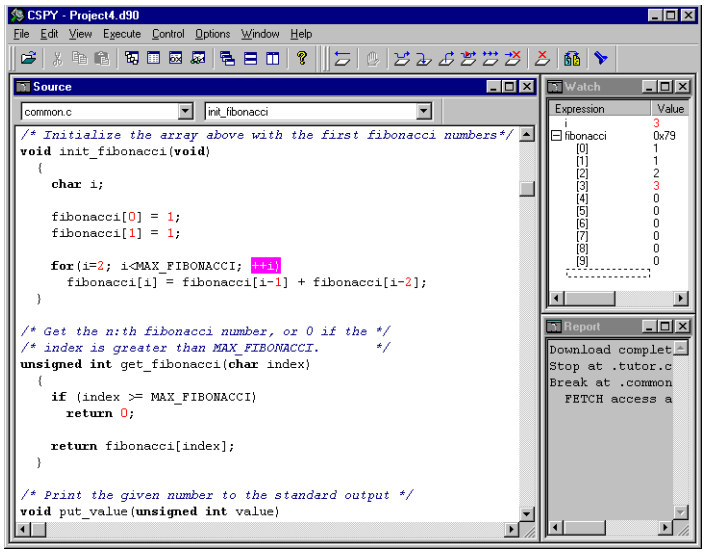
A breakpoint will be set at this statement, and the statement will be highlighted to show that there is a breakpoint there:



EXECUTING UP TO A BREAKPOINT

To execute the program continuously, until you reach a breakpoint, choose **Go** from the **Execute** menu, or click the **Go** button in the toolbar.

The program will execute up to the breakpoint you set. The Watch window will display the value of the `fibonacci` expression and the Report window will contain information about the breakpoint:



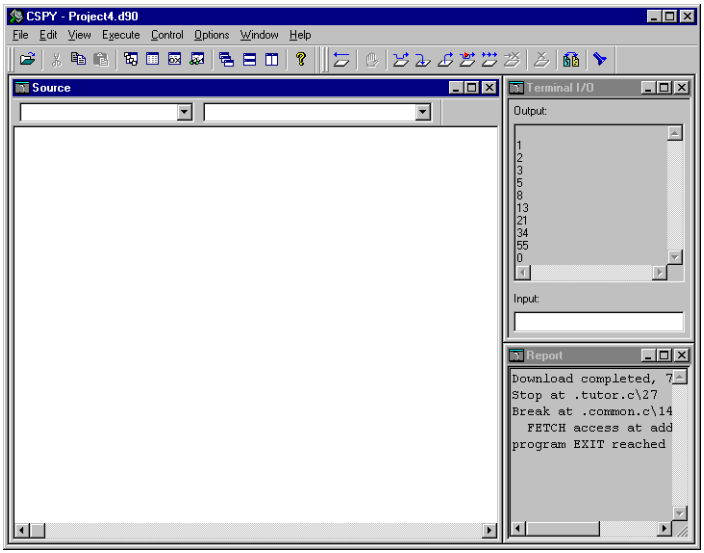
Remove the breakpoint by selecting **Edit breakpoints...** from the **Control** menu. Alternatively, click the right mouse button to display a pop-up menu and select **Edit breakpoints...**. In the **Breakpoints** dialog box, select the breakpoint in the **Breakpoints** list and click **Clear**. Then close the **Breakpoints** dialog box.

CONTINUING EXECUTION

Open the Terminal I/O window, by choosing **Terminal I/O** from the **Window** menu, to display the output from the `I/O` operations.

To complete execution of the program, select **Go** from the **Execute** menu, or click the **Go** button in the toolbar.

Since no more breakpoints are encountered, C-SPY reaches the end of the program and erases the contents of the Source window. A `program EXIT reached` message is printed in the Report window:



If you want to start again with the existing program, select **Reset** from the **Execute** menu, or click the **Reset** button in the toolbar.

EXITING FROM C-SPY

To exit from C-SPY choose **Exit** from the **File** menu.

C-SPY also provides many other debugging facilities. Some of these—for example defining virtual registers, using C-SPY macros, debugging in disassembly mode, displaying function calls, profiling the application, and displaying code coverage—are described in the following tutorial chapters.

For complete information about the features of C-SPY, see the chapter *C-SPY reference* in *Part 4: The C-SPY simulator* in this guide.

COMPILER TUTORIALS

This chapter introduces you to some of the IAR Compiler's AVR-specific features:

- Tutorial 2 demonstrates how to utilize AVR peripherals with the IAR Compiler features. The `#pragma language` directive allows us to use the AVR-specific language extensions. Our program will be extended to handle polled I/O. Finally, we run the program in C-SPY and create virtual registers.
- In Tutorial 3, which is written in Embedded C++, we modify the tutorial project by adding an interrupt handler. The system is extended to handle the real-time interrupt using the AVR IAR Compiler intrinsics and keywords. Finally, we run the program using the C-SPY interrupt system in conjunction with complex breakpoints and macros.

Before running these tutorials, you should be familiar with the IAR Embedded Workbench and the IAR C-SPY Debugger as described in the previous chapter, *IAR Embedded Workbench tutorial*.

TUTORIAL 2

This IAR Compiler tutorial will demonstrate how to simulate the AVR Universal Asynchronous Receiver/Transmitter (UART) using the IAR Compiler features.

THE TUTOR2.C SERIAL PROGRAM

The following listing shows the `tutor2.c` program. A copy of the program is provided with the product.

```
#include <stdio.h>
#include <io8515.h>
#include "common.h"

/* enable use of extended keywords */
#pragma language=extended

/* The kRXC flag is set in the USR register */
/* when a character has been received.          */
#define kRXC      (0x80)
```

```
/***********************************
 *       Global variables          *
 ***********************************/
int call_count;
int loop_count;

/***********************************
 *        Start of code            *
 ***********************************/
void do_foreground_process(void)
  {
    unsigned int fib;

    if (!(USR & kRXC))
      putchar('.');
    else
      {
        fib = get_fibonacci(call_count);
        call_count++;
        put_value(fib);
      }
  }

void main(void)
  {
    loop_count = 0;

    /* Initialize the fibonacci numbers */
    init_fibonacci();

    /* now loop forever, taking input when ready */
    while(call_count < MAX_FIBONACCI)
      {
        do_foreground_process();
        ++loop_count;
      }
  }
```

COMPILING AND LINKING THE TUTOR2.C SERIAL PROGRAM

Modify the `project1` project by replacing `tutor.c` with `tutor2.c`:

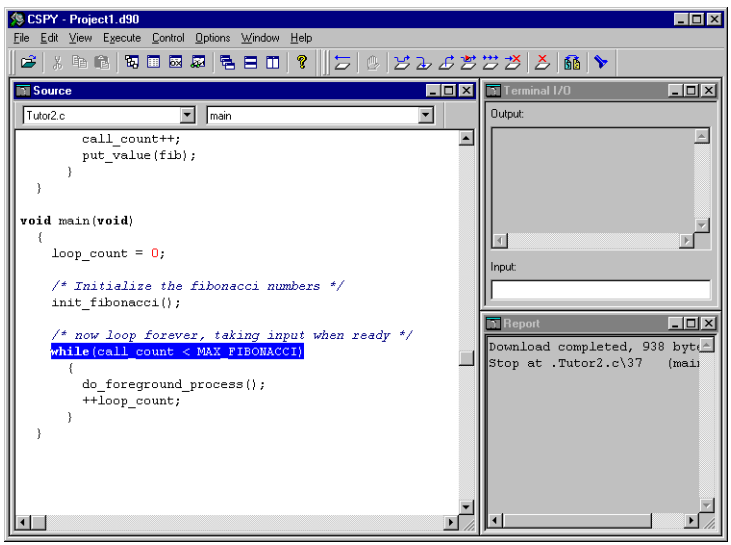
Choose **Files...** from the **Project** menu. In the dialog box **Project Files,** mark the file `tutor.c` in the **Files in Group** box. Click on the **Remove** button to remove the `tutor.c` file from the project. In the **File Name** list box, select the `tutor2.c` file and click on the **Add** button. Now the **Files in Group** should contain the files `common.c` and `tutor2.c`.

Select **Options...** from the **Project** menu. In the **General** category, select **--cpu = 8515, AT90S8515** and the **Small** memory model. In the **ICCAVR** category, disable the Embedded C++ syntax. Make sure that language extensions are enabled and that debug information will be generated.

Now you can compile and link the project by choosing **Make** from the **Project** menu.

RUNNING THE TUTOR2.C SERIAL PROGRAM

Start the IAR C-SPY Debugger and run the modified `project1` project. Step until you reach the `while` loop, where the program waits for input. Open the Terminal I/O window, where the `tutor2` result will be printed.



DEFINING VIRTUAL REGISTERS

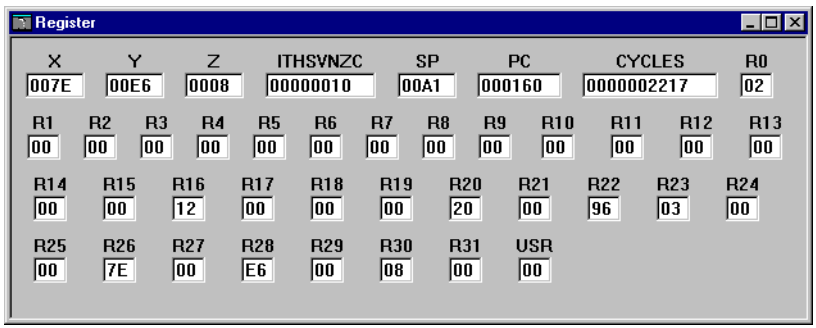
To simulate different values for the serial interface, we will make a new virtual register called USR.

Choose **Settings...** from the **Options** menu. On the **Register Setup** page, click the **New** button to add a new register. Now the **Virtual Register** dialog box will appear.

Enter the following information in the dialog box:

Input field	*Input*	*Description*
Name	USR	Virtual register name
Size	1	One byte
Base	16	Binary values
Address	0B	Memory location (in hex)
Segment	I/O-SPACE	Segment name

Then choose **OK** in both dialog boxes. Open the Register window from the **Window** menu, or select the **Register Window** button in the toolbar. USR should now be included in the list of registers. The current value of each bit in the serial interface register is shown:



As you step through the program you can enter new values into USR in the Register window. When the eighth bit (0x80) is set, a new Fibonacci number will be printed, and when the bit is cleared, a period (.) will be printed instead.

When the program has finished, you may exit from C-SPY.

TUTORIAL 3

In this tutorial we simulate a basic timer. We will define an interrupt function that handles the timer, and we will use the C-SPY macro system to simulate the timer.

THE TUTOR3.CPP TIMER PROGRAM

The following is a complete listing of the `tutor3.cpp` timer program. A copy of the program is provided with the product.

```
#include <stdio.h>
#include <io8515.h>
#include <inavr.h>
#include "common.h"

/* enable use of extended keywords */
#pragma language=extended

/***********************************
 *       Global variables          *
 ***********************************/
volatile static char  print_flag;
volatile unsigned int fib;
static int            call_count;

/***********************************
 *     Base class declaration      *
 ***********************************/
class TimerOBaseclass
  {
  public:
    // Constructor
    TimerOBaseclass();

    // Destructor
    ~TimerOBaseclass();

    // This method stops the timer.
    void Stop();

    // This method starts the timmer.
    void Start();
```

```
  // This method sets the timer period (in cycles) for
  // the next cycle.
  void SetPeriod(unsigned int period);

protected:
  // Since this method is an abstract method (the "= 0"
  // at the end of the line) it must be implemented
  // in the implementation sub-class.
  virtual void OverflowCallback() = 0;

  // This member contains the value to write to the
  // TCNT0 register every interrupt cycle to get
  // the correct period.
  unsigned char mTCNT;

  // This variable is true if the timer is running.
  bool mEnabled;

private:
  // The names look like timer 0 in the 8515 but they
  // do not work as in the 8515.
  static volatile __io unsigned char TCNT @ 0x32;
  static volatile __io struct
    {
      unsigned char Enabled:1,
                    pad:7;
    } TCCR @ 0x33;

  // The interrupt handler is a private static member.
#pragma vector=TIMER0_OVF0_vect
  static __interrupt void Overflow()
    {
      TCNT = sInstance->mTCNT;

      sInstance->OverflowCallback();
    }

  static Timer0Baseclass *sInstance;
};
```

```
/**********************************
 *    Base class implementation    *
 **********************************/
TimerOBaseclass *TimerOBaseclass::sInstance;

TimerOBaseclass::TimerOBaseclass() :
  mTCNT(0), mEnabled(false)
  {
    // Make sure the timer is stoped!
    TCCR.Enabled = false;

    // Create a pointer to this instance...
    sInstance = this;
  }

TimerOBaseclass::~TimerOBaseclass()
  {
    // Stop the timer!
    Stop();
  }

void TimerOBaseclass::Stop()
  {
    // Set flag
    mEnabled = false;

    // Stop the timer by disabling it.
    TCCR.Enabled = mEnabled;
  }

void TimerOBaseclass::Start()
  {
   // Start the timer by setting up the counter register...
    TCNT = mTCNT;

    // Set flag
    mEnabled = true;

    // Start the timer.
    TCCR.Enabled = mEnabled;
  }
```

```
void TimerOBaseclass::SetPeriod(unsigned int period)
  {
    // Calculate the timer counter value to use
    unsigned char newTCNT;

    newTCNT = 0xFF - (period / 256);

    // Disable interrupts while changing the data...
    __disable_interrupt();

    mTCNT = newTCNT;

    __enable_interrupt();
  }

/**********************************
 *    Derived class declaration    *
 **********************************/
class TimerTutorial : public TimerOBaseclass
  {
  public:
    // The "end-user" callback.
    void OverflowCallback();
  };

/**********************************
 *   Derived class implementation  *
 **********************************/

void TimerTutorial::OverflowCallback()
  {
   // Make sure that we're done printing the last fibonacci
   // number before trying to print a new one...
    if (!print_flag)
      {
        fib = get_fibonacci(call_count);
        call_count++;

        print_flag = 1;
      }
  }
```

```
/**********************************
 *     Start of program code      *
 **********************************/
void do_foreground_process()
  {
    if (!print_flag)
      putchar('.');
    else
      {
        put_value(fib);
        print_flag = 0;
      }
  }

void main()
{
  TimerTutorial timer;

  // Initialize the fibonacci number generator
  init_fibonacci();

  // Initialize the local variables
  print_flag = 0;
  call_count = 0;

  // Enable interrupts
  __enable_interrupt();

  timer.SetPeriod(10000);
  timer.Start();

  // now loop forever, taking input when ready
  while(call_count < MAX_FIBONACCI)
  {
    do_foreground_process();
  }
}
```

The address for the interrupt handler and the actual interrupt vector `TIMER0_OVF0_vect` are defined in the header file `io8515.h`.

The start address of the interrupt handler must be located at the correct offset in the interrupt table. Use the directive `#pragma vector` to specify the offset in the interrupt table. This vector offset is then valid for the subsequent interrupt-declared function.

Use the `__interrupt` keyword to define the interrupt handler:

```
#pragma vector=TIMER0_OVF0_vect
    static __interrupt void Overflow()
```

The extended keywords and `#pragma` directives are described in the *AVR IAR Compiler Reference Guide*.

The interrupt handler will fetch the latest Fibonacci value from the `get_fibonacci` function and place it in the `fib` buffer. It will then set the print flag, which makes the main program print the value by using the `put_value` function.

The main program enables interrupts, initializes the timer, and then starts printing periods (`.`) in the foreground process while waiting for interrupts.

THE C-SPY TUTOR3.MAC MACRO FILE

In the C-SPY macro file called `tutor3.mac`, we use system and user-defined macros. Notice that this example is not intended as an exact simulation; the purpose is to illustrate a situation where C-SPY macros can be useful. For detailed information about macros, see the chapter *C-SPY macros* in *Part 4: The C-SPY simulator* in this guide.

Initializing the system

The macro `execUserSetup()` is automatically executed during C-SPY setup.

First we print a message in the C-SPY Report window so that we know that this macro has been executed. For additional information, see *Report window*, page 217.

Then we initialize the two control registers at addresses `0x32` and `0x33` to zero. We then continue to set up three data break points in the `I/O-SPACE` and connect them to C-SPY macros that are also defined in the file `tutor3.cpp`. Finally we make sure that there are no pending interrupts.

```
execUserSetup()
{
  message "execUserSetup() called\n";

  // mark timer as initially inactive
  _TimerActive = 0;

  // Clear the control registers
  __writeMemoryByte(0, 0x32, "I/O-SPACE");
  __writeMemoryByte(0, 0x33, "I/O-SPACE");

  // Setup up a write and read breakpoint on the TCNT
register (0x32)
  __setBreak("0x32", "I/O-SPACE", 1, 1, "", "TRUE", "I",
"_readTCNT()");
  __setBreak("0x32", "I/O-SPACE", 1, 1, "", "TRUE", "W",
"_writeTCNT()");

  // Setup up a write breakpoint on the TCCR register
(0x33)
  __setBreak("0x33", "I/O-SPACE", 1, 1, "", "TRUE", "W",
"_writeTCCR()");

  // Cancel all pending interrupts
  __cancelAllInterrupts();
}
```

Generating interrupts

In the C-SPY macro _my_OrderInterrupt the __orderInterrupt system macro orders C-SPY to generate interrupts.

```
_myOrderInterrupt()
{
  _ActivationTime = #CYCLES;
  _ActivationTCNT = _TCNT;

  _TID = __orderInterrupt("0x0A", #CYCLES + 256L *
                          (256L - _TCNT), 256L * 256L, 0,
                          0, 100);
}
```

The following parameters are used:

`0x0A`	Specifies which interrupt vector to use.
`#CYCLES`	Specifies the activation moment for the interrupt. The interrupt is activated when the cycle counter has passed this value. The interrupt activation time is calculated as an offset from the current cycle count `#CYCLES`.
`65536L`	Specifies the repeat interval for the interrupt, measured in clock cycles.
`0`	Time variance, not used here.
`0`	Latency, not used here.
`100`	Specifies probability. Here it denotes 100 %. We want the interrupt to occur at the given frequency. Another percentage could be used for simulating a more randomized interrupt behavior.

During execution, C-SPY will wait until the cycle counter has passed the activation time. Then it will, with 100 % certainty, generate an interrupt approximately every 65536 cycles.

Using breakpoints to simulate incoming values
We must also simulate the values of the timer counter register `TCNT`. This is done by setting a breakpoint at the address of the timer counter register (`0x32`) and connecting a user-defined macro to it. Here we use the `__setBreak` system macro.

The following parameters are used:

`0x32`	Receive buffer address.
`"I/O-SPACE"`	The memory segment where this address is found. The segments `DATA`, `CODE`, `I/O-SPACE`, and `EEPROM` are valid for the AVR product.
`1`	Length.
`1`	Count.
`""`	Denotes unconditional breakpoint.
`"TRUE"`	Condition type.

`"I"`	The memory access type. Here we use "Read Immediate" which means that C-SPY breaks *before* reading the value at the specified address. This gives us the opportunity to put the correct timer counter value in the timer counter register before C-SPY reads the value.
`"_readTCNT()"`	The macro connected to the breakpoint.

During execution, when C-SPY detects a read from the timer counter address, it will temporarily halt the simulation and run the `_readTCNT` macro. Since this macro ends with a `resume` statement, C-SPY will then resume the simulation and start by reading the receive buffer value.

The `_readTCNT` macro is executed whenever C-SPY tries to read the value of the timer counter register, as defined in the `__setBreak` macro:

```
_readTCNT()
{
  if (_TimerActive != 0)
  {
    // Adjust the value in TCNT and write the new content
    // to memory.
    _adjustTCNT();
    __writeMemoryByte(_TCNT, 0x0B, "I/O-SPACE");
  }
  resume;
}
```

First we check if the timer is active. The value of the timer counter is only updated if the timer is active. Next we calculate the new value of the `_TCNT` variable by calling the `_adjustTCNT` macro. We then write the new value of the `_TCNT` variable to memory using the `__writeMemoryByte` system macro. Finally, the `resume` statement causes C-SPY to continue the simulation process.

Resetting the system
The macro `execUserReset()` is automatically executed during C-SPY reset. At reset, we want to cancel all outstanding interrupts, mark the timer as inactive, and reset the two control registers:

```
execUserReset()
{
  message "execUserReset() called\n";

  // Cancel all pending interrupts
  __cancelAllInterrupts();

  // mark timer as inactive
  _TimerActive = 0;

  // Clear the control registers
  __writeMemoryByte(0, 0x33, "I/O-SPACE");
  __writeMemoryByte(0, 0x32, "I/O-SPACE");
}
```

Exiting the system
The macro `execUserExit()` is executed automatically during C-SPY exit:

```
execUserExit()
{
  message "execUserExit() called\n";

  // Cancel all pending interrupts
  __cancelAllInterrupts();
}
```

COMPILING AND LINKING THE TUTOR3.CPP PROGRAM

Modify `Project1` by removing `tutor2.c` from the project and adding `tutor3.cpp` to it.

Select **Options...** from the **Project** menu. In the **General** category, select **--cpu = 8515, AT90S8515** and the **Small** memory model.

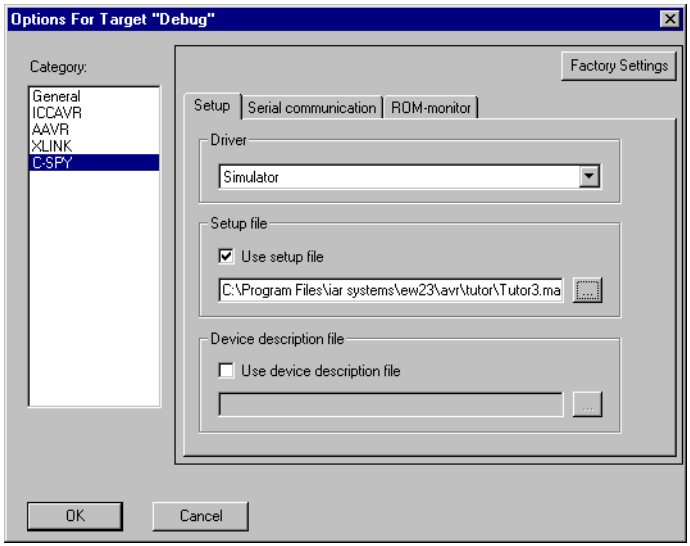
In the **ICCAVR** category, make sure that the following options are enabled:

Page	*Option*
Language	Embedded C++
	Language extensions
Code	Place string literals and constants in initialized RAM
Output	Generate debug information

Compile and link the program by choosing **Make** from the **Project** menu. Alternatively, select the **Make** button from the toolbar. The **Make** command compiles and links those files that have been modified.

RUNNING THE TUTOR3.CPP INTERRUPT PROGRAM

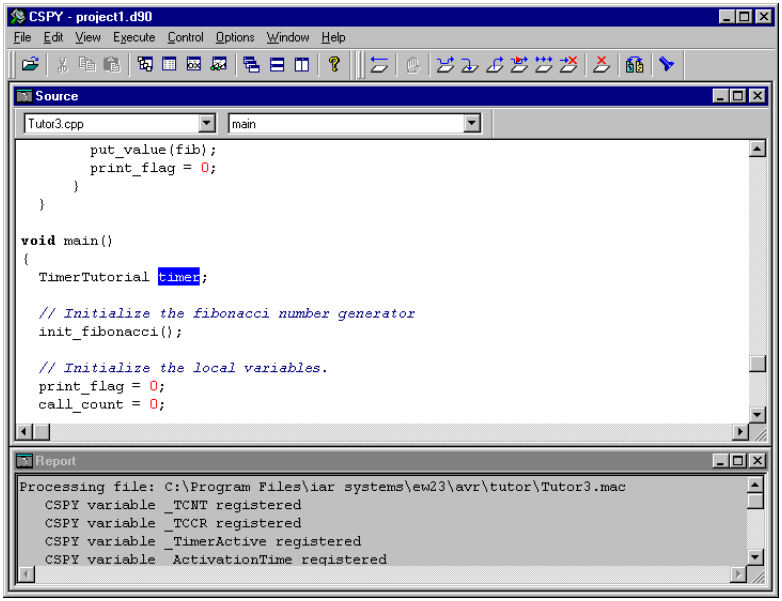
To run the `tutor3.cpp` program, we first specify the macro to be used. The macro file, `tutor3.mac`, is specified in the **C-SPY** options page in the IAR Embedded Workbench:



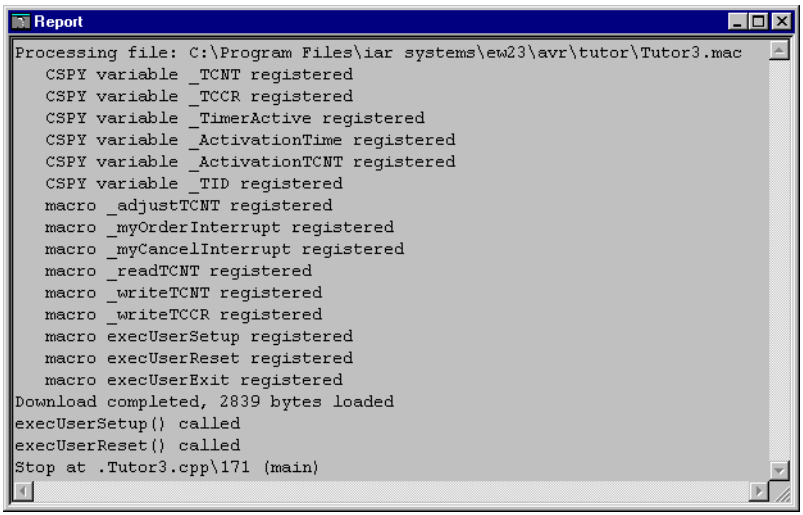
If you use the IAR C-SPY Debugger without using the IAR Embedded Workbench, the macro file can be specified via the command line option `-f` ; for additional information, see *Use setup file (-f)*, page 244.

Note: Macro files can also be loaded via the **Options** menu in the IAR C-SPY Debugger, see *Load Macro…*, page 239. Due to its contents, the `tutor3.mac` file cannot, however, be used in this manner because the `execUserSetup` macro will not be activated until you load the `project1.d90` file.

Start the IAR C-SPY Debugger by selecting **Debugger** from the **Project** menu or click the **Debugger** icon in the toolbar. The C-SPY Source window will be displayed:



The Report window will display the registered macros:



If warning or error messages should also appear in the Report window, make sure that the breakpoint has been set and that the interrupt has been registered. If not, the reason is probably that an incorrect path is specified in the `tutor3.mac` macro file.

If you need to edit the macro file, select **Load Macro...** from the **Options** menu to display the **Macro Files** dialog box. Open the `tutor3.mac` file by double-clicking on the macro name, and edit it as required. It is normally sufficient to register the macro again after saving the `mac` file.

Now you have three breakpoints connected to both timer registers in `I/O-SPACE`. To inspect the details of the breakpoint, open the **Breakpoints** dialog box by selecting **Edit Breakpoints...** from the **Control** menu. To inspect the details of the interrupt, open the **Interrupt** dialog box by selecting **Interrupt...** from the **Control** menu.

In the Source window, make sure that `tutor3.cpp` is selected in the **Source file** box. Then select the `TimerTutorial::OverflowCallback` function in the **Function** box.

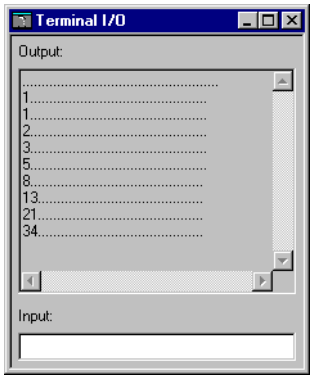
Place the cursor on the `get_fibonacci()` statement in the `TimerTutorial::OverflowCallback` function. Set a breakpoint by selecting **Toggle Breakpoint** from the **Control** menu, or clicking the **Toggle Breakpoint** button in the toolbar. Alternatively, use the pop-up menu.

Open the Terminal I/O window by selecting it from the Windows menu.

Run the program by choosing **Go** from the **Execute** menu or by clicking the **Go** button. It should stop in the interrupt function. Click **Go** again in order to see the next number being printed in the Terminal I/O window.

Since the main program has an upper limit on the Fibonacci value counter, the tutorial program will soon reach the exit label and stop.

When `tutor3` has finished running, the Terminal I/O window will display the following Fibonacci series:



ASSEMBLER TUTORIALS

These tutorials illustrate how you might use the IAR Embedded Workbench™ to develop a series of simple machine-code programs for the AVR microcontroller, and demonstrate some of the IAR Assembler's most important features:

- In Tutorial 4 we assemble and link a basic assembler program, and then run it using the IAR C-SPY® Debugger.
- Tutorial 5 demonstrates how to create library modules and use the IAR XLIB Librarian™ to maintain files of modules.

Before running these tutorials, you should be familiar with the IAR Embedded Workbench and the IAR C-SPY Debugger as described in the chapter *IAR Embedded Workbench tutorial*.

TUTORIAL 4

This assembler tutorial illustrates how to assemble and link a basic assembler program, and then run it.

CREATING A NEW PROJECT

Start the IAR Embedded Workbench and create a new project called `Project2`.

Set up the target options in the **General** category to suit the processor and memory model. In this tutorial we use the default settings. Make sure that the **Processor configuration** is set to **-v0, Max 256 byte data, 8 Kbyte code** and that the **Memory model** is set to **Tiny**.

The procedure is described in *Creating a new project*, page 29.

THE FIRST.S90 PROGRAM

The first assembler tutorial program is a simple count loop which counts up the registers `R16` and `R17` in binary-coded decimal. A copy of the program `first.s90` is provided with the product.

```
NAME  first
ORG   0
RJMP  main

ORG   1Ch
```

```
main        CLR   R17
            CLR   R16
loop        INC   R17
            CPI   R17,10
            BRNE  loop
            CLR   R17
            INC   R16
            CPI   R16,10
            BRNE  loop
done_it     RJMP  done_it

            END
```

The `ORG` directive locates the program starting address at the program reset vector address, so that the program is executed upon reset.

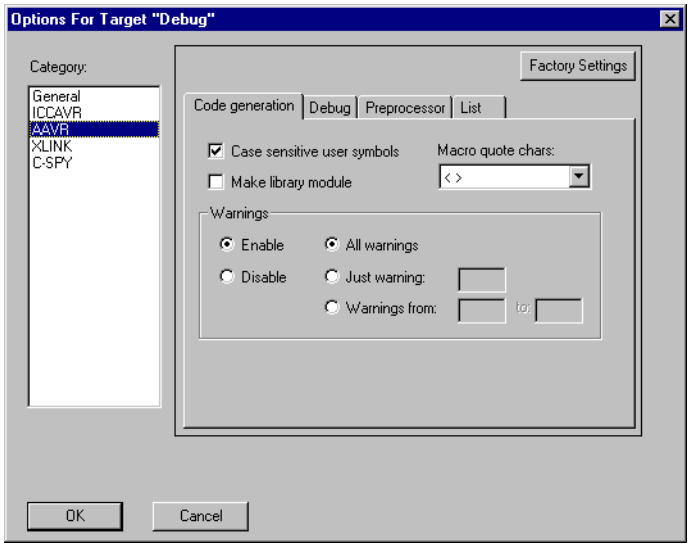
Add the program to the `Project2` project. Choose **Files...** from the **Project** menu to display the **Project Files** dialog box. Locate the file `first.s90` and click **Add** to add it to the **Common Sources** group.

You now have a source file which is ready to assemble.

ASSEMBLING THE PROGRAM

Now you should set up the assembler options for the project.

Select the **Debug** folder icon in the Project window, choose **Options...** from the **Project** menu, and select **AAVR** in the **Category** list to display the assembler options pages.

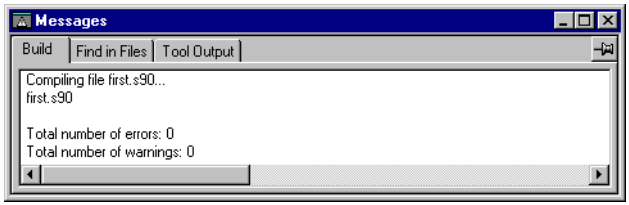


Make sure that the following options are selected on the appropriate pages of the **Options** dialog box:

Page	*Option*
Debug	Generate debug information File references
List	Create list file

Click **OK** to set the options you have specified.

To assemble the file, select it in the Project window and choose **Compile** from the **Project** menu. The progress will be displayed in the Messages window:



The listing is created in a file `first.lst` in the folder specified in the **General** options page; by default this is `Debug\list`. Open the list file by choosing **Open...** from the **File** menu, and selecting `first.lst` from the appropriate folder.

VIEWING THE FIRST.LST LIST FILE

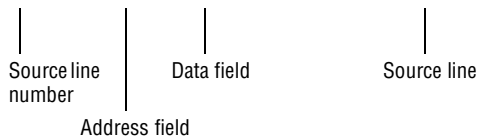
The `first.lst` list file contains the following information:

- The *header* contains product version information, the date and time when the file was created, and also specifies the options that were used.
- The *body* of the list file contains source line number, address field, data field, and source line.
- The *end* of the file contains a summary of errors and warnings that were generated, code size, and CRC.

  Note: The CRC number depends on the date of assembly, and may vary.

The format of the listing is as follows:

```
 5    0000001C                    ORG     1Ch
 6    0000001C 2711         main  CLR     R17
 7    0000001E 2700               CLR     R16
 8    00000020 9513         loop  INC     R17
 9    00000022 301A               CPI     R17,10
10    00000024 F7E9               BRNE    loop
11    00000026 2711               CLR     R17
```



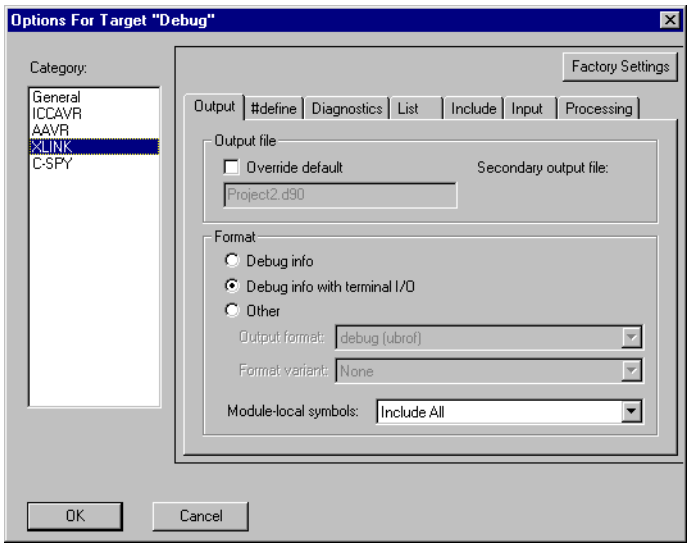
If you make any errors when writing a program, these will be displayed on the screen during the assembly and will be listed in the list file. If this happens, return to the editor by double-clicking on the error message. Check carefully through the source code to locate and correct all the mistakes, save the source file, and try assembling it again.

Assuming that the source assembled successfully, the file `first.r90`, will also be created, containing the linkable object code.

LINKING THE PROGRAM

Before linking the program you need to set up the linker options for the project.

Select the **Debug** folder in the Project window. Then choose **Options...** from the **Project** menu, and select **XLINK** in the **Category** list to display the linker option pages:

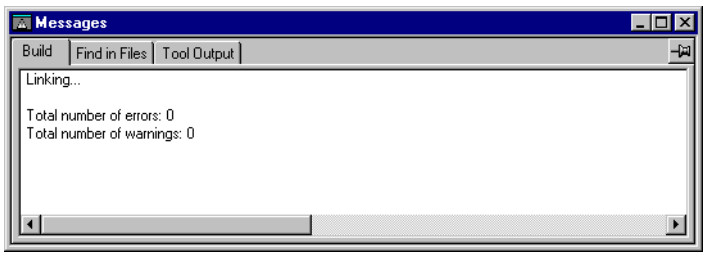


Specify the following XLINK options:

Page	*Option*
Output	Debug info with terminal I/O
Include	Ignore CSTARTUP in library

Click **OK** to set the options you have specified.

To link the file, choose **Link** from the **Project** menu. As before, the progress during linking is shown in the Messages window:



The code will be placed in a file `project2.d90`.

RUNNING THE PROGRAM

To run the example program using the IAR C-SPY Debugger, select **Debugger** from the **Project** menu.

The following warning message will be displayed in the Report window:

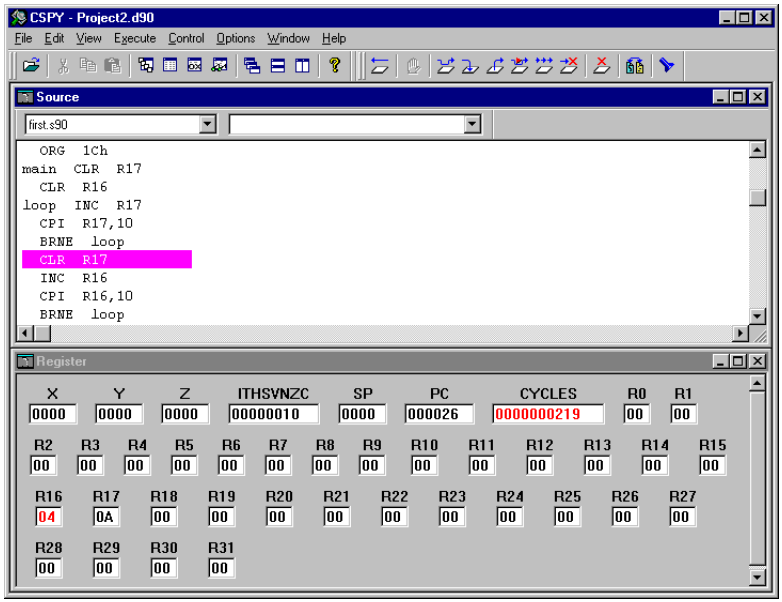
```
Warning [12]: Exit label missing
```

This message indicates that C-SPY will not know when execution of the assembler program has been completed. In a C program, this is handled automatically by the `Exit` module where the `Exit label` specifies that the program exit has been reached. Since there is no corresponding label in an assembler program, you should set a breakpoint where you want the execution of the assembler program to be completed.

In this example, set a breakpoint on the `CLR R17` instruction within the loop.

Open the Register window by selecting **Register** from the **Window** menu, or click the **Register Window** button in the toolbar. Position the windows conveniently.

Then choose **Go** from the **Execute** menu, or click the **Go** button in the debug bar. When you repeatedly click **Go**, you can watch the `R16` and `R17` registers count in binary-coded decimal format.



TUTORIAL 5

This tutorial demonstrates how to create library modules and use the IAR XLIB Librarian™ to maintain files of modules.

USING LIBRARIES

If you are working on a large project you will soon accumulate a collection of useful routines that are used by several of your programs. To avoid having to assemble a routine each time the routine is needed, you can store such routines as object files, i.e., assembled but not linked.

A collection of routines in a single object file is referred to as a *library*. It is recommended that you use library files to create collections of related routines, such as a device driver.

Use the IAR XLIB Librarian to manipulate libraries. It allows you to:

- Change modules from `PROGRAM` to `LIBRARY` type, and vice versa.
- Add or remove modules from a library file.
- Change the names of entries.
- List module names, entry names, etc.

THE MAIN.S90 PROGRAM

The following listing shows the `main.s90` program. A copy of the program is provided with the product.

```
            NAME        main

            PUBLIC      main
            EXTERN      r_shift

            RSEG        MY_CODE
main        LDI         R25,H'A
            MOV         R4,R25
            LDI         R25,5
            MOV         R5,R25
            RCALL       r_shift
done_it     RJMP        done_it

            END         main
```

This simply uses a routine called `r_shift` to shift the contents of register `R4` to the right. The data in register `R4` is set to `$A` and the `r_shift` routine is called to shift it to the right by four places as specified by the contents of register `R5`.

The `EXTERN` directive declares `r_shift` as an external symbol, to be resolved at link time.

THE LIBRARY ROUTINES

The following two library routines will form a separately assembled library. It consists of the `r_shift` routine called by `main`, and a corresponding `l_shift` routine, both of which operate on the contents of the register R4 by repeatedly shifting it to the right or left. The number of shifts performed is controlled by decrementing register R5 to zero. The file containing these library routines is called `shifts.s90`, and a copy is provided with the product.

```
          MODULE    r_shift
          PUBLIC    r_shift
          RSEG      MY_CODE

r_shift   TST    R5
          BREQ   r_shift2
          LSR    R4
```

```
          DEC     R5
          BRNE    r_shift
r_shift2  RET
          ENDMOD

          MODULE l_shift
          PUBLIC l_shift

          RSEG   MY_CODE
l_shift   TST    R5
          BREQ   l_shift2
          LSL    R4
          DEC    R5
          BRNE   l_shift
l_shift2  RET

          END
```

The routines are defined as library modules by the `MODULE` directive, which instructs the IAR XLINK Linker™ to include the modules only if they are called by another module.

The `PUBLIC` directive makes the `r_shift` and `l_shift` entry addresses public to other modules.

For detailed information about the `MODULE` and `PUBLIC` directives, see the *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide*.

CREATING A NEW PROJECT

Create a new project called `Project3`. Add the files `main.s90` and `shifts.s90` to the new project.

Then set up the target options to suit the project. Make sure that the **Processor variant** is set to **-v0, Max 256 byte data, 8 Kbyte code** and that the **Memory model** is set to **Tiny**.

The procedure is described in *Creating a new project*, page 29.

ASSEMBLING AND LINKING THE SOURCE FILES

To assemble and link the `main.s90` and `shifts.s90` source files, you must disable the `CSTARTUP` initialization module in the default run-time library.

Open the **Options** dialog box by selecting **Options...** from the **Project** menu. Select **XLINK** in the **Category** list and set the following option:

Page	*Option*
Include	Ignore CSTARTUP in library

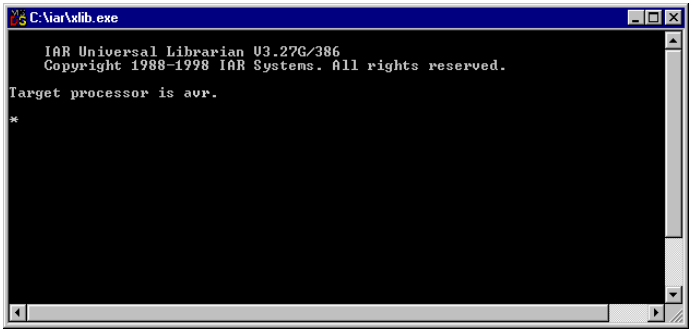
To assemble and link the `main.s90` and the `shifts.s90` files , select **Make** from the **Project** menu. Alternatively, select the **Make** button in the toolbar.

For more information about the XLINK options see the *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide*.

USING THE IAR XLIB LIBRARIAN

Once you have assembled and debugged modules intended for general use, like the `r_shift` and `l_shift` modules, you can add them to a library using the IAR XLIB Librarian.

Run the IAR XLIB Librarian by choosing **Librarian** from the **Project** menu. The XLIB window will be displayed:



You can now enter XLIB commands at the * prompt.

Giving XLIB options

Extract the modules you want from `shifts.r90` into a library called `math.r90`. To do this enter the command:

```
FETCH-MODULES
```

The IAR XLIB Librarian will prompt you for the following information:

Prompt	*Response*
Source file	Type `debug\obj\shifts` and press Enter.
Destination file	Type `debug\obj\math` and press Enter.
Start module	Press Enter to use the default start module, which is the first in the file.
End module	Press Enter to use the default end module, which is the last in the file.

This creates the file `math.r90` which contains the code for the `r_shift` and `l_shift` routines.

You can confirm this by typing:

```
LIST-MODULES
```

The IAR XLIB Librarian will prompt you for the following information:

Prompt	*Response*
Object file	Type `math` and press Enter.
List file	Press Enter to display the list file on the screen.
Start module	Press Enter to start from the first module.
End module	Press Enter to end at the last module.

You could use the same procedure to add further modules to the `math` library at any time.

Finally, leave the librarian by typing:

```
EXIT
```

or

```
QUIT
```

Then press Enter.

ADVANCED TUTORIALS

This chapter describes some of the more advanded features of the IAR development tools, which are very useful when you work on larger projects.

- The first two tutorials below both explore the features of C-SPY. In Tutorial 6 we define complex breakpoints, profile the application, and display code coverage. Tutorial 7 describes how to debug in disassembly mode.
- Tutorial 8 describes how to create a project containing both C or Embedded C++ and assembly language source files.

Before running these tutorials, you should be familiar with the IAR Embedded Workbench and the IAR C-SPY Debugger as described in the chapter *IAR Embedded Workbench tutorial*.

TUTORIAL 6

In this tutorial we explore the following features of C-SPY:

- Defining complex breakpoints
- Profiling the application
- Displaying code coverage information.

CREATING PROJECT4

In the IAR Embedded Workbench, create a new project called `Project4` and add the files `tutor.c` and `common.c` to it. Make sure that the following project options are set:

Category	*Page*	*Option*
General	Target	Processor configuration: -v0, Max 256 byte data, 8 Kbyte code (default) Memory model: Tiny (default).
ICCAVR	Optimizations	Size optimization: Low (default).
	Output	Generate debug information (default).
	List	C list file.
XLINK	Output	Debug info with terminal I/O (default).

Click **OK** to set the options.

Select **Make** from the **Project** menu, or click the **Make** button in the toolbar to compile and link the files. This creates the `project4.d90` file.

Start C-SPY to run the `project4.d90` program.

DEFINING COMPLEX BREAKPOINTS

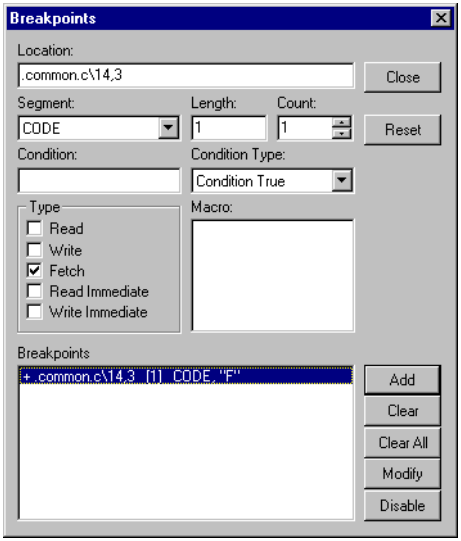
You can define complex breakpoint conditions in C-SPY, allowing you to detect when your program has reached a particular state of interest.

The file `project4.d90` should now be open in C-SPY. Execute one step. The current position should be the call to the `init_fibonacci` function.

Then select **Step into** to move to the `init_fibonacci` function. Set a breakpoint at the statement `++i`.

Now we will modify the breakpoint you have set so that C-SPY detects when the value of `i` exceeds 8.

Choose **Edit Breakpoints...** from the **Control** menu to display the **Breakpoints** dialog box. Then select the breakpoint in the **Breakpoints** list to display information about the breakpoint you have defined:

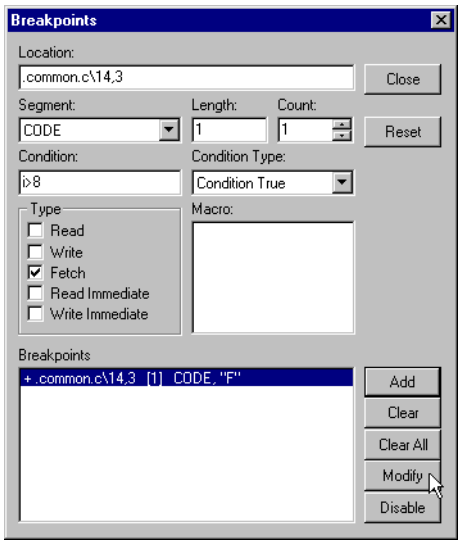


Currently the breakpoint is triggered when a fetch occurs from the location corresponding to the C or Embedded C++ statement.

Add a condition to the breakpoint using the following procedure:

Enter `i>8` in the **Condition** box and, if necessary, select **Condition True** from the **Condition Type** drop-down list.

Then choose **Modify** to modify the breakpoint with the settings you have defined:



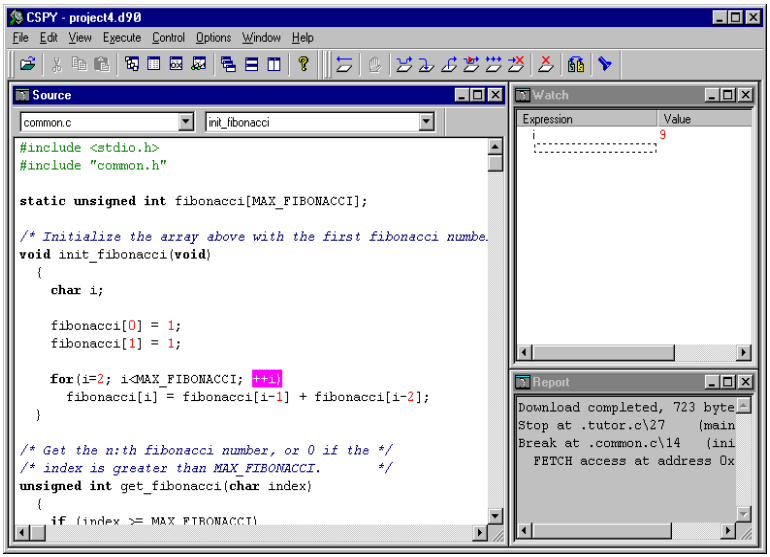
Finally, select **Close** to close the **Breakpoints** dialog box.

Open the Watch window and add the variable `i`. The procedure is described in *Watching variables*, page 44.

Position the Source, Watch, and Report windows conveniently.

EXECUTING UNTIL A CONDITION IS TRUE

Now execute the program until the breakpoint condition is true by choosing **Go** from the **Execute** menu, or clicking the **Go** button in the toolbar. The program will stop when it reaches a breakpoint and the value of `i` exceeds 8:



EXECUTING UP TO THE CURSOR

A convenient way of executing up to a particular statement in the program is to use the **Go to Cursor** option.

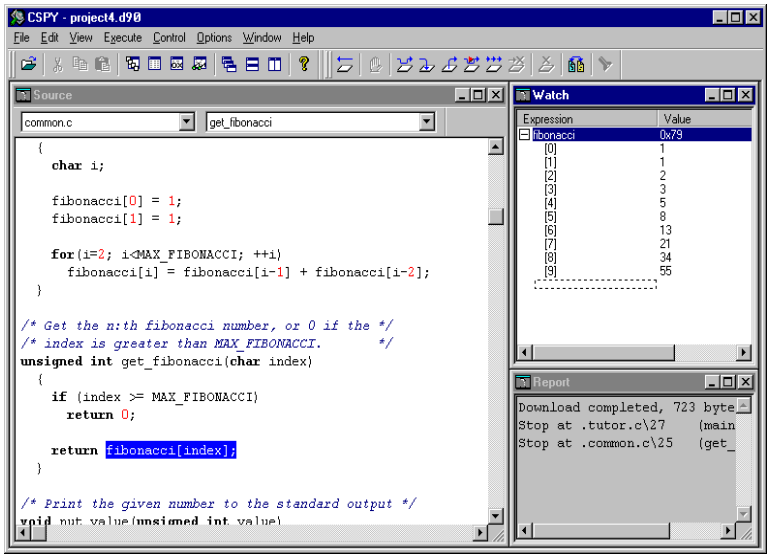
First remove the existing breakpoint. Use the **Edit Breakpoints...** command from the **Control** menu or from the pop-up menu to open the **Breakpoints** dialog box. Select the breakpoint and click on the **Clear** button.

Then remove the variable `i` from the Watch window. Select the variable in the Watch window and press the Delete key. Instead add `fibonacci` to watch the array during execution.

Position the cursor in the Source window in the statement:

```
return fibonacci[index];
```

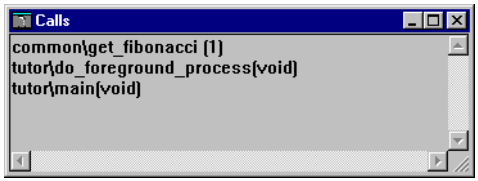
Select **Go to Cursor** from the **Execute** menu, or click the **Go to Cursor** button in the toolbar. The program will then execute up to the statement at the cursor position. Expand the contents of the `fibonacci` array to view the result:



DISPLAYING FUNCTION CALLS

The program is now executing statements inside a function called from `main`. You can display the sequence of calls to the current position in the Calls window.

Choose **Calls** from the **Window** menu to open the Calls window and display the function calls. Alternatively, click the **Calls Window** button in the toolbar.



In each case the function name is preceded by the module name.

You can now close both the Calls window and the Watch window.

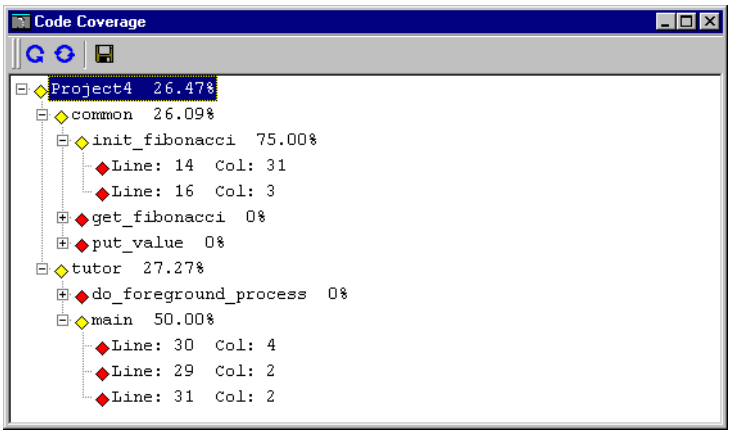
DISPLAYING CODE COVERAGE INFORMATION

The code coverage tool can be used for identifying statements not executed and functions not called in your program.

Reset the program by selecting **Reset** from the **Execute** menu or by clicking the **Reset** button in the toolbar. Display the current code coverage status by selecting **Code Coverage** from the **Window** menu. The information shows that no functions have been called.

Select the **Auto Refresh On/Off** button in the toolbar of the Code Coverage window. The information displayed in the Code Coverage window will automatically be updated.

Execute one step, and then select **Step Into** to step into the `init_fibonacci` function. Execute a few more steps and look at the code coverage status once more. At this point a few statements are reported as not executed:



For additional information about the layout of the Code Coverage window, see *Code coverage window*, page 217.

PROFILING THE APPLICATION

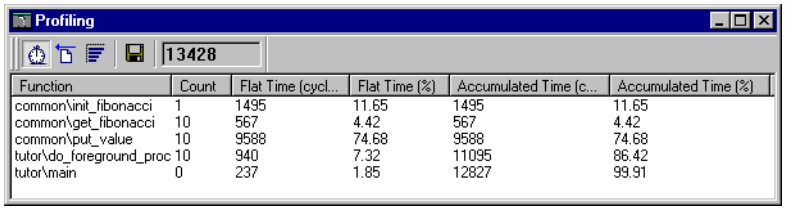
The profiling tool provides you with timing information on your application.

Reset the program by selecting **Reset** from the **Execute** menu or by clicking the **Reset** button in the toolbar. Open a Profiling window by choosing **Profiling** from the **Window** menu.

Start the profiling tool by selecting **Profiling** from the **Control** menu or by clicking the **Profiling On/Off** button in the **Profiling** toolbar.

Clear all breakpoints by selecting **Clear All** in the **Breakpoints** dialog box, which is displayed when you select **Edit Breakpoints...** from the **Control** menu. Run the program by clicking the **Go** button in the toolbar.

When the program has reached the exit point, you can study the profiling information shown in the Profiling window:



The Profiling window contains the following information:

- **Count** is the number of times each function has been called.
- **Flat Time** is the total time spent in each function in cycles or as a percentage of the total number of cycles shown in the **Profiling** toolbar.
- **Accumulated Time** is time spent in each function including all function calls made from that function in cycles or as a percentage of the total number of cycles.

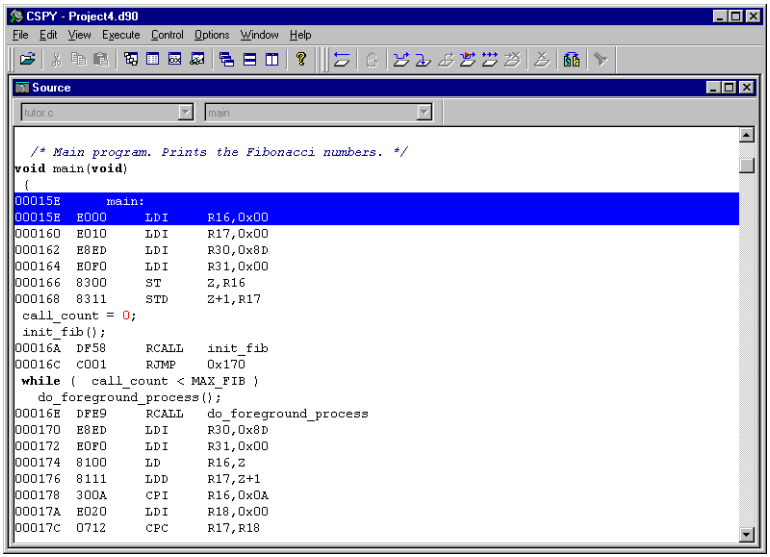
From the **Profiling** toolbar it is possible to display the profiling information graphically, to save the information to a file, or to start a new measurement. For additional information, see *Profiling window*, page 218.

TUTORIAL 7

Although debugging with C-SPY is usually quicker and more straightforward in source mode, some demanding applications can only be debugged in assembler mode. C-SPY lets you switch freely between the two.

First reset the program by clicking the **Reset** button in the toolbar. Then change the mode by choosing **Toggle Source/Disassembly** from the **View** menu or click the **Toggle Source/Disassembly** button in the toolbar.

You will see the assembler code corresponding to the current C statement. Stepping is now one assembler instruction at a time.



When you are debugging in disassembly mode, every assembler instruction that has been executed since the last reset is marked with an * (asterisk).

Note: There may be a delay before this information is displayed, due to the way the Source window is updated.

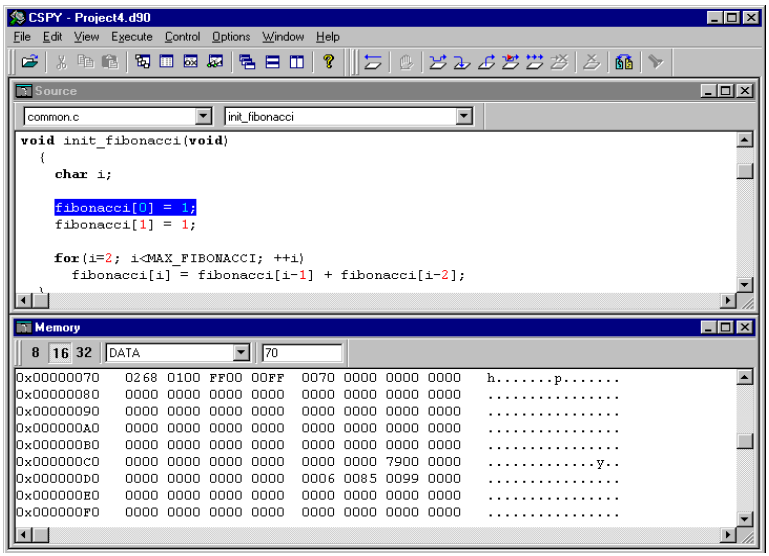
MONITORING MEMORY

The Memory window allows you to monitor selected areas of memory. In the following example we will monitor the memory corresponding to the variable `fibonacci`.

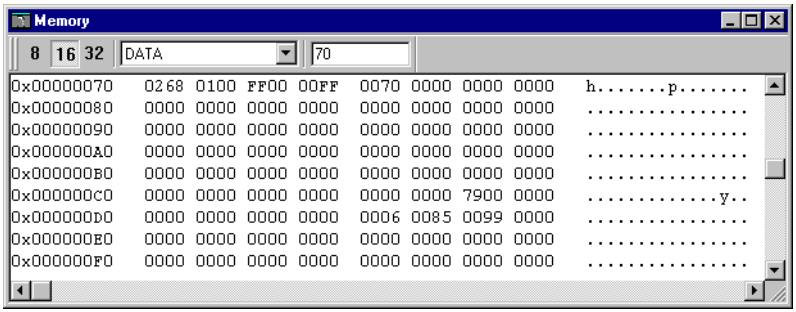
Choose **Memory** from the **Window** menu to open the Memory window or click the **Memory Window** button in the toolbar. Position the Source and Memory windows conveniently on the screen.

Change back to source mode by choosing **Toggle Source/Disassembly** or clicking the **Toggle Source/Disassembly** button in the toolbar.

Select `fibonacci` in the file `common.c`. Then drag it from the Source window and drop it into the Memory window. The Memory window will show the contents of memory corresponding to `fibonacci`:



Since we are displaying 16-bit `word` data, it is convenient to display the memory contents as long words. Click the **16** button in the Memory window toolbar:



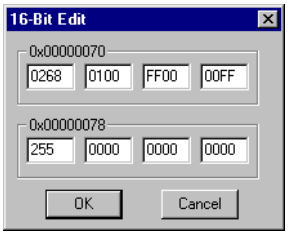
Notice that the 10 words have been initialized by the `init_fibonacci` function of the C program.

CHANGING MEMORY

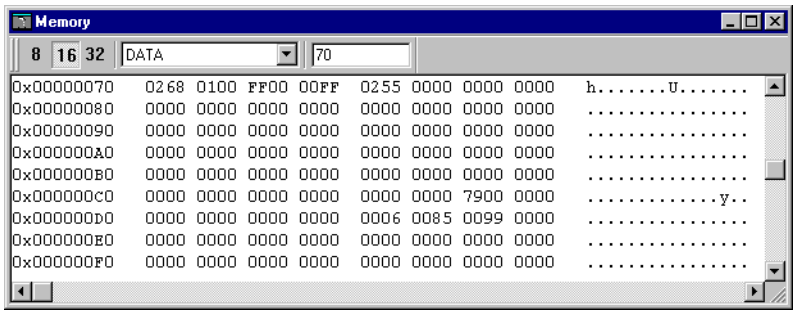
You can change the memory contents by editing the values in the Memory window. Double-click the line in memory which you want to edit. A dialog box is displayed.

You can now edit the corresponding values directly in the memory.

For example, if you want to write the number `0x255` in the first position in number in the `fibonacci` array, select the long word at address `0x78` in the Memory window and type `255` in the **16-Bit Edit** dialog box:



Then choose **OK** to display the new values in the Memory window:

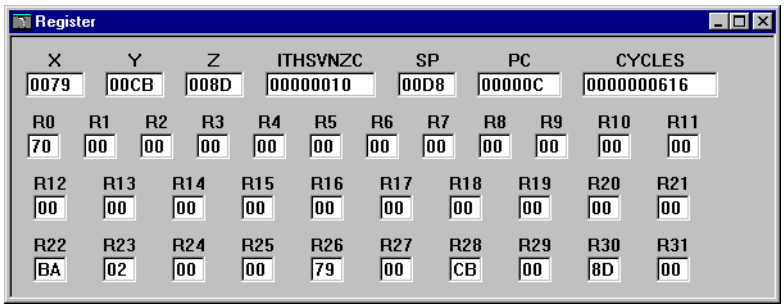


Before proceeding, close the Memory window and switch to disassembly mode.

MONITORING REGISTERS

The Register window allows you to monitor the contents of the processor registers and modify their contents.

Open the Register window by choosing **Register** from the **Window** menu. Alternatively, click the **Register Window** button in the toolbar.



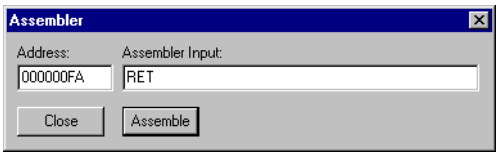
Select **Step** from the **Execute** menu, or click the **Step** button in the toolbar, to execute the next instructions, and watch how the values change in the Register window.

Then close the Register window.

CHANGING ASSEMBLER VALUES

C-SPY allows you to temporarily change and reassemble individual assembler statements during debugging.

Select disassembly mode and step towards the end of the program. Position the cursor on a `RET` instruction and double-click on it. The **Assembler** dialog box is displayed:



Change the **Assembler Input** field from `RET` to `NOP` and select **Assemble** to temporarily change the value of the statement. Notice how it changes also in the Source window.

TUTORIAL 8

CREATING A COMBINED COMPILER AND ASSEMBLER PROJECT

In large projects it may be convenient to use source files written in both C or Embedded C++ and assembly language. In this tutorial we will demonstrate how they can be combined by substituting the file `common.c` with the assembler file `common.s90` and compiling the project.

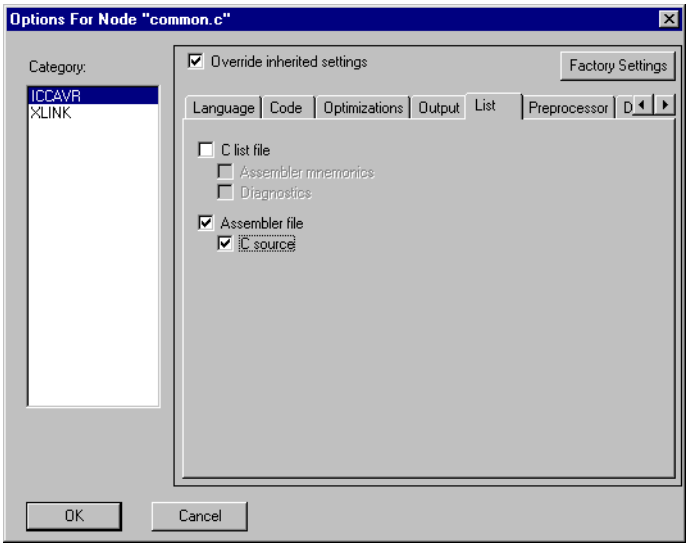
Return to or open `Project4` in the IAR Embedded Workbench. The project should contain the files `tutor.c` and `common.c`.

Now you should create the assembler file `common.s90`. In the Project window, select the file `common.c`. Then select **Options...** from the **Project** menu. You will notice that only the **ICCAVR** and **XLINK** categories are available.

In the **ICCAVR** category, select **Override inherited settings** and set the following options:

Page	*Option*
List	Deselect C list file.
	Select Assembler file.

Page	*Option*
	Select the suboption C source.



Then click **OK** and return to the Project window.

Compile each of the files. To see how the C or Embedded C++ code is represented in assembly language, open the file `common.s90` that was created from the file `common.c`.

Now modify `Project4` by removing the file `common.c` and adding the file `common.s90` instead. Then select **Make** from the **Project** menu to relink `Project4`.

Start C-SPY to run the `project4.d90` program and see that it behaves like in the previous tutorials.

PART 3: THE IAR EMBEDDED WORKBENCH

This part of the AVR IAR Embedded Workbench™ User Guide contains the following chapters:

- *General options*
- *Compiler options*
- *Assembler options*
- *XLINK options*
- *C-SPY options*
- *IAR Embedded Workbench reference.*

GENERAL OPTIONS

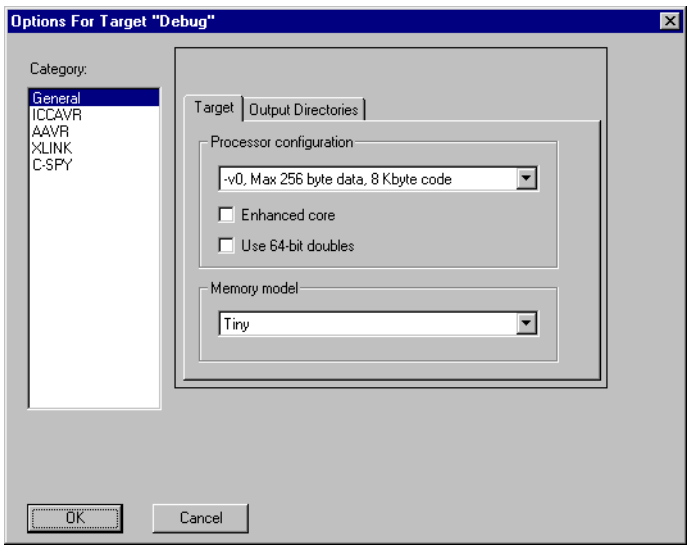
The general options specify the target processor and memory model, as well as output directories.

This chapter describes how to set general options in the IAR Embedded Workbench™ and gives full reference information about the options.

The options are divided into the following sections:

SETTING GENERAL OPTIONS

To set general options in the IAR Embedded Workbench choose **Options...** from the **Project** menu. The **Target** page in the **General** category is displayed:

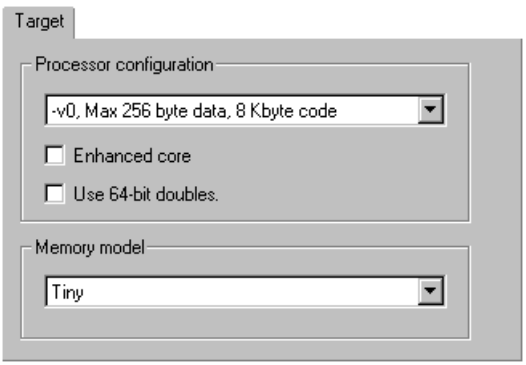


The general options are grouped into categories, and each category is displayed on an option page in the IAR Embedded Workbench.

Click the tab corresponding to the category of options that you want to view or change.

TARGET

The **Target** options in the **General** category specify processor configuration and memory model for the AVR IAR Compiler and Assembler.



PROCESSOR CONFIGURATION

Use this option to select your target processor and the maximum module and program size.

Select the target processor for your project from the drop-down list.

For a description of the available options, see the *Configuration* chapter in the *AVR IAR Compiler Reference Guide*.

Your choice of processor configuration determines the availability of memory model options.

Enhanced core
Use this option to allow the compiler to generate instructions from the enhanced instruction set that is available in some AVR derivatives, for example AT90mega161.

Use 64-bit doubles
Use this option to force the compiler to use 64-bit doubles instead of 32-bit doubles, which is the default.

MEMORY MODEL

Use this option to select the memory model for your project.

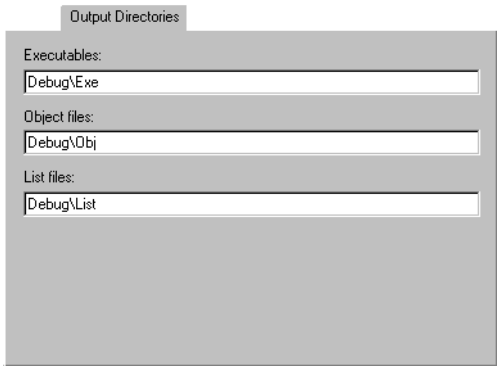
Select the memory model for your project from the drop-down list.

Your choice of processor configuration determines the availability of memory model options.

For a description of the available options, see the *Configuration* chapter in the *AVR IAR Compiler Reference Guide*.

OUTPUT DIRECTORIES

The **Output directories** options allow you to specify directories for executable files, object files, and list files. Notice that incomplete paths are relative to your project directory.



Executables
Use this option to override the default directory for executable files.

Enter the name of the directory where you want to save executable files for the project.

Object files
Use this option to override the default directory for object files.

Enter the name of the directory where you want to save object files for the project.

List files
Use this option to override the default directory for list files.

Enter the name of the directory where you want to save list files for the project.

COMPILER OPTIONS

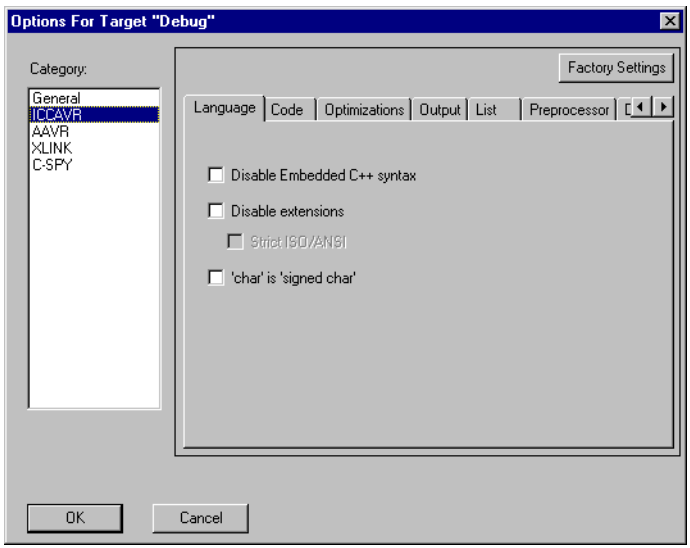
This chapter explains how to set compiler options from the IAR Embedded Workbench™, and describes each option.

The options are divided into the following sections:

SETTING COMPILER OPTIONS

SETTING COMPILER OPTIONS

To set compiler options in the IAR Embedded Workbench, select **Options...** from the **Project** menu to display the **Options** dialog box. Select **ICCAVR** in the **Category** list to display the compiler options pages:



Click the tab corresponding to the type of options that you want to view or change.

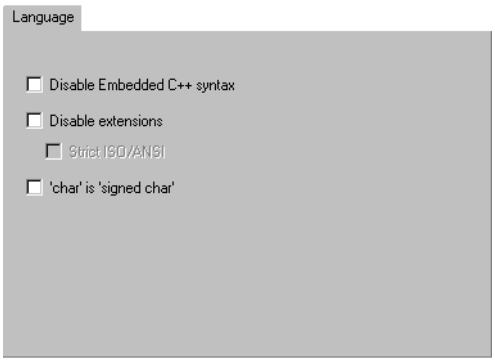
Notice that compiler options can be specified on a target level, group level, or file level. When options are set on the group or file level, you can choose to override settings inherited from a higher level.

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

The following sections give full descriptions of each compiler option.

LANGUAGE

The **Language** options enable the use of target-dependent extensions to the C or Embedded C++ language.



DISABLE EMBEDDED C++ SYNTAX

In Embedded C++ mode, the compiler treats the source code as Embedded C++. Unless Embedded C++ is enabled, the compiler runs in ANSI C mode, in which features specific to Embedded C++, such as classes and overloading, cannot be utilized.

In the IAR Embedded Workbench, Embedded C++ syntax is enabled by default. Use the **Disable Embedded C++ syntax** check box to disable Embedded C++.

DISABLE EXTENSIONS

Language extensions must be enabled for the AVR IAR Compiler to be able to accept AVR-specific keywords as extensions to the standard C or Embedded C++ language.

In the IAR Embedded Workbench, language extensions are enabled by default. Use the **Disable extensions** check box to disable language extensions.

For details about language extensions, see the *AVR IAR Compiler Reference Guide.*

Strict ISO/ANSI
By default the compiler accepts a superset of ISO/ANSI C (for additional information, see the *AVR IAR Compiler Reference Guide*). Use this option to adhere to strict ISO/ANSI.

First select **Disable extensions**, and then select **Strict ISO/ANSI** to adhere to the strict ISO/ANSI C standard.

'CHAR' IS 'SIGNED CHAR'

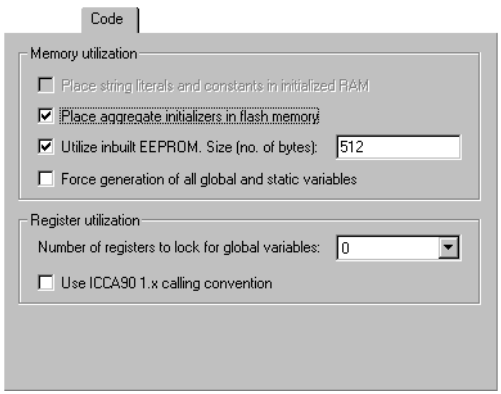
Normally, the compiler interprets the `char` type as `unsigned char`. Use this option to make the compiler interpret the `char` type as `signed char` instead, for example for compatibility with another compiler.

Select **'char' is 'signed char'** to make the compiler interpret the `char` type as `signed char`.

Note: The run-time library is compiled without the **'char' is 'signed char'** option. If you use this option, you may get type mismatch warnings from the linker since the library uses `unsigned char`.

CODE

The **Code** options determine the usage of segments and registers.



Notice that the target options you select determine which code options are available.

MEMORY UTILIZATION

Place string literals and constants in initialized RAM
Use this option to override the default placement of constants and literals.

Without this option, constans and literals are placed in an external `const` segment, *`segment`*`_C`. *With* this option, constants and literals will instead be placed in the initialized *`segment`*`_I` data segments that are copied from *`segment`*`_ID` by `cstartup`.

For reference information about segments, see the *AVR IAR Compiler Reference Guide*.

Notice that this option is implicit in the tiny memory model.

Place aggregate initializers in flash memory
Use this option to place aggregate initializers in flash memory. These initializers are otherwise placed either in the external `const` segment or in the initialized data segments if the compiler option **Place string literals and constants in initialized RAM** was also specified.

For reference information about segments, see the *AVR IAR Compiler Reference Guide*.

Utilize inbuilt EEPROM
Use this option to enable the `__eeprom` extended keyword by specifying the size of the inbuilt `EEPROM`. The size in bytes can be 0–65536.

Force generation of all global and static variables
Use this option to apply the `__root` extended keyword to all global and static variables. This will make sure that the variables are not removed by the IAR XLINK Linker.

Notice that the `__root` extended keyword is always available, even if language extensions are disabled.

For reference information about extended keywords, see the *AVR IAR Compiler Reference Guide*.

Force generation of all global and static variables
Use this option to apply the `__root` extended keyword to all global and static variables. This will make sure that the variables are not removed by the IAR XLINK Linker.

REGISTER UTILIZATION

Number of registers to lock for global variables
Use this option to lock registers that are to be used for global register variables. The value can be `0–12` where `0` means that no registers are locked. When you use this option, the registers `R15` and downwards will be locked.

In order to maintain module consistency, make sure to lock the same number of registers in all modules.

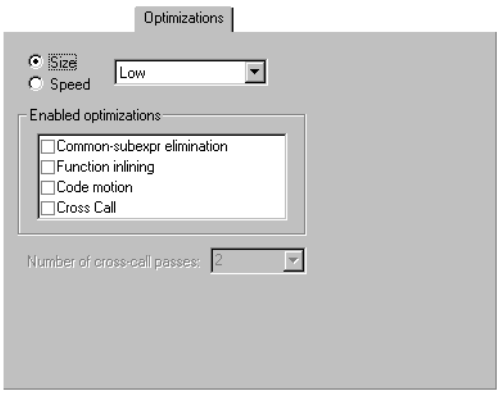
Use ICCA90 1.x calling convention
This option is provided for backward compatibility. It makes all functions and function calls use the calling convention of the A90 IAR Compiler, `ICCA90`.

To change the calling convention of a single function, use the extended keyword `__version_1` as a function type attribute.

For detailed information about calling conventions and extended keywords, see the *AVR IAR Compiler Reference Guide*.

OPTIMIZATIONS

The **Optimizations** options determine the type and level of optimization for generation of object code.



Size and speed

The AVR IAR Compiler supports two optimization models—size and speed—at different optimization levels. Code can also be generated without any optimization.

Select the optimization model using either the **Size** or **Speed** radio button. Then select the optimization level—none, low, medium, or high—from the drop-down list.

By default, a debug project will have a size optimization that is fully debuggable, while a release project will have a size optimization that generates minimum code.

The following table describes the optimization levels:

Option	*Description*
None	No optimization
Low	Fully debuggable
Medium	Heavy optimization can make the program flow hard to follow during debug
High	Full optimization

ENABLED TRANSFORMATIONS

The AVR IAR Compiler supports the following types of transformations:

- Common sub-expression elimination
- Function inlining
- Code motion
- Cross call.

In a *debug* project, the transformations are by default disabled. You can enable a transformation by selecting its check box. The compiler will then determine if this transformation is feasible.

In a *release* project, the transformations are by default enabled. You can disable a transformation by deselecting its check box.

Common sub-expression elimination
Redundant re-evaluation of common sub-expressions is by default eliminated at optimization levels Medium and High. This optimization normally reduces both code size and execution time. The resulting code may however be difficult to debug.

Note: This option has no effect at optimization levels None and Low.

Function inlining
Function inlining means that a simple function, whose definition is known at compile time, is integrated into the body of its caller to eliminate the overhead of the call. This optimization, which is performed at optimization level High, normally reduces execution time, but increases code size. The resulting code may also be difficult to debug.

The compiler decides which functions to inline. Different heuristics are used when optimizing for speed.

Note: This option has no effect at optimization levels None, Low, and Medium.

Code motion
Evaluation of loop-invariant expressions and common sub-expressions are moved to avoid redundant reevaluation. This optimization, which is performed at optimization level High, normally reduces code size and execution time. The resulting code may however be difficult to debug.

Note: This option has no effect at optimization levels None or Low.

Cross call
This optimization creates subroutines of common code sequences. It is performed as a size optimization at level High.

Notice that, although it can drastically reduce the code size, this option increases the execution time as well as the internal return data stack, RSTACK, usage.

Avoid using this option if your target processor has a hardware stack or a small RAM-based internal return stack segment, RSTACK.

When selecting this option, you must also specify the number of cross call passes to run by using the option **Number of cross-call passes**.

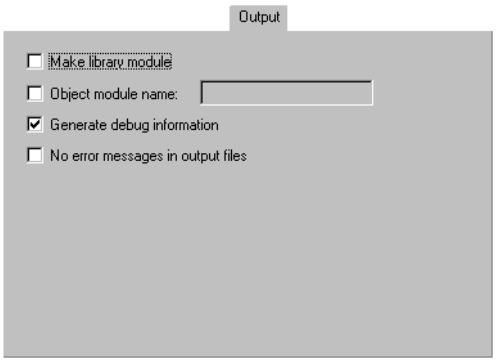
NUMBER OF CROSS-CALL PASSES

Use this option to decrease the RSTACK usage by running the cross-call optimizer *N* times, where *N* can be 1–5. The default is to run it twice.

This option becomes available when you select the size optimization at level High and enable the cross-call optimization.

OUTPUT

The **Output** options determine the output format of the compiled file, including the level of debugging information in the object code.



MAKE LIBRARY MODULE

By default the compiler generates *program* modules, which are always included during linking. Use this option to make a *library* module that will only be included if it is referenced in your program.

Select the **Make library module** option to make the object file be treated as a library module rather than as a program module.

For information about working with libraries, see the IAR XLIB Librarian chapters in the *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide*.

OBJECT MODULE NAME

Normally, the internal name of the object module is the name of the source file, without a directory name or extension. Use this option to set the object module name explicitly.

First select the **Object module name** check box, then enter a name in the entry field.

This option is particularly useful when several modules have the same filename, since the resulting duplicate module name would normally cause a linker error; for example, when the source file is a temporary file generated by a preprocessor.

GENERATE DEBUG INFORMATION

This option causes the compiler to include additional information in the object modules that is required by C-SPY® and other symbolic debuggers.

The **Generate debug information** option is specified by default. Deselect this option if you do not wish the compiler to generate debug information.

Note: The included debug information increases the size of the object files.

NO ERROR MESSAGES IN OUTPUT FILES

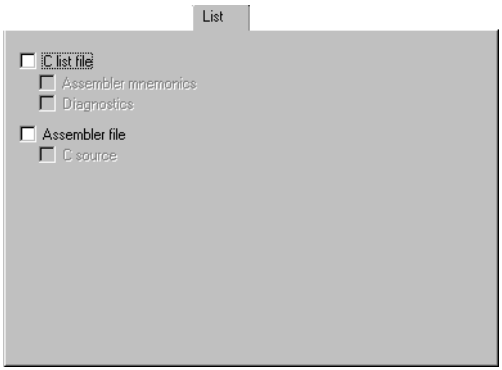
Use this option to minimize the size of your application object file by excluding messages from the UBROF files. A file size decrease of up to 60 % can be expected. The XLINK diagnostic messages will, however, be less useful when you use this option.

Notice that this option does not affect the code generation; it does not perform any optimizations and reduces the object file size only by excluding information.

For reference information about the XLINK output formats, see the *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide*.

LIST

The **List** options determine whether a list file is produced, and the information included in the list file.



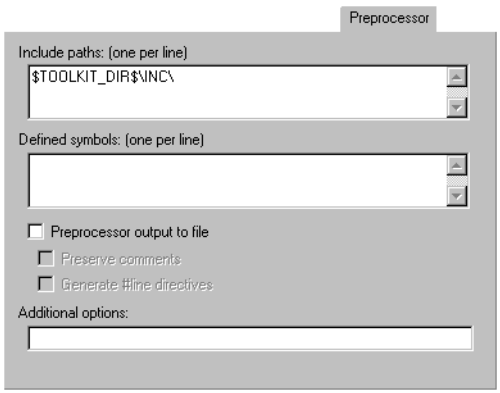
Normally, the compiler does not generate a list file. Select one of the following options to generate a list file:

Option	*Description*
C list file	Generates a C or Embedded C++ list file
Assembler mnemonics	Includes assembler mnemonics in the C or Embedded C++ list file
Diagnostics	Includes diagnostic information in the C or Embedded C++ list file
Assembler file	Generates an assembler list file
C source	Includes C or Embedded C++ source code in the assembler list file

The list file will be saved in the source file directory, and its filename will consist of the source filename, plus the filename extension `lst`.

PREPROCESSOR

The **Preprocessor** options allow you to define symbols and include paths for use by the compiler.



INCLUDE PATHS

Adds a path to the list of `#include` file path.

Enter the full file path of your `#include` files.

To make your project more portable, use the argument variable `$TOOLKIT_DIR$\inc\` for the `inc` subdirectory of the active product (that is, standard system `#include` files) and `$PROJ_DIR$\inc\` for the `inc` subdirectory of the current project directory. For an overview of the argument variables, see page 161.

DEFINED SYMBOLS

The **Defined symbols** option is useful for conveniently specifying a value or choice that would otherwise be specified in the source file.

Enter the symbols that you want to define for the project.

This option has the same effect as a `#define` statement at the top of the source file.

For example, you could arrange your source to produce either the test or production version of your program 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.

PREPROCESSOR OUTPUT TO FILE

By default the compiler does not generate preprocessor output.

Select the **Preprocessor output to file** option if you want to generate preprocessor output. You can also choose to preserve comments and/or to generate `#line` directives.

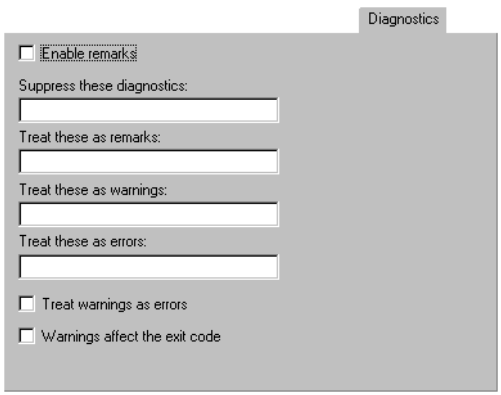
ADDITIONAL OPTIONS

Use this field to enter any other command line compiler option for your project. In particular, rarely used options such as `--segment` or `--no_rampd` can be entered here. The syntax for command line options is described in the *AVR IAR Compiler Reference Guide*.

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.



ENABLE REMARKS

The least severe diagnostic messages are called *remarks*. A remark indicates a source code construct that may cause strange behavior in the generated code.

By default remarks are not issued. Select the **Enable remarks** option if you want the compiler to generate remarks.

SUPPRESS THESE DIAGNOSTICS

This option suppresses the output of diagnostics for the tags that you specify.

For example, to suppress the warnings `Pe117` and `Pe177`, type:

```
Pe117,Pe177
```

TREAT THESE AS REMARKS

A remark is the least severe type of diagnostic message. It indicates a source code construct that may cause strange behavior in the generated code. Use this option to classify diagnostics as remarks.

For example, to classify the warning `Pe177` as a remark, type:

```
Pe177
```

TREAT THESE 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. Use this option to classify diagnostic messages as warnings.

For example, to classify the remark `Pe826` as a warning, type:

```
Pe826
```

TREAT THESE AS ERRORS

An *error* indicates a violation of the C or Embedded C++ language rules, of such severity that object code will not be generated, and the exit code will not be 0. Use this option to classify diagnostic messages as errors.

For example, to classify the warning `Pe117` as an error, type:

```
Pe117
```

TREAT WARNINGS AS ERRORS

Use this option to make the compiler treat all warnings as errors. If the compiler encounters an error, object code is not generated.

WARNINGS AFFECT THE EXIT CODE

By default the exit code is not affected by warnings, only errors produce a non-zero exit code. With this option, warnings will generate a non-zero exit code.

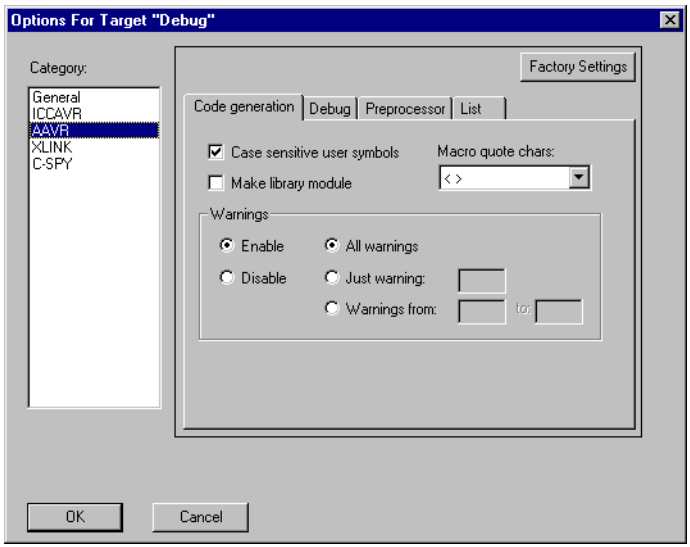
ASSEMBLER OPTIONS

This chapter first explains how to set the options from the IAR Embedded Workbench™. It then provides complete reference information for each assembler option.

The options are divided into the following sections:

SETTING ASSEMBLER OPTIONS

To set assembler options in the IAR Embedded Workbench, choose **Options...** from the **Project** menu to display the **Options** dialog box. Then select **AAVR** in the **Category** list to display the assembler options pages:



Click the tab corresponding to the type of options you want to view or change.

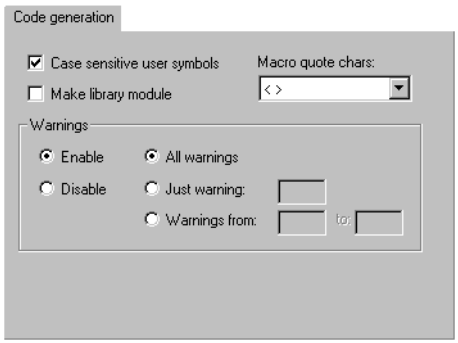
Notice that assembler options can be specified on a target level, a group level, or a file level. When options are set on the group or file level, you can choose to override settings inherited from a higher level.

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

The following sections give detailed descriptions of each assembler option.

CODE GENERATION

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



CASE SENSITIVE USER SYMBOLS

By default, case sensitivity is on. This means that, for example, `LABEL` and `label` refer to different symbols. You can choose **Case sensitive user symbols** to turn case sensitivity off, in which case `LABEL` and `label` will refer to the same symbol.

Deselect the **Case sensitive user symbols** option to turn case sensitivity off.

MAKE LIBRARY MODULE

By default, the assembler produces a *program* module ready to be linked with the IAR XLINK Linker™. Select the **Make library module** option if you instead want the assembler to make a *library* module for use with the IAR XLIB Librarian™.

Note: If the NAME directive is used in the source code (to specify the name of the program module), the **Make a LIBRARY module** option is ignored. This means that the assembler produces a program module regardless of the **Make a LIBRARY module** option.

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. The **Warnings** option allows you to enable only some warnings, or to disable all or some warnings.

Use the **Warnings** radio buttons and entry fields to specify which warnings you want to enable or disable.

For additional information about assembler warnings, see the *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide*.

MACRO QUOTE CHARS

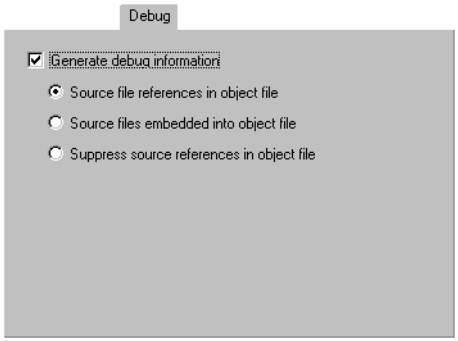
The **Macro quote chars** option sets the characters used for the left and right quotes of each macro argument.

By default, the characters are < and >. This option allows you to change the quote characters to suit an alternative convention or simply to allow a macro argument to contain < or >.

From the drop-down list, select one of four types of brackets to be used as macro quote characters:

DEBUG

The **Debug** options allow you to generate information to be used by a debugger such as the IAR C-SPY® Debugger.



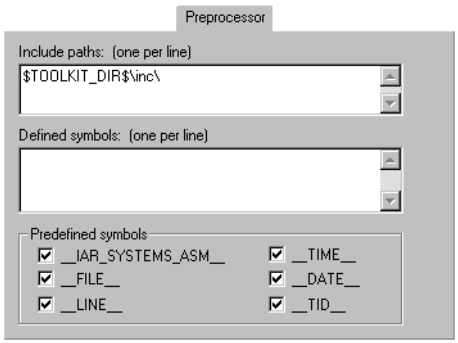
GENERATE DEBUG INFORMATION

In order to reduce the size and link time of the object file, the assembler does not generate debug information in a Release project. You must use the **Generate debug information** option if you want to use a debugger with the program.

When you select this option to generate debug information, **Source file references in object file** is selected by default. If you instead want to include the entire source file into the object file, select **Source files embedded into object file**. If you want to exclude the file references from the object file, select **Suppress source references in object file**.

PREPROCESSOR

The **Preprocessor** options allow you to define include paths and symbols, and to remove the predefined symbols in the assembler.



INCLUDE PATHS

By default the assembler searches for `#include` files in the current working directory. The **Include** option allows you to specify the names of directories that the assembler will also search if it fails to find the file.

Enter the full path of the directories that you want the assembler to search for `#include` files.

To make your project more portable, use the argument variable `$TOOLKIT_DIR$\inc\` for the `inc` subdirectory of the active product (that is, standard system `#include` files) and `$PROJ_DIR$\inc\` for the `inc` subdirectory of the current project directory. For an overview of the argument variables, see page 161.

See the *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide* for information about the `#include` directive.

Note: By default the assembler searches for `#include` files also in the paths specified in the `AAVR_INC` environment variable. We do not, however, recommend the use of environment variables in the IAR Embedded Workbench.

DEFINED SYMBOLS

This option provides a convenient way of specifying a value or choice that you would otherwise have to specify in the source file.

Enter the symbols you want to define, one per line.

- For example, you could arrange your source to produce either the test or production version of your program 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`.

To remove a user-defined symbol, select in the **Defined symbols** list and press the Delete key.

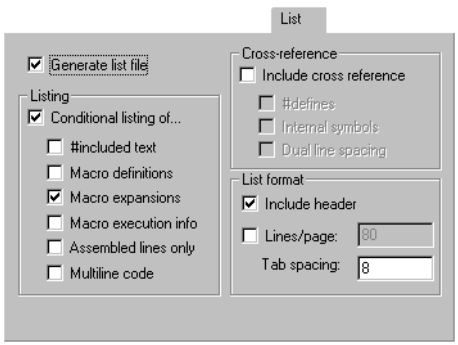
PREDEFINED SYMBOLS

By default, the assembler provides certain predefined symbols; see the *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide* for more information. This option allows you to undefine such a predefined symbol to make its name available for your own use.

To undefine a symbol, deselect it in the **Predefined symbols** list.

LIST

The **List** options are used for making the assembler generate a list file, for selecting the list file contents, and generating other listing-type output.



By default, the assembler does not generate a list file. Selecting **Generate list file** causes the assembler to generate a listing and send it to the file *sourcename*.lst.

Note: 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** category; see *Output directories*, page 97, for additional information.

When **Generate list file** is selected, the **Listing**, **Cross-reference**, and **List formats** options become available.

LISTING

Use the **Conditional listing of...** option to specify which type of information to include in the list file:

Option	*Description*
#included text	Includes #include files in the list file.
Macro definitions	Includes macro definitions in the list file.
Macro expansions	Includes macro expansions in the list file.
Macro execution info	Prints macro execution information on every call of a macro.
Assembled lines only	Excludes lines in false conditional assembly sections from the list file.
Multiline code	Lists the code generated by directives on several lines if necessary.

CROSS-REFERENCE

The **Include cross-reference** option causes the assembler to generate a cross-reference table at the end of the list file. See the *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide* for details.

LIST FORMAT

The **List format** options allow you to specify details about the format of the assembler list file.

Include 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.

Use this option to include the list file header in the list file.

Lines/page
The default number of lines per page is 44 for the assembler list file. Use the **Lines/page** option to set the number of lines per page, within the range 10 to 150.

Tab spacing
By default, the assembler sets eight character positions per tab stop. Use the **Tab spacing** option to change the number of character positions per tab stop, within the range 2 to 9.

XLINK OPTIONS

The XLINK options allow you to control the operation of the IAR XLINK Linker™.

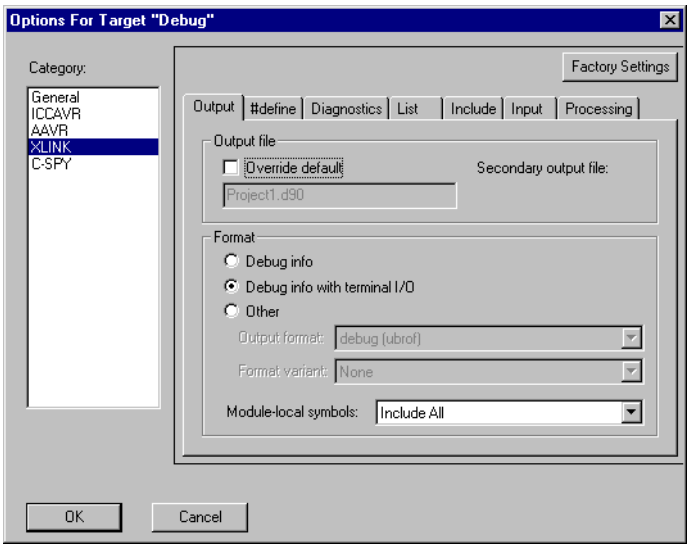
This chapter first describes how to set XLINK options, and then gives reference information about the options available in the IAR Embedded Workbench™.

The options are divided into the following sections:

Note: The XLINK command line options that are used for defining segments in a linker command file are described in the *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide.*

SETTING XLINK OPTIONS

To set XLINK options in the IAR Embedded Workbench choose **Options...** from the **Project** menu to display the **Options** dialog box. Select **XLINK** in the **Category** list to display the XLINK options pages:



Then click the tab corresponding to the type of options you want to view or change.

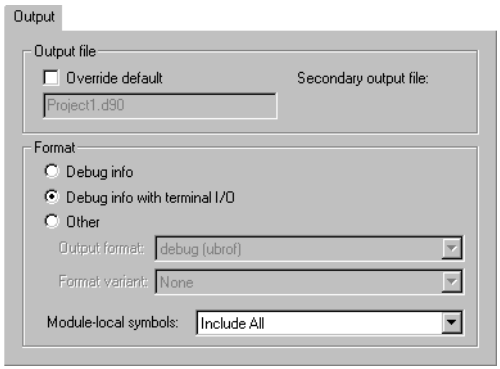
Notice that XLINK options can be specified on a target level, a group level, or a file level. When options are set on the group or file level, you can choose to override settings inherited from a higher level.

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

The following sections give full reference information about the XLINK options.

OUTPUT

The **Output** options are used for specifying the output format and the level of debugging information.



OUTPUT FILE

Use **Output file** to specify the name of the XLINK output file. If a name is not specified the linker will use the name *project*.d90. If a name is supplied without a file type, the default file type for the selected output format (see *Output format*, page 124) will be used.

Note: If you select a format that generates two output files, the file type that you specify will only affect the primary output file (first format).

Override default

Use this option to specify a filename or file type other than default.

FORMAT

The format options determine the format of the output file generated by the IAR XLINK Linker. The IAR proprietary output format is called UBROF, Universal Binary Relocatable Object Format.

Debug info

Use this option to create an output file in **debug (ubrof)** format, with a d90 extension, to be used with the IAR C-SPY® Debugger.

Specifying the option **Debug info** overrides any **Output format** option.

Note: For emulators that support the IAR Systems debug format, select **ubrof** from the **Output format** drop-down list.

Debug info with terminal I/O
Select this option to simulate terminal I/O when running C-SPY.

Output format
Use **Output format** to select an output format other than the default format.

In a *debug* project, the default output format is **debug (ubrof)**.

In a *release* project, the default output format is **Motorola**.

Note: When you specify the **Output format** option as **debug (ubrof)**, C-SPY debug information will not be included in the object code. Use the **Debug info** option instead.

Format variant
Use this option to select enhancements available for some output formats. The **Format variant** options depend on the output format chosen.

For more information, see the *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide*.

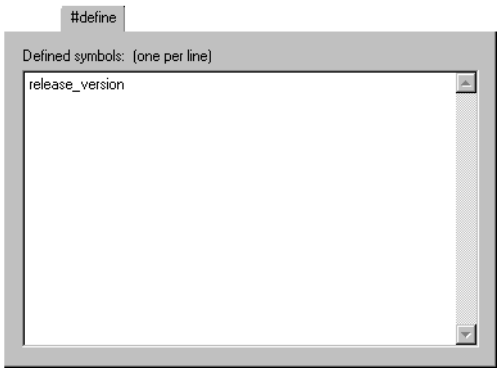
Module-local symbols
Use this option to specify whether local (non-public) symbols in the input modules should be included or not by the IAR XLINK Linker. If suppressed, the local symbols will not appear in the listing cross-reference and they will not be passed on to the output file.

You can choose to ignore just the compiler-generated local symbols, such as jump or constant labels. Usually these are only of interest when debugging at assembler level.

Note: Local symbols are only included in files if they were compiled or assembled with the appropriate option to specify this.

#define

The **#define** option allows you to define symbols.



DEFINE SYMBOL

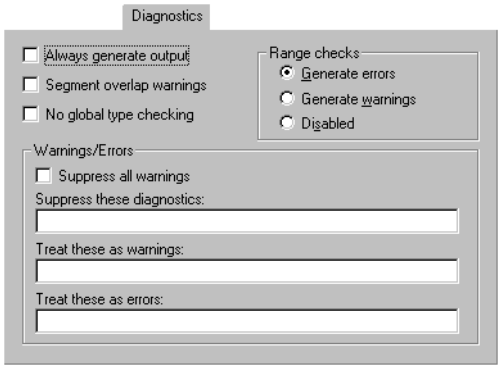
Use **Define symbol** to define absolute symbols at link time. This is especially useful for configuration purposes.

Any number of symbols can be defined in a linker command 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.

XLINK will display an error message if you attempt to redefine an existing symbol.

DIAGNOSTICS

The **Diagnostics** options determine the error and warning messages generated by the IAR XLINK Linker.



ALWAYS GENERATE OUTPUT

Use **Always generate output** to generate an output file even if a non-fatal error was encountered during the linking process, such as a missing global entry or a duplicate declaration. Normally, XLINK will not generate an output file if an error is encountered.

Note: XLINK always aborts on fatal errors, even when this option is used.

The **Always generate output** option allows missing entries to be patched in later in the absolute output image.

SEGMENT OVERLAP WARNINGS

Use **Segment overlap warnings** to reduce segment overlap errors to warnings, making it possible to produce cross-reference maps, etc.

NO GLOBAL TYPE CHECKING

Use **No global type checking** to disable type checking at link time. While a well-written program should not need this option, there may be occasions where it is helpful.

By default, XLINK performs link-time type checking between modules by comparing the external references to an entry with the `PUBLIC` entry (if the information exists in the object modules involved). A warning is generated if there are mismatches.

RANGE CHECKS

Use **Range checks** to specify the address range check. The following table shows the range check options in the IAR Embedded Workbench:

IAR Embedded Workbench	*Description*
Generate errors	An error message is generated
Generate warnings	Range errors are treated as warnings
Disabled	Disables the address range checking

If an address is relocated outside of the target CPU's address range—code, external data, or internal data address—an error message is generated. This usually indicates an error in an assembly language module or in the segment placement.

WARNINGS/ERRORS

By default, the IAR XLINK Linker generates a warning when it detects that something may be wrong, although the generated code may still be correct. The **Warning/Error** options allow you to suppress or enable all warnings, and to change the severity classification of errors and warnings.

Refer to the *XLINK diagnostics* chapter in the *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide* for information about the different warning and error messages.

Use the following options to control the generation of warning and error messages:

Suppress all warnings
Use this option to suppress all warnings.

Suppress these diagnostics
Use this option to specify which warnings or errors to suppress. For example, to disable warning 3, warning 7, and error 10, enter:

```
w3, w7, e10
```

Treat these as warnings
Use this option to specify errors that should be treated as warnings instead. For example, to make error 106 become treated as a warning, enter:

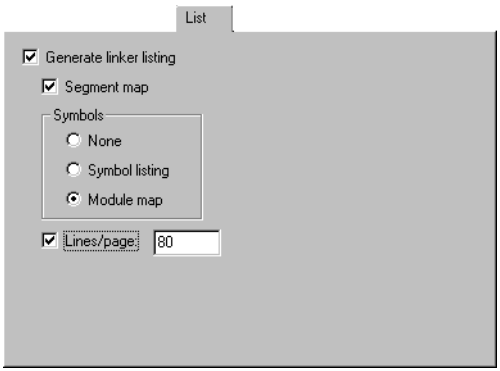
```
e106
```

Treat these as errors
Use this option to specify warnings that should be treated as errors instead. For example, to make warning 26 become treated as an error, enter:

```
w26
```

LIST

The **List** options determine the generation of an XLINK cross-reference listing.



GENERATE LINKER LISTING

Causes the linker to generate a listing and send it to the file *project*.map.

Segment map
Use **Segment map** to include a segment map in the XLINK listing file. The segment map will contain a list of all the segments in dump order.

Symbols
The following options are available:

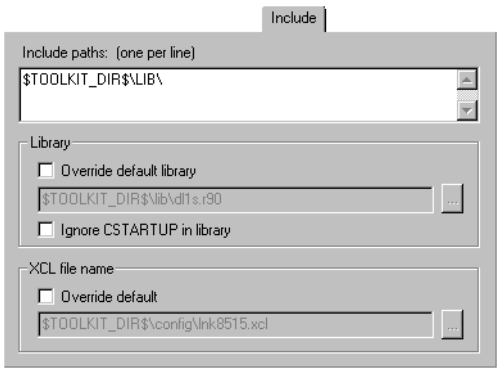
Option	*Description*
None	Symbols will be excluded from the linker listing.
Symbol listing	An abbreviated list of every entry (global symbol) in every module. This entry map is useful for quickly finding the address of a routine or data element.

Option	*Description*
Module map	A list of all segments, local symbols, and entries (public symbols) for every module in the program.

Lines/page
Sets the number of lines per page for the XLINK listings to *lines*, which must be in the range 10 to 150.

INCLUDE

The **Include** option allows you to set the include path for linker command files, and specify the linker command file.



INCLUDE PATHS

Specifies a path name to be searched for object files.

By default, XLINK searches for object files only in the current working directory. The **Include paths** option allows you to specify the names of the directories which it will also search if it fails to find the file in the current working directory.

To make products more portable, use the argument variable `$TOOLKIT_DIR$\lib\` for the `lib` subdirectory of the active product (that is, standard system `#include` files) and `$PROJ_DIR$\lib\` for the `lib` subdirectory of the current project directory. For an overview of the argument variables, see page 161.

LIBRARY

A default library file is selected automatically. You can override this by selecting **Override default library name**, and then specifying an alternative library file.

Ignore CSTARTUP in library
When you select the option **Ignore CSTARTUP in library**, all modules in the library will be treated as library modules, even if they have not been assembled or compiled as library modules.

If you want to include your own version of `cstartup.s90` in a project, use this option to prevent the `CSTARTUP` module in the library from being linked.

You should use this option also when linking assembler source files, since the functionality of `CSTARTUP` does not apply to assembler projects.

The corresponding command line option is `-C`.

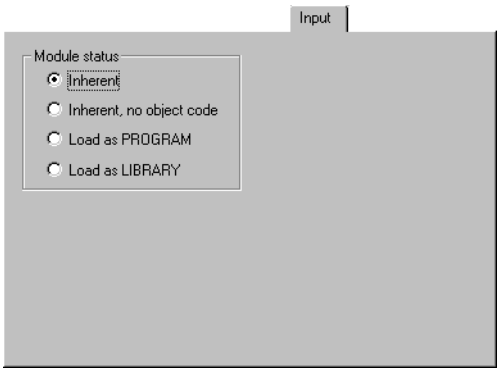
XCL FILENAME

A default linker command file is selected automatically for the **General Target** memory model and processor configuration selected. You can override this by selecting **Override default**, and then specifying an alternative file.

The argument variables `$TOOLKIT_DIR$` or `$PROJ_DIR$` can be used here too, to specify a project-specific or predefined linker command file.

INPUT

The **Input** options define the status of input modules.



MODULE STATUS

Inherent
Use **Inherent** to link files normally, and generate output code.

Inherent, no object code
Use **Inherent, no object code** to empty-load specified input files; they will be processed normally in all regards by the linker but output code will not be generated for these files.

One potential use for this feature is in creating separate output files for programming multiple EPROMs. This is done by empty-loading all input files except the ones that you want to appear in the output file.

Load as PROGRAM
Use **Load as PROGRAM** to temporarily force all of the modules within the specified input files to be loaded as if they were all program modules, even if some of the modules have the `LIBRARY` attribute.

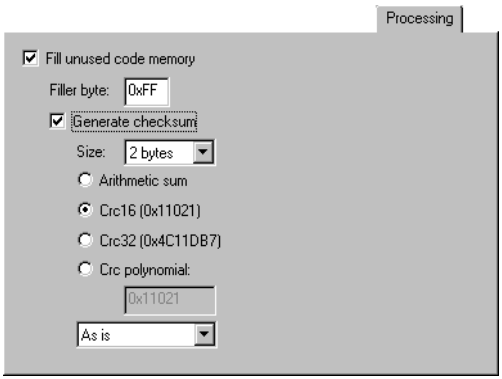
This option is particularly suited for testing library modules before they are installed in a library file, since this option will override an existing library module with the same entries. In other words, XLINK will load the module from the specified *input file* rather than from the original library.

Load as LIBRARY
Use **Load as LIBRARY** to temporarily cause all of the modules within the specified input files to be treated as if they were all library modules, even if some of the modules have the `PROGRAM` attribute. This means that the modules in the input files will be loaded only if they contain an entry that is referenced by another loaded module.

If you have made modifications to `CSTARTUP`, this option is particularly useful when testing `CSTARTUP` before you install it in the library file, since this option will override the existing program module `CSTARTUP`.

PROCESSING

The **Processing** options allow you to specify additional options determining how the code is generated.



FILL UNUSED CODE MEMORY

Use **Fill unused code memory** to fill all gaps between segment parts introduced by the linker with the value *hexvalue*. The linker can introduce gaps either because of alignment restriction, or at the end of ranges given in segment placement options.

The default behavior, when this option is not used, is that these gaps are not given a value in the output file.

Filler byte

Use this option to specify size, in hexadecimal notation, of the filler to be used in gaps between segment parts.

Generate checksum

Use **Generate checksum** to checksum all generated raw data bytes. This option can only be used if the **Fill unused code memory** option has been specified.

Size specifies the number of bytes in the checksum, which can be 1, 2, or 4.

One of the following algorithms can be used:

Algorithms	*Description*
Arithmetic sum	Simple arithmetic sum.
Crc16	CRC16, generating polynomial 0x11021 (default)

Algorithms	*Description*
Crc32	CRC32, generating polynomial 0x104C11DB7.
Crc polynomial	CRC with a generating polynomial of *hexvalue*.

You may also specify that one's complement or two's complement should be used.

In all cases it is the least significant 1, 2, or 4 bytes of the result that will be output, in the natural byte order for the processor.

The CRC checksum is calculated as if the following code was called for each bit in the input, starting with a CRC of 0:

```
unsigned long
crc(int bit, unsigned long oldcrc)
{
   unsigned long newcrc = (oldcrc << 1) ^ bit;
   if (oldcrc & 0x80000000)
      newcrc ^= POLY;
   return newcrc;
}
```

`POLY` is the generating polynomial. The checksum is the result of the final call to this routine. If the complement is specified, the checksum is the one's or two's complement of the result.

The linker will place the checksum byte(s) at the label `__checksum` in the segment `CHECKSUM`. This segment must be placed using the segment placement options like any other segment.

For additional information about segment control, see the *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide*.

C-SPY OPTIONS

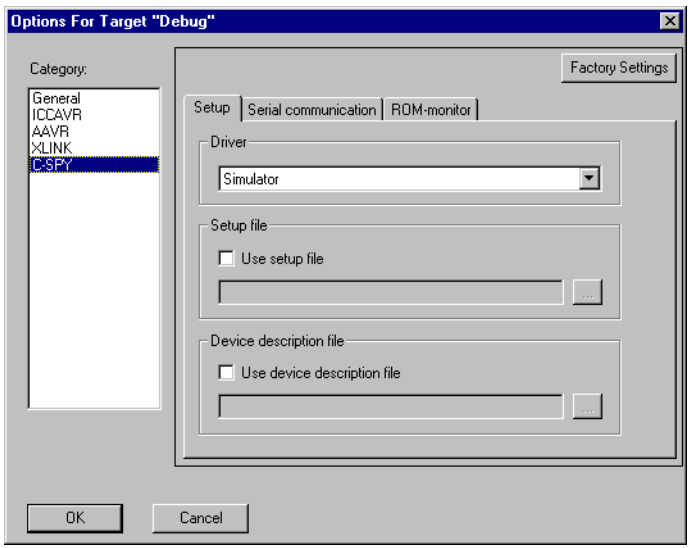
This chapter describes how to set C-SPY options in the IAR Embedded Workbench™ and gives full reference information about the options.

Note: If you prefer to run C-SPY outside the IAR Embedded Workbench, refer to the chapter *C-SPY command line options* in *Part 4: The C-SPY simulator* in this guide for information about the available options.

Reference information about the IAR C-SPY® Debugger is provided in the chapter *C-SPY reference* in *Part 4: The C-SPY simulator* in this guide.

SETTING C-SPY OPTIONS

To set C-SPY options in the IAR Embedded Workbench choose **Options...** from the **Project** menu, and select **C-SPY** in the **Category** list to display the C-SPY options page:



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

Note: The **Serial communication** and **ROM-monitor** options will be enabled in future versions of the product.

SETUP

The **Setup** options specify the C-SPY driver and the setup and device description files to be used.

DRIVER

Selects the appropriate driver for use with C-SPY, for example a simulator or an emulator. The following driver is currently available:

C-SPY version	*Driver*
Simulator	`savr.cdr`

Contact your distributor or IAR representative, or visit the IAR website at **www.iar.com** for the most recent information about the available C-SPY versions.

SETUP FILE

To register the contents of a macro file in the C-SPY startup sequence, select **Use setup file** and enter the path and name name of your setup file, for example, `watchdog.mac`. If no extension is specified, the extension `mac` is assumed. A browse button is available for your convenience.

DEVICE DESCRIPTION FILE

Use this option to load the device-specific definitions allowing you to view and edit the contents of the special function registers while debugging.

The device description files contain various device specific information such as I/O registers (SFR) definitions, vector, and control register definitions. Some files are provided with the product and have the extension `ddf`. A browse button is available for your convenience.

IAR Embedded Workbench reference

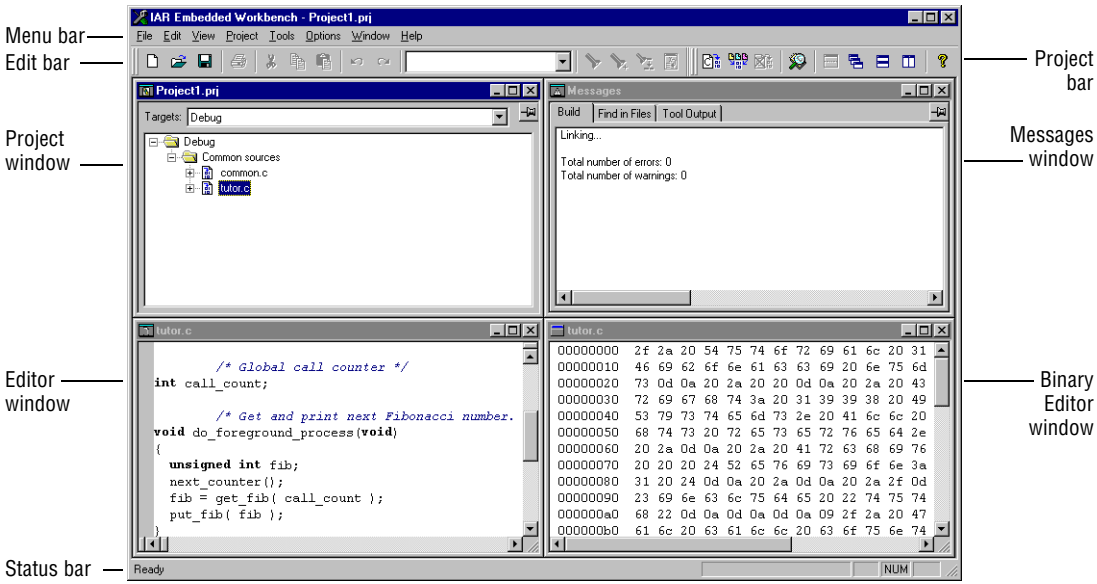
This chapter provides complete reference information about the IAR Embedded Workbench™.

It first gives information about the components of the IAR Embedded Workbench window, and each of the different types of window it encloses.

It then gives details of the menus, and the commands on each menu.

The IAR Embedded Workbench window

The following illustration shows the different components of the IAR Embedded Workbench window.



These components are explained in greater detail in the following sections.

MENU BAR

Gives access to the IAR Embedded Workbench menus.

Menu	*Description*
File	The **File** menu provides commands for opening source and project files, saving and printing, and exiting from the IAR Embedded Workbench.
Edit	The **Edit** menu provides commands for editing and searching in Editor windows.
View	The commands on the **View** menu allow you to change the information displayed in the IAR Embedded Workbench window.
Project	The **Project** menu provides commands for adding files to a project, creating groups, and running the IAR tools on the current project.
Tools	The **Tools** menu is a user-configurable menu to which you can add tools for use with the IAR Embedded Workbench.
Options	The **Options** menu allows you to customize the IAR Embedded Workbench to your requirements.
Window	The commands on the **Window** menu allow you to manipulate the IAR Embedded Workbench windows and change their arrangements on the screen.
Help	The commands on the **Help** menu provide help about the IAR Embedded Workbench.

The menus are described in greater detail on the following pages.

TOOLBARS

The IAR Embedded Workbench window contains two toolbars:

- The edit bar.
- The project bar.

The edit bar provides buttons for the most useful commands on the IAR Embedded Workbench menus, and a text box for entering a string to do a toolbar search.

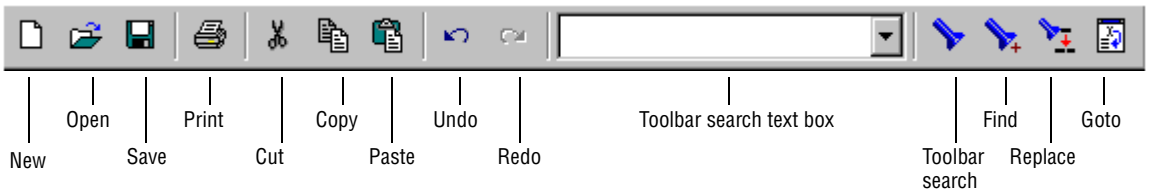
The project bar provides buttons for the build and debug options on the **Project** menu.

You can move either toolbar to a different position in the IAR Embedded Workbench window, or convert it to a floating palette, by dragging it with the mouse.

You can display a description of any button by pointing to it with the mouse button. When a command is not available the corresponding toolbar button will be grayed out, and you will not be able to select it.

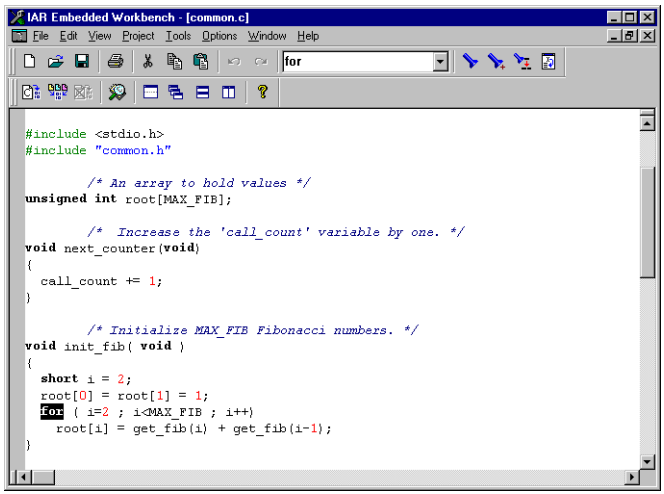
Edit bar

The following illustration shows the menu commands corresponding to each of the edit bar buttons:



Toolbar search

To search for text in the frontmost Editor window enter the text in the **Toolbar search** text box, and press Enter or click the **Toolbar search** button.

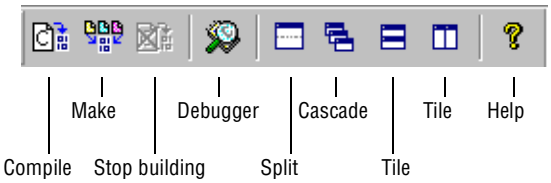


Alternatively, you can select a string you have previously searched for from the drop-down list box.

You can choose whether or not the edit bar is displayed using the **Edit Bar** command on the **View** menu.

Project bar

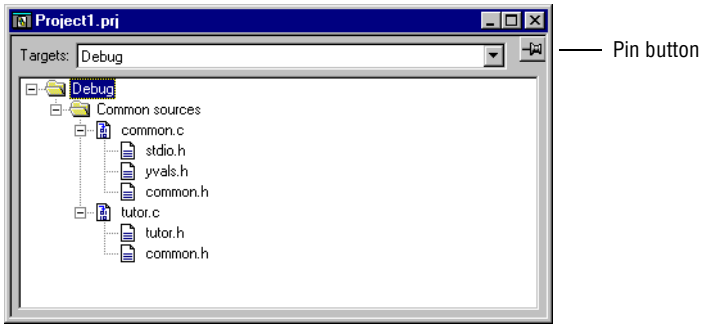
The following illustration shows the menu command corresponding to each of the project bar buttons:

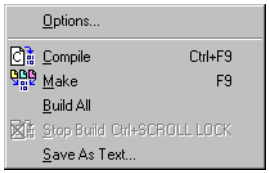


You can choose whether or not the project bar is displayed using the **Project Bar** command on the **View** menu.

PROJECT WINDOW

The Project window shows the name of the current project and a tree representation of the groups and files included in the project.





Pressing the right mouse button in the Project window displays a pop-up menu which gives you convenient access to several useful commands. **Save As Text...** allows you to save a description of the project, including all options that you have specified.

Pin button

The **Pin** button, in the top right corner of the Project window, allows you to pin the window to the desktop so that it is not affected by the **Tile** or **Cascade** commands on the **Window** menu.

Targets

The top node in the tree shows the current target. You can change the target by choosing a different target from the **Targets** drop-down list box at the top of the Project window. Each target corresponds to a different version of your project that you want to compile or assemble. For example, you might have a target called **Debug**, which includes debugging code, and one called **Release**, with the debugging code omitted.

You can expand the tree by double-clicking on the target icon, or by clicking on the plus sign icon, to display the groups included in this target.

Groups
Groups are used for collecting together related source files. Each group may be included in one or more targets, and a source file can be present in one or more groups.

Source files
You can expand each group by double-clicking on its icon, or by clicking on the plus sign icon, to show the list of source files it contains.

Once a project has been successfully built any include files are displayed in the structure below the source file that included them.

Note: The include files associated with a particular source file may depend on which target the source file appears in, since preprocessor or directory options may affect which include files are associated with a particular source file.

Editing a file
To edit a source or include file, double-click its icon in the Project window tree display.

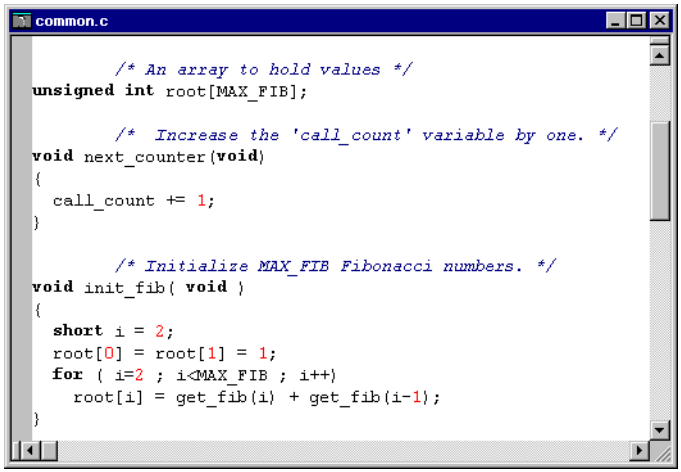
Moving a source file between groups
You can move a source file between two groups by dragging its icon between the group icons in the Project window tree display.

Removing items from a project
To remove an item from a project, click on it to select it, and then press Delete.

To remove a file from a project you can also use the **Project Files** dialog box, displayed by choosing **Files...** from the **Project** menu.

EDITOR WINDOW

Source files are displayed in the Editor window. The IAR Embedded Workbench editor automatically recognizes the syntax of C or Embedded C++ programs, and displays the different components of the program in different text styles.



The following table shows the default styles used for each component of a C or Embedded C++ program:

Item	*Style*
Default	Black plain
Keyword	Black bold
Strings	Blue
Preprocessor	Green
Integer (dec)	Red
Integer (oct)	Magenta
Integer (hex)	Magenta
Real	Blue
C++ comment: //	Dark blue italic
C comment: /*...*/	Dark blue italic

To change these styles choose **Settings...** from the **Options** menu, and then select the **Colors and Fonts** page in the **Settings** dialog box, see *Colors and Fonts*, page 168.

Auto indent
The editor automatically indents a line to the same indent as the previous line, making it easy to lay out programs in a structured way.

Matching brackets
When the cursor is next to a bracket you can automatically find the matching bracket by choosing **Match Brackets** from the **Edit** menu.

Read-only and modification indicators
The name of the open source file is displayed in the Editor window title bar.

If a file is a read-only file, the text `(Read Only)` appears after the file name, for example `Common (Read Only)`.

When a file has been modified after it was last saved, an asterisk appears after the window title, for example `Common *`.

Editor options
The IAR Embedded Workbench editor provides a number of special features, each of which can be enabled or disabled independently in the **Editor** page of the **Settings** dialog box. For more information see *Settings...*, page 163.

Editor key summary
The following tables summarize the editor's keyboard commands.

Use the following keys and key combinations for 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 move the insertion point	*Press*
To the first line in the file	Ctrl + Home
To the last line in the file	Ctrl + End

Use the following keys and key combinations for 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

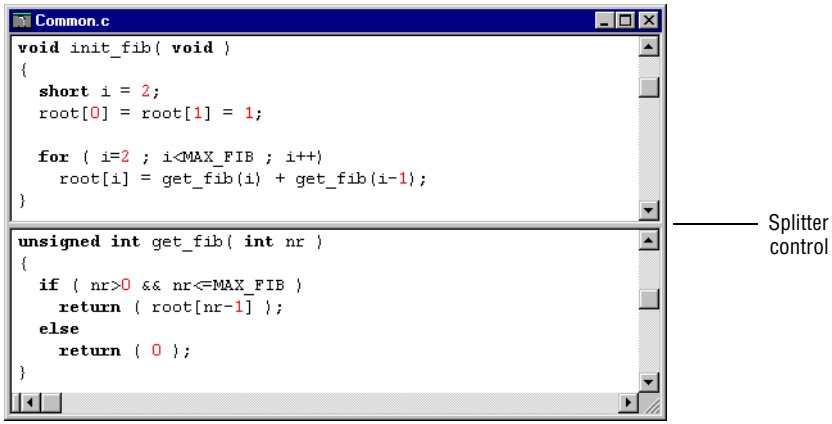
Use the following key combinations for selecting text:

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

Splitting the Editor window into panes

You can split the Editor window horizontally or vertically into multiple panes, to allow you to look at two different parts of the same source file at once, or cut and paste text between two different parts.

To split the window drag the appropriate **splitter** control to the middle of the window:



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

You can also split a window into panes using the **Split** command on the **Window** menu.

STATUS BAR

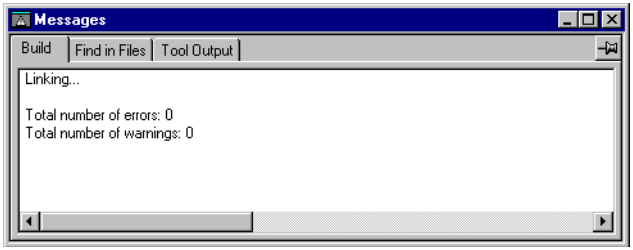
Displays the status of the IAR Embedded Workbench, and the state of the modifier keys.

As you are editing in the Editor window the status bar shows the current line and column number containing the cursor, and the Caps Lock, Num Lock, and Overwrite status:

You can choose whether or not the status bar is displayed using the **Status Bar** command on the **View** menu.

MESSAGES WINDOW

The Messages window shows the output from different IAR Embedded Workbench commands. The window is divided into multiple pages and you select the appropriate page by clicking on the corresponding tab.



Pressing the right mouse button in the Messages window displays a pop-up menu which allows you to save the contents of the window as a text file.

To specify the level of output to the Messages window, select the **Make Control** page in the Settings window. See *Make Control*, page 169.

Pin button
The **Pin** button, in the top right corner of the Messages window, allows you to pin the window to the desktop so that it is not affected by the **Tile** or **Cascade** commands on the **Window** menu.

Build
Build shows the messages generated when building a project. Double-clicking a message in the Build panel opens the appropriate file for editing, with the cursor at the correct position.

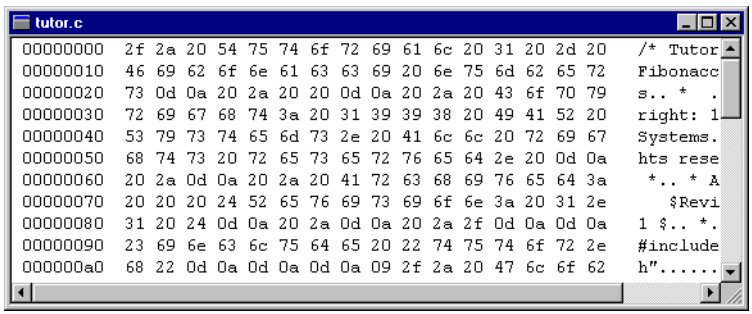
Find in Files
Find in Files displays the output from the **Find in Files...** command on the **Edit** menu. Double-clicking an entry in the panel opens the appropriate file with the cursor positioned at the correct location.

Tool Output
Tool Output displays any messages output by user-defined tools in the **Tools** menu.

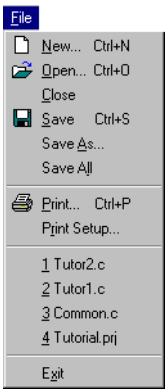
BINARY EDITOR WINDOW

The Binary Editor window displays and allows you to edit the contents of a binary file. The data is displayed in hexadecimal format, with its **ASCII** equivalent to the right of each line. You can edit the contents by inserting or overwriting data.

To open the Binary Editor window, choose **Binary Editor...** from the **Tools** menu.



FILE MENU

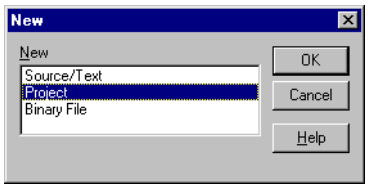


The **File** menu provides commands for opening projects and source files, saving and printing, and exiting from the IAR Embedded Workbench.

The menu also includes a numbered list of the most recently opened files to allow you to open one by selecting its name from the menu.

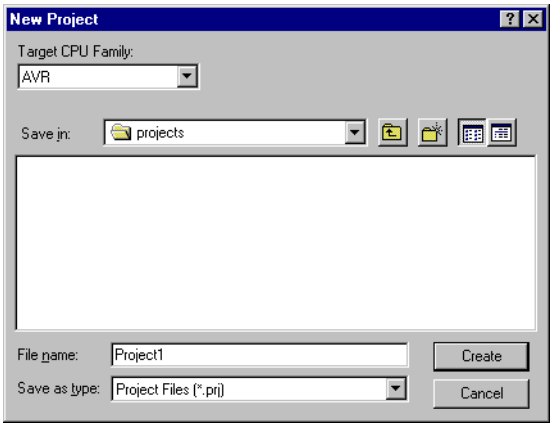
NEW...

Displays the following dialog box to allow you to specify whether you want to create a new project, or a new text file:



Choosing **Source/Text** opens a new Editor window to allow you to enter a text file.

Choosing **Project** displays the following dialog box to allow you to specify a name for the project and the target CPU family:



The project will then be displayed in a new Project window. By default new projects are created with two targets, **Release** and **Debug**.

Selecting **Binary File** opens the Binary Editor window, allowing you to enter binary data:

Note: The Binary Editor starts in overwrite mode.

OPEN...

Displays a standard **Open** dialog box to allow you to select a text or project file to open. Opening a new project file automatically saves and closes any currently open project.

CLOSE

Closes the active window.

You will be warned if a text document has changed since it was last saved, and given the opportunity to save it before closing. Projects are saved automatically.

SAVE

Saves the current text or project document.

SAVE AS...

Displays the standard **Save As** dialog box to allow you to save the active document with a different name.

SAVE ALL

Saves all open text documents.

PRINT...

Displays the standard **Print** dialog box to allow you to print a text document.

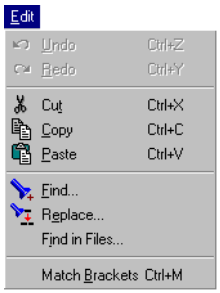
PRINT SETUP...

Displays the standard **Print Setup** dialog box to allow you to set up the printer before printing.

EXIT

Exits from the IAR Embedded Workbench. You will be asked whether to save any changes to text windows before closing them. Changes to the project are saved automatically.

EDIT MENU



The **Edit** menu provides commands for editing and searching in Editor windows.

UNDO

Undoes the last edit made to the current Editor window.

REDO

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, COPY, PASTE

Provide the standard Windows functions for editing text within Editor windows and dialog boxes.

FIND...

Displays the following dialog box to allow you to search for text within the current Editor window:

Enter the text to search for in the **Find What** text box.

Select **Match Whole Word Only** to find the specified text only if it occurs as a separate word. Otherwise `int` will also find `print`, `sprintf` etc.

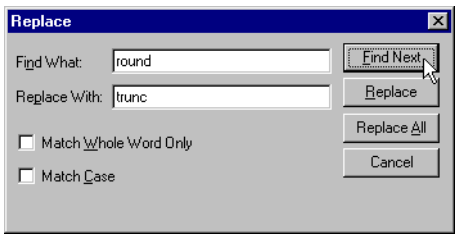
Select **Match Case** to find only occurrences that exactly match the case of the specified text. Otherwise specifying `int` will also find `INT` and `Int`.

Select **Up** or **Down** to specify the direction of the search.

Choose **Find Next** to find the next occurrence of the text you have specified.

REPLACE...

Allows you to search for a specified string and replace each occurrence with another string.

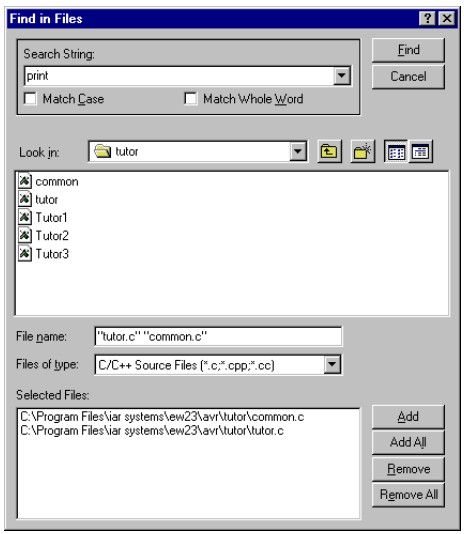


Enter the text to replace each found occurrence in the **Replace With** box. The other options are identical to those for **Find...**.

Choose **Find Next** to find the next occurrence, and **Replace** to replace it with the specified text. Alternatively choose **Replace All** to replace all occurrences in the current Editor window.

FIND IN FILES...

Allows you to search for a specified string in multiple text files. The following dialog box allows you to specify the criteria for the search:



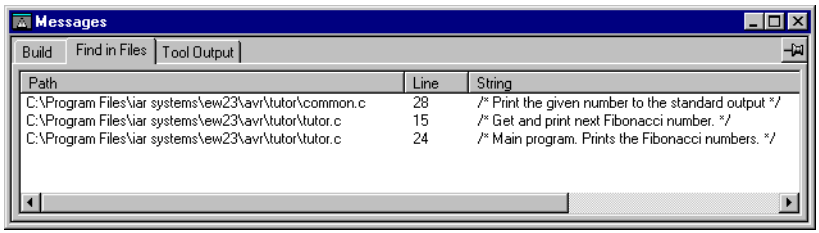
Specify the string you want to search for in the **Search String** text box, or select a string you have previously searched for from the drop-down list box.

Select **Match Whole Word** or **Match Case** to restrict the search to the occurrences that match as a whole word or match exactly in case, respectively.

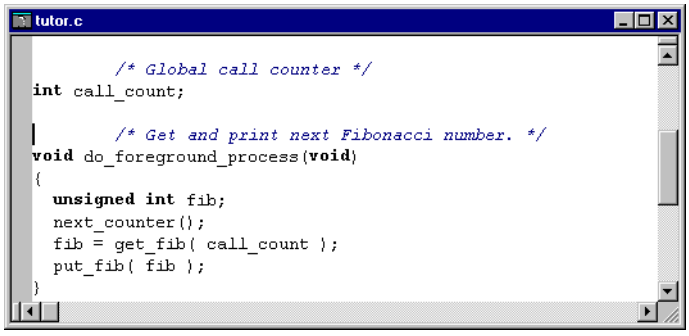
Select each file you want to search in the **File Name** list, and choose **Add** to add it to the **Selected Files** list.

You can add all the files in the **File Name** list by choosing **Add All**, or you can select multiple files using the Shift and Ctrl keys and choose **Add** to add the files you have selected. Likewise you can remove files from the **Selected Files** list using the **Remove** and **Remove All** buttons.

When you have selected the files you want to search choose **Find** to proceed with the search. All the matching occurrences are listed in the Messages window. You can then very simply edit each occurrence by double-clicking it:



This opens the corresponding file in an Editor window with the cursor positioned at the start of the line containing the specified text:



MATCH BRACKETS

If the cursor is positioned next to a bracket this command moves the cursor to the matching bracket, or beeps if there is no matching bracket.

VIEW MENU

The commands on the **View** menu allow you to change the information displayed in the IAR Embedded Workbench window.

EDIT BAR

Toggles the edit bar on and off.

PROJECT BAR

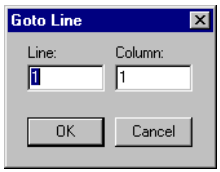
Toggles the project bar on and off.

STATUS BAR

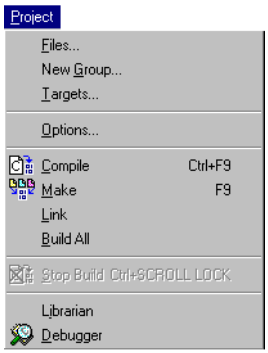
Toggles the status bar on and off.

GOTO LINE...

Displays the following dialog box to allow you to move the cursor to a specified line and column in the current Editor window:



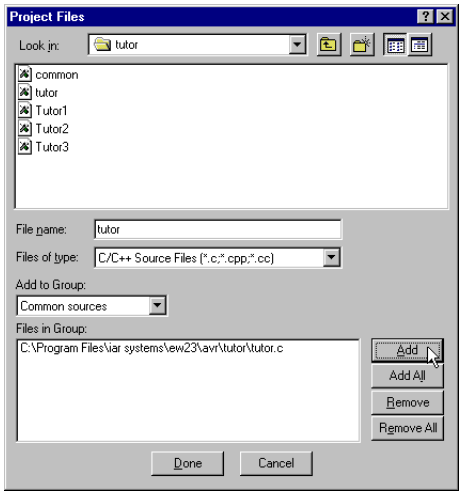
PROJECT MENU



The **Project** menu provides commands for adding files to a project, creating groups, specifying project options, and running the IAR Systems development tools on the current project.

FILES...

Displays the following dialog box to allow you to edit the contents of the current project:



The **Add to Group** drop-down list box shows all the groups included in the current target. Select the one you want to edit, and the files currently in that group are displayed in the **Files in Group** list at the bottom of the dialog box.

The upper part of the **Project Files** dialog box is a standard file dialog box, to allow you to locate and select the files you want to add to each particular group.

Adding files to a group

To add files to the currently displayed group select them using the standard file controls in the upper half of the dialog box and choose the **Add** button, or choose **Add All** to add all the files in the **File Name** list box.

Removing files from a group

To remove files from the currently displayed group select them in the **Files in Group** list and choose **Remove**, or choose **Remove All** to remove all the files from the group.

You can use the **Project Files** dialog box to make changes to several groups. Choosing **Done** will then apply all the changes to the project. Alternatively, choosing **Cancel** will discard all the changes and leave the project unaffected.

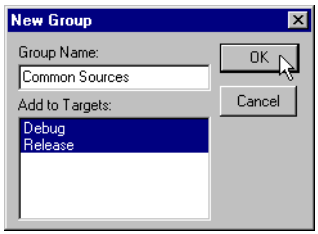
Source file paths

The IAR Embedded Workbench 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 IAR Embedded Workbench will use a path relative to the project file when accessing the source file.

NEW GROUP...

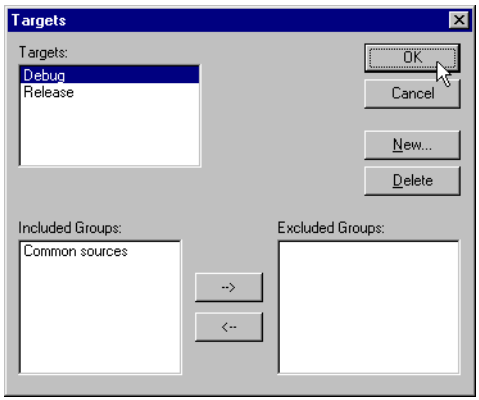
Displays the following dialog box to allow you to create a new group:



Specify the name of the group you want to create in the **Group Name** text box. Select the targets to which you want to add the new group in the **Add to Targets** list. By default the group is added to all targets.

TARGETS...

Displays the following dialog box to allow you to create new targets, and display or change the groups included in each target:



To create a new target, select **New...** and enter a name for the new target.

To delete a target, select it and click **Delete**.

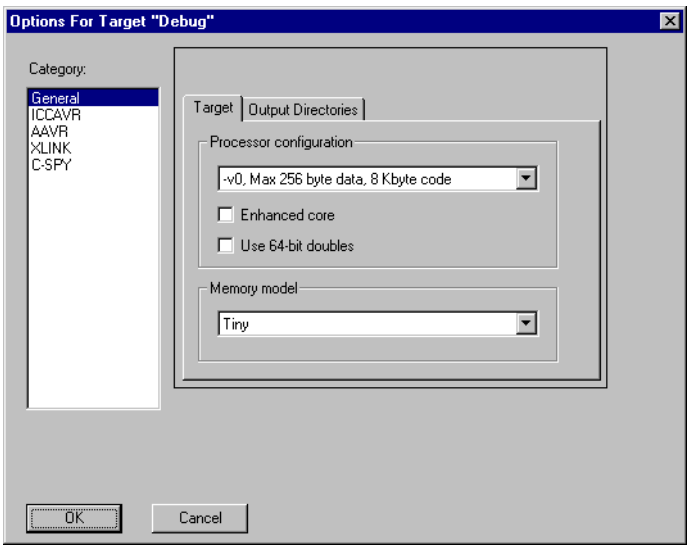
To view the groups included in a target select it in the **Targets** list.

The groups are shown in the **Included Groups** list, and you can add or remove groups using the arrow buttons.

OPTIONS...

Displays the **Options** dialog box to allow you to set directory and compiler options on the selected item in the Project window.

You can set options on the entire target, on a group of files, or on an individual file.



The **Category** list allows you to select which set of options you want to modify. The options available in the **Category** list will depend on the tools installed in your IAR Embedded Workbench, and will typically include the following options:

Category	*Description*	*Refer to the chapter*
General	General options	*General options*
ICCAVR	AVR Compiler options	*Compiler options*
AAVR	AVR Assembler options	*Assembler options*
XLINK	IAR XLINK Linker™ options	*XLINK options*
C-SPY	IAR C-SPY® Debugger options	*C-SPY options*

Selecting a category displays one or more pages of options for that component of the IAR Embedded Workbench.

For more detailed information about the tools installed, see the *AVR IAR Compiler Reference Guide* and the *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide*.

COMPILE

Compiles or assembles the currently active file or project as appropriate.

You can compile a file or project by selecting its icon in the Project window and choosing **Compile**. Alternatively, you can compile a file in the Editor window provided it is a member of the current target.

MAKE

Brings the current target up to date by compiling, assembling, and linking only the files that have changed since last build.

LINK

Explicitly relinks the current target.

BUILD ALL

Rebuilds and relinks all files in the current target.

STOP BUILD

Stops the current build operation.

LIBRARIAN

Starts the IAR XLIB Librarian™ to allow you to perform operations on library modules in library files.

DEBUGGER

Starts the IAR C-SPY Debugger so that you can debug the project object file.

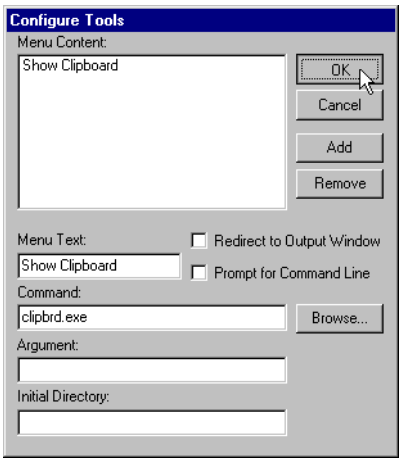
You can specify the version of C-SPY to run in the **Debug** options for the target. If necessary a Make will be performed before running C-SPY to ensure that the project is up to date.

TOOLS MENU

The **Tools** menu is a user-configurable menu to which you can add tools for use with the IAR Embedded Workbench.

CONFIGURE TOOLS...

Configure Tools... displays the following dialog box to allow you to specify a user-defined tool to add to the menu:



Specify the text for the menu item in the **Menu Text** box, and the command to be run when you select the item in the **Command** text box. Alternatively, choose **Browse** to display a standard file dialog box to allow you to locate an executable file on disk and add its path to the **Command** text box.

Specify the argument for the command in the **Argument** text box, or select **Prompt for Command Line** to display a prompt for the command line argument when the command is selected from the **Tools** menu.

Variables can be used in the arguments, allowing 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.

The following argument variables can be used:

Variable	*Description*
`$CUR_DIR$`	Current directory
`$CUR_LINE$`	Current line
`$EW_DIR$`	Directory of the IAR Embedded Workbench, for example `c:\program files\iar systems\ew23`
`$EXE_DIR$`	Directory for executable output
`$FILE_DIR$`	Directory of active file, no file name
`$FILE_FNAME$`	File name 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 file name without path
`$PROJ_PATH$`	Full path of project file
`$TARGET_DIR$`	Directory of primary output file
`$TARGET_FNAME$`	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\ew23\avr`

The **Initial Directory** text box allows you to specify an initial working directory for the tool.

Select **Redirect to Output Window** to display any console output from the tool in the Tools window.

Note: Tools that require user input or make special assumptions regarding the console that they execute in, will *not* work if you set this option.

When you have specified the command you want to add choose **Add** to add it to the **Menu Content** list. You can remove a command from the **Tools** menu by selecting it in this list and choosing **Remove**.

To confirm the changes you have made to the **Tools** menu and close the dialog box choose **OK**.

The menu items you have specified will then be displayed in the **Tools** menu:

Specifying command line commands or batch files
Command line commands or batch files need to be run from a command shell, so to add these to the **Tools** menu you need to specify an appropriate command shell in the **Command** text box, and the command line command or batch file name in the **Argument** text box.

The command shells are specified as follows:

System	*Command shell*
Windows 95/98	`command.com`
Windows NT	`cmd.exe` (recommended) or `command.com`

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 IAR Embedded Workbench to detect when the tool has completed.

For 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 `command` and **Argument** as:

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

or

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

BINARY EDITOR...

Opens the Binary Editor window where you can edit a file in hexadecimal format. It displays a standard file dialog box allowing you to select a file. For more information, see *Binary Editor window*, page 148.

RECORD MACRO

Allows you to record a sequence of editor keystrokes as a macro.

STOP RECORD MACRO

Ends the recording of a macro.

PLAY MACRO

Replays the macro you have recorded.

OPTIONS MENU

The **Settings...** command on the **Options** menu allows you to customize the IAR Embedded Workbench according to your own requirements.

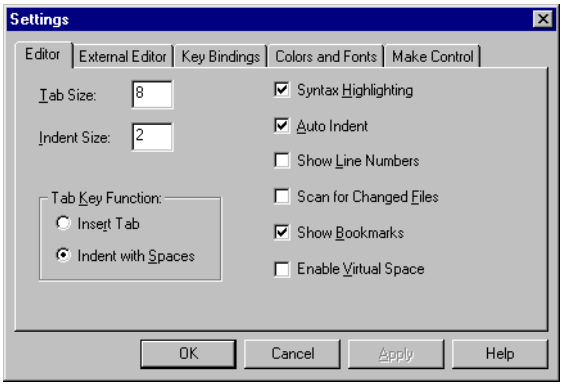
SETTINGS...

Displays the **Settings** dialog box to allow you to customize the IAR Embedded Workbench.

Select the feature you want to customize by clicking the **Editor**, **External Editor**, **Key Bindings**, **Colors and Fonts**, or **Make Control** tabs.

Editor

The **Editor** page allows you to change the editor options:



It provides the following options:

Option	*Description*
Tab Size	Specifies the number of character spaces corresponding to each tab.
Indent Size	Specifies the number of character spaces to be used for indentation.
Tab Key Function	Specifies how the tab key is used.
Syntax Highlighting	Displays the syntax of C or Embedded C++ programs in different text styles.
Auto Indent	When you insert a line, the new line will automatically have the same indentation as the previous line.
Show Line Number	Displays line numbers in the Editor window.
Scan for Changed Files	The editor will check if files have been modified by some other tool and automatically reload them. If a file has been modified in the IAR Embedded Workbench, you will be prompted first.
Show Bookmarks	Displays compiler errors and **Find in Files...** search results.

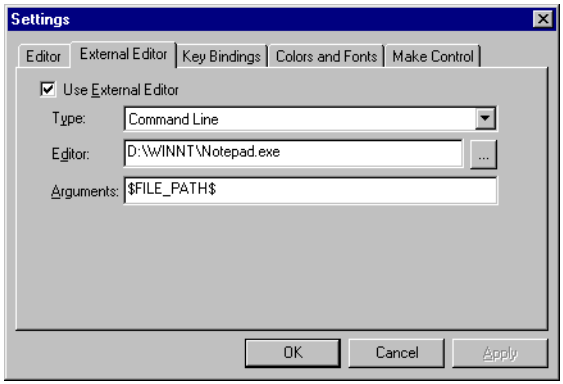
Option	*Description*
Enable Virtual Space	Allows the cursor to move outside the text area.

For more information about the IAR Embedded Workbench Editor, see *Editor window*, page 142.

External Editor

The **External Editor** page allows you to specify an external editor.

An external editor can be called either by passing command line parameters or by using DDE (Windows Dynamic Data Exchange).



Selecting **Type: Command Line** will call the external editor by passing command line parameters. Provide the file name and path of your external editor in the **Editor** field. Then specify the command line to pass to the editor in the **Arguments** field.

Note: Variables can be used in arguments. See page 161 for information about the argument variables that are available.

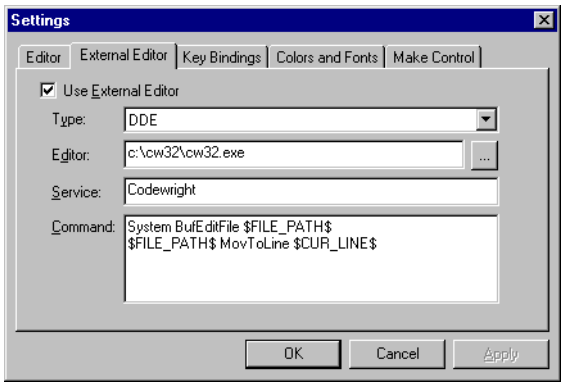
Selecting **Type: DDE** will call the external editor by using DDE. Provide the file name and path of your external editor in the **Editor** field.

Specify the DDE service name used by the editor in the **Service** field. Then specify a sequence of command strings to send to the editor in the **Command** field.

The command strings should be entered as:

```
DDE-Topic CommandString
DDE-Topic CommandString
```

as in the following example, which applies to Codewright®:

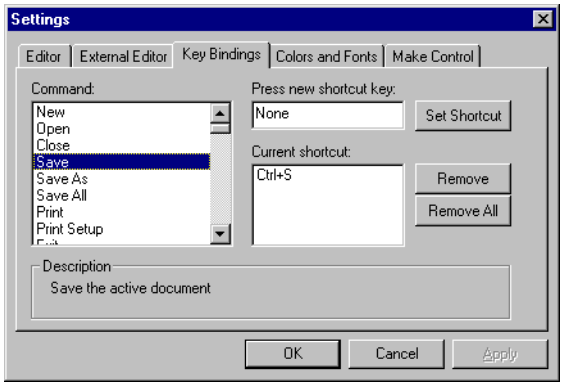


The service name and command strings depend on the external editor that you are using. Refer to the user documentation of your external editor to find the appropriate settings.

Note: Variables can be used in the arguments. See page 161 for more information about the argument variables that are available.

Key Bindings

The **Key Bindings** page displays the shortcut keys used for each of the menu options, and allows you to change them:



Select the command you want to edit in the **Command** list. Any currently defined shortcut keys are shown in the **Current shortcut** list.

To add a shortcut key to the command click in the **Press new shortcut key** box and type the key combination you want to use. Then click **Set Shortcut** to add it to the **Current shortcut** list. You will not be allowed to add it if it is already used by another command.

To remove a shortcut key select it in the **Current shortcut** list and click **Remove**, or click **Remove All** to remove all the command's shortcut keys.

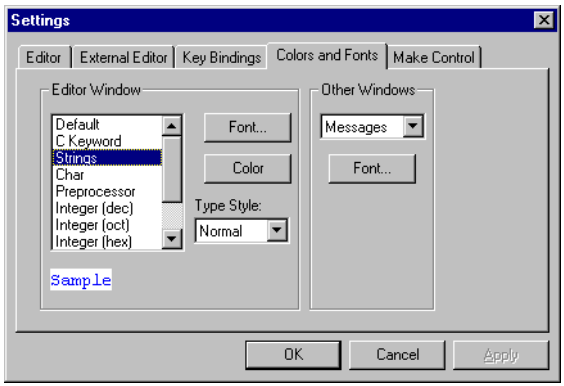
Then choose **OK** to use the new key bindings you have defined and the menus will be updated to show the shortcuts you have defined.

You can set up more than one shortcut for a command, but only one will be displayed in the menu.

Colors and Fonts

The **Colors and Fonts** page allows you to specify the colors and fonts used for text in the Editor windows, and the font used for text in the other windows.

The panel shows a list of the C or Embedded C++ syntax elements you can customize in the Editor window:



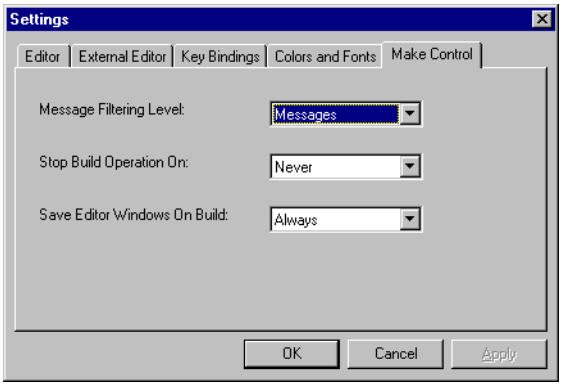
To specify the style used for each element of C or Embedded C++ syntax in the Editor window, select the item you want to define from the **Editor Window** list. The current setting is shown in the **Sample** box below the list box.

You can choose a text color by clicking **Color** and a font by clicking **Font...**. You can also choose the type style from the **Type Style** drop-down list.

Then choose **OK** to use the new styles you have defined, or **Cancel** to revert to the previous styles.

Make Control

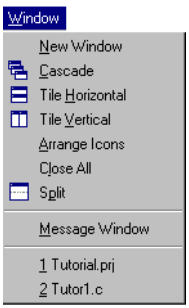
The **Make Control** page allows you to set options for Make and Build:



The following table gives the options, and the alternative settings for each option:

Option	*Setting*
Message Filtering Level	All: Show all messages. Include compiler and linker information. Messages: Show messages, warnings, and errors. Warnings: Show warnings and errors. Errors: Show errors only.
Stop Build Operation On	Never: Do not Stop. Warnings: Stop on warnings and errors. Errors: Stop on errors.
Save Editor Windows On Build	Always: Always save before Make or Build. Ask: Prompt before saving. Never: Do not save.

WINDOW MENU



The commands on the **Window** menu allow you to manipulate the Workbench windows and change their arrangement on the screen.

The last section of the **Window** menu lists the windows currently open on the screen, and allows you to activate one by selecting it.

NEW WINDOW

Opens a new window for the current file.

CASCADE, TILE HORIZONTAL, TILE VERTICAL

Provide the standard Windows functions for arranging the IAR Embedded Workbench windows on the screen.

ARRANGE ICONS

Arranges minimized window icons neatly at the bottom of the IAR Embedded Workbench window.

CLOSE ALL

Closes all open windows.

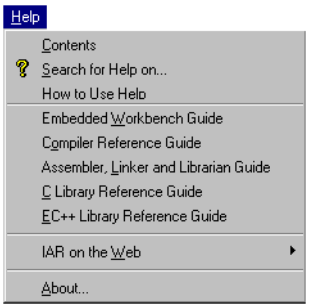
SPLIT

Allows you to split an Editor window horizontally into two panes to allow you to see two parts of a file simultaneously.

MESSAGE WINDOW

Opens the Messages window that displays messages and text output from the IAR Embedded Workbench commands.

HELP MENU



Provides help about the IAR Embedded Workbench.

CONTENTS

Displays the Contents page for help about the IAR Embedded Workbench.

SEARCH FOR HELP ON...

Allows you to search for help on a keyword.

HOW TO USE HELP

Displays help about using help.

EMBEDDED WORKBENCH GUIDE

Provides access to an online version of this user guide, available in Acrobat® Reader format.

COMPILER REFERENCE GUIDE

Provides access to an online version of the *AVR IAR Compiler Reference Guide*, available in Acrobat® Reader format.

ASSEMBLER, LINKER, AND LIBRARIAN GUIDE

Provides access to an online version of the *AVR IAR Assembler, IAR XLINK Linker™, and IAR XLIB Librarian™ Reference Guide*, available in Acrobat® Reader format.

C LIBRARY REFERENCE GUIDE

Provides access to the C library documentation, which is available in Acrobat® Reader format.

EC++ LIBRARY REFERENCE GUIDE

Provides access to the EC++ library documentation, which is available in `html` format.

IAR ON THE WEB

Allows you to browse the home page, news page, and FAQ (frequently asked questions) page of the IAR website, and to contact IAR Technical Support.

ABOUT...

Displays the version numbers of the user interface and of the AVR IAR Embedded Workbench.

PART 4: THE C-SPY SIMULATOR

This part of the AVR IAR Embedded Workbench™ User Guide contains the following chapters:

- *Introduction to C-SPY*
- *C-SPY expressions*
- *C-SPY macros*
- *C-SPY reference*
- *C-SPY command line options*.

INTRODUCTION TO C-SPY

The IAR C-SPY Debugger is a powerful interactive debugger for embedded applications.

This chapter introduces the IAR C-SPY® Debugger and gives an overview of the features it provides.

Note: For information about the C-SPY options available in the IAR Embedded Workbench, see the chapter *C-SPY options* in *Part 3: The IAR Embedded Workbench*. For information about the available command line options, see the chapter *C-SPY command line options*.

DEBUGGING PROJECTS

DISASSEMBLY AND SOURCE MODE DEBUGGING

C-SPY allows you to switch between C or assembler source and disassembly mode debugging as required.

Wherever source code is available, the source mode debugging displays the source program, and you can execute the program one statement at a time while monitoring the values of variables and data structures. Source mode debugging provides the quickest and easiest way of developing your application, without having to worry about how the compiler or assembler has implemented the code.

Disassembly mode debugging displays a mnemonic assembler listing of your program based on actual memory contents rather than source code, and lets you execute the program exactly one assembler instruction at a time. Disassembly mode debugging lets you focus on the critical sections of your application, and provides you with precise control over the simulated hardware.

During both source and disassembly mode debugging you can display the registers and memory, and change their contents.

If the file that you are debugging changes on the disk, you will be prompted to reload the file.

Source window
As you debug an application the source or disassembly source is displayed in a Source window, with the next source or disassembly statement to be executed highlighted.

You can navigate quickly to a particular file or function in the source code by selecting its name from the file or function box at the top of the Source window.

For convenience the Source window uses colors and text styles to identify key elements of the syntax. For example, by default C keywords are displayed in bold and constants in red. However, the colors and font styles are fully configurable, so that you can change them to whatever you find most convenient. See *Window Settings*, page 235 for additional information.

PROGRAM EXECUTION

C-SPY provides a flexible range of options for executing the target program.

The **Go** command continues execution from the current position until a breakpoint or program exit is reached. You can also execute up to a selected point in the program, without having to set a breakpoint, with the **Go to Cursor** command. Alternatively, you can execute out of a C function with the **Go Out** command.

Program execution is indicated by a flashing **Stop** command button in the debug toolbar. While the program is executing you may stop it either by clicking on the **Stop** button or by pressing the Escape key. You may also use any command accelerator associated with the **Stop** command.

Single stepping

The **Step** and **Step Into** commands allow you to execute the program a statement or instruction at a time. **Step Into** continues stepping inside function or subroutine calls whereas **Step** executes each function call in a single step.

The **Autostep** command steps repeatedly, and the **Multi Step** command lets you execute a specified number of steps before stopping.

Breakpoints

You can set breakpoints in the program being debugged using the **Toggle Breakpoint** command. Statements or instructions at which breakpoints are set are shown highlighted in the Source window listing.

Alternatively the **Edit Breakpoints** command allows you to define and alter complex breakpoints, including break conditions. You can optionally specify a macro which will perform certain actions when the breakpoint is encountered.

For detailed information, see *Edit Breakpoints…*, page 225.

Interrupt simulation
C-SPY includes an interrupt system allowing you to optionally simulate the execution of interrupts when debugging with C-SPY. The interrupt system can be turned on or off as required either with a system macro, or using the **Interrupt** dialog box. The interrupt system is activated by default, but if it is not required it can be turned off to speed up instruction set simulation.

The interrupt system has the following features:

- Interrupts, single or periodical, can be set up so that they are generated based on the cycle counter value.
- C-SPY provides interrupt support suitable for the AVR processor variants.
- By combining an interrupt with a data breakpoint you can simulate peripheral devices, such as a serial port.

C function information
C-SPY keeps track of the active functions and their local variables, and a list of the function calls can be displayed in the Calls window. You can also trace functions during program execution using the **Trace** command and tracing information is displayed in the Report window. For additional information, see *Trace*, page 234.

You can use the **Quick Watch** command to examine the value of any local, global, or static variable that is in scope. You can monitor the value of a macro, variable, or expression in the Watch window as you step through the program.

Viewing and editing memory and registers
You can display the contents of the processor registers in the Register window, and specified areas of memory in the Memory window.

The Register window allows you to edit the content of any register, and the register is automatically updated to reflect the change.

The Memory window can display the contents of memory in groups of 8, 16, or 32 bits, and you can double-click any memory address to edit the contents of memory at that address.

Terminal I/O
C-SPY can simulate terminal input and output using the Terminal I/O window.

Macro language
C-SPY includes a powerful internal macro language, to allow you to define complex sets of actions to be performed; for example, calculating the stack depth or when breakpoints are encountered. The macro language includes conditional and loop constructs, and you can use variables and expressions.

Tutorial 3, page 53, shows how C-SPY macros can be used.

Profiling
The profiling tool provides you with timing information on your application. This is useful for identifying the most time-consuming parts of the code and optimizing your program.

Code coverage
The code coverage tool can be used for identifying unused code in an application, as well as providing you with code coverage status at different stages during execution.

C-SPY EXPRESSIONS

In addition to C symbols defined in your program, C-SPY® allows you to define C-SPY variables and macros, and to use them when evaluating expressions. Expressions that are built with these components are called *C-SPY expressions* and can be used in the Watch and QuickWatch windows and in C-SPY macros.

This chapter describes the syntax of C-SPY expressions.

EXPRESSION SYNTAX

C-SPY expressions can include any type of C expression, except function calls. The following types of symbols can be used in expressions:

- C symbols.
- Assembler symbols; i.e. CPU register names and assembler labels.
- C-SPY variables and C-SPY macros; see the chapter *C-SPY macros*.

C SYMBOLS

C symbols can be referenced by their names or using an extended C-SPY format which allows you to reference symbols outside the current scope.

Expression	*What it means*
`i`	C variable `i` in the current scope or C-SPY variable `i`.
`\i`	C variable `i` in the current function.
`\func\i`	C variable `i` in the function `func`.
`mod\func\i`	C variable `i` in the function `func` in the module `mod`.

Note: When using the module name to reference a C symbol, the module name must be a valid C identifier or it must be encapsulated in backquotes ` (ASCII character 0x60), for example:

```
nice_module_name\func\i
`very strange () module + - name`\func\i
```

In case of a name conflict, C-SPY variables have a higher precedence than C variables. Extended C-SPY format can be used for solving such ambiguities.

Examples of valid C-SPY expressions are:

```
i = my_var * my_mac() + #asm_label
another_mac(2, my_var)
mac_var = another_module\another_func\my_var
```

ASSEMBLER SYMBOLS

Assembler symbols can be used in C expressions if they are preceded by #. These symbols can be assembler labels or CPU register names.

Example	*What it does*
`#pc++`	Increments the value of the program counter.
`myptr = #main`	Sets `myptr` to point to label `main`.

In case of a name conflict between a hardware register and an assembler label, hardware registers have a higher precedence. To refer to an assembler label in such a case, you must encapsulate the label in backquotes ` (ASCII character 0x60). For example:

Example	*What it does*
`#pc`	Refers to program counter.
#`pc`	Refers to assembler label `pc`.

The following processor-specific symbols are available:

Symbol	*Description*	*Size*
`PC`	Program Counter	Depends on the processor variant.
`R0-R31`	General purpose registers	8 bits
`X`	`R27`, `R26`	16 bits
`Y`	`R29`, `R28`	16 bits
`Z`	`R31`, `R30`	16 bits
`SREG`	Status Register	8 bits
`SP`	Stack pointer	16 bits
`CYCLES`	Cycle Counter	32 bits

FORMAT SPECIFIERS

The following format specifiers can be used in the **Display Format** drop-down list in the **Symbol Properties** dialog box (see *Inspecting expression properties*, page 215) and in a macro `message` statement:

Macro message specifier	*Description*
`%b`	Binary format
`%c`	Char format
`%o`	Unsigned octal format
`%s`	String format
`%x`	Unsigned hexadecimal format
`%X`	Unsigned hexadecimal format (capital letters)

The precision for the default float format is seven or 15 decimals for four and eight byte floats.

Strings with format `%s` are printed in quotation marks. If no `NULL` character (`'\0'`) is found within 1000 characters, the printout will stop without a final quotation mark.

C-SPY MACROS

The IAR C-SPY® Debugger provides comprehensive macro capabilities allowing you to automate the debugging process and to simulate peripheral devices. Macros can be used in conjunction with complex breakpoints and interrupt simulation to perform a wide variety of tasks. The C-SPY macros work in a way similar to the C or Embedded C++ functions, and for your convenience they follow the C or Embedded C++ language naming conventions and statement syntax as closely as possible.

This chapter first describes how to use the C-SPY macros. It then describes the C-SPY *setup macros*. Finally, there is complete reference information for each built-in *system macro* provided with C-SPY.

Tutorial 3 in the chapter *Compiler tutorials* gives an example of how the C-SPY macros can be used.

USING C-SPY MACROS

C-SPY allows you to define both *macro variables* (global or local) and *macro functions*. In addition, several predefined system macro variables and macro functions are provided; they return information about the system status, and perform complex tasks such as opening and closing files, and file I/O operations. System macro names start with double underscore and are reserved names.

Note: To view the available macros, select **Load macro...** from the **Options** menu. The available macros will be displayed in the **Macro Files** dialog box, under **Registered Macros**. You can select whether to view all macros, the predefined system macros, or the user-defined macros. For more information, see *Load Macro…*, page 239.

Defining macros

To define a macro variable or macro function, you should first create a text file containing its definition. You can use any suitable text editor, such as the IAR Embedded Workbench™ Editor. Then you should register the macro file. There are several ways to do this:

- You can register a macro by choosing **Load Macro…** from the **Options** menu. For more information, see *Load Macro…*, page 239.

- In the IAR Embedded Workbench, you can specify which setup file to use with a project. See *Setting C-SPY options*, page 135, for more information.
- When starting C-SPY with the Windows **Run...** command, you can use the `-f` command line option to specify the setup file. See *Setting C-SPY options from the command line*, page 243, for more information.
- Macros can also be registered using the system macro `__registerMacroFile`. This macro allows you to register macro files from other macros. This means that you can dynamically select which macro files to register, depending on the run-time conditions. For more information, see *__registerMacroFile*, page 201.

Executing C-SPY macros
You can assign values to a macro variable, or execute a macro function, using the **Quick Watch...** command on the **Control** menu, or from within another C-SPY macro including setup macros. For details of the setup macros, see *C-SPY setup macros*, page 188.

A macro can also be executed if it is associated with a breakpoint that is activated.

MACRO VARIABLES

A macro variable is a variable defined and allocated outside the user program space. It can then be used in a C-SPY expression.

The command to define one or more macro variables has the following form:

```
var nameList;
```

where *nameList* is a list of C-SPY variable names separated by commas.

A macro variable defined outside a macro body has global scope, and retains its value and type through the whole debugging session. A macro variable defined within a macro body is created when its definition is executed and deallocated on return from the macro.

By default a macro variable is initialized to signed integer 0. When a C-SPY variable is assigned a value in an expression its type is also converted to the type of the operand.

For example:

Expression	*What it means*
`myvar = 3.5`	`myvar` is now type `float`, value `3.5`.
`myvar = (int*)i`	`myvar` is now type pointer to `int`, and the value is the same as `i`.

A complex type (`struct` or `union`) cannot be assigned to a macro variable but a macro variable can contain an address to such an object.

MACRO FUNCTIONS

C-SPY macro functions consist of a series of C-SPY variable definitions and macro statements which are executed when the macro is called. An unlimited number of parameters can be passed to a macro, and macros can return a value on exit.

A C-SPY macro has the following form:

```
macroName (parameterList)
{
  macroBody
}
```

where *`parameterList`* is a list of macro formal names separated by commas, and *`macroBody`* is any series of C-SPY variable definitions and C-SPY statements.

Type checking is not performed on the values passed to the macro parameters. When an `array`, `struct`, or `union` is passed, only its address is passed.

MACRO STATEMENTS

The following C-SPY macro statements are accepted:

Expressions

```
expression;
```

Conditional statements

```
if (expression)
  statement
```

```
if (expression)
  statement
```

```
else
  statement
```

Loop statements

```
for (init_expression; cond_expression; after_expression)
  statement
```

```
while (expression)
  statement
```

```
do
  statement
while (expression);
```

Return statements

```
return;
```

```
return (expression);
```

If the return value is not explicitly set by default, `signed int 0` is returned.

Blocks

```
{
  statement1
  statement2
  .
  .
  .
  statementN
}
```

In the above example, *expression* means a C-SPY expression; statements are expected to behave in the same way as corresponding C statements would do.

Printing messages

The `message` statement allows you to print messages while executing a macro. Its definition is as follows:

```
message argList;
```

where *argList* is a list of C-SPY expressions or strings separated by commas. The value of expression arguments or strings are printed to the Report window.

It is possible to override the default display format of an element in *argList* by suffixing it with a : followed by a format specifier, for example:

```
message int1:%X, int2;
```

This will print `int1` in hexadecimal format and `int2` in default format (decimal for an integer type).

Resume statement
The `resume` statement allows you to resume execution of a program after a breakpoint is encountered. For example, specifying:

```
resume;
```

in a breakpoint macro will resume execution after the breakpoint.

Error handling in macros
Two types of errors can occur while a macro is being executed:

- Stop errors, which stop execution.
- Minor errors, which cause the macro to return an error number.

Stop errors are caused by mismatched macro parameter types, missing parameters, illegal addresses when setting a breakpoint or map, or illegal interrupt vectors when setting up an interrupt. They are handled by the C-SPY error handler, and execution stops with an appropriate error message.

Minor errors are caused by actions such as failing to open a file, or cancelling a non-existing interrupt. You can test for minor errors by checking the value returned by the system macro; zero indicates successful execution, any other value is a C-SPY error number.

C-SPY SETUP MACROS

The setup macros are reserved macro names that will be called by C-SPY at specific stages during execution. To use them you should create and register a macro with the name specified in the following table:

Macro	*Description*
`execUserExit()`	Called each time the program is about to exit. Implement this macro to save status data, etc.
`execUserInit()`	Called before communication with the target system is established. If you have not already chosen the processor option using the IAR Embedded Workbench or the C-SPY command line option, you can use this macro. It can also be used for performing other initialization, for example, port initialization for the emulator and ROM-monitor variants. Notice that since there is still no code loaded, you cannot, for example, set a breakpoint from this macro.
`execUserPreload()`	Called after communication with the target system is established but before downloading the target program. Implement this macro to initialize memory locations and/or registers which are vital for loading data properly.
`execUserReset()`	Called each time the reset command is issued. Implement this macro to set up and restore data.
`execUserSetup()`	Called once after the target program is downloaded. Implement this macro to set up the memory map, breakpoints, interrupts, register macro files, etc.
`execUserTrace()`	Called each time C-SPY issues a trace printout (when the `Trace` command is active).

The following sections provide reference information for each of the C-SPY system macros.

__autoStep

Steps continuously, with selectable time delay, until a breakpoint or the program exit is detected.

SYNTAX

```
__autoStep(delay)
```

PARAMETERS

delay	Delay between steps in tenth of a second (integer)

RETURN VALUE

```
int 0
```

EXAMPLE

```
__autoStep(12);
```

For additional information, see *Autostep...*, page 223.

__calls

Toggles calls mode on or off.

SYNTAX

```
__calls(what)
```

PARAMETERS

what	Predefined string, one of:
	`"ON"` turns calls mode on
	`"OFF"` turns calls mode off

RETURN VALUE

```
int 0
```

EXAMPLE

```
__calls("ON");
```

For additional information, see *Calls window*, page 214.

__cancelAllInterrupts

Cancels all ordered interrupts.

SYNTAX

```
__cancelAllInterrupts()
```

RETURN VALUE

```
int 0
```

EXAMPLE

```
__cancelAllInterrupts();
```

For additional information, see *Interrupt…*, page 231.

__cancelInterrupt

Cancels an interrupt.

SYNTAX

```
__cancelInterrupt(interrupt_id)
```

PARAMETERS

interrupt_id	The value returned by the corresponding `__orderInterrupt` macro call (unsigned long)

RETURN VALUE

Result	*Value*
Successful	`int 0`
Unsuccessful	Non-zero error number

EXAMPLE

```
__cancelInterrupt(interrupt_id)
```

For additional information, see *Interrupt…*, page 231.

__clearAllBreaks

Clears all user-defined breakpoints.

SYNTAX

```
__clearAllBreaks()
```

RETURN VALUE

```
int 0
```

EXAMPLE

```
__clearAllBreaks();
```

For additional information, see *Edit Breakpoints...*, page 225.

__clearAllMaps

Clears all user-defined memory mappings.

SYNTAX

```
__clearAllMaps()
```

RETURN VALUE

```
int 0
```

EXAMPLE

```
__clearAllMaps();
```

For additional information, see *Memory Map...*, page 228.

__clearBreak

Clears a given breakpoint.

SYNTAX

```
__clearBreak(address, segment, access)
```

PARAMETERS

address	The breakpoint location (string)
segment	The memory segment name (string), one of: `CODE`, `DATA`, `EEPROM`, and `I/O-SPACE`

access The memory access type (string); concatenation of any of "R", "W", "F", "I", or "O":

Type	*Description*
R	Read
W	Write
F	Fetch
I	Read immediate
O	Write immediate

RETURN VALUE

Result	*Value*
Successful	int 0
Unsuccessful	Non-zero error number

EXAMPLE

The following example shows how a line in the source file is specified as *address*:

```
__clearBreak(".demo\\12", "CODE", "F");
```

The following example shows how *address* is specified in hexadecimal notation:

```
__clearBreak("0x1300", "CODE", "F");
```

For additional information, see *Edit Breakpoints...*, page 225.

__clearMap

Clears a given memory mapping.

SYNTAX

```
__clearMap(address, segment)
```

PARAMETERS

address The address (integer)

segment	The memory segment name (string), one of: CODE, DATA, EEPROM, and I/O-SPACE

RETURN VALUE

Result	*Value*
Successful	int 0
Unsuccessful	Non-zero error number

EXAMPLE

```
__clearMap(0x0040, "DATA");
```

For additional information, see *Memory Map…*, page 228.

__closeFile

Closes a file previously opened by __openFile.

SYNTAX

```
__closeFile(filehandle)
```

PARAMETERS

filehandle	The macro variable used as filehandle by the __openFile macro

RETURN VALUE

int 0.

EXAMPLE

```
__closeFile(filehandle);
```

__disableInterrupts

Disables the generation of interrupts.

SYNTAX

```
__disableInterrupts()
```

RETURN VALUE

Result	*Value*
Successful	`int 0`
Unsuccessful	Non-zero error number

EXAMPLE

```
__disableInterrupts();
```

For additional information, see *Interrupt…*, page 231.

__enableInterrupts

Enables the generation of interrupts.

SYNTAX

```
__enableInterrupts()
```

RETURN VALUE

Result	*Value*
Successful	`int 0`
Unsuccessful	Non-zero error number

EXAMPLE

```
__enableInterrupts();
```

For additional information, see *Interrupt…*, page 231.

__getLastMacroError

Returns the last macro error code (excluding stop errors).

SYNTAX

```
__getLastMacroError()
```

RETURN VALUE

Value of the last system macro error code.

EXAMPLE

```
__getLastMacroError();
```

__go

Starts execution.

SYNTAX

```
__go()
```

RETURN VALUE

```
int 0
```

EXAMPLE

```
__go();
```

For additional information, see *Go*, page 223.

__multiStep

Executes a sequence of steps.

SYNTAX

```
__multiStep(kindOf, noOfSteps)
```

PARAMETERS

kindOf	Predefined string, one of:
	`"OVER"` does not enter C or Embedded C++ functions or assembler subroutines
	`"INTO"` enters C or Embedded C++ functions or assembler subroutines
noOfSteps	Number of steps to execute (integer)

RETURN VALUE

```
int 0
```

EXAMPLE

```
__multiStep("INTO", 12);
```

For additional information, see *Multi Step...*, page 223.

__openFile

Opens a file for I/O operations.

SYNTAX

```
__openFile(filehandle, filename, access)
```

PARAMETERS

filehandle	The macro variable to contain the file handle
filename	The filename as a string
access	The access type (string); one of the following: "r" ASCII read "rb" Binary read "w" ASCII write "wb" Binary write

RETURN VALUE

Result	*Value*
Successful	int 0
Unsuccessful	Non-zero error number

EXAMPLE

```
var filehandle;
__openFile(filehandle, "C:\\TESTDIR\\TEST.TST", "r");
```

__orderInterrupt

Generates an interrupt.

SYNTAX

```
__orderInterrupt(address, activation_time,
repeat_interval, jitter, latency, probability)
```

PARAMETERS

address	The interrupt vector (string)
activation_time	The activation time in cycles (integer)
repeat_interval	The periodicity in cycles (integer)

jitter	The timing variation range (integer between 0 and 100)
latency	The latency (integer)
probability	The probability in percent (integer between 0 and 100)

RETURN VALUE

The macro returns an interrupt identifier (`unsigned long`).

EXAMPLE

The following example generates a reset after 5000 cycles:

```
__orderInterrupt ("0", 5000, 0, 50, 0, 75);
```

For additional information, see *Interrupt…*, page 231.

__printLastMacroError

Prints the last system macro error message (excluding stop errors) to the Report window.

SYNTAX

```
__printLastMacroError()
```

RETURN VALUE

```
int 0
```

EXAMPLE

```
__printLastMacroError();
```

__processorOption

Sets a given processor option.

SYNTAX

```
__processorOption(procOption)
```

PARAMETERS

procOption	The processor option given in the same way it would have been given on the command line (string)

RETURN VALUE

`int 0`

DESCRIPTION

This macro can only be called from the `execUserInit()` macro.

For additional information, see *Processor option (-v)*, page 245.

EXAMPLE

```
__processorOption("-v2");
```

__readFile

Reads from a file.

SYNTAX

`__readFile(`*filehandle*`)`

PARAMETERS

filehandle	The macro variable used as the filehandle by the `__openFile` macro

RETURN VALUE

The return value depends on the access type of the file.

In ASCII mode a series of hex digits, delimited by space, are read and converted to an `unsigned long`, which is returned by the macro.

In binary mode one byte is read and returned.

DESCRIPTION

When the end of the file is reached, the file is rewound and a message is printed in the Report window. For additional information, see *Report window*, page 217.

EXAMPLE

Assuming a file was opened with `r` access type containing the following data:

```
1234 56 78
```

Calls to `__readFile()` would return the numeric values `0x1234`, `0x56`, and `0x78`.

__readFileGuarded

Reads from a file.

SYNTAX

```
__readFileGuarded(filehandle, errorstatus)
```

PARAMETERS

filehandle	The macro variable used as the file handle by the `__openFile` macro
errorstatus	A C-SPY variable to contain the error status

RETURN VALUE

Result	*Value*
Successful	The value read
Unsuccessful	`-1L`

DESCRIPTION

This macro works in exactly the same way as `__readFile`, except that when the end of the file is encountered `-1L` is returned, and the value of *errorstatus* is set to the corresponding error number.

EXAMPLE

```
__readFileGuarded(filehandle, errorstatus)
```

__readMemoryByte

Reads one byte from a given memory location.

SYNTAX

```
__readMemoryByte(address, segment)
```

PARAMETERS

address	The memory address (integer)
segment	The memory segment name (string), one of: `CODE`, `DATA`, `EEPROM`, and `I/O-SPACE`

RETURN VALUE

The macro returns the value from memory.

EXAMPLE

```
__readMemoryByte(0x0108, "DATA");
```

__realtime

Toggles real-time mode on or off. Notice that real-time mode only applies to the emulator and ROM-monitor versions of C-SPY.

SYNTAX

```
__realtime(what)
```

PARAMETERS

what	Predefined string, one of:
	`"ON"` turns real-time mode on
	`"OFF"` turns real-time mode off

RETURN VALUE

`int 0`

EXAMPLE

```
__realtime("OFF");
```

For additional information, see *Realtime*, page 234.

__registerMacroFile

Registers macros from a specified macro file.

SYNTAX

```
__registerMacroFile(filename)
```

PARAMETERS

filename	A file containing the macros to be registered (string)

RETURN VALUE

```
int 0
```

EXAMPLE

```
__registerMacroFile("c://testdir//macro.mac");
```

For additional information, see *Load Macro...*, page 239.

__reset

Resets the target processor.

SYNTAX

```
__reset()
```

RETURN VALUE

```
int 0
```

EXAMPLE

```
__reset();
```

For additional information, see *Reset*, page 224.

__rewindFile

Rewinds the file previously opened by `__openFile`.

SYNTAX

```
__rewindFile(filehandle)
```

PARAMETERS

filehandle	The macro variable used as filehandle by the `__openFile` macro

RETURN VALUE

`int 0`

EXAMPLE

```
__rewindFile(filehandle);
```

__setBreak

Sets a given breakpoint.

SYNTAX

```
__setBreak(address, segment, length, count, condition,
cond_type, access, macro)
```

PARAMETERS

address	The address in memory or any expression that evaluates to a valid address, for example a function or variable name. A `.` (period) must precede a code breakpoint.
segment	The memory segment name (string), one of: `CODE`, `DATA`, `EEPROM`, and `I/O-SPACE`
length	The number of bytes to be covered by the breakpoint (integer)
count	The number of times that a breakpoint condition must be fulfilled before a break occurs (integer)
condition	The breakpoint condition (string)
cond_type	The condition type; either "`CHANGED`" or "`TRUE`" (string)

access The memory access type (string); concatenation of any of "R", "W", "F", "I", or "O".

Type	*Description*
R	Read
W	Write
F	Fetch
I	Read immediate
O	Write immediate

macro The expression to be executed after the breakpoint is accepted (string)

RETURN VALUE

Result	*Value*
Successful	0
Unsuccessful	Non-zero error number

EXAMPLES

The following example shows a *code* breakpoint:

```
__setBreak(".demo.c\\12", "CODE", 1, 3, "d>16", "TRUE",
"RF", "afterMacro ()");
```

The following example shows a *data* breakpoint:

```
__setBreak("0x32", "I/O-SPACE", 1, 1, "", "TRUE", "I",
"_readTCNT()");
```

For additional information, see *Edit Breakpoints...*, page 225.

__setMap

Sets a given memory mapping.

SYNTAX

```
__setMap(address, segment, length, type)
```

PARAMETERS

address	The start location (integer)
segment	The memory segment name (string), one of: `CODE`, `DATA`, `EEPROM`, and `I/O-SPACE`
length	The number of bytes to be covered by mapping (integer)
type	The memory mapping type; "`G`" (guarded) or "`P`" (protected) (string)

RETURN VALUE

`int 0`

EXAMPLE

```
__setMap(0x0500, "DATA", 1000, "G");
```

For additional information, see *Memory Map...*, page 228.

__step

Executes the next statement or instruction.

SYNTAX

```
__step(kindOf)
```

PARAMETERS

kindOf	Predefined string, one of: `"OVER"` does not enter C or Embedded C++ functions or assembler subroutines `"INTO"` enters C or Embedded C++ functions or assembler subroutines

RETURN VALUE

`int 0`

EXAMPLE

```
__step("OVER");
```

For additional information, see *Step*, page 223.

__writeFile

Writes to a file.

SYNTAX

```
__writeFile(filehandle, value)
```

PARAMETERS

filehandle	The macro variable used as the file handle set by the `__openFile` macro
value	The value to be written to the file. *value* is written using a format depending on with what access type the file was opened. In ASCII mode the value is written to the file as a string of hex digits corresponding to *value*. In binary mode the lowest byte of *value* is written as a binary byte

RETURN VALUE

`int 0`

EXAMPLE

```
__writeFile(filehandle, 123);
```

__writeMemoryByte

Writes one byte to a given memory location.

SYNTAX

```
__writeMemoryByte(value, address, segment)
```

PARAMETERS

value	The value to be written (integer)

address	The memory address (integer)
segment	The memory segment name (string), one of: CODE, DATA, EEPROM, and I/O-SPACE

RETURN VALUE

int 0

EXAMPLE

```
__writeMemoryByte(0x2F, 0x1F, "I/O-SPACE");
```

C-SPY REFERENCE

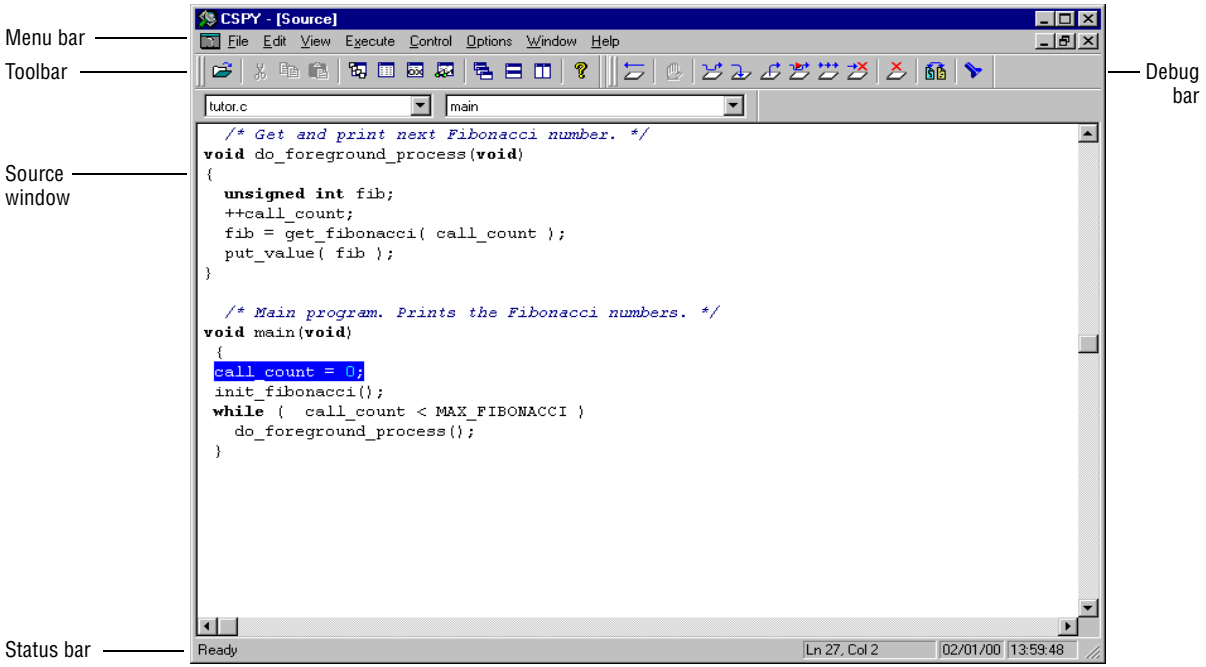
This chapter provides complete reference information about the IAR C-SPY® Debugger.

It first gives information about the components of the C-SPY window, and each of the different types of window it encloses.

It then gives details of the menus, and the commands on each menu.

THE C-SPY WINDOW

The following illustration shows the main C-SPY window:



TYPES OF C-SPY WINDOWS

The following windows are available in C-SPY:

- Source window
- Watch window
- Report window
- Register window
- SFR window
- Profiling window
- Terminal I/O window
- Locals window
- Memory window
- Calls window
- Code Coverage window.

These windows are described in greater detail on the following pages.

MENU BAR

Gives access to the C-SPY menus:

Menu	*Description*
File	The **File** menu provides commands for opening and closing files, and exiting from C-SPY.
Edit	The **Edit** menu provides commands for use with the Source window.
View	The **View** menu provides commands to allow you to select which windows are displayed in the C-SPY window.
Execute	The **Execute** menu provides commands for executing and debugging the source program. Most of the commands are also available as icon buttons in the debug bar.
Control	The **Control** menu provides commands allowing you to control the execution of the program.

Menu	*Description*
Options	The commands on the **Options** menu allow you to change the configuration of your C-SPY environment, register and display macros.
Window	The **Window** menu lets you select or open C-SPY windows and control the order and arrangement of the windows.
Help	The **Help** menu provides help about C-SPY.

The menus are described in greater detail on the following pages.

TOOLBAR AND DEBUG BAR

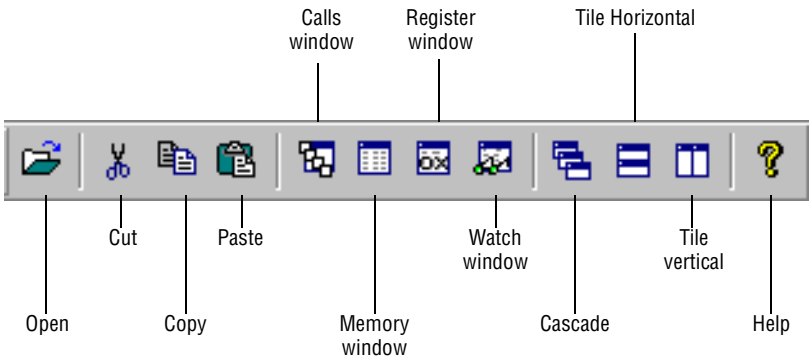
The toolbar and debug bar provide buttons for the most frequently-used commands on the menus.

You can move each bar to a different position in the C-SPY window, or convert it to a floating palette, by dragging it with the mouse.

You can display a description of any button by pointing to it with the mouse pointer. When a command is not available the corresponding button will be grayed out and you will not be able to select it.

Toolbar

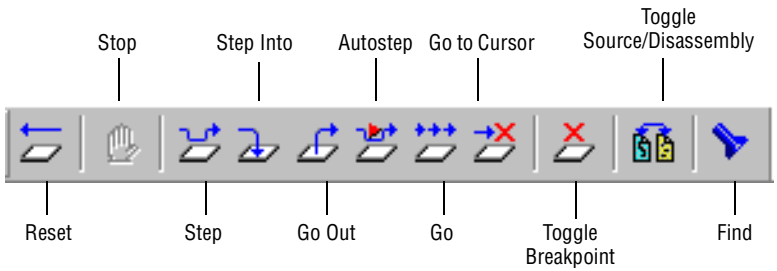
The following diagram shows the command corresponding to each of the toolbar buttons:



You can choose whether the toolbar is displayed by using the **Toolbar** command on the **View** menu.

Debug bar

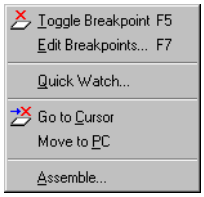
The following diagram shows the command corresponding to each button:



Use the **Debug Bar** command on the **View** menu to toggle the debug bar on and off.

SOURCE WINDOW

The C-SPY Source window shows the source program being debugged, as either C or assembler source code or disassembled program code. You can switch between source mode and disassembly mode by choosing **Toggle Source/Disassembly** from the **View** menu, or by clicking the **Toggle Source/Disassembly** button in the debug bar.



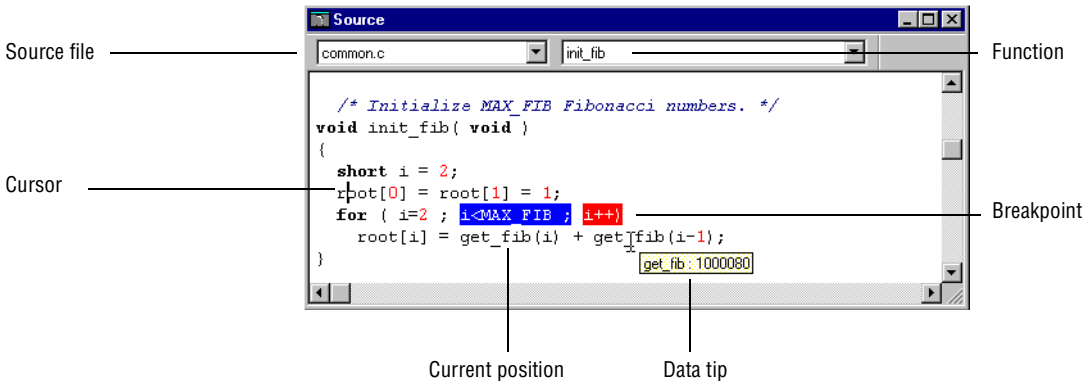
Clicking the right mouse button in the Source window displays a pop-up menu which gives you access to several useful commands.

When you start C-SPY, the first executable statement in the `main` function will be displayed in the Source window. If the Source window is initially blank, a `main` function has not been found and the program actually starts in a low-level assembler module—assembled without debug information—so there is no source code corresponding to this.

Source file and function

The **Source file** and **Function** boxes show the name of the current source file and function displayed in the Source window, and allow you to move to a different module or function by selecting a name from the corresponding drop-down list.

The following types of highlighting are used in the Source window:



Current position

The current position indicates the next C statement or assembler instruction to be executed, and is highlighted in blue.

Cursor

Any statement in the Source window can be selected by clicking on it with the mouse pointer. The selected statement is indicated by the cursor.

Alternatively, you can move the cursor using the navigation keys.

The **Go to Cursor** command on the **Execute** menu will execute the program from the current position up to the statement containing the cursor.

Breakpoint

C statements or assembler instructions at which breakpoints have been set are highlighted in the Source window.

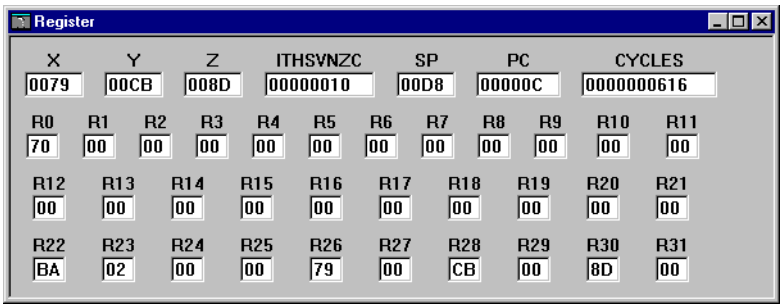
To set a breakpoint choose **Toggle Breakpoint...** from the **Control** menu or click the **Toggle Breakpoint** button in the toolbar.

Data tip

If you position the mouse pointer over a function, variable, or constant name in the C source shown in the Source window, the function start address or the current value of the variable or constant is shown below the mouse pointer.

REGISTER WINDOW

The Register window gives a continuously updated display of the contents of the processor registers, and allows you to edit them. When a value changes it becomes highlighted.



To change the contents of a register, edit the corresponding text box. The register will be updated when you tab to the next register or press Enter.

You can configure the registers displayed in the Register window using the **Register Setup** page in the **Settings** dialog box.

Note: If the contents of a register changes during execution, the change will be highlighted in this window.

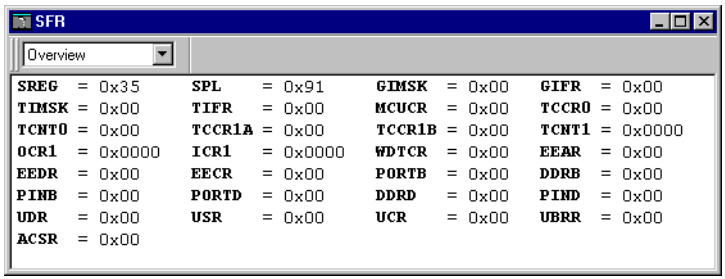
SFR WINDOW

The SFR window allows you to view and edit the contents of the special function registers.

In order to make this window available, you must specify a device description file (`ddf`) with SFR definitions when you start C-SPY. Use the option **Device description file** in the IAR Embedded Workbench as described on page 136, or the command line option `-p`, page 244, to specify the device description file.

The contents of the window is controlled by the SFR settings; see *Settings...*, page 235, for additional information.

Use the drop-down list to select which group of SFRs to display:



To edit the contents of an SFR, double-click its current value and type a new value. Then press Enter. The new contents will be highlighted, both in the SFR window and in the Memory window. If C-SPY during execution changes the memory or SFR contents, the change will be highlighted in this window.

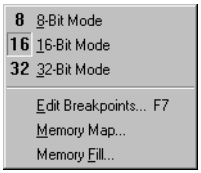
Note: The value `0x--` signifies a write-only register.

MEMORY WINDOW

The Memory window gives a continuously updated display of a specified block of memory and allows you to edit it.

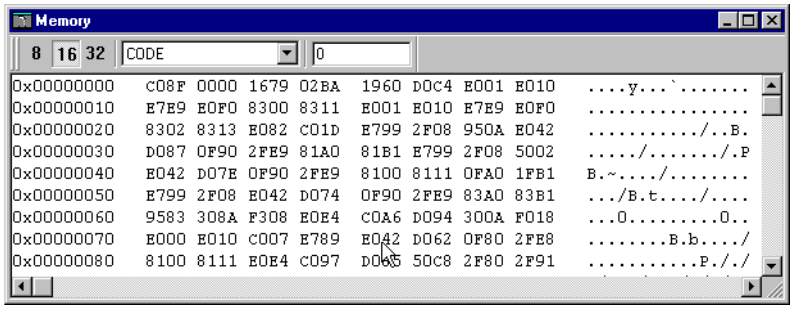
If C-SPY during execution changes the memory or SFR contents, the change will be highlighted in this window.

Choose **8**, **16**, or **32** to display the memory contents in blocks of 8, 16, or 32 bits. The available memory segments are called **CODE**, **DATA**, **EEPROM**, and **I/O-SPACE**.

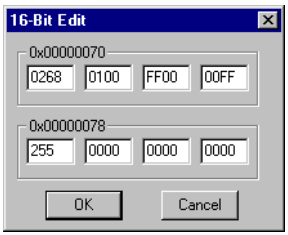


Clicking the right mouse button in the Memory window displays a pop-up menu which gives you access to several useful commands.

To edit the contents of memory, double-click the address or value you want to edit:



The following dialog box then allows you to edit the memory:

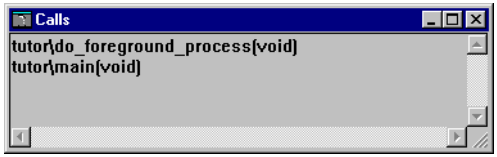


CALLS WINDOW

Displays the C call stack. Each entry has the format:

*module**function*(*values*)

where *values* is a list of the parameter values, or *void* if the function does not take any parameters.



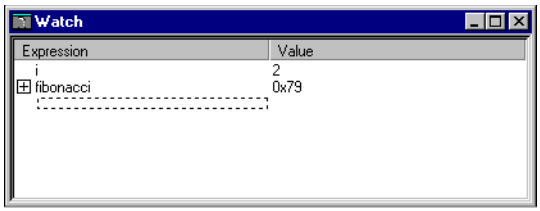
STATUS BAR

Shows help text, and the position of the cursor in the Source window.

Use the **Status Bar** command on the **View** menu to toggle the status bar on and off.

WATCH WINDOW

Allows you to monitor the values of C expressions or variables:



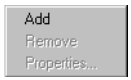
Viewing the contents of an expression

To view the contents of an expression such as an array, a structure or a union, click the plus sign icon to expand the tree structure.

Adding an expression to the Watch window

To add an expression to the Watch window, click in the dotted rectangle, then hold down and release the mouse button.

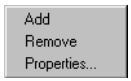
Alternatively, click the right mouse button in the Watch window and choose **Add** from the pop-up menu. Then type the expression and press Enter.

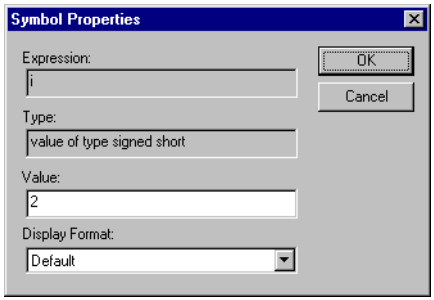


You can also drag and drop an expression from the Source window.

Inspecting expression properties

Select an expression in the Watch window and choose **Properties...** from the pop-up menu. You can then edit the value of the expression and change the display format in the **Symbol Properties** dialog box:





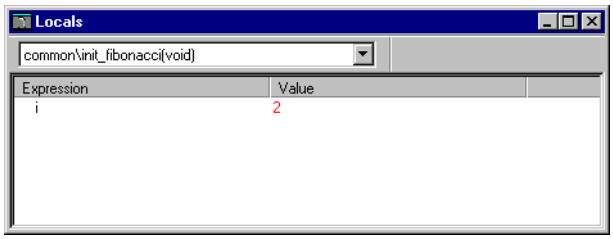
Removing an expression

Select the expression and press the Delete key, or choose **Remove** from the pop-up menu.

When a value changes it becomes highlighted.

LOCALS WINDOW

Automatically displays the local variables and their parameters:



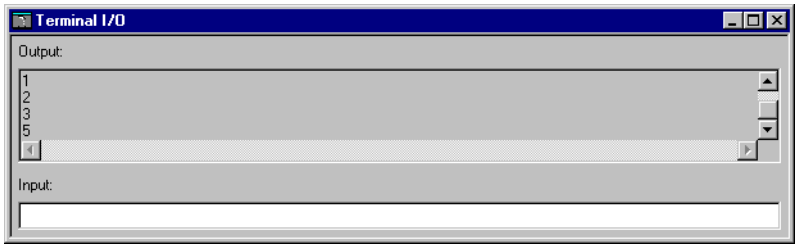
Editing the value of a local variable

To change the value of a local variable, click the right mouse button, and choose **Properties...** from the pop-up menu.

You can then change the value or display format in the **Symbol Properties** dialog box.

TERMINAL I/O WINDOW

Allows you to enter input to your program, and display output from it.



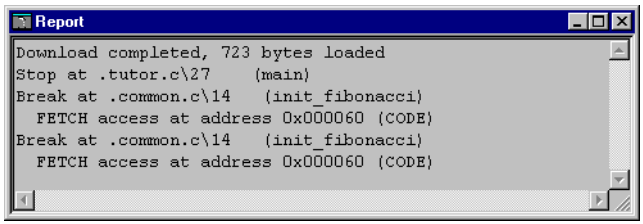
To use this window, you need to link the program with the option **Debug info with terminal I/O**. C-SPY will then direct `stdout` and `stderr` to this window. The window will only be available if your program uses the terminal I/O functions in the C library.

If the Terminal I/O window is open, C-SPY will write output to it, and read input from it.

If the Terminal I/O window is closed, C-SPY will open it automatically when input is required, but not for output.

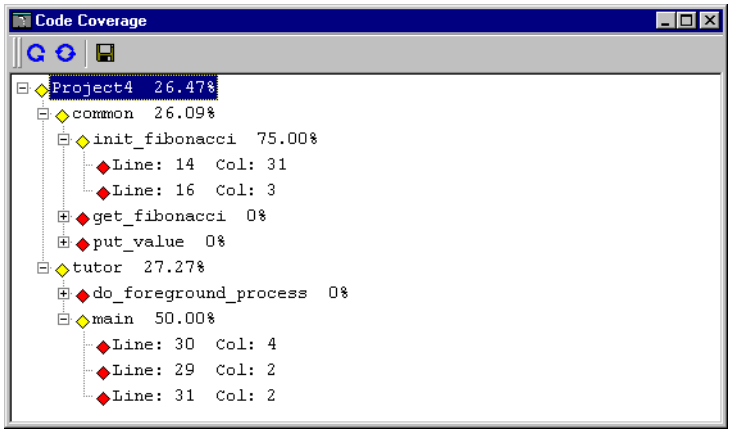
REPORT WINDOW

Displays debugger output, such as diagnostic messages and trace information.



CODE COVERAGE WINDOW

Reports the current code coverage status. The report includes all modules and functions and the statements that have not yet been executed.



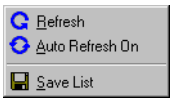
The code coverage information is displayed in a tree structure, showing the program, module, function, and statement levels. The plus sign and minus sign icons allow you to expand and collapse the structure.

The percentage displayed at the end of every line shows the amount of code that has been covered so far. In addition, the following colors are used for giving you an overview of the current status on all levels:

- Red signifies that 0 % of the code has been covered.
- Yellow signifies that some of the code has been covered.
- Green signifies that 100 % of the code has been covered.

When a statement has been executed, it is removed from this window.

When the contents becomes dimmed and an asterisk (*) appears in the title bar, this indicates that C-SPY has continued to execute and that the Code Coverage window needs to be refreshed because the displayed information is no longer up to date.



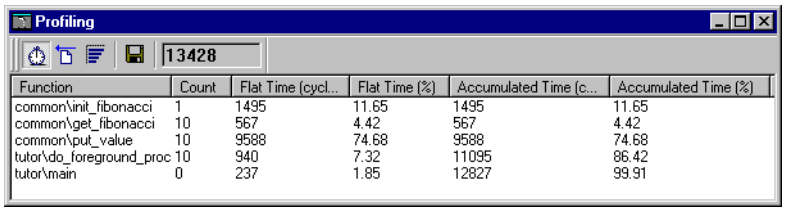
Clicking the right mouse button in the Code Coverage window displays a pop-up menu that gives you access to several useful commands.

Double-clicking on a statement line in the Code Coverage window displays that statement as current position in the Source window, which becomes the active window.

The code coverage information is reset when the processor is reset.

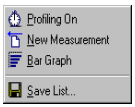
PROFILING WINDOW

Displays profiling information:



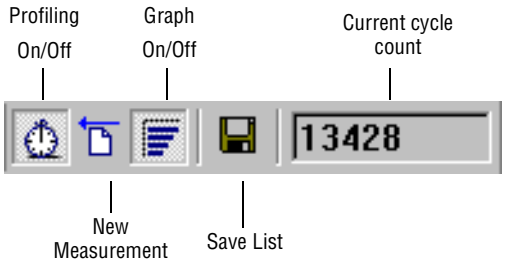
Clicking on the column header sorts the complete list depending on column. Double-clicking on an item in the **Function** column automatically displays the function in the Source window.

The information in the colums **Flat time** and **Accumulated time** can be displayed either as digits or as column diagrams. Flat time signifies the time in a function *excluding* child functions, while accumulated time signifies time in a function *including* its child functions.



Clicking the right mouse button in the Profiling window displays a pop-up menu which gives you access to several useful commands.

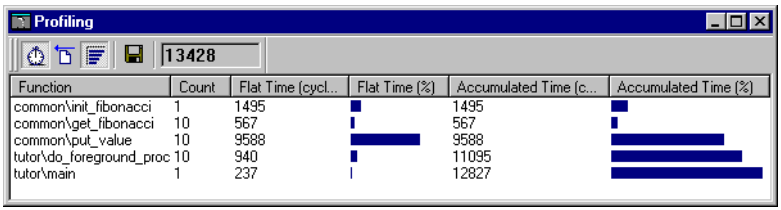
The following diagram shows the commands corresponding to the Profiling bar buttons:



Profiling On/Off
Switches profiling on and off during execution. Alternatively, use the **Profiling** command on the **Control** menu to toggle profiling on and off.

New Measurement
Starts a new measurement. By clicking on the icon the values displayed are reset to zero.

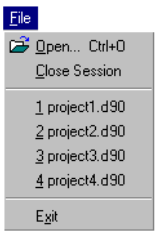
Graph On/Off
Displays the relative numbers as graph or numbers.



Save List
Saves list to file.

Current Cycle Count
Displays the current value of the cycle counter.

FILE MENU



The **File** menu provides commands for opening and closing files, and exiting from C-SPY.

OPEN...

Displays a standard **Open** dialog box to allow you to select a program file to debug.

If another file is already open it will be closed first.

CLOSE SESSION

Closes the current C-SPY session.

RECENT FILES

Displays a list of the files most recently opened, and allows you to select one to open it.

EXIT

Exits from C-SPY.

EDIT MENU

The **Edit** menu provides commands for use with the Source window.

UNDO, CUT, COPY, PASTE

Provides the usual Windows editing features for editing text in some of the windows and dialog boxes.

FIND...

Allows you to search for text in the Source window.

This dialog box allows you to specify the text to search for:

Enter the text you want to search for in the **Find What** text box.

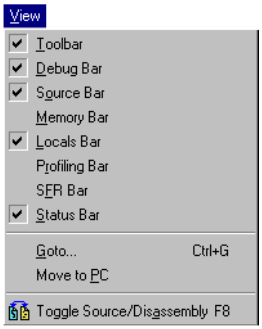
Select **Match Whole Word Only** to find the specified text only if it occurs as a separate word. Otherwise `int` will also find `print`, `sprintf` etc.

Select **Match Case** to find only occurrences that exactly match the case of the specified text. Otherwise specifying `int` will also find `INT` and `Int`.

Select **Up** or **Down** to specify the direction to search.

Choose **Find Next** to start searching. The source pointer will be moved to the next occurrence of the specified text.

VIEW MENU



The **View** menu provides commands to allow you to select which windows are displayed in the C-SPY window.

TOOLBAR

Toggles on or off the display of the toolbar.

DEBUG BAR

Toggles on or off the display of the debug bar.

SOURCE BAR

Toggles on or off the Source window toolbar.

MEMORY BAR

Toggles on or off the Memory window toolbar.

LOCALS BAR

Toggles on or off the Locals window toolbar.

PROFILING BAR

Toggles on or off the Profiling window toolbar.

SFR BAR

Toggles on or off the SFR window toolbar, provided that you have specified the device description file (`ddf`) to be used. This file contains information about the SFRs, such as I/O registers (SFR) definitions, vector, and control register definitions.

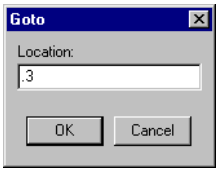
For additional information, see *Device description file*, page 136.

STATUS BAR

Toggles on or off the display of the status bar, along the bottom of the C-SPY window.

GOTO...

Displays the following dialog box to allow you to move the source pointer to a specified location in the Source window.



To go to a specified source line, prefix the line number with a period (`.`). For example:

Location	*Description*
`.12`	Moves to line 12 in current file.
`.tutor.c\12`	Moves to line 12 in file `tutor.c`.
`main`	Moves to function `main` in current scope.
`0x1000`	Moves to address `0x1000` (C-level debugging)
`1000`	Moves to address `1000` (assembly-level debugging)

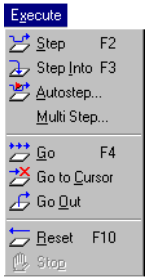
MOVE TO PC

Moves the source pointer to the current program counter (PC) position in the Source window.

TOGGLE SOURCE/DISASSEMBLY

Switches between source and disassembly mode debugging.

EXECUTE MENU



The **Execute** menu provides commands for executing and debugging the source program. Most of the commands are also available as icon buttons in the debug bar.

STEP

Executes the next statement or instruction, without entering C functions or assembler subroutines.

STEP INTO

Executes the next statement or instruction, entering C functions or assembler subroutines.

AUTOSTEP...

Steps continuously, with a selectable time delay, until a breakpoint or program exit is detected.

MULTI STEP...

Allows you to execute a specified number of **Step** or **Step Into** commands.

Displays the following dialog box to allow you to specify the number of steps:

Select **Over** to step over C functions or assembler subroutines, or **Into** to step into each C function or assembler subroutine.

Then choose **OK** to execute the steps.

GO

Executes from the current statement or instruction until a breakpoint or program exit is reached.

GO TO CURSOR

Executes from the current statement or instruction up to a selected statement or instruction.

GO OUT

Executes from the current statement up to the statement after the call to the current function.

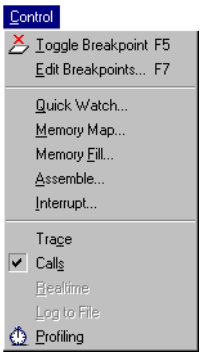
RESET

Resets the target processor.

STOP

Stops program execution or automatic stepping.

CONTROL MENU



The **Control** menu provides commands to allow you to define breakpoints and change the memory mapping.

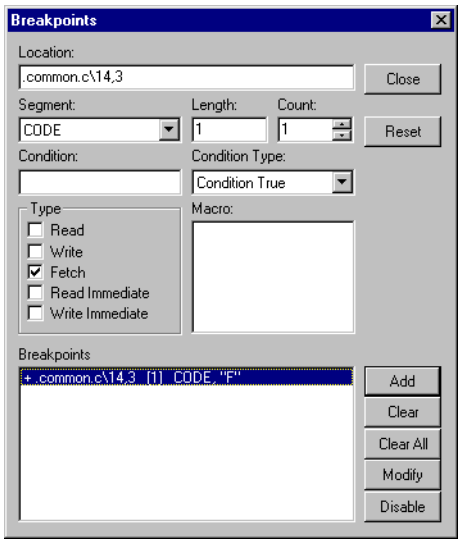
TOGGLE BREAKPOINT

Toggles on or off a breakpoint at the statement or instruction containing the cursor in the Source window.

This command is also available as an icon button in the debug bar.

EDIT BREAKPOINTS...

Displays the following dialog box which shows the currently defined breakpoints, and allows you to edit them or define new breakpoints:



This dialog box lists the breakpoints you have set with the **Toggle Breakpoint** command, and allows you to define, modify, or remove breakpoints with break conditions.

To define a new breakpoint, enter the characteristics of the breakpoint you want to define and choose **Add**.

To modify an existing breakpoint, select it in the **Breakpoints** list and choose one of the following buttons:

Choose this	*To do this*
Clear	Remove the selected breakpoint.
Clear All	Removes all the breakpoints in the list.
Modify	Modifies the breakpoint to the settings you select.
Disable/Enable	Toggles the breakpoint on or off. Enabled breakpoints are prefixed with a + in the **Breakpoints** list.

For each breakpoint you can define the following characteristics:

Location
The address in memory or any expression that evaluates to a valid address, for example a function or variable name.

When setting a *code* breakpoint, you can specify a location in the C source program with the formats `.source\line` or `.line`.

Notice that the line must start with a `.` (period) which indicates the code breakpoint. For example, `.common.c\12` sets a breakpoint at the first statement on line 12 in the source file `common.c`.

For an example of how to use a conditional code breakpoint, see *Defining complex breakpoints*, page 80.

When setting a *data* breakpoint, enter the name of a variable or any expression that evaluates to a valid memory location. For example, `my_var` refers to the location of the variable `my_var`, and `arr[3]` refers to the third element of the array `arr`.

For an example of how to use data breakpoints, see *Initializing the system* in Tutorial 3, page 58.

Note: You cannot set a breakpoint on a variable that does not have a constant address in memory.

Segment
The memory segment in which the location or address belongs. The valid segment names are **CODE**, **DATA**, **EEPROM**, and **I/O-SPACE**.

Length
The number of bytes to be guarded by the breakpoint.

Count
The number of times that the breakpoint condition must be fulfilled before a break takes place. Click **Reset** to reset this to 1.

Condition
A valid expression conforming to C-SPY expression syntax; see the chapter *C-SPY expressions*.

Condition type	*Description*
Condition True	The breakpoint is triggered if the value of the expression is true.

Condition type	*Description*
Condition Changed	The breakpoint is triggered if the value of the condition expression has changed.

Note: The condition is evaluated only when the breakpoint is encountered.

For an example of a breakpoint with a condition, see *Defining complex breakpoints* in Tutorial 6, page 80.

Type
Specifies the type of memory access guarded by the breakpoint:

Type	*Description*
Read	Read from location.
Write	Write to location.
Fetch	Fetch opcode from location.
Read Immediate	Read from location, immediate break.
Write Immediate	Write to location, immediate break.

The **Read**, **Write**, and **Fetch** breakpoint types never break execution within a single assembler instruction. **Read** and **Write** breakpoints are recorded and reported after the instruction is completed. If a **Fetch** breakpoint is detected on the first byte of an instruction, it will be reported before the instruction is executed; otherwise the breakpoint is reported after the instruction is completed.

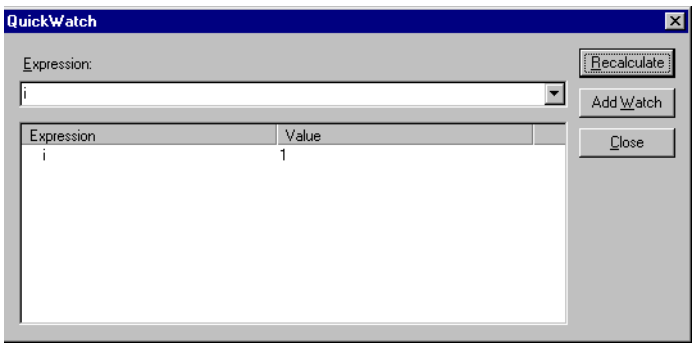
The **Read Immediate** and **Write Immediate** breakpoint types are only applicable to simulators and will cause a break as soon as encountered, even in the middle of executing an instruction. Execution will automatically continue, and the only action is to execute the associated macro. They are provided to allow you to simulate the behavior of a port. For example, you can set a **Read Immediate** breakpoint at a port address, and assign a macro to the breakpoint that reads a value from a file and writes it to the port location.

Macro
An expression to be executed once the breakpoint is activated.

QUICK WATCH...

Allows you to watch the value of a variable or expression and to execute macros.

Displays the following dialog box to allow you to specify the expression to watch:



Enter the C-SPY variable or expression you want to evaluate in the **Expression** box. Alternatively, you can select an expression you have previously watched from the drop-down list. For detailed information about C-SPY expressions, see *Expression syntax*, page 179.

Then choose **Recalculate** to evaluate the expression, or **Add Watch** to evaluate the expression and add it to the Watch window.

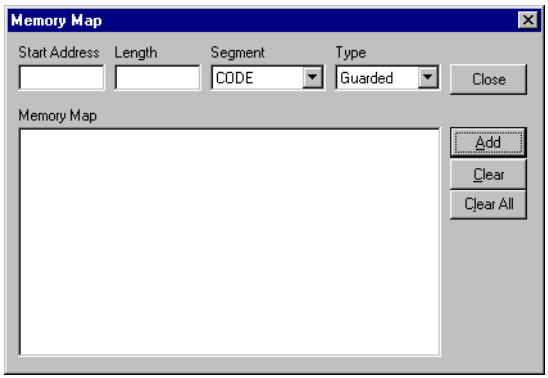
Choose **Close** to close the **Quick Watch** dialog box.

MEMORY MAP...

C-SPY allows the simulation of non-existing and read-only memory by the use of memory maps.

A memory map is a specified memory area with an access type attached to it, either no memory or read-only memory.

The **Memory Map** dialog box allows you to define memory maps:

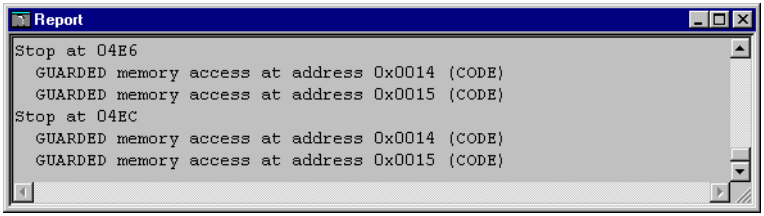


To define a new memory map, enter the **Start Address**, **Length**, and **Segment** and choose the **Type** according to the following table:

Type	*Description*
Guarded	Simulates addresses with no memory by flagging all accesses as illegal.
Protected	Simulates ROM memory by flagging all write accesses to this address as illegal.

To delete an existing memory map, select it in the **Memory Map** list and choose **Clear**.

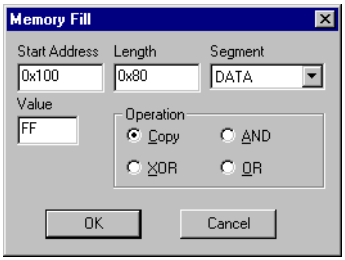
If a memory access occurs that violates the access type of that memory map, C-SPY will regard this access as illegal and display it in the Report window:



MEMORY FILL...

Allows you to fill a specified area of memory with a value.

The following dialog box is displayed to allow you to specify the area to fill:



Enter the **Start Address** and **Length** in hexadecimal notation, and select the segment type from the **Segment** drop-down list.

Enter the **Value** to be used for filling each memory location and select the logical operation. The default is **Copy**, but you may choose one of the following operations:

Operation	*Description*
Copy	The **Value** will be copied to the specified memory area.
AND	An AND operation will be performed between the **Value** and the existing contents of memory before writing the result to memory.
XOR	An XOR operation will be performed between the **Value** and the existing contents of memory before writing the result to memory.
OR	An OR operation will be performed between the **Value** and the existing contents of memory before writing the result to memory.

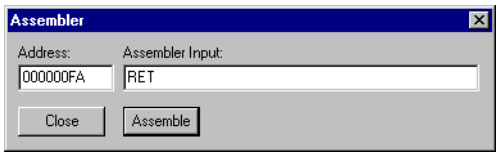
Finally choose **OK** to proceed with the memory fill.

ASSEMBLE...

Displays the assembler mnemonic for a machine-code instruction, and allows you to modify it and assemble it into memory.

Assemble... is only available in disassembly mode. If you are debugging in source mode, choose **Toggle Source/Disassembly** from the **View** menu to change mode.

Then double-click a line in the Source window, or position the cursor in the line and choose **Assemble...**. This dialog box shows the address and assembler instruction at that address:



To modify the instruction, edit the text in the **Assembler Input** field and click **Assemble**.

You can also enter an address in the **Address** field and then press Tab to display the assembler instruction at that address.

INTERRUPT...

The interrupt simulation can be used in conjunction with macros and complex breakpoints to simulate interrupt-driven ports. For example, to simulate port input, first specify an interrupt that will cause the appropriate interrupt handler to be called. Then set a breakpoint at the entry of the interrupt-handler routine, and associate it with a macro that sets up the input data by reading it from a file or by generating it using an appropriate algorithm.

Note: C-SPY only polls for interrupts between instructions, regardless of how many cycles an instruction takes.

The C-SPY interrupt system uses the cycle counter as a clock to determine when an interrupt should be raised in the simulator. Changing the cycle counter will affect any interrupts you have set up in the **Interrupt** dialog box.

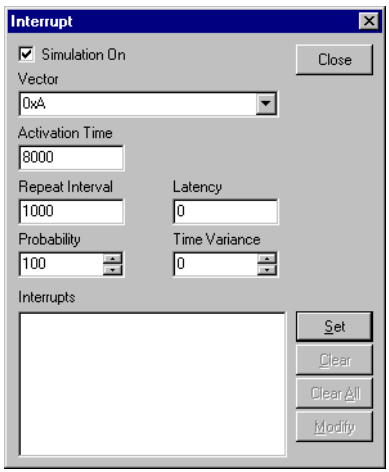
Performing a C-SPY reset will reset the cycle counter, and any interrupt orders with a fixed activation time will be cleared.

For example, consider the case where the cycle counter is 123456, a repeatable order raises an interrupt every 4000 cycles, and a single order is about to raise an interrupt at 123500 cycles.

After a system reset the repeatable interrupt order remains and will raise an interrupt every 4000 cycles, with the first interrupt at 4000 cycles. The single order is removed.

For an example where interrupts are used, see *Tutorial 3*, page 53.

The **Interrupt...** command displays the following dialog box to allow you to configure C-SPY's interrupt simulation:



To define a new interrupt enter the characteristics of the interrupt you want to simulate and choose **Set**.

To edit an existing interrupt select it in the **Interrupts** list and choose **Modify** to display or edit its characteristics, or **Clear** to delete it. Notice that deleting an interrupt does not remove any pending interrupt from the system.

For each interrupt you can define the following characteristics:

Vector
The interrupt vector table for a specific derivative can be defined in a device definitions file (`ddf`) which you specify using the C-SPY option **Use description file** in the IAR Embedded Workbench; see page 136 for additional information.

An interrupt vector can be specified in the entry box or selected from the drop-down list in the **Interrupt** dialog box.

Use the following syntax to specify a vector:

```
<vector number>
```

For example, to generate a reset:

```
0
```

Activation Time
The time, in cycles, after which the specified type of interrupt can be generated.

Repeat Interval
The periodicity of the interrupt in cycles.

Latency
Describes how long, in cycles, the interrupt remains pending until removed if it has not been processed. Latency is not implemented for the AVR microcontroller.

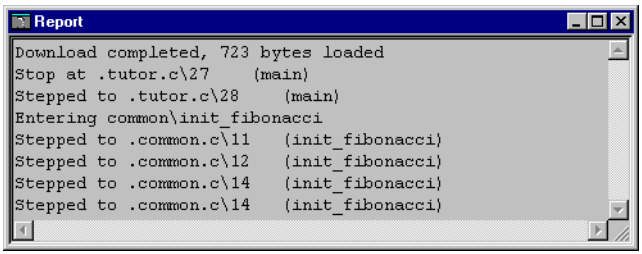
Probability
The probability, in percent, that the interrupt will actually appear in a period.

Time Variance
A timing variation range, as a percentage of the repeat interval, in which the interrupt may occur for a period. For example, if the repeat interval is 100 and the variance 5 %, the interrupt may occur anywhere between T = 95 and T = 105, to simulate a variation in the timing.

Simulation On/Off
Enables or disables interrupt simulation. If the interrupt is disabled the definition remains but no interrupts will be generated.

TRACE

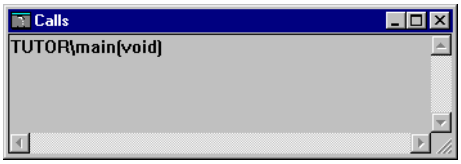
Toggles trace mode on or off. When trace mode is on, each step and function call is listed in the Report window:



Note: The trace information will be reduced if calls mode is off.

CALLS

Toggles calls mode on or off. Toggling calls mode off does not affect the recognition of the program exit breakpoint. When calls mode is on, the function calls are listed in the Calls window:



REALTIME

Reserved for the emulator and ROM-monitor versions of C-SPY.

LOG TO FILE

Toggles writing to the log file on or off. When log file mode is on, the contents of the Report window are logged to a file. Choose **Select Log File...** from the **Options** menu to enable the log file function.

PROFILING

Toggles profiling on or off. For further information regarding **Profiling**, see *Profiling window*, page 218.

OPTIONS MENU

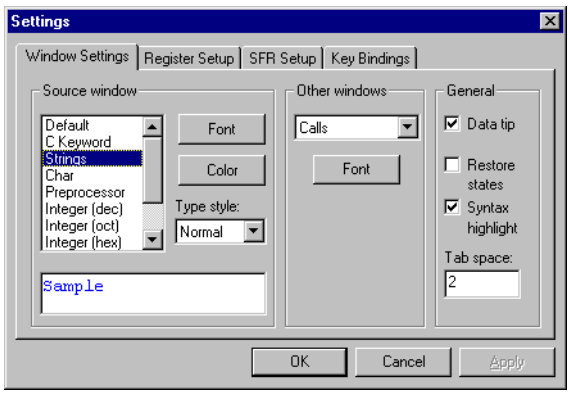
The commands on the **Options** menu allow you to change the configuration of your C-SPY environment, and register macros.

SETTINGS...

Displays the **Settings** dialog box to allow you to define the colors and fonts used in the windows, and to set up registers.

Window Settings

Allows you to specify the colors and fonts used for text in the Source window, the font used for text in other windows, and several general settings:



To specify the style used for each element of C syntax in the Source window, select the item you want to define from the **Source window** list. The current setting is shown by the **Sample** below the list box.

You can choose a text color by clicking **Color**, and a font by clicking **Font**. You can also choose the type style from the **Type Style** drop-down list.

To specify the font used in other windows, choose a window from the drop-down list and click **Font**.

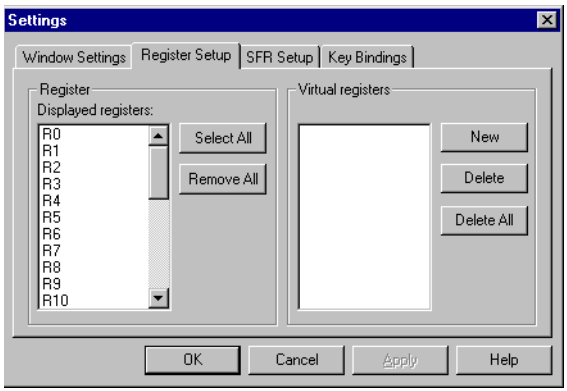
You can also specify the following general settings:

Settings	*Description*
Data tip	Shows the current value or function start address when the mouse pointer is moved over a variable constant, or function name in the Source window.

Settings	*Description*
Restore states	Restores breakpoints, memory maps, and interrupts between sessions.
Syntax highlight	Highlights the syntax of C programs in the Source window.
Tab space	Specifies the number of spaces used for expanding tabs.

Then choose **OK** to use the new styles you have defined, or **Cancel** to revert to the previous styles.

Register Setup
Allows you to specify which registers to be displayed in the Register window and to define virtual registers.

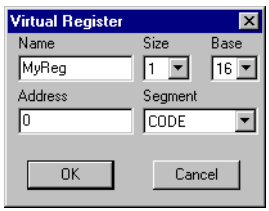


To specify which registers are displayed in the Register window, select them in the **Displayed Register** list and click **OK**.

Click **Select All** or **Remove All** to select or deselect all the registers.

The **Virtual registers** field allows you to specify any memory locations to be displayed in the Register window, in addition to the standard registers.

To define virtual registers, click **New** to display the **Virtual Register** dialog box:

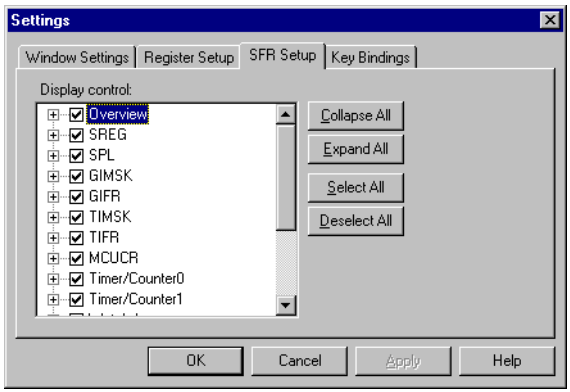


Enter the **Name** and **Address** for the virtual register, and select the **Size** in bytes, **Base**, and **Segment** from the drop-down lists. Then choose **OK** to define the register. It will be displayed in the Register window after the standard registers you have selected.

For information about the processor-specific symbols, see *Assembler symbols*, page 180.

SFR Setup

The special function registers are categorized into groups, and the **Display control** tree structure allows you to select the registers or groups of registers to be displayed in the SFR window (see *SFR window*, page 212, for additional information).



To view the contents of a group, click on its plus-sign icon or select the **Expand All** button to view the contents of all groups. To hide the contents of a group, click on its minus-sign icon or select the **Collapse All** button to hide the contents of all groups.

To select a register or a group of registers, click on its check box. When you select a group, all registers in that group become selected. If you want to deselect one or more registers from a selected group, first expand the group and then uncheck each register that should not be displayed in the SFR window.

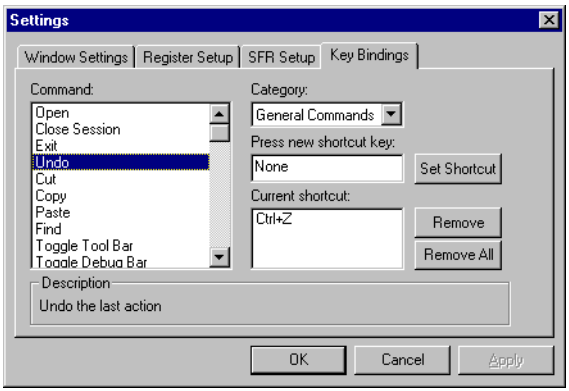
The **Select All** button selects all registers in all groups, and the **Deselect All** button deselects all groups and the registers in them.

Choose **OK** to set the display of SFRs.

Note: The **SFR Setup** page is available only if you have specified the device description file (`.ddf`) to be used. This file includes necessary information about the SFRs. For additional information, see *Device description file*, page 136.

Key Bindings

Displays the shortcut keys used for each of the menu options, and allows you to change them:



To define a shortcut key select the command category from the **Category** list, and then select the command you want to edit in the **Command** list. Any currently defined shortcut keys are shown in the **Current shortcut** list.

To add a shortcut key to the command click in the **Press new shortcut key** box and type the key combination you want to use. Then click **Set Shortcut** to add it to the **Current shortcut** list. You will not be allowed to add it if it is already used by another command.

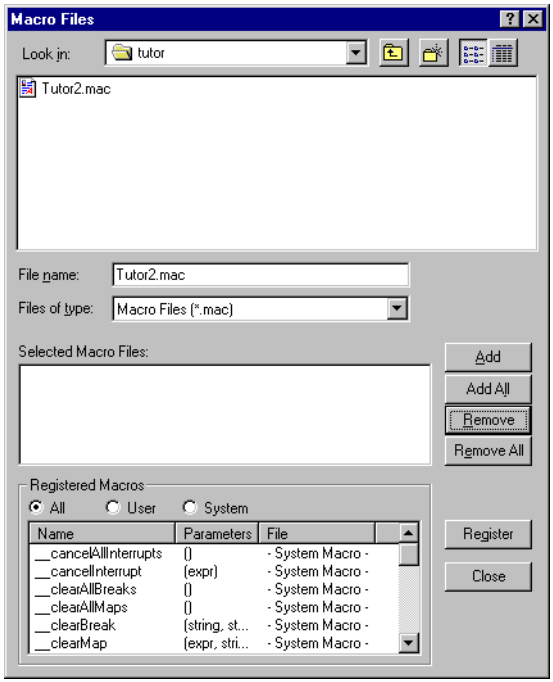
To remove a shortcut key select it in the **Current shortcut** list and click **Remove**, or click **Remove All** to remove all shortcut keys for a command.

Then choose **OK** to use the key bindings you have defined, and the menus will be updated to show the new shortcuts.

You can set up more than one shortcut for a command, but only one will be displayed in the menu.

LOAD MACRO...

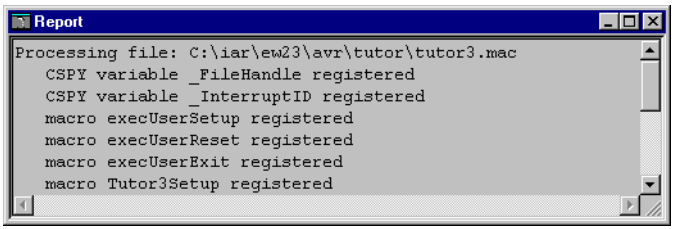
Displays the following dialog box, to allow you to specify a list of files from which to read macro definitions into C-SPY:



Select the macro definition files you want to use in the file selection list, and click **Add** to add them to the **Selected Macro Files** list or **Add All** to add all the listed files.

You can remove files from the **Selected Macro Files** list using **Remove** or **Remove All**.

Once you have selected the macro definition files you want to use click **Register** to register them, replacing any previously defined macros or variables. The macros are listed in the Report window as they are registered:



Registered macros are also displayed in the scroll window under **Registered Macros**.

Clicking on either **Name** or **File** under **Registered Macros** displays the column contents sorted by macro names or by file. Clicking once again sorts the contents in the reverse order. Selecting **All** displays all macros, selecting **User** displays all user macros, and selecting **System** displays all system macros.

Double-clicking on a user-defined macro in the **Name** column automatically opens the file in Microsoft Notepad, where it is available for editing.

Click **Close** to exit the **Macro Files** window.

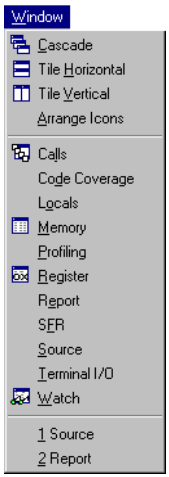
SELECT LOG FILE...

Allows you to log input and output from C-SPY to a file. This command displays a standard **Save As** dialog box to allow you to select the name and the location of the log file.

Browse to a suitable folder and type in a filename; the default extension is `log`. Then click **Save** to select the specified file.

Choose **Log to File** in the **Control** menu to turn on or off the logging to the file.

WINDOW MENU



The first section of the **Window** menu contains commands to let you control the order and arrangement of the C-SPY windows.

The central section of the menu lists each of the C-SPY windows. Select a menu command to open the corresponding window.

The last section of the menu lists the open windows. Selecting a window makes it the active window.

CASCADE

Rearranges the windows in a cascade on the screen.

TILE HORIZONTAL

Tiles the windows horizontally in the main C-SPY window.

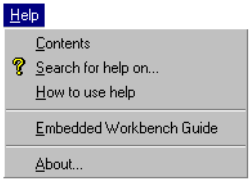
TILE VERTICAL

Tiles the windows vertically in the main C-SPY window.

ARRANGE ICONS

Tidies minimized window icons in the main C-SPY window.

HELP MENU



Provides help about C-SPY.

CONTENTS

Displays the Contents page for help about C-SPY.

SEARCH FOR HELP ON...

Allows you to search for help on a keyword.

HOW TO USE HELP

Displays help about using help.

EMBEDDED WORKBENCH GUIDE

Provides access to a hypertext version of this user guide.

ABOUT...

Displays the version number of the IAR C-SPY Debugger user interface and of the C-SPY driver for the AVR.

C-SPY COMMAND LINE OPTIONS

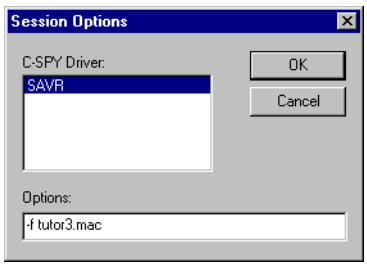
This chapter describes the C-SPY® command line options.

Normally you specify the C-SPY options among the other project options in the IAR Embedded Workbench; see the chapter *C-SPY options* in *Part 3: The IAR Embedded Workbench* in this guide for detailed information.

SETTING C-SPY OPTIONS

Setting C-SPY options from the command line

You can specify C-SPY options when you start the IAR C-SPY Debugger, `cw23.exe`, with the Windows **Run...** command or from the command line. When you run C-SPY outside the IAR Embedded Workbench, the following dialog box will appear when you open a project:



The following command line options are available:

Option	*Description*
`-d` *`driver`*	Selects C-SPY driver.
`--enhanced_core`	Specifies the enhanced instruction set.
`-f` *`file`*	Use setup file.
`--no_rampd`	Use `RAMPZ` instead of `RAMPD`
`-p` *`file`*	Loads device description file.
`-v{`*`option`*`}`	Specifies processor option.
`--64bit_doubles`	Use 64-bit doubles

C-SPY DRIVER (-d)

Use this option to select the appropriate driver for use with C-SPY, for example a simulator or an emulator. The following table shows the currently available driver:

Driver	*C-SPY version*
`savr.cdr`	Simulator

Syntax: `-d driver`

Example
To debug `project1.d90` with the simulator driver:

```
cw23 -d savr project1.d90
```

ENHANCED CORE (--enhanced_core)

Use this option to use the enhanced instruction set that is available in some AVR derivatives, for example AT90mega161.

Syntax: `--enhanced_core`

USE SETUP FILE (-f)

Use this option to register the contents of the specified macro file in the C-SPY startup sequence. If no extension is specified, the extension `mac` is assumed.

Syntax: `-f file`

Example
To register `watchdog.mac` at startup when debugging `watchdog.d90`:

```
cw23 -f watchdog.mac watchdog.d90
```

USE RAMPZ INSTEAD OF RAMPD (--no_rampd)

Use this option to specify that the `RAMPZ` register is used instead of `RAMPD` in direct address mode. Using the RAMPZ registers corresponds to the AVR instructions `LDS` and `STS`.

Syntax: `--no_rampd`

USE DEVICE DESCRIPTION FILE (-p)

Syntax: `-p file`

Use this option to load a device description file which contains various device specific information such as I/O registers (SFR) definitions, vector, and control register definitions.

Example
To use the `io2313.ddf` description file, enter the following command:

```
cw23 -p io2313.ddf
```

PROCESSOR OPTION (-v)

Syntax: `-v{option}`

Specifies the processor type as follows:

Option	*Description*
`-v0`	Max 256 byte data, 8 Kbyte code
`-v1`	Max 64 Kbyte data, 8 Kbyte code
`-v2`	Max 256 byte data, 128 Kbyte code
`-v3`	Max 64 Kbyte data, 128 Kbyte code
`-v4`	Max 16 Mbyte data, 128 Kbyte code
`-v5`	Max 64 Kbyte data, 8 Mbyte code
`-v6`	Max 16 Mbyte data, 8 Mbyte code

64-BIT DOUBLES (--64bit_doubles)

Use this option to specify that the code contains 64-bit doubles instead of 32-bit doubles which is the default.

Syntax: `--64bit_doubles`

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Symbols