MSP430 IAR Embedded Workbench® IDE

User Guide

for Texas Instruments'

MSP430 Microcontroller Family

EDITION NOTICE
Fifth edition: March 2006

Part number: U430-5

This guide describes version 3.x of IAR Embedded Workbench® for Texas Instruments' MSP430 microcontroller family.

Brief contents

Contents

Tables

Figures

Preface

Welcome to the MSP430 IAR Embedded Workbench® IDE User Guide. The purpose of this guide is to help you fully utilize the features in MSP430 IAR Embedded Workbench with its integrated Windows development tools for the MSP430 microcontroller. The IAR Embedded Workbench IDE is a very powerful Integrated Development Environment that allows you to develop and manage a complete embedded application project.

The user guide includes product overviews and reference information, as well as tutorials that will help you get started. It also describes the processes of editing, project managing, building, and debugging.

Who should read this guide

You should read this guide if you want to get the most out of the features and tools available in the IAR Embedded Workbench IDE. In addition, you should have a working knowledge of:

- The C or C++ programming language
- Application development for embedded systems
- The architecture and instruction set of the MSP430 microcontroller (refer to the chip manufacturer's documentation)
- The operating system of your host computer.

Refer to the *MSP430 IAR C/C++ Compiler Reference Guide*, *MSP430 IAR Assembler Reference Guide*, and *IAR Linker and Library Tools Reference Guide* for more information about the other development tools incorporated in the IAR Embedded Workbench IDE.

How to use this guide

If you are new to using this product, we suggest that you start by reading *Part 1. Product overview* to give you an overview of the tools and the functions that the IAR Embedded Workbench IDE can offer.

If you already have had some experience using IAR Embedded Workbench, but need refreshing on how to work with the IAR development tools, *Part 2. Tutorials* is a good place to begin. The process of managing projects and building, as well as editing, can be found in *Part 3. Project management and building*, page 71, whereas information about how to use the C-SPY® Debugger can be found in *Part 4. Debugging*, page 105.

If you are an experienced user and need this guide only for reference information, see the reference chapters in *Part 7. Reference information* and the online help system available from the IAR Embedded Workbench **Help** menu.

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

What this guide contains

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

Part 1. Product overview

This section provides a general overview of all the IAR development tools so that you can become familiar with them:

- *Product introduction* provides a brief summary and lists the features offered in each of the IAR Systems development tools—IAR Embedded Workbench® IDE, IAR C/C++ Compiler, IAR Assembler, IAR XLINK Linker, IAR XAR Library Builder, IAR XLIB Librarian, and IAR C-SPY Debugger—for the MSP430 microcontroller.
- *Installed files* describes the directory structure and the types of files it contains. The chapter also includes an overview of the documentation supplied with the IAR development tools.

Part 2. Tutorials

The tutorials give you hands-on training in order to help you get started with using the tools:

- *Creating an application project* guides you through setting up a new project, compiling your application, examining the list file, and linking your application. The tutorial demonstrates a typical development cycle, which is continued with debugging in the next chapter.
- *Debugging using the IAR C-SPY® Debugger* explores the basic facilities of the debugger.
- *Mixing C and assembler modules* demonstrates how you can easily combine source modules written in C with assembler modules. The chapter also demonstrates how the compiler can be used for examining the calling convention.

- *Using C++* shows how to create a C++ class, which creates two independent objects. The application is then built and debugged.
- *Simulating an interrupt* shows how you can add an interrupt handler to the project and how this interrupt can be simulated using C-SPY facilities for simulated interrupts, breakpoints, and macros.
- *Working with library modules* demonstrates how to create library modules.

Part 3. Project management and building

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

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

Part 4. Debugging

This section gives conceptual information about C-SPY functionality and how to use it:

- *The IAR C-SPY® Debugger* introduces some of the concepts that are related to debugging in general and to the IAR C-SPY Debugger in particular. It also introduces you to the C-SPY environment and how to setup, start, and configure C-SPY to reflect the target hardware.
- *Executing your application* describes how you initialize the IAR C-SPY Debugger, the conceptual differences between source and disassembly mode debugging, the facilities for executing your application, and finally, how you can handle terminal input and output.
- *Working with variables and expressions* defines the syntax of the expressions and variables used in C-SPY, as well as the limitations on variable information. The chapter also demonstrates the different methods for monitoring variables and expressions.
- *Using breakpoints* describes the breakpoint system and the different ways to define breakpoints.
- *Monitoring memory and registers* shows how you can examine memory and registers.
- *Using the C-SPY® macro system* describes the C-SPY macro system, its features, for what purposes these features can be used, and how to use them.
- *Analyzing your application* presents facilities for analyzing your application.

Part 5. IAR C-SPY Simulator

- *Simulator-specific debugging* describes the functionality specific to the simulator.
- *Simulating interrupts* contains detailed information about the C-SPY interrupt simulation system and how to configure the simulated interrupts to make them reflect the interrupts of your target hardware.

Part 6. IAR C-SPY® FET debugger

- *Introduction to the IAR C-SPY® FET Debugger* introduces you to the C-SPY Emulator Debugger. The chapter briefly shows the difference in functionality provided by the different debugger systems.
- *C-SPY® FET-specific debugging* describes the additional options, menus, and features provided by the C-SPY FET driver.
- *Design considerations for in-circuit programming* describes the design considerations related to the bootstrap loader, device signals, and external power if you want to use C-SPY with your own hardware.

Part 7. Reference information

- *IAR Embedded Workbench® IDE reference* contains detailed reference information about the development environment, such as details about the graphical user interface.
- *C-SPY® Debugger reference* provides detailed reference information about the graphical user interface of the IAR C-SPY Debugger.
- *General options* specifies the target, output, library, heap, stack, and MISRA C options.
- *Compiler options* specifies compiler options for language, code, output, list file, preprocessor, diagnostics, and MISRA C.
- *Assembler options* describes the assembler options for language, output, list, preprocessor, and diagnostics.
- *Custom build options* describes the options available for custom tool configuration.
- *Build actions options* describes the options available for pre-build and post-build actions.
- *Linker options* describes the XLINK options for output, defining symbols, diagnostics, list generation, setting up the include paths, input, and processing.
- *Library builder options* describes the XAR options available in the Embedded Workbench.
- *Debugger options* gives reference information about generic C-SPY options.
- *C-SPY® macros reference* gives reference information about C-SPY macros, such as a syntax description of the macro language, summaries of the available setup macro functions, and pre-defined system macros. Finally, a description of each system macro is provided.

Glossary

The glossary contains definitions of programming terms.

Other documentation

The complete set of IAR development tools for the MSP430 microcontroller are described in a series of guides. For information about:

- Programming for the MSP430 IAR C/C++ Compiler, refer to the *MSP430 IAR C/C++ Compiler Reference Guide*
- Programming for the MSP430 IAR Assembler, refer to the *MSP430 IAR Assembler Reference Guide*
- Using the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian, refer to the *IAR Linker and Library Tools Reference Guide*
- Using the IAR DLIB Library, refer to the *DLIB Library Reference information*, available in the MSP430 IAR Embedded Workbench IDE online help system.
- Using the IAR CLIB Library, refer to the *IAR C Library Functions Reference Guide*, available in the MSP430 IAR Embedded Workbench IDE online help system.
- Porting application code and projects created with a previous version of the MSP430 IAR Embedded Workbench IDE, refer to the *MSP430 IAR Embedded Workbench Migration Guide*.
- Developing safety-critical applications using the MISRA C guidelines, refer to the *IAR Embedded Workbench® MISRA C Reference Guide*.

All of these guides are delivered in hypertext PDF or HTML format on the installation media. Some of them are also delivered as printed books. Note that additional documentation might be available on the **Help** menu depending on your product installation.

Recommended web sites:

- The Texas Instruments web site, **www.ti.com**, contains information and news about the MSP430 microcontrollers.
- The IAR Systems web site, **www.iar.com**, holds application notes and other product information.
- Finally, the Embedded C++ Technical Committee web site, **www.caravan.net/ec2plus**, contains information about the Embedded C++ standard.

Document conventions

This book uses the following typographic conventions:

Style	Used for
`computer`	Text that you type 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.
`{option}`	A mandatory 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.
reference	A cross-reference within this guide or to another guide.
…	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
	Identifies instructions specific to the IAR Embedded Workbench IDE interface.
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.

Table 1: Typographic conventions used in this guide

Part 1. Product overview

This part of the MSP430 IAR Embedded Workbench® IDE User Guide includes the following chapters:

- Product introduction
- Installed files.

Product introduction

The IAR Embedded Workbench® IDE is a very powerful Integrated Development Environment, that allows you to develop and manage complete embedded application projects. It is a development platform, with all the features you would expect to find in your everyday working place.

This chapter describes the IAR Embedded Workbench IDE and provides a general overview of all the tools that are integrated in this product.

The IAR Embedded Workbench IDE

The IAR Embedded Workbench IDE is the framework where all necessary tools are seamlessly integrated:

- The highly optimizing MSP430 IAR C/C++ Compiler
- The MSP430 IAR Assembler
- The versatile IAR XLINK Linker
- The IAR XAR Library Builder and the IAR XLIB Librarian
- A powerful editor
- A project manager
- A command line build utility
- IAR C-SPY® debugger, a state-of-the-art high-level language debugger.

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

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

AN EXTENSIBLE AND MODULAR ENVIRONMENT

Although the IAR Embedded Workbench IDE provides all the features required for a successful project, we also recognize the need to integrate other tools. Therefore the IAR Embedded Workbench IDE can be easily adapted to work with your favorite editor and source code control system. The IAR XLINK Linker can produce a large number of output formats, allowing for debugging on most third-party emulators. Support for RTOS-aware debugging can also be added to the product.

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

FEATURES

The IAR Embedded Workbench IDE 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.

Project management

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

- Project templates to create a project that can be built and executed *out of the box* for a smooth development startup
- Hierarchical project representation
- Source browser with an hierarchical symbol presentation
- Options can be set globally, on groups of source files, or on individual source files
- The Make utility recompiles, reassembles, and links files only when necessary
- Text-based project files
- Custom Build utility to expand the standard tool chain in an easy way
- Command line build with the project file as input.

Source code control

Source code control (SCC)—or revision control—is useful for keeping track of different versions of your source code. IAR Embedded Workbench can identify and access any third-party source code control system that conforms to the SCC interface published by Microsoft.

Window management

To give you full and convenient control of the placement of the windows, each window is *dockable* and you can optionally organize the windows in tab groups. The system of dockable windows also provides a space-saving way to keep many windows open at the same time. It also makes it easy to rearrange the size of the windows.

The text editor

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

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

DOCUMENTATION

The MSP430 IAR Embedded Workbench IDE is documented in the *MSP430 IAR Embedded Workbench® IDE User Guide* (this guide). There is also help and hypertext PDF versions of the user documentation available online.

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 Systems compilers and assemblers, and it is completely integrated in the IAR Embedded Workbench IDE, providing seamless switching between development and debugging. This will give you possibilities such as:

- Editing while debugging. During a debug session, corrections can be made directly into the same source code window that is used to control the debugging. Changes will be included in the next project rebuild.

- Setting source code breakpoints before starting the debugger. Breakpoints in source code will be associated with the same piece of source code even if additional code is inserted.

The IAR C-SPY Debugger consists both of a general part which provides a basic set of C-SPY features, and of a driver. The C-SPY driver is the part that provides communication with and control of the target system. The driver also provides a user interface—special menus, windows, and dialog boxes—to the functions provided by the target system, for instance, special breakpoints.

Contact your software distributor or IAR Systems representative for information about available C-SPY drivers. You can also find information on the IAR website, **www.iar.com**.

Depending on your product installation, IAR C-SPY Debugger is available with a simulator driver and optional drivers for hardware debugger systems.

For a brief overview of the available C-SPY drivers, see *IAR C-SPY Debugger systems*, page 8.

GENERAL C-SPY DEBUGGER FEATURES

Because IAR Systems provides an entire tool chain, the output provided by the compiler and linker can include extensive debug information for the debugger, resulting in good debugging possibilities for you. The IAR C-SPY Debugger offers the general features described in this section.

Source and disassembly level debugging

The IAR C-SPY Debugger allows you to switch between source and disassembly debugging as required, for both C or C++ and assembler source code.

Debugging the C or C++ source code provides the quickest and easiest way of verifying the program logic of your application whereas disassembly debugging lets you focus on the critical sections of your application, and provides you with precise control over the hardware. In Mixed-Mode display, the debugger also displays the corresponding C/C++ source code interleaved with the disassembly listing.

Single-stepping on a function call level

Compared to traditional debuggers, where the finest granularity for source level stepping is line by line, C-SPY provides a finer level of control by identifying every statement and function call as a step point. This means that each function calls—inside expressions, as well as function calls being part of parameter lists to other functions—can be single-stepped. The latter is especially useful when debugging C++ code, where numerous extra function calls are made, for example to object constructors.

The debug information also presents inlined functions as if a call was made, making the source code of the inlined function available.

Code and data breakpoints

The C-SPY breakpoint system lets you set breakpoints of various kinds in the application being debugged, allowing you to stop at locations of particular interest. You can set a *code* breakpoint to investigate whether your program logic is correct. You can also set a *data* breakpoint, to investigate how and when the data changes. Finally, you can add conditions and connect actions to your breakpoints.

Monitoring variables and expressions

When you work with variables and expressions you are presented with a wide choice of facilities. Any variable and expression can be evaluated in one-shot views. You can easily both monitor and log values of a defined set of expressions during a longer period of time. You have instant control over local variables, and real-time data is displayed non-intrusively. Finally, the last referred variables are displayed automatically.

Container awareness

When you run your application in the IAR C-SPY Debugger, you can view the elements of library data types such as STL lists and vectors. This gives you a very good overview and premium debugging opportunities when you work with C++ STL containers.

Call stack information

The MSP430 IAR C/C++ Compiler generates extensive call stack information. This allows C-SPY to show, without any runtime penalty, the complete stack of function calls wherever the program counter is. You can select any function in the call stack, and for each function you get valid information for local variables and registers available.

Powerful macro system

The IAR C-SPY Debugger includes a powerful internal macro system, to allow you to define complex sets of actions to be performed. C-SPY macros can be used solely or in conjunction with complex breakpoints and—if you are using the simulator—the interrupt simulation system to perform a wide variety of tasks.

Additional general C-SPY Debugger features

This list shows some additional features:

- A modular and extensible architecture allowing third-party extensions to the debugger, for example, real-time operating systems, peripheral simulation modules, and emulator drivers

- Threaded execution keeps the IDE responsive while running the target application
- Automatic stepping
- Source browser provides easy navigation to functions, types and variables
- Extensive type recognition of variables
- Configurable registers (CPU and peripherals) and memory windows
- Dedicated Stack window
- Support for code coverage and function level profiling
- Optional terminal I/O emulation
- UBROF, Intel-extended, and Motorola input formats supported.

RTOS AWARENESS

The IAR C-SPY Debugger supports Real-time OS awareness debugging.

RTOS plugin modules can be provided by IAR, as well as by third-party suppliers. Contact your software distributor or IAR representative, alternatively visit the IAR Systems web site, for information about supported RTOS modules.

DOCUMENTATION

The IAR C-SPY Debugger is documented in the *MSP430 IAR Embedded Workbench® IDE User Guide* (this guide). Generic debugger features are described in *Part 4. Debugging*, whereas features specific to each debugger driver are described in *Part 5. IAR C-SPY Simulator*, and *Part 6. IAR C-SPY® FET debugger*. There are also help and hypertext PDF versions of the documentation available online.

IAR C-SPY Debugger systems

At the time of writing this guide, the IAR C-SPY Debugger for the MSP430 microcontroller is available with drivers for the following target systems:

- Simulator
- FET Debugger

Contact your software distributor or IAR representative for information about available C-SPY drivers. You can also find information on the IAR Systems web site, **www.iar.com**.

For further details about the concepts that are related to the IAR C-SPY Debugger, see *Debugger concepts*, page 107. In the following sections you can find general descriptions of the different drivers.

IAR C-SPY SIMULATOR

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

Features

In addition to the general features of the C-SPY Debugger the simulator driver also provides:

- Instruction-level simulation
- Memory configuration and validation
- Interrupt simulation
- Peripheral simulation, using the C-SPY macro system in conjunction with immediate breakpoints.

For additional information about the IAR C-SPY Simulator, refer to *Part 5. IAR C-SPY Simulator* in this guide.

IAR C-SPY FET DEBUGGER

The IAR C-SPY Flash Emulation Tool Debugger is a JTAG debugger that supports all Texas Instruments' boards. It provides automatic flash download and takes advantage of on-chip debug facilities.

The IAR C-SPY FET Debugger provides real-time debugging at a low cost.

Features

In addition to the general features of the IAR C-SPY Debugger, the FET Debugger driver also provides:

- Execution in real time with full access to the microcontroller
- High-speed communication through a JTAG interface
- Zero memory footprint on target system
- Hardware breakpoints for both code and data
- Built-in flash downloader.

On devices with the Enhanced Emulation Module (EEM), you have access also to:

- State storage
- Sequencer
- Clock control

Note: Code coverage and live watch are not supported by the C-SPY FET Debugger. Trace and data breakpoints are available if the device has support for it.

For additional information about the IAR C-SPY Emulator, refer to *Part 6. IAR C-SPY® FET debugger* in this guide.

IAR C/C++ Compiler

The MSP430 IAR C/C++ Compiler is a state-of-the-art compiler that offers the standard features of the C or C++ languages, plus many extensions designed to take advantage of the MSP430-specific facilities.

The compiler is integrated with other IAR Systems software in the IAR Embedded Workbench IDE.

FEATURES

The MSP430 IAR C/C++ Compiler provides the following features:

Code generation

- Generic and MSP430-specific optimization techniques produce very efficient machine code
- Comprehensive output options, including relocatable object code, assembler source code, and list files with optional assembler mnemonics
- The object code can be linked together with assembler routines
- Generation of extensive debug information.

Language facilities

- Support for the C and C++ programming languages
- Support for IAR Extended EC++ with features such as full template support, namespace support, the cast operators `static_cast`, `const_cast`, and `reinterpret_cast`, as well as the Standard Template Library (STL)
- Placement of classes in different memory types
- Conformance to the ISO/ANSI C standard for a free-standing environment
- Target-specific language extensions, such as special function types, extended keywords, pragma directives, predefined symbols, intrinsic functions, absolute allocation, and inline assembler
- Standard library of functions applicable to embedded systems
- IEEE-compatible floating-point arithmetic
- Interrupt functions can be written in C or C++.

Type checking

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

RUNTIME ENVIRONMENT

The MSP430 IAR Embedded Workbench provides two sets of runtime libraries:

- The IAR DLIB Library, which supports ISO/ANSI C and C++. This library also supports floating-point numbers in IEEE 754 format, multi-byte characters, and locales.
- The IAR CLIB Library is a light-weight library, which is not fully compliant with ISO/ANSI C. Neither does it fully support floating-point numbers in IEEE 754 format or C++.

There are several mechanisms available for customizing the runtime environment and the runtime libraries. For both sets of runtime libraries, library source code is included.

DOCUMENTATION

The MSP430 IAR C/C++ Compiler is documented in the *MSP430 IAR C/C++ Compiler Reference Guide.*

IAR Assembler

The MSP430 IAR Assembler is integrated with other IAR Systems software for the MSP430 microcontroller. It is a powerful relocating macro assembler (supporting the Intel/Motorola style) with a versatile set of directives and expression operators. The assembler features a built-in C language preprocessor and supports conditional assembly.

The MSP430 IAR Assembler uses the same mnemonics and operand syntax as the Texas Instruments MSP430 Assembler, which simplifies the migration of existing code. For detailed information, see the *MSP430 IAR Assembler Reference Guide*.

FEATURES

The MSP430 IAR Assembler provides the following features:

- C preprocessor
- List file with extensive 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.

DOCUMENTATION

The MSP430 IAR Assembler is documented in the *MSP430 IAR Assembler Reference Guide.*

IAR XLINK Linker

The IAR XLINK Linker links one or more relocatable object files produced by the MSP430 IAR Assembler or MSP430 IAR C/C++ Compiler to produce machine code for the MSP430 microcontroller. It is equally well suited for linking small, single-file, absolute assembler applications as for linking large, relocatable, multi-module, C/C++, or mixed C/C++ and assembler applications.

It can generate one out of more than 30 industry-standard loader formats, in addition to the IAR Systems proprietary debug format used by the IAR C-SPY Debugger—UBROF (Universal Binary Relocatable Object Format). An application can be made up of any number of UBROF relocatable files, in any combination of assembler and C or C++ applications.

The final output produced by the IAR XLINK Linker is an absolute, target-executable object file that can be downloaded to the MSP430 microcontroller or to a hardware emulator. Optionally, the output file might or might not contain debug information depending on the output format you choose.

The IAR XLINK Linker supports user libraries, and will load only those modules that are actually needed by the application you are linking. Before linking, the IAR XLINK Linker performs a full C-level type checking across all modules as well as a full dependency resolution of all symbols in all input files, independent of input order. It also checks for consistent compiler settings for all modules and makes sure that the correct version and variant of the C or C++ runtime library is used.

FEATURES

- Full inter-module type checking
- Simple override of library modules
- Flexible segment commands allow detailed control of code and data placement
- Link-time symbol definition enables flexible configuration control
- Optional code checksum generation for runtime checking
- Removes unused code and data.

DOCUMENTATION

The IAR XLINK Linker is documented in the *IAR Linker and Library Tools Reference Guide*.

IAR XAR Library Builder and IAR XLIB Librarian

A library is a single file that contains a number of relocatable object modules, each of which can be loaded independently from other modules in the file as it is needed. The IAR XAR Library Builder assists you to build libraries easily. In addition the IAR XLIB Librarian enables you to manipulate the relocatable library object files produced by the IAR Systems assembler and compiler.

A library file is no different from any other relocatable object file produced by the assembler or compiler, except that it includes a number of modules of the `LIBRARY` type. All C or C++ applications make use of libraries, and the MSP430 IAR C/C++ Compiler is supplied with a number of standard library files.

FEATURES

The IAR XAR Library Builder and IAR XLIB Librarian both provide the following features:

- Modules can be combined into a library file
- Interactive or batch mode operation.

The IAR XLIB Librarian provides the following additional features:

- Modules can be listed, added, inserted, replaced, or removed
- Modules can be changed between program and library type
- Segments can be listed
- Symbols can be listed.

DOCUMENTATION

The IAR XLIB Librarian and the IAR XAR Library Builder are documented in the *IAR Linker and Library Tools Reference Guide*, a PDF document available from the IAR Embedded Workbench IDE **Help** menu.

Installed files

This chapter describes which directories are created during installation and what file types are used. At the end of the chapter, there is a section that describes what information you can find in the various guides and online documentation.

Refer to the *QuickStart Card* and the *Installation and Licensing Guide*, which are delivered with the product, for system requirements and information about how to install and register the IAR Systems products.

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\Embedded Workbench 4.n\` directory where `x` is the drive where Microsoft Windows is installed and `4.n` is the version number of the IAR Embedded Workbench IDE.

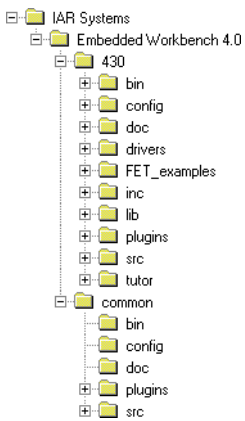


Figure 1: Directory structure

Note: The installation path can be different from the one shown above depending on previously installed IAR products, and on your preferences.

THE 430 DIRECTORY

The `430` directory contains all product-specific subdirectories.

The 430\bin directory

The `430\bin` subdirectory contains executable files for MSP430-specific components, such as the MSP430 IAR C/C++ Compiler, the MSP430 IAR Assembler, and the MSP430 IAR C-SPY® drivers.

The 430\config directory

The `430\config` subdirectory contains files used for configuring the development environment and projects, for example:

- Linker command files (`*.xcl`)
- Special function register description files (`*.sfr`)
- The C-SPY device description files (`*.ddf`)
- Syntax coloring configuration files (`*.cfg`)
- Project templates for both application and library projects (`*.ewp`), and for the library projects, the corresponding library configuration files.

The 430\doc directory

The `430\doc` subdirectory contains release notes with recent additional information about the MSP430 tools. We recommend that you read all of these files. The directory also contains online hypertext versions in hypertext PDF format of this user guide, and of the MSP430 reference guides, as well as online help files (CHM format).

The 430\drivers directory

The `430\drivers` directory contains hardware debugger drivers.

The 430\FET_examples directory

The `430\FET_examples` directory contains FET debugger example files.

The 430\inc directory

The `430\inc` subdirectory holds include files, such as the header files for the standard C or C++ library. There are also specific header files defining special function registers (SFRs); these files are used by both the compiler and the assembler.

The 430\lib directory

The `430\lib` subdirectory holds prebuilt libraries and the corresponding library configuration files, used by the compiler.

The 430\plugins directory

The `430\plugins` subdirectory contains executable files and description files for components that can be loaded as plugin modules.

The 430\src directory

The `430\src` subdirectory holds source files for some configurable library functions, and application code examples. This directory also holds the library source code.

The 430\tutor directory

The `430\tutor` subdirectory contains the files used for the tutorials in this guide.

THE COMMON DIRECTORY

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

The common\bin directory

The `common\bin` subdirectory contains executable files for components common to all IAR Embedded Workbench products, such as the IAR XLINK Linker, the IAR XLIB Librarian, the IAR XAR Library Builder, the editor and the graphical user interface components. The executable file for the IAR Embedded Workbench IDE is also located here.

The common\config directory

The `common\config` subdirectory contains files used by IAR Embedded Workbench for holding settings in the development environment.

The common\doc directory

The `common\doc` subdirectory contains readme files with recent additional information about the components common to all IAR Embedded Workbench products, such as the linker and library tools. We recommend that you read these files. The directory also contains an online version in PDF format of the *IAR Linker and Library Tools Reference Guide*.

The common\plugins directory

The `common\plugins` subdirectory contains executable files and description files for components that can be loaded as plugin modules.

The common\src directory

The `common\src` subdirectory contains source files for components common to all IAR Embedded Workbench products, such as a sample reader of the IAR XLINK Linker output format `SIMPLE`.

File types

The MSP430 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
`a43`	Target application	XLINK	EPROM, C-SPY, etc.
`asm`	Assembler source code	Text editor	Assembler
`c`	C source code	Text editor	Compiler
`cfg`	Syntax coloring configuration	Text editor	IAR Embedded Workbench
`cpp`	Embedded C++ source code	Text editor	Compiler
`d43`	Target application with debug information	XLINK	C-SPY and other symbolic debuggers
`dbg`	Target application with debug information	XLINK	C-SPY and other symbolic debuggers
`dbgt`	Debugger desktop settings	C-SPY	C-SPY
`ddf`	Device description file	Text editor	C-SPY
`dep`	Dependency information	IAR Embedded Workbench	IAR Embedded Workbench
`dni`	Debugger initialization file	C-SPY	C-SPY
`ewd`	Project settings for C-SPY	IAR Embedded Workbench	IAR Embedded Workbench
`ewp`	IAR Embedded Workbench project (current version)	IAR Embedded Workbench	IAR Embedded Workbench
`eww`	Workspace file	IAR Embedded Workbench	IAR Embedded Workbench

Table 2: File types

Ext.	Type of file	Output from	Input to
fmt	Formatting information for the Locals and Watch windows	IAR Embedded Workbench	IAR Embedded Workbench
h	C/C++ or assembler header source	Text editor	Compiler or assembler #include
i	Preprocessed source	Compiler	Compiler
inc	Assembler header source	Text editor	Assembler #include
lst	List output	Compiler and assembler	–
mac	C-SPY macro definition	Text editor	C-SPY
map	List output	XLINK	–
pbd	Source browse information	IAR Embedded Workbench	IAR Embedded Workbench
pbi	Source browse information	IAR Embedded Workbench	IAR Embedded Workbench
pew	IAR Embedded Workbench project (old project format)	IAR Embedded Workbench	IAR Embedded Workbench
prj	IAR Embedded Workbench project (old project format)	IAR Embedded Workbench	IAR Embedded Workbench
r43	Object module	Compiler and assembler	XLINK, XAR, and XLIB
s43	MSP430 assembler source code	Text editor	MSP430 IAR Assembler
sfr	Special function register definitions	Text editor	C-SPY
wsdt	Workspace desktop settings	IAR Embedded Workbench	IAR Embedded Workbench
xcl	Extended command line	Text editor	Assembler, compiler, XLINK
xlb	Extended librarian batch command	Text editor	XLIB

Table 2: File types (Continued)

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

Files with the extensions ini and dni are created dynamically when you run the IAR Embedded Workbench tools. These files, which contain information about your project configuration and other settings, are located in a settings directory under your project directory.

Note: If you run the tools from the command line, the XLINK listings (map files) will by default have the extension `lst`, which might 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 MSP430 user and reference guides, in the online help, and on the Internet.

You can access the MSP430 online documentation from the **Help** menu in the IAR Embedded Workbench IDE. Help is also available via the F1 key in the IAR Embedded Workbench IDE.

We recommend that you read the file `readme.htm` for recent information that might not be included in the user guides. It is located in the `430\doc` directory.

THE USER AND REFERENCE GUIDES

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

MSP430 IAR Embedded Workbench® IDE User Guide

This guide.

MSP430 IAR C/C++ Compiler Reference Guide

This guide provides reference information about the MSP430 IAR C/C++ 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 assembler language interface and the calling convention
- The available data types
- The runtime libraries
- The IAR language extensions.

MSP430 IAR Assembler Reference Guide

This guide provides reference information about the MSP430 IAR Assembler, including details of the assembler source format, and reference information about the assembler operators, directives, mnemonics, and diagnostics.

IAR Linker and Library Tools Reference Guide

This online PDF guide provides reference information about the IAR linker and library tools:

- The IAR XLINK Linker reference sections provide information about XLINK options, output formats, environment variables, and diagnostics.
- The IAR XAR Library Builder reference sections provide information about XAR options and output.
- The IAR XLIB Librarian reference sections provide information about XLIB commands, environment variables, and diagnostics.

DLIB Library Reference information

This online documentation in HTML format provides reference information about the IAR DLIB library functions. It is available from the MSP430 IAR Embedded Workbench® IDE online help system.

CLIB Library Reference Guide

This online guide in hypertext PDF format contains reference information about the IAR CLIB Library. It is available from the MSP430 IAR Embedded Workbench® IDE online help system.

IAR Embedded Workbench® MISRA C Reference Guide

This online guide in hypertext PDF format describes how IAR Systems has interpreted and implemented the rules given in *Guidelines for the Use of the C Language in Vehicle Based Software* to enforce measures for stricter safety in the ISO standard for the C programming language [ISO/IEC 9899:1990].

ONLINE HELP

The context-sensitive online help contains reference information about the menus and dialog boxes in the IAR Embedded Workbench IDE. There is also keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1.

Note: If you select a function name in the editor window and press F1 while using the CLIB library, you will get reference information for the DLIB library.

IAR ON THE WEB

The latest news from IAR Systems can be found at the web site **www.iar.com**, available from the **Help** menu in the Embedded Workbench IDE. Visit it for information about:

- Product announcements

- Updates and news about current versions
- Special offerings
- Evaluation copies of the IAR Systems products
- Technical Support, including technical notes
- Application notes
- Links to chip manufacturers and other interesting sites
- Distributors; the names and addresses of distributors in each country.

Part 2. Tutorials

This part of the MSP430 IAR Embedded Workbench® IDE User Guide contains the following chapters:

- Creating an application project
- Debugging using the IAR C-SPY® Debugger
- Mixing C and assembler modules
- Using C++
- Simulating an interrupt
- Working with library modules.

Creating an application project

This chapter introduces you to the IAR Embedded Workbench® integrated development environment (IDE). The tutorial demonstrates a typical development cycle and shows how you use the compiler and the linker to create a small application for the MSP430 microcontroller. For instance, creating a workspace, setting up a project with C source files, and compiling and linking your application.

The development cycle continues in the next chapter, see *Debugging using the IAR C-SPY® Debugger*, page 37.

Setting up a new project

Using the IAR Embedded Workbench IDE, you can design advanced project models. You create a *workspace* to which you add one or several *projects*. There are ready-made *project templates* for both application and library projects. Each project can contain a hierarchy of *groups* in which you collect your *source files*. For each project you can define one or several *build configurations*. For more details about designing project models, see the chapter *Managing projects* in this guide.

Because the application in this tutorial is a simple application with very few files, the tutorial does not need an advanced project model.

We recommend that you create a specific directory where you can store all your project files. In this tutorial we call the directory `projects`. You can find all the files needed for the tutorials in the `430\tutor` directory. Make a copy of the `tutor` directory in your `projects` directory.

Before you can create your project you must first create a workspace.

CREATING A WORKSPACE WINDOW

The first step is to create a new workspace for the tutorial application. When you start the IAR Embedded Workbench IDE for the first time, there is already a ready-made workspace, which you can use for the tutorial projects. If you are using that workspace, you can ignore the first step.

Choose **File>New>Workspace**. Now you are ready to create a project and add it to the workspace.

CREATING THE NEW PROJECT

1 To create a new project, choose **Project>Create New Project**. The **Create New Project** dialog box appears, which lets you base your new project on a project template.

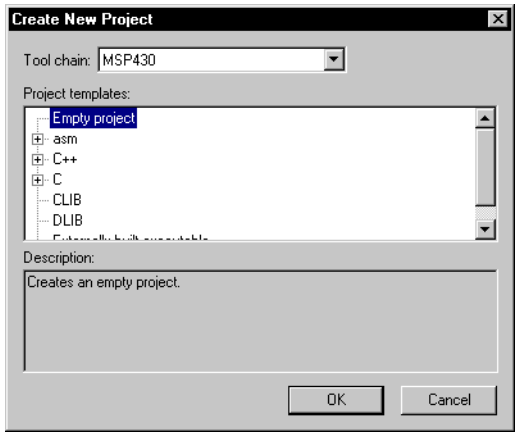


Figure 2: Create New Project dialog box

2 Make sure the **Tool chain** is set to **MSP430**, and click **OK.**

3 For this tutorial, select the project template **Empty project**, which simply creates an empty project that uses default project settings.

4 In the standard **Save As** dialog box that appears, specify where you want to place your project file, that is, in your newly created `projects` directory. Type `project1` in the **File name** box, and click **Save** to create the new project.

The project will appear in the Workspace window.

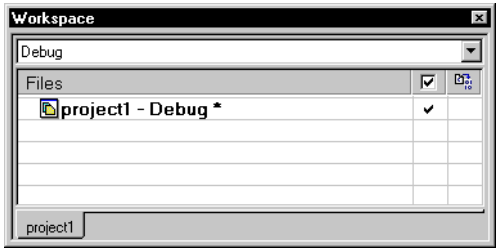


Figure 3: Workspace window

By default two build configurations are created: Debug and Release. In this tutorial only Debug will be used. You choose the build configuration from the drop-down menu at the top of the window. The asterisk in the project name indicates that there are changes that have not been saved.

A project file—with the filename extension `ewp`—will be created in the `projects` directory, not immediately, but later on when you save the workspace. This file contains information about your project-specific settings, such as build options.

5 Before you add any files to your project, you should save the workspace. Choose **File>Save Workspace** and specify where you want to place your workspace file. In this tutorial, you should place it in your newly created `projects` directory. Type `tutorials` in the **File name** box, and click **Save** to create the new workspace.

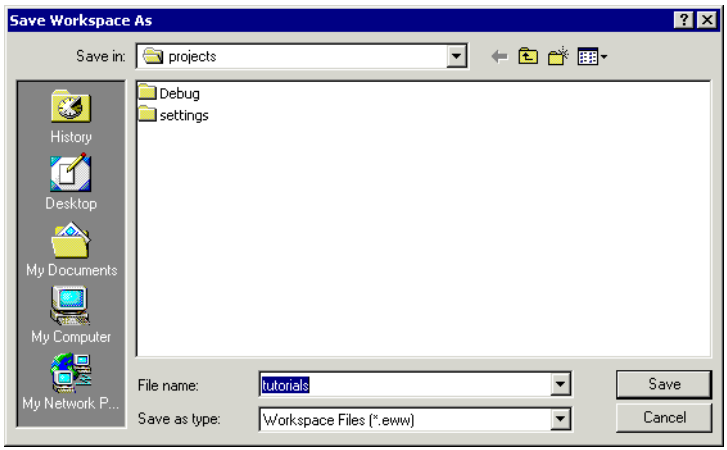


Figure 4: New Workspace dialog box

A workspace file—with the filename extension `eww`—has now been created in the `projects` directory. This file lists all projects that you will add to the workspace. Information related to the current session, such as the placement of windows and breakpoints is located in the files created in the `projects\settings` directory.

ADDING FILES TO THE PROJECT

This tutorial uses the source files `Tutor.c` and `Utilities.c`.

- The `Tutor.c` application is a simple program using only standard features of the C language. It initializes an array with the ten first Fibonacci numbers and prints the result to `stdout`.
- The `Utilities.c` application contains utility routines for the Fibonacci calculations.

Creating several *groups* is a possibility for you to organize your source files logically according to your project needs. However, because there are only two files in this project there is no need for creating a group. For more information about how to create complex project structures, see the chapter *Managing projects*.

1 In the Workspace window, select the destination to which you want to add a source file; a group or, as in this case, directly to the project.

2 Choose **Project>Add Files** to open a standard browse dialog box. Locate the files `Tutor.c` and `Utilities.c`, select them in the file selection list, and click **Open** to add them to the `project1` project.

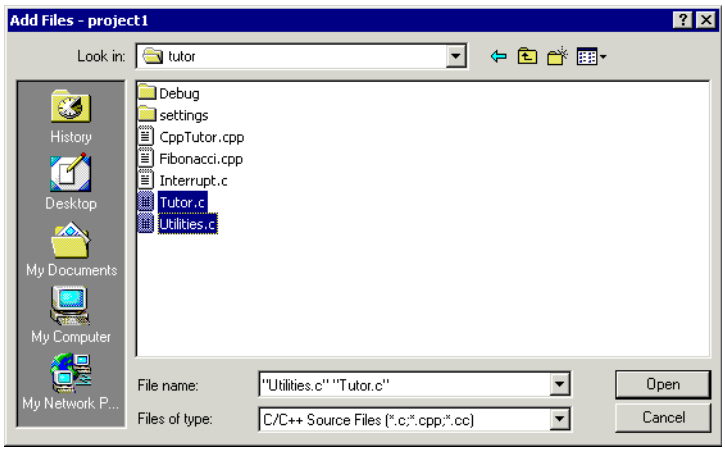


Figure 5: Adding files to project1

SETTING PROJECT OPTIONS

Now you will set the project options. For application projects, options can be set on all levels of nodes. First you will set the general options to suit the processor configuration in this tutorial. Because these options must be the same for the whole build configuration, they must be set on the project node.

1. Select the project folder icon **project1 - Debug** in the Workspace window and choose **Project>Options**.

   The **Target** options page in the **General Options** category is displayed.

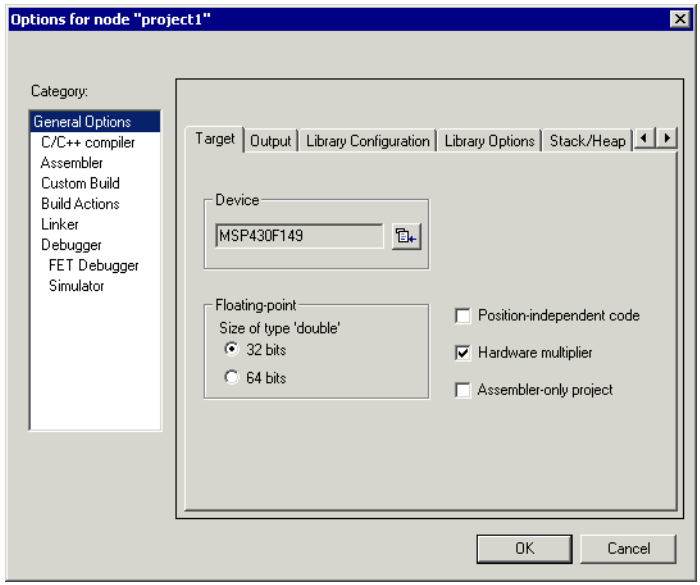


Figure 6: Setting general options

Verify the following settings:

Page	Setting
Target	Device: msp430F149
Output	Output file: Executable
Library Configuration	Library: CLIB

Table 3: General settings for project1

Then set up the compiler options for the project.

2 Select **C/C++ Compiler** in the **Category** list to display the compiler option pages.

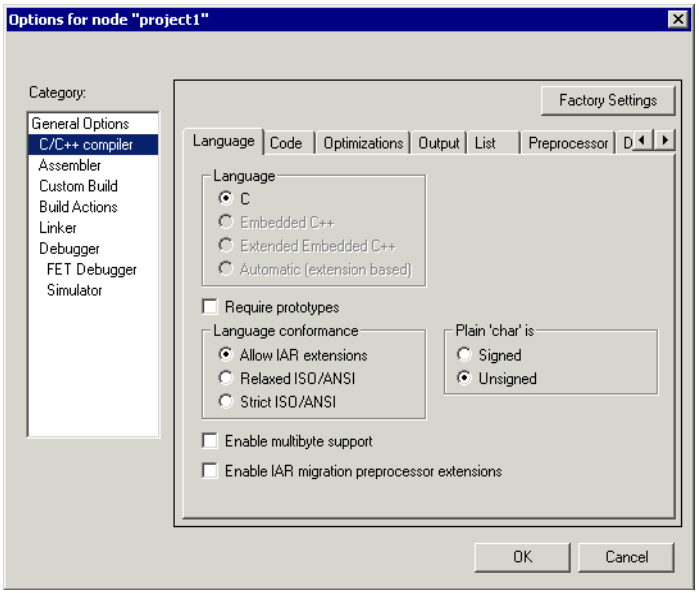


Figure 7: Setting compiler options

3 Verify the following settings:

Page	Setting
Optimizations	Optimizations, Size: None (Best debug support)
Output	Generate debug information
List	Output list file Assembler mnemonics

Table 4: Compiler options for project1

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

Note: It is possible to customize the amount of information to be displayed in the Build messages window. In this tutorial, the default setting is not used. Thus, the contents of the Build messages window on your screen might differ from the screen shots.

The project is now ready to be built.

Compiling and linking the application

You can now compile and link the application. You should also create a compiler list file and a linker map file and view both of them.

COMPILING THE SOURCE FILES

1. To compile the file `Utilities.c`, select it in the Workspace window.
2. Choose **Project>Compile**.

Alternatively, click the **Compile** button in the toolbar or choose the **Compile** command from the context menu that appears when you right-click on the selected file in the Workspace window.

The progress will be displayed in the Build messages window.

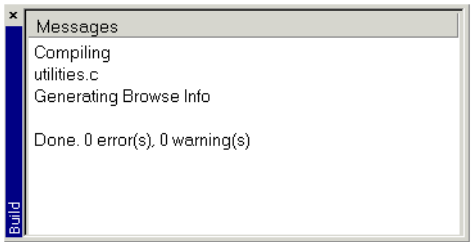


Figure 8: Compilation message

3. Compile the file `Tutor.c` in the same manner.

The IAR Embedded Workbench IDE has now created new directories in your project directory. Because you are using the build configuration **Debug**, a `Debug` directory has been created containing the directories `List`, `Obj`, and `Exe`:

- The `List` directory is the destination directory for the list files. The list files have the extension `lst`.
- The `Obj` directory is the destination directory for the object files from the compiler and the assembler. These files have the extension `r43` and will be used as input to the IAR XLINK Linker.
- The `Exe` directory is the destination directory for the executable file. It has the extension `d43` and will be used as input to the IAR C-SPY® Debugger. Note that this directory will be empty until you have linked the object files.

Click on the plus signs in the Workspace window to expand the view. As you can see, IAR Embedded Workbench has also created an output folder icon in the Workspace window containing any generated output files. All included header files are displayed as well, showing the dependencies between the files.

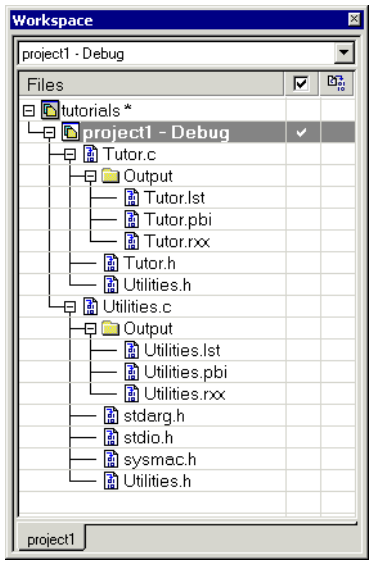


Figure 9: Workspace window after compilation

VIEWING THE LIST FILE

Now examine the compiler list file and notice how it is automatically updated when you, as in this case, will investigate how different optimization levels affect the generated code size.

1. Open the list file `Utilities.lst` by double-clicking it in the Workspace window. 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
   - The *body* of the list file shows the assembler code and binary code generated for each statement. It also shows how the variables are assigned to different segments
   - 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 might have been generated.

   Notice the amount of generated code at the end of the file and keep the file open.

2 Choose **Tools>Options** to open the **IDE Options** dialog box and click the **Editor** tab. Select the option **Scan for Changed Files**. This option turns on the automatic update of any file open in an editor window, such as a list file. Click the **OK** button.

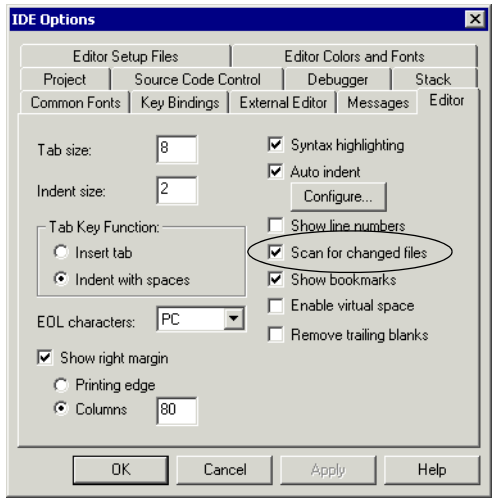


Figure 10: Setting the option Scan for Changed Files

3 Select the file `Utilities.c` in the Workspace window. Open the **C/C++ Compiler** options dialog box by right-clicking on the selected file in the Workspace window. Click the **Optimizations** tab and select the **Override inherited settings** option. Choose **High** from the **Optimizations** drop-down list. Click **OK**.

Notice that the options override on the file node is indicated in the Workspace window.

4 Compile the file `Utilities.c`. Now you will notice two things. First, note the automatic updating of the open list file due to the selected option **Scan for Changed Files**. Second, look at the end of the list file and notice the effect on the code size due to the increased optimization.

5 For this tutorial, the optimization level **None** should be used, so before linking the application, restore the default optimization level. Open the **C/C++ Compiler** options dialog box by right-clicking on the selected file in the Workspace window. Deselect the **Override inherited settings** option and click **OK**. Recompile the file `Utilities.c`.

LINKING THE APPLICATION

Now you should set up the options for the IAR XLINK Linker.

1 Select the project folder icon **project1 - Debug** in the Workspace window and choose **Project>Options**. Then select **Linker** in the **Category** list to display the XLINK option pages.

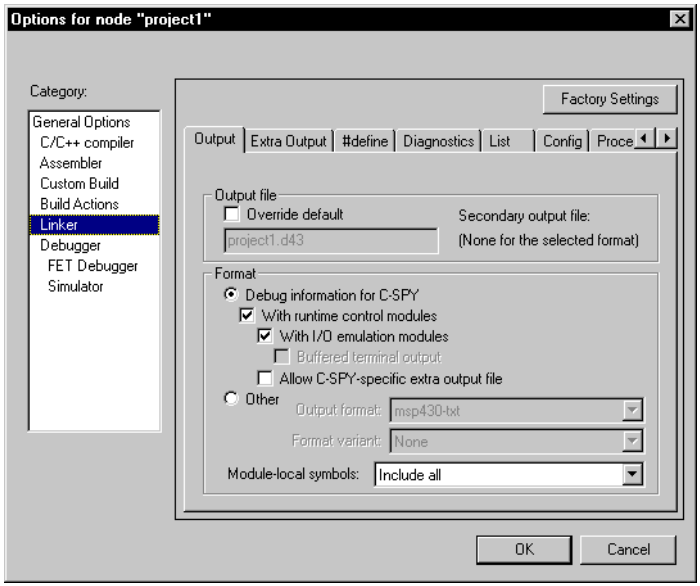


Figure 11: XLINK options dialog box for project1

For this tutorial, default factory settings are used. However, pay attention to the choice of output format and linker command file.

Output format

It is important to choose the output format that suits your purpose. You might want to load it to a debugger—which means that you need output with debug information. In this tutorial you will use the default output options suitable for the C-SPY debugger—**Debug information for C-SPY**, **With runtime control modules**, and **With I/O emulation modules**—which means that some low-level routines will be linked that direct `stdin` and `stdout` to the Terminal I/O window in the C-SPY Debugger. You find these options on the **Output** page.

Alternatively, in your real application project, you might want to load the output to a PROM programmer—in which case you need an output format without debug information, such as Intel-hex or Motorola S-records.

Linker command file

In the linker command file, the XLINK command line options for segment control are used for placing segments. It is important to be familiar with the linker command file and placement of segments. You can read more about this in the *MSP430 IAR C/C++ Compiler Reference Guide*.

The linker command file templates supplied with the product can be used as is in the simulator, but when using them for your target system you might have to adapt them to your actual hardware memory layout. You can find supplied linker command files in the `config` directory.

In this tutorial you will use the default linker command file, which you can see on the **Config** page.

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 definitions match your requirements.

Linker map file

By default no linker map file is generated. To generate a linker map file, click the **List** tab and select the options **Generate linker listing**, **Segment map**, and **Module map**.

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

Now you should link the object file, to generate code that can be debugged.

3 Choose **Project>Make**. The progress will as usual be displayed in the Build messages window. The result of the linking is a code file `project1.d43` with debug information and a map file `project1.map`.

VIEWING THE MAP FILE

Examine the file `project1.map` to see how the segment definitions and code were placed in memory. These 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 your application, including segments and global symbols declared within each segment, is displayed.

- The `SEGMENTS IN ADDRESS ORDER` section lists all the segments that constitute your application.

The `project1.d43` application is now ready to be run in the IAR C-SPY Debugger.

Debugging using the IAR C-SPY® Debugger

This chapter continues the development cycle started in the previous chapter and explores the basic features of the IAR C-SPY Debugger.

Note that, depending on what IAR product package you have installed, the IAR C-SPY Debugger may or may not be included. The tutorials assume that you are using the C-SPY Simulator.

Debugging the application

The `project1.d43` application, created in the previous chapter, is now ready to be run in the IAR C-SPY Debugger where you can watch variables, set breakpoints, view code in disassembly mode, monitor registers and memory, and print the program output in the Terminal I/O window.

STARTING THE DEBUGGER

Before starting the IAR C-SPY Debugger you must set a few C-SPY options.

1 Choose **Project>Options** and then the **Debugger** category. On the **Setup** page, make sure that you have chosen **Simulator** from the **Driver** drop-down list and that **Run to main** is selected. Click **OK**.

2 Choose **Project>Debug**. Alternatively, click the **Debugger** button in the toolbar. The IAR C-SPY Debugger starts with the `project1.d43` application loaded. In addition to the windows already opened in the Embedded Workbench, a set of C-SPY-specific windows are now available.

ORGANIZING THE WINDOWS

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

The status bar, located at the bottom of the Embedded Workbench main window, contains useful help about how to arrange windows. For further details, see *Organizing the windows on the screen*, page 75.

Make sure the following windows and window contents are open and visible on the screen: the Workspace window with the active build configuration **tutorials – project1**, the editor window with the source files `Tutor.c` and `Utilities.c`, and the Debug Log window.

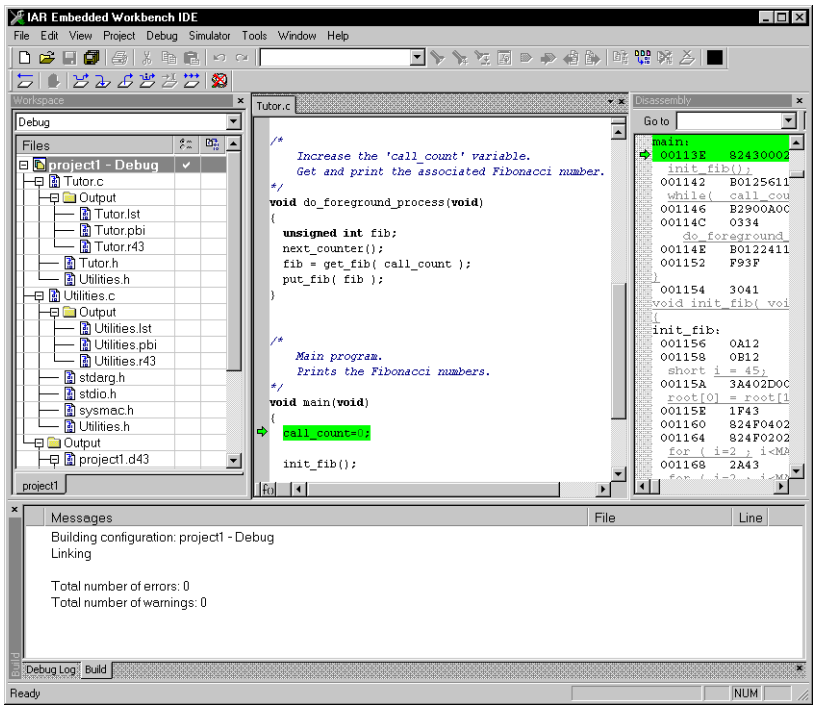


Figure 12: The C-SPY Debugger main window

INSPECTING SOURCE STATEMENTS

1 To inspect the source statements, double-click the file `Tutor.c` in the Workspace window.

2 With the file `Tutor.c` displayed in the editor window, first step over with the **Debug>Step Over** command.

Alternatively, click the **Step Over** button on the toolbar.

The current position should be the call to the `init_fib` function.

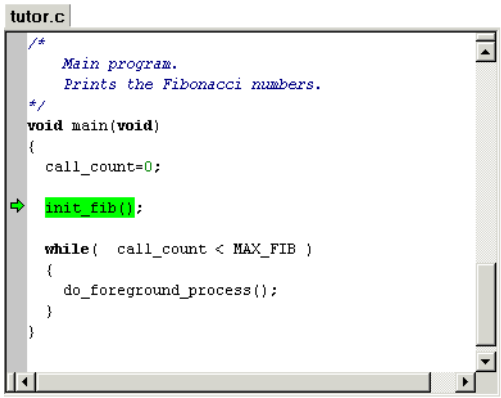


Figure 13: Stepping in C-SPY

3 Choose **Debug>Step Into** to step into the function `init_fib`.

Alternatively, click the **Step Into** button on the toolbar.

At source level, the **Step Over** and **Step Into** commands allow you to execute your application a statement or instruction at a time. **Step Into** continues stepping inside function or subroutine calls, whereas **Step Over** executes each function call in a single step. For further details, see *Step*, page 118.

When **Step Into** is executed you will notice that the active window changes to `Utilities.c` as the function `init_fib` is located in this file.

4 Use the **Step Into** command until you reach the `for` loop.

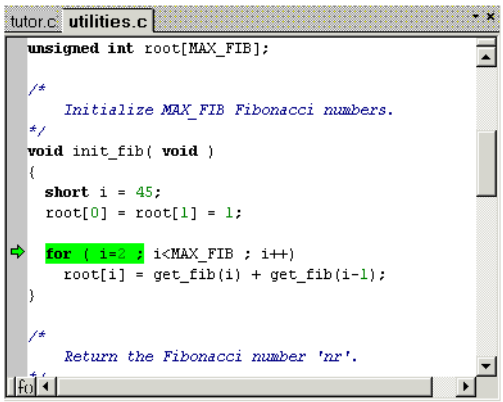


Figure 14: Using Step Into in C-SPY

5 Use **Step Over** until you are back in the header of the `for` loop. You will notice that the step points are on a function call level, not on a statement level.

You can also step on a statement level. Choose **Debug>Next statement** to execute one statement at a time. Alternatively, click the **Next statement** button on the toolbar.

Notice how this command differs from the **Step Over** and the **Step Into** commands.

INSPECTING VARIABLES

C-SPY allows you to watch variables or expressions in the source code, so that you can keep track of their values as you execute your application. You can look at a variable in a number of ways; for example by pointing at it in the source window with the mouse pointer, or by opening one of the Locals, Watch, Live Watch, or Auto windows. For more information about inspecting variables, see the chapter *Working with variables and expressions*.

Note: When optimization level **None** is used, all non-static variables will live during their entire scope and thus, the variables are fully debuggable. When higher levels of optimizations are used, variables might not be fully debuggable.

Using the Auto window

1 Choose **View>Auto** to open the Auto window.

The Auto window will show the current value of recently modified expressions.

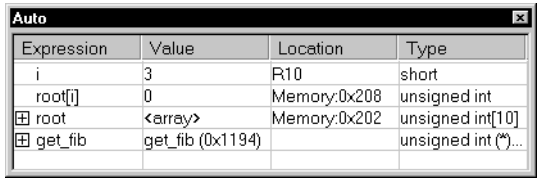


Figure 15: Inspecting variables in the Auto window

2 Keep stepping to see how the values change.

Setting a watchpoint

Next you will use the Watch window to inspect variables.

3 Choose **View>Watch** to open the Watch window. Notice that it is by default grouped together with the currently open Auto window; the windows are located as a *tab group*.

4 Set a watchpoint on the variable `i` using the following procedure: Click the dotted rectangle in the Watch window. In the entry field that appears, type `i` and press the Enter key.

You can also drag a variable from the editor window to the Watch window.

5 Select the `root` array in the `init_fib` function, then drag it to the Watch window.

The Watch window will show the current value of `i` and `root`. You can expand the `root` array to watch it in more detail.

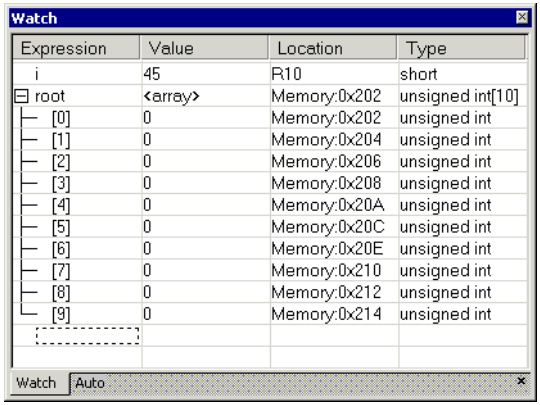


Figure 16: Watching variables in the Watch window

6 Execute some more steps to see how the values of `i` and `root` change.

7 To remove a variable from the Watch window, select it and press **Delete**.

SETTING AND MONITORING BREAKPOINTS

The IAR C-SPY Debugger contains a powerful breakpoint system with many features. For detailed information about the different breakpoints, see *The breakpoint system*, page 129.

The most convenient way is usually to set breakpoints interactively, simply by positioning the insertion point in or near a statement and then choosing the **Toggle Breakpoint** command.

1 Set a breakpoint on the statement `get_fib(i)` using the following procedure: First, click the `Utilities.c` tab in the editor window and click in the statement to position the insertion point. Then choose **Edit>Toggle Breakpoint**.

Alternatively, click the **Toggle Breakpoint** button on the toolbar.

A breakpoint will be set at this statement. The statement will be highlighted and there will be an **X** in the margin to show that there is a breakpoint there.

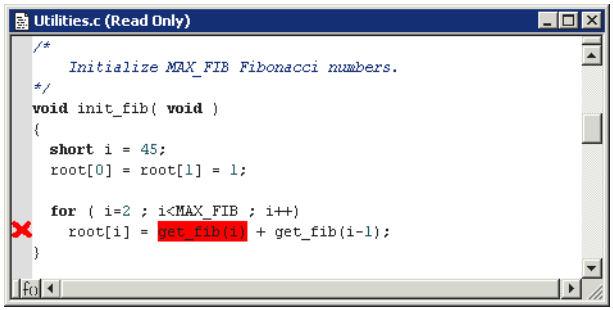


Figure 17: Setting breakpoints

To view all defined breakpoints, choose **View>Breakpoints** to open the Breakpoints window. You can find information about the breakpoint execution in the Debug Log window.

Executing up to a breakpoint

2 To execute your application until it reaches the breakpoint, choose **Debug>Go**.

Alternatively, click the **Go** button on the toolbar.

The application will execute up to the breakpoint you set. The Watch window will display the value of the `root` expression and the Debug Log window will contain information about the breakpoint.

3 Select the breakpoint and choose **Edit>Toggle Breakpoint** to remove the breakpoint.

DEBUGGING IN DISASSEMBLY MODE

Debugging with C-SPY is usually quicker and more straightforward in C/C++ source mode. However, if you want to have full control over low-level routines, you can debug in disassembly mode where each step corresponds to one assembler instruction. C-SPY lets you switch freely between the two modes.

1 First reset your application by clicking the **Reset** button on the toolbar.

2 Choose **View>Disassembly** to open the Disassembly window, if it is not already open.You will see the assembler code corresponding to the current C statement.

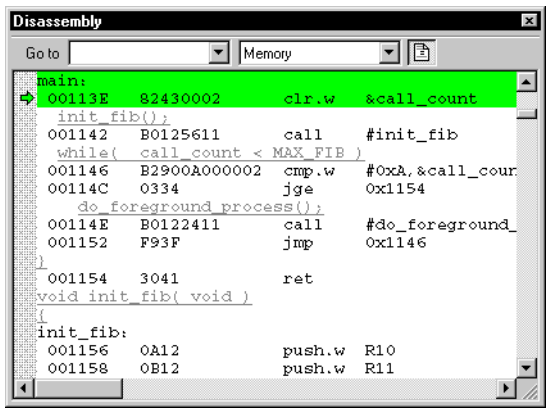


Figure 18: Debugging in disassembly mode

Try the different step commands also in the Disassembly window.

MONITORING REGISTERS

The Register window lets you monitor and modify the contents of the processor registers.

1 Choose **View>Register** to open the Register window.

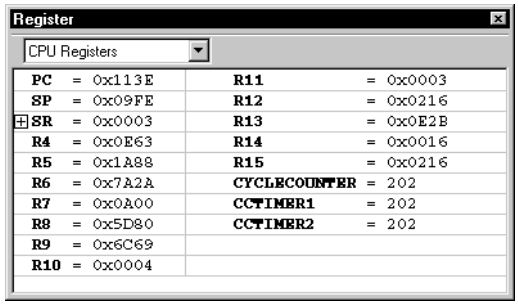


Figure 19: Register window

2 **Step Over** to execute the next instructions, and watch how the values change in the Register window.

3 Close the Register window.

MONITORING MEMORY

The Memory window lets you monitor selected areas of memory. In the following example, the memory corresponding to the variable `root` will be monitored.

1 Choose **View>Memory** to open the Memory window.

2 Make the Utilities.c window active and select `root`. Then drag it from the C source window to the Memory window.

The memory contents in the Memory window corresponding to `root` will be selected.

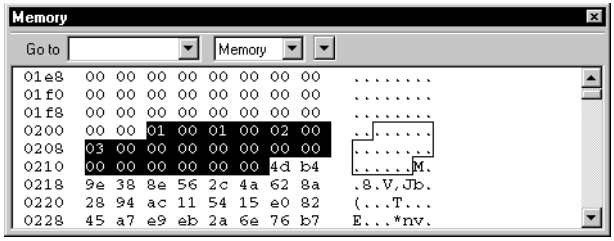


Figure 20: Monitoring memory

3 To display the memory contents as 16-bit data units, choose the **x2 Units** command from the drop-down arrow menu on the Memory window toolbar.

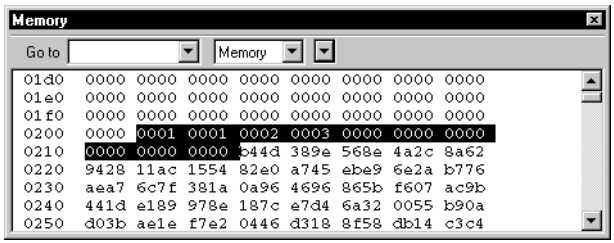


Figure 21: Displaying memory contents as 16-bit units

If not all of the memory units have been initialized by the `init_fib` function of the C application yet, continue to step over and you will notice how the memory contents will be updated.

You can change the memory contents by editing the values in the Memory window. Just place the insertion point at the memory content that you want to edit and type the desired value.

Close the Memory window.

VIEWING TERMINAL I/O

Sometimes you might need to debug constructions in your application that make use of `stdin` and `stdout` without the possibility of having hardware support. C-SPY lets you simulate `stdin` and `stdout` by using the Terminal I/O window.

Note: The Terminal I/O window is only available in C-SPY if you have linked your project using the output option **With I/O emulation modules**. This means that some low-level routines will be linked that direct `stdin` and `stdout` to the Terminal I/O window, see *Linking the application*, page 34.

1. Choose **View>Terminal I/O** to display the output from the I/O operations.

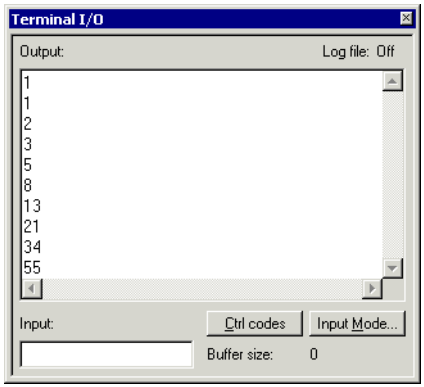


Figure 22: Output from the I/O operations

The contents of the window depends on how far you have executed the application.

REACHING PROGRAM EXIT

1. To complete the execution of your application, choose **Debug>Go**.

Alternatively, click the **Go** button on the toolbar.

As no more breakpoints are encountered, C-SPY reaches the end of the application and a `program exit reached` message is printed in the Debug Log window.

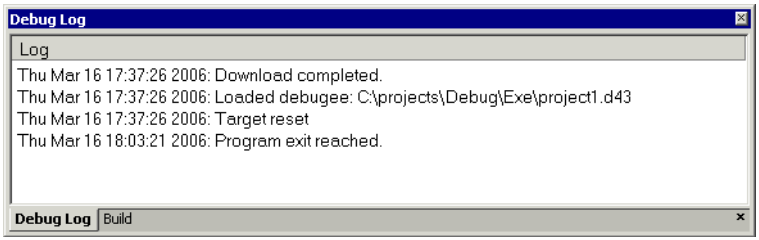


Figure 23: Reaching program exit in C-SPY

All output from the application has now been displayed in the Terminal I/O window.

If you want to start again with the existing application, choose **Debug>Reset**, or click the **Reset** button on the toolbar.

2. To exit from C-SPY, choose **Debug>Stop Debugging**. Alternatively, click the **Stop Debugging** button on the toolbar. The Embedded Workbench workspace is displayed.

C-SPY also provides many other debugging facilities. Some of these—for example macros and interrupt simulation—are described in the following tutorial chapters.

For further details about how to use C-SPY, see *Part 4. Debugging*. For reference information about the features of C-SPY, see *Part 7. Reference information* and the online help system.

Mixing C and assembler modules

In some projects it may be necessary to write certain pieces of source code in assembler language. The chapter first demonstrates how the compiler can be helpful in examining the calling convention, which you need to be familiar with when calling assembler modules from C/C++ modules or vice versa. Furthermore, this chapter demonstrates how you can easily combine source modules written in C with assembler modules, but the procedure is applicable to projects containing source modules written in C++, too.

This tutorial assumes that you are familiar with the basics of the IAR Embedded Workbench® IDE described in the previous tutorial chapters.

Examining the calling convention

When writing an assembler routine that will be called from a C routine, it is necessary to be aware of the calling convention used by the compiler. By creating skeleton code in C and letting the compiler produce an assembler output file from it, you can study the produced assembler output file and find the details of the calling convention.

In this example you will make the compiler create an assembler output file from the file `Utilities.c`.

1. Create a new project in the workspace `tutorials` used in previous tutorials, and name the project `project2`.
2. Add the files `Tutor.c` and `Utilities.c` to the project.

   To display an overview of the workspace, click the **Overview** tab available at the bottom of the Workspace window. To view only the newly created project, click the **project2** tab. For now, the **project2** view should be visible.
3. To set options, choose **Project>Options**, and select the **General Options** category. On the **Target** page, choose **msp430F149** from the **Device** drop-down menu.
4. To set options on file level node, in the Workspace window, select the file `Utilities.c`.

   Choose **Project>Options**. You will notice that only the **C/C++ Compiler** and **Custom Build** categories are available.

5 In the **C/C++ Compiler** category, select **Override inherited settings** and verify the following settings:

Page	Option
Optimizations	Size: None (Best debug support)
List	Output assembler file Include source Include runtime information (deselected).

Table 5: Compiler options for project2

Note: In this example it is necessary to use a low optimization level when compiling the code to show local and global variable accesses. If a higher level of optimization is used, the required references to local variables can be removed. The actual function declaration is not changed by the optimization level.

6 Click **OK** and return to the Workspace window.

7 Compile the file `Utilities.c`. You can find the output file `Utilities.s43` in the subdirectory `projects\debug\list`.

8 To examine the calling convention and to see how the C or C++ code is represented in assembler language, open the file `Utilities.s43`.

You can now study where and how parameters are passed, how to return to the program location from where a function was called, and how to return a resulting value. You can also see which registers an assembler-level routine must preserve.

To obtain the correct interface for your own application functions, you should create skeleton code for each function that you need.

For more information about the calling convention used in the compiler, see the *MSP430 IAR C/C++ Compiler Reference Guide.*

Adding an assembler module to the project

This tutorial demonstrates how you can easily create a project containing both assembler modules and C modules. You will also compile the project and view the assembler output list file.

SETTING UP THE PROJECT

1 Modify `project2` by removing the file `Utilities.c` and adding the file `Utilities.s43` instead..

Note: To view assembler files in the **Add files** dialog box, choose **Project>Add Files** and choose **Assembler Files** from the **Files of type** drop-down list.

2 Select the project level node in the Workspace window, choose **Project>Options**. Use the default settings in the **General Options**, **C/C++ Compiler**, and **Linker** categories. Select the **Assembler** category, click the **List** tab, and select the option **Output list file**.

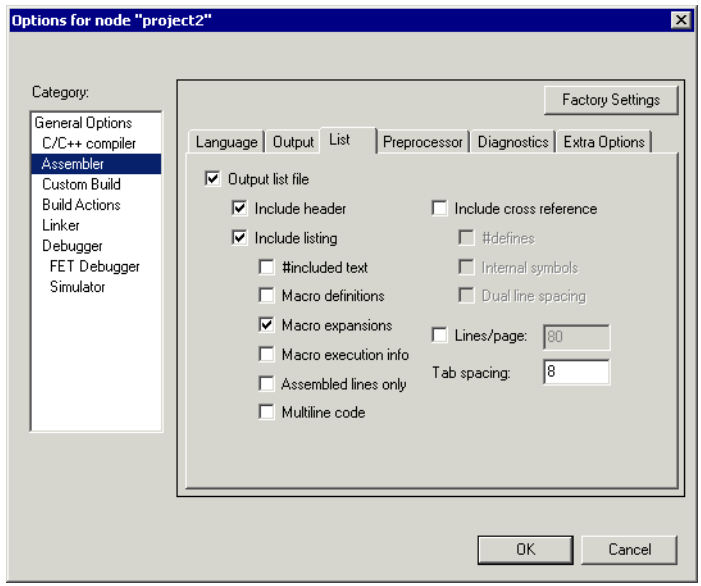


Figure 24: Assembler settings for creating a list file

Click **OK**.

3 Select the file `Utilities.s43` in the Workspace window and choose **Project>Compile** to assemble it.

Assuming that the source file was assembled successfully, the file `Utilities.r43` will be created, containing the linkable object code.

Viewing the assembler list file

4 Open the list file by double-clicking the file `Utilities.lst` available in the `Output` folder icon in the Workspace window.

The *end* of the file contains a summary of errors and warnings that were generated.

For further details of the list file format, see the *MSP430 IAR Assembler Reference Guide*.

5 Choose **Project>Make** to relink `project2`.

6. Start C-SPY to run the `project2.d43` application and see that it behaves like in the previous tutorial.
7. Exit the debugger when you are done.

Using C++

In this chapter, C++ is used to create a C++ class. The class is then used for creating two independent objects, and the application is built and debugged. We also show an example of how to set a conditional breakpoint.

This tutorial assumes that you are familiar with the basics of the IAR Embedded Workbench® IDE described in the previous tutorial chapters.

Note that, depending on what IAR product package you have installed, support for C++ may or may not be included. This tutorial assumes that there is support for C++.

Creating a C++ application

This tutorial will demonstrate how to use the MSP430 IAR Embedded Workbench C++ features. The tutorial consists of two files:

- `Fibonacci.cpp` creates a class `fibonacci` that can be used to extract a series of Fibonacci numbers
- `CPPtutor.cpp` creates two objects, `fib1` and `fib2`, from the class `fibonacci` and extracts two sequences of Fibonacci numbers using the `fibonacci` class.

To demonstrate that the two objects are independent of each other, the numbers are extracted at different speeds. A number is extracted from `fib1` each turn in the loop while a number is extracted from `fib2` only every second turn.

The object `fib1` is created using the default constructor while the definition of `fib2` uses the constructor that takes an integer as its argument.

COMPILING AND LINKING THE C++ APPLICATION

1. In the workspace `tutorials` used in the previous chapters, create a new project, `project3`.
2. Add the files `Fibonacci.cpp` and `CPPtutor.cpp` to `project3`.

3 Choose **Project>Options** and make sure the following options are selected:

Category	Page	Option
General Options	Target	Device: msp430F149
	Library Configuration	Library: Normal DLIB
C/C++ Compiler	Language	Embedded C++

Table 6: Project options for Embedded C++ tutorial

All you need to do to switch to the C++ programming language is to select the options **Normal DLIB (C/EC++ Library)** and **Embedded C++.**

4 Choose **Project>Make** to compile and link your application.

Alternatively, click the **Make** button on the toolbar. The **Make** command compiles and links those files that have been modified.

5 Choose **Project>Debug** to start the IAR C-SPY® Debugger.

SETTING A BREAKPOINT AND EXECUTING TO IT

1 Open the CPPtutor.cpp window if it is not already open.

2 To see how the object is constructed, set a breakpoint on the C++ object `fib1` on the following line:

```
fibonacci fib1;
```

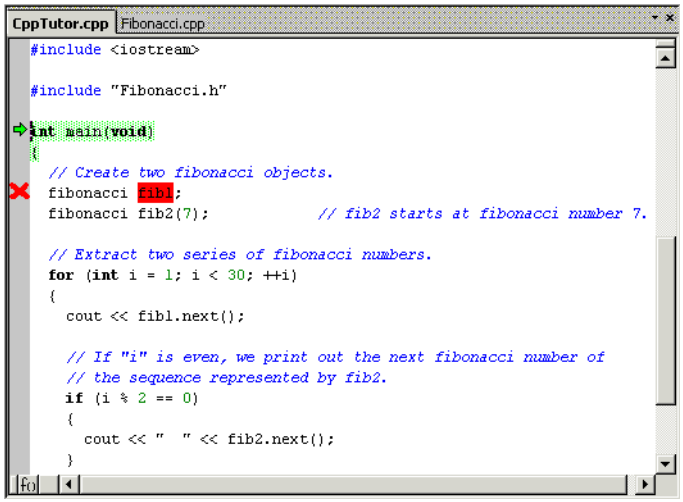


Figure 25: Setting a breakpoint in CPPtutor.cpp

3 Choose **Debug>Go**, or click the **Go** button on the toolbar.

The cursor should now be placed at the breakpoint.

4 To step into the constructor, choose **Debug>Step Into** or click the **Step Into** button in the toolbar. Then click **Step Out** again.

5 **Step Over** until the line:

```
cout << fib1.next();
```

Step Into until you are in the function `next` in the file `Fibonacci.cpp`.

6 Use the **Go to function** button in the lower left corner of the editor window to find and go to the function `nth` by double-clicking the function name. Set a breakpoint on the function call `nth(n-1)` at the line

```
value = nth(n-1) + nth(n-2);
```

7 It can be interesting to backtrace the function calls a few levels down and to examine the value of the parameter for each function call. By adding a condition to the breakpoint, the break will not be triggered until the condition is true, and you will be able to see each function call in the Call Stack window.

To open the Breakpoints window, choose **View>Breakpoints**. Select the breakpoint in the Breakpoints window, right-click to open the context menu, and choose **Edit** to open the **Edit Breakpoints** dialog box. Set the value in the **Skip count** text box to `4` and click **OK**.

Close the dialog box.

Looking at the function calls

8 Choose **Debug>Go** to execute the application until the breakpoint condition is fulfilled.

9 When C-SPY stops at the breakpoint, choose **View>Call Stack** to open the Call Stack window.

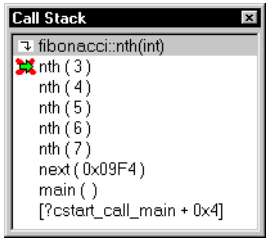


Figure 26: Inspecting the function calls

There are five instances of the function `nth` displayed on the call stack. Because the Call Stack window displays the values of the function parameters, you can see the different values of `n` in the different function instances.

You can also open the Register window to see how it is updated as you trace the function calls by double-clicking on the function instances.

PRINTING THE FIBONACCI NUMBERS

1. Open the Terminal I/O window from the **View** menu.
2. Remove the breakpoints and run the application to the end and verify the Fibonacci sequences being printed.

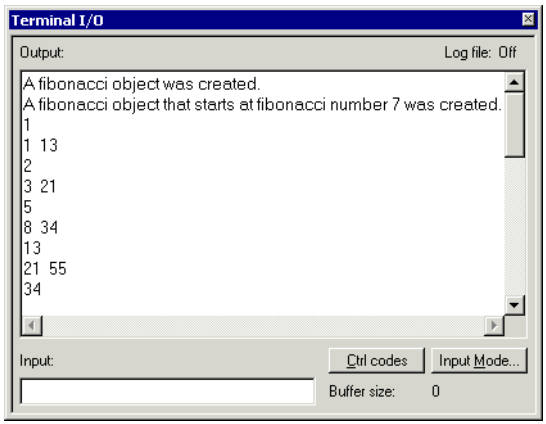


Figure 27: Printing Fibonacci sequences

Simulating an interrupt

In this tutorial an interrupt handler for a serial port is added to the project. The Fibonacci numbers will be read from an on-chip communication peripheral device (USART0).

This tutorial will show how the MSP430 IAR C/C++ Compiler interrupt keyword and the #pragma vector directive can be used. The tutorial will also show how an interrupt can be simulated using the features that support interrupts, breakpoints, and macros. Notice that this example does not describe an exact simulation; the purpose is to illustrate a situation where C-SPY® macros, breakpoints, and the interrupt system can be useful to simulate hardware.

This tutorial assumes that you are familiar with the basics of the IAR Embedded Workbench® IDE described in the previous tutorial chapters.

Note that interrupt simulation is possible only when you are using the IAR C-SPY Simulator.

Adding an interrupt handler

This section will demonstrate how to write an interrupt in an easy way. It starts with a brief description of the application used in this project, followed by a description of how to set up the project.

THE APPLICATION—A BRIEF DESCRIPTION

The interrupt handler will read values from the serial communication port receive register (USART0), `U0RXBUF`. It will then print the value. The main program enables interrupts and starts printing periods (.) in the foreground process while waiting for interrupts.

WRITING AN INTERRUPT HANDLER

The following lines define the interrupt handler used in this tutorial (the complete source code can be found in the file `Interrupt.c` supplied in the `430\tutor` directory):

```
/* define the interrupt handler */
#pragma vector=USART0RX_VECTOR
__interrupt void uartRecieveHandler( void )
```

The `#pragma vector` directive is used for specifying the interrupt vector address—in this case the interrupt vector for the USART0 receive interrupt—and the keyword `__interrupt` is used for directing the compiler to use the calling convention needed for an interrupt function.

For detailed information about the extended keywords and pragma directives used in this tutorial, see the *MSP430 IAR C/C++ Compiler Reference Guide*.

SETTING UP THE PROJECT

1. Add a new project—`project4`—to the workspace `tutorials` used in previous tutorials.
2. Add the files `Utilities.c` and `Interrupt.c` to it.
3. In the Workspace window, select the project level node, and choose **Project>Options**. Select the **General Options** category, and click the **Target** tab. Choose **msp430F149** from the **Core** drop-down menu.

   In addition, make sure the factory settings are used in the **C/C++ Compiler** and **Linker** categories.

   Next you will set up the simulation environment.

Setting up the simulation environment

The C-SPY interrupt system is based on the cycle counter. You can specify the amount of cycles to pass before C-SPY generates an interrupt.

To simulate the input to USART0, values will be read from the file `InputData.txt`, which contains the Fibonacci series. You will set an *immediate read breakpoint* on the USART0 receive register, `U0RXBUF`, and connect a user-defined macro function to it (in this example the `Access` macro function). The macro reads the Fibonacci values from the text file.

Whenever an interrupt is generated, the interrupt routine will read `U0RXBUF` and the breakpoint will be triggered, the `Access` macro function will be executed and the Fibonacci values will be fed into the USART0 receive register.

The immediate read breakpoint will trigger the break *before* the processor reads the U0RXBUF register, allowing the macro to store a new value in the register that is immediately read by the instruction.

This section will demonstrate the steps involved in setting up the simulator for simulating a serial port interrupt. The steps involved are:

- Defining a C-SPY setup file which will open the file InputData.txt and define the Access macro function
- Specifying C-SPY options
- Building the project
- Starting the simulator
- Specifying the interrupt request
- Setting the breakpoint and associating the Access macro function to it.

Note: For a simple example of a system timer interrupt simulation, see *Simulating a simple interrupt*, page 186.

DEFINING A C-SPY SETUP MACRO FILE

In C-SPY, you can define setup macros that will be registered during the C-SPY startup sequence. In this tutorial you will use the C-SPY macro file SetupSimple.mac, available in the 430\tutor directory. It is structured as follows:

First the setup macro function execUserSetup is defined, which is automatically executed during C-SPY setup. Thus, it can be used to set up the simulation environment automatically. A message is printed in the Log window to confirm that this macro has been executed:

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

Then the file InputData.txt, which contains the Fibonacci series to be fed into USART0, will be opened:

```
  _fileHandle = __openFile(
"$TOOLKIT_DIR$\\tutor\\InputData.txt", "r" );
```

After that, the macro function `Access` is defined. It will read the Fibonacci values from the file `InputData.txt`, and assign them to the receive register address:

```
Access()
{
  __message "Access() called\n";
  __var _fibValue;
  if( 0 == __readFile( _fileHandle, &_fibValue ) )
  {
    U0RXBUF = _fibValue;
  }
}
```

You will have to connect the `Access` macro to an immediate read breakpoint. However, this will be done at a later stage in this tutorial.

Finally, the file contains two macro functions for managing correct file handling at reset and exit.

For detailed information about macros, see the chapters *Using the C-SPY® macro system* and *C-SPY® macros reference*.

Next you will specify the macro file and set the other C-SPY options needed.

SPECIFYING C-SPY OPTIONS

1. To select C-SPY options, choose **Project>Options**. In the **Debugger** category, click the **Setup** tab.
2. Use the **Use macro file** browse button to specify the macro file to be used:

   `SetupSimple.mac`

   Alternatively, use an argument variable to specify the path:

   `$TOOLKIT_DIR$\tutor\SetupSimple.mac`

See *Argument variables summary*, page 279, for details.

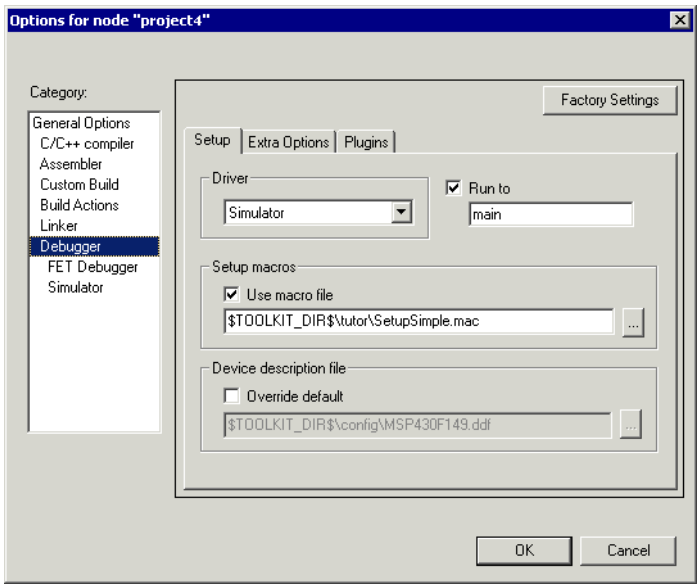


Figure 28: Specifying setup macro file

3 Set the **Device description file** option to `msp430F149.ddf`. This file provides interrupt definitions which are needed by the interrupt system.

4 Select **Run to main** and click **OK**. This will ensure that the debug session will start by running to the `main` function.

The project is now ready to be built.

BUILDING THE PROJECT

1 Compile and link the project by choosing **Project>Make**.

Alternatively, click the **Make** button on the toolbar. The **Make** command compiles and links those files that have been modified.

STARTING THE SIMULATOR

1 Start the IAR C-SPY Debugger to run the `project4` project.

The Interrupt.c window is displayed (among other windows). Click in it to make it the active window.

2 Examine the Log window. Note that the macro file has been loaded and that the `execUserSetup` function has been called.

SPECIFYING A SIMULATED INTERRUPT

Now you will specify your interrupt to make it simulate an interrupt every 2000 cycles.

1 Choose **Simulator>Interrupt Setup** to display the **Interrupt Setup** dialog box. Click **New** to display the **Edit Interrupt** dialog box and make the following settings for your interrupt:

Setting	Value	Description
Interrupt	`USART0RX_VECTOR`	Specifies which interrupt to use; the name is defined in the `ddf` file.
Description	As is	The interrupt definition that the simulator uses to be able to simulate the interrupt correctly.
First activation	`4000`	Specifies the first activation moment for the interrupt. The interrupt is activated when the cycle counter has passed this value.
Repeat Interval	`2000`	Specifies the repeat interval for the interrupt, measured in clock cycles.
Hold time	`Infinite`	Hold time, not used here.
Probability %	`100`	Specifies probability. 100% specifies that the interrupt will occur at the given frequency. Another percentage might be used for simulating a more random interrupt behavior.
Variance %	`0`	Time variance, not used here.

Table 7: Interrupts dialog box

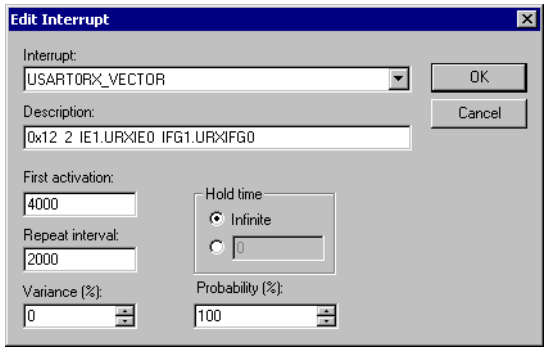


Figure 29: Inspecting the interrupt settings

During execution, C-SPY will wait until the cycle counter has passed the activation time. When the current assembler instruction is executed, C-SPY will generate an interrupt which is repeated approximately every 2000 cycles.

2 When you have specified the settings, click **OK** to close the **Edit Interrupt** dialog box, and then click **OK** to close the **Interrupt Setup** dialog box.

For information about how you can use the system macro `__orderInterrupt` in a C-SPY setup file to automate the procedure of defining the interrupt, see *Using macros for interrupts and breakpoints*, page 65.

SETTING AN IMMEDIATE BREAKPOINT

By defining a macro and connecting it to an immediate breakpoint, you can make the macro simulate the behavior of a hardware device, for instance an I/O port, as in this tutorial. The immediate breakpoint will not halt the execution, only temporarily suspend it to check the conditions and execute any connected macro.

In this example, the input to the USART0 is simulated by setting an immediate read breakpoint on the `U0RXBUF` address and connecting the defined `Access` macro to it. The macro will simulate the input to the USART0. These are the steps involved:

1 Choose **View>Breakpoints** to open the Breakpoints window, right-click to open the context menu, choose **New Breakpoint>Immediate** to open the **Immediate** tab.

2 Add the following parameters for your breakpoint.

Setting	Value	Description
Break at	`U0RXBUF`	Receive buffer address.
Access Type	`Read`	The breakpoint type (Read or Write)

Table 8: Breakpoints dialog box

Setting	Value	Description
Action	`Access()`	The macro connected to the breakpoint.

Table 8: Breakpoints dialog box (Continued)

During execution, when C-SPY detects a read access from the `U0RXBUF` address, C-SPY will temporarily suspend the simulation and execute the `Access` macro. The macro will read a value from the file `InputData.txt` and write it to `U0RXBUF`. C-SPY will then resume the simulation by reading the receive buffer value in `U0RXBUF`.

3 Click **OK** to close the breakpoints dialog box.

For information about how you can use the system macro `__setSimBreak` in a C-SPY setup file to automate the breakpoint setting, see *Using macros for interrupts and breakpoints*, page 65.

Simulating the interrupt

In this section you will execute your application and simulate the serial port interrupt.

EXECUTING THE APPLICATION

1 Step through the application and stop when it reaches the `while` loop, where the application waits for input.

2 In the Interrupt.c source window, locate the function `uartReciveHandler`.

3 Place the insertion point on the `++callCount;` statement in this function and set a breakpoint by choosing **Edit>Toggle Breakpoint**, or click the **Toggle Breakpoint** button on the toolbar. Alternatively, use the context menu.

If you want to inspect the details of the breakpoint, choose **Edit>Breakpoints**.

4 Open the Terminal I/O window and run your application by choosing **Debug>Go** or clicking the **Go** button on the toolbar.

The application should stop in the interrupt function.

5 Click **Go** again in order to see the next number being printed in the Terminal I/O window.

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

The Terminal I/O window will display the Fibonacci series.

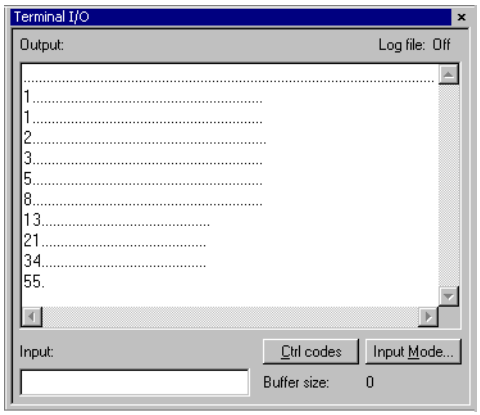


Figure 30: Printing the Fibonacci values in the Terminal I/O window

Using macros for interrupts and breakpoints

To automate the setting of breakpoints and the procedure of defining interrupts, the system macros `__setSimBreak` and `__orderInterrupt`, respectively, can be executed by the setup macro `execUserSetup`.

The file `SetupAdvanced.mac` is extended with system macro calls for setting the breakpoint and specifying the interrupt:

```
SimulationSetup()
{...
  _interruptID = __orderInterrupt( "USART0RX_VECTOR", 4000,
                                   2000, 0, 1, 0, 100 );

  if( -1 == _interruptID )
  {
    __message "ERROR: failed to order interrupt";
  }

  _breakID = __setSimBreak( "U0RXBUF", "R", "Access()" );

}
```

By replacing the file `SetupSimple.mac`, used in the previous tutorial, with the file `SetupAdvanced.mac`, setting the breakpoint and defining the interrupt will be automatically performed at C-SPY startup. Thus, you do not need to start the simulation by manually filling in the values in the **Interrupts** and **Breakpoints** dialog boxes.

Note: Before you load the file `SetupAdvanced.mac` you should remove the previously defined breakpoint and interrupt.

Working with library modules

This tutorial demonstrates how to create library modules and how you can combine an application project with a library project.

This tutorial assumes that you are familiar with the basics of the IAR Embedded Workbench® IDE described in the previous tutorial chapters.

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 applications. To avoid having to assemble a routine each time the routine is needed, you can store such routines as object files, that is, 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 XAR Library Builder to build libraries. The IAR XLIB Librarian lets you manipulate libraries. It allows you to:

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

The Main.s43 program

The `Main.s43` program uses a routine called `r_shift` to right-shift the contents of the register `R4` the number of times of the value stored in register `R5`. The result is returned in `R4`. The `EXTERN` directive declares `r_shift` as an external symbol, to be resolved at link time.

A copy of the program is provided in the `430\tutor` directory.

The library routines

The 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 registers `A` and `B` and return the result in `A`. The file containing these library routines is called `Shifts.s43`, and a copy is provided with the product.

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 referenced 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 *MSP430 IAR Assembler Reference Guide*.

CREATING A NEW PROJECT

1. In the workspace `tutorials` used in previous chapters, add a new project called `project5`.
2. Add the file `Main.s43` to the new project.
3. To set options, choose **Project>Options.** Select the **General Options** category and click the **Library Configuration** tab. Choose **None** from the **Library** drop-down list, which means that a standard C/C++ library will not be linked.

   The default options are used for the other option categories.
4. To assemble the file `Main.s43`, choose **Project>Compile**.

You can also click the **Compile** button on the toolbar.

CREATING A LIBRARY PROJECT

Now you are ready to create a library project.

1. In the same workspace `tutorials`, add a new project called `tutor_library`.
2. Add the file `Shift.s43` to the project.
3. To set options, choose **Project>Options.** In the **General Options** category, verify the following settings:

Page	Option
Output	Output file: Library

Table 9: XLINK options for a library project

Page	Option
Library Configuration	Library: None

Table 9: XLINK options for a library project (Continued)

Note that **Library Builder** appears in the list of categories, which means that the IAR XAR Library Builder is added to the build tool chain. It is not necessary to set any XAR-specific options for this tutorial.

Click **OK.**

4 Choose **Project>Make**.

The library output file `tutor_library.r43` has now been created.

USING THE LIBRARY IN YOUR APPLICATION PROJECT

You can now add your library containing the shift routine to `project5`.

1 In the Workspace window, click the **project5** tab. Choose **Project>Add Files** and add the file `tutor_library.r43` located in the `projects\Debug\Exe` directory. Click **Open**.

2 Click **Make** to build your project.

3 You have now combined a library with an executable project, and the application is ready to be executed. For information about how to manipulate the library, see the *IAR Linker and Library Tools Reference Guide*.

Part 3. Project management and building

This part of the MSP430 IAR Embedded Workbench® IDE User Guide contains the following chapters:

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

The development environment

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

The IAR Embedded Workbench IDE

The IAR Embedded Workbench IDE is the framework where all necessary tools are seamlessly integrated: a C/C++ compiler, an assembler, the IAR XLINK Linker, the IAR XAR Library Builder, the IAR XLIB Librarian, an editor, a project manager with Make utility, and the IAR C-SPY® Debugger, a high-level language debugger.

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

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

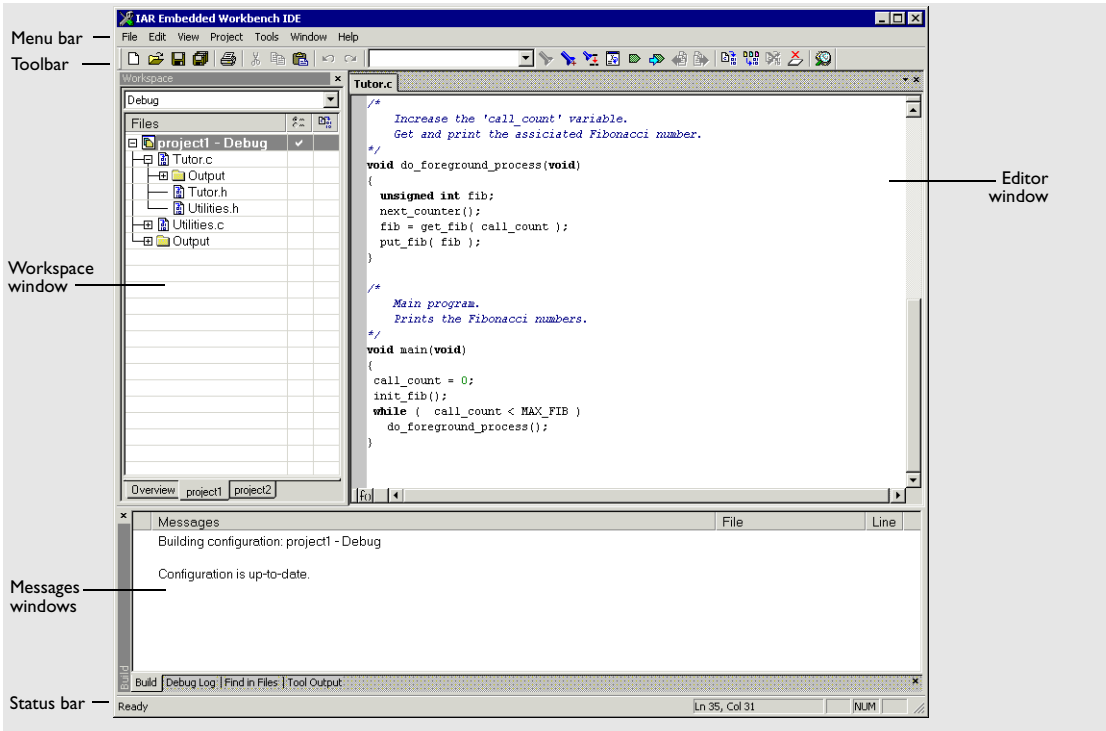


Figure 31: IAR Embedded Workbench IDE window

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

RUNNING THE IAR EMBEDDED WORKBENCH IDE

Click the **Start** button on the taskbar and choose **All Programs>IAR Systems>IAR Embedded Workbench for MSP430 V3>IAR Embedded Workbench**.

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

Double-clicking the workspace filename

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

EXITING

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

Customizing the environment

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

ORGANIZING THE WINDOWS ON THE SCREEN

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

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

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

Using docked versus floating windows

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

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

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

Note: The editor window is always docked. When you open the editor window, its placement is decided automatically depending on other currently open windows. For more information about how to work with the editor window, see *Using the IAR Embedded Workbench editor*, page 95.

Organizing windows

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

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

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

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

CUSTOMIZING THE IDE

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

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

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

For reference information about the commands for customizing the IDE, see *Tools menu*, page 286. You can also find further information related to customizing the editor in the section *Customizing the editor environment*, page 101. For further information about customizations related to C-SPY, see *Part 4. Debugging*.

COMMUNICATING WITH EXTERNAL TOOLS

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

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

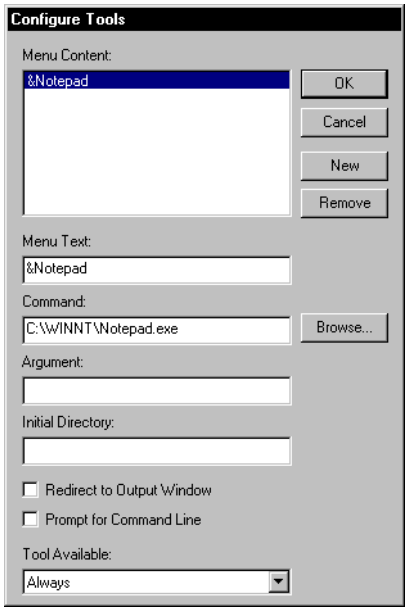


Figure 32: Configure Tools dialog box

For reference information about this dialog box, see *Configure Tools dialog box*, page 303.

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

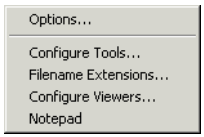


Figure 33: Customized Tools menu

Note: If you intend to add an external tool to the standard build tool chain, see *Extending the tool chain*, page 93.

Adding command line commands

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

1 To add commands to the **Tools** menu, you must specify an appropriate command shell.

Type one of the following command shells in the **Command** text box:

System	Command shell
Windows 98/Me	`command.com`
Windows NT/2000/XP	`cmd.exe` (recommended) or `command.com`

Table 10: Command shells

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

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

`/C` *`name`*

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

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

Example

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

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

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

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

Managing projects

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

The project model

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

The IAR Embedded Workbench IDE is a flexible environment for developing projects also with a number of different target processors in the same project, and a selection of tools for each target processor.

HOW PROJECTS ARE ORGANIZED

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

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

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

Projects and workspaces

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

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

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

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

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

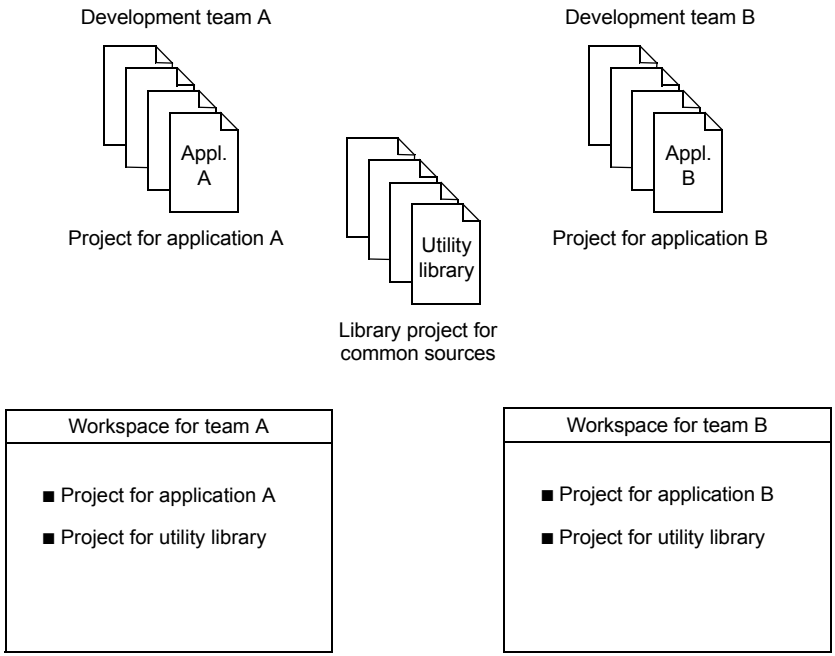


Figure 34: Examples of workspaces and projects

For an example where a library project has been combined with an application project, see the chapter *Working with library modules* in *Part 2. Tutorials*.

Projects and build configurations

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

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

- Project A - Device 1:Release
- Project A - Device 1:Debug
- Project A - Device 2:Release
- Project A - Device 2:Debug

Groups

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

Source files

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

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

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

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

CREATING AND MANAGING WORKSPACES

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

For reference information about these menus, menu commands, and dialog boxes, see the chapter *IAR Embedded Workbench® IDE reference*.

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

- Creating a workspace.

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

- Adding new or existing projects to the workspace.

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

- Creating groups.

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

- Adding files to the project.

  A file can be added either to the project's top node or to a group within the project.

- Creating new build configurations.

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

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

  Note that you do not have to use the same tool chain for the new build configuration as for other build configurations in the same project.

- Excluding groups and files from a build configuration.

Note that the icon indicating the excluded group or file will change to white in the Workspace window.

- Removing items from a project.

For a detailed example, see *Creating an application project*, page 25.

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

Drag and drop

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

Source file paths

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

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

Navigating project files

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

VIEWING THE WORKSPACE

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

1 Choose which project you want to view by clicking its tab at the bottom of the Workspace window.

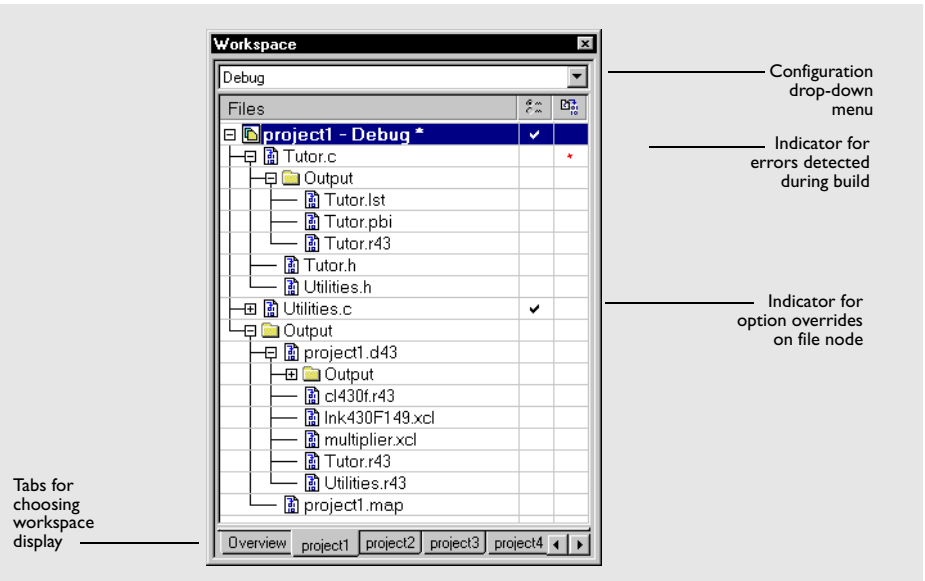


Figure 35: Displaying a project in the Workspace window

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

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

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

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

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

An overview of all project members is displayed.

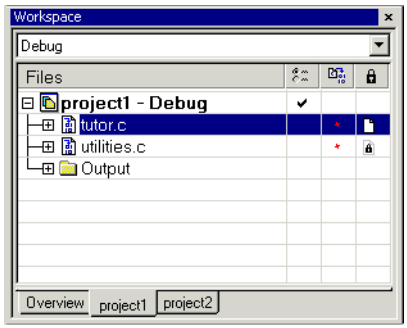


Figure 36: Workspace window—an overview

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

DISPLAYING BROWSE INFORMATION

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

To open the Source Browser window, choose **View>Source Browser**. The Source Browser window is by default docked with the Workspace window. Source browse information is displayed for the active build configuration. For reference information, see *Source Browser window*, page 253.

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

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

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

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

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

Source code control

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

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

Note: Different SCC systems use very different terminology even for some of the most basic concepts involved. It is important to keep this in mind when reading the description below.

INTERACTING WITH SOURCE CODE CONTROL SYSTEMS

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

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

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

Setting up an SCC project in the SCC client application

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

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

Connecting projects in IAR Embedded Workbench

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

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

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

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

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

Viewing the SCC states

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

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

There are also icons for some combinations of these states. Note that the interpretation of these states depends on the SCC client application you are using. For reference information about the icons and the different states they represent, see *Source code control states*, page 244.

For reference information about the commands available for accessing the SCC system, see *Source Code Control menu*, page 243.

Configuring the source code control system

To customize the source code control system, choose **Tools>Options** and click the **Source Code Control** tab. For reference information about the available commands, see *Terminal I/O page*, page 299.

Building

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

Building your application

The building process consists of the following steps:

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

To make the build process more efficient, you can use the **Batch Build** command. This gives you the possibility to perform several builds in one operation.

In addition to use the IAR Embedded Workbench IDE for building projects, it is also possible to use the command line utility `iarbuild.exe` for building projects.

For examples of building application and library projects, see *Part 2. Tutorials* in this guide. For further information about building library projects, see the *MSP430 IAR C/C++ Compiler Reference Guide*.

SETTING OPTIONS

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

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

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

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

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

Using the Options dialog box

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

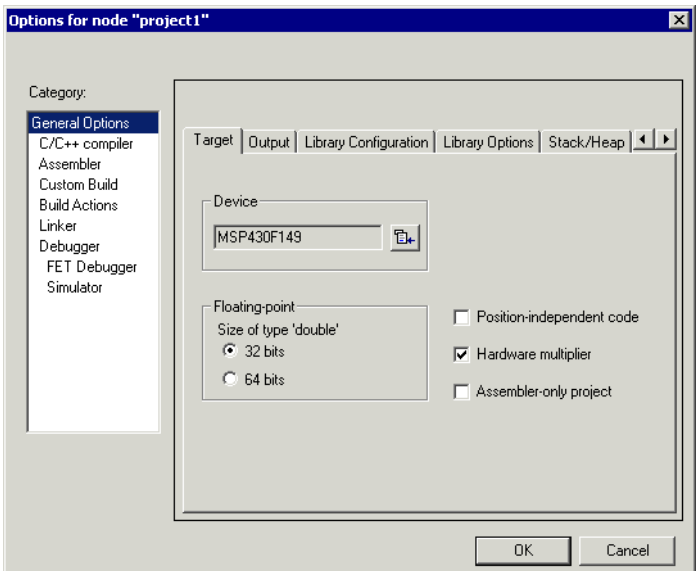


Figure 37: General options

The **Category** list allows you to select which building tool to set options for. The tools available in the **Category** list depends on which tools are included in your product. If you select **Library** as output file on the **Output** page, **Linker** will be replaced by **Library Builder** in the category list. When you select a category, one or more pages containing options for that component are displayed.

Click the tab corresponding to the type of options you want to view or change. To restore all settings to the default factory settings, click the **Factory Settings** button, which is available for all categories except **General Options** and **Custom Build**. Note that there are two sets of factory settings available: Debug and Release. Which one that will be used depends on your build configuration; see *New Configuration dialog box*, page 281.

For information about each option and how to set options, see the chapters *General options*, *Compiler options*, *Assembler options*, *Linker options*, *Library builder options*, *Custom build options*, and *Debugger options* in *Part 7. Reference information* in this guide. For information about options specific to the debugger driver you are using, see the part of this book that corresponds to your driver.

Note: If you add to your project a source file with a non-recognized filename extension, you cannot set options on that source file. However, you can add support for additional filename extensions. For reference information, see *Filename Extensions dialog box*, page 305.

BUILDING A PROJECT

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

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

For further reference information, see *Project menu*, page 277.

BUILDING MULTIPLE CONFIGURATIONS IN A BATCH

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

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

For detailed information about the **Batch Build** dialog box, see *Batch Build dialog box*, page 284.

CORRECTING ERRORS FOUND DURING BUILD

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

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

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

For reference information about the Build messages window, see *Build window*, page 261.

BUILDING FROM THE COMMAND LINE

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

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

Parameter	Description
project`.ewp`	Your IAR Embedded Workbench IDE project file.
`-clean`	Removes any intermediate files.
`-build`	Rebuilds and relinks all files in the current build configuration.
`-make`	Brings the current build configuration up to date by compiling, assembling, and linking only the files that have changed since the last build.
configuration	The name of the configuration you want to build, which can either be one of the predefined configurations Debug or Release, or a name that you define yourself. For more information about build configurations, see *Projects and build configurations*, page 81.
`-log errors`	Displays build error messages.
`-log warnings`	Displays build warning and error messages.
`-log info`	Displays build warning messages and messages issued by the `#pragma message` preprocessor directive.
`-log all`	Displays all messages generated from the build, for example compiler sign-on information and the full command line.

Table 11: iarbuild.exe command line options

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

Extending the tool chain

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

By specifying custom build options, on the **Custom tool configuration** page, the build commands treat the external tool and its associated files in the same way as the standard tools within the IAR Embedded Workbench IDE and their associated files. The relation between the external tool and its input files and generated output files is similar to the relation between the C/C++ Compiler, `c` files, `h` files, and `r43` files. See *Custom build options*, page 373, for details about available custom build options.

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

You must specify the name of the external tool. You can also specify any necessary command line options needed by the external tool, as well as the name of the output files generated by the external tool. Note that it is possible to use argument variables for substituting file paths.

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

It is possible to specify custom build options to any level in the project tree. The options you specify are inherited by any sublevel in the project tree.

TOOLS THAT CAN BE ADDED TO THE TOOL CHAIN

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

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

ADDING AN EXTERNAL TOOL

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

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

1. Add the file you want to work with to your project, for example `foo.lex`.
2. Select this file in the Workspace window and choose **Project>Options**. Select **Custom Build** from the list of categories.
3. In the **Filename extensions** field, type the filename extension `.lex`. Remember to specify the leading period (`.`).
4. In the **Command line** field, type the command line for executing the external tool, for example

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

   During the build process, this command line will be expanded to:

   ```
   flex foo.lex -ofoo.c
   ```

   Note the usage of *argument variables*. For further details of these variables, see *Argument variables summary*, page 279.

   Take special note of the use of `$FILE_BNAME$` which gives the base name of the input file, in this example appended with the `c` extension to provide a C source file in the same directory as the input file `foo.lex`.
5. In the **Output files** field, describe the output files that are relevant for the build. In this example, the tool Flex would generate two files—one source file and one header file. The text in the **Output files** text box for these two files would look like this:

   ```
   $FILE_BPATH$.c
   $FILE_BPATH$.h
   ```
6. If there are any additional files used by the external tool during the build, these should be added in the **Additional input files** field: for instance:

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

   This is important, because if the dependency files change, the conditions will no longer be the same and the need for a rebuild is detected.
7. Click **OK**.
8. To build your application, choose **Project>Make**.

Editing

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

Using the IAR Embedded Workbench editor

The integrated text editor allows editing of multiple files in parallel, and provides all basic editing features expected from a modern editor. In addition, it provides features specific to software development. It also recognizes C or C++ language elements.

EDITING A FILE

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

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

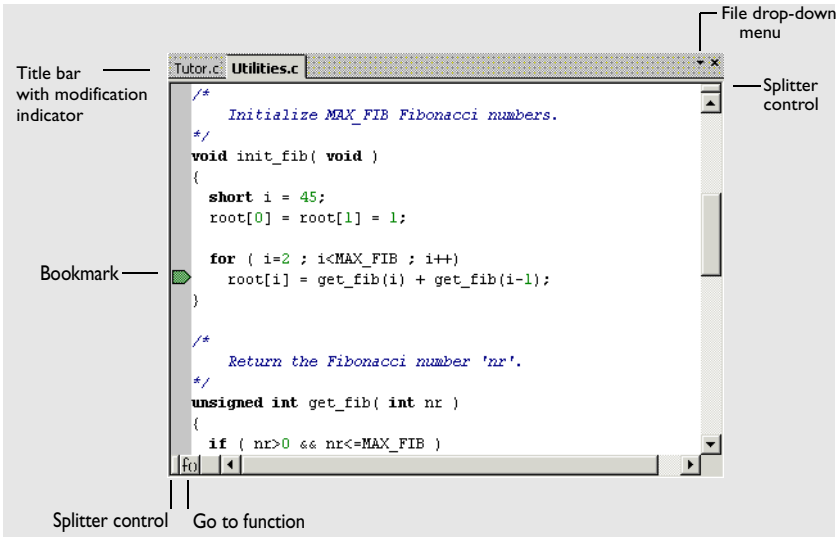


Figure 38: Editor window

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

The commands on the **Window** menu allow you to split the editor window into panes. On the **Window** menu you also find commands for opening multiple editor windows, as well as commands for moving files between different editor windows. For reference information about each command on the menu, see *Window menu*, page 308. For reference information about the editor window, see *Editor window*, page 248.

Accessing reference information for DLIB library functions

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

Using and customizing editor commands and shortcut keys

The **Edit** menu provides commands for editing and searching in editor windows. For instance, unlimited undo/redo by using the **Edit>Undo** and **Edit>Redo** commands, respectively. You can also find some of these commands on the context menu that appears when you right-click in the editor window. For reference information about each command, see *Edit menu*, page 267.

There are also editor shortcut keys for:

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

For detailed information about these shortcut keys, see *Editor key summary*, page 251.

To change the default shortcut key bindings, choose **Tools>Options**, and click the **Key Bindings** tab. For further details, see *Key Bindings page*, page 289.

Splitting the editor window into panes

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

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

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

Dragging and dropping of text

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

Syntax coloring

The IAR Embedded Workbench editor automatically recognizes the syntax of:

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

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

To change these styles, choose **Tools>Options**, and click the **Editor Colors and Fonts** tab in the **IDE Options** dialog box. For additional information, see *Editor Colors and Fonts page*, page 295.

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

1. In a text file, list all the keywords that you want to be automatically syntax-colored. Separate each keyword with either a space or a new line.
2. Choose **Tools>Options** and click the **Editor Setup Files** tab.
3. Select the **Use Custom Keyword File** option and specify your newly created text file. A browse button is available for your convenience.
4. Click the **Edit Colors and Fonts** tab and choose **User Keyword** from the **Syntax Coloring** list. Specify the font, color, and type style of your choice. For additional information, see *Editor Colors and Fonts page*, page 295.
5. In the editor window, type any of the keywords you listed in your keyword file; see how the keyword is syntax-colored according to your specification.

Automatic text indentation

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

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

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

To enable or disable the indentation:

1. Choose **Tools>Options**
2. Click the **Editor** tab
3. Select or deselect the **Auto indent** option.

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

For additional information, see *Configure Auto Indent dialog box*, page 292.

Matching brackets and parentheses

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

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

Figure 39: Parentheses matching in editor window

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

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

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

Displaying status information

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

Figure 40: Editor window status bar

USING AND ADDING CODE TEMPLATES

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

Enabling code templates

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

1. Choose **Tools>Options**.
2. Go to the **Editor Setup Files** page.
3. Select or deselect the **Use Code Templates** option.

4 In the text field, specify which template file you want to use; either the default file or one of your own template files. A browse button is available for your convenience.

Inserting a code template in your source code

To insert a code template in your source code, place the insertion point at the location where you want the template to be inserted and choose **Edit>Insert Template**. This command displays a list in the editor window from which you can choose a code template.

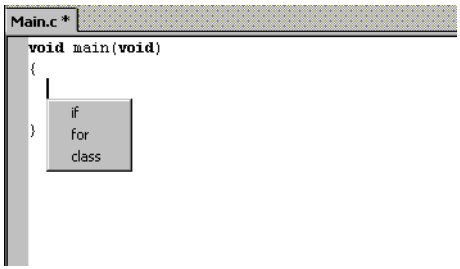


Figure 41: Editor window code template menu

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

Adding your own code templates

The source code templates are defined in a normal text file. The original template file `CodeTemplates.txt` is located in the `common\config` installation directory. The first time you use IAR Embedded Workbench, the original template file is copied to a directory for local settings, and this is the file that will be used by default if code templates are enabled. To use your own template file, follow the procedure described in *Enabling code templates*, page 99.

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

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

NAVIGATING IN AND BETWEEN FILES

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

- Switching between source and header files

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

- Function navigation

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

- Adding bookmarks

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

SEARCHING

There are several standard search functions available in the editor:

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

To use the **Quick search** text box on the toolbar, type the text you want to search for and press Enter. Press Esc to cancel the search. This is a quick method for searching for text in the active editor window.

To use the **Find**, **Replace**, **Find in Files**, and **Incremental Search** functions, choose the corresponding command from the **Edit** menu. For reference information about each search function, see *Edit menu*, page 267.

Customizing the editor environment

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

For details about these pages, see *Tools menu*, page 286.

USING AN EXTERNAL EDITOR

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

1. Select the option **Use External Editor**.
2. An external editor can be called in one of two ways, using the **Type** drop-down menu.

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

   DDE calls the external editor by using DDE (Windows Dynamic Data Exchange).
3. If you use the command line, specify the command line to pass to the editor, that is, the name of the editor and its path, for instance:

   ```
   C:\WINNT\NOTEPAD.EXE.
   ```

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

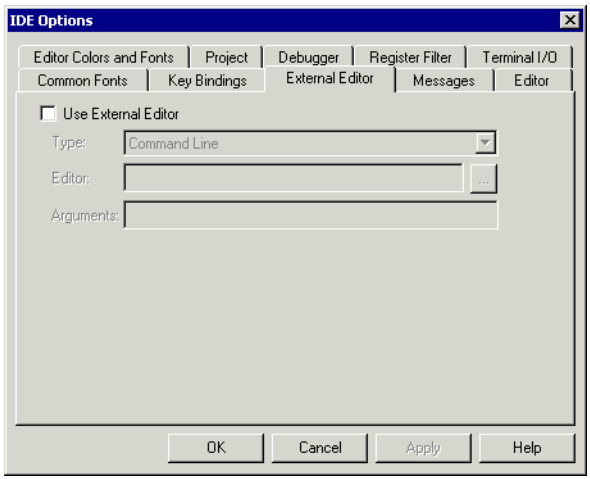


Figure 42: Specifying external command line editor

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

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

The command strings should be entered as:

```
DDE-Topic CommandString
DDE-Topic CommandString
```

as in the following example, which applies to Codewright®:

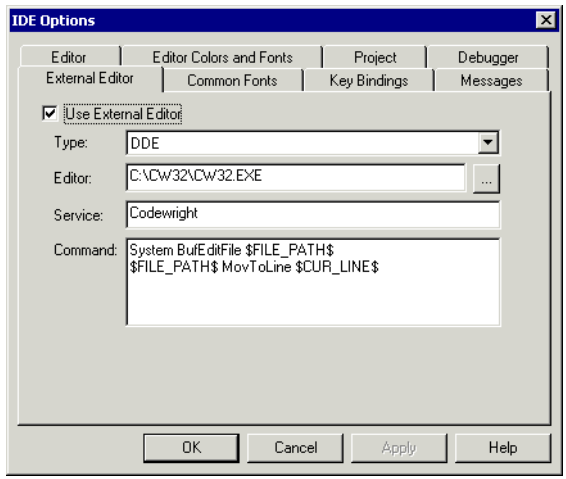


Figure 43: External editor DDE settings

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

5 Click **OK**.

When you open a file by double-clicking it in the Workspace window, the file will be opened by the external editor.

Variables can be used in the arguments. For more information about the argument variables that are available, see *Argument variables summary*, page 279.

Part 4. Debugging

This part of the MSP430 IAR Embedded Workbench® IDE User Guide contains the following chapters:

- The IAR C-SPY® Debugger
- Executing your application
- Working with variables and expressions
- Using breakpoints
- Monitoring memory and registers
- Using the C-SPY® macro system
- Analyzing your application.

The IAR C-SPY® Debugger

This chapter introduces you to the IAR C-SPY Debugger. First some of the concepts are introduced that are related to debugging in general and to the IAR C-SPY Debugger in particular. Then the debugger environment is presented, followed by a description of how to setup, start, and finally adapt C-SPY to target hardware.

Debugger concepts

This section introduces some of the concepts that are related to debugging in general and to the IAR C-SPY Debugger in particular. This section does not contain specific conceptual information related to the functionality of the IAR C-SPY Debugger. Instead, such information can be found in each chapter of this part of the guide. The IAR Systems user documentation uses the following terms when referring to these concepts.

IAR C-SPY DEBUGGER AND TARGET SYSTEMS

The IAR C-SPY Debugger can be used for debugging either a software target system or a hardware target system.

Figure 44, *IAR C-SPY Debugger and target systems*, shows an overview of C-SPY and possible target systems.

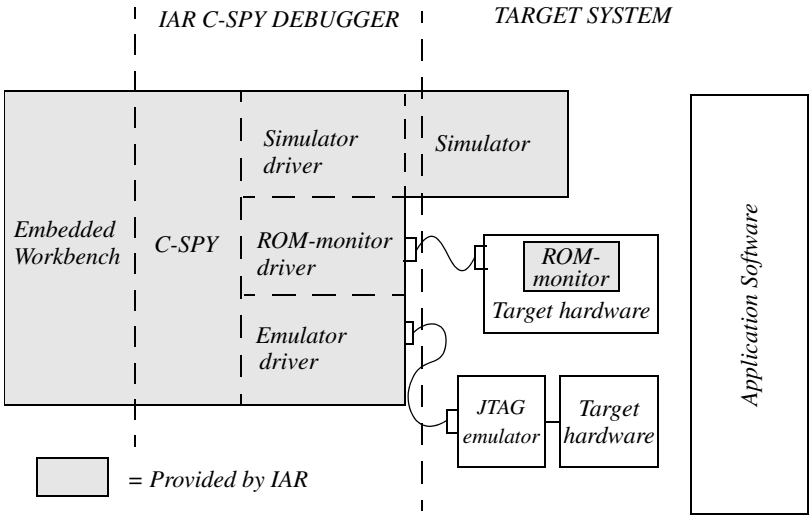


Figure 44: IAR C-SPY Debugger and target systems

DEBUGGER

The debugger, for instance C-SPY, is the program that you use for debugging your applications on a target system.

TARGET SYSTEM

The target system is the system on which you execute your application when you are debugging it. The target system can consist of hardware, either an evaluation board or your own hardware design. It can also be completely or partially simulated by software. Each type of target system needs a dedicated C-SPY driver.

USER APPLICATION

A user application is the software you have developed and which you want to debug using the IAR C-SPY Debugger.

IAR C-SPY DEBUGGER SYSTEMS

The IAR C-SPY Debugger consists of both a general part which provides a basic set of C-SPY features, and a driver. The C-SPY driver is the part that provides communication with and control of the target system. The driver also provides the user interface—menus, windows, and dialog boxes—to the functions provided by the target system, for instance, special breakpoints. There are three main types of C-SPY drivers:

- Simulator driver
- ROM-monitor driver
- Emulator driver

If you have more than one C-SPY driver installed on your computer you can switch between them by choosing the appropriate driver from within the IAR Embedded Workbench IDE.

For an overview of the general features of IAR C-SPY Debugger, see *IAR C-SPY Debugger*, page 5. In that chapter you can also find an overview of the functionality provided by each driver. Contact your software distributor or IAR representative for information about available C-SPY drivers. You can also find information on the IAR Systems website, **www.iar.com**.

ROM-MONITOR PROGRAM

The ROM-monitor program is a piece of firmware that is loaded to non-volatile memory on your target hardware; it runs in parallel with your application. The ROM-monitor communicates with the debugger and provides services needed for debugging the application, for instance stepping and breakpoints.

THIRD-PARTY DEBUGGERS

It is possible to use a third-party debugger together with the IAR Systems tool chain as long as the third-party debugger can read any of the output formats provided by XLINK, such as UBROF, ELF/DWARF, COFF, Intel-extended, Motorola, or any other available format. For information about which format to use with third-party debuggers, see the user documentation supplied with that tool.

The C-SPY environment

AN INTEGRATED ENVIRONMENT

The IAR C-SPY Debugger is a high-level-language debugger for embedded applications. It is designed for use with the MSP430 IAR C/C++ Compiler and MSP430 IAR Assembler, and is completely integrated in the IAR Embedded Workbench IDE, providing development and debugging within the same application.

All windows that are open in the Embedded Workbench workspace will stay open when you start the C-SPY Debugger. In addition, a set of C-SPY-specific windows will be opened.

You can modify your source code in an editor window during the debug session, but changes will not take effect until you exit from the debugger and rebuild your application.

The integration also makes it possible to set breakpoints in the text editor at any point during the development cycle. It is also possible to inspect and modify breakpoint definitions also when the debugger is not running. Breakpoints are highlighted in the editor windows and breakpoint definitions flow with the text as you edit. Your debug settings, such as watch properties, window layouts, and register groups will remain between your debug sessions.

In addition to the features available in the IAR Embedded Workbench IDE, the debugger environment consists of a set of C-SPY-specific items, such as a debugging toolbar, menus, windows, and dialog boxes.

Reference information about each item specific to C-SPY can be found in the chapter *C-SPY® Debugger reference*, page 313.

For specific information about a C-SPY driver, see the part of the book corresponding to the driver.

Setting up the IAR C-SPY Debugger

Before you start the IAR C-SPY Debugger you should set options to set up the debugger system. These options are available on the **Setup** page of the **Debugger** category, available with the **Project>Options** command. On the **Plugins** page you can find options for loading plug-in modules.

In addition to the options for setting up the debugger system, you can also set debugger-specific IDE options. These options are available with the **Tools>Options** command. For further information about these options, see *Debugger page*, page 297.

For information about how to configure the debugger to reflect the target hardware, see *Adapting C-SPY to target hardware*, page 113.

CHOOSING A DEBUG DRIVER

Before starting C-SPY, you must choose a driver for the debugger system from the **Driver** drop-down list on the **Setup** page. If you choose a driver for a hardware debugger system, you also need to set hardware-specific options. For information about these options, see the chapter *C-SPY® FET-specific debugging* and *Part 7. Reference information*.

Note: You can only choose a driver you have installed on your computer.

EXECUTING FROM RESET

Using the **Run to** option, you can specify a location you want C-SPY to run to when you start the debugger as well as after each reset. C-SPY will place a breakpoint at this location and all code up to this point will be executed prior to stopping at the location.

The default location to run to is the `main` function. Type the name of the location if you want C-SPY to run to a different location. You can specify assembler labels or whatever can be evaluated to such, for instance function names.

If you leave the check box empty, the program counter will then contain the regular hardware reset address at each reset.

If there are no breakpoints available when C-SPY starts, a warning message appears notifying you that single stepping will be required and that this is time consuming. You can then continue execution in single step mode or stop at the first instruction. If you choose to stop at the first instruction, the debugger starts executing with the `PC` (program counter) at the default reset location instead of the location you typed in the **Run to** box.

Note: This message will never be displayed in the C-SPY Simulator, where breakpoints are not limited.

USING A SETUP MACRO FILE

A setup macro file is a standard macro file that you choose to load automatically when C-SPY starts. You can define the setup macro file to perform actions according to your needs, by using setup macro functions and system macros. Thus, by loading a setup macro file you can initialize C-SPY to perform actions automatically.

To register a setup macro file, select **Use macro file** and type the path and name of your setup macro file, for example `Setup.mac`. If you do not type a filename extension, the extension `mac` is assumed. A browse button is available for your convenience.

For detailed information about setup macro files and functions, see *The macro file*, page 144. For an example about how to use a setup macro file, see the chapter *Simulating an interrupt* in *Part 2. Tutorials*.

SELECTING A DEVICE DESCRIPTION FILE

C-SPY handles several of the target-specific adaptations by using device description files. They contain device-specific information about for example, definitions of peripheral units and CPU registers, and groups of these.

If you want to use the device-specific information provided in the device description file during your debug session, you must select the appropriate device description file. Device description files are provided in the `430\config` directory and they have the filename extension `ddf`.

By default, a suitable device description file is always selected. To load a different device description file, you must, before you start the C-SPY debugger, choose **Project>Options** and select the **Debugger** category. On the **Setup** page, enable the use of a description file and select a file using the **Device description file** browse button.

For more information about device description files, see *Adapting C-SPY to target hardware*, page 113. For an example about how to use a setup macro file, see *Simulating an interrupt* in *Part 2. Tutorials*.

LOADING PLUGIN MODULES

On the **Plugins** page you can specify C-SPY plugin modules that are to be loaded and made available during debug sessions. Plugin modules can be provided by IAR, as well as by third-party suppliers. Contact your software distributor or IAR representative, or visit the IAR Systems web site, for information about available modules.

For information about how to load plugin modules, see *Plugins*, page 396.

The IAR C-SPY RTOS awareness plugin modules

Provided that there is one or more real-time operating systems plugin modules supported for the IAR Embedded Workbench version you are using, you can load one for use with the IAR C-SPY Debugger. C-SPY RTOS awareness plugin modules give you a high level of control and visibility over an application built on top of a real-time operating system. It displays RTOS-specific items like task lists, queues, semaphores, mailboxes and various RTOS system variables. Task-specific breakpoints and task-specific stepping make it easier to debug tasks.

A loaded plugin will add its own set of windows and buttons when a debug session is started (provided that the RTOS is linked with the application). For information about other RTOS awareness plugin modules, refer to the manufacturer of the plugin module.

Starting the IAR C-SPY Debugger

When you have setup the debugger, you can start it.

To start the IAR C-SPY Debugger and load the current project, click the **Debug** button. Alternatively, choose the **Project>Debug** command.

For information about how to execute your application and how to use the C-SPY features, see the remaining chapters in *Part 4. Debugging*.

Executable files built outside of the Embedded Workbench

It is also possible to load C-SPY with a project that was built outside the Embedded Workbench, for example projects built on the command line. To be able to set C-SPY options for the externally built project, you must create a project within the Embedded Workbench.

To load an externally built executable file, you must first create a project for it in your workspace. Choose **Project>Create New Project**, and specify a project name. To add the executable file to the project, choose **Project>Add Files** and make sure to choose **All Files** in the **Files of type** drop-down list. Locate the executable file (filename extension `d43`). To start the executable file, select the project in the Workspace window and click the **Debug** button. The project can be reused whenever you rebuild your executable file.

The only project options that are meaningful to set for this kind of project are options in the **General Options** and **Debugger** categories. Make sure to set up the general project options in the same way as when the executable file was built.

REDIRECTING DEBUGGER OUTPUT TO A FILE

The Debug Log window—available from the **View** menu—displays debugger output, such as diagnostic messages and trace information. It can sometimes be convenient to log the information to a file where it can be easily inspected. The **Log Files** dialog box—available from the **Debug** menu—allows you to log output from C-SPY to a file. The two main advantages are:

- The file can be opened in another tool, for instance an editor, so you can navigate and search within the file for particularly interesting parts
- The file provides history about how you have controlled the execution, for instance, what breakpoints have been triggered etc.

The information printed in the file is by default the same as the information listed in the Log window. However, you can choose what you want to log in the file: errors, warnings, system information, user messages, or all of these. For reference information about the Log File options, see *Log File dialog box*, page 340.

Adapting C-SPY to target hardware

This section describes how to configure the debugger to reflect the target hardware. The C-SPY device description file and its contents is described.

DEVICE DESCRIPTION FILE

C-SPY handles several of the target-specific adaptations by using device description files provided with the product. They contain device-specific information such as:

- Memory information for device-specific memory zones
- Definitions of memory-mapped peripheral units, device-specific CPU registers, and groups of these
- Definitions for interrupt simulation in the simulator.

You can find device description files for each MSP430 device in the `430\config` directory.

For information about how to load a device description file, see *Selecting a device description file*, page 111.

Memory zones

Memory information for device-specific memory zones are defined in the device description files. By default there is only one address zone in the debugger, `Memory`. If you load a device description file, additional zones that adhere better to the specific device memory layout are defined.

If your hardware does not have the same memory layout as any of the predefined device description files, you can define customized zones by adding them to the file. For further details about customizing the file, see *Modifying a device description file*, page 115.

For information about memory zones, see *Memory addressing*, page 135.

Registers

For each device there is a hardwired group of CPU registers. Their contents can be displayed and edited in the Register window. Additional registers are defined in a specific register definition file—with the filename extension `sfr`—which is included from the register section of the device description file. These registers are the device-specific memory-mapped control and status registers for the peripheral units on the MSP430 microcontrollers.

Due to the large amount of registers it is inconvenient to list all registers concurrently in the Register window. Instead the registers are divided into logical *register groups*. By default there is one register group in the MSP430 debugger, namely *CPU Registers*.

For details about how to work with the Register window, view different register groups, and how to configure your own register groups to better suit the use of registers in your application, see the section *Working with registers*, page 138.

Interrupts

Device description files also contain a section that defines all device-specific interrupts, which makes it possible to simulate these interrupts in the C-SPY Simulator. You can read more about how to do this in *Simulating interrupts*, page 177.

Modifying a device description file

There is normally no need to modify the device description file. However, if the predefinitions are not sufficient for some reason, you can edit the file. The syntax of the device descriptions is described in the files. Note, however, that the format of these descriptions might be updated in future upgrade versions of the product.

Make a copy of the device description file that best suits your needs, and modify it according to the description in the file.

Executing your application

The IAR C-SPY® Debugger provides a flexible range of facilities for executing your application during debugging. This chapter contains information about:

- The conceptual differences between source mode and disassembly mode debugging
- Executing your application
- The call stack
- Handling terminal input and output.

Source and disassembly mode debugging

The IAR C-SPY Debugger allows you to switch seamlessly between source mode and disassembly mode debugging as required.

Source debugging provides the fastest and easiest way of developing your application, without having to worry about how the compiler or assembler has implemented the code. In the editor windows you can execute the application one statement at a time while monitoring the values of variables and data structures.

Disassembly mode debugging lets you focus on the critical sections of your application, and provides you with precise control over the hardware. You can open a disassembly window which displays a mnemonic assembler listing of your application based on actual memory contents rather than source code, and lets you execute the application exactly one instruction at a time. In Mixed-Mode display, the debugger also displays the corresponding C/C++ source code interleaved with the disassembly listing.

Regardless of which mode you are debugging in, you can display registers and memory, and change their contents.

For an example of a debug session both in C source mode and disassembly mode, see *Debugging the application*, page 37.

Executing

The IAR C-SPY Debugger provides a flexible range of features for executing your application. You can find commands for executing on the Debug menu as well as on the toolbar.

STEP

C-SPY allows more stepping precision than most other debuggers in that it is not line-oriented but statement-oriented. The compiler generates detailed stepping information in the form of *step points* at each statement, as well as at each function call. That is, source code locations where you might consider whether to execute a step into or a step over command. Because the step points are located not only at each statement but also at each function call, the step functionality allows a finer granularity than just stepping on statements. There are four different step commands:

- Step Into
- Step Over
- Next Statement
- Step Out

Consider this example and assume that the previous step has taken you to the `f(i)` function call (highlighted):

```
int f(int n)
{
 value = f(n-1) + f(n-2) + f(n-3);
 return value;
}
...
f(i);
value ++;
```

While stepping, you typically consider whether to step into a function and continue stepping inside the function or subroutine. The **Step Into** command takes you to the first step point within the subroutine, `f(n-1)`:

```
int f(int n)
{
 value = f(n-1) + f(n-2) + f(n-3);
 return value;
}
...
f(i);
value ++;
```

The **Step Into** command executes to the next step point in the normal flow of control, regardless of whether it is in the same or another function.

The **Step Over** command executes to the next step point in the same function, without stopping inside called functions. The command would take you to the `f(n-2)` function call, which is not a statement on its own but part of the same statement as `f(n-1)`. Thus, you can skip uninteresting calls which are parts of statements and instead focus on critical parts:

```
int f(int n)
{
 value = f(n-1) + f(n-2) + f(n-3);
 return value;
}
...
f(i);
value ++;
```

The **Next Statement** command executes directly to the next statement `return value`, allowing faster stepping:

```
int f(int n)
{
 value = f(n-1) + f(n-2) + f(n-3);
 return value;
}
...
f(i);
value ++;
```

When inside the function, you have the choice of stepping out of it before reaching the function exit, by using the **Step Out** command. This will take you directly to the statement immediately after the function call:

```
int f(int n)
{
 value = f(n-1) + f(n-2) f(n-3);
 return value;
 ...
}
...
f(i);
value ++;
```

The possibility of stepping into an individual function that is part of a more complex statement is particularly useful when you use C code containing many nested function calls. It is also very useful for Embedded C++, which tends to have many implicit function calls, such as constructors, destructors, assignment operators, and other user-defined operators.

This detailed stepping can in some circumstances be either invaluable or unnecessarily slow. For this reason, it is also possible to step only on statements, which means faster stepping.

GO

The **Go** command continues execution from the current position until a breakpoint or program exit is reached.

RUN TO CURSOR

The **Run to Cursor** command executes to the position in the source code where you have placed the cursor. The **Run to Cursor** command also works in the Disassembly window and in the Call Stack window.

HIGHLIGHTING

At each stop, C-SPY highlights the corresponding C or C++ source with a green color.

For simple statements without function calls, the whole statement is typically highlighted. When stopping at a statement with function calls, C-SPY highlights the first call because this illustrates more clearly what **Step Into** and **Step Over** would mean at that time.

Occasionally, you will notice that a statement in the source window is highlighted using a pale variant of the normal highlight color. This happens when the program counter is at an assembler instruction which is part of a source statement but not exactly at a step point. This is often the case when stepping in the Disassembly window. Only when the program counter is at the first instruction of the source statement, the ordinary highlight color is used.

USING BREAKPOINTS TO STOP

You can set breakpoints in the application to stop at locations of particular interest. These locations can be either at code sections where you want to investigate whether your program logic is correct, or at data accesses to investigate when and how the data is changed. Depending on which debugger system you are using you might also have access to additional types of breakpoints. For instance, if you are using C-SPY Simulator there is a special kind of breakpoint to facilitate simulation of simple hardware devices. See the chapter *Simulator-specific debugging* for further details.

For a more advanced simulation, you can stop under certain conditions, which you specify. It is also possible to connect a C-SPY macro to the breakpoint. The macro can be defined to perform actions, which for instance can simulate specific hardware behavior.

All these possibilities provide you with a flexible tool for investigating the status of, for example, variables and registers at different stages during the application execution.

For detailed information about the breakpoint system and how to use the different breakpoint types, see the chapter *Using breakpoints*.

USING THE BREAK BUTTON TO STOP

While your application is executing, the **Break** button on the debug toolbar is highlighted in red. You can stop the application execution by clicking the **Break** button, alternatively by choosing the **Debug>Break** command.

STOP AT PROGRAM EXIT

Typically, the execution of an embedded application is not intended to end, which means that the application will not make use of a traditional exit. However, there are situations where a controlled exit is necessary, such as during debug sessions. You can link your application with a special library that contains an exit label. A breakpoint will be automatically set on that label to stop execution when it gets there. Before you start C-SPY, choose **Project>Options**, and select the **Linker** category. On the **Output** page, select the option **With runtime control modules (-r)**.

Call stack information

The MSP430 IAR C/C++ Compiler generates extensive backtrace information. This allows C-SPY to show, without any runtime penalty, the complete call chain at any time. Typically, this is useful for two purposes:

- Determining in what context the current function has been called
- Tracing the origin of incorrect values in variables and incorrect values in parameters, thus locating the function in the call chain where the problem occurred.

The Call Stack window—available from the **View** menu—shows a list of function calls, with the current function at the top. When you inspect a function in the call chain, by double-clicking on any function call frame, the contents of all affected windows will be updated to display the state of that particular call frame. This includes the editor, Locals, Register, Watch and Disassembly windows. A function would normally not make use of all registers, so these registers might have undefined states and be displayed as dashes (---). For reference information about the Call Stack window, see *Call Stack window*, page 326.

In the editor and Disassembly windows, a green highlight indicates the topmost, or current, call frame; a yellow highlight is used when inspecting other frames.

For your convenience, it is possible to select a function in the call stack and click the **Run to Cursor** command—available on the **Debug** menu, or alternatively on the context menu—to execute to that function.

Assembler source code does not automatically contain any backtrace information. To be able to see the call chain also for your assembler modules, you can add the appropriate `CFI` assembler directives to the source code. For further information, see the *MSP430 IAR Assembler Reference Guide*.

Terminal input and output

Sometimes you might need to debug constructions in your application that use `stdin` and `stdout` without an actual hardware device for input and output. The Terminal I/O window—available on the **View** menu—lets you enter input to your application, and display output from it.

This facility can be useful in two different contexts:

- If your application uses `stdin` and `stdout`
- For producing debug trace printouts.

To use this window, you need to link your application with the option **With I/O emulation modules**. C-SPY will then direct `stdin`, `stdout`, and `stderr` to this window.

For reference information, see *Terminal I/O window*, page 328.

Directing stdin and stdout to a file

You can also direct `stdin` and `stdout` directly to a file. You can then open the file in another tool, for instance an editor, to navigate and search within the file for particularly interesting parts. The **Terminal I/O Log Files** dialog box—available by choosing **Debug>Logging**—allows you to select a destination log file, and to log terminal I/O input and output from C-SPY to this file.

For reference information, see *Terminal I/O Log File dialog box*, page 341.

Working with variables and expressions

This chapter defines the variables and expressions used in C-SPY®. It also demonstrates the different methods for examining variables and expressions.

C-SPY expressions

C-SPY lets you examine the C variables, C expressions, and assembler symbols that you have defined in your application code. In addition, C-SPY allows you to define C-SPY macro variables and macro functions and use them when evaluating expressions. Expressions that are built with these components are called C-SPY expressions and there are several methods for monitoring these in C-SPY.

C-SPY expressions can include any type of C expression, except function calls. The following types of symbols can be used in expressions:

- C/C++ symbols
- Assembler symbols (register names and assembler labels)
- C-SPY macro functions
- C-SPY macro variables

Examples of valid C-SPY expressions are:

```
i + j
i = 42
#asm_label
#R2
#PC
my_macro_func(19)
```

C SYMBOLS

C symbols are symbols that you have defined in the C source code of your application, for instance variables, constants, and functions. C symbols can be referenced by their names.

Using sizeof

According to the ISO/ANSI C standard, there are two syntactical forms of `sizeof`:

```
sizeof(type)
sizeof expr
```

The former is for types and the latter for expressions.

In C-SPY, do not use parentheses around an expression when you use the `sizeof` operator. For example, use `sizeof x+2` instead of `sizeof (x+2)`.

ASSEMBLER SYMBOLS

Assembler symbols can be assembler labels or register names. That is, general purpose registers, such as `R4–R15`, and special purpose registers, such as the program counter and the status register. If a device description file is used, all memory-mapped peripheral units, such as I/O ports, can also be used as assembler symbols in the same way as the CPU registers. See *Device description file*, page 113.

Assembler symbols can be used in C-SPY expressions if they are prefixed by `#`.

Example	What it does
`#pc++`	Increments the value of the program counter.
`myptr = #label7`	Sets `myptr` to the integral address of `label7` within its zone.

Table 12: C-SPY assembler symbols expressions

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 enclose the label in back quotes ` (ASCII character 0x60). For example:

Example	What it does
`#pc`	Refers to the program counter.
`#`pc``	Refers to the assembler label `pc`.

Table 13: Handling name conflicts between hardware registers and assembler labels

Which processor-specific symbols are available by default can be seen in the **Register** window, using the `CPU Registers` register group. See *Register groups*, page 138.

MACRO FUNCTIONS

Macro functions consist of C-SPY variable definitions and macro statements which are executed when the macro is called.

For details of C-SPY macro functions and how to use them, see *The macro language*, page 144.

MACRO VARIABLES

Macro variables are defined and allocated outside your application, and can be used in a C-SPY expression. In case of a name conflict between a C symbol and a C-SPY macro variable, the C-SPY macro variable will have a higher precedence than the C variable. Assignments to a macro variable assigns both its value and type.

For details of C-SPY macro variables and how to use them, see *The macro language*, page 397.

Limitations on variable information

The value of a C variable is valid only on step points, that is, the first instruction of a statement and on function calls. This is indicated in the editor window with a bright green highlight color. In practice the value of the variable is accessible and correct more often than that.

When the program counter is inside a statement, but not at a step point, the statement or part of the statement is highlighted with a pale variant of the ordinary highlight color.

EFFECTS OF OPTIMIZATIONS

The compiler is free to optimize the application software as much as possible, as long as the expected behavior remains. Depending on your project settings, a high level of optimization results in smaller or faster code, but also in increased compile time. Debugging might be more difficult because it will be less clear how the generated code relates to the source code. Typically, using a high optimization level can affect the code in a way that will not allow you to view a value of a variable as expected.

Consider this example:

```
foo()
{
 int i = 42;
 ...
 x = bar(i); //Not until here the value of i is known to C-SPY
 ...
}
```

From the point where the variable `i` is declared until it is actually used there is no need for the compiler to waste stack or register space on it. The compiler can optimize the code, which means C-SPY will not be able to display the value until it is actually used. If you try to view a value of a variable that is temporarily unavailable, C-SPY will display the text:

```
Unavailable
```

If you need full information about values of variables during your debugging session, you should make sure to use the lowest optimization level during compilation, that is, **None**.

Viewing variables and expressions

There are several methods for looking at variables and calculating their values:

- Tooltip watch provides the simplest way of viewing the value of a variable or more complex expressions. Just point at the variable with the pointer. The value will be displayed next to the variable.
- The Auto window—available from the **View** menu—automatically displays a useful selection of variables and expressions in, or near, the current statement.
- The Locals window—available from the **View** menu—automatically displays the local variables, that is, auto variables and function parameters for the active function.
- The Watch window—available from the **View** menu—allows you to monitor the values of C-SPY expressions and variables.
- The Live Watch window—available from the **View** menu—repeatedly samples and displays the value of expressions while your application is executing. Variables in the expressions must be statically located, such as global variables.
- The Quick Watch window, see *Using the Quick Watch window*, page 126.
- The Trace system, see *Using the trace system*, page 127.

For reference information about the different windows, see *C-SPY windows*, page 313.

WORKING WITH THE WINDOWS

All the windows are easy to use. You can add, modify, and remove expressions, and change the display format.

A context menu containing useful commands is available in all windows if you right-click in each window. Convenient drag-and-drop between windows is supported, except for in the Locals window and the Quick Watch window where it is not applicable.

To add a value you can also click in the dotted rectangle and type the expression you want to examine. To modify the value of an expression, click in the **Value** field and modify its content. To remove an expression, select it and press the Delete key.

Using the Quick Watch window

The Quick Watch window—available from the **View** menu—lets you watch the value of a variable or expression and evaluate expressions.

The Quick Watch window is different from the Watch window in the following ways:

- The Quick Watch window offers a fast method for inspecting and evaluating expressions. Right-click on the expression you want to examine and choose **Quick Watch** from the context menu that appears. The expression will automatically appear in the Quick Watch window.

- In contrast to the Watch window, the Quick Watch window gives you precise control over when to evaluate the expression. For single variables this might not be necessary, but for expressions with side effects, such as assignments and C-SPY macro functions, it allows you to perform evaluations under controlled conditions.

USING THE TRACE SYSTEM

A *trace* is a recorded sequence of events in the target system, typically executed machine instructions. Depending on what C-SPY driver you are using, additional types of trace data can be recorded. For example, read and write accesses to memory, as well as the values of C-SPY expressions.

By using the trace system, you can trace the program flow up to a specific state, for instance an application crash, and use the trace information to locate the origin of the problem. Trace information can be useful for locating programming errors that have irregular symptoms and occur sporadically. Trace information can also be useful as test documentation. You can save the trace information to a file to be analyzed later.

The trace system is only supported by the simulator driver and not by the FET debugger driver. For detailed information about the trace system and the components provided by the simulator, see *Simulator-specific debugging*, page 159.

The Trace window and its browse mode

The type of information that is displayed in the Trace window depends on the C-SPY driver you are using. The different trace data is displayed in separate columns, but the **Trace** column is always available regardless of what driver you are using. The corresponding source code can also be shown.

You can follow the execution history by simply looking and scrolling in the Trace window. Alternatively, you can enter *browse mode*. To enter browse mode, double-click an item in the Trace window, or click the **Browse** toolbar button. The selected item turns yellow and the source and disassembly windows will highlight the corresponding location. You can now move around in the Trace window by using the up and down arrow keys, or by scrolling and clicking; the source and Disassembly windows will be updated to show the corresponding location. Double-click again to leave browse mode.

Searching in the trace data

You can perform advanced searches in the recorded trace data. You specify the search criteria in the **Find in Trace** dialog box and view the result in the Find in Trace window.

The Find in Trace window is very similar to the Trace window, showing the same columns and data, but *only* those rows that match the specified search criteria. Double-clicking an item in the Find in Trace window brings up the same item in the Trace window.

VIEWING ASSEMBLER VARIABLES

An assembler label does not convey any type information at all, which means C-SPY cannot easily display data located at that label without getting extra information. To view data conveniently, C-SPY treats, by default, all data located at assembler labels as variables of type `int`. However, in the Watch, Quick Watch, and Live Watch windows, you can select a different interpretation to better suit the declaration of the variables.

In this figure, you can see four variables in the Watch window and their corresponding declarations in the assembler source file to the left:

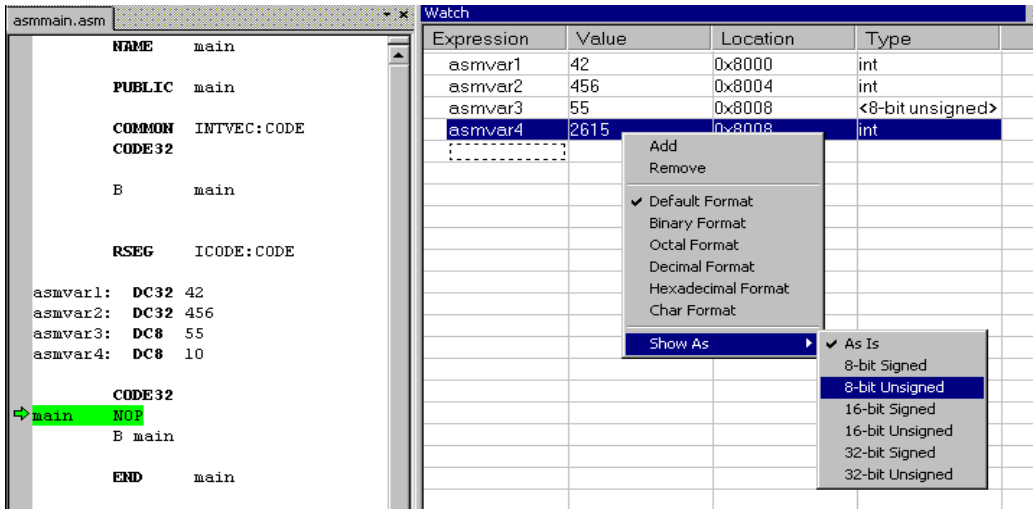


Figure 45: Viewing assembler variables in the Watch window

Note that `asmvar4` is displayed as an `int`, although the original assembler declaration probably intended for it to be a single byte quantity. From the context menu you can make C-SPY display the variable as, for example, an 8-bit unsigned variable. This has already been specified for the `asmvar3` variable.

Using breakpoints

This chapter describes the breakpoint system and different ways to create and monitor breakpoints.

The breakpoint system

The C-SPY® breakpoint system lets you set various kinds of breakpoints in the application you are debugging, allowing you to stop at locations of particular interest. You can set a breakpoint at a *code* location to investigate whether your program logic is correct, or to get trace printouts. In addition to code breakpoints, and depending on what C-SPY driver you are using, additional breakpoint types might be available. For example, you might be able to set a *data* breakpoint, to investigate how and when the data changes. If you are using the simulator driver you can also set *immediate* breakpoints.

All your breakpoints are listed in the *Breakpoints window* where you can conveniently monitor, enable, and disable them.

For a more advanced simulation, you can stop under certain *conditions*, which you specify. It is also possible to let the breakpoint trigger a *side effect*, for instance executing a C-SPY macro function, without stopping the execution. The macro function can be defined to perform a wide variety of actions, for instance, simulating hardware behavior.

You can set breakpoints while you edit your code even if no debug session is active. The breakpoints will then be validated when the debug session starts. Breakpoints are preserved between debug sessions. C-SPY provides different ways of defining breakpoints.

All these possibilities provide you with a flexible tool for investigating the status of your application.

Defining breakpoints

The breakpoints you define will appear in the Breakpoints window. From this window you can conveniently view all breakpoints, enable and disable breakpoints, and open a dialog box for defining new breakpoints. For more details, see *Breakpoints window*, page 255.

Breakpoints are set with a higher precision than single lines, in analogy with the step mechanism; for more details about the step precision, see *Step*, page 118.

You can set a breakpoint in several different ways: using the **Toggle Breakpoint** command, from the Memory window, from a dialog box, or using predefined system macros. The different methods allow different levels of complexity and automation.

TOGGLING A SIMPLE CODE BREAKPOINT

Toggling a code breakpoint is a quick method of setting a breakpoint. The following methods are available, either in the editor window, the Disassembly window, or both:

- Double-click in the gray left-side margin of the editor window
- Place the insertion point in the C source statement or assembler instruction where you want the breakpoint, and click the **Toggle Breakpoint** button in the toolbar
- Choose **Edit>Toggle Breakpoint**
- Right-click and choose **Toggle Breakpoint** from the context menu.

The breakpoint is marked with a red X in the left margin of the editor window:

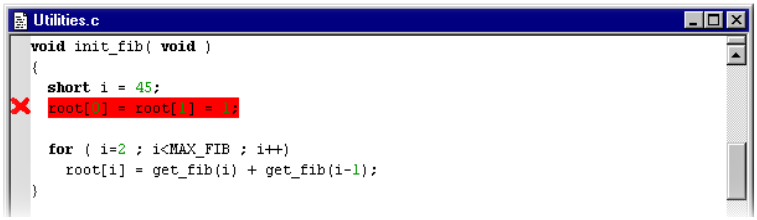


Figure 46: Breakpoint on a function call

If the red X does not appear, make sure the option **Show bookmarks** is selected, see *Editor page*, page 291.

SETTING A BREAKPOINT IN THE MEMORY WINDOW

For information about how to set breakpoints using the Memory window, see *Setting a breakpoint in the Memory window*, page 137.

DEFINING BREAKPOINTS USING THE DIALOG BOX

The advantage of using the dialog box is that it provides you with a graphical interface where you can interactively fine tune the characteristics of the breakpoints. You can set the options and quickly test whether the breakpoint works according to your intentions.

To define a new breakpoint:

1. Choose **View>Breakpoints** to open the Breakpoints window.
2. In the Breakpoints window, right-click to open the context menu.
3. On the context menu, choose **New Breakpoint**.

4 On the submenu, choose the breakpoint type you want to set. Depending on the C-SPY driver you are using, different breakpoint types might be available.

To modify an existing breakpoint:

1 Choose **View>Breakpoints** to open the Breakpoints window.

2 In the Breakpoints window, select the breakpoint you want to modify and right-click to open the context menu.

3 On the context menu, choose **Edit**.

A breakpoint dialog box appears. Specify the breakpoint settings and click **OK**. The breakpoint will be displayed in the Breakpoints window.

All breakpoints you define using a breakpoint dialog box are preserved between debug sessions.

For reference information about code and log breakpoints, see *Code breakpoints dialog box*, page 256 and *Log breakpoints dialog box*, page 258, respectively. For details about any additional breakpoint types, see the driver-specific documentation.

Tracing incorrect function arguments

If a function with a pointer argument is sometimes incorrectly called with a `NULL` argument, it is useful to put a breakpoint on the first line of the function with a condition that is true only when the parameter is 0. The breakpoint will then not be triggered until the problematic situation actually occurs.

Performing a task with or without stopping execution

You can perform a task when a breakpoint is triggered *with* or *without* stopping the execution.

You can use the **Action** text box to associate an action with the breakpoint, for instance a C-SPY macro function. When the breakpoint is triggered and the execution of your application has stopped, the macro function will be executed.

If you instead want to perform a task without stopping the execution, you can set a condition which returns `0` (false). When the breakpoint is triggered, the condition will be evaluated and since it is not true execution will continue.

Consider the following example where the C-SPY macro function performs a simple task:

```
__var my_counter;

count()
{
  my_counter += 1;
```

```
  return 0;
}
```

To use this function as a condition for the breakpoint, type `count()` in the **Expression** text box under **Conditions**. The task will then be performed when the breakpoint is triggered. Because the macro function `count` returns `0`, the condition is false and the execution of the program will resume automatically, without any stop.

DEFINING BREAKPOINTS USING SYSTEM MACROS

You can define breakpoints not only by using the **Breakpoints** dialog box but also by using built-in C-SPY system macros. When you use macros for defining breakpoints, the breakpoint characteristics are specified as function parameters.

Macros are useful when you have already specified your breakpoints so that they fully meet your requirements. You can define your breakpoints in a macro file by using built-in system macros and execute the file at C-SPY startup. The breakpoints will then be set automatically each time you start C-SPY. Another advantage is that the debug session will be documented, and that several engineers involved in the development project can share the macro files.

If you use system macros for setting breakpoints it is still possible to view and modify them in the Breakpoints window. In contrast to using the dialog box for defining breakpoints, all breakpoints that are defined using system macros will be removed when you exit the debug session.

The following breakpoint macros are available:

```
__setCodeBreak
__setDataBreak
__setSimBreak
__clearBreak
```

For details of each breakpoint macro, see the chapter *C-SPY® macros reference*.

Defining breakpoints at C-SPY startup using a setup macro file

You can use a setup macro file to define breakpoints at C-SPY startup. Follow the procedure described in *Registering and executing using setup macros and setup files*, page 147.

Viewing all breakpoints

To view breakpoints, you can use the Breakpoints window and the **Breakpoints Usage** dialog box.

For information about the Breakpoints window, see *Breakpoints window*, page 255.

USING THE BREAKPOINT USAGE DIALOG BOX

The **Breakpoint Usage** dialog box—available from C-SPY driver-specific menus, for example the **Simulator** menu—lists all active breakpoints.

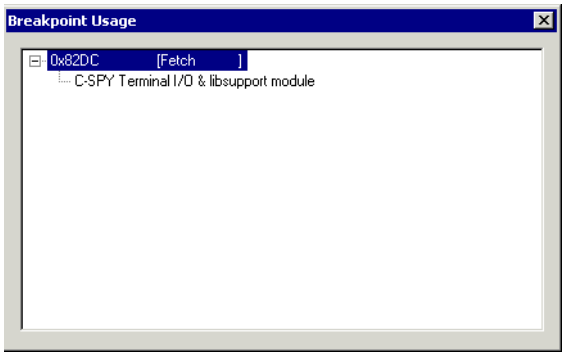


Figure 47: Breakpoint Usage dialog box

The **Breakpoint Usage** dialog box lists all breakpoints currently set in the target system, both the ones you have defined and the ones used internally by C-SPY. For each breakpoint in the list, the address and access type are shown. Each breakpoint can also be expanded to show its originator. The format of the items in this dialog box depends on which C-SPY driver you are using.

The dialog box gives a low-level view of all breakpoints, related but not identical to the list of breakpoints shown in the **Breakpoints** dialog box.

Exceeding the number of available low-level breakpoints will cause the debugger to single step. This will significantly reduce the execution speed. Therefore, in a debugger system with a limited amount of breakpoints, the **Breakpoint Usage** dialog box can be useful for:

- Identifying all consumers of breakpoints
- Checking that the number of active breakpoints is supported by the target system
- Configuring the debugger to utilize the available breakpoints in a better way, if possible.

For information about the available number of breakpoints in the debugger system you are using and how to use the available breakpoints in a better way, see the section about breakpoints in the part of this book that corresponds to the debugger system you are using.

Breakpoint consumers

There are several consumers of breakpoints in a debugger system.

User breakpoints—the breakpoints you define by using the **Breakpoints** dialog box or by toggling breakpoints in the editor window—often consume one low-level breakpoint each, but this can vary greatly. Some user breakpoints consume several low-level breakpoints and conversely, several user breakpoints can share one low-level breakpoint. User breakpoints are displayed in the same way both in the **Breakpoint Usage** dialog box and in the **Breakpoints** dialog box, for example `Data @[R] callCount`.

C-SPY itself also consumes breakpoints. C-SPY will set a breakpoint if:

- the C-SPY option **Run to** has been selected, and any step command is used. These are temporary breakpoints which are only set when the debugger system is running. This means that they are not visible in the Breakpoint Usage window.
- the linker options **With I/O emulation modules** has been selected.

These types of breakpoint consumers are displayed in the **Breakpoint Usage** dialog box, for example, `C-SPY Terminal I/O & libsupport module`.

In addition, C-SPY plugin modules, for example modules for real-time operating systems, can consume additional breakpoints.

Monitoring memory and registers

This chapter describes how to use the features available in the IAR C-SPY® Debugger for examining memory and registers:

- The Memory window
- The Register window
- Predefined and user-defined register groups
- The Stack window.

Memory addressing

In C-SPY, the term *zone* is used for a named memory area. A memory address, or *location*, is a combination of a zone and a numerical offset into that zone. The MSP430 architecture has only one zone, `Memory`, which covers the whole MSP430 memory range. If you load a device description file, additional zones that adhere better to the specific device memory layout are defined.

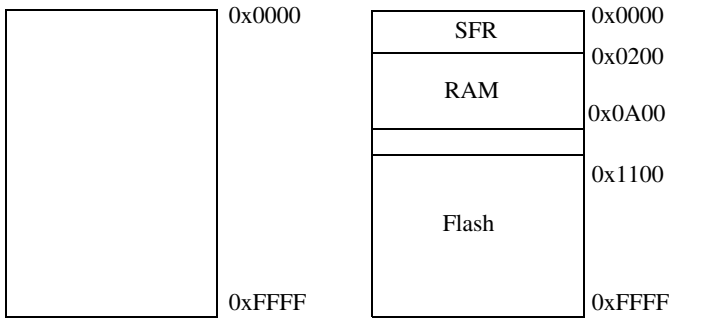


Figure 48: Zones in C-SPY

Memory zones are used in several contexts, perhaps most importantly in the Memory and Disassembly windows. The **Zone** box in these windows allows you to choose which memory zone to display.

Memory zones are defined in the device description files. For further information, see *Device description file*, page 113.

Using the Memory window

The Memory window—available from the **View** menu—gives an up-to-date display of a specified area of memory and allows you to edit it. You can open several instances of this window, which is very convenient if you want to monitor different memory or register areas.

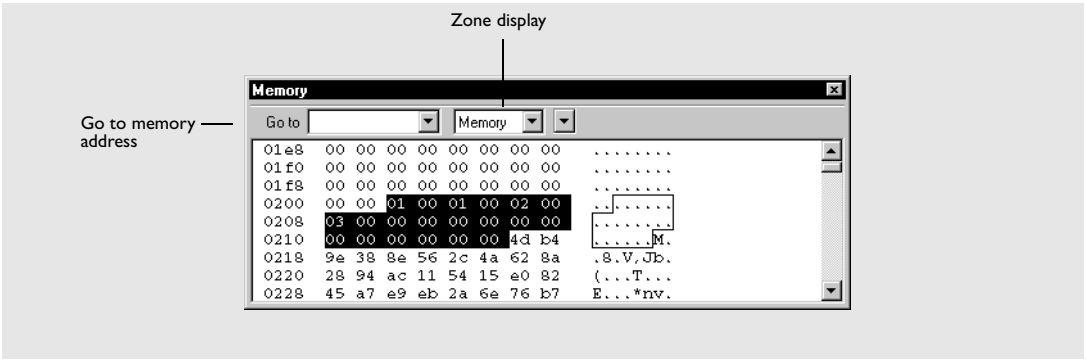


Figure 49: Memory window

The window consists of three columns. The left-most part displays the addresses currently being viewed. The middle part of the window displays the memory contents in the format you have chosen. Finally, the right-most part displays the memory contents in ASCII format. You can edit the contents of the Memory window, both in the hexadecimal part and the ASCII part of the window.

You can easily view the memory contents for a specific variable by dragging the variable to the Memory window. The memory area where the variable is located will appear.

Memory window operations

At the top of the window there are commands for navigation and configuration. These commands are also available on the context menu that appears when you right-click in the Memory window. In addition, commands for editing, opening the **Fill** dialog box, and setting breakpoints are available.

For reference information about each command, see *Memory window*, page 318.

Memory Fill

The **Fill** dialog box allows you to fill a specified area of memory with a value.

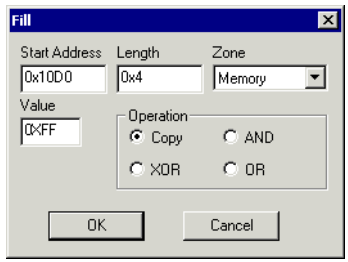


Figure 50: Memory Fill dialog box

For reference information about the dialog box, see *Fill dialog box*, page 320.

Setting a breakpoint in the Memory window

It is possible to set breakpoints directly on a memory location in the Memory window. Right-click in the window and choose the breakpoint command from the context menu that appears. To set the breakpoint on a range, select a portion of the memory contents.

The breakpoint is not highlighted; you can see, edit, and remove it by using the Breakpoints window, which is available from the **View** menu. The breakpoints you set in this window will be triggered for both read and write access. All breakpoints defined in the Memory window are preserved between debug sessions.

Note: Setting different types of breakpoints in the Memory window is only supported if the driver you use supports these types of breakpoints.

Working with registers

The Register window—available from the **View** menu—gives an up-to-date display of the contents of the processor registers, and allows you to edit them.

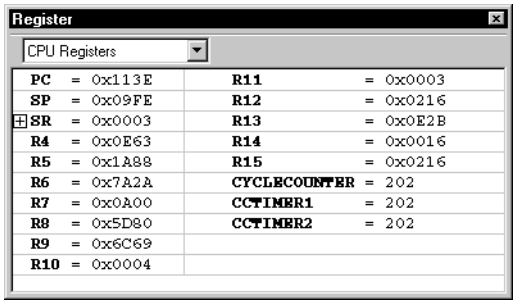


Figure 51: Register window

Every time C-SPY stops, a value that has changed since the last stop is highlighted. To edit the contents of a register, click it, and modify the value. Some registers are expandable, which means that the register contains interesting bits or subgroups of bits.

You can change the display format by changing the **Base** setting on the **Register Filter** page—available by choosing **Tools>Options**.

REGISTER GROUPS

Due to the large amount of registers—memory-mapped peripheral unit registers and CPU registers—it is inconvenient to list all registers concurrently in the Register window. Instead you can divide registers into *register groups*. By default there is only one register group in the debugger: **CPU Registers**.

In addition to the **CPU Registers** there are additional register groups predefined in the device description files—available in the `430\config` directory—that make all SFR registers available in the register window. The device description file contains a section that defines the special function registers and their groups.

You can select which register group to display in the Register window using the drop-down list. You can conveniently keep track of different register groups simultaneously, as you can open several instances of the Register window.

Enabling predefined register groups

To use any of the predefined register groups, select a device description file that suits your device, see *Selecting a device description file*, page 111.

The available register groups will be listed on the **Register Filter** page available if you choose the **Tools>Options** command when C-SPY is running.

Defining application-specific groups

In addition to the predefined register groups, you can design your own register groups that better suit the use of registers in your application.

To define new register groups, choose **Tools>Options** and click the **Register Filter** tab. This page is only available when the IAR C-SPY Debugger is running.

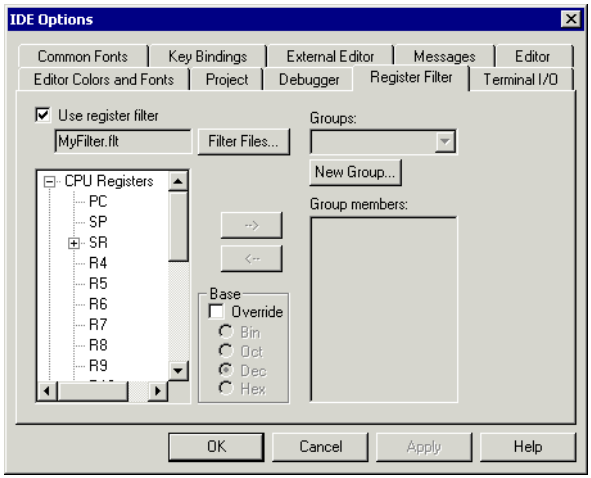


Figure 52: Register Filter page

For reference information about this dialog box, see *Register Filter page*, page 298.

Using the Stack window

The Stack window is a memory window that displays the contents of the stack. In addition, some integrity checks of the stack can be performed to detect and warn about problems with stack overflow. For example, the Stack window is useful for determining the optimal size of the stack.

Before you can open the Stack window you must make sure it is enabled; Choose **Project>Options>Debugger>Plugins** and select **Stack** from the list of plugins. In C-SPY, you can then open a Stack window by choosing **View>Stack**. You can open several Stack windows, each showing a different stack—if several stacks are available—or the same stack with different display settings.

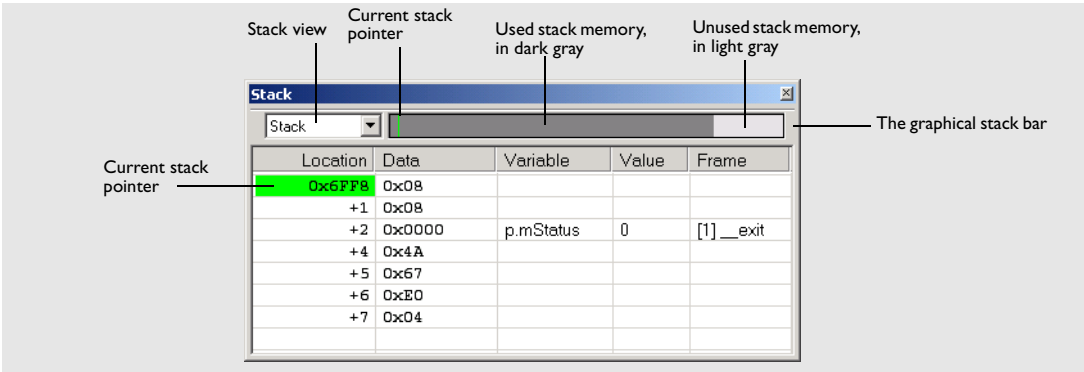


Figure 53: Stack window

For detailed reference information about the Stack window, and the method used for computing the stack usage and its limitations, see *Stack window*, page 332. For reference information about the options specific to the window, see *Stack page*, page 301.

GRAPHICAL STACK DISPLAY

At the top of the window, a stack bar displays the state of the stack graphically. To view the stack bar you must make sure it is enabled: choose **Tools>Options>Stack** and select the option **Enable stack checks**.

The left end of the stack bar represents the bottom of the stack, in other words, the position of the stack pointer when the stack is empty. The right end represents the end of the memory space reserved for the stack. A green line represents the current value of the stack pointer. The part of the stack memory that has been used during execution is displayed in a dark-gray color, and the unused part in a light-gray color. The graphical stack bar turns red when the stack usage exceeds a threshold that you can specify.

Place the mouse pointer over the stack bar to get tool tip information about stack usage.

DETECTING STACK OVERFLOWS

If you have selected the option **Enable stack checks**, available by choosing **Tools>Options>Stack**, you have also enabled the functionality needed to detect stack overflows. This means that C-SPY can issue warnings for stack overflow when the application stops executing. Warnings are issued either when the stack usage exceeds a threshold that you can specify, or when the stack pointer is outside the stack memory range.

VIEWING THE STACK CONTENTS

The main part of the Stack window displays the contents of the stack, which can be useful in many contexts. Some examples are:

- Investigating the stack usage when assembler modules are called from C modules and vice versa
- Investigating whether the correct elements are located on the stack
- Investigating whether the stack is restored properly.

Using the C-SPY® macro system

The IAR C-SPY Debugger includes a comprehensive macro system which allows 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.

This chapter describes the macro system, its features, for what purpose these features can be used, and how to use them.

The macro system

C-SPY macros can be used solely or in conjunction with complex breakpoints and interrupt simulation to perform a wide variety of tasks. Some examples where macros can be useful:

- Automating the debug session, for instance with trace printouts, printing values of variables, and setting breakpoints.
- Hardware configuring, such as initializing hardware registers.
- Developing small debug utility functions, for instance calculating the stack depth.
- Simulating peripheral devices, see the chapter *Simulating interrupts*. This only applies if you are using the simulator driver.

The macro system has several features:

- The similarity between the *macro language* and the C language, which lets you write your own macro functions.
- Predefined *system macros* which perform useful tasks such as opening and closing files, setting breakpoints and defining simulated interrupts.
- Reserved *setup macro functions* which can be used for defining at which stage the macro function should be executed. You define the function yourself, in a *setup macro file*.
- The option of collecting your macro functions in one or several *macro files.*
- A *dialog box* where you can view, register, and edit your macro functions and files. Alternatively, you can register and execute your macro files and functions using either the setup functionality or system macros.

Many C-SPY tasks can be performed either by using a dialog box or by using macro functions. The advantage of using a dialog box is that it provides you with a graphical interface where you can interactively fine-tune the characteristics of the task you want to perform, for instance setting a breakpoint. You can add parameters and quickly test whether the breakpoint works according to your intentions.

Macros, on the other hand, are useful when you already have specified your breakpoints so that they fully meet your requirements. You can set up your simulator environment automatically by writing a macro file and executing it, for instance when you start C-SPY. Another advantage is that the debug session will be documented, and if there are several engineers involved in the development project you can share the macro files within the group.

THE MACRO LANGUAGE

The syntax of the macro language is very similar to the C language. There are *macro statements*, which are similar to C statements. You can define *macro functions*, with or without parameters and return values. You can use built-in system macros, similar to C library functions. Finally, you can define global and local *macro variables*. For a detailed description of the macro language components, see *The macro language*, page 397.

Example

Consider this example of a macro function which illustrates the different components of the macro language:

```
CheckLatest(value)
{
 oldvalue;
 if (oldvalue != value)
 {
  __message "Message: Changed from ", oldvalue, " to ", value;
  oldvalue = value;
 }
}
```

Note: Reserved macro words begin with double underscores to prevent name conflicts.

THE MACRO FILE

You collect your macro variables and functions in one or several macro files. To define a macro variable or macro function, first create a text file containing the definition. You can use any suitable text editor, such as the editor supplied with IAR Embedded Workbench. Save the file with a suitable name using the filename extension `mac`.

Setup macro file

It is possible to load a macro file at C-SPY startup; such a file is called a *setup macro file*. This is especially convenient if you want to make C-SPY perform actions before you load your application software, for instance to initialize some CPU registers or memory-mapped peripheral units. Other reasons might be if you want to automate the initialization of C-SPY, or if you want to register multiple setup macro files. An example of a C-SPY setup macro file `SetupSimple.mac` can be found in the `430\tutor` directory.

For information about how to load a setup macro file, see *Registering and executing using setup macros and setup files*, page 147. For an example of how to use setup macro files, see the chapter *Simulating an interrupt* in *Part 2. Tutorials*.

SETUP MACRO FUNCTIONS

The *setup macro functions* are reserved macro function names that will be called by C-SPY at specific stages during execution. The stages to choose between are:

- After communication with the target system has been established but before downloading the application software
- Once after your application software has been downloaded
- Each time the reset command is issued
- Once when the debug session ends.

To define a macro function to be called at a specific stage, you should define and register a macro function with the name of a setup macro function. For instance, if you want to clear a specific memory area before you load your application software, the macro setup function `execUserPreload` is suitable. This function is also suitable if you want to initialize some CPU registers or memory mapped peripheral units before you load your application software. For detailed information about each setup macro function, see *Setup macro functions summary*, page 402.

As with any macro function, you collect your setup macro functions in a macro file. Because many of the setup macro functions execute before `main` is reached, you should define these functions in a *setup macro file*.

Using C-SPY macros

If you decide to use C-SPY macros, you first need to create a macro file in which you define your macro functions. C-SPY needs to know that you intend to use your defined macro functions, and thus you must *register* (load) your macro file. During the debug session you might need to list all available macro functions as well as execute them.

To list the registered macro functions, you can use the **Macro Configuration** dialog box. There are various ways to both register and execute macro functions:

- You can register a macro interactively by using the **Macro Configuration** dialog box.
- You can register and execute macro functions at the C-SPY startup sequence by defining setup macro functions in a setup macro file.
- A file containing macro function definitions can be registered using the system macro `__registerMacroFile`. This means that you can dynamically select which macro files to register, depending on the runtime conditions. Using the system macro also lets you register multiple files at the same moment. For details about the system macro, see *__registerMacroFile*, page 412.
- The Quick Watch window lets you evaluate expressions, and can thus be used for executing macro functions.
- A macro can be connected to a breakpoint; when the breakpoint is triggered the macro will be executed.

USING THE MACRO CONFIGURATION DIALOG BOX

The **Macro Configuration** dialog box—available by choosing **Debug>Macros**—lets you list, register, and edit your macro files and functions. The dialog box offers you an interactive interface for registering your macro functions which is convenient when you develop macro functions and continuously want to load and test them.

Macro functions that have been registered using the dialog box will be deactivated when you exit the debug session, and will not automatically be registered at the next debug session.

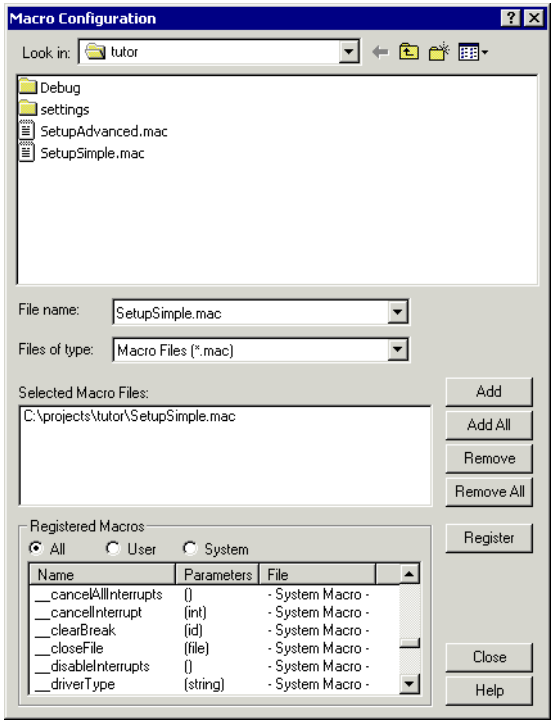


Figure 54: Macro Configuration dialog box

For reference information about this dialog box, see *Macro Configuration dialog box*, page 338.

REGISTERING AND EXECUTING USING SETUP MACROS AND SETUP FILES

It can be convenient to register a macro file during the C-SPY startup sequence, especially if you have several ready-made macro functions. C-SPY can then execute the macros before `main` is reached. You achieve this by specifying a macro file which you load before starting the debugger. Your macro functions will be automatically registered each time you start the C-SPY Debugger.

If you define the macro functions by using the setup macro function names you can define exactly at which stage you want the macro function to be executed.

Follow these steps:

1. Create a new text file where you can define your macro function.

   For example:

```
execUserSetup()
{
 ...
 __registerMacroFile(MyMacroUtils.mac);
 __registerMacroFile(MyDeviceSimulation.mac);

}
```

   This macro function registers the macro files `MyMacroUtils.mac` and `MyDeviceSimulation.mac`. Because the macro function is defined with the `execUserSetup` function name, it will be executed directly after your application has been downloaded.

2. Save the file using the filename extension `mac`.

3. Before you start C-SPY, choose **Project>Options** and click the **Setup** tab in the **Debugger** category. Select the check box **Use Setup file** and choose the macro file you just created.

   The interrupt macro will now be loaded during the C-SPY startup sequence.

EXECUTING MACROS USING QUICK WATCH

The Quick Watch window—available from the **View** menu—lets you watch the value of any variables or expressions and evaluate them. For macros, the Quick Watch window is especially useful because it is a method which lets you dynamically choose when to execute a macro function.

Consider the following simple macro function which checks the status of a watchdog timer interrupt enable bit:

```
WDTstatus()
{
  if (#IE1 & 0x01 != 0)   /* Checks the status of WDTIE */
    return "Timer enabled"; /* C-SPY macro string used */
  else
    return "Timer disabled"; /* C-SPY macro string used */
}
```

1. Save the macro function using the filename extension `mac`. Keep the file open.

2. To register the macro file, choose **Debug>Macros**. The **Macro Configuration** dialog box appears. Locate the file, click **Add** and then **Register**. The macro function appears in the list of registered macros.

3 In the macro file editor window, select the macro function name `WDTstatus`. Right-click, and choose **Quick Watch** from the context menu that appears.

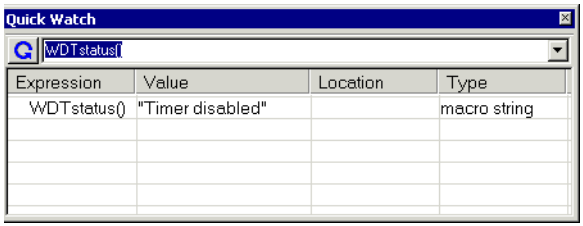


Figure 55: Quick Watch window

The macro will automatically be displayed in the Quick Watch window.

Click **Close** to close the window.

EXECUTING A MACRO BY CONNECTING IT TO A BREAKPOINT

You can connect a macro to a breakpoint. The macro will then be executed at the time when the breakpoint is triggered. The advantage is that you can stop the execution at locations of particular interest and perform specific actions there.

For instance, you can easily produce log reports containing information such as how the values of variables, symbols, or registers changes. To do this you might set a breakpoint on a suspicious location and connect a log macro to the breakpoint. After the execution you can study how the values of the registers have changed.

For an example of how to create a log macro and connect it to a breakpoint, follow these steps:

1 Assume this skeleton of a C function in your application source code:

```
int fact(int x)
{
  ...
}
```

2 Create a simple log macro function like this example:

```
logfact()
{
 __message "fact(" ,x, ")";
}
```

The `__message` statement will log messages to the Log window.

Save the macro function in a macro file, with the filename extension `mac`.

3 Before you can execute the macro it must be registered. Open the **Macro Configuration** dialog box—available by choosing **Debug>Macros**—and add your macro file to the list **Selected Macro Files**. Click **Register** and your macro function will appear in the list **Registered Macros**. Close the dialog box.

4 Next, you should toggle a code breakpoint—using the **Toggle Breakpoint** button—on the first statement within the function `fact` in your application source code. Open the **Breakpoint** dialog box—available by choosing **Edit>Breakpoints**—your breakpoint will appear in the list of breakpoints at the bottom of the dialog box. Select the breakpoint.

5 Connect the log macro function to the breakpoint by typing the name of the macro function, `logfact()`, in the **Action** field and clicking **Apply**. Close the dialog box.

6 Now you can execute your application source code. When the breakpoint has been triggered, the macro function will be executed. You can see the result in the Log window.

You can easily enhance the log macro function by, for instance, using the `__fmessage` statement instead, which will print the log information to a file. For information about the `__fmessage` statement, see *Formatted output*, page 400.

For a complete example where a serial port input buffer is simulated using the method of connecting a macro to a breakpoint, see the chapter *Simulating an interrupt* in *Part 2. Tutorials*.

Analyzing your application

It is important to locate an application's bottle-necks and to verify that all parts of an application have been tested. This chapter presents facilities available in the IAR C-SPY® Debugger for analyzing your application so that you can efficiently spend time and effort on optimizations.

Code coverage is only supported by the IAR C-SPY Simulator.

Function-level profiling

The profiler will help you find the functions where most time is spent during execution, for a given stimulus. Those functions are the parts you should focus on when spending time and effort on optimizing your code. A simple method of optimizing a function is to compile it using speed optimization. Alternatively, you can move the function into memory which uses the most efficient addressing mode. For detailed information about efficient memory usage, see the *MSP430 IAR C/C++ Compiler Reference Guide*.

The Profiling window displays profiling information, that is, timing information for the functions in an application. Profiling must be turned on explicitly using a button on the window's toolbar, and will stay active until it is turned off.

The profiler measures the time between the entry and return of a function. This means that time consumed in a function is not added until the function returns or another function is called. You will only notice this if you are stepping into a function.

For reference information about the Profiling window, see *Profiling window*, page 330.

USING THE PROFILER

Before you can use the Profiling window, you must build your application using the following options:

Category	Setting
C/C++ Compiler	Output>Generate debug information
Linker	Format>Debug information for C-SPY
Debugger	Plugins>Profiling

Table 14: Project options for enabling profiling

1 After you have built your application and started C-SPY, choose **View>Profiling** to open the window, and click the **Activate** button to turn on the profiler.

2 Click the **Clear** button, alternatively use the context menu available by right-clicking in the window, when you want to start a new sampling.

3 Start the execution. When the execution stops, for instance because the program exit is reached or a breakpoint is triggered, click the **Refresh** button.

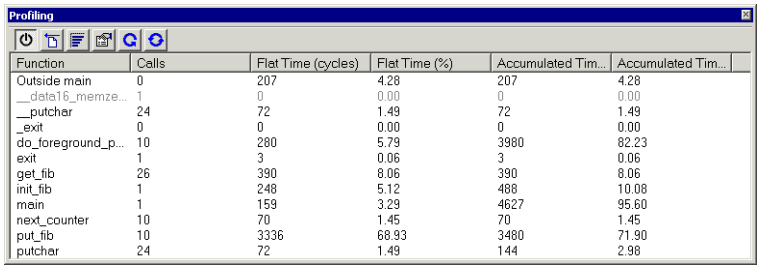


Figure 56: Profiling window

Profiling information is displayed in the window.

Viewing the figures

Clicking on a column header sorts the complete list according to that column.

A dimmed item in the list indicates that the function has been called by a function which does not contain source code (compiled without debug information). When a function is called by functions that do not have their source code available, such as library functions, no measurement in time is made.

There is always an item in the list called Outside main. This is time that cannot be placed in any of the functions in the list. That is, code compiled without debug information, for instance, all startup and exit code, and C/C++ library code.

Clicking the **Graph** button toggles the percentage columns to be displayed either as numbers or as bar charts.

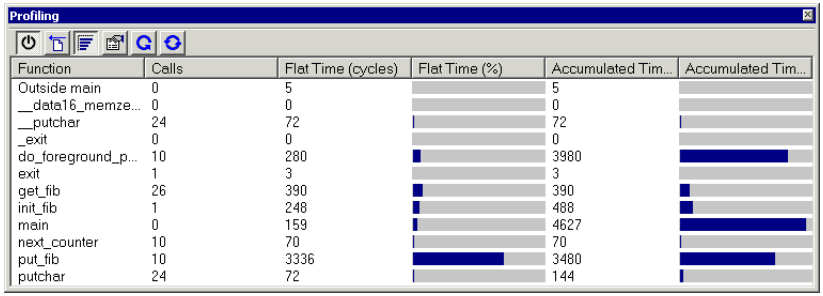


Figure 57: Graphs in Profiling window

Clicking the **Show details** button displays more detailed information about the function selected in the list. A window is opened showing information about callers and callees for the selected function:

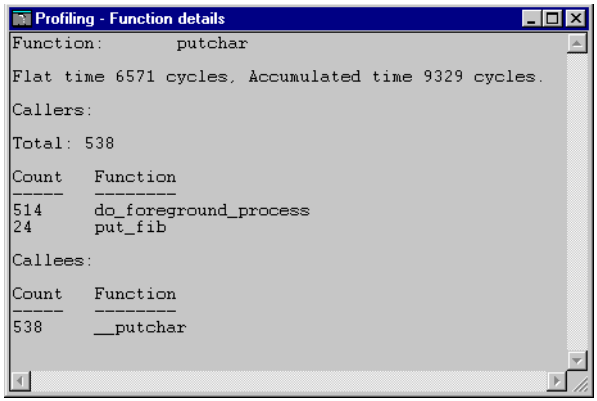


Figure 58: Function details window

Producing reports

To produce a report, right-click in the window and choose the **Save As** command on the context menu. The contents of the Profiling window will be saved to a file.

Code coverage

The code coverage functionality helps you verify whether all parts of your code have been executed. This is useful when you design your test procedure to make sure that all parts of the code have been executed. It also helps you identify parts of your code that are not reachable.

USING CODE COVERAGE

The Code Coverage window—available from the **View** menu—reports the status of the current code coverage analysis, that is, what parts of the code have been executed at least once since the start of the analysis. The compiler generates detailed stepping information in the form of *step points* at each statement, as well as at each function call. The report includes information about all modules and functions. It reports the amount of all step points, in percentage, that have been executed and lists all step points that have not been executed up to the point where the application has been stopped. The coverage will continue until turned off.

For reference information about the Code Coverage window, see *Code Coverage window*, page 329.

Before using the Code Coverage window you must build your application using the following options:

Category	Setting
C/C++ Compiler	Output>Generate debug information
Linker	Format>Debug information for C-SPY
Debugger	Plugins>Code Coverage

Table 15: Project options for enabling code coverage

After you have built your application and started C-SPY, choose **View>Code Coverage** to open the Code Coverage window and click **Activate** to switch on the code coverage analyzer. The following window will be displayed:

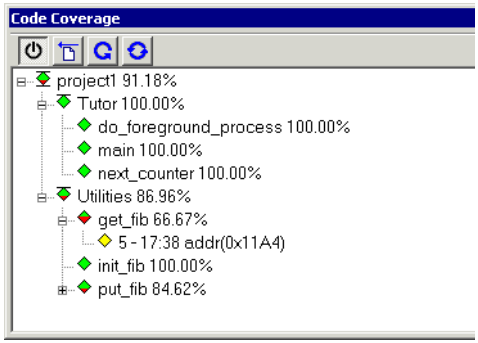


Figure 59: Code Coverage window

Viewing the figures

The code coverage information is displayed in a tree structure, showing the program, module, function and step point levels. The plus sign and minus sign icons allow you to expand and collapse the structure.

The following icons are used to give you an overview of the current status on all levels:

- A red diamond signifies that 0% of the code has been executed
- A green diamond signifies that 100% of the code has been executed
- A red and green diamond signifies that some of the code has been executed
- A yellow diamond signifies a step point that has not been executed.

The percentage displayed at the end of every program, module and function line shows the amount of code that has been covered so far, that is, the number of executed step points divided with the total number of step points.

For step point lines, the information displayed is the column number range and the row number of the step point in the source window, followed by the address of the step point.

```
<column start>-<column end>:row.
```

A step point is considered to be executed when one of its instructions has been executed. When a step point has been executed, it is removed from the window.

Double-clicking a step point or a function in the Code Coverage window displays that step point or function as the current position in the source window, which becomes the active window. Double-clicking a module on the program level expands or collapses the tree structure.

An asterisk (*) in the title bar 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. To update the information, use the **Refresh** command.

What parts of the code are displayed?

The window displays only **s**tatements that have been compiled with debug information. Thus, startup code, exit code and library code will not be displayed in the window. Furthermore, coverage information for statements in inlined functions will not be displayed. Only the statement containing the inlined function call will be marked as executed.

Producing reports

To produce a report, right-click in the window and choose the **Save As** command on the context menu. The contents of the Code Coverage window will be saved to a file.

Part 5. IAR C-SPY Simulator

This part of the MSP430 IAR Embedded Workbench® IDE User Guide contains the following chapters:

- Simulator-specific debugging
- Simulating interrupts.

Simulator-specific debugging

In addition to the general C-SPY® features, the C-SPY Simulator provides some simulator-specific features, which are described in this chapter.

You will get reference information, as well as information about driver-specific characteristics, such as memory access checking and breakpoints.

The IAR C-SPY Simulator introduction

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

FEATURES

In addition to the general features listed in the chapter *Product introduction*, the IAR C-SPY Simulator also provides:

- Instruction-accurate simulated execution
- Memory configuration and validation
- Interrupt simulation
- Immediate breakpoints with resume functionality
- Peripheral simulation (using the C-SPY macro system).

SELECTING THE SIMULATOR DRIVER

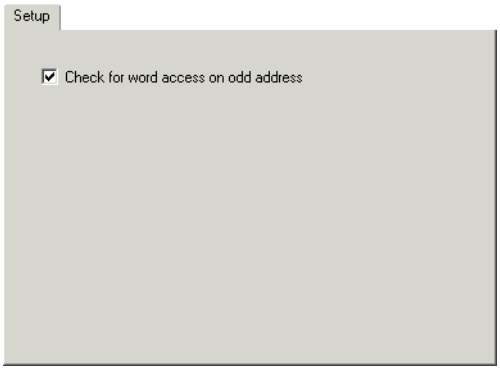
Before starting the IAR C-SPY Debugger you must choose the simulator driver. In the IAR Embedded Workbench IDE, choose **Project>Options** and click the **Setup** tab in the **Debugger** category. Choose **Simulator** from the **Driver** drop-down list.

To set simulator-specific options, choose Simulator from the **Category** list.

Note: You can only choose a driver you have installed on your computer.

SIMULATOR SETUP

The simulator **Setup** options specify the simulator-specific options.



CHECK FOR WORD ACCESS ON ODD ADDRESS

Use this option to make the simulator issue a warning if there is a word access to an odd address.

Simulator-specific menus

When you use the simulator driver, the **Simulator** menu is added in the menu bar.

SIMULATOR MENU

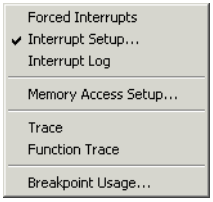


Figure 60: Simulator menu

The **Simulator** menu contains the following commands:

Menu command	Description
Interrupt Setup	Displays a dialog box to allow you to configure C-SPY interrupt simulation; see *Interrupt Setup dialog box*, page 180.
Forced Interrupts	Displays a window from which you can trigger an interrupt; see *Forced interrupt window*, page 183.
Interrupt Log	Displays a window which shows the status of all defined interrupts; see *Interrupt Log window*, page 185.
Memory Access Setup	Displays a dialog box to simulate memory access checking by specifying memory areas with different access types; see *Memory Access setup dialog box*, page 168.
Trace	Opens the Trace window with the recorded trace data; see *Trace window*, page 162.
Function Trace	Opens the Function Trace window with the trace data for which functions were called or returned from; see *Function Trace window*, page 164.
Breakpoint Usage	Displays the Breakpoint Usage dialog box which lists all active breakpoints; see *Breakpoint Usage dialog box*, page 175.

Table 16: Description of Simulator menu commands

Using the trace system in the simulator

In the C-SPY simulator, a *trace* is a recorded sequence of executed machine instructions. In addition, you can record the values of C-SPY expressions by selecting the expressions in the Trace Expressions window. The Function Trace window only shows trace data corresponding to calls to and returns from functions, whereas the Trace window displays all instructions.

For more detailed information about using the common features in the trace system, see *Using the trace system*, page 127.

TRACE WINDOW

The Trace window—available from the **Simulator** menu—displays a recorded sequence of executed machine instructions. In addition, the window can display trace data for expressions.

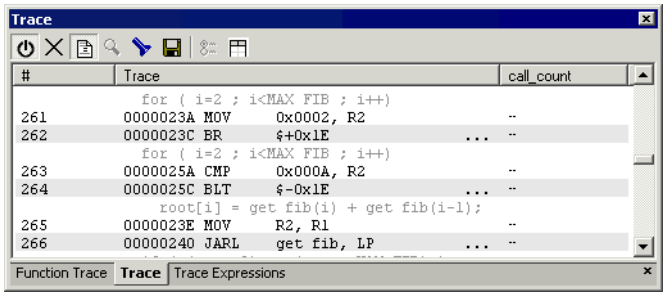


Figure 61: Trace window

C-SPY generates trace information based on the location of the program counter.

The Trace window contains the following columns:

Trace window column	Description
#	A serial number for each row in the trace buffer. Simplifies the navigation within the buffer.
Trace	The recorded sequence of executed machine instructions. Optionally, the corresponding source code can also be displayed.
Expression	Each expression you have defined to be displayed appears in a separate column. Each entry in the expression column displays the value *after* executing the instruction on the same row. You specify the expressions for which you want to record trace information in the Trace Expressions window; see *Trace Expressions window*, page 164.

Table 17: Trace window columns

For more information about using the trace system, see *Using the trace system*, page 127.

TRACE TOOLBAR

The Trace toolbar is available in the Trace window and in the Function trace window:

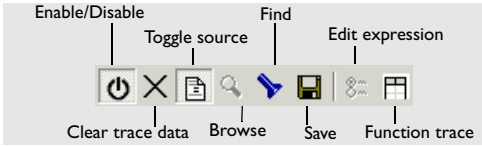


Figure 62: Trace toolbar

The following function buttons are available on the toolbar:

Toolbar button	Description
Enable/Disable	Enables and disables tracing. This button is not available in the Function trace window.
Clear trace data	Clears the trace buffer. Both the Trace window and the Function trace window are cleared.
Toggle Source	Toggles the Trace column between showing only disassembly or disassembly together with corresponding source code.
Browse	Toggles browse mode on and off for a selected item in the Trace column. For more information about browse mode, see *The Trace window and its browse mode*, page 127.
Find	Opens the **Find In Trace** dialog box where you can perform a search; see *Find in Trace dialog box*, page 166.
Save	Opens a standard Save dialog box where you can save the recorded trace information to a text file, with tab-separated columns.
Edit settings	This button is not enabled in the C-SPY simulator.
Edit Expressions	Opens the Trace Expressions window; see *Trace Expressions window*, page 164.

Table 18: Trace toolbar commands

FUNCTION TRACE WINDOW

The Function Trace window—available from the **Simulator** menu—displays a subset of the trace data displayed in the Trace window. Instead of displaying all rows, the Function Trace window only shows trace data corresponding to calls to and returns from functions.

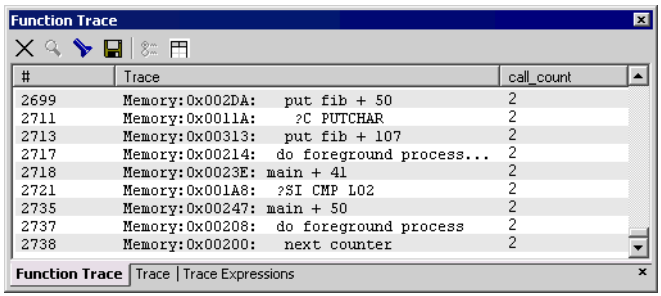


Figure 63: Function Trace window

For information about the toolbar, see *Trace toolbar*, page 163.

For more information about using the trace system, see *Using the trace system*, page 127.

TRACE EXPRESSIONS WINDOW

In the Trace Expressions window—available from the Trace window toolbar—you can specify specific expressions for which you want to record trace information.

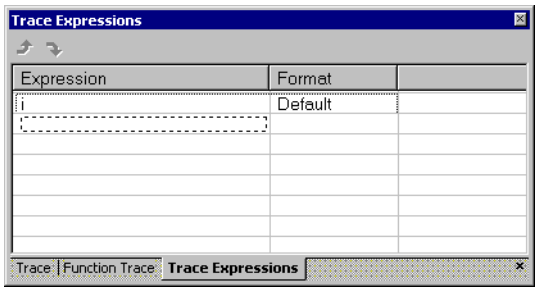


Figure 64: Trace Expressions window

In the **Expression** column, you specify any expression you want to be recorded. You can specify any expression that can be evaluated, such as variables and registers.

The **Format** column shows which display format is used for each expression.

Each row in this window will appear as an extra column in the Trace window.

For more information about using the trace system, see *Using the trace system*, page 127.

Use the toolbar buttons to change the order between the expressions:

Toolbar button	Description
Arrow up	Moves the selected row up
Arrow down	Moves the selected row down

Table 19: Toolbar buttons in the Trace Expressions window

FIND IN TRACE WINDOW

The Find In Trace window—available from the **View>Messages** menu—displays the result of searches in the trace data.

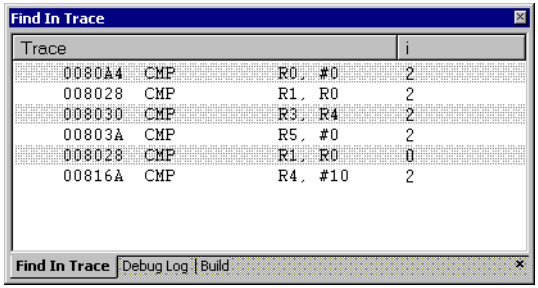


Figure 65: Find In Trace window

The Find in Trace window looks like the Trace window, showing the same columns and data, but *only* those rows that match the specified search criteria. Double-clicking an item in the Find in Trace window brings up the same item in the Trace window.

You specify the search criteria in the **Find In Trace** dialog box. For information about how to open this dialog box, see *Find in Trace dialog box*, page 166.

For more information about using the trace system, see *Using the trace system*, page 127.

FIND IN TRACE DIALOG BOX

Use the **Find in Trace** dialog box—available by choosing **Edit>Find and Replace>Find** or from the Trace window toolbar—to specify the search criteria for advanced searches in the trace data. Note that the **Edit>Find and Replace>Find** command is context-dependent. It displays the **Find in Trace** dialog box if the Trace window is the current window or the **Find** dialog box if the editor window is the current window.

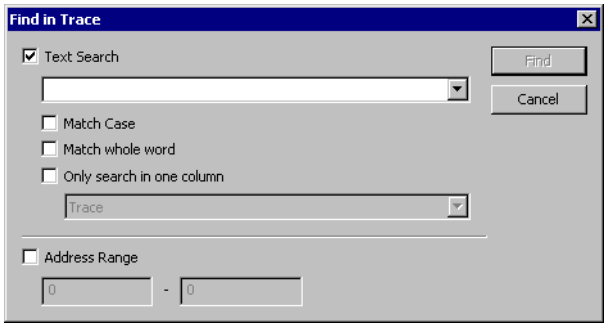


Figure 66: Find in Trace dialog box

The search results are displayed in the Find In Trace window—available by choosing the **View>Messages** command, see *Find In Trace window*, page 165.

In the **Find in Trace** dialog box, you specify the search criteria with the following settings:

Text search

A text field where you type the string you want to search for. Use the following options to fine-tune the search:

Match Case	Searches only for occurrences that exactly match the case of the specified text. Otherwise specifying `int` will also find `INT` and `Int`.
Match whole word	Searches only for the string when it occurs as a separate word. Otherwise `int` will also find `print`, `sprintf` and so on.
Only search in one column	Searches only in the column you selected from the drop-down menu.

Address Range

Use the text fields to specify an address range. The trace data within the address range is displayed. If you also have specied a text string in the **Text search** field, the text string will be searched for within the address range.

For more information about using the trace system, see *Using the trace system*, page 127.

Memory access checking

C-SPY can simulate different memory access types of the target hardware and detect illegal accesses, for example a read access to write-only memory. If a memory access occurs that does not agree with the access type specified for the specific memory area, C-SPY will regard this as an illegal access. The purpose of memory access checking is to help you to identify any memory access violations.

The memory areas can either be the zones predefined in the device description file, or memory areas based on the segment information available in the debug file. In addition to these, you can define your own memory areas. The access type can be read and write, read only, or write only. It is not possible to map two different access types to the same memory area. You can choose between checking access type violation or checking accesses to unspecified ranges. Any violations are logged in the Debug Log window. You can also choose to have the execution halted.

Choose **Simulator>Memory Access Setup** to open the **Memory Access Setup** dialog box.

MEMORY ACCESS SETUP DIALOG BOX

The **Memory Access Setup** dialog box—available from the **Simulator** menu—lists all defined memory areas, where each column in the list specifies the properties of the area. In other words, the dialog box displays the memory access setup that will be used during the simulation.

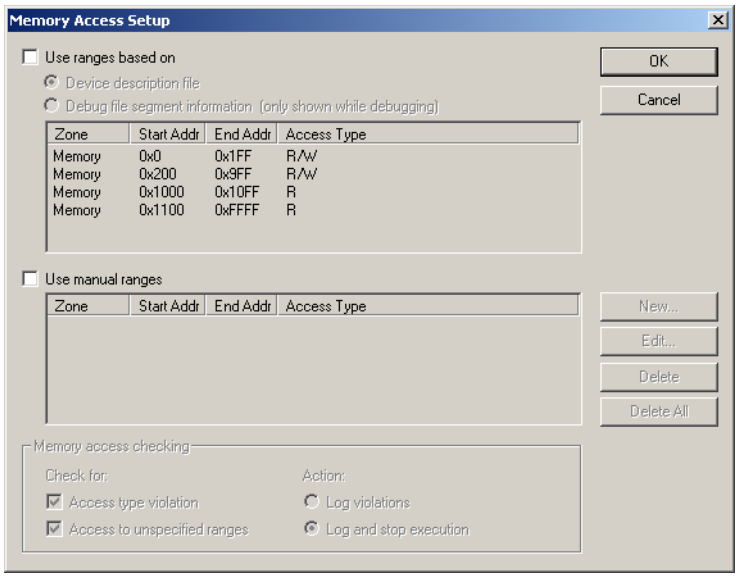


Figure 67: Memory Access Setup dialog box

Note: If you enable both the **Use ranges based on** and the **Use manual ranges** option, memory accesses will be checked for all defined ranges.

For information about the columns and the properties displayed, see *Edit Memory Access dialog box*, page 170.

Use ranges based on

Use the **Use ranges based on** option to choose any of the predefined alternatives for the memory access setup. You can choose between:

- **Device description file**, which means the properties will be loaded from the device description file
- **Debug file segment information**, which means the properties will be based on the segment information available in the debug file. This information is only available while debugging. The advantage of using this option, is that the simulator can catch memory accesses outside the linked application.

Use manual ranges

Use the **Use manual ranges** option to specify your own ranges manually via the **Edit Memory Access** dialog box. To open this dialog box, choose **New** to specify a new memory range, or select a memory zone and choose **Edit** to modify it. For more details, see *Edit Memory Access dialog box*, page 170.

The ranges you define manually are saved between debug sessions.

Memory access checking

Use the **Check for** options to specify what to check for. Choose between:

- Access type violation
- Access to unspecified ranges.

Use the **Action** options to specify the action to be performed if there is an access violation. Choose between:

- Log violations
- Log and stop execution.

Any violations are logged in the Debug Log window.

Buttons

The **Memory Access Setup** dialog box contains the following buttons:

Button	Description
OK	Standard OK.
Cancel	Standard Cancel.
New	Opens the **Edit Memory Access** dialog box, where you can specify a new memory range and attach an access type to it; see *Edit Memory Access dialog box*, page 170.
Edit	Opens the **Edit Memory Access** dialog box, where you can edit the selected memory area. See *Edit Memory Access dialog box*, page 170.
Delete	Deletes the selected memory area definition.
Delete All	Deletes all defined memory area definitions.

Table 20: Function buttons in the Memory Access Setup dialog box

Note: Except for the OK and Cancel buttons, buttons are only available when the option **Use manual ranges** is selected.

EDIT MEMORY ACCESS DIALOG BOX

In the **Edit Memory Access** dialog box—available from the **Memory Access Setup** dialog box—you can specify the memory ranges, and assign an access type to each memory range, for which you want to detect illegal accesses during the simulation.

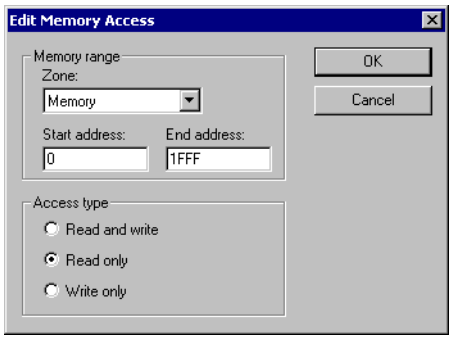


Figure 68: Edit Memory Access dialog box

For each memory range you can define the following properties:

Memory range

Use these settings to define the memory area for which you want to check the memory accesses:

Zone	The memory zone; see *Memory addressing*, page 135.
Start address	The start address for the address range, in hexadecimal notation.
End address	The end address for the address range, in hexadecimal notation.

Access type

Use one of these options to assign an access type to the memory range; the access type can be one of **Read and write**, **Read only**, or **Write only**. It is not possible to assign two different access types to the same memory area.

Using breakpoints

Using the C-SPY Simulator, you can set an unlimited amount of breakpoints. For code and data breakpoints you can define a size attribute, that is, you can set the breakpoint on a range. You can also set immediate breakpoints.

For information about the breakpoint system, see the chapter *Using breakpoints* in this guide. For detailed information about code breakpoints, see *Code breakpoints dialog box*, page 256.

DATA BREAKPOINTS

Data breakpoints are triggered when data is accessed at the specified location. Data breakpoints are primarily useful for variables that have a fixed address in memory. If you set a breakpoint on an accessible local variable, the breakpoint will be set on the corresponding memory location. The validity of this location is only guaranteed for small parts of the code. The execution will usually stop directly after the instruction that accessed the data has been executed.

You can set a data breakpoint in three different ways; by using:

- A dialog box, see *Data breakpoints dialog box*, page 171
- A system macro, see *__setDataBreak*, page 416
- The Memory window, see *Setting a breakpoint in the Memory window*, page 137.

Data breakpoints dialog box

The options for setting data breakpoints are available from the context menu that appears when you right-click in the Breakpoints window. On the context menu, choose **New Breakpoint>Data** to set a new breakpoint. Alternatively, to modify an existing breakpoint, select a breakpoint in the Breakpoint window and choose **Edit** on the context menu.

The **Data** breakpoints dialog box appears.

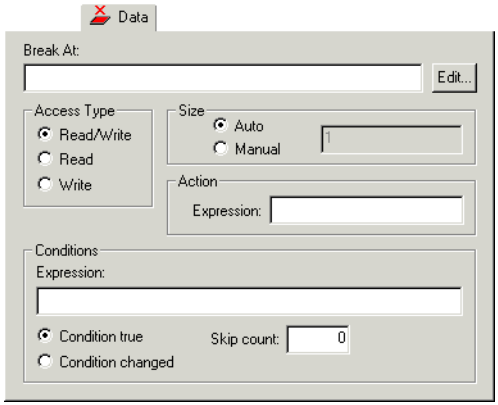


Figure 69: Data breakpoints dialog box

Break At

Specify the location for the breakpoint in the **Break At** text box. Alternatively, click the **Edit** browse button to open the **Enter Location** dialog box; see *Enter Location dialog box*, page 260.

Access Type

Use the options in the **Access Type** area to specify the type of memory access that triggers data or immediate breakpoints.

Memory Access type	Description
Read/Write	Read or write from location (not available for immediate breakpoints).
Read	Read from location.
Write	Write to location.

Table 21: Memory Access types

Note: Data breakpoints never stop execution within a single instruction. They are recorded and reported after the instruction is executed. (Immediate breakpoints do not stop execution at all, they only suspend it temporarily. See *Immediate breakpoints*, page 173.)

Size

Optionally, you can specify a size—in practice, a *range* of locations. Each read and write access to the specified memory range will trigger the breakpoint. For data breakpoints, this can be useful if you want the breakpoint to be triggered on accesses to data structures, such as arrays, structs, and unions.

There are two different ways the size can be specified:

- **Auto**, the size will automatically be based on the type of expression the breakpoint is set on. For example, if you set the breakpoint on a 12-byte structure, the size of the breakpoint will be 12 bytes
- **Manual**, you specify the size of the breakpoint manually in the **Size** text box.

Action

You can optionally connect an action to a breakpoint. You specify an expression, for instance a C-SPY macro function, which is evaluated when the breakpoint is triggered and the condition is true.

Conditions

You can specify simple and complex conditions.

Conditions	Description
Expression	A valid expression conforming to the C-SPY expression syntax.
Condition true	The breakpoint is triggered if the value of the expression is true.
Condition changed	The breakpoint is triggered if the value of the expression has changed since it was last evaluated.
Skip count	The number of times that the breakpoint must be fulfilled before a break occurs (integer).

Table 22: Breakpoint conditions

IMMEDIATE BREAKPOINTS

In addition to generic breakpoints, the C-SPY Simulator lets you set *immediate* breakpoints, which will halt instruction execution only temporarily. This allows a C-SPY macro function to be called when the processor is about to read data from a location or immediately after it has written data. Instruction execution will resume after the action.

This type of breakpoint is useful for simulating memory-mapped devices of various kinds (for instance serial ports and timers). When the processor reads at a memory-mapped location, a C-SPY macro function can intervene and supply appropriate data. Conversely, when the processor writes to a memory-mapped location, a C-SPY macro function can act on the value that was written.

The two different methods of setting an immediate breakpoint are by using:

- A dialog box, see *Immediate breakpoints dialog box*, page 173
- A system macro, see *__setSimBreak*, page 419.

Immediate breakpoints dialog box

The options for setting immediate breakpoints are available from the context menu that appears when you right-click in the Breakpoints window. On the context menu, choose **New Breakpoint>Immediate** to set a new breakpoint. Alternatively, to modify an existing breakpoint, select a breakpoint in the Breakpoint window and choose **Edit** on the context menu.

The **Immediate** breakpoints dialog box appears.

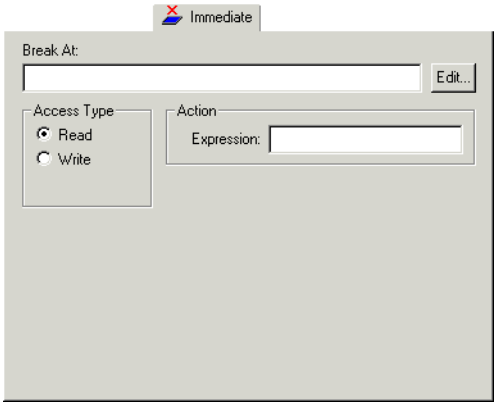


Figure 70: Immediate breakpoints page

Break At

Specify the location for the breakpoint in the **Break At** text box. Alternatively, click the **Edit** browse button to open the **Enter Location** dialog box; see *Enter Location dialog box*, page 260.

Access Type

Use the options in the **Access Type** area to specify the type of memory access that triggers data or immediate breakpoints.

Memory Access type	Description
Read	Read from location.
Write	Write to location.

Table 23: Memory Access types

Note: Immediate breakpoints do not stop execution at all; they only suspend it temporarily. See *Using breakpoints*, page 170.

Action

You should connect an action to the breakpoint. Specify an expression, for instance a C-SPY macro function, which is evaluated when the breakpoint is triggered and the condition is true.

BREAKPOINT USAGE DIALOG BOX

The **Breakpoint Usage** dialog box—available from the **Simulator** menu—lists all active breakpoints.

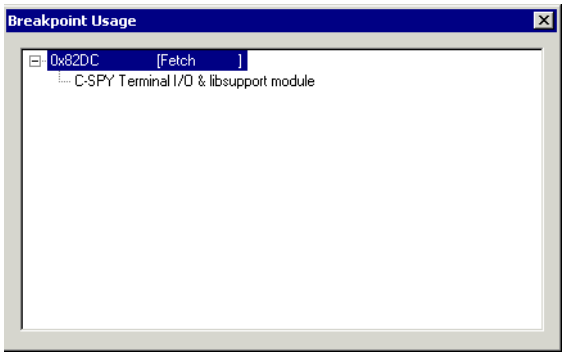


Figure 71: Breakpoint Usage dialog box

In addition to listing all breakpoints that you have defined, this dialog box also lists the internal breakpoints that the debugger is using.

For each breakpoint in the list the address and access type are shown. Each breakpoint in the list can also be expanded to show its originator.

For more information, see *Viewing all breakpoints*, page 132.

Simulating interrupts

By being able to simulate interrupts, you can debug the program logic long before any hardware is available. This chapter contains detailed information about the C-SPY® interrupt simulation system and how to configure the simulated interrupts to make them reflect the interrupts of your target hardware. Finally, reference information about each interrupt system macro is provided.

For information about the interrupt-specific facilities useful when writing interrupt service routines, see the *MSP430 IAR C/C++ Compiler Reference Guide*.

The C-SPY interrupt simulation system

The IAR C-SPY Simulator includes an interrupt simulation system that allows you to simulate the execution of interrupts during debugging. It is possible to configure the interrupt simulation system so that it resembles your hardware interrupt system. By using simulated interrupts in conjunction with C-SPY macros and breakpoints, you can compose a complex simulation of, for instance, interrupt-driven peripheral devices. Having simulated interrupts also lets you test the logic of your interrupt service routines.

The interrupt system has the following features:

- Simulated interrupt support for the MSP430 microcontroller
- Single-occasion or periodical interrupts based on the cycle counter
- Predefined interrupts for different devices
- Configuration of hold time, probability, and timing variation
- State information for locating timing problems
- Two interfaces for configuring the simulated interrupts—a dialog box and a C-SPY system macro—that is, one interactive and one automating interface
- Activation of interrupts either instantly or based on parameters you define
- A log window which continuously displays the status for each defined interrupt.

The interrupt system is activated by default, but if it is not required it can be turned off to speed up the simulation. You can turn the interrupt system on or off as required either in the **Interrupt Setup** dialog box, or by using a system macro. Defined interrupts will be preserved until you remove them. All interrupts you define using the **Interrupt Setup** dialog box are preserved between debug sessions.

INTERRUPT CHARACTERISTICS

The simulated interrupts consist of a set of characteristics which lets you fine-tune each interrupt to make it resemble the real interrupt on your target hardware. You can specify a *first activation time*, a *repeat interval*, a *hold time*, and a *variance*.

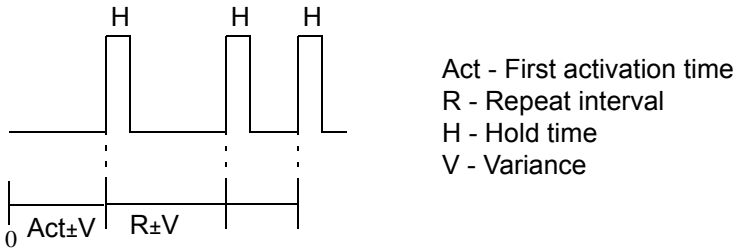


Figure 72: Simulated interrupt configuration

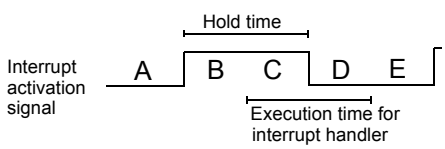
The interrupt simulation system uses the cycle counter as a clock to determine when an interrupt should be raised in the simulator. You specify the *first activation time*, which is based on the cycle counter. C-SPY will generate an interrupt when the cycle counter has passed the specified activation time. However, interrupts can only be raised between instructions, which means that a full assembler instruction must have been executed before the interrupt is generated, regardless of how many cycles an instruction takes.

To define the periodicity of the interrupt generation you can specify the *repeat interval* which defines the amount of cycles after which a new interrupt should be generated. In addition to the repeat interval, the periodicity depends on the two options *probability*—the probability, in percent, that the interrupt will actually appear in a period—and *variance*—a time variation range as a percentage of the repeat interval. These options make it possible to randomize the interrupt simulation. You can also specify a *hold time* which describes how long the interrupt remains pending until removed if it has not been processed. If the hold time is set to *infinite*, the corresponding pending bit will be set until the interrupt is acknowledged or removed.

INTERRUPT SIMULATION STATES

The interrupt simulation system contains status information that can be used for locating timing problems in your application. The **Interrupt Setup** dialog box displays the available status information. The interrupt activation signal can exist in one of the states *Idle* or *Pending*. For an interrupt, the following states can be displayed: *Executing*, *Removed*, or *Expired*.

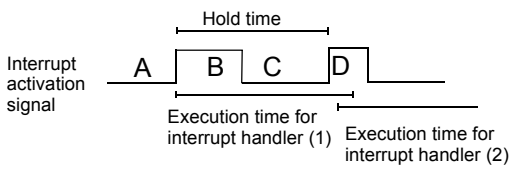
For a repeatable interrupt that has a specified repeat time which is longer than the execution time, the status information at different times can look like this:



Time	Status
A	Idle
B	Pending
C	Idle (1 executing)
D	Idle (1 executing)
E	Idle

Figure 73: Simulation states - example 1

If the interrupt repeat interval is shorter than the execution time, and the interrupt is re-entrant (or non-maskable), the status information at different times can look like this:



Time	Status
A	Idle
B	Executing
C	Idle (1 executing)
D	Executing (1 executing)

Figure 74: Simulation states - example 2

In this case, the execution time of the interrupt handler is too long compared to the repeat time, which might indicate that you should rewrite your interrupt handler and make it shorter, or that you should specify a longer repeat time for the interrupt simulation system.

Using the interrupt simulation system

The interrupt simulation system is easy to use. However, to take full advantage of the interrupt simulation system you should be familiar with how to adapt it for the processor you are using, and know how to use:

- The Forced Interrupt window
- The **Interrupts** and **Interrupt Setup** dialog boxes

- The C-SPY system macros for interrupts
- The Interrupt Log window.

TARGET-ADAPTING THE INTERRUPT SIMULATION SYSTEM

The interrupt simulation has the same behavior as the hardware. This means that the execution of an interrupt is dependent on the status of the global interrupt enable bit. The execution of maskable interrupts is also dependent on the status of the individual interrupt enable bits.

To be able to perform these actions for various derivatives, the interrupt system must have detailed information about each available interrupt. Except for default settings, this information is provided in the device description files.You can find preconfigured `ddf` files in the `430\config` directory. The default settings will be used if no device description file has been specified.

1 To load a device description file before you start C-SPY, choose **Project>Options** and click the **Setup** tab of the **Debugger** category.

2 Choose a device description file that suits your target.

Note: In case you do not find a preconfigured device description file that resembles your device, you can define one according to your needs. For details of device description files, see *Device description file*, page 113.

INTERRUPT SETUP DIALOG BOX

The **Interrupt Setup** dialog box—available by choosing **Simulator>Interrupt Setup**—lists all defined interrupts.

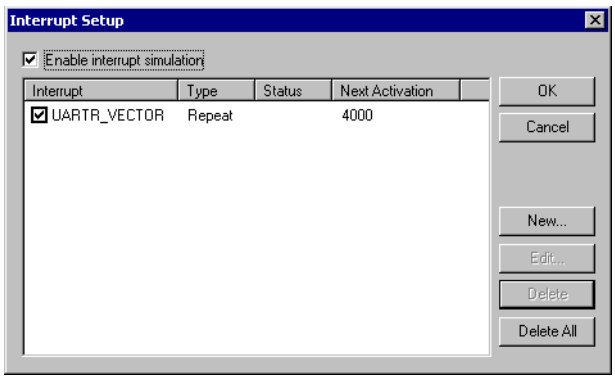


Figure 75: Interrupt Setup dialog box

The option **Enable interrupt simulation** enables or disables interrupt simulation. If the interrupt simulation is disabled, the definitions remain but no interrupts will be generated. You can also enable and disable installed interrupts individually by using the check box to the left of the interrupt name in the list of installed interrupts.

The columns contain the following information:

Interrupt	Lists all interrupts.
Type	Shows the type of the interrupt. The type can be **Forced**, **Single**, or **Repeat**.
Status	Shows the status of the interrupt. The status can be **Idle**, **Removed**, **Pending**, **Executing**, or **Expired**.
Next Activation	Shows the next activation time in cycles.

Note: For repeatable interrupts there might be additional information in the **Type** column about how many interrupts of the same type that is simultaneously executing (*n* `executing`). If *n* is larger than one, there is a reentrant interrupt in your interrupt simulation system that never finishes executing, which might indicate that there is a problem in your application.

Only non-forced interrtupts may be edited or removed.

Click **New** or **Edit** to open the **Edit Interrupt** dialog box.

EDIT INTERRUPT DIALOG BOX

Use the **Edit Interrupt** dialog box—available from the **Interrupt Setup** dialog box—to add and modify interrupts. This dialog box provides you with a graphical interface where you can interactively fine-tune the interrupt simulation parameters. You can add the parameters and quickly test that the interrupt is generated according to your needs.

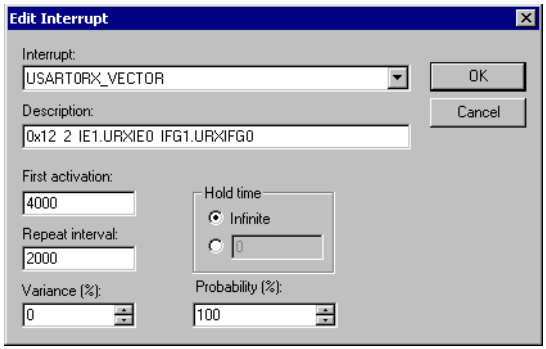


Figure 76: Edit Interrupt dialog box

For each interrupt you can set the following options:

Interrupt — A drop-down list containing all available interrupts. Your selection will automatically update the **Description** box. The list is populated with entries from the device description file that you have selected.

Description — Contains the description of the selected interrupt, if available. The description is retrieved from the selected device description file and consists of a string describing the vector address, priority, enable bit, and pending bit, separated by space characters. For interrupts specified using the system macro `__orderInterrupt`, the **Description** box will be empty.

First activation — The value of the cycle counter after which the specified type of interrupt will be generated.

Repeat interval — The periodicity of the interrupt in cycles.

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 might occur anywhere between T=95 and T=105, to simulate a variation in the timing.

Hold time	Describes how long, in cycles, the interrupt remains pending until removed if it has not been processed. If you select **Infinite**, the corresponding pending bit will be set until the interrupt is acknowledged or removed.
Probability %	The probability, in percent, that the interrupt will actually occur within the specified period.

FORCED INTERRUPT WINDOW

From the **Forced Interrupt** window—available from the **Simulator** menu—you can force an interrupt instantly. This is useful when you want to check your interrupt logistics and interrupt routines.

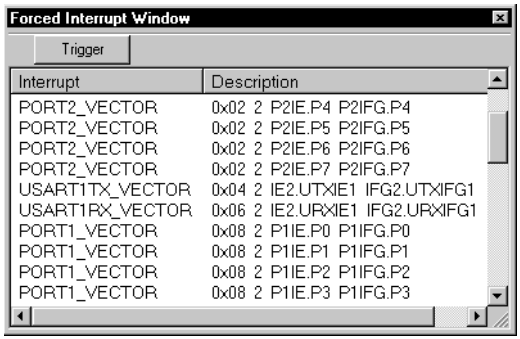


Figure 77: Forced Interrupt window

To force an interrupt, the interrupt simulation system must be enabled. To enable the interrupt simulation system, see *Interrupt Setup dialog box*, page 180.

The Forced Interrupt window lists all available interrupts and their definitions. The description field is editable and the information is retrieved from the selected device description file and consists of a string describing the vector address, priority, enable bit, and pending bit, separated by space characters.

By selecting an interrupt and clicking the **Trigger** button, an interrupt of the selected type is generated.

A triggered interrupt will have the following characteristics:

Characteristics	Settings
First Activation	As soon as possible (0)
Repeat interval	0

Table 24: Characteristics of a forced interrupt

Characteristics	Settings
Hold time	Infinite
Variance	0%
Probability	100%

Table 24: Characteristics of a forced interrupt

C-SPY SYSTEM MACROS FOR INTERRUPTS

Macros are useful when you already have sorted out the details of the simulated interrupt so that it fully meets your requirements. By writing a macro function containing definitions for the simulated interrupts you can automatically execute the functions when C-SPY starts. Another advantage is that your simulated interrupt definitions will be documented if you use macro files, and if you are several engineers involved in the development project you can share the macro files within the group.

The C-SPY Simulator provides a set of predefined system macros for the interrupt simulation system. The advantage of using the system macros for specifying the simulated interrupts is that it lets you automate the procedure.

These are the available system macros related to interrupts:

```
__enableInterrupts

__disableInterrupts

__orderInterrupt

__cancelInterrupt

__cancelAllInterrupts

__popSimulatorInterruptExecutingStack
```

The parameters of the first five macros correspond to the equivalent entries of the **Interrupts** dialog box. To read more about how to use the `__popSimulatorInterruptExecutingStack` macro, see *Interrupt simulation in a multi-task system*, page 185.

For detailed reference information about each macro, see *Description of C-SPY system macros*, page 404.

Defining simulated interrupts at C-SPY startup using a setup file

If you want to use a setup file to define simulated interrupts at C-SPY startup, follow the procedure described in *Registering and executing using setup macros and setup files*, page 147.

Interrupt simulation in a multi-task system

If you are using interrupts in such a way that the normal instruction used for returning from an interrupt handler is not used, for example in an operating system with task-switching, the simulator cannot automatically detect that the interrupt has finished executing. The interrupt simulation system will work correctly, but the status information in the **Interrupt Setup** dialog box might not look as you expect. If there are too many interrupts executing simultaneously, a warning might be issued.

To avoid these problems, you can use the `__popSimulatorInterruptExecutingStack` macro to inform the interrupt simulation system that the interrupt handler has finished executing, as if the normal instruction used for returning from an interrupt handler was executed. You can use the following procedure:

1. Set a code breakpoint on the instruction that returns from the interrupt function.
2. Specify the `__popSimulatorInterruptExecutingStack` macro as a condition to the breakpoint.

When the breakpoint is triggered, the macro is executed and then the application continues to execute automatically.

INTERRUPT LOG WINDOW

The **Interrupt Log** window—available from the **Simulator** menu—displays runtime information about the interrupts that you have activated in the **Interrupts** dialog box or forced via the **Forced Interrupt** window. The information is useful for debugging the interrupt handling in the target system.

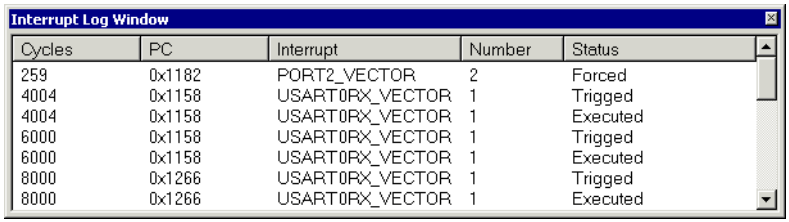


Figure 78: Interrupt Log window

The columns contain the following information:

Column	Description
Cycles	The point in time, measured in cycles, when the event occurred.
PC	The value of the program counter when the event occurred.

Table 25: Description of the Interrupt Log window

Column	Description
Interrupt	The interrupt as defined in the device description file.
Number	A unique number assigned to the interrupt. The number is used for distinguishing between different interrupts of the same type.
Status	Shows the status of the interrupt, which can be Triggered, Forced, Executing, Finished, or Expired. • Triggered: The interrupt has passed its activation time. • Forced: The same as Triggered, but the interrupt has been forced from the Forced Interrupt window. • Executing: The interrupt is currently executing. • Finished: The interrupt has been executed. • Expired: The interrupt hold time has expired without the interrupt being executed.

Table 25: Description of the Interrupt Log window (Continued)

When the Interrupt Log window is open it will be updated continuously during runtime.

Note: If the window becomes full of entries, the first entries will be erased.

Simulating a simple interrupt

In this example you will simulate a timer interrupt. However, the procedure can also be used for other types of interrupts.

This simple application contains an interrupt service routine for the BasicTimer, which increments a tick variable. The main function sets the necessary status registers. The application exits when 100 interrupts have been generated.

```
#include "io430x41x.h"
#include <intrinsics.h>

volatile int ticks = 0;
void main (void)
{
 /* Timer setup code */
 WDTCTL = WDTPW + WDTHOLD;    /* Stop WDT */
 IE2 |= BTIE;                 /* Enable BT interrupt */
 BTCTL = BTSSEL+BTIP2+BTIP1+BTIP0;
 __enable_interrupt();        /* Enable interrupts */

 while (ticks < 100);         /* Endless loop */
 printf("Done\n");
}
```

```
/* Timer interrupt service routine */
#pragma vector = BASICTIMER_VECTOR
__interrupt void basic_timer(void)
{
 ticks += 1;
}
```

To simulate and debug an interrupt, perform the following steps:

1. Add your interrupt service routine to your application source code and add the file to your project.
2. C-SPY needs information about the interrupt to be able to simulate it. This information is provided in the device description files. To select a device description file, choose **Project>Options**, and click the **Setup** tab in the **Debugger** category. Use the **Use device description file** browse button to locate the file `ddf` file.
3. Build your project and start the simulator.
4. Choose **Simulator>Interrupt Setup** to open the **Interrupts Setup** dialog box. Select the **Enable interrupt simulation** option to enable interrupt simulation. Click **New** to open the **Edit Interrupt** dialog box. For the BasicTimer example, verify the following settings:

Option	Settings
Interrupt	BASICTIMER_VECTOR
First Activation	4000
Repeat interval	2000
Hold time	Infinite
Probability %	100
Variance %	0

Table 26: Timer interrupt settings

Click **OK**.

5. Execute your application. If you have enabled the interrupt properly in your application source code, C-SPY will:
   - Generate an interrupt when the cycle counter has passed 4000
   - Continuously repeat the interrupt after approximately 2000 cycles.

Part 6. IAR C-SPY® FET debugger

This part of the MSP430 IAR Embedded Workbench® IDE User Guide contains the following chapters:

- Introduction to the IAR C-SPY® FET Debugger
- C-SPY® FET-specific debugging
- Design considerations for in-circuit programming.

Introduction to the IAR C-SPY® FET Debugger

This chapter introduces you to the IAR C-SPY Flash Emulation Tool Debugger (C-SPY FET Debugger), as well as to how it differs from the C-SPY Simulator. This chapter describes how you install the hardware and then run the demo applications. The chapter also briefly describes the communication between the C-SPY FET driver and the target system, and gives some troubleshooting hints.

The chapters specific to the C-SPY FET Debugger assumes that you already have some working knowledge of the FET Debugger, as well as some working knowledge of the IAR C-SPY Debugger. For a quick introduction, see *Part 2. Tutorials*.

Note that additional features may have been added to the software after the *MSP430 IAR Embedded Workbench® IDE User Guide* was printed. The release notes contain the latest information.

The FET C-SPY Debugger

The MSP430 microcontroller has built-in, on-chip debug support. To make the C-SPY FET Debugger work, a communication driver must be installed on the host PC. This driver is automatically installed during the installation of the IAR Embedded Workbench IDE. Because the hardware debugger kernel is built into the microcontroller, no ordinary ROM-monitor program or extra specific hardware is needed to make the debugging work. It is also possible to use the debugger on your own hardware design.

The C-SPY FET Debugger provides general C-SPY Debugger features, and features specific to the C-SPY FET driver. For detailed information about the general debugger features, see *Part 4. Debugging* in this guide.

The C-SPY FET driver uses the parallel port to communicate with the FET Interface module. The FET Interface module communicates which the JTAG interface on the hardware.

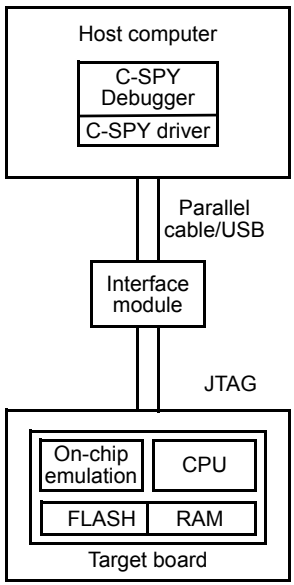


Figure 79: Communication overview

For more details about the communication, see *C-SPY FET communication*, page 225.

DIFFERENCES BETWEEN THE C-SPY DRIVERS

The following table summarizes the key differences between the FET and simulator drivers:

Feature	Simulator	FET
OP-fetch	x	x
Data breakpoints[1]	x	x
Execution in real time		x
Simulated interrupts	x	
Real interrupts		x
Cycle counter[2]	x	x
Code coverage	x	

Table 27: Simulator and FET differences

Feature	Simulator	FET
Profiling	x	x[3]
Enhanced Emulation Module support		x
Trace	x	

Table 27: Simulator and FET differences (Continued)

1. Data breakpoints are supported for the devices with the Enhanced Emulation module. For further details, see *Emulator menu*, page 202.
2. Cycle counter is supported during single step, you can then view the value of the cycle counter in the Register window.
3. The FET Debugger must single step during profiling.

Hardware installation

MSP-FET430X110

1. Connect the 25-conductor cable originating from the FET to the parallel port of your PC.
2. Ensure that the MSP430 device is securely seated in the socket, and that its pin 1 (indicated with a circular indentation on the top surface) aligns with the `1` mark on the PCB.
3. Ensure that jumpers J1 (near the non-socketed IC on the FET) and J5 (near the LED) are in place.

MSP-FET430PXX0

(P120, P140, P410, P430, P440)

1. Use the 25-conductor cable to connect the FET Interface module to the parallel port or USB port of your PC.
2. Use the 14-conductor cable to connect the FET Interface module to the Target Socket module.
3. Ensure that the MSP430 device is securely seated in the socket, and that its pin 1 (indicated with a circular indentation on the top surface) aligns with the `1` mark on the PCB.
4. Ensure that the two jumpers (LED and Vcc) near the 2x7 pin male connector are in place.

IAR J-LINK OR TI USB FET INTERFACE MODULE

1. Use the USB cable to connect the IAR J-Link or TI USB FET interface module to the USB port of your PC.

2 Windows will search for a USB driver. Since this is the first time you are using the USB interface module, Windows will open a dialog box and ask you to browse to the USB drivers. The USB drivers can be found in the product installation in the following directories:

   IAR J-Link: `430\drivers\JLink`

   TI USB FET interface module: `430\drivers\TIUSBFET\WinXP`

   Once the initial setup is completed, you will not have to repeat this step. Note that the USB interface module will blink each time it is connected until Windows makes the connection.

3 Use the 14-conductor cable to connect the USB Interface module to the Target Socket module.

4 Ensure that the MSP430 device is securely seated in the socket, and that its pin 1 (indicated with a circular indentation on the top surface) aligns with the `1` mark on the PCB.

5 Make sure that the two jumpers (LED and Vcc) near the 2x7 pin male connector are in place.

Firmware upgrade

When the C-SPY FET Debugger driver starts up, it will check that the firmware version is compatible. If an old firmware version is detected, you can choose whether it should be automatically upgrade or not. If any problems occur, follow this procedure:

1 Close the IAR Embedded Workbench IDE.

2 Unplug and replug the USB FET cable.

3 Start the IAR Embedded Workbench IDE and the C-SPY FET Debugger.

4 The debugger should ask again for firmware upgrade.

Getting started

This section demonstrates two demo applications—one in assembler language and one in C—that flash the LED. The applications are built and downloaded to the FET Debugger, and then executed.

There is one demo workspace file supplied with the C-SPY FET Debugger `fet_projects.eww`. This workspace contains two projects per FET variant—one in C and one in assembler. The files are provided in the directory `430\FET_examples`.

The majority of the examples use the various resources of the MSP430 to time the flashing of the LED.

Note: The examples often assume the presence of a 32kHz crystal, and not all FET Debuggers are supplied with a 32kHz crystal.

RUNNING A DEMO APPLICATION

The following examples assume that you are using an MSP430F149 device. See the HTML document `which FET project suits my device.htm`.

C Example

1. In the IAR Embedded Workbench IDE, choose **File>Open Workspace** to open the workspace file `fet_projects.eww`.
2. To display the C project, click the appropriate project tab at the bottom of the workspace window, for instance **fet140_1_C**.

   If you want to run the application for a different FET Debugger, click the appropriate project tab.
3. Select the **Debug** build configuration from the drop-down list at the top of the workspace window.
4. Choose **Project>Options**. In addition to the factory settings, verify the following settings:

Category	Page	Option/Setting
General Options	Target	Device: `msp430F149`
C/C++ Compiler	Output	Generate debug info
Debugger	Setup	Driver: FET Debugger
FET Debugger	Setup	**Deselect** Suppress download Connection: Select the connection type you are using

Table 28: Project options for FET C example

   For more information about the C-SPY FET Debugger options and how to configure C-SPY to interact with the target board, see *Options for debugging using the C-SPY FET debugger*, page 197.

   Click **OK** to close the **Options** dialog box.
5. Choose **Project>Make** to compile and link the source code.
6. Start C-SPY by clicking the **Debug** button or by choosing **Project>Debug**. C-SPY will erase the flash memory of the device, and then download the application to the target system.

7 In C-SPY, choose **Debug>Go** or click the **Go** button to start the application. The LED should flash.

8 Click the **Stop** button to stop the execution.

Assembler example

1 In the IAR Embedded Workbench IDE, choose **File>Open Workspace** to open the workspace file `fet_projects.eww`.

2 To display the assembler project, click the appropriate project tab at the bottom of the workspace window, for instance **fet140_1_asm**.

If you want to run the application for a different FET Debugger, click the appropriate project tab.

3 Select the **Debug** build configuration from the drop-down list at the top of the workspace window.

4 Choose **Project>Options**. In addition to the factory settings, verify the following settings:

Category	Page	Option/Setting
General Options	Target	Device: `msp430F149`
C/C++ Compiler	Output	Generate debug info
Debugger	Setup	Driver: FET Debugger
FET Debugger	Setup	**Deselect** Suppress download Connection: Select the connection type you are using

Table 29: Project options for FET assembler example

For more information about the C-SPY FET Debugger options and how to configure C-SPY to interact with the target board, see *Options for debugging using the C-SPY FET debugger*, page 197.

Click **OK** to close the **Options** dialog box.

5 Choose **Project>Make** to assemble and link the source code.

6 Start C-SPY by clicking the **Debug** button or by choosing **Project>Debug**. C-SPY will erase the flash memory of the device, and then download the application object file to the target system.

7 In C-SPY, choose **Debug>Go** or click the **Go** button to start the application. The LED should flash.

8 Click the **Stop** button to stop the execution.

C-SPY® FET-specific debugging

This chaptersection describes the additional options, menus, and features provided by the C-SPY® FET debugger driver. The chaptersection contains the following sections:

- Options for debugging using the C-SPY FET debugger
- The Emulator menu
- Using breakpoints
- Using state storage
- Using the sequencer
- Stepping
- C-SPY FET communication.

Options for debugging using the C-SPY FET debugger

Before you start any C-SPY hardware debugger you must set some options for the debugger system—both C-SPY generic options and options required for the hardware system (C-SPY driver-specific options). Follow this procedure:

1. To open the **Options** dialog box, choose **Project>Options**.
2. To set C-SPY generic options and select a C-SPY driver:
   - Select **Debugger** from the **Category** list
   - On the **Setup** page, select the **FET Debugger** driver from the **Driver** list.

   For information about the settings **Setup macros**, **Run to**, and **Device descriptions**, as well as for information about the pages **Extra Options** and **Plugins**, see the chapter *Debugger options*the *MSP430 IAR Embedded Workbench® IDE User Guide*.

   Note that a default device description file and linker command file is automatically selected depending on your selection of a device on the **General Options>Target** page.

3 To set the driver-specific options, select **FET Debugger** from the **Category** list. For details about the options specific to the FET debugger, see:

- *Setup*, page 198
- *Breakpoints*, page 200.

4 When you have set all the required options, click **OK** in the **Options** dialog box.

SETUP

The **Setup** page in the **FET debugger** category contains setup options specific to the C-SPY FET debugger.

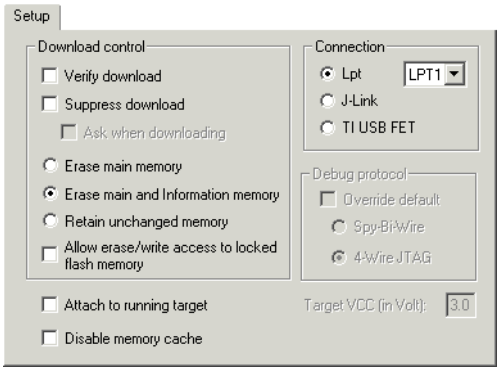


Figure 80: FET debugger setup options

Download control

Use the following options to control the download:

Verify download	Use this option to verify that the downloaded code image can be read back from target memory with the correct contents.
Suppress download	Disables the downloading of code, while preserving the present content of the flash. This command is useful if you need to exit C-SPY for a while and then continue the debug session without downloading code. The implicit RESET performed by C-SPY at startup is not disabled, though. If this option is combined with **Verify all**, the debugger will read your application back from the flash memory and verify that it is identical with the application currently being debugged.
Erase main memory	Erases only the main flash memory before download. The information memory is not erased.
Erase main and Information memory	Erases both the flash memories—main and Information memory—before download.
Retain unchanged memory	Reads the main and Information memories into a buffer. Only the flash segments needed are erased. If data that is to be written into a segment matches the data on the target, the data on the target is left as is, and no download is performed. The new data effectively replaces the old data, and unaffected old data is retained.
Allow erase/write access to locked flash memory	Controls if it should be possible to erase Info Segment A. This option can only be used with an MSP430F2xx device.

Attach to running target

Use this option to make the debugger attach to a running application at its current location, without resetting the target system. To avoid unexpected behavior when using this option, deselect the options **Debugger>Setup>Run to** and **Debugger>Plugins>Stack**.

This option must be disabled when you download the application for the first time.

Disable memory cache

Use this option to disable the memory cache in the FET debugger.

Connection

The C-SPY FET debugger can communicate with the target device via the parallel port or the USB port. Select **Lpt**, **J-Link** (USB), or **TI USB FET** to specify the connection type. If you select **Lpt** you must also specify which parallel port to use; LPT1, LPT2, or LPT3.

Debug protocol

The C-SPY FET debugger supports both the ordinary 4-wire JTAG interface and the 2-wire JTAG debug interface, also referred to as the Spy-Bi-Wire interface.
Spy-By-Wire works for the parallel port FET module and the TI USB FET module.

Target VCC

Use the **Target VCC** option to specify the voltage provided by the USB interface. Type the value in Volts with one decimal's precision in the range 1.0–4.0 V. This option can only be used with a USB connection.

BREAKPOINTS

The **Breakpoints** page in the **FET debugger** category contains options specific to breakpoints.

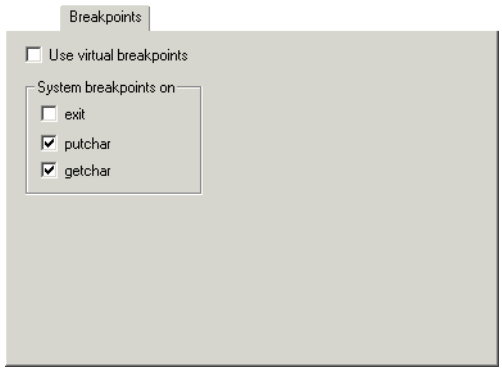


Figure 81: FET debugger breakpoint options

Use virtual breakpoints

The option **Use virtual breakpoints** allows C-SPY to use virtual breakpoints when all available hardware breakpoints have been used. When virtual breakpoints are used, C-SPY is forced into single-step mode.

To prevent C-SPY form entering single-step mode, disable this option. In this case C-SPY will not use virtual breakpoints, even though all hardware breakpoints are already used. For further information, see *Available breakpoints*, page 204the *MSP430 IAR Embedded Workbench® IDE User Guide*.

System breakpoints on

The option **System breakpoints on** can be used for fine-tuning the use of system breakpoints in the CLIB runtime environment. If the C-SPY Terminal I/O window is not required or if you do not need a breakpoint on the `exit` label, you can save hardware breakpoints by not reserving system breakpoints. Select or deselect the options **exit**, **putchar**, and **getchar** respectively, if you want, or not want, C-SPY to use system breakpoints for these. For further information, see *Available breakpoints*, page 204the *MSP430 IAR Embedded Workbench® IDE User Guide*.

In the DLIB runtime environment, C-SPY will always set a system breakpoint on the `__DebugBreak` label. You cannot disable this behavior.

Emulator menu

Using the C-SPY FET driver creates a new menu on the menu bar—the **Emulator** menu.

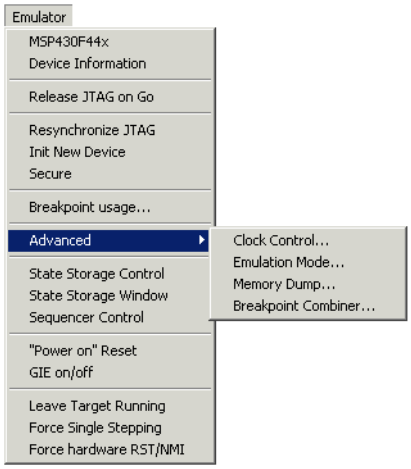


Figure 82: Emulator menu

The following commands are available on the menu:

Menu Command	Description
Connected device	The name of the device used for debugging.
Device information	Opens a window with information about the target device being used.
Release JTAG on Go	Sets the JTAG drivers in tri-state so that the device is released from JTAG control—TEST pin is set to GND—when **GO** is activated.
Resynchronize JTAG	Regains control of the device. It is not possible to Resynchronize JTAG while the device is operating.
Init New Device	Initializes the device according to the specified options on the **Flash Emulation Tool** page. The current program file is downloaded to the device memory, and the device is then reset. This command can be used to program multiple devices with the same program from within the same C-SPY session. It is not possible to choose **Init New Device** while the device is operating, thus the command will be dimmed.

Table 30: Emulator menu commands

	Menu Command	Description
	Secure	Blows the fuse on the target device. After the fuse is blown, no communication with the device is possible.
	Breakpoint Usage	Displays the **Breakpoint Usage** dialog box which lists all active breakpoints; see *Breakpoint Usage dialog box*, page 216.
	Advanced>Clock Control	Depending on the hardware support, clock control comes in one of two variants, General Clock Control or Extended Clock Control. Extended Clock Control gives you module level control over the clocks.
	Advanced> Emulation Mode	Specifies the device to be emulated. The device must be reset (or reinitialized by using the menu command **Init New Device**) following a change to the emulation mode.
	Advanced> Memory Dump	Writes the specified device memory contents to a specified file. A dialog box is displayed where you can specify a filename, a memory starting address, and a length. The addressed memory is then written in a text format to the named file. Options permit you to select word or byte text format, and address information and register contents can also be appended to the file. The Dump Memory length specifier is restricted to four hexadecimal digits (`0-FFFF`). This limits the number of bytes that can be written from **0** to **65535**. Consequently, it is not possible to write memory from `0` to `0xFFFF` inclusive as this would require a length specifier of **65536** (or `0x10000`).
	Advanced> Breakpoint Combiner	Combines two already defined breakpoints. Select a breakpoint in the **Breakpoint combiner** dialog box, then right-click to display a list to select the breakpoint to combine it with. Only available if you are using a device that supports the Enhanced Emulation Module. The settings are not saved when the debug session is closed.
	State Storage Control	Opens the State Storage Control window, which lets you define the use of the state storage module. This is only possible if you are using a device that contains support for the Enhanced Emulation Module.
	State Storage Window	Opens the State Storage window which contains state storage information according to your definitions.
	Sequencer Control	Opens the Sequencer Control window, which lets you define a state machine.
	"Power on" Reset	The device is reset by cycling power to the device.
GIE	GIE on/off	Clears the General Interrupt Enable bit (GIE) in the Processor Status register.
	Leave Target Running	Leaves the application running on the target hardware after the debug session is closed.

Table 30: Emulator menu commands (Continued)

	Menu Command	Description
	Force Single Stepping	Forces single step debugging.
	Force hardware RST/NMI	Forces an RST/NMI clear reset when the Reset button is pressed.

Table 30: Emulator menu commands (Continued)

Note: Not all **Emulator>Advanced** submenus are available on all MSP430 devices.

Using breakpoints

This section provides an overview of the available breakpoints for the C-SPY FET Debugger. The following is described:

- *Available breakpoints*, page 204
- *Customizing the use of breakpoints, page 206*
- *Range breakpoints*, page 207
- *Conditional breakpoints*, page 210
- *Advanced trigger breakpoints*, page 213
- *Breakpoint Usage dialog box*, page 216.

For information about the different methods for setting breakpoints, the facilities for monitoring breakpoints, and the different breakpoint consumers, see the chapter *Using breakpoints*, page 129 in this guidethe *MSP430 IAR Embedded Workbench® IDE User Guide*.

AVAILABLE BREAKPOINTS

With the C-SPY FET Debugger you can set *code* breakpoints. If you are using a device that supports the *Enhanced Emulation Module* you also have access to an extended breakpoint system with support for:

- breakpoints on addresses, data, and registers
- defining which type of access that should trigger the breakpoint: read, write, or fetch
- range breakpoints
- setting conditional breakpoints
- triggering different actions: stopping the execution, or starting the state storage module.

The Enhanced Emulation Module also gives you access to the sequencer module which is a state machine that uses breakpoints for triggering new states.

Hardware and virtual breakpoints

To set breakpoints, the C-SPY FET Debugger uses the hardware breakpoints available on the device. When all hardware breakpoints are used, C-SPY can use *virtual breakpoints* (can also be referred to as software breakpoints), which means that you can set an unlimited amount of breakpoints.

The number of available hardware breakpoints for each device is:

Device	Breakpoints (N)	Range breakpoints
MSP430F11x1	2	
MSP430F11x2	2	
MSP430F12x	2	
MSP430F12x2	2	
MSP430F13x	3	x
MSP430F14x	3	x
MSP430F15x	8	x
MSP430F16x	8	x
MSP430F20xx	2	
MSP430F21xx	2	
MSP430F41x	2	
MSP430F42x	2	
MSP430F43x	8	x
MSP430F44x	8	x
MSP430FE42x	2	
MSP430FG43x	2	
MSP430FW42x	2	
MSP430FG43x	2	
MSP430FW42x	2	

Table 31: Available hardware breakpoints

For the latest device information, see the release notes.

If there are N or fewer breakpoints active, C-SPY will always operate at full speed. If there are more than N breakpoints active, and virtual breakpoints are enabled, C-SPY will be forced to single step between the breakpoints. This means that execution will not be at full speed.

System breakpoints

Sometimes C-SPY must set breakpoints for internal use. These breakpoints are called *system breakpoints*. In the CLIB runtime environment, C-SPY will set a system breakpoint when:

- the library functions `putchar()` and `getchar()` are used (low-level routines used by functions like `printf` and `scanf`)
- the application has an `exit` label.

In the DLIB runtime environment, C-SPY will set a system breakpoint on the `__DebugBreak` label.

C-SPY will also set a temporary system breakpoint when:

- the command **Edit>Run to Cursor** is used
- the option **Run to** is selected

The system breakpoints will use hardware breakpoints when available. When the number of available hardware breakpoints is exceeded, virtual breakpoints will be used instead.

When the **Run to** option is selected and all hardware breakpoints have already been used, a virtual breakpoint will be set even if you have deselected the **Use virtual breakpoints** option. When you start the debugger under these conditions, C-SPY will prompt you to choose whether you want to execute in single-step mode or stop at the first instruction.

CUSTOMIZING THE USE OF BREAKPOINTS

It is possible to prevent the debugger from executing in single-step mode. You do this by disabling the use of virtual breakpoints and—in the CLIB runtime environment—by fine-tuning the use of system breakpoints. This will increase the performance of the debugger, but you will only have access to the available number of hardware breakpoints. For further information about the necessary options, see *Breakpoints*, page 200.

Periodically monitoring data

If you are using a device that does not support the Enhanced Emulation Module, the break-on-data capability of the MSP430 is not utilized. In that case, breakpoints can only be set to occur during an instruction fetch. However, C-SPY provides a non-realtime data breakpoint mechanism, which lets you periodically monitor data without using data breakpoints. For a description of the data breakpoint mechanism, see the chapter *Using breakpointsMSP430 IAR Embedded Workbench® IDE User Guide*.

Using breakpoints when programming flash memory

When programming the flash memory, do not set a breakpoint on the instruction immediately following the *write to flash* operation. A simple work-around is to follow the *write to flash* operation with a `NOP` instruction, and set a breakpoint on the instruction following the `NOP` instruction.

RANGE BREAKPOINTS

Range breakpoints can be set on a data or an address range, and the action can be specified to occur on an access inside or outside the specified range. These breakpoints are only available if you are using a device that supports the Enhanced Emulation Module.

The options for setting range breakpoints are available from the context menu that appears when you right-click in the Breakpoints window. On the context menu, choose **New Breakpoint>Range** to set a new breakpoint. Alternatively, to modify an existing breakpoint, select a breakpoint in the Breakpoint window and choose **Edit** on the context menu.

The **Range** breakpoints dialog box appears.

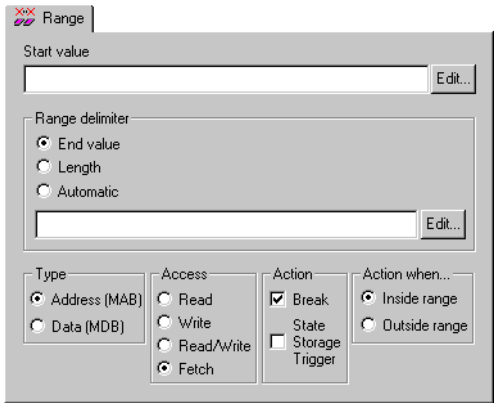


Figure 83: Range breakpoints dialog box

Note: You can also use a C-SPY system macro to set a range breakpoint, see *__setRangeBreak*, page 417.

Start value

Set the start value location for the range breakpoint using the **Edit** button. These are the locations you can choose between and their possible settings:

Location	Description/Examples
Expression	Any expression that evaluates to a valid address, such as a variable name. 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`.
Absolute Address	An absolute location on the form `zone:hexaddress` or simply `hexaddress`. Zone specifies in which memory the address belongs. For example `Memory:0x42` If you enter a combination of a Zone and an address that is not valid, C-SPY will indicate the mismatch.
Source Location	A location in the C source program using the syntax `{file path}.row.column` *File* specifies the filename and full path. *Row* specifies the row in which you want the breakpoint. *Column* specifies the column in which you want the breakpoint. For example, `{C:\IAR Systems\xxx\Utilities.c}.22.3` sets a breakpoint on the third character position on line 22 in the source file `Utilities.c`.

Table 32: Range breakpoint start value types

Range delimiter

This option sets the end location of the range. It can be one of the value types used for the **Start value**, the **Length** of the range in hexadecimal notation, or **Automatic**. Automatic means that the range will automatically be based on the type of expression the breakpoint is set on. For example, if you set the breakpoint on a 12-byte structure, the range of the breakpoint will be 12 bytes.

Type

To choose which breakpoint type to use, select one of the following options:

Breakpoint type	Description
Address (MAB)	Sets a breakpoint on a specified address, or anything that can be evaluated to one. The breakpoint is triggered when the specified location is accessed. If you have set the breakpoint on a specific machine instruction, the breakpoint will be triggered and the execution will stop exactly before the instruction will be executed.
Data (MDB)	Sets a breakpoint on a specified value. It is the value on the data bus that triggers the breakpoint.

Table 33: Range breakpoint types

Access type

You can specify the type of access that triggers the selected breakpoint. Select one of the following options:

Access	Description
Read	Read from location.
Write	Write to location.
Read/Write	Read from or write to location.
Fetch	At instruction fetch.

Table 34: Range breakpoint access types

Action

There are two action options—**Break** and **State Storage Trigger**.

If you select the option **Break,** the execution will stop when the breakpoint is triggered.

If you select the option **State Storage Trigger**, the breakpoint is defined as a state storage trigger. To define the behavior of the state storage module further, use the options in the State Storage Control window.

Action when

Specifies whether the action should occur at an access inside or outside of the specified range.

CONDITIONAL BREAKPOINTS

Conditional breakpoints are only available if you are using a device that supports the Enhanced Emulation Module.

The options for setting conditional breakpoints are available from the context menu that appears when you right-click in the Breakpoints window. On the context menu, choose **New Breakpoint>Conditional** to set a new breakpoint. Alternatively, to modify an existing breakpoint, select a breakpoint in the Breakpoint window and choose **Edit** on the context menu.

The **Conditional** breakpoints dialog box appears.

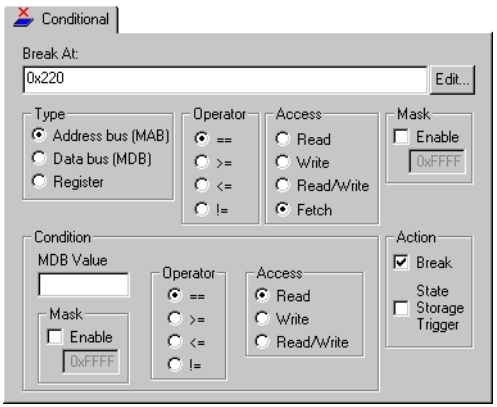


Figure 84: Conditional breakpoints dialog box

Note: You can also use a system macro to set a conditional breakpoint, see *__setConditionalBreak*, page 415.

Break At location

Set the break location using the **Edit** button. These are the locations you can choose between and their possible settings:

Location	Description/Examples
Expression	Any expression that evaluates to a valid address, such as a function or variable name. 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`.

Table 35: Conditional break at location types

Location	Description/Examples
Absolute Address	An absolute location on the form `zone:hexaddress` or simply `hexaddress`. Zone specifies in which memory the address belongs. For example `Memory:0x42` If you enter a combination of a Zone and an address that is not valid, C-SPY will indicate the mismatch.
Source Location	A location in the C source program using the syntax `{file path}.row.column` *File* specifies the filename and full path. *Row* specifies the row in which you want the breakpoint. *Column* specifies the column in which you want the breakpoint. Note that the Source Location type is only meaningful for code breakpoints. For example, `{C:\IAR Systems\xxx\Utilities.c}.22.3` sets a breakpoint on the third character position on line 22 in the source file `Utilities.c`.

Table 35: Conditional break at location types (Continued)

Type

To choose which breakpoint type to use, select one of the following options:

Breakpoint type	Description
Address bus (MAB)	Sets a breakpoint on a specified address, or anything that can be evaluated to one. The breakpoint is triggered when the specified location is accessed. If you have set the breakpoint on a specific machine instruction, the breakpoint will be triggered and the execution will stop exactly before the instruction will be executed.
Data bus (MDB)	Sets a breakpoint on a specified value. It is the value on the data bus that triggers the breakpoint.
Register	Sets a breakpoint on a register. In the **Register Value** text box, type the value that should trigger the breakpoint. Specify the register, or anything that can be evaluated to such, in the **Break At** text box.

Table 36: Conditional breakpoint types

Operator

You can specify one of the following condition operators for when the breakpoint should be triggered:

Condition	Description
==	Equal to.
>=	Greater than or equal to.
<=	Less than or equal to.
!=	Not equal to.

Table 37: Conditional breakpoint condition operators

Access

You can specify the type of access that triggers the selected breakpoint. Select one of the following options:

Access	Description
Read	Read from location.
Write	Write to location.
Read/Write	Read from or write to location.
Fetch	At instruction fetch.

Table 38: Conditional breakpoint access types

Mask

You can specify a bit mask value that the breakpoint address or value will be masked with. (On the FET hardware the mask is inverted, but this is *not* the case in the FET Debugger driver.)

Condition

You can specify an additional condition to a conditional breakpoint. This means that a conditional breakpoint can be a single data breakpoint or a combination of two breakpoints that must occur at the same time. The following settings can be specified for the additional condition:

Access	Description
MDB/Register Value	The extra conditional data value.
Mask	The bit mask value that the breakpoint value will be masked with.

Table 39: Conditional breakpoint condition types

Access	Description
Operator	The operator of condition, either ==, >=, <=, or !=.
Access	The access type of the condition, either Read, Write, or Read/Write.

Table 39: Conditional breakpoint condition types (Continued)

Action

There are two action options—**Break** and **State Storage Trigger**.

If you select the option **Break,** the execution will stop when the breakpoint is triggered.

If you select the option **State Storage Trigger**, the breakpoint is defined as a state storage trigger. To define the behavior of the state storage module further, use the options in the State Storage Control window.

ADVANCED TRIGGER BREAKPOINTS

Advanced trigger breakpoints are only available if you are using a device that supports the Enhanced Emulation Module.

The options for setting advanced trigger breakpoints are available from the context menu that appears when you right-click in the Breakpoints window. On the context menu, choose **New Breakpoint>Advanced Trigger** to set a new breakpoint. Alternatively, to modify an existing breakpoint, select a breakpoint in the Breakpoint window and choose **Edit** on the context menu.

The **Advanced Trigger** breakpoints dialog box appears.

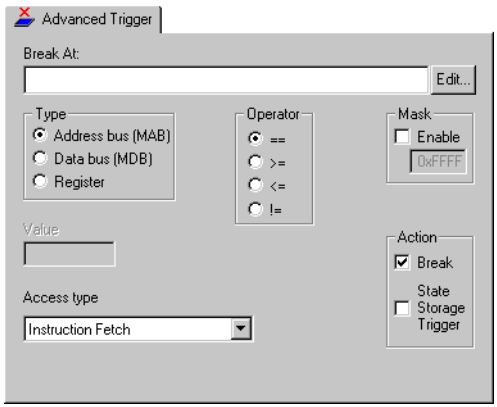


Figure 85: Advanced trigger dialog box

Note: You can also use a C-SPY system macro to set an advanced trigger breakpoint, *__setAdvancedTriggerBreak*, page 412.

Break At location

Set the break location using the **Edit** button. These are the locations you can choose between and their possible settings:

Location	Description/Examples
Expression	Any expression that evaluates to a valid address, such as a function or variable name. 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`.
Absolute Address	An absolute location on the form `zone:hexaddress` or simply `hexaddress`. Zone specifies in which memory the address belongs. For example `Memory:0x42` If you enter a combination of a Zone and an address that is not valid, C-SPY will indicate the mismatch.
Source Location	A location in the C source program using the syntax `{file path}.row.column` `file` specifies the filename and full path. `row` specifies the row in which you want the breakpoint. `column` specifies the column in which you want the breakpoint. Note that the Source Location type is only meaningful for code breakpoints. For example, `{C:\IAR Systems\xxx\Utilities.c}.22.3` sets a breakpoint on the third character position on line 22 in the source file `Utilities.c`.

Table 40: Advanced triggers break at location types

Type

To choose which breakpoint type to use, select one of the following options:

Breakpoint type	Description
Address bus (MAB)	Sets a breakpoint on a specified address, or anything that can be evaluated to one. The breakpoint is triggered when the specified location is accessed. If you have set the breakpoint on a specific machine instruction, the breakpoint will be triggered and the execution will stop exactly before the instruction will be executed.
Data bus (MDB)	Sets a breakpoint on a specified value. It is the value on the data bus that triggers the breakpoint.

Table 41: Advanced trigger types

Breakpoint type	Description
Register	Sets a breakpoint on a register. In the **Register Value** text box, type the value that should trigger the breakpoint. Specify the register, or anything that can be evaluated to such, in the **Break At** text box.

Table 41: Advanced trigger types (Continued)

Operator

You can specify one of the following condition operators for when the breakpoint should be triggered:

Condition	Description
==	Equal to.
>=	Greater than or equal to.
<=	Less than or equal to.
!=	Not equal to.

Table 42: Advanced trigger condition operators

Mask

You can specify a bit mask value that the breakpoint address or value will be masked with. (On the FET hardware the mask is inverted, but this is *not* the case in the FET Debugger driver.)

Value

The data value in the specified register that should be triggered.

Access type

Use this option to specify the type of access that triggers the selected breakpoint.

Action

There are two action options—**Break** and **State Storage Trigger**.

If you select the option **Break,** the execution will stop when the breakpoint is triggered.

If you select the option **State Storage Trigger**, the breakpoint is defined as a state storage trigger. To define the behavior of the state storage module further, use the options in the State Storage Control window.

BREAKPOINT USAGE DIALOG BOX

HistoryThe **Breakpoint Usage** dialog box—available from the driver-specific menu—lists all active breakpoints.

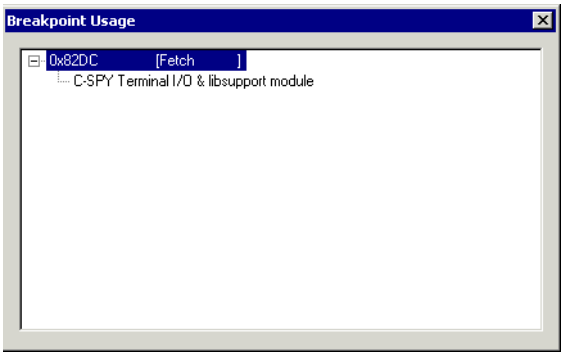


Figure 86: Breakpoint Usage dialog box

In addition to listing all breakpoints that you have defined, this dialog box also lists the internal breakpoints that the debugger is using.

For each breakpoint in the list the address and access type are shown. Each breakpoint in the list can also be expanded to show its originator.

For more information, see *Viewing all breakpoints*, page 132the *MSP430 IAR Embedded Workbench® IDE User Guide*.

Using state storage

The state storage module is a limited variant of a traditional trace module. It can store eight states and can be used for monitoring program states or program flow, without interfering with the execution. The state storage module is only available if you are using a device that supports the Enhanced Emulation Module.

To use the state storage module, you must:

- Define one or multiple range breakpoints or conditional breakpoints that you want to trigger the state storage module. In the breakpoints dialog box, make sure to select the action **State Storage Trigger**. This means that the breakpoint is defined as a state storage trigger. (State storage can also be triggered from the Sequencer Control window.)

Note that depending on the behavior you want when the state storage module is triggered, it is useful to consider the combination of the **Action** options and the options available in the State Storage Control window. See the examples following immediately after these steps.

2 Choose **Emulator>State Storage Control** to open the State Storage Control window.

3 Select the option **Enable state storage**. Set the options **Buffer wrap around**, **Trigger action**, and **Storage action** according to your preferences.

In the list **State Storage Triggers**, all breakpoints defined as state storage triggers are listed.

For further details about the options, see *State Storage Control window*, page 218.

4 Click **Apply**.

5 Choose **Emulator>State Storage window** to open the State Storage window.

6 Choose **Debug>Go** to execute your application. Before you can view the state storage information, you must stop the execution. You can do this, for instance, by using the **Break** command.

For information about the window contents, see *State Storage Window*, page 220.

As an example, assume the following setup:

- There is a conditional breakpoint which has both of the action options selected—**Break** and **State Storage Trigger**
- The state storage options **Instruction fetch** and **Buffer wrap around** are selected in the State Storage Control window.

This will start the state storage immediately when you start executing your application. When the breakpoint is triggered, the execution will stop and the last eight states can be inspected in the State Storage window.

However, if you do not want the state storage module to start until a specific state is reached, you would usually not want the execution to stop, because no state information has been stored yet.

In this case, select the **State Storage Trigger** action in the **Conditional breakpoints** dialog box (making sure that **Break** is deselected), and deselect the option **Buffer wrap around** in the State Storage Control window.

When the breakpoint is triggered, the execution will not stop, but the state storage will start. Because the option **Buffer wrap around** is deselected, you have ensured that the data in the buffer will not be overwritten.

When another breakpoint (which has **Break** selected) is triggered, or if you stop the execution by clicking the **Break** button, the State Storage window will show eight states starting with the breakpoint that was used for starting the state storage module.

STATE STORAGE CONTROL WINDOW

Use the State Storage Control window—available from the **Emulator** menu—to define how to use the state storage module available on devices that support the Enhanced Emulation Module.

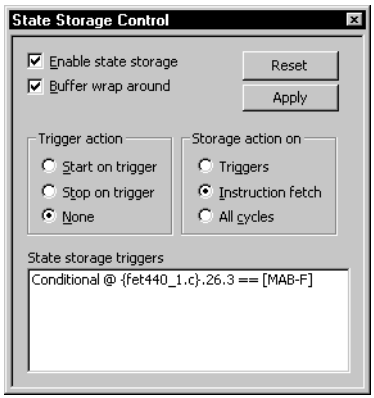


Figure 87: State Storage Control window

Enable state storage

This option enables the state storage module.

Buffer wrap around

This option controls if the state storage buffer should wrap around. If you select the option **Buffer wrap around** the state storage buffer is continuously overwritten until the execution is stopped or a breakpoint is triggered. Only the eight last states are stored.

Alternatively, in order not to overwrite the information in the state storage buffer, deselect this option. To guarantee that the eight first states will be stored, you should also click **Reset**.

Reset

Resets the state storage module.

Trigger action

This option acts upon the breakpoints that are defined as state storage triggers. The option defines what action should take place when these breakpoints have been triggered. You can choose between the following options:

Start on trigger	State storage starts when the breakpoint is triggered.
Stop on trigger	State storage starts immediately when execution starts. State storage stops when the breakpoint is triggered.
None	State storage starts immediately when execution starts. State storage does not stop when the breakpoint is triggered. However, if execution stops, state storage also stops but it will resume when execution resumes.

Storage action on

This option defines when the state information should be collected. You can choose between the following options:

Triggers	Stores state information at the time when the state storage trigger is triggered. That is, when the breakpoint defined as a state storage trigger is triggered.
Instruction fetch	Stores state information at all instruction fetches.
All cycles	Stores state information for all cycles.

State storage triggers

Lists all the breakpoints that are defined as state storage triggers. That is, the breakpoints that have the action **State Storage Trigger** selected.

STATE STORAGE WINDOW

The State Storage window—available from the **Emulator** menu—displays state storage information for eight states. Invalid data is displayed in red color.

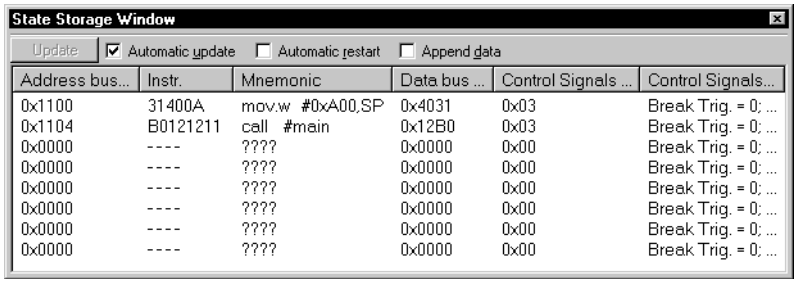


Figure 88: State Storage window

Update

Click the update button to refresh the data in the State Storage window, alternatively to append new data.

Automatic update

Select this option to automatically update the data in the state storage window each time new data is available in the state storage buffer.

Automatic restart

Select this option to reset the state storage module for consecutive data readouts after each readout.

Append data

Select this option to append collected data from the state storage buffer to the data that is already present in the State Storage window. The new data is added below the data that is already present.

The window contains the following columns:

Column	Description
Address bus	Displays the stored value of the address bus.
Instruction	Displays the instruction.
Mnemonic	Displays the mnemonic.

Table 43: Columns in State Storage window

Column	Description
Data bus	Displays the stored content of the data bus.
Control signals (byte)	Displays the stored value of the control signals during storage. Bit 1: Instruction fetch Bit 2: Byte/Word Bit 3: Interrupt request Bit 4: CPU off Bit 5: The value of the Power Up Clear (PUC) signal Bit 6:ZERO|HALT (which one depends on the used device) Bit 7: Break trigger
Control signals (bits)	Displays each bit in the stored value of the control signals during storage.

Table 43: Columns in State Storage window (Continued)

Using the sequencer

The sequencer module lets you break the execution or trigger the state storage module by using a more complex method than a standard breakpoint. This is useful if you want to stop the execution under certain conditions, for instance a specific program flow. You can combine this with letting the state storage module continuously store information. At the time when the execution stops, you will have useful state information logged in the State storage window.

Consider this example:

```
void my_putchar(char c)
{
 ...
 /* Code suspected to be erroneous */
 ...
}

void my_function(void)
{
 ...
 my_putchar('a');
 ...
 my_putchar('x');
 ...
 my_putchar('@');
 ...
}
```

In this example, the customized `putchar` function `my_putchar()` has for some reason a problem with special characters. To locate the problem, it might be useful to stop execution when the function is called, but only when it is called with one of the problematic characters as the argument.

To achieve this, you can:

1. Set a breakpoint on the statement `my_putchar('@');`.
2. Set another breakpoint on the suspected code within the function `my_putchar()`.
3. Define these breakpoints as transition triggers. Choose **Emulator>Sequencer Control** to open the Sequencer Control window. Select the option **Enable sequencer**.
4. In this simple example you will only need two transition triggers. Make sure the following options are selected:

Option	Setting
Transition trigger 0	The breakpoint which is set on the function call `my_putchar('@');`
Transition trigger 1	The breakpoint which is set on the suspected code within the function `my_putchar()`
Action	Break

Table 44: Sequencer settings - example

The transition trigger 1 depends on the transition trigger 0. This means that the execution will stop only when the function `my_putchar()` is called by the function call `my_putchar('@');`

Click **OK**.

5. Now you should set up the state storage module. Choose **Emulator>State Storage Control** to open the State Storage Control window. Make sure the following options are selected:

Option	Setting
Enable state storage	Selected
Buffer wrap around	Selected
Storage action	Instruction fetch
Trigger action	None

Table 45: State Storage Control settings - example

Click **OK**.

6 Start the program execution. The state storage module will continuously store trace information. Execution stops when the function `my_putchar()` has been called by the function call `my_putchar('@');`

7 Choose **Emulator>State Storage Window** to open the **State Storage** window. You can now examine the stored trace information. For further details, see *State Storage Window*, page 220.

8 When the sequencer is in state 3, C-SPY's breakpoint mechanism—which is used for all breakpoints, not only transition triggers—can be locked. Therefore, you should always end the session with one of these steps:

- Disabling the sequencer module. This will restore all breakpoint actions.
- Resetting the state machine by clicking the **Reset States** button. The sequencer will still be active and trigger on the defined setup during the program execution.

SEQUENCER CONTROL WINDOW

The Sequencer Control window—available from the **Emulator** menu—lets you break the execution or trigger the state storage module by using a more complex method than a standard breakpoint. This is useful if you, for instance, want to stop the execution or start the state storage module under certain conditions, for instance a specific program flow. The sequencer is only available if you are using a device that supports the Enhanced Emulation Module.

The sequencer works as a state machine. In a simple setup, you can define three transition triggers, where the last one triggers an action.

In an advanced setup, the state machine can have four states (0-3). State 0 is the starting state, and state 3 is the state that triggers a breakpoint. This breakpoint can be designed either to stop execution, or to trigger the state storage module.

For each state you can define up to two different transitions (a-b) to other states. For each transition you define a transition trigger and which the next state should be. For state 3 you must also define an action: stop the execution or start the state storage module.

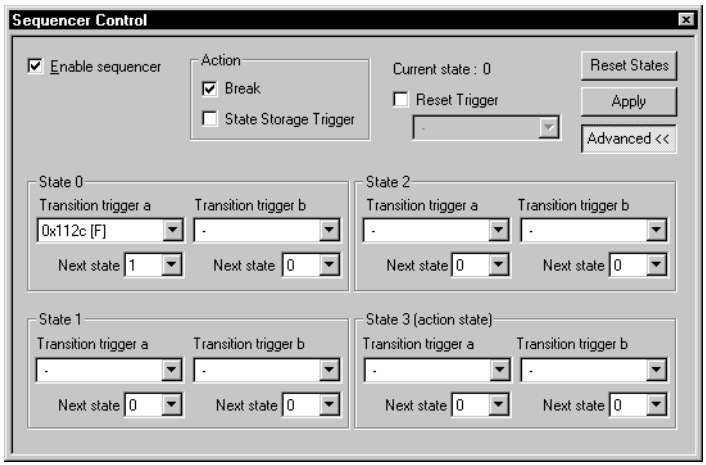


Figure 89: Sequencer Control window (advanced setup)

To enable the sequencer, select the option **Enable Sequencer**. From the eight available hardware breakpoints (0-7) of the device, the breakpoint number 7 will be reserved for state 3.

The **Transition trigger** drop-down lists let you define one breakpoint each, where the breakpoint should act as a transition trigger.

To define an advanced setup, click the **Advanced** button. This will let you define 4 states (0-3) with two transition triggers each (a and b). For each transition trigger, you can define which state should be the next state after the transition.

Use the following options:

State Storage Trigger	Triggers to move the state machine from one state to another. Select a breakpoint from the drop-down list. Note: to do this you must first define the required conditional breakpoints.
Next state	Defines which state should be the next state after the transition. Select one state, out of four, from the drop-down list.

Finally, you must define an action. This option defines the result of the final transition trigger. If you select the option **Break**, the execution will stop. If you select the option **State Storage Trigger**, the state storage module will be triggered.

The **Reset States** button will set the state machine to state 0. **Current state** shows the current state of the state machine.

Stepping

Be aware that stepping might cause some unexpected side-effects.

PROGRAMMING FLASH

Multiple internal machine cycles are required to clear and program the flash memory. When single-stepping over instructions that manipulate the flash, control is given back to C-SPY before these operations are complete. Consequently, C-SPY will update its memory window with erroneous information. A workaround to this behavior is to follow the flash access instruction with a `NOP` instruction, and then step past the `NOP` before reviewing the effects of the flash access instruction.

SINGLE-STEPPING WITH ACTIVE INTERRUPTS

When you single-step with active and enabled interrupts, it can seem as if only the interrupt service routine (ISR) is active. That is, the non-ISR code never appears to execute, and the single-step operation always stops on the first line of the ISR. However, this behavior is correct because the device will always process an active and enabled interrupt.

There is a workaround for this behavior. While within the ISR, disable the `GIE` bit on the stack so that interrupts will be disabled after exiting the ISR. This will permit the non-ISR code to be debugged (but without interrupts). Interrupts can later be re-enabled by setting `GIE` in the status register in the Register window.

On devices with Clock Control, it may be possible to suspend a clock between single steps and delay an interrupt request.

C-SPY FET communication

C-SPY uses the JTAG pins of the device to debug the device. On some MSP430 devices, these JTAG pins are shared with the device port pins. Normally, C-SPY maintains the pins in JTAG mode so that the device can be debugged. During this time the port functionality of the shared pins is not available.

RELEASING JTAG

When you choose **Emulator>Release JTAG on Go**, the JTAG drivers are set to tri-state and the device will be released from JTAG control (the TEST pin is set to GND) when GO is activated. Any active on-chip breakpoints are retained and the shared JTAG port pins revert to their port functions.

At this time, C-SPY has no access to the device and cannot determine if an active breakpoint has been triggered. C-SPY must be manually told to stop the device, at which time the state of the device will be determined (that is, has a breakpoint been reached?).

If you choose **Emulator>Release JTAG on Go**, the JTAG pins will be released if, and only if, there are `N` or fewer active breakpoints.

When making current measurements of the device, ensure that the JTAG control signals are released (**Emulator>Release JTAG on Go**), otherwise the device will be powered by the signals on the JTAG pins and the measurements will be erroneous.

PARALLEL PORT DESIGNATORS

The parallel port designators (LPTx) have the following physical addresses: LPT1: `0x378`, LPT2: `0x278`, LPT3: `0x3BC`. The configuration of the parallel port (ECP, Compatible, Bidirectional, Normal) is not significant; ECP is recommended.

TROUBLESHOOTING

If establishing communication between the C-SPY FET driver and the target system fails, possible solutions to this problem include:

- Restart your host computer.
- Ensure that `R6` on the MSP-FET430X110 and the FET Interface module has a value of 82 ohms. Early units were built using a 330-ohm resistor for `R6`. The FET Interface module can be opened by inserting a thin blade between the case halves, and then carefully twisting the blade to pry the case halves apart.
- Ensure that the correct parallel port has been specified in the options category **FET Debugger** available from the **Project>Options** menu. Check the PC BIOS for the parallel port address (`0x378`, `0x278`, `0x3BC`), and the parallel port configuration (ECP, Compatible, Bidirectional, or Normal).
- Ensure that no other software application has reserved/taken control of the parallel port (for instance, printer drivers, ZIP drive drivers, etc.). Such software can prevent the C-SPY FET driver from accessing the parallel port, and therefore also from communicating with the device.
- Revisions 1.0, 1.1, and 1.2 of the FET Interface module require a hardware modification; a 0.1µF capacitor needs to be installed between U1 pin 1 (signal VCC_MSP) and ground. A convenient (electrically equivalent) installation point for this capacitor is between pins 4 and 5 of U1.

Note: The hardware modification may already have been performed during manufacturing, or your tool might contain an updated version of the FET Interface module.

- Revisions 0.1 and 1.0 of the MSP-TS430PM64 Target Socket module require a hardware modification; the PCB trace connecting pin 6 of the JTAG connector to pin 9 of the MSP430 (signal XOUT) needs to be severed.

Note: The hardware modification may already have been performed during manufacturing, or your tool might contain an updated version of the Target Socket module.

Also note that if the modified Target Socket module is used with the PRGS, Version 1.10 or later of the PRGS software is required.

For revisions 1.0, 1.1, and 1.2 of the FET Interface module, install a 0.1µF capacitor between the indicated points (pins 4 and 5 of U1).

Design considerations for in-circuit programming

This chapter describes the design considerations related to the bootstrap loader, device signals, and external power for in-circuit programming. This chapter also describes how you can adapt your own target hardware to be run with C-SPY.

Bootstrap loader

The JTAG pins provide access to the flash memory of the MSP430Fxxx devices. On some devices, these pins must be shared with the device port pins, and this sharing of pins can complicate a design (or it might simply not be possible to do so). As an alternative to using the JTAG pins, MSP430Fxxx devices contain a program—a *bootstrap loader*—that permits the flash memory to be easily erased and programmed, using a reduced set of signals.

Device signals

The following device signals should be made accessible so that the FET and PRGS (serial programming adapter) tools can be utilized:

RST/NMI, TMS, TCK, TDI, TDO, GND, VCC, and TEST (if present).

Note: Connections to XIN and XOUT are not required, and should not be made. PRGS software Version 1.10 or later must be used.

The BSL tool requires the following device signals: RST/NMI, TCK, GND, VCC, P1.1, P2.2, and TEST (if present).

External power

The PC parallel port is capable of supplying a limited amount of current. Because of the ultra low power requirement of the MSP430, a stand-alone FET Debugger can run on the available current. However, if additional circuitry is added to the tool, this current might not be enough. In this case, external power can be supplied to the tool via the connections provided on the MSP-FET430X110 and the Target Socket modules. Refer to Figure 90, *JTAG signal connection (MSP-FET430X110)* and Figure 91, *JTAG signal connection (MSP-FET430Pxx0)*, respectively, to locate the external power connections.

When an MSP-FET430X110 device is powered from an external supply, an on-board device regulates the external voltage to the level required by the MSP430.

When a Target Socket module is powered from an external supply, the external supply powers the device on the Target Socket module and any user circuitry connected to the Target Socket module, and the FET Interface module continues to be powered from the PC via the parallel port. If the externally supplied voltage differs from that of the FET Interface module, the Target Socket module must be modified so that the externally supplied voltage is routed to the FET Interface module (so that it can adjust its output voltage levels accordingly). For details of the Target Socket module schematic, see the documentation supplied by the chip manufacturer.

Signal connections for in-system programming

With the proper connections, you can use the C-SPY Debugger and the MSP-FET430X110, as well as the MSP-FET430Pxx0 ('P120, 'P140, 'P410, 'P440), to program and debug code on your own target board. In addition, the connections will support the MSP430 Serial Programming Adapter (PRGS), thus providing an easy way to program prototype boards, if desired.

Note: The IAR XLINK Linker can be configured to output objects in `msp430-txt` format for use with the PRGS tool. Choose **Project>Options** and click the **Output** tab in the **Linker** category. Select the option **Other** and then choose **msp430-txt** from the **Output format** drop-down list. The Intel and Motorola formats can also be used.

MSP-FET430X110

Figure 90, *JTAG signal connection (MSP-FET430X110)*, shows the connections between the FET device and the target device required to support in-system programming and debugging using C-SPY. If your target board has its own “local” power supply, such as a battery, do not connect Vcc to pin 2 of the JTAG header. Otherwise, contention might occur between the FET device and your local power supply.

The figure shows a 14-pin header (available from Digi-Key, p/n MHB14K-ND), being used for the connections on your target board. It is recommended that you build a wiring harness from the FET device with a connector which mates to the 14-pin header, and mount the 14-pin header on your target board. This will allow you to unplug your target board from the FET device as well as use the Serial Programming Adapter to program prototype boards, if desired.

The signals required are routed on the FET device to header locations for easy accessibility. Refer to the hardware documentation for more details.

After you make the connections from the FET device to your target board, remove the MSP430 device from the socket on the FET device so that it does not conflict with the MSP430 device in your target board. Now simply use C-SPY as you would normally to program and debug.

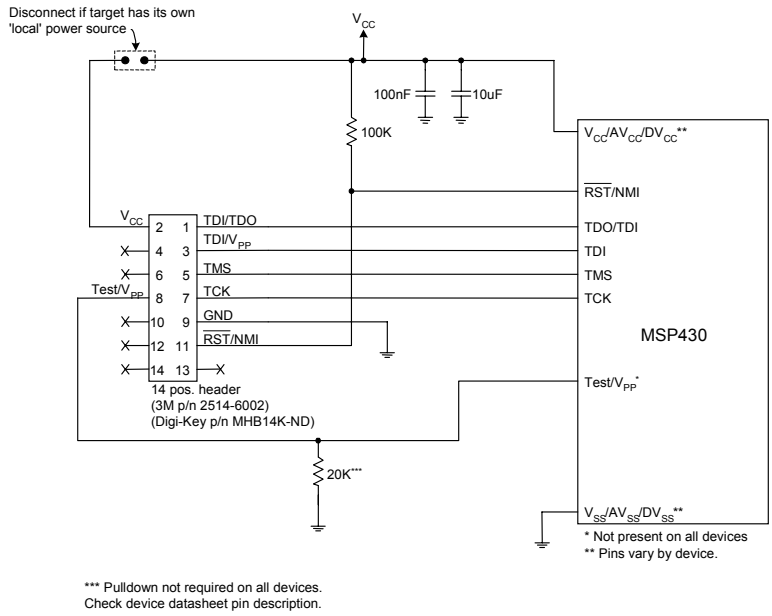


Figure 90: JTAG signal connection (MSP-FET430X110)

Note: No JTAG connection is required to the XOUT pin of the MSP430 device as shown on some schematics.

MSP-FET430PXX0 (‘P120, ‘P140, ‘P410, ‘P440)

Figure 91, *JTAG signal connection (MSP-FET430Pxx0)* shows the connections between the FET Interface module and the target device required to support in-system programming and debugging using C-SPY. The figure shows a 14-pin header (available from Digi-Key, p/n MHB14K-ND) connected to the MSP430. With this header mounted on your target board, the FET Interface module can be plugged directly into your target. Then simply use C-SPY as you would normally to program and debug.

The connections for the FET Interface module and the Serial Programming Adapter (PRGS) are identical with the exception of VCC. Both the FET Interface module and PRGS can supply VCC to your target board (via pin 2). In addition, the FET Interface module has a VCC-sense feature that, if used, requires an alternate connection (pin 4 instead of pin 2). The FET Interface module VCC-sense feature senses the local VCC (present on the target board, i.e. a battery or other “local” power supply) and adjusts its output signals accordingly. The PRGS does not support this feature, but does provide the user the ability to adjust its JTAG signal levels to the VCC level on your target board through the GUI.

If the target board is to be powered by a local VCC, then the connection to pin 4 on the JTAG should be made, and not the connection to pin 2. This utilizes the VCC-sense feature of the FET Interface module and prevents any contention that might occur if the local on-board VCC were connected to the VCC supplied from the FET Interface module or the PRGS. If the VCC-sense feature is not necessary (that is, the target board is to be powered from the FET Interface module or the PRGS) the VCC connection is made to pin 2 on the JTAG header and no connection is made to pin 4.

The figure shows a jumper block in use. The jumper block supports both scenarios of supplying VCC to the target board. If this flexibility is not required, the desired VCC connections can be hard-wired eliminating the jumper block.

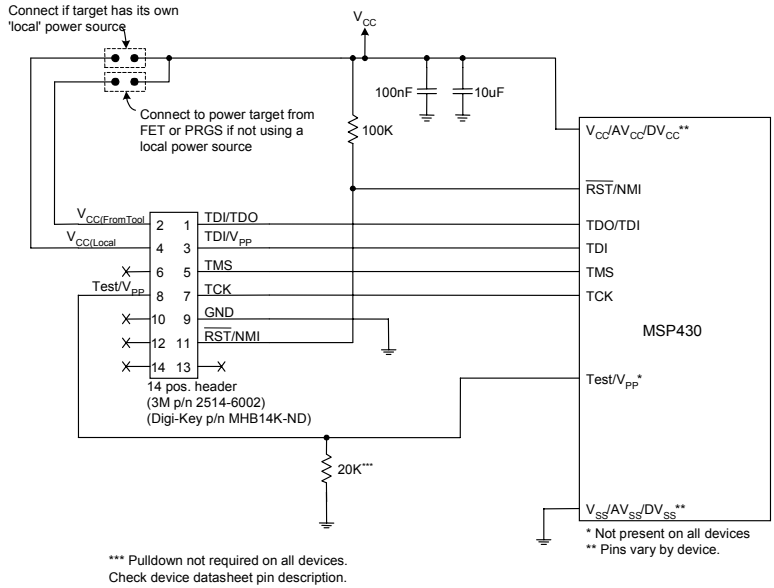


Figure 91: JTAG signal connection (MSP-FET430Pxx0)

Note: No JTAG connection is required to the XOUT pin of the MSP430 as shown on some schematics.

Part 7. Reference information

This part of the MSP430 IAR Embedded Workbench® IDE User Guide contains the following chapters:

- IAR Embedded Workbench® IDE reference
- C-SPY® Debugger reference
- General options
- Compiler options
- Assembler options
- Custom build options
- Build actions options
- Linker options
- Library builder options
- Debugger options
- C-SPY® macros reference.

IAR Embedded Workbench® IDE reference

This chaptersection contains reference information about the windows, menus, menu commands, and the corresponding components that are found in the IAR Embedded Workbench IDE. Information about how to best use the Embedded Workbench for your purposes can be found in parts 3 to 7 in this guide.Information about how to best use the Embedded Workbench for your purposes can be found in the *MSP430 IAR Embedded Workbench® IDE User Guide.*

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

Windows

The available windows are:

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

In addition, a set of C-SPY®-specific windows becomes available when you start the IAR C-SPY Debugger. Reference information about these windows can be found in the chapter *C-SPY® Debugger reference* in this guide.

IAR EMBEDDED WORKBENCH IDE WINDOW

The figure shows the main window of the IAR Embedded Workbench IDE and its different components. The window might look different depending on which plugin modules you are using.

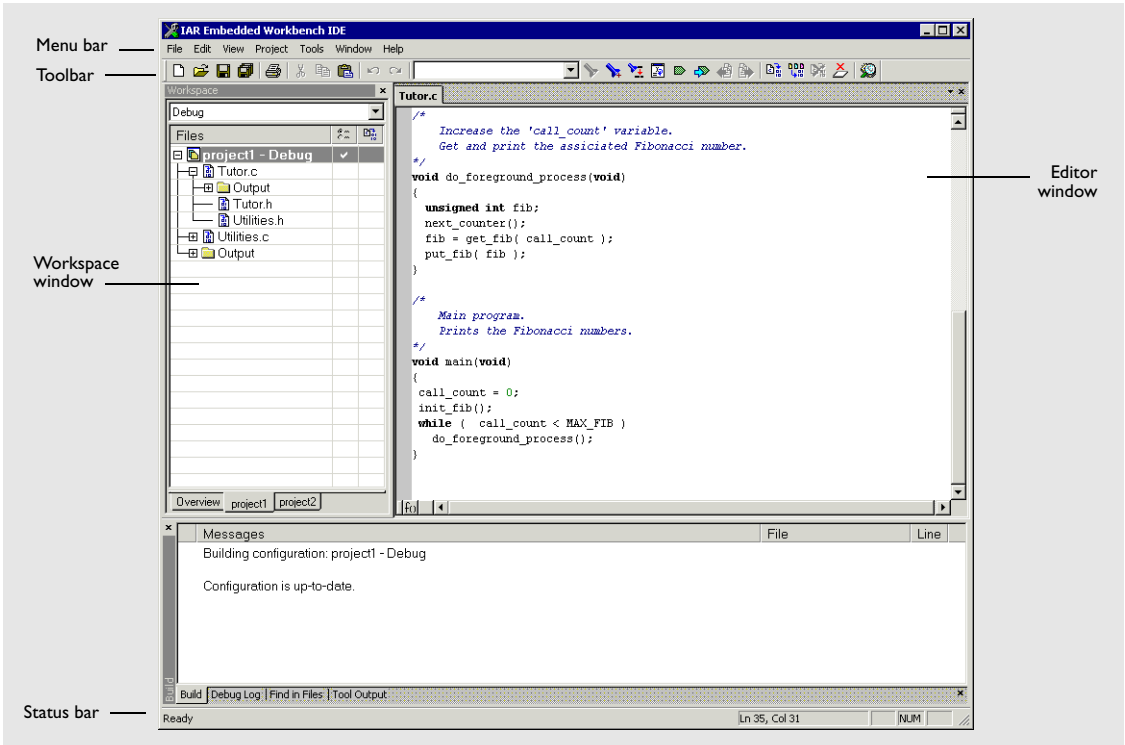


Figure 92: IAR Embedded Workbench IDE window

Each window item is explained in greater detail in the following sections.

Menu bar

Gives access to the IAR Embedded Workbench IDE 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 IDE.
Edit	The Edit menu provides commands for editing and searching in editor windows and for enabling and disabling breakpoints in C-SPY.

Table 46: IAR Embedded Workbench IDE menu bar

Menu	Description
View	Use the commands on the View menu to open windows and decide which toolbars to display.
Project	The Project menu provides commands for adding files to a project, creating groups, and running the IAR Systems 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 IDE.
Window	With the commands on the Window menu you can manipulate the IAR Embedded Workbench IDE windows and change their arrangement on the screen.
Help	The commands on the Help menu provide help about the IAR Embedded Workbench IDE.

Table 46: IAR Embedded Workbench IDE menu bar (Continued)

For reference information for each menu, see *Menus*, page 264.

Toolbar

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

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 dimmed, and you will not be able to click it.

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

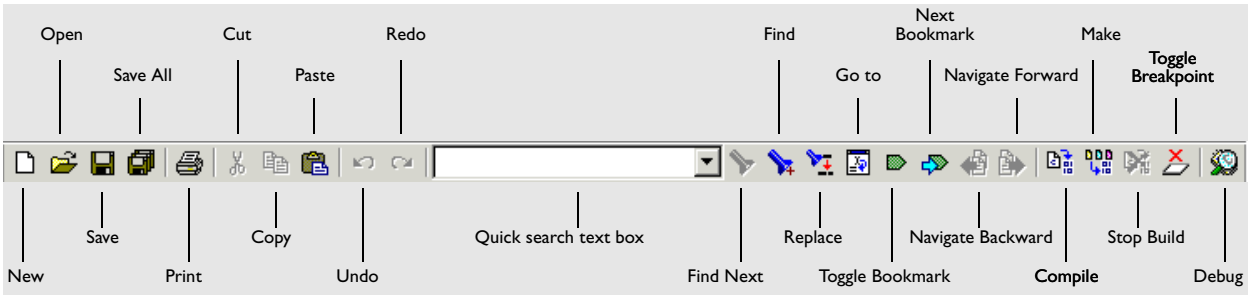


Figure 93: IAR Embedded Workbench IDE toolbar

Note: When you start C-SPY, the **Debug** button will change to a **Make and Debug** button.

Status bar

The Status bar at the bottom of the window—available from the **View** menu—displays the status of the IAR Embedded Workbench IDE, and the state of the modifier keys.

As you are editing, the status bar shows the current line and column number containing the insertion point, and the Caps Lock, Num Lock, and Overwrite status.

Figure 94: IAR Embedded Workbench IDE window status bar

WORKSPACE WINDOW

The Workspace window, available from the **View** menu, shows the name of the current workspace and a tree representation of the projects, groups and files included in the workspace.

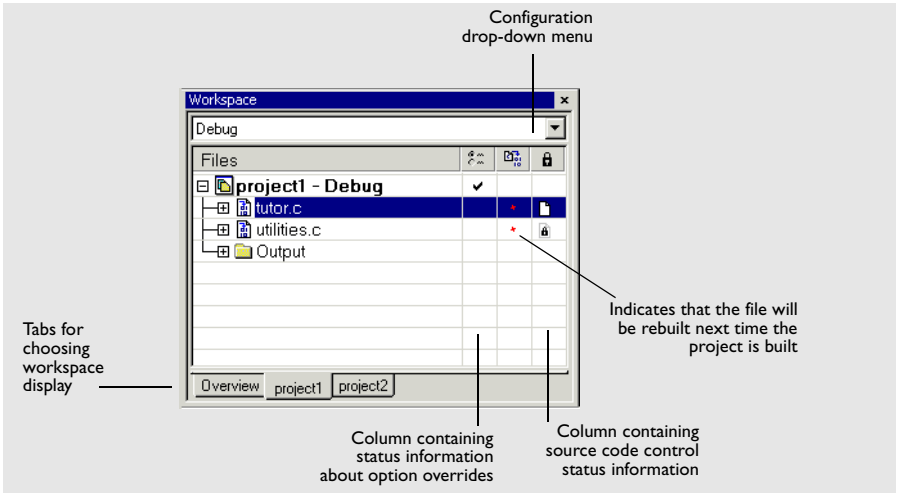


Figure 95: Workspace window

In the drop-down list at the top of the window you can choose a build configuration to display in the window for a specific project.

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

Blank	There are no settings/overrides for this file/group
Black check mark	There are local settings/overrides for this file/group
Red check mark	There are local settings/overrides for this file/group, but they are identical with the inherited settings, which means the overrides are superfluous.

For details about the different source code control icons, see *Source code control states*, page 244.

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

For more information about project management and using the Workspace window, see the chapter *Managing projects* in *Part 3. Project management and building* in this guide.For more information about project management and using the Workspace window, see the *MSP430 IAR Embedded Workbench® IDE User Guide*.

Workspace window context menu

Clicking the right mouse button in the Workspace window displays a context menu which gives you convenient access to several commands.

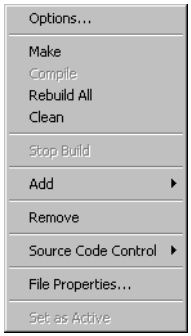


Figure 96: Workspace window context menu

The following commands are available on the context menu:

Menu command	Description
Options	Displays a dialog box where you can set options for each build tool on the selected item in the Workspace window. You can set options on the entire project, on a group of files, or on an individual file.
Make	Brings the current target up to date by compiling, assembling, and linking only the files that have changed since the last build.
Compile	Compiles or assembles the currently active file as appropriate. You can choose the file either by selecting it in the Workspace window, or by selecting the editor window containing the file you want to compile.
Rebuild All	Recompiles and relinks all files in the selected build configuration.
Clean	Deletes intermediate files.
Stop Build	Stops the current build operation.
Add>Add Files	Opens a dialog box where you can add files to the project.
Add>Add "*filename*"	Adds the indicated file to the project. This command is only available if there is an open file in the editor.
Add>Add Group	Opens a dialog box where you can add new groups to the project.
Remove	Removes selected items from the Workspace window.
Source Code Control	Opens a submenu with commands for source code control, see *Source Code Control menu*, page 243.
File Properties	Opens a standard **File Properties** dialog box for the selected file.
Set as Active	Sets the selected project in the overview display to be the active project. It is the active project that will be built when the **Make** command is executed.

Table 47: Workspace window context menu commands

Source Code Control menu

The **Source Code Control** menu is available from the **Project** menu and from the context menu in the Workspace window. This menu contains some of the most commonly used commands of external, third-party source code control systems.

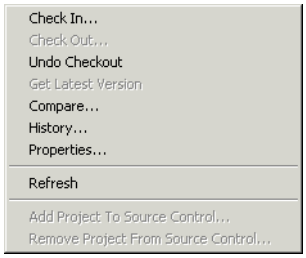


Figure 97: Source Code Control menu

For more information about interacting with an external source code control system, see *Source code control*, page 86the *MSP430 IAR Embedded Workbench® IDE User Guide*.

The following commands are available on the submenu:

Menu command	Description
Check In	Opens the **Check In Files** dialog box where you can check in the selected files; see *Check In Files dialog box*, page 246. Any changes you have made in the files will be stored in the archive. This command is enabled when currently checked-out files are selected in the Workspace window.
Check Out	Checks out the selected file or files. Depending on the SCC system you are using, a dialog box may appear; see *Check Out Files dialog box*, page 247. This means you get a local copy of the file(s), which you can edit. This command is enabled when currently checked-in files are selected in the Workspace window.
Undo Check out	The selected files revert to the latest archived version; the files are no longer checked-out. Any changes you have made to the files will be lost. This command is enabled when currently checked-out files are selected in the Workspace window.
Get Latest Version	Replaces the selected files with the latest archived version.
Compare	Displays—in a SCC-specific window—the differences between the local version and the most recent archived version.

Table 48: Description of source code control commands

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

Table 48: Description of source code control commands (Continued)

Source code control states

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

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

Table 49: Description of source code control states

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

Select Source Code Control Provider dialog box

The **Select Source Code Control Provider** dialog box is displayed if there are several SCC systems from different vendors available. Use this dialog box to choose the SCC system you want to use.

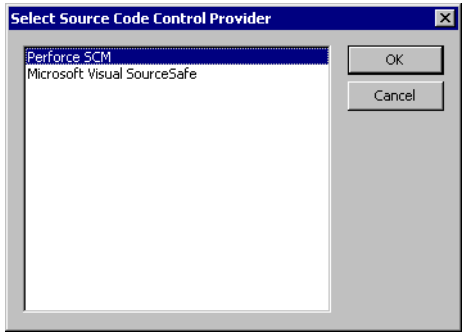


Figure 98: Select Source Code Control Provider dialog box

Check In Files dialog box

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

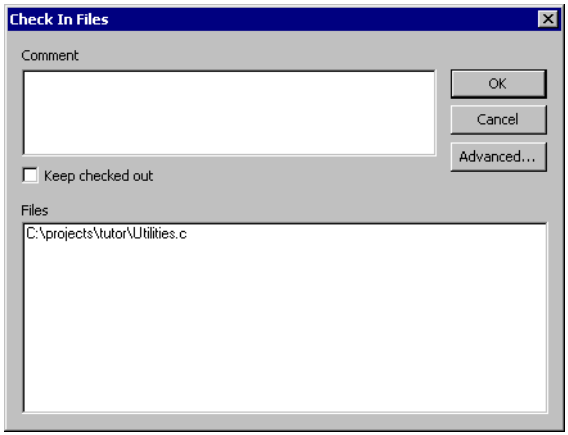


Figure 99: Check In File dialog box

Comment

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

Keep checked out

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

Advanced

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

Files

A list of the files that will be checked in. The list will contain all files that were selected in the Workspace window when this dialog box was opened.

Check Out Files dialog box

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

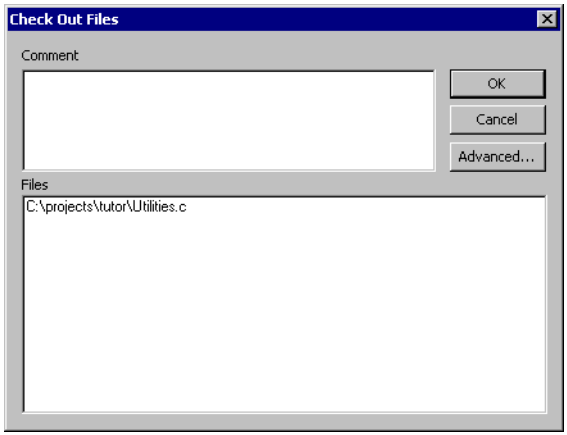


Figure 100: Check Out File dialog box

Comment

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

Advanced

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

Files

A list of files that will be checked out. The list will contain all files that were selected in the Workspace window when this dialog box was opened.

EDITOR WINDOW

Source files are displayed in editor windows. You can have one or several editor windows open at the same time. The editor window is always docked, and its size and position depends on other currently open windows.

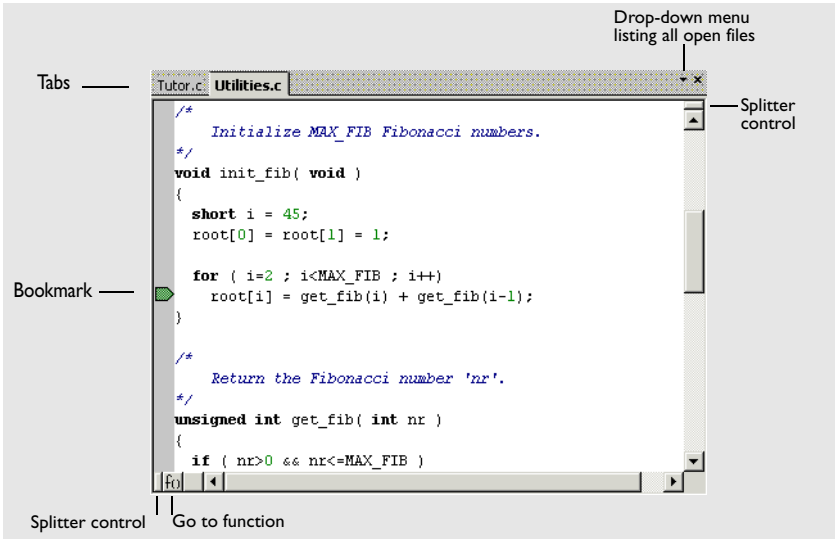


Figure 101: Editor window

The name of the open source file is displayed on the tab. If a file is read-only, a padlock icon is visible at the bottom left corner of the editor window. If a file has been modified after it was last saved, an asterisk appears after the filename on the tab, for example `Utilities.c *`. All open files are available from the drop-down menu in the upper right corner of the editor window.

For information about using the editor, see the chapter *Editing*, page 95.For information about using the editor, see the *MSP430 IAR Embedded Workbench® IDE User Guide*.

Split commands

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

On the **Window** menu you also find commands for opening multiple editor windows, as well as commands for moving files between the different editor windows.

Go to function

With the **Go to function** button in the bottom left-hand corner of the editor window you can display all functions in the C or C++ editor window. You can then choose to go directly to one of them.

Editor window tab context menu

The context menu that appears if you right-click on a tab in the editor window provides access to commands for saving and closing the file.

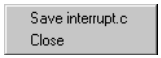


Figure 102: Editor window tab context menu

Editor window context menu

The context menu available in the editor window provides convenient access to several commands.

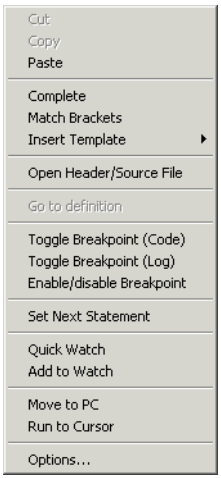


Figure 103: Editor window context menu

Note: The contents of this menu depend on different circumstances, which means it may contain other commands compared to this figure. All commands available are described in the Table 50, *Description of commands on the editor window context menu*.

The following commands are available on the editor window context menu:

Menu command	Description
Cut, Copy, Paste	Standard window commands.
Complete	Attempts to complete the word you have begun to type, basing the guess on the contents of the rest of the editor document.
Match Brackets	Selects all text between the brackets immediately surrounding the insertion point, increases the selection to the next hierarchic pair of brackets, or beeps if there is no higher bracket hierarchy.
Insert Template	Displays a list in the editor window from which you can choose a code template to be inserted at the location of the insertion point. If the code template you choose requires any field input, the **Template** dialog box appears; for information about this dialog box, see *Template dialog box*, page 274. For information about using code templates, see *Using and adding code templates*, page 99the *MSP430 IAR Embedded Workbench® IDE User Guide*.
Open "*header*.h"	Opens the header file "*header*.h" in an editor window. This menu command is only available if the insertion point is located on an `#include` line when you open the context menu.
Open Header/Source File	Jumps from the current file to the corresponding header or source file. If the destination file is not open when performing the command, the file will first be opened. This menu command is only available if the insertion point is located on any line except an `#include` line when you open the context menu. This command is also available from the **File>Open** menu.
Go to definition	Shows the declaration of the symbol where the insertion point is placed.
Check In Check Out Undo Checkout	Commands for source code control; for more details, see *Source Code Control menu*, page 243. These menu commands are only available if the current source file in the editor window is SCC-controlled. The file must also be a member of the current project.
Toggle Breakpoint (Code)	Toggles a code breakpoint at the statement or instruction containing or close to the cursor in the source window. For information about code breakpoints, see *Code breakpoints dialog box*, page 256.
Toggle Breakpoint (Log)	HistoryToggles a log breakpoint at the statement or instruction containing or close to the cursor in the source window. For information about log breakpoints, see *Log breakpoints dialog box*, page 258.
Enable/disable Breakpoint	Toggles a breakpoint between being disabled, but not actually removed—making it available for future use—and being enabled again.

Table 50: Description of commands on the editor window context menu

Menu command	Description
Set Next Statement	Sets the PC directly to the selected statement or instruction without executing any code. Use this menu command with care. This menu command is only available when you are using the debugger.
Quick Watch	Opens the Quick Watch window, see *Quick Watch window*, page 325. This menu command is only available when you are using the debugger.
Add to Watch	Adds the selected symbol to the Watch window. This menu command is only available when you are using the debugger.
Move to PC	Moves the insertion point to the current PC position in the editor window. This menu command is only available when you are using the debugger.
Run to Cursor	Executes from the current statement or instruction up to a selected statement or instruction. This menu command is only available when you are using the debugger.
Options	HistoryDisplays the **IDE Options** dialog box, see *Tools menu*, page 286.

Table 50: Description of commands on the editor window context menu (Continued)

Source file paths

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

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

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

Table 51: Editor keyboard commands for insertion point navigation

To move the insertion point	Press
To the end of the line	End
To the first line in the file	Ctrl+Home
To the last line in the file	Ctrl+End

Table 51: Editor keyboard commands for insertion point navigation (Continued)

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

Table 52: Editor keyboard commands for scrolling

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

Table 53: Editor keyboard commands for selecting text

SOURCE BROWSER WINDOW

The Source Browser window—available from the **View** menu—displays an hierarchical view in alphabetical order of all symbols defined in the active build configuration.

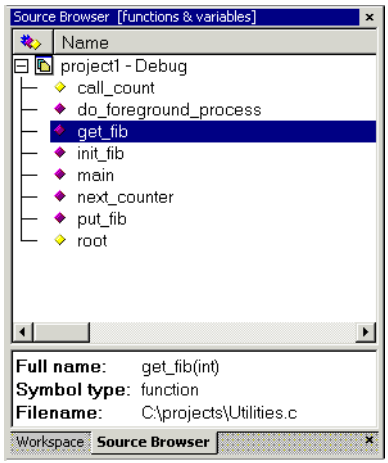


Figure 104: Source Browser window

The window consists of two separate panes. The top pane displays the names of global symbols and functions defined in the project.

Each row is prefixed with an icon, which corresponds to the *Symbol type* classification, see Table 54, *Information in Source Browser window*. By clicking in the window header, you can sort the symbols either by name or by symbol type.

In the top pane you can also access a context menu; see *Source Browser window context menu*, page 254.

For a symbol selected in the top pane, the bottom pane displays the following information:

Type of information	Description
Full name	Displays the unique name of each element, for instance `classname::membername`.
Symbol type	Displays the symbol type for each element: enumeration, enumeration constant, class, typedef, union, macro, field or variable, function, template function, template class, and configuration.
Filename	Specifies the path to the file in which the element is defined.

Table 54: Information in Source Browser window

For further details about how to use the Source Browser window, see *Displaying browse information*, page 85.For further details about how to use the Source Browser window, see the *MSP430 IAR Embedded Workbench® IDE User Guide*.

Source Browser window context menu

Right-clicking in the Source Browser window displays a context menu with convenient access to several commands.

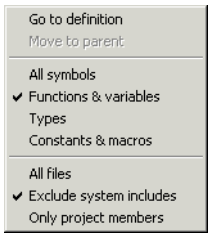


Figure 105: Source Browser window context menu

The following commands are available on the context menu:

Menu command	Description
Go to Source	The editor window will display the definition of the selected item.
Move to parent	If the selected element is a member of a class, struct, union, enumeration, or namespace, this menu command can be used for moving to its enclosing element.
All symbols	Type filter; all global symbols and functions defined in the project will be displayed.
Functions & variables	Type filter; all functions and variables defined in the project will be displayed.
Types	Type filter; all types such as structures and classes defined in the project will be displayed.
Constants & macros	Type filter; all constants and macros defined in the project will be displayed.
All files	File filter; symbols from all files that you have explicitly added to your project and all files included by them will be displayed.
Exclude system includes	File filter; symbols from all files that you have explicitly added to your project and all files included by them will be displayed, except the include files in the IAR Embedded Workbench installation directory.

Table 55: Source Browser window context menu commands

Menu command	Description
Only project members	File filter; symbols from all files that you have explicitly added to your project will be displayed, but no include files.

Table 55: Source Browser window context menu commands (Continued)

BREAKPOINTS WINDOW

The Breakpoints window—available from the **View** menu—lists all breakpoints. From the window you can conveniently monitor, enable, and disable breakpoints; you can also define new breakpoints and modify existing breakpoints.

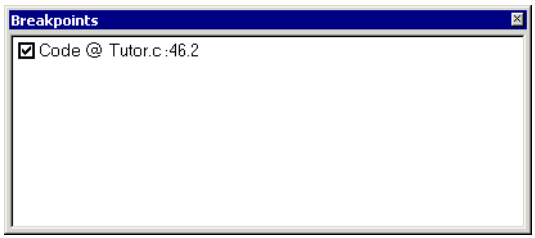


Figure 106: Breakpoints window

All breakpoints you define are displayed in the Breakpoints window.

For more information about the breakpoint system and how to set breakpoints, see the chapter *Using breakpoints* in *Part 4. Debugging*.For more information about the breakpoint system and how to set breakpoints, see the *MSP430 IAR Embedded Workbench® IDE User Guide*.

Breakpoints window context menu

Right-clicking in the Breakpoints window displays a context menu with several commands.

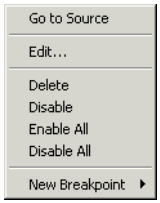


Figure 107: Breakpoints window context menu

The following commands are available on the context menu:

Menu command	Description
Go to Source	Moves the insertion point to the location of the breakpoint, if the breakpoint has a source location. Double-click a breakpoint in the Breakpoints window to perform the same command.
Edit	Opens the **Edit Breakpoint** dialog box for the selected breakpoint.
Delete	Deletes the selected breakpoint. Press the Delete key to perform the same command.
Enable	Enables the selected breakpoint. The check box at the beginning of the line will be selected. You can also perform the command by manually selecting the check box. This command is only available if the selected breakpoint is disabled.
Disable	Disables the selected breakpoint. The check box at the beginning of the line will be cleared. You can also perform this command by manually deselecting the check box.This command is only available if the selected breakpoint is enabled.
Enable All	Enables all defined breakpoints.
Disable All	Disables all defined breakpoints.
New Breakpoint	Displays a submenu where you can open the **New Breakpoint** dialog box for the available breakpoint types. All breakpoints you define using the **New Breakpoint** dialog box are preserved between debug sessions. In addition to code and log breakpoints—see *Code breakpoints dialog box*, page 256 and —other types of breakpoints might be available depending on the C-SPY driver you are using. For information about driver-specific breakpoint types, see the driver-specific debugger documentation.

Table 56: Breakpoints window context menu commands

Code breakpoints dialog box

Code breakpoints are triggered when an instruction is fetched from the specified location. If you have set the breakpoint on a specific machine instruction, the breakpoint will be triggered and the execution will stop, before the instruction is executed.

To set a code breakpoint, right-click in the Breakpoints window and choose **New Breakpoint>Log** on the context menu. To modify an existing breakpoint, select it in the Breakpoints window and choose **Edit** on the context menu.

The **Code** breakpoints dialog box appears.

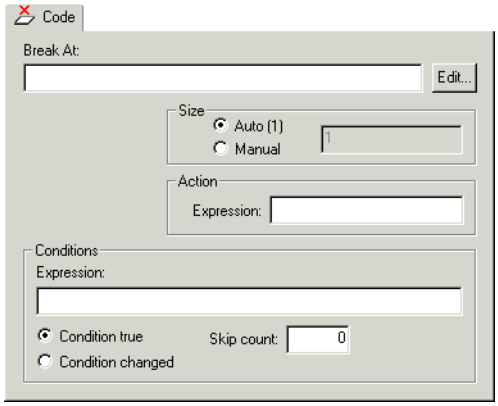


Figure 108: Code breakpoints page

Break At

Specify the location of the breakpoint in the **Break At** text box. Alternatively, click the **Edit** browse button to open the **Enter Location** dialog box; see *Enter Location dialog box*, page 260.

Size

Optionally, you can specify a size—in practice, a *range*—of locations. Each fetch access to the specified memory range will trigger the breakpoint. There are two different ways the size can be specified:

- **Auto**, the size will be set automatically, typically to 1
- **Manual**, you specify the size of the breakpoint range manually in the **Size** text box.

Action

You can optionally connect an action to a breakpoint. Specify an expression, for instance a C-SPY macro function, which is evaluated when the breakpoint is triggered and the condition is true.

Conditions

You can specify simple and complex conditions.

Conditions	Description
Expression	A valid expression conforming to the C-SPY expression syntax.
Condition true	The breakpoint is triggered if the value of the expression is true.
Condition changed	The breakpoint is triggered if the value of the expression has changed since it was last evaluated.
Skip count	The number of times that the breakpoint must be fulfilled before a break occurs (integer).

Table 57: Breakpoint conditions

Log breakpoints dialog box

Log breakpoints are triggered when an instruction is fetched from the specified location. If you have set the breakpoint on a specific machine instruction, the breakpoint will be triggered and the execution will temporarily halt and print the specified message in the C-SPY Debug Log window. This is a convenient way to add trace printouts during the execution of your application, without having to add any code to the application source code.

To set a log breakpoint, right-click in the Breakpoints window and choose **New Breakpoint>Log** on the context menu. To modify an existing breakpoint, select it in the Breakpoints window and choose **Edit** on the context menu.

The **Log** breakpoints dialog box appears.

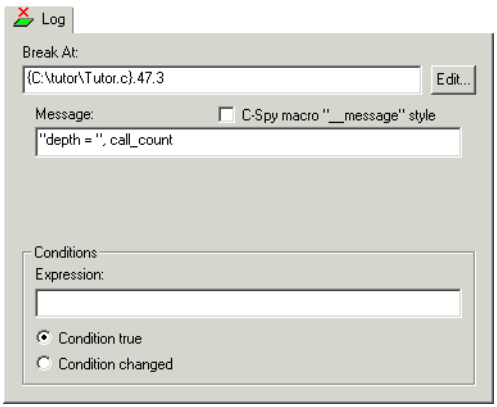


Figure 109: Log breakpoints page

The quickest—and typical—way to set a log breakpoint is by choosing **Toggle Breakpoint (Log)** from the context many available by right-clicking in either the editor or the Disassembly window. For more information about how to set breakpoints, see *Defining breakpoints*, page 129.

Break At

Specify the location of the breakpoint in the **Break At** text box. Alternatively, click the **Edit** button to open the **Enter Location** dialog box; see *Enter Location dialog box*, page 260.

Message

Specify the message you want to be displayed in the C-SPY Debug Log window. The message can either be plain text, or—if you also select the option **C-SPY macro "__message" style**—a comma-separated list of arguments.

C-SPY macro "__message" style

Select this option to make a comma-separated list of arguments specified in the Message text box be treated exactly as the arguments to the C-SPY macro language statement `__message`, see *Formatted output*, page 400.

Conditions

You can specify simple and complex conditions.

Conditions	Description
Expression	A valid expression conforming to the C-SPY expression syntax.
Condition true	The breakpoint is triggered if the value of the expression is true.
Condition changed	The breakpoint is triggered if the value of the expression has changed since it was last evaluated.

Table 58: Log breakpoint conditions

Enter Location dialog box

Use the **Enter Location** dialog box—available from a breakpoints dialog box—to specify the location of the breakpoint.

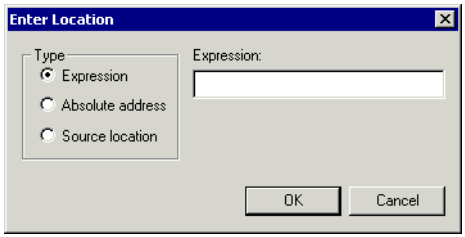


Figure 110: Enter Location dialog box

You can choose between these locations and their possible settings:

Location type	Description/Examples
Expression	Any expression that evaluates to a valid address, such as a function or variable name. Code breakpoints are set on functions and data breakpoints are set on variable names. 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`.
Absolute Address	An absolute location on the form *`zone`*`:`*`hexaddress`* or simply *`hexaddress`*. Zone specifies in which memory the address belongs. For example `Memory:0x42`. If you enter a combination of a zone and address that is not valid, C-SPY will indicate the mismatch.
Source Location	A location in the C source code using the syntax: `{`*`file path`*`}.`*`row`*`.`*`column`*`.` *File* specifies the filename and full path. *Row* specifies the row in which you want the breakpoint. *Column* specifies the column in which you want the breakpoint. Note that the Source Location type is usually meaningful only for code breakpoints. For example, `{C:\`*`my_projects`*`\Utilities.c}.22.3` sets a breakpoint on the third character position on line 22 in the source file `Utilities.c`.

Table 59: Location types

BUILD WINDOW

The Build window—available by choosing **View>Messages**—displays the messages generated when building a build configuration. When opened, this window is by default grouped together with the other message windows, see *Windows*, page 237.

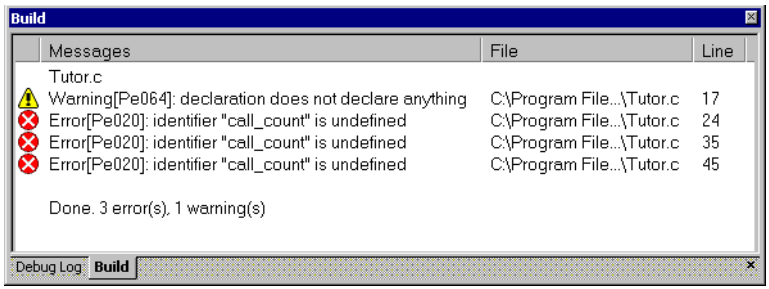


Figure 111: Build window (message window)

Double-clicking a message in the Build window opens the appropriate file for editing, with the insertion point at the correct position.

Right-clicking in the Build window displays a context menu which allows you to copy, select, and clear the contents of the window.

Figure 112: Build window context menu

The **Options** command opens the **Messages** page of the **IDE options** dialog box. On this page you can set options related to messages; see *Messages page*, page 290.

FIND IN FILES WINDOW

The Find in Files window—available by choosing **View>Messages**—displays the output from the **Edit>Find and Replace>Find in Files** command. When opened, this window is by default grouped together with the other message windows, see *Windows*, page 237.

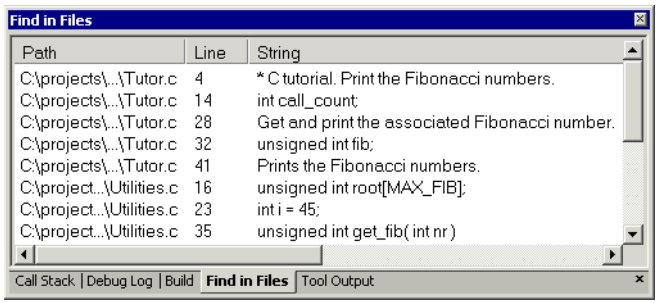


Figure 113: Find in Files window (message window)

Double-clicking an entry in the page opens the appropriate file with the insertion point positioned at the correct location.

Right-clicking in the Find in Files window displays a context menu which allows you to copy, select, and clear the contents of the window.

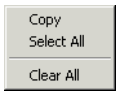


Figure 114: Find in Files window context menu

TOOL OUTPUT WINDOW

The Tool Output window—available by choosing **View>Messages**—displays any messages output by user-defined tools in the Tools menu, provided that you have selected the option **Redirect to Output Window** in the **Configure Tools** dialog box; see *Configure Tools dialog box*, page 303. When opened, this window is by default grouped together with the other message windows, see *Windows*, page 237.

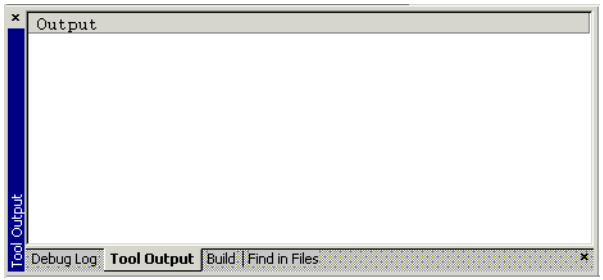


Figure 115: Tool Output window (message window)

Right-clicking in the Tool Output window displays a context menu which allows you to copy, select, and clear the contents of the window.

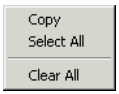


Figure 116: Tool Output window context menu

DEBUG LOG WINDOW

The Debug Log window—available by choosing **View>Messages**—displays debugger output, such as diagnostic messages and trace information. This output is only available when the C-SPY Debugger is running. When opened, this window is by default grouped together with the other message windows, see *Windows*, page 237.

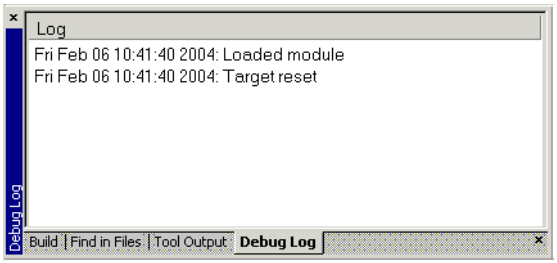


Figure 117: Debug Log window (message window)

Right-clicking in the Tool Output window displays a context menu which allows you to copy, select, and clear the contents of the window.

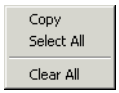


Figure 118: Debug Log window context menu

Menus

The following menus are available in the IAR Embedded Workbench IDE:

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

In addition, a set of C-SPY-specific menus become available when you start the IAR C-SPY Debugger. Reference information about these menus can be found in the chapter *C-SPY® Debugger reference*, page 313.

FILE MENU

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

The menu also includes a numbered list of the most recently opened files and workspaces to allow you to open one by selecting its name from the menu.

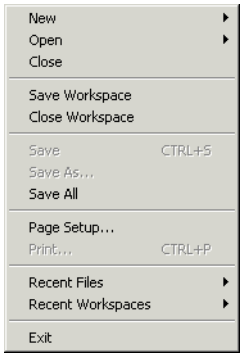


Figure 119: File menu

The following commands are available on the **File** menu:

	Menu command	Shortcut	Description
	New	CTRL+N	Displays a submenu with commands for creating a new workspace, or a new text file.
	Open>File	CTRL+O	Displays a submenu from which you can select a text file to open.
	Open> Workspace		Displays a submenu from which you can select a workspace file to open. Before a new workspace is opened you will be prompted to save and close any currently open workspaces.
	Open> Header/Source File	CTRL+ SHIFT+H	Opens the header file or source file that corresponds to the current file, and jumps from the current file to the newly opened file. This command is also available from the context menu available from the editor window.
	Close		Closes the active window. You will be given the opportunity to save any files that have been modified before closing.

Table 60: File menu commands

	Menu command	Shortcut	Description
	Open Workspace		Displays a dialog box where you can open a workspace file. You will be given the opportunity to save and close any currently open workspace file that has been modified before opening a new workspace.
	Save Workspace		Saves the current workspace file.
	Close Workspace		Closes the current workspace file.
	Save	CTRL+S	Saves the current text file or workspace file.
	Save As		Displays a dialog box where you can save the current file with a new name.
	Save All		Saves all open text documents and workspace files.
	Page Setup		Displays a dialog box where you can set printer options.
	Print	CTRL+P	Displays a dialog box where you can print a text document.
	Recent Files		Displays a submenu where you can quickly open the most recently opened text documents.
	Recent Workspaces		Displays a submenu where you can quickly open the most recently opened workspace files.
	Exit		Exits from the IAR Embedded Workbench IDE. You will be asked whether to save any changes to text windows before closing them. Changes to the project are saved automatically.

Table 60: File menu commands (Continued)

EDIT MENU

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

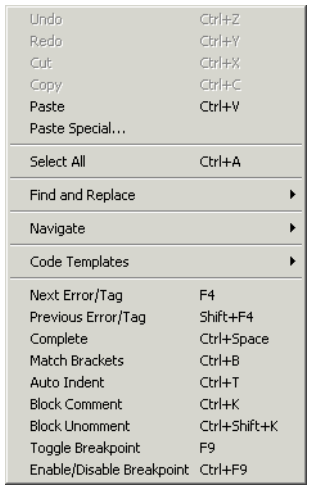


Figure 120: Edit menu

	Menu command	Shortcut	Description
	Undo	CTRL+Z	Undoes the last edit made to the current editor window.
	Redo	CTRL+Y	Redoes the last **Undo** in the current editor window. You can undo and redo an unlimited number of edits independently in each editor window.
	Cut	CTRL+X	The standard Windows command for cutting text in editor windows and text boxes.
	Copy	CTRL+C	The standard Windows command for copying text in editor windows and text boxes.
	Paste	CTRL+V	The standard Windows command for pasting text in editor windows and text boxes.
	Paste Special		Provides you with a choice of the most recent contents of the clipboard to choose from when pasting in editor documents.
	Select All	CTRL+A	Selects all text in the active editor window.

Table 61: Edit menu commands

	Menu command	Shortcut	Description
	Find and Replace>Find	CTRL+F	Displays the **Find** dialog box where you can search for text within the current editor window. Note that if the insertion point is located in the Memory window when you choose the **Find** command, the dialog box will contain a different set of options than it would otherwise do. If the insertion point is located in the Trace window when you choose the **Find** command, the **Find in Trace** dialog box is opened; the contents of this dialog box depend on the C-SPY driver you are using, see the driver documentation for more information.
	Find and Replace> Find Next	F3	Finds the next occurrence of the specified string.
	Find and Replace> Replace	CTRL+H	Displays a dialog box where you can search for a specified string and replace each occurrence with another string. Note that if the insertion point is located in the Memory window when you choose the **Replace** command, the dialog box will contain a different set of options than it would otherwise do.
	Find and Replace> Find in Files		Displays a dialog box where you can search for a specified string in multiple text files; see *Find in Files dialog box*, page 271.
	Find and Replace> Incremental Search	CTRL+I	Displays a dialog box where you can gradually fine-tune or expand the search by continuously changing the search string.
	Navigate>Go To	CTRL+G	Displays a dialog box where you can move the insertion point to a specified line and column in the current editor window.
	Navigate> Toggle Bookmark	CTRL+F2	Toggles a bookmark at the line where the insertion point is located in the active editor window.
	Navigate> Go to Bookmark	F2	Moves the insertion point to the next bookmark that has been defined with the Toggle Bookmark command.
	Navigate> Navigate Backward	ALT+Left arrow	Navigates backward in the insertion point history. The current position of the insertion point is added to the history by actions like **Go to definition** and clicking on a result from the **Find in Files** command.
	Navigate> Navigate Forward	ALT+Right arrow	Navigates forward in the insertion point history. The current position of the insertion point is added to the history by actions like **Go to definition** and clicking on a result from the **Find in Files** command.

Table 61: Edit menu commands (Continued)

Menu command	Shortcut	Description
Code Templates> Insert Template	CTRL+ SHIFT+ SPACE	Displays a list in the editor window from which you can choose a code template to be inserted at the location of the insertion point. If the code template you choose requires any field input, the **Template** dialog box appears; for information about this dialog box, see *Template dialog box*, page 274. For information about using code templates, see *Using and adding code templates*, page 99the *MSP430 IAR Embedded Workbench® IDE User Guide*.
Code Templates> Edit Templates		Opens the current code template file, where you can modify existing code templates and add your own code templates. For information about using code templates, see *Using and adding code templates*, page 99the *MSP430 IAR Embedded Workbench® IDE User Guide*.
Next Error/Tag	F4	If there is a list of error messages or the results from a **Find in Files** search in the Messages window, this command will display the next item from that list in the editor window.
Previous Error/Tag	SHIFT+F4	If there is a list of error messages or the results from a **Find in Files** search in the Messages window, this command will display the previous item from that list in the editor window.
Complete	CTRL+ SPACE	Attempts to complete the word you have begun to type, basing the guess on the contents of the rest of the editor document.
Auto Indent	CTRL+T	Indents one or several lines you have selected in a C/C++ source file. To configure the indentation, see *Configure Auto Indent dialog box*, page 292.
Match Brackets		Selects all text between the brackets immediately surrounding the insertion point, increases the selection to the next hierarchic pair of brackets, or beeps if there is no higher bracket hierarchy.
Block Comment	CTRL+K	HistoryPlaces the C++ comment character sequence // at the beginning of the selected lines.
Block Uncomment	CTRL+K	HistoryRemoves the C++ comment character sequence // from the beginning of the selected lines.

Table 61: Edit menu commands (Continued)

Menu command	Shortcut	Description
Toggle Breakpoint	F9	Toggles a breakpoint at the statement or instruction that contains or is located near the cursor in the source window. This command is also available as an icon button in the debug bar.
Enable/Disable Breakpoint	CTRL+F9	Toggles a breakpoint between being disabled, but not actually removed—making it available for future use—and being enabled again.

Table 61: Edit menu commands (Continued)

Find dialog box

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

Option	Description
Find What	Selects the text to search for.
Match Whole Word Only	Searches the specified text only if it occurs as a separate word. Otherwise specifying `int` will also find `print`, `sprintf` etc. This option is not available when you perform the search in the Memory window.
Match Case	Searches only occurrences that exactly match the case of the specified text. Otherwise specifying `int` will also find `INT` and `Int`. This option is not available when you perform the search in the Memory window.
Direction	Specifies the direction of the search. Choose between the options **Up** and **Down**.
Search as Hex	Searches for the specified hexadecimal value. This option is only available when you perform the search in the Memory window.
Find Next	Searches the next occurrence of the selected text.
Stop	Stops an ongoing search. This function button is only available during a search.

Table 62: Find dialog box options

Replace dialog box

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

Option	Description
Find What	Selects the text to search for.

Table 63: Replace dialog box options

Option	Description
Replace With	Selects the text to replace each found occurrence in the Replace With box.
Match Whole Word Only	Searches the specified text only if it occurs as a separate word. Otherwise `int` will also find `print`, `sprintf` etc. This checkbox is not available when you perform the search in the Memory window.
Match Case	Searches only occurrences that exactly match the case of the specified text. Otherwise specifying `int` will also find `INT` and `Int`. This checkbox is not available when you perform the search in the Memory window.
Search as Hex	Searches for the specified hexadecimal value. This checkbox is only available when you perform the search in the Memory window.
Find Next	Searches the next occurrence of the text you have specified.
Replace	Replaces the searched text with the specified text.
Replace All	Replaces all occurrences of the searched text in the current editor window.

Table 63: Replace dialog box options (Continued)

Find in Files dialog box

Use the **Find in Files** dialog box—available from the **Edit** menu—to search for a string in files.

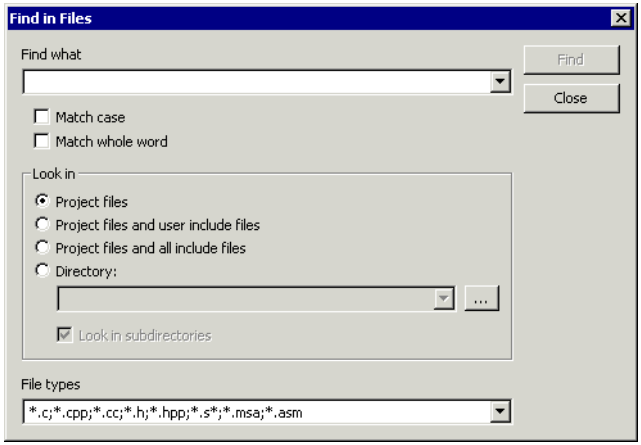


Figure 121: Find in Files dialog box

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

In the **Find in Files** dialog box, you specify the search criteria with the following settings.

Find what

A text field in which you type the string you want to search for. There are two options for fine-tuning the search:

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

Look in

The options in the **Look in** area lets you specify which files you want to search in for a specified string. Choose between:

Project files	The search will be performed in all files that you have explicitly added to your project.
Project files and user include files	The search will be performed in all files that you have explicitly added to your project and all files included by them, except the include files in the IAR Embedded Workbench installation directory.
Project files and all include files	The search will be performed in all project files that you have explicitly added to your project and all files included by them.
Directory	The search will be performed in the directory that you specify. Recent search locations are saved in the drop-down list. Locate the directory using the browse button.
Look in subdirectories	The search will be performed in the directory that you have specified and all its subdirectories.

File types

This is a filter for choosing which type of files to search; the filter applies to all options in the **Look in** area. Choose the appropriate filter from the drop-down list. HistoryNote that the **File types** text field is editable, which means that you can add your own filters. Use the `*` character to indicate zero or more unknown characters of the filters, and the `?` character to indicate one unknown character.

Stop

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

Incremental Search dialog box

The **Incremental Search** dialog box—available from the **Edit** menu—lets you gradually fine-tune or expand the search string.

Figure 122: Incremental Search dialog box

Find What

Type the string to search for. The search will be performed from the location of the insertion point—the *start point*. Gradually incrementing the search string will gradually expand the search criteria. Backspace will remove a character from the search string; the search will be performed on the remaining string and will start from the start point.

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

Match Case

Use this option to find only occurrences that exactly match the case of the specified text. Otherwise searching for `int` will also find `INT` and `Int`.

Function buttons

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

Table 64: Incremental Search function buttons

Template dialog box

Use the **Template** dialog box to specify any field input that is required by the source code template you insert. This dialog box appears when you insert a code template that requires any field input.

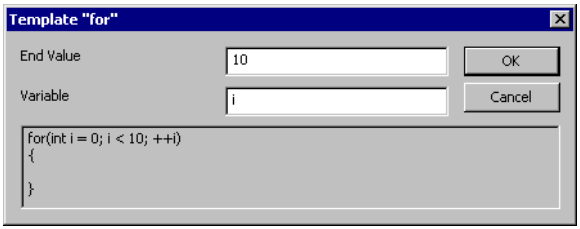


Figure 123: Template dialog box

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

The contents of this dialog box match the code template. In other words, which fields that appear depends on how the code template is defined.

At the bottom of the dialog box, the code that would result from the code template is displayed.

For more information about using code templates, see *Using and adding code templates*, page 99the *MSP430 IAR Embedded Workbench® IDE User Guide*.

VIEW MENU

With the commands on the **View** menu you can choose what to display in the IAR Embedded Workbench IDE. During a debug session you can also open debugger-specific windows from the **View** menu.

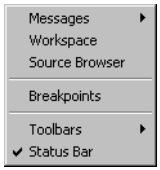


Figure 124: View menu

Menu command	Description
Messages	Opens a submenu which gives access to the message windows—Build, Find in Files, Tool Output, Debug Log—that display messages and text output from the IAR Embedded Workbench commands. If the window you choose from the menu is already open, it becomes the active window.
Workspace	Opens the current Workspace window.
Source Browser	Opens the Source Browser window.
Breakpoints	Opens the Breakpoints window.
Toolbars	The options **Main** and **Debug** toggle the two toolbars on and off.
Status bar	Toggles the status bar on and off.

Table 65: View menu commands

Menu command	Description
Debugger windows	During a debugging session, the different debugging windows are also available from the **View** menu: Disassembly window Memory window Register window Watch window Locals window Auto window Live Watch window Quick Watch window Call Stack window Terminal I/O window Code Coverage window Profiling window Stack window LCD window For descriptions of these windows, see *C-SPY windows*, page 313.

Table 65: View menu commands (Continued)

PROJECT MENU

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

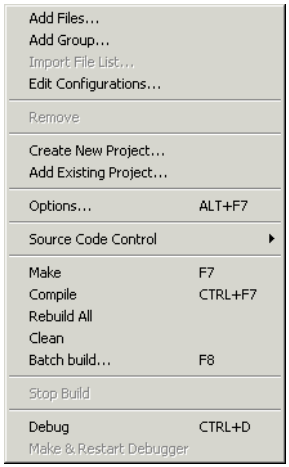


Figure 125: Project menu

Menu Command	Description
Add Files	Displays a dialog box that where you can select which files to include to the current project.
Add Group	Displays a dialog box where you can create a new group. The Group Name text box specifies the name of the new group. The Add to Target list selects the targets to which the new group should be added. By default the group is added to all targets.
Import File List	Displays a standard **Open** dialog box where you can import information about files and groups from projects created using another IAR tool chain. To import information from project files which have one of the older filename extensions `pew` or `prj` you must first have exported the information using the context menu command **Export File List** available in your own IAR Embedded Workbench.
Edit Configurations	Displays the **Configurations for project** dialog box, where you can define new or remove existing build configurations.

Table 66: Project menu commands

Menu Command	Description
Remove	In the Workspace window, removes the selected item from the workspace.
Create New Project	Displays a dialog box where you can create a new project and add it to the workspace.
Add Existing Project	Displays a dialog box where you can add an existing project to the workspace.
Options	Displays the **Options for node** dialog box, where you can set options for the build tools on the selected item in the Workspace window. You can set options on the entire project, on a group of files, or on an individual file.
Source Code Control	Opens a submenu with commands for source code control, see *Source Code Control menu*, page 243.
 Make	Brings the current build configuration up to date by compiling, assembling, and linking only the files that have changed since the last build.
 Compile	Compiles or assembles the currently selected file, files, or group. One or more files can be selected in the Workspace window—all files in the same project, but not necessarily in the same group. You can also select the editor window containing the file you want to compile. The **Compile** command is only enabled if *every* file in the selection is individually suitable for the command. You can also select a *group*, in which case the command is applied to each file in the group (including inside nested groups) that can be compiled, even if the group contains files that cannot be compiled, such as header files.
Rebuild All	Rebuilds and relinks all files in the current target.
Clean	Removes any intermediate files.
Batch Build	Displays a dialog box where you can configure named batch build configurations, and build a named batch.
Stop Build	Stops the current build operation.
 Debug	Starts the IAR C-SPY Debugger so that you can debug the project object file. If necessary, a make will be performed before running C-SPY to ensure the project is up to date. Depending on your IAR product installation, you can choose which debugger drive to use by selecting the appropriate C-SPY driver on the C-SPY Setup page available by using the **Project>Options** command.

Table 66: Project menu commands (Continued)

Menu Command	Description
 Make & Restart Debugger	Stops the debugger, makes the active build configuration, and starts the debugger again; all in a single command. This button is only available during debugging.

Table 66: Project menu commands (Continued)

Argument variables summary

Variables can be used for paths and arguments. The following argument variables can be used:

Variable	Description
`$CUR_DIR$`	Current directory
`$CUR_LINE$`	Current line
`$EW_DIR$`	Top directory of IAR Embedded Workbench, for example `c:\program files\iar systems\embedded workbench 4.`*`n`*
`$EXE_DIR$`	Directory for executable output
`$FILE_BNAME$`	Filename without extension
`$FILE_BPATH$`	Full path without extension
`$FILE_DIR$`	Directory of active file, no filename
`$FILE_FNAME$`	Filename of active file without path
`$FILE_PATH$`	Full path of active file (in Editor, Project, or Message window)
`$LIST_DIR$`	Directory for list output
`$OBJ_DIR$`	Directory for object output
`$PROJ_DIR$`	Project directory
`$PROJ_FNAME$`	Project file name without path
`$PROJ_PATH$`	Full path of project file
`$TARGET_DIR$`	Directory of primary output file
`$TARGET_BNAME$`	Filename without path of primary output file and without extension
`$TARGET_BPATH$`	Full path of primary output file without extension
`$TARGET_FNAME$`	Filename without path of primary output file
`$TARGET_PATH$`	Full path of primary output file
`$TOOLKIT_DIR$`	Directory of the active product, for example `c:\program files\iar systems\embedded workbench 4.`*`n`*`\430`

Table 67: Argument variables

Configurations for project dialog box

In the **Configuration for project** dialog box—available by choosing **Project>Edit Configurations**—you can define new build configurations for the selected project; either entirely new, or based on a previous project.

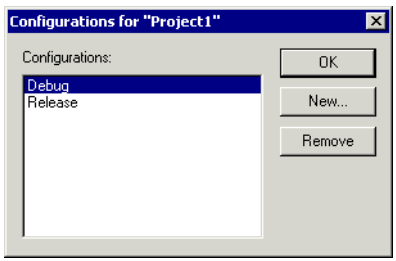


Figure 126: Configurations for project dialog box

The dialog box contains the following:

Operation	Description
Configurations	Lists existing configurations, which can be used as templates for new configurations.
New	Opens a dialog box where you can define new build configurations.
Remove	Removes the configuration that is selected in the **Configurations** list.

Table 68: Configurations for project dialog box options

New Configuration dialog box

In the **New Configuration** dialog box—available by clicking **New** in the **Configurations for project** dialog box—you can define new build configurations; either entirely new, or based on any currently defined configuration.

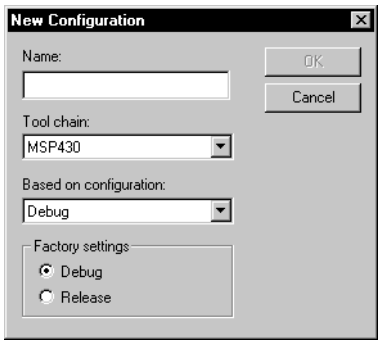


Figure 127: New Configuration dialog box

The dialog box contains the following:

Item	Description
Name	The name of the build configuration.
Tool chain	The target to build for. If you have several versions of IAR Embedded Workbench for different targets installed on your host computer, the drop-down list can contain these targets.
Based on configuration	A currently defined build configuration that you want the new configuration to be based on. The new configuration will inherit the project settings as well as information about the factory settings from the old configuration. If you select None, the new configuration will have default factory settings and not be based on an already defined configuration.
Factory settings	Specifies the default factory settings—either Debug or Release—that you want to apply to your new build configuration. These factory settings will be used by your project if you press the **Factory Settings** button in the **Options** dialog box.

Table 69: New Configuration dialog box options

Create New Project dialog box

The **Create New Project** dialog box is available from the **Project** menu, and lets you create a new project based on a template project. There are template projects available for C/C++ applications, assembler applications, and library projects. You can also create your own template projects.

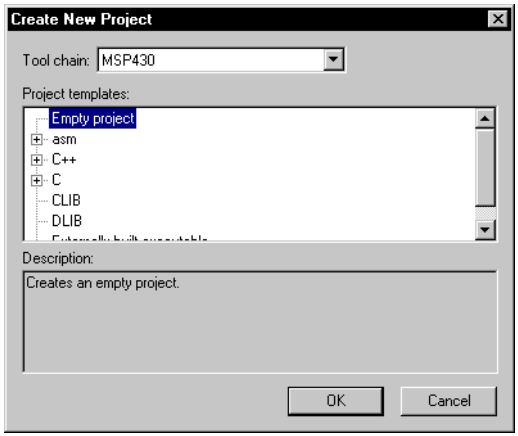


Figure 128: Create New Project dialog box

The dialog box contains the following:

Item	Description
Tool chain	The target to build for. If you have several versions of IAR Embedded Workbench for different targets installed on your host computer, the drop-down list can contain these targets.
Project templates	Lists all available template projects that you can base a new project on.

Table 70: Description of Create New Project dialog box

Options dialog box

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

In the **Category** list you can select the build tool for which you want to set options. The options available in the **Category** list will depend on the tools installed in your IAR Embedded Workbench IDE, and will typically include the following options:

Category	Description
General Options	General options
C/C++ Compiler	MSP430 IAR C/C++ Compiler options
Assembler	MSP430 IAR Assembler options
Custom Build	Options for extending the tool chain
Build Actions	Options for pre-build and post-build actions
Linker	IAR XLINK Linker options. This category is available for application projects.
Library Builder	IAR XAR Library Builder options. This category is available for library projects.
Debugger	IAR C-SPY Debugger options
FET Debugger	FET-specific options
Simulator	Simulator-specific options

Table 71: Project option categories

Note: Additional debugger categories might be available depending on the debugger drivers installed.

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

For detailed information about each option, see the option reference chapters: available in the *MSP430 IAR Embedded Workbench® IDE User Guide.*

- *General options*
- *Compiler options*
- *Assembler options*
- *Custom build options*
- *Build actions options*
- *Linker options*
- *Library builder options*
- *Debugger options*.

For information about the options related to available hardware debugger systems, see the online help system.

Batch Build dialog box

The **Batch Build** dialog box—available by choosing **Project>Batch build**—lists all defined batches of build configurations.

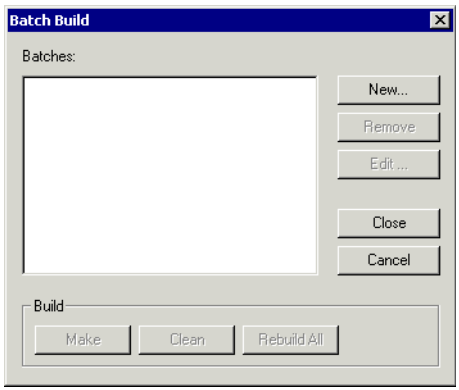


Figure 129: Batch Build dialog box

The dialog box contains the following:

Item	Description
Batches	Lists all currently defined batches of build configurations.
New	Displays the **Edit Batch Build** dialog box, where you can define new batches of build configurations.
Remove	Removes the selected batch.
Edit	Displays the **Edit Batch Build** dialog box, where you can modify already defined batches.
Build	Consists of the three build commands **Make**, **Clean**, and **Rebuild All**.

Table 72: Description of the Batch Build dialog box

Edit Batch Build dialog box

In the **Edit Batch Build** dialog box—available from the **Batch Build** dialog box—you can create new batches of build configurations, and edit already existing batches.

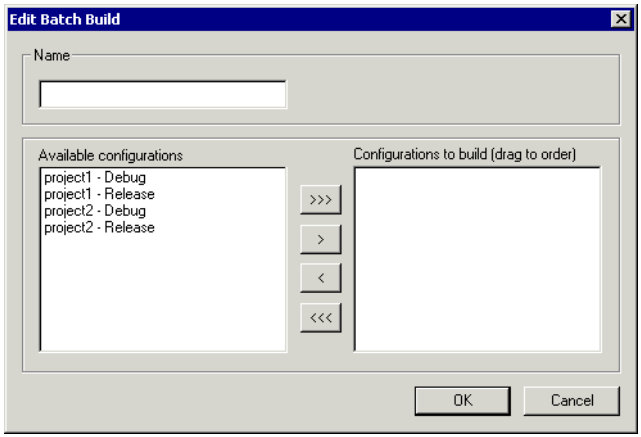


Figure 130: Edit Batch Build dialog box

The dialog box contains the following:

Item	Description
Name	The name of the batch.
Available configurations	Lists all build configurations that are part of the workspace.
Configurations to build	Lists all the build configurations you select to be part of a named batch.

Table 73: Description of the Edit Batch Build dialog box

To move appropriate build configurations from the **Available configurations** list to the **Configurations to build** list, use the arrow buttons. Note also that you can drag the build configurations in the **Configurations to build** field to specify the order between the build configurations.

TOOLS MENU

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

It is a user-configurable menu to which you can add tools for use with IAR Embedded Workbench. Thus, it might look different depending on which tools have been preconfigured to appear as menu items. See *Configure Tools dialog box*, page 303.

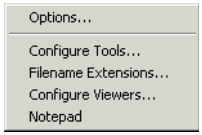


Figure 131: Tools menu

Tools menu commands

Menu command	Description
Options	Displays the **IDE Options** dialog box where you can customize the IAR Embedded Workbench IDE. Select the feature you want to customize by clicking the appropriate tab. Which pages are available in this dialog box depends on your IAR Embedded Workbench IDE configuration, and whether the IDE is in a debugging session or not
Configure Tools	Displays a dialog box where you can set up the interface to use external tools.
Filename Extensions	Displays a set of dialog boxes where you can define the filename extensions to be accepted by the build tools.
Configure Viewers	Displays a dialog box where you can configure viewer applications to open documents with.
Notepad	User-configured. This is an example of a user-configured addition to the Tools menu.

Table 74: Tools menu commands

External Editor page

On the **External Editor** page—available by choosing **Tools>Options**—you can specify an external editor.

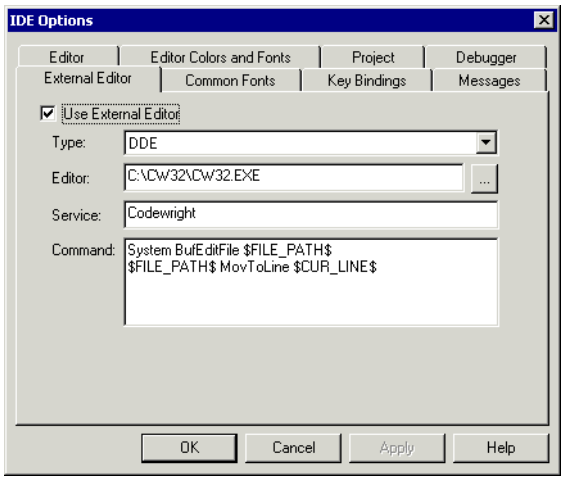


Figure 132: External Editor page with command line settings

Options

Option	Description
Use External Editor	Enables the use of an external editor.
Type	Selects the method for interfacing with the external editor. The type can be either Command Line or DDE (Windows Dynamic Data Exchange).
Editor	Type the filename and path of your external editor. A browse button is available for your convenience.
Arguments	Type any arguments to pass to the editor. Only applicable if you have selected **Type** as `Command Line`.
Service	Type the DDE service name used by the editor. Only applicable if you have selected **Type** as `DDE`.
Command	Type a sequence of command strings to send to the editor. The command strings should be typed as: *`DDE-Topic CommandString`* *`DDE-Topic CommandString`* Only applicable if you have selected **Type** as `DDE`.

Table 75: External Editor options

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 arguments. See *Argument variables summary*, page 279, for information about available argument variables.

Common fonts page

The **Common Fonts** page—available by choosing **Tools>Options**—displays the fonts used for all project windows except the editor windows.

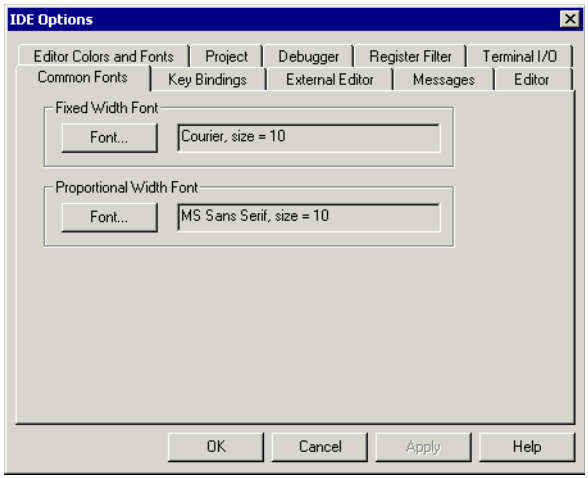


Figure 133: Common Fonts page

With the **Font** buttons you can change the fixed and proportional width fonts, respectively.

Any changes to the **Fixed Width Font** options will apply to the Disassembly, Register, and Memory windows. Any changes to the **Proportional Width Font** options will apply to all other windows.

None of the settings made on this page apply to the editor windows. For information about how to change the font in the editor windows, see *Editor Colors and Fonts page*, page 295.

Key Bindings page

The **Key Bindings** page—available by choosing **Tools>Options**—displays the shortcut keys used for each of the menu options, which you can change, if you wish.

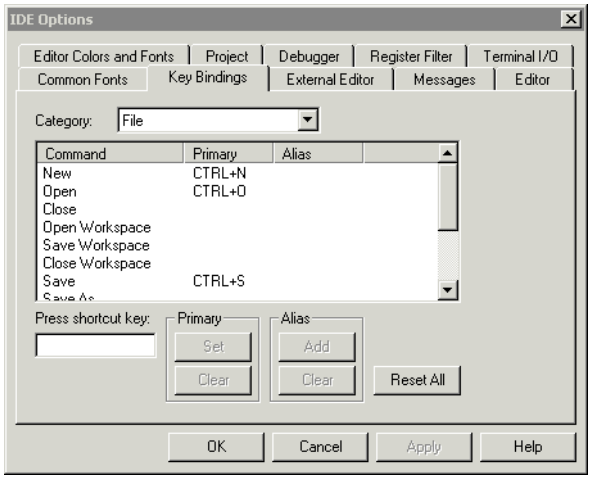


Figure 134: Key Bindings page

Options

Option	Description
Category	Drop-down menu to choose the menu you want to edit. Any currently defined shortcut keys are shown in the scroll list below.
Press shortcut key	Type the key combination you want to use as shortcut key.
Primary	The shortcut key will be displayed next to the command on the menu. Click Set to set the combination, or Clear to delete the shortcut.
Alias	The shortcut key will work but not be displayed on the menu. Click either Add to make the key take effect, or Clear to delete the shortcut.
Reset All	Reverts all command shortcut keys to the factory settings.

Table 76: Key Bindings page options

It is not possible to set or add the shortcut if it is already used by another command.

To delete a shortcut key definition, select the corresponding menu command in the scroll list and click **Clear** under **Primary** or **Alias**. To revert all command shortcuts to the factory settings, click **Reset All**. Click **OK** to make the new shortcut key bindings take effect.

Messages page

On the **Messages** page—available by choosing **Tools>Options**—you can choose the amount of output in the Messages window.

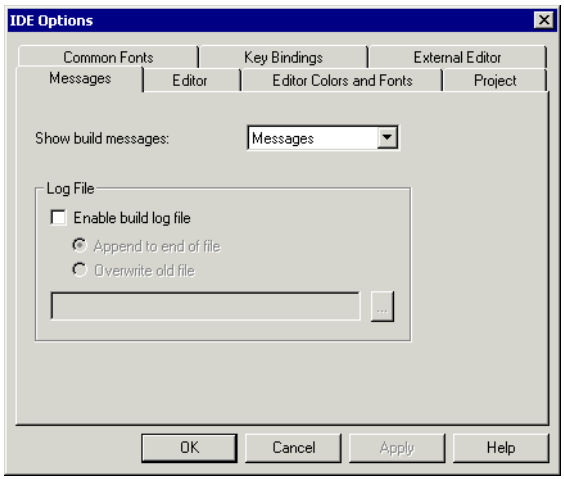


Figure 135: Messages page

Show build messages

Use this drop-down menu to specify the amount of output in the Messages window. Choose between:

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

Log File

Use the options in this area to log build messages in a file. To enable the options, select the **Enable build log file** option. Choose between:

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

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

Editor page

On the **Editor** page—available by choosing **Tools>Options**—you can change the editor options.

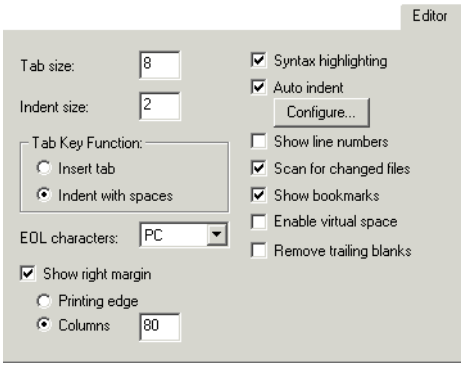


Figure 136: Editor page

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. Either as Insert Tab or as Indent with Spaces.
EOL character	Selects line break character. **PC** (default) uses Windows and DOS end of line character. **Unix** uses UNIX end of line characters. **Preserve** uses the same end of line character as the file had when it was read from the disc drive. The PC format is used by default, and if the read file did not have any breaks, or if there is a mixture of break characters used in the file.
Show right margin	HistoryShows the area of the editor window outside the right-side margin as a light gray field. You can choose to set the size of the text field between the left-side margin and the right-side margin using one of the options **Printing edge** or **Columns**.
Syntax Highlighting	Displays the syntax of C or C++ applications in different text styles.

Table 77: Editor page options

Option	Description
Auto Indent	Ensures that when you press Return, the new line will automatically be indented. For C/C++ source files, indentation will be performed as configured in the **Configure Auto Indent** dialog box. Click the **Configure** button to open the dialog box where you can configure the automatic indentation; see *Configure Auto Indent dialog box*, page 292. For all other text files, the new line will have the same indentation as the previous line.
Show Line Numbers	Displays line numbers in the Editor window.
Scan for Changed Files	Checks if files have been modified by some other tool and automatically reloads them. If a file has been modified in the IAR Embedded Workbench IDE, you will be prompted first.
Show Bookmarks	Displays a column on the left side in the editor window, with icons for compiler errors and warnings, **Find in Files** results, user bookmarks and breakpoints.
Enable Virtual Space	Allows the insertion point to move outside the text area.
Remove trailing blanks	Removes trailing blanks from files when they are saved to disk. Trailing blanks are blank spaces between the last non-blank character and the end of line character.

Table 77: Editor page options (Continued)

For more information about the IAR Embedded Workbench IDE Editor and how it can be used, see *Editing*, page 95.For more information about the IAR Embedded Workbench IDE editor and how it can be used, see the *MSP430 IAR Embedded Workbench® IDE User Guide*.

Configure Auto Indent dialog box

Use the **Configure Auto Indent** dialog box to configure the automatic indentation performed by the editor for C/C++ source code. To open the dialog box:

1. Choose **Tools>Options**.
2. Click the **Editor** tab.
3. Select the **Auto indent** option.

4 Click the **Configure** button.

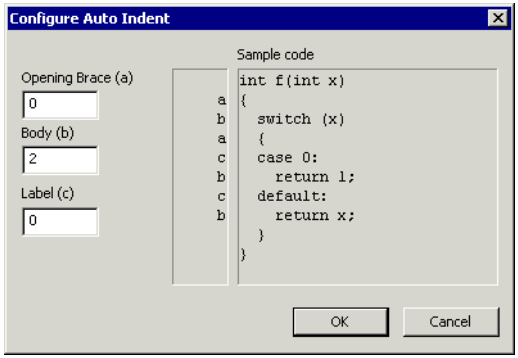


Figure 137: Configure Auto Indent dialog box

To read more about indentation, see *Automatic text indentation*, page 98the *MSP430 IAR Embedded Workbench® IDE User Guide*.

Type the number of spaces to indent in the appropriate text box for each category of indentation:

Opening Brace (a)	The number of spaces used to indent an opening brace.
Body (b)	The number of additional spaces used to indent code after an opening brace, or a statement that continues onto a second line.
Label (c)	The number of additional spaces used to indent a label, including case labels.

Sample code

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

Editor Setup Files page

On the **Editor Setup Files** page—available by choosing **Tools>Options**—you can specify setup files for the editor.

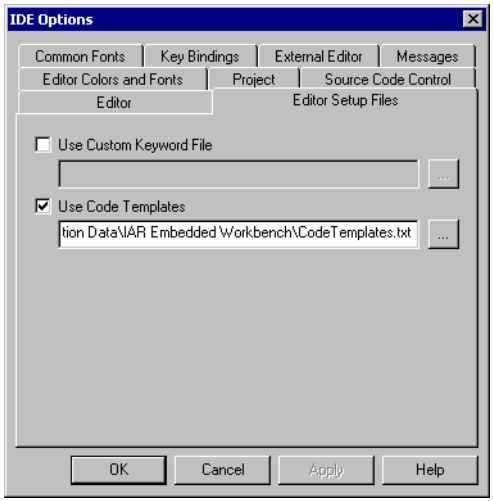


Figure 138: Editor Setup Files page

Use Custom Keyword File

Use this option to specify a text file containing keywords that you want the editor to highlight. For information about syntax coloring, see *Syntax coloring*, page 97the *MSP430 IAR Embedded Workbench® IDE User Guide.*

Use Code Templates

Use this option to specify a text file with code templates that you can use for inserting frequently used code in your source file. For information about using code templates, see *Using and adding code templates*, page 99the *MSP430 IAR Embedded Workbench® IDE User Guide.*

Editor Colors and Fonts page

The **Editor Colors and Fonts** page—available by choosing **Tools>Options**—allows you to specify the colors and fonts used for text in the Editor windows.

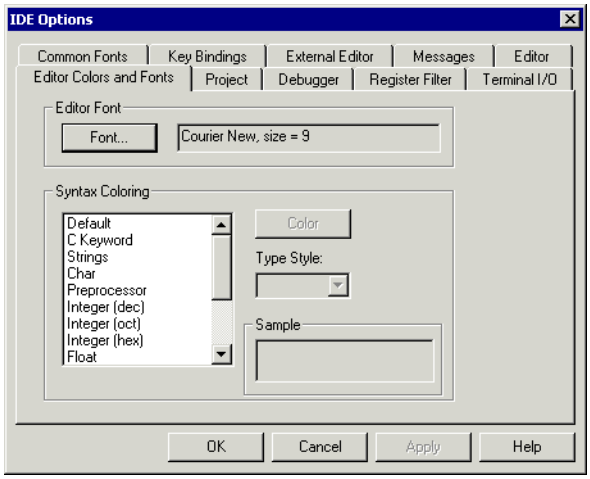


Figure 139: Editor Colors and Fonts page

Options

Option	Description
Font	Opens a dialog box to choose font and its size.
Syntax Coloring	Lists the possible items for which you can specify font and style of syntax. The elements you can customize are: C or C++, compiler keywords, assembler keywords, and user-defined keywords.
Color	Chooses a color from a list of colors.
Type Style	Chooses a type style from a drop-down list.
Sample	Displays the current setting.

Table 78: Editor Colors and Fonts page options

The keywords controlling syntax highlighting for assembler and C or C++ source code are specified in the files `syntax_icc.cfg` and `syntax_asm.cfg`, respectively. These files are located in the `config` directory.

Project page

On the **Project** page—available by choosing **Tools>Options**—you can set options for Make and Build. The following table describes the options and their available settings.

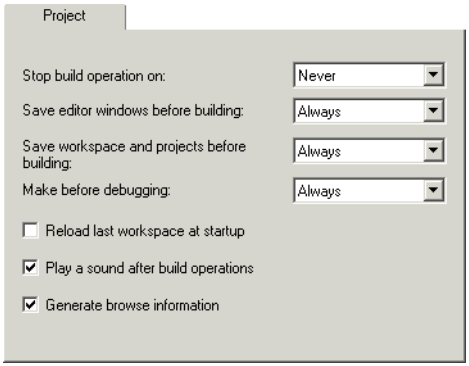


Figure 140: Projects page

Options

Option	Description
Stop build operation on	Specifies when the build operation should stop. Never: Do not stop. Warnings: Stop on warnings and errors. Errors: Stop on errors.
Save editor windows before building	Always: Always save before Make or Build. Ask: Prompt before saving. Never: Do not save.
Save workspace and projects before building	Always: Always save before Make or Build. Ask: Prompt before saving. Never: Do not save.
Make before debugging	Always: Always make before debugging. Ask: Always prompt before Making. Never: Do not make.
Reload last workspace at startup	Select this option if you want the last active workspace to load automatically the next time you start IAR Embedded Workbench.
Play a sound after build operations	Plays a sound when the build operations are finished.

Table 79: Project page options

Option	Description
Generate browse information	Enables the use of the Source Browser window.

Table 79: Project page options (Continued)

Debugger page

On the **Debugger** page—available by choosing **Tools>Options**—you can set options for configuring the debugger environment.

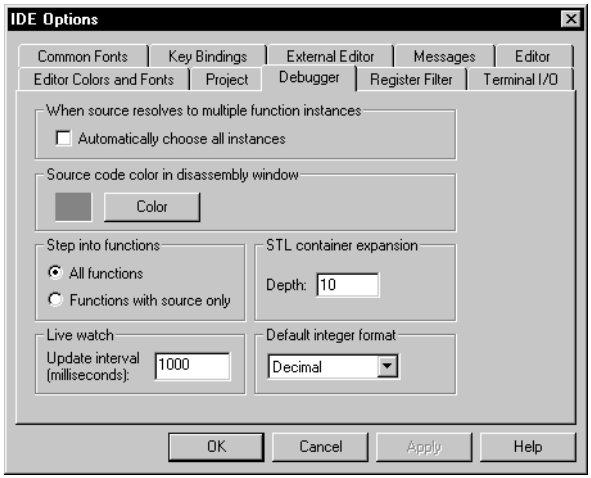


Figure 141: Debugger page

Options

Option	Description
When source resolves to multiple function instances: Automatically choose all instances	Some source code corresponds to multiple code instances, for example template code. When specifying a source location in such code, for example when setting a source breakpoint, you can make C-SPY act on all instances or a subset of instances. This option lets C-SPY act on all instances without first asking.
Source code color in Disassembly window	Specifies the color of the source code in the Disassembly window.

Table 80: Debugger page options

Option	Description
Step into functions	This option controls the behavior of the Step Into command. If you choose the **Functions with source only** option, the debugger will only step into functions for which the source code is known. This helps you avoid stepping into library functions or entering disassembly mode debugging.
STL container expansion	The value decides how many elements that are shown initially when a container value is expanded in, for example, the Watch window. Additional elements can be shown by clicking the expansion arrow.
Live watch	The value decides how often the C-SPY Live Watch window is updated during execution.
Default integer format	Sets the default integer format in the Watch, Locals, and related windows.

Table 80: Debugger page options (Continued)

Register Filter page

On the **Register Filter** page—available by choosing **Tools>Options** when the IAR C-SPY Debugger is running—you can choose to display registers in the Register window in groups you have created yourself. See *Register groups*, page 138, for more information about how to create register groups.For information about register groups, see the *MSP430 IAR Embedded Workbench® IDE User Guide*.

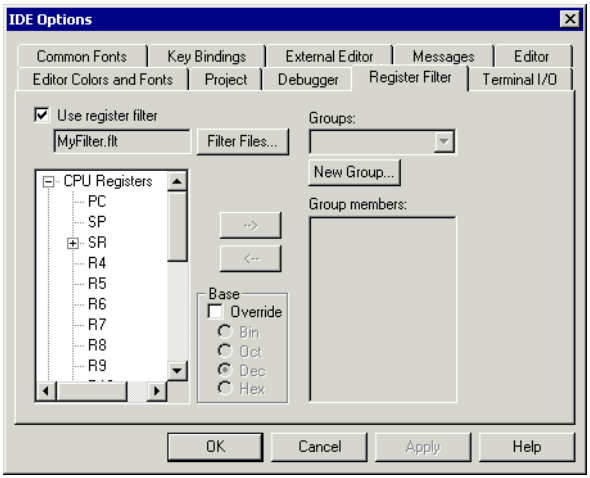


Figure 142: Register Filter page

Options

Option	Description
Use register filter	Enables the usage of register filters.
Filter Files	Displays a dialog box where you can select or create a new filter file.
Groups	Lists available groups in the register filter file, alternatively displays the new register group.
New Group	The name for the new register group.
Group members	Lists the registers selected from the register scroll bar window.
Base	Changes the default integer base.

Table 81: Register Filter options

Terminal I/O page

On the **Terminal I/O** page—available by choosing **Tools>Options** when the IAR C-SPY Debugger is running—you can configure the C-SPY terminal I/O functionality.

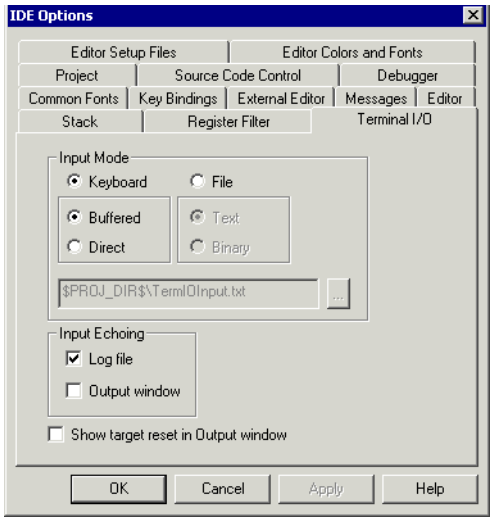


Figure 143: Terminal I/O page

Options

Option	Description
Input Mode: Keyboard	Buffered: All input characters are buffered. Direct: Input characters are not buffered.
Input Mode: File	Input characters are read from a file, either a text file or a binary file. A browse button is available for locating the file.
Input Echoing	Input characters can be echoed either in a log file, or in the C-SPY Terminal I/O window. To echo input in a file requires that you have enabled the option **Enable log file** that is available by choosing **Debug>Logging**.
Show target reset in Output window	When the target resets, a message is displayed in the C-SPY Terminal I/O window.

Table 82: Terminal I/O options

Source Code Control page

On the **Source Code Control** page—available by choosing **Tools>Options**—you can configure the interaction between an IAR Embedded Workbench project and an SCC project.

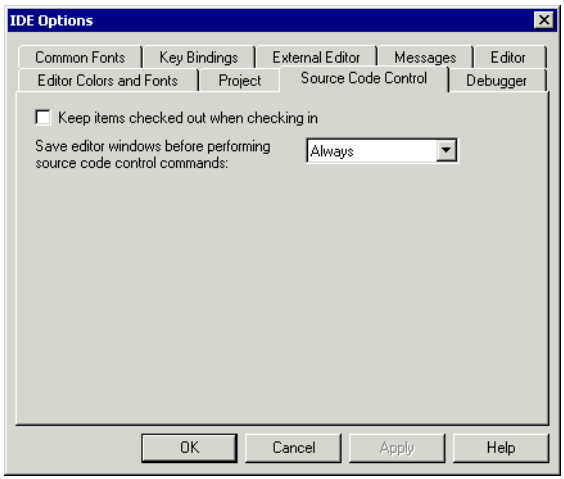


Figure 144: Source Code Control page

Keep items checked out when checking in

Determines the default setting for the option **Keep Checked Out** in the **Check In Files** dialog box; see *Check In Files dialog box*, page 246.

Save editor windows before performing source code control commands

Specifies whether editor windows should be saved before you perform any source code control commands. The following options are available:

Ask	When you perform any source code control commands, you will be asked about saving editor windows first.
Never	Editor windows will *never* be saved first when you perform any source code control commands.
Always	Editor windows will *always* be saved first when you perform any source code control commands.

Stack page

HistoryOn the **Stack** page—available by choosing **Tools>Options**—you can set options specific to the Stack window.

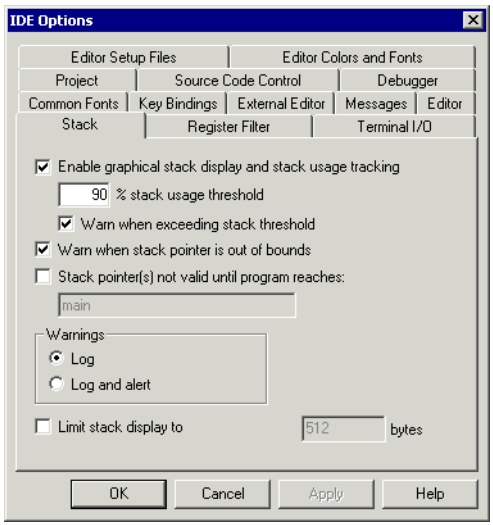


Figure 145: Stack page

Enable graphical stack display and stack usage tracking

Use this option to enable the graphical stack bar available at the top of the Stack window. At the same time, it enables the functionality needed to detect stack overflows. To read more about the stack bar and the information it provides, see *The graphical stack bar*, page 333.

% stack usage threshold

Use this text field to specify the percentage of stack usage above which C-SPY should issue a warning for stack overflow.

Warn when exceeding stack threshold

Use this option to make C-SPY issue a warning when the stack usage exceeds the threshold specified in the **% stack usage threshold** option.

Warn when stack pointer is out of bounds

Use this option to make C-SPY issue a warning when the stack pointer is outside the stack memory range.

Stack pointer(s) not valid until reaching

Use this option to specify a *location* in your application code from where you want the stack display and verification to take place. The Stack window will not display any information about stack usage until execution has reached this location. By default, C-SPY will not track the stack usage before the `main` function. If your application does not have a `main` function, for example, if it is an assembler-only project, your should specify your start label.

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

Warnings

You can choose to issue warnings using one of the following options:

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

Limit stack display to

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

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

Configure Tools dialog box

In the **Configure Tools** dialog box—available from the **Tools** menu—you can specify a user-defined tool to add to the Tools menu.

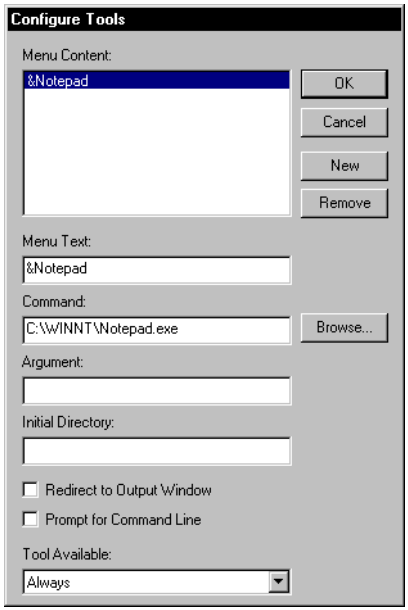


Figure 146: Configure Tools dialog box

Note: If you intend to add an external tool to the standard build tool chain, see *Extending the tool chain*, page 93the *MSP430 IAR Embedded Workbench® IDE User Guide*.

Options

Option	Description
Menu Content	Lists all available user defined menu commands.
Menu Text	Specifies the text for the menu command. By adding the sign `&`, the following letter, `N` in this example, will then appear as the mnemonic key for this command. The text you type in this field will be reflected in the **Menu Content** field.
Command	Specifies the command, and its path, to be run when you choose the command from the menu. A browse button is available for your convenience.
Argument	Optionally type an argument for the command.
Initial Directory	Specifies an initial working directory for the tool.
Redirect to Output window	Specifies any console output from the tool to the **Tool Output** page in the Messages window. Tools that are launched with this option cannot receive any user input, for instance input from the keyboard. Tools that *require* user input or make special assumptions regarding the console that they execute in, will *not* work at all if launched with this option.
Prompt for Command Line	Displays a prompt for the command line argument when the command is chosen from the **Tools** menu.
Tool Available	Specifies in which context the tool should be available, only when debugging or only when not debugging.

Table 83: Configure Tools dialog box options

Note: 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.

You can remove a command from the **Tools** menu by selecting it in this list and clicking **Remove**.

Click **OK** to confirm the changes you have made to the **Tools** menu.

The menu items you have specified will then be displayed on the **Tools** menu.

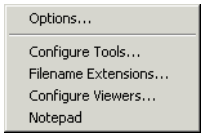


Figure 147: Customized Tools menu

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. These are the command shells that can be entered as commands:

System	Command shell
Windows 98/Me	`command.com`
Windows NT/2000/XP	`cmd.exe` (recommended) or `command.com`

Table 84: Command shells

Filename Extensions dialog box

In the **Filename Extensions** dialog box—available from the **Tools** menu—you can customize the filename extensions recognized by the build tools. This is useful if you have many source files that have a different filename extension.

If you have an IAR Embedded Workbench for a different microprocessor installed on your host computer, it can appear in the **Tool Chain** box. In that case you should select the tool chain you want to customize.

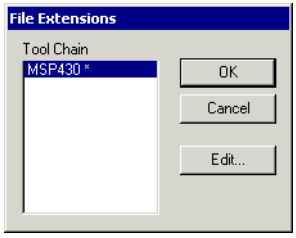


Figure 148: Filename Extensions dialog box

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

Click **Edit** to open the **Filename Extension Overrides** dialog box.

Filename Extension Overrides dialog box

The **Filename Extension Overrides** dialog box—available by clicking **Edit** in the **Filename Extensions** dialog box—lists the available tools in the build chain, their factory settings for filename extensions, and any defined overrides.

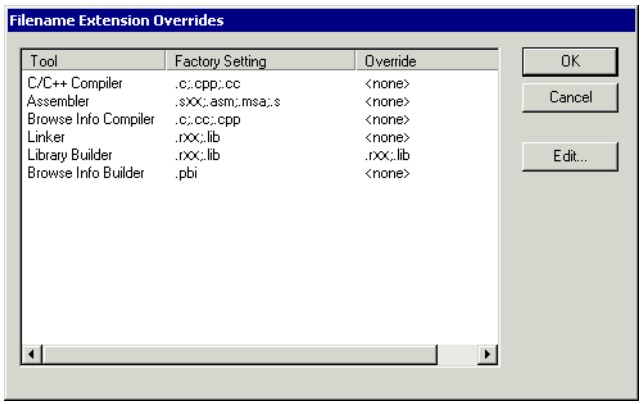


Figure 149: Filename Extension Overrides dialog box

Select the tool for which you want to define more recognized filename extensions, and click **Edit** to open the **Edit Filename Extensions** dialog box.

Edit Filename Extensions dialog box

The **Edit File Extensions** dialog box—available by clicking **Edit** in the **Filename Extension Overrides** dialog box—lists the filename extensions accepted by default, and you can also define new filename extensions.

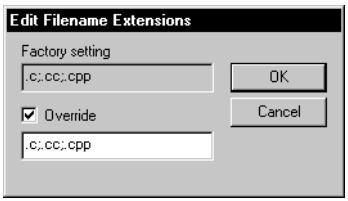


Figure 150: Edit Filename Extensions dialog box

Click **Override** and type the new filename extension you want to be recognized. Extensions can be separated by commas or semicolons, and should include the leading period.

Configure Viewers dialog box

The **Configure Viewers** dialog box—available from the **Tools** menu—lists the filename extensions of document formats that IAR Embedded Workbench can handle, and which viewer application that will be used for opening the document type. **Explorer Default** in the **Action** column means that the default application associated with the specified type in Windows Explorer is used for opening the document type.

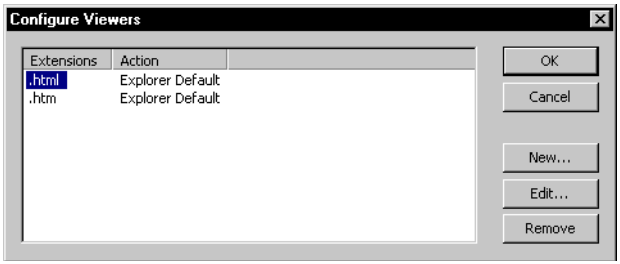


Figure 151: Configure Viewers dialog box

To specify how to open a new document type or editing the setting for an existing document type, click **New** or **Edit** to open the **Edit Viewer Extensions** dialog box.

Edit Viewer Extensions dialog box

Type the filename extension for the document type—including the separating period (.)—in the **Filename extensions** box.

Figure 152: Edit Viewer Extensions dialog box

Then choose one of the **Action** options:

- **Built-in text editor**—select this option to open all documents of the specified type with the IAR Embedded Workbench text editor.
- **Use file explorer associations**—select this option to open all documents with the default application associated with the specified type in Windows Explorer.

- **Command line**—select this option and type or browse your way to the viewer application, and give any command line options you would like to the tool.

WINDOW MENU

Use the commands on the **Window** menu to manipulate the IAR Embedded Workbench IDE windows and change their arrangement on the screen.

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

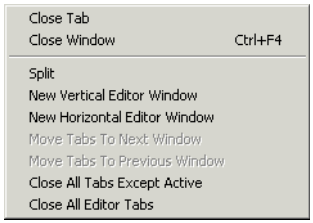


Figure 153: Window menu

Window menu commands

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

Table 85: Window menu commands

HELP MENU

The **Help** menu provides help about the IAR Embedded Workbench IDE and displays the version numbers of the user interface and of the MSP430 IAR Embedded Workbench IDE.

Menu command	Description
Content	Opens the contents page of the IAR Embedded Workbench IDE online help.
Index	Opens the index page of the IAR Embedded Workbench IDE online help.
Search	Opens the search page of the IAR Embedded Workbench IDE online help.
Release notes	Provides access to late-breaking information about IAR Embedded Workbench.
MSP430 Embedded Workbench User Guide	Provides access to an online version of this user guide, available in PDF format.
MSP430 Assembler Reference Guide	Provides access to an online version of the *MSP430 IAR Assembler Reference Guide*, available in PDF format.
MSP430 C/C++ Compiler Reference Guide	Provides access to an online version of the *MSP430 IAR C/C++ Compiler Reference Guide*, available in PDF format.
MSP430 Migration Guide	Provides access to an online version of the *MSP430 IAR Embedded Workbench Migration Guide*, available in hypertext PDF format.
IAR MISRA C Reference Guide	Provides access to the online version of the *IAR Embedded Workbench® MISRA C Reference Guide*, available in PDF format.
Product updates	Provides access to the latest product updates available on the IAR Systems web site.
Linker and Library Tools Reference Guide	Provides access to the online version of the *IAR Linker and Library Tools Reference Guide*, available in PDF format.
IAR on the Web	Allows you to browse the home page, the news page, and the technical notes search page of the IAR Systems web site, and to contact IAR Technical Support.

Table 86: Help menu commands

Menu command	Description
Startup Screen	Displays the **Embedded Workbench Startup** dialog box; see *Embedded Workbench Startup dialog box*, page 311.
About>Product Info	Displays detailed information about the installed IAR products. Copy this information (using the **Ctrl+C** keyboard shortcut) and include it in your message if you contact IAR Technical Support via electronic mail.
About>Install Log	Opens the license manager log file `lms.log` in the editor. Attach this file to the email message if you contact IAR Technical Support regarding any problems related to the license management system.

Table 86: Help menu commands (Continued)

Note: Additional documentation might be available on the **Help** menu depending on your product installation.

Embedded Workbench Startup dialog box

The **Embedded Workbench Startup** dialog box—available from the Help menu—provides an easy access to ready-made example workspaces that can be built and executed *out of the box* for a smooth development startup.

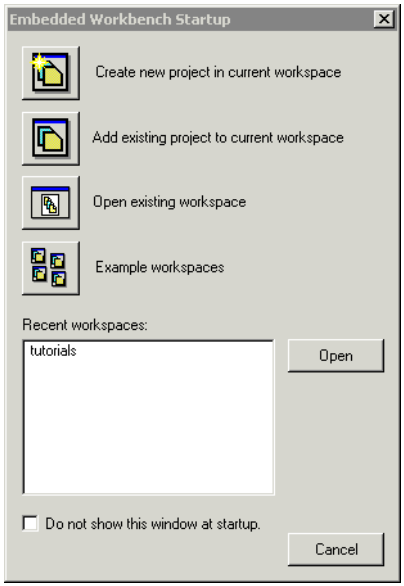


Figure 154: Embedded Workbench Startup dialog box

C-SPY® Debugger reference

This chaptersection contains detailed reference information about the windows, menus, menu commands, and the corresponding components that are specific for the IAR C-SPY Debugger.

C-SPY windows

The following windows specific to C-SPY are available in the IAR C-SPY Debugger:

- IAR C-SPY Debugger main window
- Disassembly window
- Memory window
- Register window
- Watch window
- Locals window
- Auto window
- Live Watch window
- Quick Watch window
- Call Stack window
- Terminal I/O window
- Code Coverage window
- Profiling window
- Stack windowHistory
- LCD window.

Additional windows will be available depending on which C-SPY driver you are using. For information about driver-specific windows, see the driver-specific documentation.

EDITING IN C-SPY WINDOWS

You can edit the contents of the Memory, Register, Auto, Watch, Locals, Live Watch, and Quick Watch windows.

Use the following keyboard keys to edit the contents of the Register and Watch windows:

Key	Description
Enter	Makes an item editable and saves the new value.

Table 87: Editing in C-SPY windows

Key	Description
Esc	Cancels a new value.

Table 87: Editing in C-SPY windows (Continued)

IAR C-SPY DEBUGGER MAIN WINDOW

When you start the IAR C-SPY Debugger, the following debugger-specific items appear in the main IAR Embedded Workbench IDE window:

- A dedicated debug menu with commands for executing and debugging your application
- Depending on the C-SPY driver you are using, a driver-specific menu. Typically, this menu contains menu commands for opening driver-specific windows and dialog boxes. See the driver-specific documentation for more information
- A special debug toolbar
- Several windows and dialog boxes specific to C-SPY.

The window might look different depending on which components you are using.

Each window item is explained in greater detail in the following sections.

Menu bar

In addition to the menus available in the development environment, the **Debug** menu is available when C-SPY is running. The **Debug** menu provides commands for executing and debugging the source application. Most of the commands are also available as icon buttons in the debug toolbar. The following menus are available when C-SPY is running:

Menu	Description
Debug	The Debug menu provides commands for executing and debugging the source application. Most of the commands are also available as icon buttons in the debug toolbar.
Simulator	The Simulator menu provides access to the dialog boxes for setting up interrupt simulation and memory maps. Only available when the C-SPY Simulator is used.
Emulator	The Emulator menu provides access to commands specific to the C-SPY FET debugger. Only available when the C-SPY FET debugger is used.

Table 88: C-SPY menu

Additional menus might be available, depending on which debugger drivers have been installed; for information, see the driver-specific documentation.

Debug toolbar

The debug toolbar provides buttons for the most frequently-used commands on the **Debug** menu.

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 dimmed and you will not be able to select it.

The following diagram shows the command corresponding to each button:

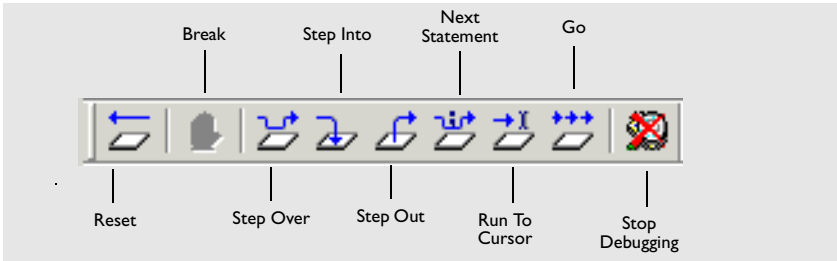


Figure 155: C-SPY debug toolbar

DISASSEMBLY WINDOW

The C-SPY Disassembly window—available from the **View** menu—shows the application being debugged as disassembled application code.

The current position—highlighted in green—indicates the next assembler instruction to be executed. You can move the cursor to any line in the Disassembly window by clicking on the line. Alternatively, you can move the cursor using the navigation keys. Breakpoints are indicated in red. Code that has been executed—code coverage—is indicated with a green diamond.

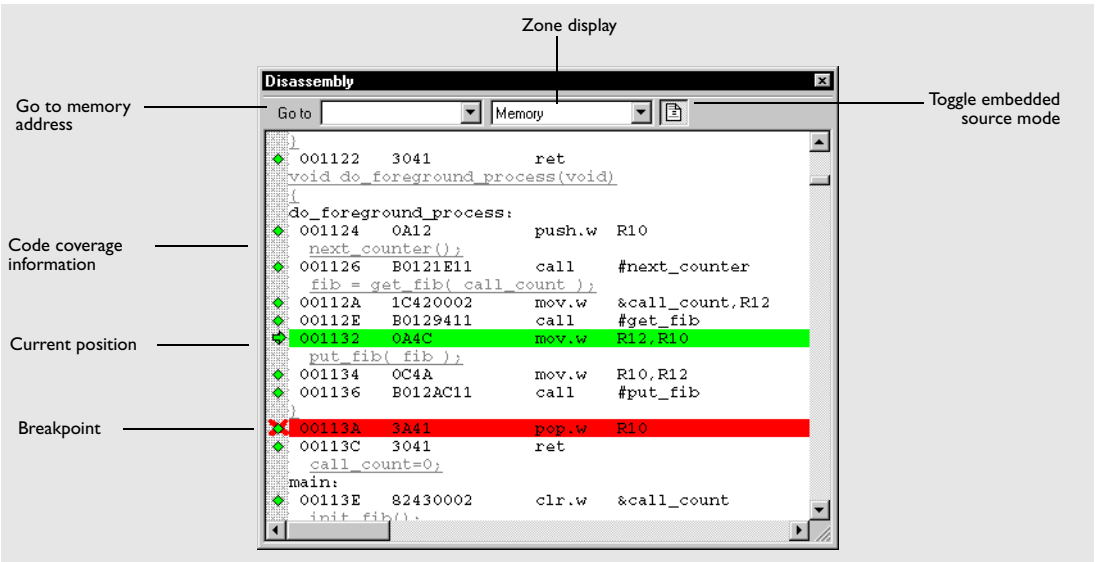


Figure 156: C-SPY Disassembly window

To change the default color of the source code in the Disassembly window, choose **Tools>Options>Debugger**. Set default color using the **Set source code coloring in Disassembly window** option.

To view the corresponding assembler code for a function, you can select it in the editor window and drag it to the Disassembly window.

Disassembly window operations

At the top of the window you can find a set of useful text boxes, drop-down lists and command buttons:

Operation	Description
Go to	The memory location you want to view.

Table 89: Disassembly window operations

Operation	Description
Zone display	Lists the available memory or register zones to display. Read more about Zones in section *Memory zones*, page 114.
Disassembly mode	Toggles between showing only disassembly or disassembly together with the corresponding source code. Source code requires that the corresponding source file has been compiled with debug information.

Table 89: Disassembly window operations (Continued)

Disassembly context menu

Clicking the right mouse button in the Disassembly window displays a context menu which gives you access to some extra commands.

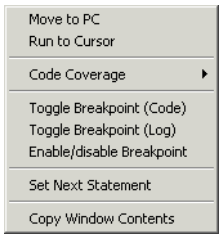


Figure 157: Disassembly window context menu

Operation	Description
Move to PC	Displays code at the current program counter location.
Run to Cursor	Executes the application from the current position up to the line containing the cursor.
Code Coverage	Opens a submenu with commands for controlling code coverage.
Enable	**Enable** toggles code coverage on and off.
Show	**Show** toggles between displaying and hiding code coverage. Executed code is indicated by a green diamond.
Clear	**Clear** clears all code coverage information.
Toggle Breakpoint (Code)	Toggles a code breakpoint. Assembler instructions at which code breakpoints have been set are highlighted in red. For information about code breakpoints, see *Code breakpoints dialog box*, page 256.
Toggle Breakpoint (Log)	Toggles a log breakpoint for trace printouts. Assembler instructions at which log breakpoints have been set are highlighted in red. For information about log breakpoints, see *Log breakpoints dialog box*, page 258.
Enable/Disable Breakpoint	Enables and Disables a breakpoint.

Table 90: Disassembly context menu commands

Operation	Description
Set Next Statement	Sets program counter to the location of the insertion point.
Copy Window Contents	Copies the selected contents of the Disassembly window to the clipboard.

Table 90: Disassembly context menu commands (Continued)

MEMORY WINDOW

The Memory window—available from the **View** menu—gives an up-to-date display of a specified area of memory and allows you to edit it. You can open several instances of this window, which is very convenient if you want to keep track of different memory or register zones, or monitor different parts of the memory.

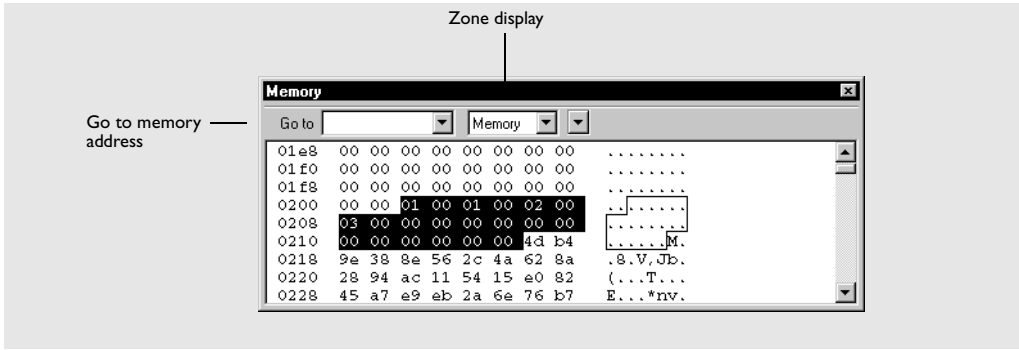


Figure 158: Memory window

To view the memory corresponding to a variable, you can select it in the editor window and drag it to the Memory window.

Memory window operations

At the top of the window you can find commands for navigation:

Operation	Description
Go to	The address of the memory location you want to view.
Zone display	Lists the available memory or register zones to display. Read more about Zones in section *Memory zones*, page 114.

Table 91: Memory window operations

Memory window context menu

The context menu available in the Memory window provides above commands, edit commands, and a command for opening the **Fill** dialog box.

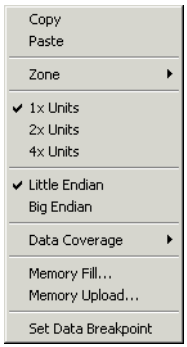


Figure 159: Memory window context menu

Menu command	Description
Copy, Paste	Standard editing commands.
Zone	Lists the available memory or register zones to display. Read more about Zones in *Memory zones*, page 114.
x1, x2, x4 Units	Switches between displaying the memory contents in units of 8, 16, or 32 bits
Little Endian Big Endian	Switches between displaying the contents in big-endian or little-endian order. An asterisk (*) indicates the default byte order.
Data Coverage	
Enable	**Enable** toggles data coverage on and off.
Show	**Show** toggles between showing and hiding data coverage.
Clear	**Clear** clears all data coverage information.
Memory Fill	Opens the Fill dialog box, where you can fill a specified area with a value.
Memory Upload	Displays the Memory Upload dialog box, where you can save a selected memory area to a file in Intel Hex format.
Set Data Breakpoint	Sets breakpoints directly in the Memory window. The breakpoint is not highlighted; you can see, edit, and remove it in the Breakpoints dialog box. The breakpoints you set in this window will be triggered for both read and write access.

Table 92: Commands on the memory window context menu

Data coverage display

Data coverage is displayed with the following colors:

- Yellow indicates data that has been read
- Blue indicates data that has been written
- Green indicates data that has been both read and written.

Fill dialog box

In the **Fill** dialog box—available from the context menu available in the Window memory—you can fill a specified area of memory with a value.

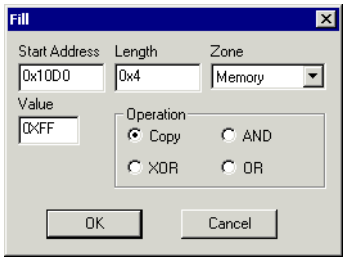


Figure 160: Fill dialog box

Options

Option	Description
Start Address	Type the start address—in binary, octal, decimal, or hexadecimal notation.
Length	Type the length—in binary, octal, decimal, or hexadecimal notation.
Zone	Select memory zone.
Value	Type the 8-bit value to be used for filling each memory location.

Table 93: Fill dialog box options

These are the available memory fill 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.

Table 94: Memory fill operations

Operation	Description
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.

Table 94: Memory fill operations (Continued)

REGISTER WINDOW

The Register window—available from the **View** menu—gives an up-to-date display of the contents of the processor registers, and allows you to edit them. When a value changes it becomes highlighted. Some registers are expandable, which means that the register contains interesting bits or sub-groups of bits.

You can open several instances of this window, which is very convenient if you want to keep track of different register groups.

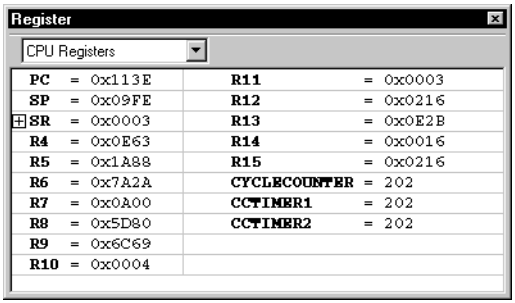


Figure 161: Register window

You can select which register group to display in the Register window using the drop-down list. To define application-specific register groups, see *Defining application-specific groups*, page 139.For more information about register groups, see the *MSP430 IAR Embedded Workbench® IDE User Guide*.

WATCH WINDOW

The Watch window—available from the **View** menu—allows you to monitor the values of C-SPY expressions or variables. You can view, add, modify, and remove expressions in the Watch window. Tree structures of arrays, structs, and unions are expandable, which means that you can study each item of these.

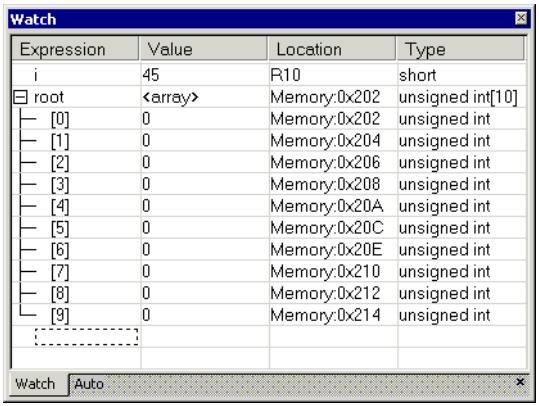


Figure 162: Watch window

Every time execution in C-SPY stops, a value that has changed since the last stop is highlighted. In fact, every time memory changes, the values in the Watch window are recomputed, including updating the red highlights.History

Watch window context menu

The context menu available in the Watch window provides commands for adding and removing expressions, changing the display format of expressions, as well as commands for changing the default type interpretation of variables.

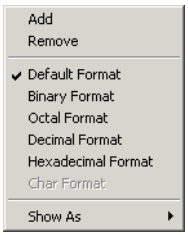


Figure 163: Watch window context menu

The menu contains the following commands:

Menu command	Description
Add, Remove	Adds or removes the selected expression.
Default Format, Binary Format, Octal Format, Decimal Format, Hexadecimal Format, Char Format	Changes the display format of expressions. The display format setting affects different types of expressions in different ways, see Table 96, *Effects of display format setting on different types of expressions*. Your selection of display format is saved between debug sessions.
Show As	Provides a submenu with commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—as these are by default displayed as integers. For more information, see *Viewing assembler variables*, page 128.

Table 95: Watch window context menu commands

The display format setting affects different types of expressions in different ways:

Type of expressions	Effects of display format setting
Variable	The display setting affects only the selected variable, not other variables.
Array element	The display setting affects the complete array, that is, same display format is used for each array element.
Structure field	All elements with the same definition—the same field name and C declaration type—are affected by the display setting.

Table 96: Effects of display format setting on different types of expressions

LOCALS WINDOW

The Locals window—available from the **View** menu—automatically displays the local variables and function parameters.

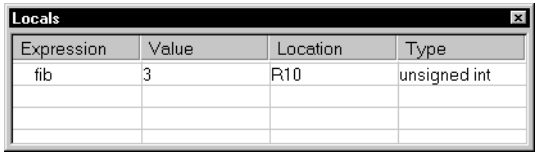


Figure 164: Locals window

Locals window context menu

The context menu available in the Locals window provides commands for changing the display format of expressions; for information about these commands, see *Watch window context menu*, page 322.

AUTO WINDOW

The Auto window—available from the **View** menu—automatically displays a useful selection of variables and expressions in, or near, the current statement.

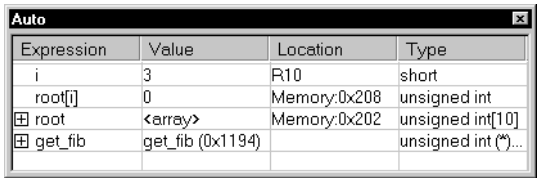


Figure 165: Auto window

Auto window context menu

The context menu available in the Auto window provides commands for changing the display format of expressions; for information about these commands, see *Watch window context menu*, page 322.

LIVE WATCH WINDOW

The Live Watch window—available from the **View** menu—repeatedly samples and displays the value of expressions while your application is executing. Variables in the expressions must be statically located, such as global variables.

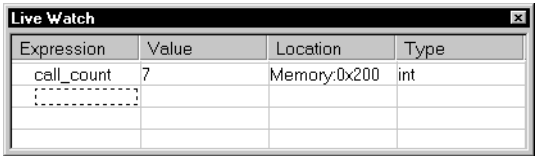


Figure 166: Live Watch window

Typically, this window is useful for hardware target systems supporting this feature.

Live Watch window context menu

The context menu available in the Live Watch window provides commands for adding and removing expressions, changing the display format of expressions, as well as commands for changing the default type interpretation of variables. For information about these commands, see *Watch window context menu*, page 322.

In addition, the menu contains the **Options** command, which opens the **Debugger** dialog box where you can set the **Update interval** option. The default value of this option is 1000 milliseconds, which means the **Live Watch** window will be updated once every second during program execution.

QUICK WATCH WINDOW

In the Quick Watch window—available from the **View** menu—you can watch the value of a variable or expression and evaluate expressions.

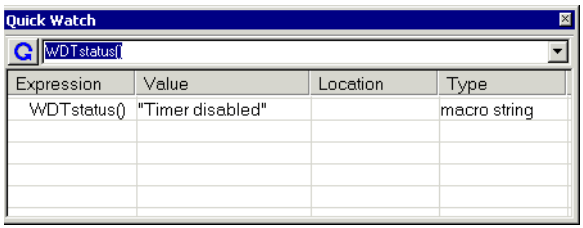


Figure 167: Quick Watch window

Type the expression you want to examine in the **Expressions** text box. Click the **Recalculate** button to calculate the value of the expression. For examples about how to use the Quick Watch window, see *Using the Quick Watch window*, page 126 and *Executing macros using Quick Watch*, page 148the *MSP430 IAR Embedded Workbench® IDE User Guide*.

Quick Watch window context menu

The context menu available in the Quick Watch window provides commands for changing the display format of expressions, as well as commands for changing the default type interpretation of variables. For information about these commands, see *Watch window context menu*, page 322.

In addition, the menu contains the **Add to Watch window** command, which adds the selected expression to the Watch window.

CALL STACK WINDOW

The Call stack window—available from the **View** menu—displays the C function call stack with the current function at the top. To inspect a function call, double-click it. C-SPY now focuses on that call frame instead.

Figure 168: Call Stack window

Each entry has the format:

`function(values)`

where `(values)` is a list of the current value of the parameters, or empty if the function does not take any parameters.

If the **Step Into** command steps into a function call, the name of the function is displayed in the grey bar at the top of the window. This is especially useful for implicit function calls, such as C++ constructors, destructors, and operators.

Call Stack window context menu

The context menu available by right-clicking in the Call Stack window provides the following commands:

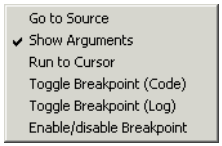


Figure 169: Call Stack window context menu

Commands

Go to Source	Displays the selected functions in the Disassembly or editor windows.
Show Arguments	Shows function arguments.
Run to Cursor	Executes to the function selected in the call stack.
Toggle Breakpoint (Code)	Toggles a code breakpoint.
Toggle Breakpoint (Log)	Toggles a log breakpoint.
Enable/Disable Breakpoint	Enables or disables the selected breakpoint.

TERMINAL I/O WINDOW

In the Terminal I/O window—available from the **View** menu—you can enter input to the application, and display output from it. To use this window, you need to link the application with the option **Debug info with terminal I/O**. C-SPY will then direct `stdin`, `stdout` and `stderr` to this window. If the Terminal I/O window is closed, C-SPY will open it automatically when input is required, but not for output.

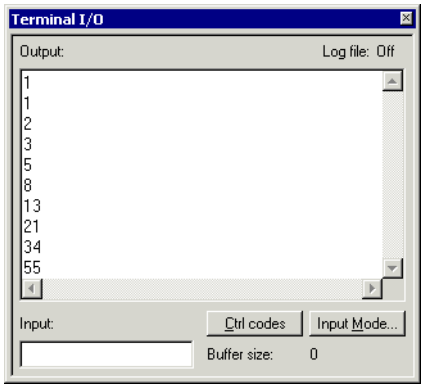


Figure 170: Terminal I/O window

Clicking the **Ctrl codes** button opens a menu with submenus for input of special characters, such as `EOF` (end of file) and `NUL`.

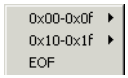


Figure 171: Ctrl codes menu

Clicking the **Input Mode** button opens the **Change Input Mode** dialog box where you choose whether to input data from the keyboard or from a text file.

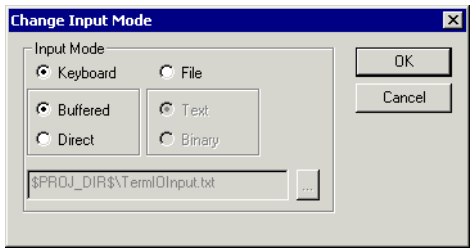


Figure 172: Change Input Mode dialog box

For reference information about the options available in the dialog box, see *Terminal I/O page*, page 299.

CODE COVERAGE WINDOW

Code coverage is only supported by the C-SPY Simulator.

The Code Coverage window—available from the **View** menu—reports the status of the current code coverage analysis, that is, what parts of the code that have been executed at least once since the start of the analysis. The compiler generates detailed stepping information in the form of *step points* at each statement, as well as at each function call. The report includes information about all modules and functions. It reports the amount of all step points, in percentage, that have been executed and lists all step points that have not been executed up to the point where the application has been stopped. The coverage will continue until turned off.

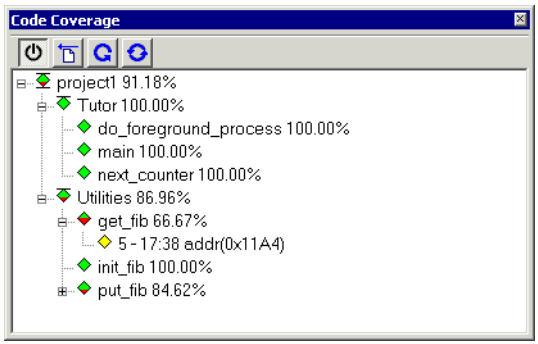


Figure 173: Code Coverage window

Note:

- You can enable the Code Coverage plugin module on the **Debugger>Plugins** page available in the **Options** dialog box.
- Code coverage is not supported by all C-SPY drivers. For information about whether the C-SPY driver you are using supports code coverage, see *Differences between the C-SPY drivers*, page 192the driver-specific documentation. Code coverage is supported by the C-SPY Simulator.

Code coverage commands

In addition to the commands available as icon buttons in the toolbar, clicking the right mouse button in the Code Coverage window displays a context menu that gives you access to these and some extra commands.

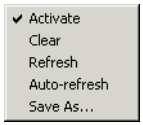


Figure 174: Code coverage context menu

You can find the following commands on the menu:

Activate/Deactivate	Switches code coverage on and off during execution.
Clear	Clears the code coverage information. All step points are marked as not executed.
Refresh	Updates the code coverage information and refreshes the window. All step points that has been executed since the last refresh are removed from the tree.
Auto-refresh	Toggles the automatic reload of code coverage information on and off. When turned on, the code coverage information is reloaded automatically when C-SPY stops at a breakpoint, at a step point, and at program exit.
Save As	Saves the current code coverage information in a text file.

The following icons are used to give you an overview of the current status on all levels:

- A red diamond signifies that 0% of the code has been executed
- A green diamond signifies that 100% of the code has been executed
- A red and green diamond signifies that some of the code has been executed
- A yellow diamond signifies a step point that has not been executed.

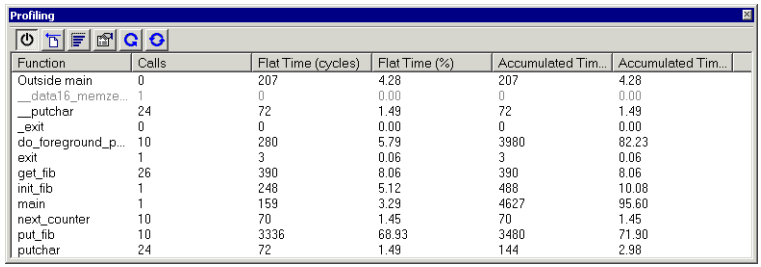
For step point lines, the information displayed is the column number range and the row number of the step point in the source window, followed by the address of the step point.

```
<column start>-<column end>:<row>.
```

PROFILING WINDOW

The Profiling window—available from the **View** menu—displays profiling information, that is, timing information for the functions in an application. Profiling must be turned on explicitly using a button in the window's toolbar, and will stay active until it is turned off.

The profiler measures time at the entry and return of a function. This means that time consumed in a function is not added until the function returns or another function is called. You will only notice this if you are stepping into a function.



Function	Calls	Flat Time (cycles)	Flat Time (%)	Accumulated Tim...	Accumulated Tim...
Outside main	0	207	4.28	207	4.28
__data16_memze...	1	0	0.00	0	0.00
__putchar	24	72	1.49	72	1.49
_exit	0	0	0.00	0	0.00
do_foreground_p...	10	280	5.79	3980	82.23
exit	1	3	0.06	3	0.06
get_fib	26	390	8.06	390	8.06
init_fib	1	248	5.12	488	10.08
main	1	159	3.29	4627	95.60
next_counter	10	70	1.45	70	1.45
put_fib	10	3336	68.93	3480	71.90
putchar	24	72	1.49	144	2.98

Figure 175: Profiling window

Note:

- You can enable the Profiling plugin module on the **Debugger>Plugins** page available in the **Options** dialog box.
- Profiling is not supported by all C-SPY drivers. For information about whether the C-SPY driver you are using supports profiling, see *Differences between the C-SPY drivers*, page 192the driver-specific documentation. Profiling is supported by the C-SPY Simulator.

Profiling commands

In addition to the toolbar buttons, the context menu available in the Profiling window gives you access to these and some extra commands:

Figure 176: Profiling context menu

You can find the following commands on the menu:

Activate — Toggles profiling on and off during execution.

New measurement — Starts a new measurement. By clicking the button, the values displayed are reset to zero.

	Graph	Displays the percentage information for Flat Time and Accumulated Time as graphs (bar charts) or numbers.
	Show details	Shows more detailed information about the function selected in the list. A window is opened showing information about callers and callees for the selected function.
	Refresh	Updates the profiling information and refreshes the window.
	Auto refresh	Toggles the automatic update of profiling information on and off. When turned on, the profiling information is updated automatically when C-SPY stops at a breakpoint, at a step point, and at program exit.
	Save As	Saves the current profiling information in a text file.

Profiling columns

The Profiling window contains the following columns:

Column	Description
Function	The name of each function.
Calls	The number of times each function has been called.
Flat Time	The total time spent in each function in cycles or as a percentage of the total number of cycles, excluding all function calls made from that function.
Accumulated Time	Time spent in each function in cycles or as a percentage of the total number of cycles, including all function calls made from that function.

Table 97: Profiling window columns

There is always an item in the list called **Outside main**. This is time that cannot be placed in any of the functions in the list. That is, code compiled without debug information, for instance, all startup and exit code, and C/C++ library code.

STACK WINDOW

HistoryThe Stack window is a memory window that displays the contents of the stack. In addition, some integrity checks of the stack can be performed to detect and warn about problems with stack overflow. For example, the Stack window is useful for determining the optimal size of the stack.

Before you can open the Stack window you must make sure it is enabled: choose **Project>Options>Debugger>Plugins** and select **Stack** from the list of plugins. In C-SPY, you can then open a Stack window by choosing **View>Stack**. You can open several Stack windows, each showing a different stack—if several stacks are available—or the same stack with different display settings.

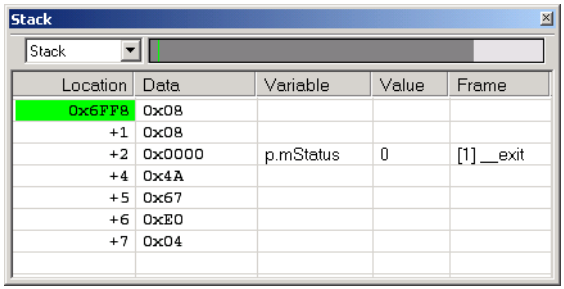


Figure 177: Stack window

The stack drop-down menu

If the microcontroller you are using has multiple stacks, you can use the stack drop-down menu at the top of the window to select which stack to view.

The graphical stack bar

At the top of the window, a stack bar displays the state of the stack graphically. To view the stack bar you must make sure it is enabled: choose **Tools>Options>Stack** and select the option **Enable graphical stack display and stack usage tracking**.

The left end of the stack bar represents the bottom of the stack, in other words, the position of the stack pointer when the stack is empty. The right end represents the end of the memory space reserved for the stack. A green line represents the current value of the stack pointer. The part of the stack memory that has been used during execution is displayed in a dark gray color, and the unused part in a light gray color. The graphical stack bar turns red when the stack usage exceeds a threshold that you can specify.

When your application is first loaded, and upon each reset, the memory for the stack area is filled with the dedicated byte value `0xCD` before the application starts executing. Whenever execution stops, the stack memory is searched from the end of the stack until a byte with a value different from `0xCD` is found, which is assumed to be how far the stack has been used. Although this is a reasonably reliable way to track stack usage, there is no guarantee that a stack overflow is detected. For example, a stack *can* incorrectly grow outside its bounds, and even modify memory outside the stack range,

without actually modifying any of the bytes near the stack range. Likewise, your application might modify memory within the stack range by mistake. Furthermore, the Stack window cannot detect a stack overflow when it happens, but can only detect the signs it leaves behind.

Note: The size and location of the stack is retrieved from the definition of the segment holding the stack, typically `CSTACK`, made in the linker command file. If you, for some reason, modify the stack initialization made in the system startup code, `cstartup`, you should also change the segment definition in the linker command file accordingly; otherwise the Stack window cannot track the stack usage. To read more about this, see the *MSP430 IAR C/C++ Compiler Reference Guide*.

When the stack bar is enabled, the functionality needed to detect and warn about stack overflows is also enabled, see *Stack page*, page 301.

The Stack window columns

The main part of the window displays the contents of stack memory in the following columns:

Column	Description
Location	Displays the location in memory. The addresses are displayed in increasing order. The address referenced by the stack pointer, in other words the top of the stack, is highlighted in a green color.
Data	Displays the contents of the memory unit at the given location. From the Stack window context menu, you can select how the data should be displayed; as a 1-, 2-, or 4-byte group of data.
Variable	Displays the name of a variable, if there is a local variable at the given location. Variables are only displayed if they are declared locally in a function, and located on the stack and not in registers.
Value	Displays the value of the variable that is displayed in the **Variable** column.
Frame	Displays the name of the function the call frame corresponds to.

Table 98: Stack window columns

The Stack window context menu

The following context menu is available if you right-click in the Stack window:

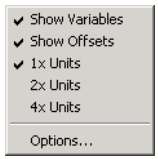


Figure 178: Stack window context menu

The following commands are available in the context window:

Show variables	Separate columns named **Variables**, **Value**, and **Frame** are displayed in the Stack window. Variables located at memory addresses listed in the Stack window are displayed in these columns.
Show offsets	When this option is selected, locations in the **Location** column are displayed as offsets from the stack pointer. When deselected, locations are displayed as absolute addresses.
1x Bytes	The data in the **Data** column is displayed as single bytes.
2x Bytes	The data in the **Data** column is displayed as 2-byte groups.
4x Bytes	The data in the **Data** column is displayed as 4-byte groups.
Options	Opens the **IDE Options** dialog box where you can set options specific to the Stack window, see *Stack page*, page 301.

LCD WINDOW

The LCD window—available from the **View** menu—simulates a 7- or 14-segments LCD display.

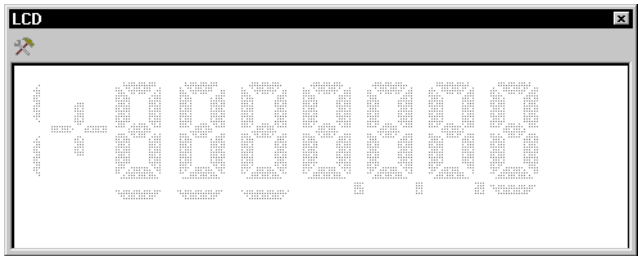


Figure 179: LCD window

LCD Settings dialog box

Click the **Settings** button in the LCD window to display the **LCD Settings** dialog box.

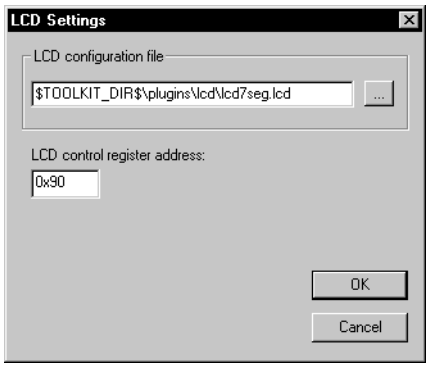


Figure 180: LCD Settings dialog box

These are the available settings:

Setting	Description
LCD configuration file	Selects the LCD display to simulate. Available displays are a 7 segment display and a 14 segment display.
LCD control register address	Sets up the address to the LCD control register.

Table 99: LCD window settings

C-SPY menus

In addition to the menus available in the development environment, the **Debug** menu is available when C-SPY is running.

Additional menus will be available depending on which C-SPY driver you are using. For information about driver-specific menus, see the online help system available from the **Help** menu for information about driver-specific documentation.

DEBUG MENU

The **Debug** menu provides commands for executing and debugging your application. Most of the commands are also available as toolbar buttons.

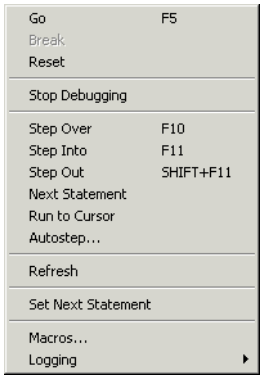


Figure 181: Debug menu

	Menu Command		Description
	Go	F5	Executes from the current statement or instruction until a breakpoint or program exit is reached.
	Break		Stops the application execution.
	Reset		Resets the target processor.
	Stop Debugging		Stops the debugging session and returns you to the project manager.
	Step Over	F10	Executes the next statement or instruction, without entering C or C++ functions or assembler subroutines.
	Step Into	F11	Executes the next statement or instruction, entering C or C++ functions or assembler subroutines.
	Step Out	SHIFT+F11	Executes from the current statement up to the statement after the call to the current function.
	Next Statement		If stepping into and out of functions is unnecessarily slow, use this command to step directly to the next statement.
	Run to Cursor		Executes from the current statement or instruction up to a selected statement or instruction.

Table 100: Debug menu commands

Menu Command	Description
Autostep	Displays the **Autostep settings** dialog box which lets you customize and perform autostepping.
Refresh	Refreshes the contents of the Memory, Register, Watch, and Locals windows.
Set Next Statement	Moves the program counter directly to where the cursor is, without executing any source code. Note, however, that this creates an anomaly in the program flow and might have unexpected effects.
Macros	Displays the **Macro Configuration** dialog box to allow you to list, register, and edit your macro files and functions.
Logging>Set Log file	Displays a dialog box to allow you to log input and output from C-SPY to a file. You can select the type and the location of the log file. You can choose what you want to log: errors, warnings, system information, user messages, or all of these.
Logging>Set Terminal I/O Log file	Displays a dialog box to allow you to log terminal input and output from C-SPY to a file. You can select the destination of the log file.

Table 100: Debug menu commands (Continued)

Autostep settings dialog box

In the **Autostep settings** dialog box—available from the **Debug** menu—you can customize autostepping.

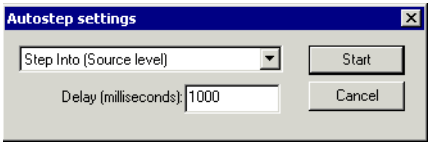


Figure 182: Autostep settings dialog box

The drop-down menu lists the available step commands.

The **Delay** text box lets you specify the delay between each step.

Macro Configuration dialog box

In the **Macro Configuration** dialog box—available by choosing **Debug>Macros**—you can list, register, and edit your macro files and functions.

Macro functions that have been registered using the dialog box will be deactivated when you exit the debug session, and will not automatically be registered at the next debug session.

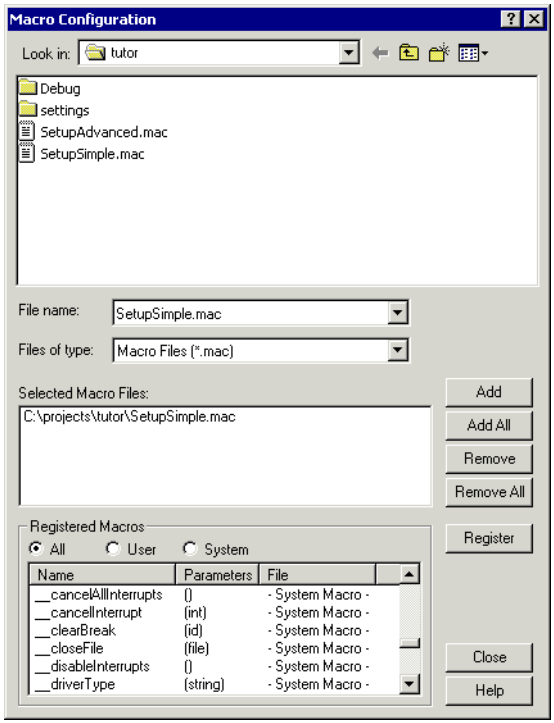


Figure 183: Macro Configuration dialog box

Registering macro files

Select the macro files you want to register in the file selection list, and click **Add** or **Add All** to add them to the **Selected Macro Files** list. Conversely, you can remove files from the **Selected Macro Files** list using **Remove** or **Remove All**.

Once you have selected the macro files you want to use click **Register** to register them, replacing any previously defined macro functions or variables. Registered macro functions are displayed in the scroll window under **Registered Macros**. Note that system macros cannot be removed from the list, they are always registered.

Listing macro functions

Selecting **All** displays all macro functions, selecting **User** displays all user-defined macros, and selecting **System** displays all system macros.

Clicking on either **Name** or **File** under **Registered Macros** displays the column contents sorted by macro names or by file. Clicking a second time sorts the contents in the reverse order.

Modifying macro files

Double-clicking a user-defined macro function in the **Name** column automatically opens the file in which the function is defined, allowing you to modify it, if needed.

Log File dialog box

The **Log File** dialog box—available by choosing **Debug>Logging>Set Log File**—allows you to log output from C-SPY to a file.

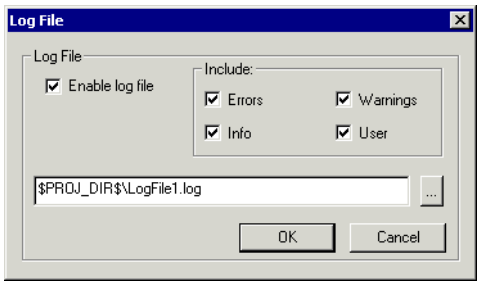


Figure 184: Log File dialog box

Enable or disable logging to the file with the **Enable Log file** check box.

The information printed in the file is by default the same as the information listed in the Log window. To change the information logged, use the **Include** options:

Option	Description
Errors	C-SPY has failed to perform an operation.
Warnings	A suspected error.
Info	Progress information about actions C-SPY has performed.
User	Printouts from C-SPY macros, that is, your printouts using the `__message` statement.

Table 101: Log file options

Click the browse button, to override the default file type and location of the log file. Click **Save** to select the specified file—the default filename extension is `log`.

Terminal I/O Log File dialog box

The **Terminal I/O Log Files** dialog box—available by choosing **Debug>Logging**—allows you to select a destination log file, and to log terminal I/O input and output from C-SPY to this file.

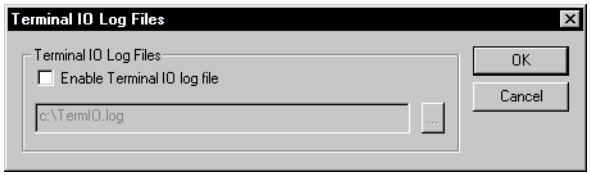


Figure 185: Terminal I/O Log File dialog box

Click the browse button to open a standard **Save As** dialog box. Click **Save** to select the specified file—the default filename extension is `log`.

General options

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

For information about how options can be set, see *Setting options*, page 89.

Target

The **Target** options specify the device, the size of the floating-point type `double`, whether position-independent code should be generated, whether code for the hardware multiplier unit should be generated, and if the project is an assembler-only project.

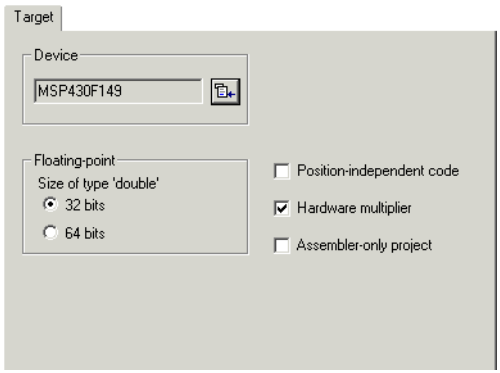


Figure 186: Target options

DEVICE

Use the drop-down list to select the device for which you will build your application. The choice of device controls which linker command file and device description file that will be used.

FLOATING-POINT

The compiler represents floating-point values by 32- and 64-bit numbers in standard IEEE 754 format. The **Size of type 'double'** option specifies the size of the type `double`. Choose between:

32 bits (default) The data type `double` is represented by the 32-bit floating-point format.

64 bits The data type `double` is represented by the 64-bit floating-point format.

For more details about the floating-point format, see the *MSP430 IAR C/C++ Compiler Reference Guide.*

POSITION-INDEPENDENT CODE

Select normal or position-independent code generation. Note that position-independent code will lead to a rather large overhead in code size. For more details about position-independent code, see *MSP430 IAR C/C++ Compiler Reference Guide.*

HARDWARE MULTIPLIER

Generates code for the MSP430 hardware multiplier peripheral unit. The option is only enabled when you have chosen a device containing the hardware multiplier from the **Device** drop-down list.

ASSEMBLER-ONLY PROJECT

Use this option if your project only contains assembler source files. The option will make the necessary settings required for an assembler only project, for instance, disabling the use of a C or C++ runtime library and the cstartup system. The **Run to** option will be disabled.

Output

With the **Output** options you can specify the type of output file—**Executable** or **Library**. You can also specify the destination directories for executable files, object files, and list files.

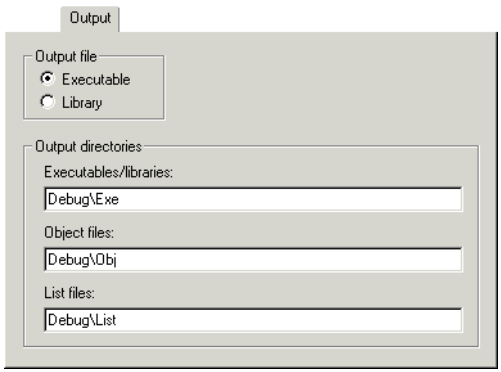


Figure 187: Output options

OUTPUT FILE

Use these options to choose the type of output file. Choose between:

Executable (default) — As a result of the build process, the XLINK linker will create an *application* (an executable output file). When this option is selected, linker options will be available in the **Options** dialog box. Before you create the output you should set the appropriate linker options.

Library — As a result of the build process, the XAR library builder will create a *library file*. When this option is selected, XAR library builder options will be available in the **Options** dialog box, and **Linker** will disappear from the list of categories. Before you create the library you can set the XAR options.

OUTPUT DIRECTORIES

Use these options to specify paths to destination directories. Note that incomplete paths are relative to your project directory. You can specify the paths to the following destination directories:

Executables/libraries — Use this option to override the default directory for executable or library files. Type the name of the directory where you want to save executable files for the project.

Object files	Use this option to override the default directory for object files. Type 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. Type the name of the directory where you want to save list files for the project.

Library Configuration

With the **Library Configuration** options you can specify which library to use.

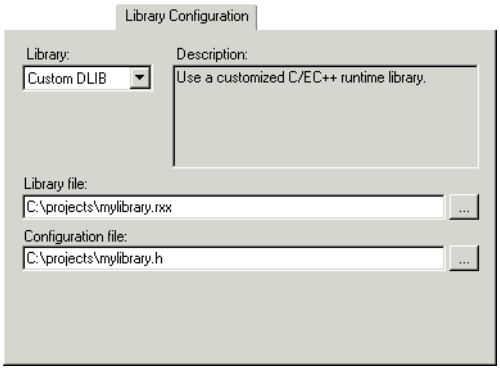


Figure 188: Library Configuration options

For information about the runtime library, library configurations, the runtime environment they provide, and the possible customizations, see *MSP430 IAR C/C++ Compiler Reference Guide*.

LIBRARY

In the **Library** drop-down list you choose which runtime library to use. For information about available libraries, see the *MSP430 IAR C/C++ Compiler Reference Guide*.

Note: For C++ projects, you must use one of the DLIB library variants.

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

LIBRARY FILE

The **Library file** text box displays the library object file that will be used. A library object file is automatically chosen depending on your project settings.

If you have chosen **Custom DLIB** or **Custom CLIB** in the **Library** drop-down list, you must specify your own library object file.

CONFIGURATION FILE

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

Note: A library configuration file is only required for the DLIB library.

Library Options

With the options on the **Library Options** page you can choose `printf` and `scanf` formatters.

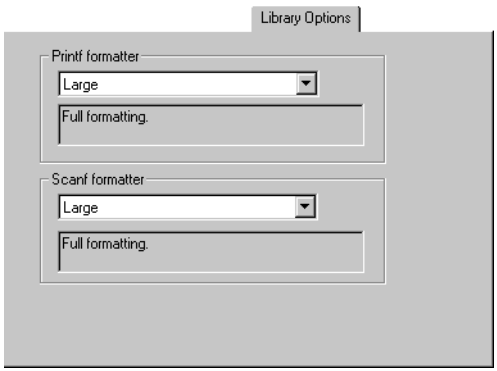


Figure 189: Library Options page

See the *MSP430 IAR C/C++ Compiler Reference Guide* for more information about the formatting capabilities.

PRINTF FORMATTER

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

- Printf formatters in the IAR DLIB Library are: **Full**, **Large**, **Small**, and **Tiny**
- Printf formatters in the IAR CLIB Library are: **Large**, **Medium**, and **Small**.

SCANF FORMATTER

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

- Scanf formatters in the IAR DLIB Library are: **Full**, **Large**, and **Small**
- Scanf formatters in the IAR CLIB Library are: **Large**, and **Medium**.

Stack/Heap

With the options on the **Stack/Heap** page you can customize the heap and stack sizes.

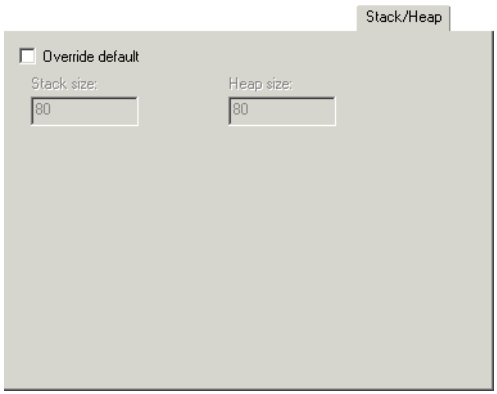


Figure 190: Stack/Heap page

OVERRIDE DEFAULT

Use this option to override the default heap and stack size settings.

STACK SIZE

Enter the required stack size in the **Stack size** text box, using decimal notation.

HEAP SIZE

Enter the required heap size in the **Heap size** text box, using decimal notation.

MISRA C

Use the options on the **MISRA C** page to control how IAR Embedded Workbench checks the source code for deviations from the MISRA C rules. The settings will be used for both the compiler and the linker.

If you want the compiler to check different sets of rules, you can override these settings in the **C/C++ Compiler** category.

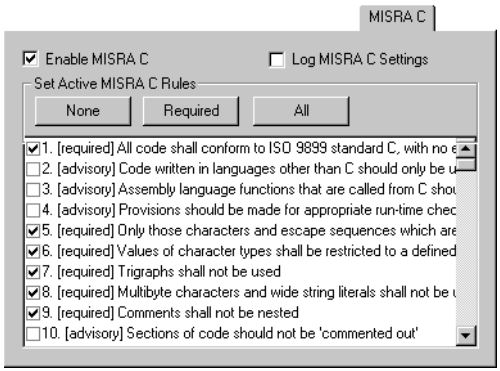


Figure 191: MISRA C general options

ENABLE MISRA C

Select this option to enable checking the source code for deviations from the MISRA C rules during compilation and linking. Only the rules selected in the scroll list will be checked.

LOG MISRA C SETTINGS

Select this option to generate a MISRA C log during compilation and linking. This is a list of the rules that are enabled—but not necessarily checked—and a list of rules that are actually checked.

SET ACTIVE MISRA C RULES

Only the rules you select in the scroll list will be checked during compilation and linking. Click one of the buttons **None**, **Required**, or **All** to select or deselect several rules with one click. The **Required** button selects all 93 rules that are categorized by the *Guidelines for the Use of the C Language in Vehicle Based Software* as *required* and deselects the rules that are categorized as *advisory*.

Compiler options

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

For information about how to set options, see *Setting options*, page 89.

Language

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

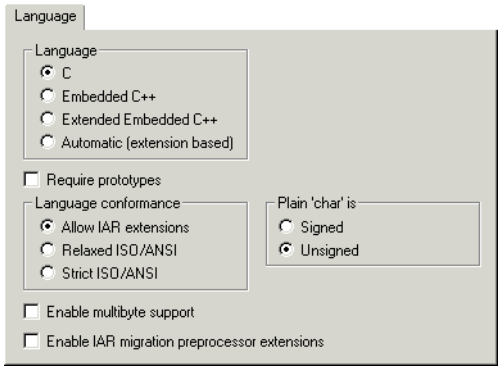


Figure 192: Compiler language options

LANGUAGE

With the **Language** options you can specify the language support you need.

For information about Embedded C++ and Extended Embedded C++, see the *MSP430 IAR C/C++ Compiler Reference Guide*.

C

By default, the MSP430 IAR C/C++ Compiler runs in ISO/ANSI C mode, in which features specific to Embedded C++ and Extended Embedded C++ cannot be utilized.

Embedded C++

In Embedded C++ mode, the compiler treats the source code as Embedded C++. This means that features specific to Embedded C++, such as classes and overloading, can be utilized.

Embedded C++ requires that a DLIB library (C/C++ library) is used.

Extended Embedded C++

In Extended Embedded C++ mode, you can take advantage of features like namespaces or the standard template library in your source code.

Extended Embedded C++ requires that a DLIB library (C/C++ library) is used.

Automatic

If you select **Automatic**, language support will be decided automatically depending on the filename extension of the file being compiled:

- Files with the filename extension `c` will be compiled as C source files
- Files with the filename extension `cpp` will be compiled as Extended Embedded C++ source files.

This option requires that a DLIB library (C/C++ library) is used.

REQUIRE PROTOTYPES

This option forces the compiler to verify that all functions have proper prototypes. Using this option means that code containing any of the following will generate an error:

- A function call of a function with no declaration, or with a Kernighan & Ritchie C declaration
- A function definition of a public function with no previous prototype declaration
- An indirect function call through a function pointer with a type that does not include a prototype.

LANGUAGE CONFORMANCE

Language extensions must be enabled for the MSP430 IAR C/C++ Compiler to be able to accept MSP430-specific keywords as extensions to the standard C or C++ language. In the IAR Embedded Workbench IDE, the option **Allow IAR extensions** is enabled by default.

The option **Relaxed ISO/ANSI** disables IAR extensions, but does not adhere to strict ISO/ANSI.

Select the option **Strict ISO/ANSI** to adhere to the strict ISO/ANSI C standard.

For details about language extensions, see the *MSP430 IAR C/C++ Compiler Reference Guide*.

PLAIN 'CHAR' IS

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.

Note: The runtime library is compiled with unsigned plain characters. If you select the radio button **Signed**, you might get type mismatch warnings from the linker as the library uses `unsigned char`.

ENABLE MULTIBYTE SUPPORT

By default, multibyte characters cannot be used in C or Embedded C++ source code. If you use this option, multibyte characters in the source code are interpreted according to the host computer's default setting for multibyte support.

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

ENABLE IAR MIGRATION PREPROCESSOR EXTENSIONS

Migration preprocessor extensions extend the preprocessor in order to ease migration of code from earlier IAR compilers. If you need to migrate code from an earlier IAR C or C++ compiler, you may want to use this option.

Note: If you use this option, not only will the compiler accept code that is not standard conformant, but it will also reject some code that *does* conform to standard.

Important! Do not depend on these extensions in newly written code. Support for them may be removed in future compiler versions.

Code

The **Code** options specify the use of registers and the stack.

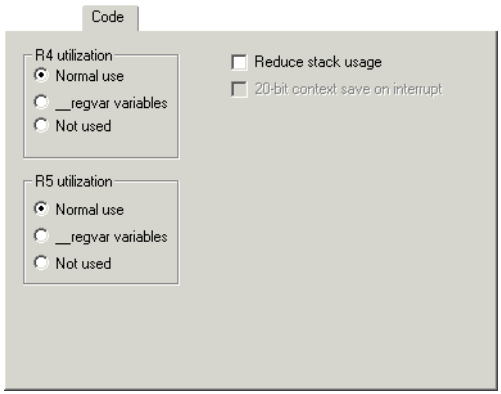


Figure 193: Compiler code options

R4 UTILIZATION

This option controls how register `R4` can be used. There are three possible settings:

- **Normal use**. This setting allows the compiler to use the register in generated code.
- **__regvar variables**. When this setting is selected, the compiler uses the register for locating global register variables declared with the extended keyword `__regvar`.
- **Not used**. If you select this setting, `R4` is locked and can be used for a special purpose by the application.

R5 UTILIZATION

This option controls how register `R5` can be used. There are three possible settings:

- **Normal use**. This setting allows the compiler to use the register in generated code.
- **__regvar variables**. When this setting is selected, the compiler uses the register for locating global register variables declared with the extended keyword `__regvar`.
- **Not used**. If you select this setting, `R5` is locked and can be used for a special purpose by the application.

REDUCE STACK USAGE

Use this option to make the compiler minimize the use of stack space at the cost of somewhat larger and slower code.

20-BIT CONTEXT SAVE ON INTERRUPT

Use this option to make all interrupt functions be treated as a `__save_reg_20` declared function without explicitly using the `__save_reg20` keyword.

This is useful if your application requires that all 20 bits of registers are preserved. The drawback is that the code will be somewhat slower.

Note: This option has no effect when compiling for the MSP430 architecture.

Optimizations

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

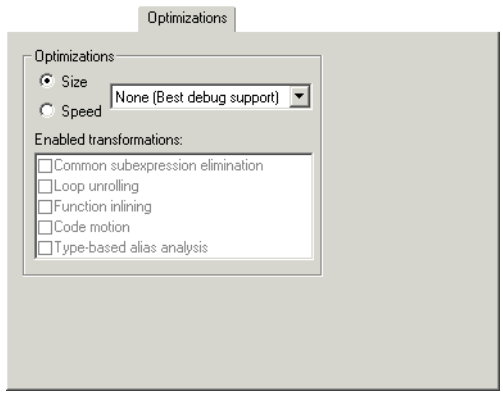


Figure 194: Compiler optimizations options

OPTIMIZATIONS

Size or speed, and level

The MSP430 IAR C/C++ Compiler supports two optimization models—size and speed—at different optimization levels.

Select the optimization model using either the **Size** or **Speed** radio button. Then choose the optimization level—None, Low, Medium, or High—from the drop-down list next to the radio buttons.

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 an absolute minimum of code.

For a list of optimizations performed at each optimization level, see the *MSP430 IAR C/C++ Compiler Reference Guide*.

Enabled transformations

The following transformations are available on different level of optimizations:

- Common subexpression elimination
- Loop unrolling
- Function inlining
- Code motion
- Type-based alias analysis.

When a transformation is available, you can enable or disable it by selecting its check box.

In a *debug* project, the transformations are by default disabled. In a *release* project, the transformations are by default enabled.

For a brief description of the transformations that can be individually disabled, see the *MSP430 IAR C/C++ Compiler Reference Guide*.

Output

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

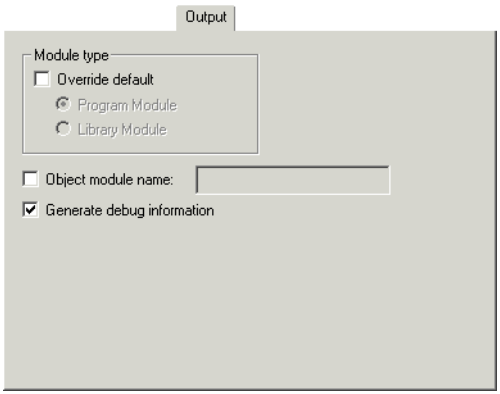


Figure 195: Compiler output options

MODULE TYPE

By default, the compiler generates *program* modules. Use this option to make a *library* module that will only be included if it is referenced in your application. Select the **Override default** check box and choose one of:

Program Module	The object file will be treated as a program module rather than as a library module.
Library Module	The object file will be treated as a library module rather than as a program module.

For information about program and library modules, and working with libraries, see the XLIB and XAR chapters in the *IAR Linker and Library Tools Reference Guide*, available from the **Help** menu.

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 type a name in the entry field.

This option is particularly useful when several modules have the same filename, because 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 selected by default. Deselect this option if you do not want the compiler to generate debug information.

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

List

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

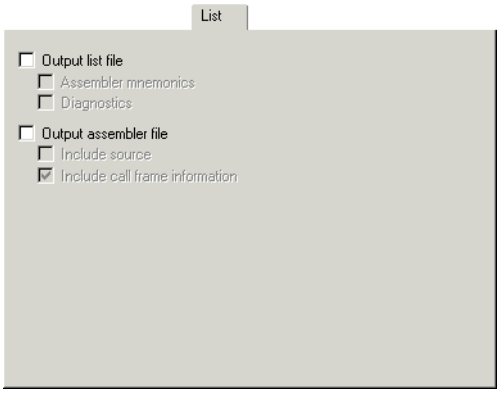


Figure 196: Compiler list file options

Normally, the compiler does not generate a list file. Select any of the following options to generate a list file or an assembler file. The list file will be saved in the `List` directory, and its filename will consist of the source filename, plus the filename extension `lst`. You can open the output files directly from the **Output** folder which is available in the Workspace window.

OUTPUT LIST FILE

Select the **Output list file** option and choose the type of information to include in the list file:

Assembler mnemonics	Includes assembler mnemonics in the list file.
Diagnostics	Includes diagnostic information in the list file.

OUTPUT ASSEMBLER FILE

Select the **Output assembler file** option and choose the type of information to include in the list file:

Include source	Includes source code in the assembler file.
Include call frame information	Includes compiler-generated information for runtime model attributes, call frame information, and frame size information.

Preprocessor

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

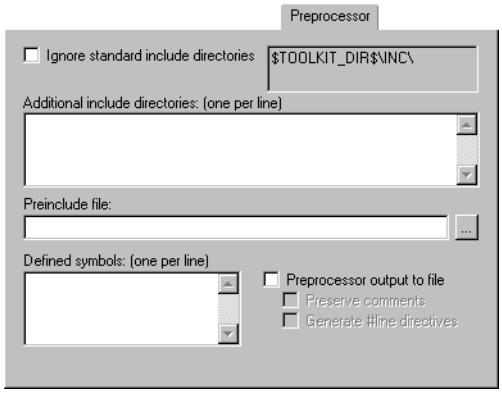


Figure 197: Compiler preprocessor options

IGNORE STANDARD INCLUDE DIRECTORIES

If you select this option, the standard include files will not be used when the project is built.

ADDITIONAL INCLUDE DIRECTORIES

The **Additional include directories** option adds a path to the list of `#include` file paths. The paths required by the product are specified by default depending on your choice of runtime library.

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

Note: Any additional directories specified using this option will be searched before the standard include directories.

To make your project more portable, use the argument variable `$TOOLKIT_DIR$` for the subdirectories of the active product and `$PROJ_DIR$` for the directory of the current project. For an overview of the argument variables, see *Argument variables summary*, page 279.

PREINCLUDE FILE

Use this option to make the compiler include the specified include file before it starts to read the source file. This is useful if you want to change something in the source code for the entire application, for instance if you want to define a new symbol.

DEFINED SYMBOLS

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

Type the symbols that you want to define for the project, for example:

```
TESTVER=1
```

Note that there should be no space around the equal sign.

The **Defined symbols** option has the same effect as a `#define` statement at the top of the source file.

For example, you might arrange your source to produce either the test or production version of your application depending on whether the symbol `TESTVER` was defined. To do this you would use include sections such as:

```
#ifdef  TESTVER
   ...  ; additional code lines for test version only
#endif
```

You would then define the symbol `TESTVER` in the Debug target but not in the Release target.

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.

Diagnostics

The **Diagnostics** options determine how diagnostics are classified and displayed. Use the diagnostics options to override the default classification of the specified diagnostics.

Note: The diagnostics cannot be suppressed for fatal errors, and fatal errors cannot be reclassified.

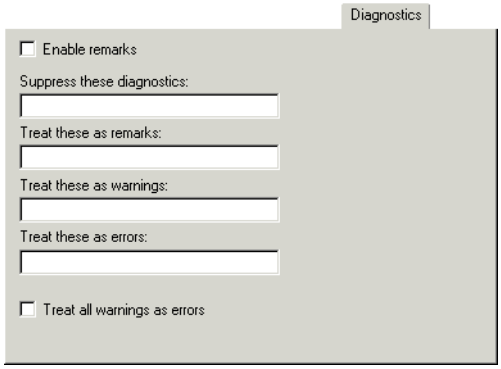


Figure 198: Compiler diagnostics options

ENABLE REMARKS

The least severe diagnostic messages are called *remarks*. A remark indicates a source code construct that might 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 might 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 C++ language rules, of such severity that object code will not be generated, and the exit code will be non-zero. Use this option to classify diagnostic messages as errors.

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

```
Pe117
```

TREAT ALL 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.

MISRA C

Use these options to override the options set on the **MISRA C** page of the **General Options** category.

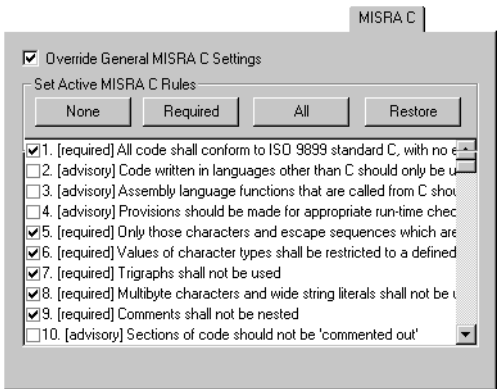


Figure 199: MISRA C compiler options

OVERRIDE GENERAL MISRA C SETTINGS

Select this option if you want the compiler to check a different selection of rules than the rules selected in the **General Options** category.

SET ACTIVE MISRA C RULES

Only the rules that have been selected in the scroll list will be checked during compilation. To select or deselect several rules with one click, click one of the buttons **None**, **Required**, **All**, or **Restore**. The **Required** button selects all 93 rules that are categorized by the *Guidelines for the Use of the C Language in Vehicle Based Software* as *required* and deselects the rules that are categorized as *advisory*. The **Restore** button restores the MISRA C settings used in the **General Options** category.

Extra Options

The **Extra Options** page provides you with a command line interface to the compiler.

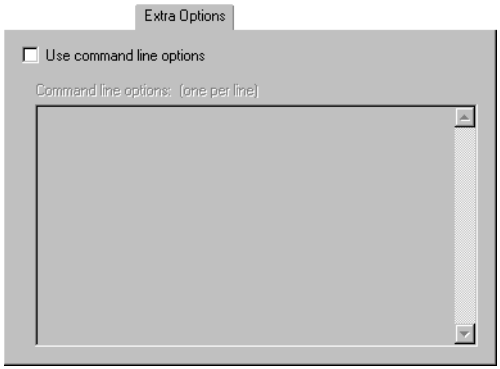


Figure 200: Extra Options page for the compiler

USE COMMAND LINE OPTIONS

Additional command line arguments for the compiler (not supported by the GUI) can be specified here.

Assembler options

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

For information about how to set options, see *Setting options*, page 89.

Language

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

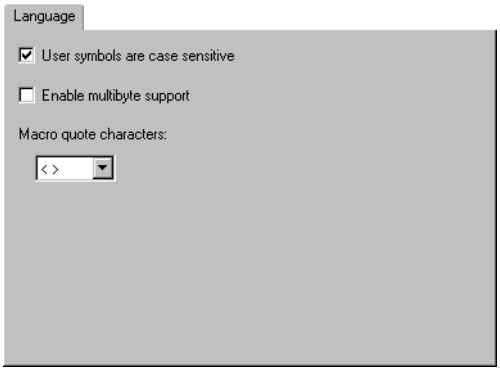


Figure 201: Assembler language options

USER SYMBOLS ARE CASE SENSITIVE

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

ENABLE MULTIBYTE SUPPORT

By default, multibyte characters cannot be used in assembler source code. If you use this option, multibyte characters in the source code are interpreted according to the host computer's default setting for multibyte support.

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

MACRO QUOTE CHARACTERS

The **Macro quote characters** 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, choose one of four types of brackets to be used as macro quote characters:

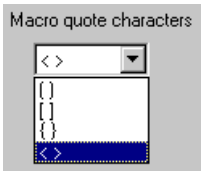


Figure 202: Choosing macro quote characters

Output

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

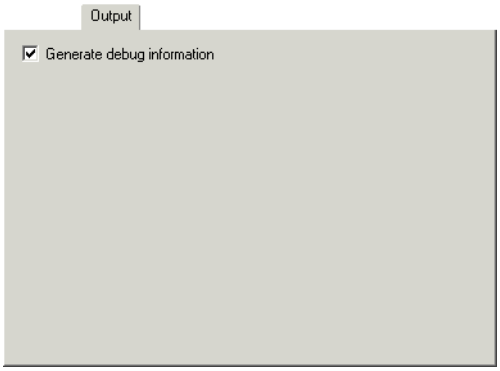


Figure 203: Assembler output options

GENERATE DEBUG INFORMATION

The **Generate debug information** option must be selected if you want to use a debugger with your application. By default, this option is selected in a Debug project, but not in a Release project.

List

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

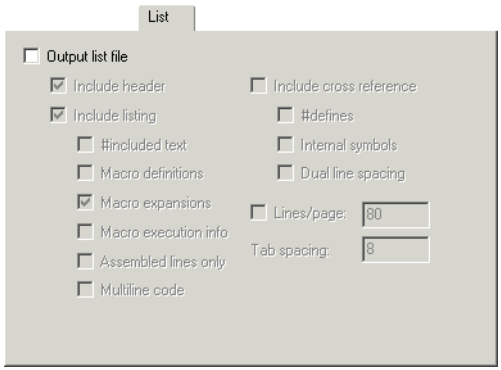


Figure 204: Assembler list file options

By default, the assembler does not generate a list file. Selecting **Output 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 Options** category; see *Output*, page 345, for additional information.

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.

INCLUDE LISTING

Use the suboptions under **Include listing** 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 assembler sections from the list file.
Multiline code	Lists the code generated by directives on several lines if necessary.

Table 102: Assembler list file options

INCLUDE 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 *MSP430 IAR Assembler Reference Guide* for details.

LINES/PAGE

The default number of lines per page is 80 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.

Preprocessor

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

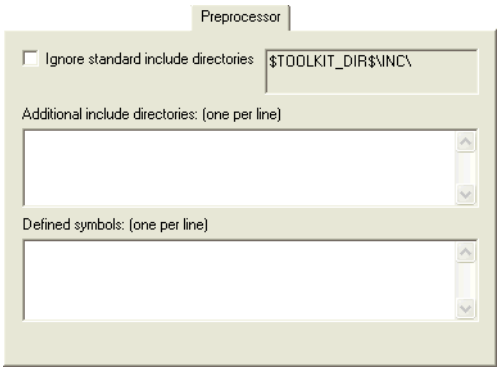


Figure 205: Assembler preprocessor options

IGNORE STANDARD INCLUDE DIRECTORIES

If you select this option, the standard include files will not be used when the project is built.

ADDITIONAL INCLUDE DIRECTORIES

The **Additional include directories** option adds paths to the list of `#include` file paths. The path required by the product is specified by default.

Type 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 `$TOOLKIT_DIR$` for the subdirectories of the active product and `$PROJ_DIR$` for the directory of the current project. For an overview of the argument variables, see Table 67, *Argument variables*, page 279.

See the *MSP430 IAR Assembler Reference Guide* for information about the `#include` directive.

Note: By default the assembler also searches for `#include` files in the paths specified in the `A430_INC` environment variable. We do not, however, recommend that you use environment variables in the IAR Embedded Workbench IDE.

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.

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

- For example, you might arrange your source to produce either the test or production version of your application depending on whether the symbol `TESTVER` was defined. To do this you would use include sections such as:

  ```
  #ifdef  TESTVER
  ...  ; additional code lines for test version only
  #endif
  ```

  You would then define the symbol `TESTVER` in the Debug target but not in the Release target.

- Alternatively, your source might use a variable that you need to change often, for example `FRAMERATE`. You would leave the variable undefined in the source and use this option to specify a value for the project, for example `FRAMERATE=3`.

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

Diagnostics

Use the **Diagnostics** options to disable or enable individual warnings or ranges of warnings.

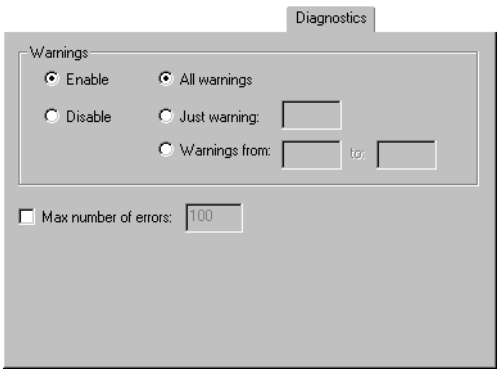


Figure 206: Assembler diagnostics options

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 **Diagnostics** options allow you to enable only some warnings, or to disable all or some warnings.

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

For additional information about assembler warnings, see the *MSP430 IAR Assembler Reference Guide*.

MAX NUMBER OF ERRORS

By default, the maximum number of errors reported by the assembler is 100. This option allows you to decrease or increase this number, for example, to see more errors in a single assembly.

Extra Options

The **Extra Options** page provides you with a command line interface to the assembler.

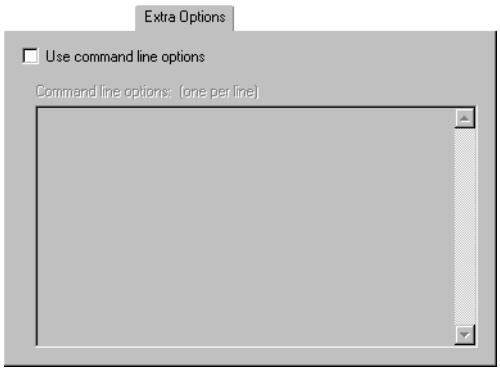


Figure 207: Extra Options page for the assembler

USE COMMAND LINE OPTIONS

Additional command line arguments for the assembler (not supported by the GUI) can be specified here.

Custom build options

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

For information about how to set options, see *Setting options*, page 89.

Custom Tool Configuration

To set custom build options in the IAR Embedded Workbench IDE, choose **Project>Options** to display the **Options** dialog box. Then select **Custom Build** in the **Category** list to display the **Custom Tool Configuration** page:

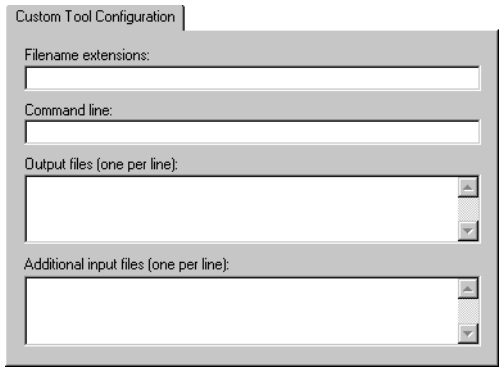


Figure 208: Custom tool options

In the **Filename extensions** text box, specify the filename extensions for the types of files that are to be processed by this custom tool. You can enter several filename extensions. Use commas, semicolons, or blank spaces as separators.

In the **Command line** text box, type the command line for executing the external tool.

In the **Output files** text box, enter the output files from the external tool.

If there are any additional files that are used by the external tool during the building process, these files should be added in the **Additional input files** text box. If these additional input files, so-called dependency files, are modified, the need for a rebuild is detected.

For an example, see *Extending the tool chain*, page 93.

Build actions options

This chapter describes the options for pre-build and post-build actions available in the IAR Embedded Workbench® IDE.

For information about how to set options, see *Setting options*, page 89.

Build Actions Configuration

To set options for pre-build and post-build actions in the IAR Embedded Workbench IDE, choose **Project>Options** to display the **Options** dialog box. Then select **Build Actions** in the **Category** list to display the **Build Actions Configuration** page.

These options apply to the whole build configuration, and cannot be set on groups or files.

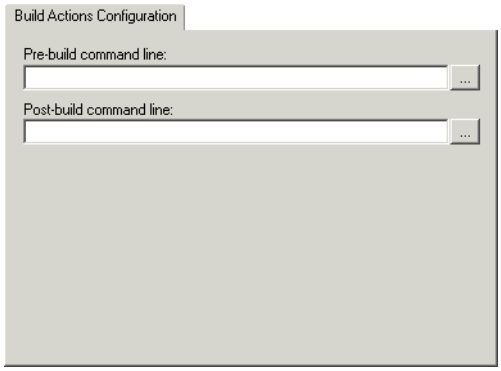


Figure 209: Build actions options

PRE-BUILD COMMAND LINE

Type a command line to be executed directly before a build; a browse button for locating an extended command line file is available for your convenience. The commands will not be executed if the configuration is already up-to-date.

POST-BUILD COMMAND LINE

Type a command line to be executed directly after each successful build; a browse button is available for your convenience. The commands will not be executed if the configuration was up-to-date. This is useful for copying or post-processing the output file.

Linker options

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

For information about how to set options, see *Setting options*, page 89.

Note that the XLINK command line options that are used for defining segments in a linker command file are described in the *IAR Linker and Library Tools Reference Guide*.

Output

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

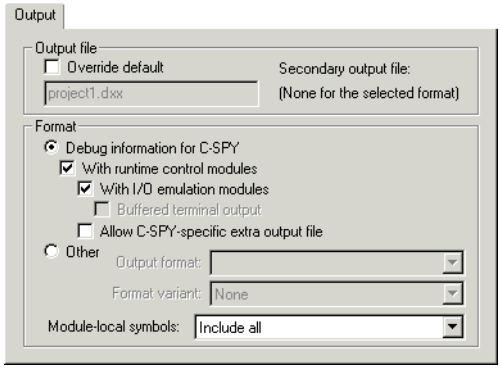


Figure 210: XLINK output file options

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 project name with a filename extension. The filename extension depends on which output format you choose. If you choose **Debug information for C-SPY**, the output file will have the filename extension `d43`.

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 filename extension other than the default.

FORMAT

The output options determine the format of the output file generated by the IAR XLINK Linker. The output file is used as input to either a debugger or as input for programming the target system. The IAR Systems proprietary output format is called UBROF, Universal Binary Relocatable Object Format.

The default output settings are:

- In a *debug* project, **Debug information for C-SPY**, **With runtime control modules**, and **With I/O emulation modules** are selected by default
- In a *release* project, msp430-txt is selected by default, which is an output format without debug information suitable for target download.

Note: For debuggers other than C-SPY®, check the user documentation supplied with that debugger for information about which format/variant should be used.

Debug information for C-SPY

This option creates a UBROF output file, with a `d43` filename extension, to be used with the IAR C-SPY Debugger.

With runtime control modules

This option produces the same output as the **Debug information for C-SPY** option, but also includes debugger support for handling program abort, exit, and assertions. Special C-SPY variants for the corresponding library functions are linked with your application. For more information about the debugger runtime interface, see the *MSP430 IAR C/C++ Compiler Reference Guide.*

With I/O emulation modules

This option produces the same output as the **Debug information for C-SPY** and **With runtime control modules** options, but also includes debugger support for I/O handling, which means that `stdin` and `stdout` are redirected to the Terminal I/O window, and that it is possible to access files on the host computer during debugging.

For more information about the debugger runtime interface, see the *MSP430 IAR C/C++ Compiler Reference Guide.*

Buffered terminal output

During program execution in C-SPY, instead of instantly printing each new character to the C-SPY Terminal I/O window, this option will buffer the output. This option is useful when using debugger systems that have slow communication.

Allow C-SPY-specific extra output file

Use this option to enable the options available on the **Extra Output** page.

If you choose any of the options **With runtime control modules** or **With I/O emulation modules**, the generated output file will contain dummy implementations for certain library functions, such as `putchar`, and extra debug information required by C-SPY to handle those functions. In this case, the options available on the **Extra Output** page are disabled, which means you cannot generate an extra output file. The reason is that the extra output file would still contain the dummy functions, but would lack the required extra debug information, and would therefore normally be useless.

However, for *some* debugger systems, two output files from the same build process are required—one with the required debug information, and one that you can burn to your hardware before debugging. This is useful when you want to debug code that is located in non-volatile memory. In this case, you must choose the **Allow C-SPY-specific extra output file** option to make it possible to generate an extra output file.

Other

Use this option to generate output other than those generated by the options **Debug information for C-SPY**, **With runtime control modules**, and **With I/O emulation modules**.

Use the **Output format** drop-down list to select the appropriate output. If applicable, use **Format variant** to select variants available for some of the output formats. The alternatives depend on the output format chosen.

When you specify the **Other>Output format** option as either **debug (ubrof)**, or **ubrof**, a UBROF output file with the filename extension `dbg` will be created. The generated output file will not contain debugging information for simulating facilities such as stop at program exit, long jump instructions, and terminal I/O. If you need support for these facilities during debugging, use the **Debug information for C-SPY**, **With runtime control modules**, and **With I/O emulation modules** options, respectively.

For more information, see the *IAR Linker and Library Tools 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.

Extra Output

The **Extra Output** options are used for generating an extra output file and for specifying its format.

Note: If you have chosen any of the options **With runtime control modules** or **With I/O emulation modules** available on the **Output** page, you must also choose the option **Allow C-SPY-specific extra output file** to enable the **Extra Output** options.

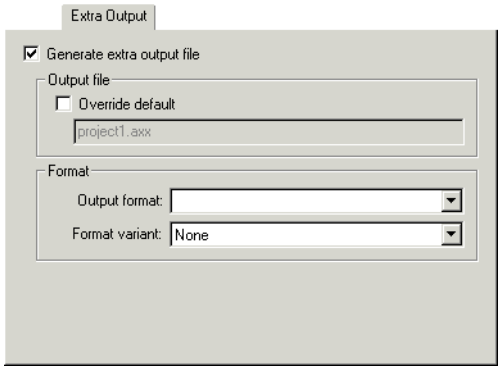


Figure 211: XLINK extra output file options

Use the **Generate extra output file** option to generate an additional output file from the build process.

Use the **Override default** option to override the default file name. If a name is not specified, the linker will use the project name and a filename extension which depends on the output format you choose.

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).

Use the **Output format** drop-down list to select the appropriate output. If applicable, use **Format variant** to select variants available for some of the output formats. The alternatives depend on the output format you have chosen.

When you specify the **Output format** option as either **debug (ubrof)**, or **ubrof**, a UBROF output file with the filename extension `dbg` will be created.

#define

You can define symbols with the **#define** option.

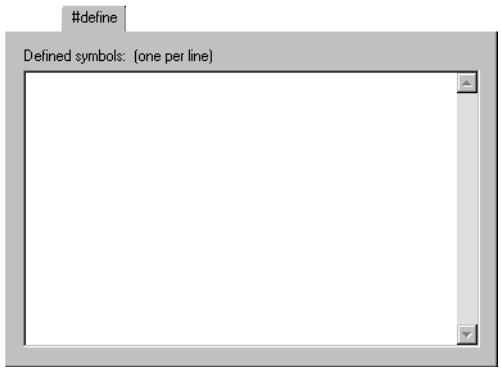


Figure 212: XLINK defined symbols options

DEFINE SYMBOL

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

Type the symbols that you want to define for the project, for example:

```
TESTVER=1
```

Note that there should be no space around the equal sign.

Any number of symbols can be defined in a linker 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.

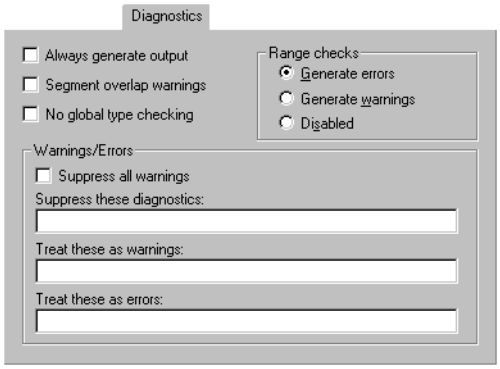


Figure 213: XLINK diagnostics options

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 application 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 IDE:

Option	Description
Generate errors	An error message is generated
Generate warnings	Range errors are treated as warnings
Disabled	Disables the address range checking

Table 103: XLINK range check options

If an address is relocated outside address range of the target CPU —code, external data, or internal data address—an error message is generated. This usually indicates an error in an assembler 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 might still be correct. The **Warnings/Errors** options allow you to suppress or enable all warnings, and to change the severity classification of errors and warnings.

Refer to the *IAR Linker and Library Tools 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

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

For example, to suppress the warnings `w117` and `w177`, type `w117,w177`.

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, type `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, type `w26`.

List

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

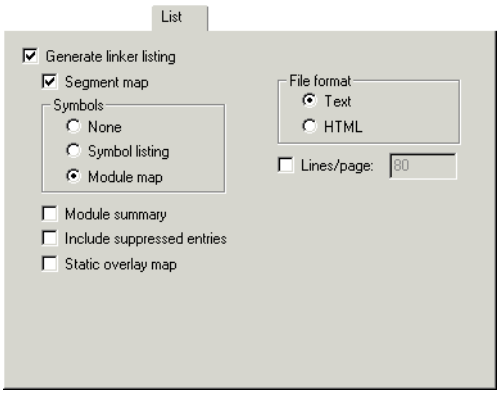


Figure 214: XLINK list file options

GENERATE LINKER LISTING

Causes the linker to generate a listing and send it to the file `projectname.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.
Module map	A list of all segments, local symbols, and entries (public symbols) for every module in the application.

Table 104: XLINK list file options

Module summary

Use the **Module summary** option to generate a summary of the contributions to the total memory use from each module.

Only modules with a contribution to memory use are listed.

Include suppressed entries

Use this option to include all segment parts in a linked module in the list file, not just the segment parts that were included in the output. This makes it possible to determine exactly which entries that were not needed.

Static overlay map

If the compiler uses static overlay, this option includes a listing of the static overlay system in the list file. Read more about static overlay maps in the *IAR Linker and Library Tools Reference Guide.*

File format

The following options are available:

Option	Description
Text	Plain text file
HTML	HTML format, with hyperlinks

Table 105: XLINK list file format options

Lines/page

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

Config

With the **Config** options you can specify the path and name of the linker command file, override the default program entry, and specify the library search path.

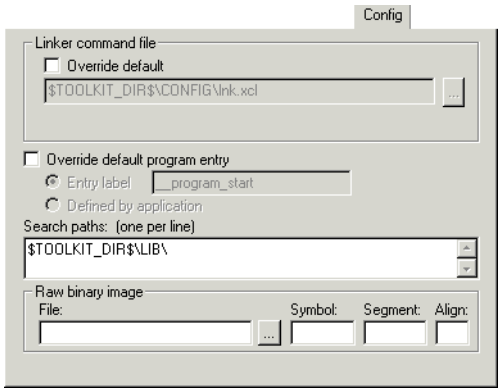


Figure 215: XLINK config options

LINKER COMMAND FILE

A default linker command file is selected automatically for the chosen **Target** settings in the **General Options** category. You can override this by selecting the **Override default** option, 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.

OVERRIDE DEFAULT PROGRAM ENTRY

By default, the program entry is the label `__program_start`. The linker will make sure that a module containing the program entry label is included, and that the segment part containing the label is not discarded.

The default program handling can be overridden by selecting **Override default program entry**.

Selecting the option **Entry label** will make it possible to specify a label other than `__program_start` to use for the program entry.

Selecting the option **Defined by application** will disable the use of a start label. The linker will, as always, include all program modules, and enough library modules to satisfy all symbol references, keeping all segment parts that are marked with the `root` attribute or that are referenced, directly or indirectly, from such a segment part.

SEARCH PATHS

The **Search paths** option specifies the names of the directories which XLINK will search if it fails to find the object files to be linked in the current working directory. Add the full paths of any further directories that you want XLINK to search.

The paths required by the product are specified by default, depending on your choice of runtime library. If the box is left empty, XLINK searches for object files only in the current working directory.

Type the full file path of your `#include` files. To make your project more portable, use the argument variable `$TOOLKIT_DIR$` for the subdirectories of the active product and `$PROJ_DIR$` for the directory of the current project. For an overview of the argument variables, see *Argument variables summary*, page 279.

RAW BINARY IMAGE

Use the **Raw binary image** options to link pure binary files in addition to the ordinary input files. Use the text boxes to specify the following parameters:

File	The pure binary file you want to link.
Symbol	The symbol defined by the segment part where the binary data is placed.
Segment	The segment where the binary data will be placed.
Align	The alignment of the segment part where the binary data is placed.

The entire contents of the file are placed in the segment you specify, which means it can only contain pure binary data, for example, the raw-binary output format. The segment part where the contents of the specified file is placed, is only included if the specified symbol is required by your application. Use the `-g` linker option if you want to force a reference to the symbol. Read more about single output files and the `-g` option in the *IAR Linker and Library Tools Reference Guide*.

Processing

With the **Processing** options you can specify details about how the code is generated.

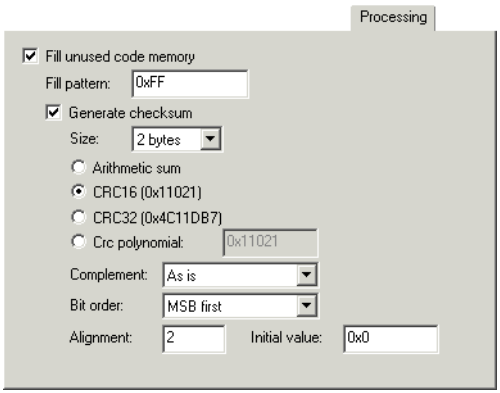


Figure 216: XLINK processing options

FILL UNUSED CODE MEMORY

Use **Fill unused code memory** to fill all gaps between segment parts introduced by the linker with the value you enter. 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.

Fill pattern

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

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

Algorithms

One of the following algorithms can be used:

Algorithms	Description
Arithmetic sum	Simple arithmetic sum
CRC16	CRC16, generating polynomial 0x11021 (default)
CRC32	CRC32, generating polynomial 0x104C11DB7
Crc polynomial	CRC with a generating polynomial of the value you enter

Table 106: XLINK checksum algorithms

Complement

Use the **Complement** drop-down list to specify the one's complement or two's complement.

Bit order

By default it is the most significant 1, 2, or 4 bytes (**MSB**) of the result that will be output, in the natural byte order for the processor. Choose **LSB** from the **Bit order** drop-down list if you want the least significant bytes to be output.

Alignment

Use this option to specify an optional alignment for the checksum. If you do not specify an alignment explicitly, an alignment of 2 is used.

Initial value

Use this option to specify the initial value of the checksum. This is useful if the microcontroller you are using has its own checksum calculation and you want that calculation to correspond to the calculation performed by XLINK.

THE CHECKSUM CALCULATION

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 `__checksum` label in the `CHECKSUM` segment. This segment must be placed using the segment placement options like any other segment.

For additional information about segment control, see the *IAR Linker and Library Tools Reference Guide*.

Extra Options

The **Extra Options** page provides you with a command line interface to the linker.

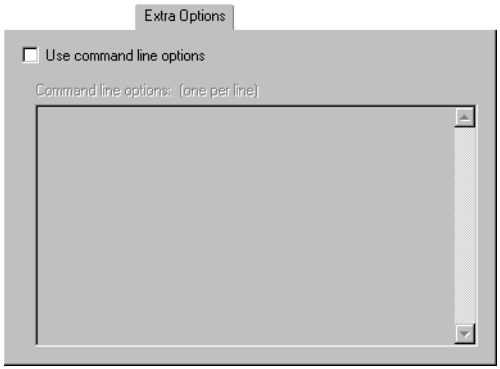


Figure 217: Extra Options page for the linker

USE COMMAND LINE OPTIONS

Additional command line arguments for the linker (not supported by the GUI) can be specified here.

Library builder options

This chapter describes the XAR Library builder options available in the IAR Embedded Workbench® IDE.

For information about how to set options, see *Setting options*, page 89.

Output

XAR options are not available by default. Before you can set XAR options in the IAR Embedded Workbench IDE, you must add the XAR Library Builder tool to the list of categories. Choose **Project>Options** to display the **Options** dialog box, and select the **General Options** category. On the **Output** page, select the **Library** option.

If you select the **Library** option, **Library Builder** appears as a category in the **Options** dialog box. As a result of the build process, the XAR Library Builder will create a library output file. Before you create the library you can set the XAR options.

To set XAR options, select **Library Builder** from the category list to display the XAR options.

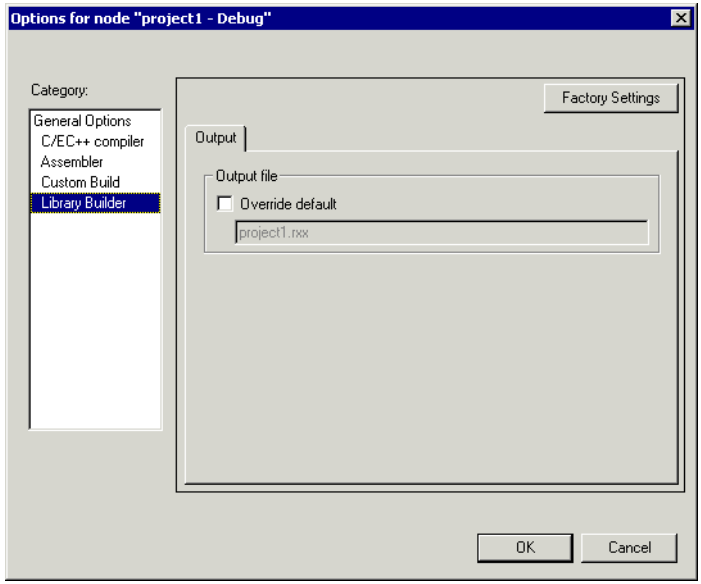


Figure 218: XAR output options

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

The **Output file** option overrides the default name of the output file. Enter a new name in the **Override default** text box.

Debugger options

This chapter describes the C-SPY® options available in the IAR Embedded Workbench® IDE.

For information about how to set options, see *Setting options*, page 89.

In addition, options specific to the C-SPY FET debugger are described in the chapter *C-SPY® FET-specific debugging*.

Setup

To set C-SPY options in the IAR Embedded Workbench IDE, choose **Project>Options** to display the **Options** dialog box. Then select **Debugger** in the **Category** list. The **Setup** page contains the generic C-SPY options.

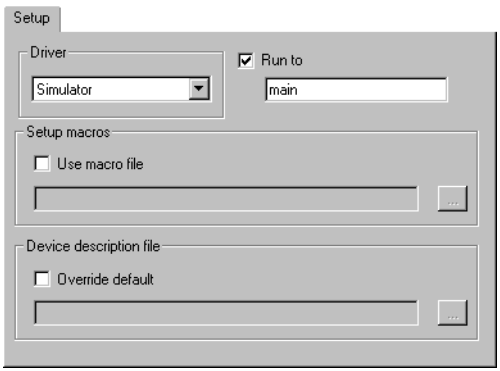


Figure 219: Generic C-SPY options

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

The **Setup** options specify the C-SPY driver, the setup macro file, and device description file to be used, and which default source code location to run to.

DRIVER

Selects the appropriate driver for use with C-SPY, the **Simulator** driver or the **FET Debugger** driver.

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

RUN TO

Use this option to specify a location you want C-SPY to run to when you start the debugger and after a reset.

The default location to run to is the `main` function. Type the name of the location if you want C-SPY to run to a different location. You can specify assembler labels or whatever can be evaluated to such, for example function names.

If you leave the check-box empty, the program counter will contain the regular hardware reset address at each reset.

SETUP MACROS

To register the contents of a setup macro file in the C-SPY startup sequence, select **Use macro file** and enter the path and name of the setup file, for example `SetupSimple.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 a device description file that contains device-specific information.

For details about the device description file, see *Device description file*, page 113.

Device description files for each MSP430 device are provided in the directory `430\config` and have the filename extension `ddf`.

Extra Options

The **Extra Options** page provides you with a command line interface to the C-SPY debugger.

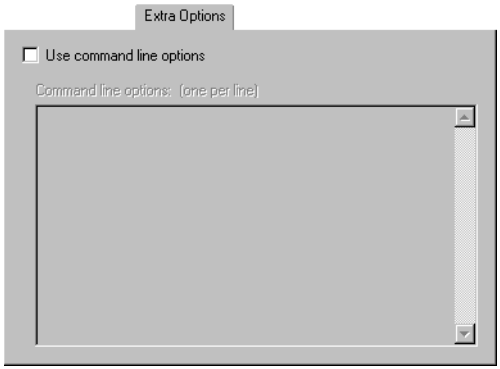


Figure 220: Extra Options page for the C-SPY debugger

USE COMMAND LINE OPTIONS

Additional command line arguments for the C-SPY debugger (not supported by the GUI) can be specified here.

Plugins

On the **Plugins** page you can specify C-SPY plugin modules to be loaded and made available during debug sessions. Plugin modules can be provided by IAR Systems, as well as by third-party suppliers. Contact your software distributor or IAR representative, or visit the IAR Systems web site, for information about available modules.

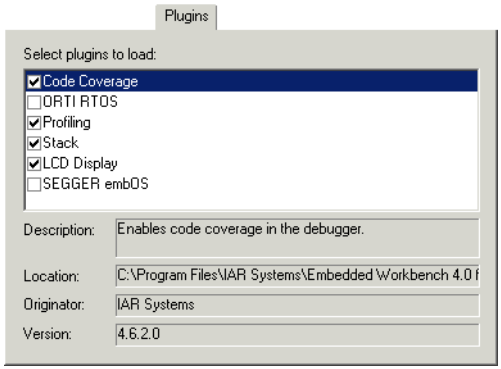


Figure 221: C-SPY plugin options

By default, **Select plugins to load** lists the plugin modules delivered with the product installation.

If you have any C-SPY plugin modules delivered by any third-party vendor, these will also appear in the list.

The `common\plugins` directory is intended for generic plugin modules. The `430\plugins` directory is intended for target-specific plugin modules.

C-SPY® macros reference

This chaptersection gives reference information about the C-SPY macros. First a syntax description of the macro language is provided. Then, the available setup macro functions and the pre-defined system macros are summarized. Finally, each system macro is described in detail.

The macro language

The syntax of the macro language is very similar to the C language. There are *macro statements*, which are similar to C statements. You can define *macro functions*, with or without parameters and return value. You can use built-in system macros, similar to C library functions. Finally, you can define global and local *macro variables*. You can collect your macro functions in a *macro file* (filename extension `mac`).

MACRO FUNCTIONS

C-SPY macro functions consist 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 function, and macro functions can return a value on exit.

A C-SPY macro has the following form:

```
macroName (parameterList)
{
  macroBody
}
```

where *parameterList* is a list of macro parameters separated by commas, and *macroBody* is any series of C-SPY variable definitions and C-SPY statements.

Type checking is neither performed on the values passed to the macro functions nor on the return value.

PREDEFINED SYSTEM MACRO FUNCTIONS

The macro language also includes a wide set of predefined system macro functions (built-in functions), similar to C library functions. For detailed information about each system macro, see .

MACRO VARIABLES

A macro variable is a variable defined and allocated outside your application space. It can then be used in a C-SPY expression. For detailed information about C-SPY expressions, see the chapter *C-SPY expressions*, page 123.For details about C-SPY expressions, see the *MSP430 IAR Embedded Workbench® IDE User Guide*.

The syntax for defining one or more macro variables is:

```
__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 it exists throughout the whole debugging session. A macro variable defined within a macro body is created when its definition is executed and destroyed on return from the macro.

By default, macro variables are treated as signed integers and initialized to 0. When a C-SPY variable is assigned a value in an expression, it also acquires the type of that expression. 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`.

Table 107: Examples of C-SPY macro variables

In case of a name conflict between a C symbol and a C-SPY macro variable, C-SPY macro variables have a higher precedence than C variables. Note that macro variables are allocated on the debugger host and do not affect your application.

Macro strings

In addition to C types, macro variables can hold values of *macro strings*. Note that macro strings differ from C language strings.

When you write a string literal, such as `"Hello!"`, in a C-SPY expression, the value is a macro string. It is not a C-style character pointer `char*`, because `char*` must point to a sequence of characters in target memory and C-SPY cannot expect any string literal to actually exist in target memory.

You can manipulate a macro string using a few built-in macro functions, for example `__strFind` or `__subString`. The result can be a new macro string. You can concatenate macro strings using the `+` operator, for example *`str`* `+ "tail"`. You can also access individual characters using subscription, for example *`str`*`[3]`. You can get the length of a string using `sizeof(`*`str`*`)`. Note that a macro string is not NULL-terminated.

The macro function `__toString` is used for converting from a NULL-terminated C string in your application (`char*` or `char[]`) to a macro string. For example, assume the following definition of a C string in your application:

```
char const *cstr = "Hello";
```

Then examine the following examples:

```
__var str;          /* A macro variable */
str = cstr          /* str is now just a pointer to char */
sizeof str          /* same as sizeof (char*), typically 2 or 4 */
str = __toString(cstr,512)  /* str is now a macro string */
sizeof str          /* 5, the length of the string */
str[1]              /* 101, the ASCII code for 'e' */
str += " World!"    /* str is now "Hello World!" */
```

See also *Formatted output*, page 400.

MACRO STATEMENTS

Statements are expected to behave in the same way as the corresponding C statements would do. The following C-SPY macro statements are accepted:

Expressions

expression;

For detailed information about C-SPY expressions, see *C-SPY expressions*, page 123.For details about C-SPY expressions, see the *MSP430 IAR Embedded Workbench® IDE User Guide*.

Conditional statements

```
if (expression)
  statement

if (expression)
  statement
else
  statement
```

Loop statements

```
for (init_expression; cond_expression; update_expression)
  statement

while (expression)
  statement
```

```
do
  statement
while (expression);
```

Return statements

```
return;

return expression;
```

If the return value is not explicitly set, `signed int 0` is returned by default.

Blocks

Statements can be grouped in blocks.

```
{
  statement1
  statement2
  .
  .
  .
  statementN
}
```

FORMATTED OUTPUT

C-SPY provides different methods for producing formatted output:

`__message argList;`	Prints the output to the Debug Log window.
`__fmessage file, argList;`	Prints the output to the designated file.
`__smessage argList;`	Returns a string containing the formatted output.

where *argList* is a comma-separated list of C-SPY expressions or strings, and *file* is the result of the `__openFile` system macro, see *__openFile*, page 407.

Examples

Use the `__message` statement, as in the following example:

```
var1 = 42;
var2 = 37;
__message "This line prints the values ", var1, " and ", var2,
" in the Log window.";
```

This should produce the following message in the Log window:

```
This line prints the values 42 and 37 in the Log window.
```

Use `__fmessage` to write the output to the designated file, for example:

```
__fmessage myfile, "Result is ", res, "!\n";
```

Finally, use `__smessage` to produce strings, for example:

```
myMacroVar = __smessage 42, " is the answer.";
```

`myMacroVar` now contains the string `"42 is the answer"`.

Specifying display format of arguments

It is possible to override the default display format of a scalar argument (number or pointer) in *`argList`* by suffixing it with a `:` followed by a format specifier. Available specifiers are `%b` for binary, `%o` for octal, `%d` for decimal, `%x` for hexadecimal and `%c` for character. These match the formats available in the Watch and Locals windows, but number prefixes and quotes around strings and characters are not printed. Another example:

```
__message "The character '", cvar:%c, "' has the decimal value
", cvar;
```

This might produce:

```
The character 'A' has the decimal value 65
```

Note: A character enclosed in single quotes (a character literal) is an integer constant and is not automatically formatted as a character. For example:

```
__message 'A', " is the numeric value of the character ",
'A':%c;
```

would produce:

```
65 is the numeric value of the character A
```

Note: The default format for certain types is primarily designed to be useful in the Watch window and other related windows. For example, a value of type `char` is formatted as 'A' (0x41), while a pointer to a character (potentially a C string) is formatted as 0x8102 "Hello", where the string part shows the beginning of the string (currently up to 60 characters).

When printing a value of type `char*`, use the `%x` format specifier to print just the pointer value in hexadecimal notation, or use the system macro `__toString` to get the full string value.

Setup macro functions summary

The following table summarizes the available setup macro functions:

Macro	Description
`execUserPreload`	Called after communication with the target system is established but before downloading the target application. Implement this macro to initialize memory locations and/or registers which are vital for loading data properly.
`execUserSetup`	Called once after the target application is downloaded. Implement this macro to set up the memory map, breakpoints, interrupts, register macro files, etc.
`execUserReset`	Called each time the reset command is issued. Implement this macro to set up and restore data.
`execUserExit`	Called once when the debug session ends. Implement this macro to save status data etc.

Table 108: C-SPY setup macros

Note: If you define interrupts or breakpoints in a macro file that is executed at system start (using `execUserSetup`) we strongly recommend that you also make sure that they are removed at system shutdown (using `execUserExit`). An example is available in `SetupSimple.mac`, see *Simulating an interrupt*, page 57.For an example, see the tutorials in the *MSP430 IAR Embedded Workbench® IDE User Guide*.

The reason for this is that the simulator saves interrupt settings between sessions and if they are not removed they will get duplicated every time `execUserSetup` is executed again. This seriously affects the execution speed.

C-SPY system macros summary

The following table summarizes the pre-defined system macros:

Macro	Description
`__cancelAllInterrupts`	Cancels all ordered interrupts
`__cancelInterrupt`	Cancels an interrupt
`__clearBreak`	Clears a breakpoint
`__closeFile`	Closes a file that was opened by `__openFile`
`__disableInterrupts`	Disables generation of interrupts
`__driverType`	Verifies the driver type
`__enableInterrupts`	Enables generation of interrupts

Table 109: Summary of system macros

Macro	Description
__evaluate	Interprets the input string as an expression and evaluates it.
__openFile	Opens a file for I/O operations
__orderInterrupt	Generates an interrupt
__popSimulatorInterruptExecutingStack	Informs the interrupt simulation system that an interrupt handler has finished executing
__readFile	Reads from the specified file
__readFileByte	Reads one byte from the specified file
__readMemoryByte	Reads one byte from the specified memory location
__readMemory8	Reads one byte from the specified memory location
__readMemory16	Reads two bytes from the specified memory location
__readMemory32	Reads four bytes from the specified memory location
__registerMacroFile	Registers macros from the specified file
__resetFile	Rewinds a file opened by __openFile
__setAdvancedTriggerBreak	Sets an advanced trigger breakpoint
__setCodeBreak	Sets a code breakpoint
__setConditionalBreak	Sets a conditional breakpoint
__setDataBreak	Sets a data breakpoint
__setRangeBreak	Sets a range breakpoint
__setSimBreak	Sets a simulation breakpoint
__sourcePosition	Returns the file name and source location if current execution location corresponds to a source location
__strFind	Searches a given string for the occurrence of another string
__subString	Extracts a substring from another string
__toLower	Returns a copy of the parameter string where all the characters have been converted to lower case
__toString	Prints strings
__toUpper	Returns a copy of the parameter string where all the characters have been converted to upper case
__writeFile	Writes to the specified file
__writeFileByte	Writes one byte to the specified file
__writeMemoryByte	Writes one byte to the specified memory location

Table 109: Summary of system macros (Continued)

Macro	Description
`__writeMemory8`	Writes one byte to the specified memory location
`__writeMemory16`	Writes a two-byte word to the specified memory location
`__writeMemory32`	Writes a four-byte word to the specified memory location

Table 109: Summary of system macros (Continued)

Description of C-SPY system macros

HistoryThis section gives reference information about each of the C-SPY system macros.

__cancelAllInterrupts

Syntax `__cancelAllInterrupts()`

Return value `int 0`

Description Cancels all ordered interrupts.

Applicability This system macro is only available in IAR C-SPY Simulator.

__cancelInterrupt

Syntax `__cancelInterrupt(`*`interrupt_id`*`)`

Parameter

`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

Table 110: __cancelInterrupt return values

Description Cancels the specified interrupt.

Applicability This system macro is only available in IAR C-SPY Simulator.

__clearBreak

Syntax `__clearBreak(break_id)`

Parameter

break_id	The value returned by any of the set breakpoint macros

Return value `int 0`

Description Clears a user-defined breakpoint.

See also *Defining breakpoints*, page 129.For details about the breakpoint system, see the *MSP430 IAR Embedded Workbench® IDE User Guide*.

__closeFile

Syntax `__closeFile(filehandle)`

Parameter

filehandle	The macro variable used as filehandle by the `__openFile` macro

Return value `int 0`

Description Closes a file previously opened by `__openFile`.

__disableInterrupts

Syntax `__disableInterrupts()`

Return value

Result	Value
Successful	`int 0`
Unsuccessful	Non-zero error number

Table 111: __disableInterrupts return values

Description Disables the generation of interrupts.

Applicability This system macro is only available in IAR C-SPY Simulator.

__driverType

Syntax

```
__driverType(driver_id)
```

Parameter

driver_id	A string corresponding to the driver you want to check for; one of the following: `"sim"` corresponds to the simulator driver `"fet"` corresponds to the FET debugger driver

Return value

Result	Value
Successful	`1`
Unsuccessful	`0`

Table 112: __driverType return values

Description

Checks to see if the current IAR C-SPY Debugger driver is identical to the driver type of the *driver_id* parameter.

Example

```
__driverType("sim")
```

If a simulator is the current driver, the value `1` is returned. Otherwise `0` is returned.

__enableInterrupts

Syntax

```
__enableInterrupts()
```

Return value

Result	Value
Successful	`int 0`
Unsuccessful	Non-zero error number

Table 113: __enableInterrupts return values

Description

Enables the generation of interrupts.

Applicability

This system macro is only available in IAR C-SPY Simulator.

__evaluate

Syntax

```
__evaluate(string, valuePtr)
```

Parameter

string	Expression string
valuePtr	Pointer to macro variable storing the result

Return value

Result	Value
Successful	`int 0`
Unsuccessful	`int 1`

Table 114: __evaluate return values

Description

This macro interprets the input string as an expression and evaluates it. The result is stored in a variable pointed to be *valuePtr*.

Example

The following example assumes that the variable `i` is defined and has the value 5:

```
__evaluate("i + 3", &myVar)
```

The macro variable `myVar` is assigned the value 8.

__openFile

Syntax

```
__openFile(file, access)
```

Parameters

file	The filename as a string
access	The access type (string); one of the following: `"r"` ASCII read `"w"` ASCII write

Return value

Result	Value
Successful	The file handle
Unsuccessful	An invalid file handle, which tests as False

Table 115: __openFile return values

Description

Opens a file for I/O operations. The default base directory of this macro is where the currently open project file (`*.pew` or `*.prj`) is located. The argument to `__openFile` can specify a location relative to this directory. In addition, you can use argument variables such as `$PROJ_DIR$` and `$TOOLKIT_DIR$` in the path argument.

Example

```
__var filehandle;         /* The macro variable to contain */
                          /* the file handle */
filehandle = __openFile("Debug\\Exe\\test.tst", "r");
if (filehandle)
{
    /* successful opening */
}
```

See also

Argument variables summary, page 279.

__orderInterrupt

Syntax

```
__orderInterrupt(specification, first_activation,
                 repeat_interval, variance, infinite_hold_time,
                 hold_time, probability)
```

Parameters

specification	The interrupt (string). The specification can either be the full specification used in the device description file (`ddf`) or only the name. In the latter case the interrupt system will automatically get the description from the device description file.
first_activation	The first activation time in cycles (integer)
repeat_interval	The periodicity in cycles (integer)
variance	The timing variation range in percent (integer between 0 and 100)
infinite_hold_time	1 if infinite, otherwise 0.
hold_time	The hold time (integer)
probability	The probability in percent (integer between 0 and 100)

Return value

The macro returns an interrupt identifier (`unsigned long`).

If the syntax of *specification* is incorrect, it returns -1.

Description

Generates an interrupt.

Applicability

This system macro is only available in IAR C-SPY Simulator.

Example

The following example generates a repeating interrupt using an infinite hold time first activated after 4000 cycles:

```
__orderInterrupt( "USART0RX_VECTOR", 4000, 2000, 0, 1, 0, 100 );
```

__popSimulatorInterruptExecutingStack

Syntax

```
__popSimulatorInterruptExecutingStack(void)
```

Return value

This macro has no return value.

Description

Informs the interrupt simulation system that an interrupt handler has finished executing, as if the normal instruction used for returning from an interrupt handler was executed.

This is useful if you are using interrupts in such a way that the normal instruction for returning from an interrupt handler is not used, for example in an operating system with task-switching. In this case, the interrupt simulation system cannot automatically detect that the interrupt has finished executing.

Applicability

This system macro is only available in IAR C-SPY Simulator.

__readFile

Syntax

```
__readFile(file, valuePtr)
```

Parameters

file	A file handle
valuePtr	A pointer to a variable

Return value

Result	Value
Successful	0
Unsuccessful	Non-zero error number

Table 116: __readFile return values

Description

Reads a sequence of hexadecimal digits from the given file and converts them to an `unsigned long` which is assigned to the *value* parameter, which should be a pointer to a macro variable.

Example

```
__var number;
if (__readFile(myFile, &number) == 0)
{
```

```
  // Do something with number
}
```

__readFileByte

Syntax `__readFileByte(file)`

Parameter

file	A file handle

Return value `-1` upon error or end-of-file, otherwise a value between `0` and `255`.

Description Reads one byte from the file *file*.

Example

```
__var byte;
while ( (byte = __readFileByte(myFile)) != -1 )
{
  // Do something with byte
}
```

__readMemoryByte

Syntax `__readMemoryByte(address, zone)`

Parameters

address	The memory address (integer)
zone	The memory zone name (string); for a list of available zones, see *Memory addressing*, page 135

Return value The macro returns the value from memory.

Description Reads one byte from a given memory location.

Example

```
__readMemoryByte(0x0108, "Memory");
```

__readMemory8

Syntax `__readMemory8(address, zone)`

Parameters

address	The memory address (integer)
zone	The memory zone name (string); for a list of available zones, see *Memory addressing*, page 135

Return value The macro returns the value from memory.

Description Reads one byte from a given memory location.

Example
```
__readMemory8(0x0108, "Memory");
```

__readMemory16

Syntax
```
__readMemory16(address, zone)
```

Parameters

address	The memory address (integer)
zone	The memory zone name (string); for a list of available zones, see *Memory addressing*, page 135

Return value The macro returns the value from memory.

Description Reads a two-byte word from a given memory location.

Example
```
__readMemory16(0x0108, "Memory");
```

__readMemory32

Syntax
```
__readMemory32(address, zone)
```

Parameters

address	The memory address (integer)
zone	The memory zone name (string); for a list of available zones, see *Memory addressing*, page 135

Return value The macro returns the value from memory.

Description Reads a four-byte word from a given memory location.

Example
```
__readMemory32(0x0108, "Memory");
```

__registerMacroFile

Syntax `__registerMacroFile(filename)`

Parameter

filename	A file containing the macros to be registered (string)

Return value `int 0`

Description Registers macros from a setup macro file. With this function you can register multiple macro files during C-SPY startup.

Example
```
__registerMacroFile("c:\\testdir\\macro.mac");
```

See also *Registering and executing using setup macros and setup files*, page 147.For details about the macro system, see the *MSP430 IAR Embedded Workbench® IDE User Guide*.

__resetFile

Syntax `__resetFile(filehandle)`

Parameter

filehandle	The macro variable used as filehandle by the `__openFile` macro

Return value `int 0`

Description Rewinds a file previously opened by `__openFile`.

__setAdvancedTriggerBreak

Syntax `__setAdvancedTriggerBreak(type, condition, access, action, mask cond_value)`

Parameters All parameters are strings.

type	The breakpoint type; either `"Address"`, `"Data"`, or `"Register"`.
condition	The breakpoint condition operator, either `"=="`, `">="`, `"<="`, or `"!="`.

access	The memory access type. One of the following:
	`"Read"`
	`"Write"`
	`"ReadWrite"`
	`"Fetch"`
	`"FetchHold"`
	`"NoFetch"`
	`"NoFetchRead"`
	`"NoFetchNoDMA"`
	`"DMA"`
	`"NoDMA"`
	`"WriteNoDMA"`
	`"NoFetchReadNoDMA"`
	`"ReadNoDMA"`
	`"ReadDMA"`
	`"WriteDMA"`

action	The action type: `"Break"`, `"Trigger"`, or `"BreakTrigger"`.
mask	A 16-bit value that the breakpoint address or value will be masked with.
cond_value	An extra conditional data value.

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	0

Table 117: __setAdvancedTriggerBreak return values

Description

Sets an advanced trigger breakpoint.

Applicability

This macro can only be used with the FET Debugger version of C-SPY.

Example

```
__var brk;
brk = __setAdvancedTriggerBreak("Register", ">=", "Write",
                                "Trigger", "0x0000", "0x4000");
...
__clearBreak(brk);
```

See also

Defining breakpoints, page 129 and *Advanced trigger breakpoints*, page 213. For details about the breakpoint system, see the *MSP430 IAR Embedded Workbench® IDE User Guide*.

__setCodeBreak

Syntax

```
__setCodeBreak(location, count, condition, cond_type, action)
```

Parameters

location	A string with a location description. This can be either: A source location on the form {*filename*}.*line*.*col* (for example {D:\\src\\prog.c}.12.9) An absolute location on the form *zone*:*hexaddress* or simply *hexaddress* (for example Memory:0x42) An expression whose value designates a location (for example main)
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)
action	An expression, typically a call to a macro, which is evaluated when the breakpoint is detected

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	0

Table 118: __setCodeBreak return values

Description

Sets a code breakpoint, that is, a breakpoint which is triggered just before the processor fetches an instruction at the specified location.

Examples

```
__setCodeBreak("{D:\\src\\prog.c}.12.9", 3, "d>16", "TRUE",
"ActionCode()");
```

The following example sets a code breakpoint on the label main in your assembler source:

```
__setCodeBreak("#main", 0, "1", "TRUE", "");
```

See also

Defining breakpoints, page 129.For details about the breakpoint system, see the *MSP430 IAR Embedded Workbench® IDE User Guide*.

__setConditionalBreak

Syntax

```
__setConditionalBreak(location, type, operator, access, action,
                      mask, cond_value, cond_operator,
                       cond_access, cond_mask)
```

Parameters

All parameters are strings.

location	The breakpoint location. This can be either: A *source location* on the form `"{filename}.line.col"` (for example `"{D:\\src\\prog.c}.12.9"`) An *absolute location* on the form `"zone:hexaddress"` or simply `"hexaddress"` (for example `"Memory:0x42"`) An *expression* whose value designates a location (for example `"my_global_variable"`). A register (for example `"R10"`)
type	The breakpoint type; either `"Address"`, `"Data"`, or `"Register"`.
operator	The breakpoint operator, either `"=="`, `">="`, `"<="`, or `"!="`.
access	The memory access type: `"Read"`, `"Write"`, `"ReadWrite"`, or `"Fetch"`.
action	The action type: `"Break"`, `"Trigger"`, or `"BreakTrigger"`.
mask	A 16-bit value that the breakpoint address or value will be masked with.
cond_value	An extra conditional data value.
cond_operator	The condition operator, either `"=="`, `">="`, `"<="`, or `"!="`.
cond_access	The access type of the condition: `"Read"` or `"Write"`.
cond_mask	The mask value of the condition.

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	0

Table 119: __setConditionalBreak return values

Description

Sets a conditional breakpoint.

Applicability

This macro can only be used with the FET Debugger version of C-SPY.

Example

```
__var brk;
brk = __setConditionalBreak("R10", "Register", "0x5000", ">=",
      "Write", "Trigger", "0x0000", "0x4000", "<=", "Write",
      "0x00FF");
...
__clearBreak(brk);
```

See also

Defining breakpoints, page 129 and *Conditional breakpoints*, page 210. For details about the breakpoint system, see the *MSP430 IAR Embedded Workbench® IDE User Guide*.

__setDataBreak

Syntax

```
__setDataBreak(location, count, condition, cond_type, access,
               action)
```

Parameters

location	A string with a location description. This can be either: A *source location* on the form `{filename}.line.col` (for example `{D:\\src\\prog.c}.12.9`), although this is not very useful for data breakpoints An *absolute location* on the form `zone:hexaddress` or simply `hexaddress` (for example `Memory:0x42`) An *expression* whose value designates a location (for example `my_global_variable`).
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: `"R"` for read, `"W"` for write, or `"RW"` for read/write
`action`	An expression, typically a call to a macro, which is evaluated when the breakpoint is detected

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	0

Table 120: __setDataBreak return values

Description

Sets a data breakpoint, that is, a breakpoint which is triggered directly after the processor has read or written data at the specified location.

Applicability

This system macro is only available in IAR C-SPY Simulator.

Example

```
__var brk;
brk = __setDataBreak("Memory:0x4710", 3, "d>6", "TRUE",
      "W", "ActionData()");
...
__clearBreak(brk);
```

See also

Defining breakpoints, page 129.For details about the breakpoint system, see the *MSP430 IAR Embedded Workbench® IDE User Guide*.

__setRangeBreak

Syntax

```
__setRangeBreak(start_loc, end_loc, end_cond, type, access,
          action, action_when)
```

Parameters

All parameters are strings.

start_loc The start location. This can be either:
A *source location* on the form `"{filename}.line.col"` (for example `"{D:\\src\\prog.c}.12.9"`)

An *absolute location* on the form `"zone:hexaddress"` or simply `"hexaddress"` (for example `"Memory:0x42"`)

An *expression* whose value designates a location (for example `"my_global_variable"`).

end_loc The end location. This can be either the same as for *start_loc* above or the length of the range.

end_cond The type of end condition, either `"Location"`, `"Length"`, or `"Automatic"`.

type	The breakpoint type; either `"Address"` or `"Data"`.
access	The memory access type: `"Read"`, `"Write"`, `"ReadWrite"`, or `"Fetch"`.
action	The action type: `"Break"`, `"Trigger"`, or `"BreakTrigger"`.
action_when	Specifies if the action should happen at an access inside or outside of the specified range, either `"Inside"` or `"Outside"`.

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	0

Table 121: __setRangeBreak return values

Description

Sets a range breakpoint.

Applicability

This macro can only be used with the FET Debugger version of C-SPY.

Example

```
__var brk;
brk = __setRangeBreak("Memory:0x1240", "Memory:0x1360",
      "Location", "Address", "Fetch", "Trigger", "Inside");
...
__clearBreak(brk);
```

See also

Defining breakpoints, page 129 and *Range breakpoints*, page 207. For details about the breakpoint system, see the *MSP430 IAR Embedded Workbench® IDE User Guide*.

__setSimBreak

Syntax

```
__setSimBreak(location, access, action)
```

Parameters

location	A string with a location description. This can be either: A *source location* on the form `{`*filename*`}.`*line*`.`*col* (for example `{D:\\src\\prog.c}.12.9`), although this is not very useful for simulation breakpoints. An *absolute location* on the form *zone*`:`*hexaddress* or simply *hexaddress* (for example `Memory:0xE01E`). An *expression* whose value designates a location (for example `my_global_variable`).
access	The memory access type: `"R"` for read or `"W"` for write
action	An expression, typically a call to a macro function, which is evaluated when the breakpoint is detected

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	`0`

Table 122: __setSimBreak return values

Description

Use this system macro to set *immediate* breakpoints, which will halt instruction execution only temporarily. This allows a C-SPY macro function to be called when the processor is about to read data from a location or immediately after it has written data. Instruction execution will resume after the action.

This type of breakpoint is useful for simulating memory-mapped devices of various kinds (for instance serial ports and timers). When the processor reads at a memory-mapped location, a C-SPY macro function can intervene and supply the appropriate data. Conversely, when the processor writes to a memory-mapped location, a C-SPY macro function can act on the value that was written.

Applicability

This system macro is only available in the IAR C-SPY Simulator.

__sourcePosition

Syntax

```
__sourcePosition(linePtr, colPtr)
```

Parameters

linePtr	Pointer to the variable storing the line number
colPtr	Pointer to the variable storing the column number

Return value

Result	Value
Successful	Filename string
Unsuccessful	Empty ("") string

Table 123: __sourcePosition return values

Description

If the current execution location corresponds to a source location, this macro returns the filename as a string. It also sets the value of the variables, pointed to by the parameters, to the line and column numbers of the source location.

__strFind

Syntax

```
__strFind(macroString, pattern, position)
```

Parameters

macroString	The macro string to search in
pattern	The string pattern to search for
position	The position where to start the search. The first position is `0`

Return value

The position where the pattern was found or `-1` if the string is not found.

Description

This macro searches a given string for the occurrence of another string.

Example

```
__strFind("Compiler", "pile", 0)  = 3
__strFind("Compiler", "foo", 0)   = -1
```

See also

Macro strings, page 398.

__subString

Syntax

```
__subString(macroString, position, length)
```

Parameters

macroString	The macro string from which to extract a substring
position	The start position of the substring. The first position is 0.
length	The length of the substring

Return value — A substring extracted from the given macro string.

Description — This macro extracts a substring from another string.

Example

```
__subString("Compiler", 0, 2)
```

The resulting macro string contains `Co`.

```
__subString("Compiler", 3, 4)
```

The resulting macro string contains `pile`.

See also — *Macro strings*, page 398.

__toLower

Syntax — `__toLower(`*macroString*`)`

Parameter — *macroString* is any macro string.

Return value — The converted macro string.

Description — This macro returns a copy of the parameter string where all the characters have been converted to lower case.

Example

```
__toLower("IAR")
```

The resulting macro string contains `iar`.

```
__toLower("Mix42")
```

The resulting macro string contains `mix42`.

See also — *Macro strings*, page 398.

__toString

Syntax `__toString(C_string, maxlength)`

Parameter

string	Any null-terminated C string
maxlength	The maximum length of the returned macro string

Return value Macro string.

Description This macro is used for converting C strings (`char*` or `char[]`) into macro strings.

Example Assuming your application contains the following definition:

```
char const * hptr = "Hello World!";
```

the following macro call:

```
__toString(hptr, 5)
```

would return the macro string containing `Hello`.

See also *Macro strings*, page 398.

__toUpper

Syntax `__toUpper(macroString)`

Parameter *macroString* is any macro string.

Return value The converted string.

Description This macro returns a copy of the parameter *macroString* where all the characters have been converted to upper case.

Example

```
__toUpper("string")
```

The resulting macro string contains `STRING`.

See also *Macro strings*, page 398.

__writeFile

Syntax `__writeFile(file, value)`

Parameters

file	A file handle
value	An integer

Return value `int 0`

Description Prints the integer value in hexadecimal format (with a trailing space) to the file *file*.

Note: The `__fmessage` statement can do the same thing. The `__writeFile` macro is provided for symmetry with `__readFile`.

__writeFileByte

Syntax `__writeFileByte(file, value)`

Parameters

file	A file handle
value	An integer in the range `0-255`

Return value `int 0`

Description Writes one byte to the file *file*.

__writeMemoryByte

Syntax `__writeMemoryByte(value, address, zone)`

Parameters

value	The value to be written (integer)
address	The memory address (integer)
zone	The memory zone name (string); for a list of available zones, see *Memory addressing*, page 135

Return value `int 0`

Description Writes one byte to a given memory location.

Example
```
__writeMemoryByte(0x2F, 0x1F, "Memory");
```

__writeMemory8

Syntax `__writeMemory8(value, address, zone)`

Parameters

value	The value to be written (integer)
address	The memory address (integer)
zone	The memory zone name (string); for a list of available zones, see *Memory addressing*, page 135

Return value `int 0`

Description Writes one byte to a given memory location.

Example
```
__writeMemory8(0x2F, 0x8020, "Memory");
```

__writeMemory16

Syntax `__writeMemory16(value, address, zone)`

Parameters

value	The value to be written (integer)
address	The memory address (integer)
zone	The memory zone name (string); for a list of available zones, see *Memory addressing*, page 135

Return value `int 0`

Description Writes two bytes to a given memory location.

Example
```
__writeMemory16(0x2FFF, 0x8020, "Memory");
```

__writeMemory32

Syntax `__writeMemory32(value, address, zone)`

Parameters

value	The value to be written (integer)
address	The memory address (integer)

	zone	The memory zone name (string); for a list of available zones, see *Memory addressing*, page 135

Return value `int 0`

Description Writes four bytes to a given memory location.

Example

```
__writeMemory32(0x5555FFFF, 0x8020, "Memory");
```

Glossary

A

Absolute location
A specific memory address for an object specified in the source code, as opposed to the object being assigned a location by the IAR XLINK Linker.

Absolute segments
Segments that have fixed locations in memory before linking.

Address expression
An expression which has an address as its value.

Application
The program developed by the user of the IAR Systems toolkit and which will be run as an embedded application on a target processor.

Architecture
A term used by computer designers to designate the structure of complex information-processing systems. It includes the kinds of instructions and data used, the memory organization and addressing, and the methods by which the system is implemented. The two main architecture types used in processor design are *Harvard architecture* and *von Neumann architecture*.

Assembler directives
The set of commands that control how the assembler operates.

Assembler options
Parameters you can specify to change the default behavior of the assembler.

Assembler language
A machine-specific set of mnemonics used to specify operations to the target processor and input or output registers or data areas. Assembler language might sometimes be preferred over C/Embedded C++ to save memory or to enhance the execution speed of the application.

Auto variables
The term refers to the fact that each time the function in which the variable is declared is called, a new instance of the variable is created automatically. This can be compared with the behavior of local variables in systems using static overlay, where a local variable only exists in one instance, even if the function is called recursively. Also called local variables. Compare *Register variables*.

B

Backtrace
Information that allows the IAR C-SPY® Debugger to show, without any runtime penalty, the complete stack of function calls wherever the program counter is, provided that the code comes from compiled C functions.

Bank
See *Memory bank*.

Bank switching
Switching between different sets of memory banks. This software technique is used to increase a computer's usable memory by allowing different pieces of memory to occupy the same address space.

Banked code
Code that is distributed over several banks of memory. Each function must reside in only one bank.

Banked data
Data that is distributed over several banks of memory. Each data object must fit inside one memory bank.

Banked memory
Has multiple storage locations for the same address. See also *Memory bank*.

Bank-switching routines
Code that selects a memory bank.

Batch files
A text file containing operating system commands which are executed by the command line interpreter. In Unix, this is called a "shell script" because it is the Unix shell which includes the command line interpreter. Batch files can be used as a simple way to combine existing commands into new commands.

Bitfield
A group of bits considered as a unit.

Breakpoint
1. Code breakpoint. A point in a program that, when reached, triggers some special behavior useful to the process of debugging. Generally, breakpoints are used for stopping program execution or dumping the values of some or all of the program variables. Breakpoints can be part of the program itself, or they can be set by the programmer as part of an interactive session with a debugging tool for scrutinizing the program's execution.

2. Data breakpoint. A point in memory that, when accessed, triggers some special behavior useful to the process of debugging. Generally, data breakpoints are used to stop program execution when an address location is accessed either by a read operation or a write operation.

3. Immediate breakpoint. A point in memory that, when accessed, trigger some special behavior useful in the process of debugging. Immediate breakpoints are generally used for halting the program execution in the middle of a memory access instruction (before or after the actual memory access depending on the access type) while performing some user-specified action. The execution is then resumed. This feature is only available in the simulator version of C-SPY.

C

Calling convention
A calling convention describes the way one function in a program calls another function. This includes how register parameters are handled, how the return value is returned, and which registers that will be preserved by the called function. The compiler handles this automatically for all C and C++ functions. All code written in assembler language must conform to the rules in the calling convention in order to be callable from C or C++, or to be able to call C and C++ functions. The C calling convention and the C++ calling conventions are not necessarily the same.

Cheap
As in *cheap memory access*. A cheap memory access either requires few cycles to perform, or few bytes of code to implement. A cheap memory access is said to have a low cost. See *Memory access cost*.

Checksum
A computed value which depends on the contents of a block of data and which is stored along with the data in order to detect corruption of the data. Compare *CRC (cyclic redundancy checking)*.

Code banking
See *Banked code*.

Code model
The code model controls how code is generated for an application. Typically, the code model controls behavior such as how functions are called and in which code segment functions will be located. All object files of an application must be compiled using the same code model.

Code pointers
A code pointer is a function pointer. As many microcontrollers allow several different methods of calling a function, compilers for embedded systems usually provide the users with the ability to use all these methods.

Do not confuse code pointers with data pointers.

Compilation unit
See *Translation unit*.

Compiler function directives
The compiler function directives are generated by the compiler to pass information about functions and function calls to the IAR XLINK Linker. To view these directives, you must create an assembler list file. These directives are primarily intended for compilers that support static overlay, a feature which is useful in smaller microcontrollers.

Compiler options
Parameters you can specify to change the default behavior of the compiler.

Cost
See *Memory access cost*.

CRC (cyclic redundancy checking)
A number derived from, and stored with, a block of data in order to detect corruption. A CRC is based on polynomials and is a more advanced way of detecting errors than a simple arithmetic checksum. Compare *Checksum*.

C-SPY options
Parameters you can specify to change the default behavior of the IAR C-SPY Debugger.

Cstartup
Code that sets up the system before the application starts executing.

C-style preprocessor
A preprocessor is either a stand-alone application or an integrated part of a compiler, that performs preprocessing of the input stream before actual compilation takes place. A C-style preprocessor follows the rules set up in the ANSI specification of the C language and implements commands like #define, #if, and #include, which are used to handle textual macro substitution, conditional compilation, and inclusion of other files.

D

Data banking
See *Banked data*.

Data model
The data model specifies the default memory type. This means that the data model typically controls one or more of the following: The method used and the code generated to access static and global variables, dynamically allocated data, and the runtime stack. It also controls the default pointer type and in which data segments static and global variables will be located. A project can only use one data model at a time, and the same model must be used by all user modules and all library modules in the project.

Data pointers
Many microcontrollers have different addressing modes in order to access different memory types or address spaces. Compilers for embedded systems usually have a set of different data pointer types so they can access the available memory efficiently.

Data representation
How different data types are laid out in memory and what value ranges they represent.

Declaration
A specification to the compiler that an object, a variable or function, exists. The object itself must be defined in exactly one translation unit (source file). An object must either be declared or defined before it is used. Normally an object that is used in many files is defined in one source file. A declaration is normally placed in a header file that is included by the files that use the object.

For example:

```
/* Variable "a" exists somewhere. Function
 "b" takes two int parameters and returns an
 int. */

extern int a;
int b(int, int);
```

Definition
The variable or function itself. Only one definition can exist for each variable or function in an application. See also *Tentative definition*.

For example:

```
int a;
int b(int x, int y)
{
   return x + y;
}
```

Derivative
One of two or more processor variants in a series or family of microprocessors or microcontrollers.

Device description file
A file used by the IAR C-SPY Debugger that contains various device-specific information such as I/O registers (SFR) definitions, interrupt vectors, and control register definitions.

Device driver
Software that provides a high-level programming interface to a particular peripheral device.

Digital signal processor (DSP)
A device that is similar to a microprocessor, except that the internal CPU has been optimized for use in applications involving discrete-time signal processing. In addition to standard microprocessor instructions, digital signal processors usually support a set of complex instructions to perform common signal-processing computations quickly.

Disassembly window
A C-SPY window that shows the memory contents disassembled as machine instructions, interspersed with the corresponding C source code (if available).

Dynamic initialization
Variables in a program written in C are initialized during the initial phase of execution, before the main function is called. These variables are always initialized with a static value, which is determined either at compile-time or at link-time. This is called static initialization. In Embedded C++, variables might require initialization to be performed by executing code, for example, running the constructor of global objects, or performing dynamic memory allocation.

Dynamic memory allocation
There are two main strategies for storing variables: statically at link-time, or dynamically at runtime. Dynamic memory allocation is often performed from the heap and it is the size of the heap that determines how much memory that can be used for dynamic objects and variables. The advantage of dynamic memory allocation is that several variables or objects that are not active at the same time can be stored in the same memory, thus reducing the memory need of an application. See also *Heap memory*.

Dynamic object
An object that is allocated, created, destroyed, and released at runtime. Dynamic objects are almost always stored in memory that is dynamically allocated. Compare *Static object*.

E

EEPROM
Electrically Erasable, Programmable Read-Only Memory. A type of ROM that can be erased electronically, and then be re-programmed.

EPROM
Erasable, Programmable Read-Only Memory. A type of ROM that can be erased by exposing it to ultraviolet light, and then be re-programmed.

Embedded C++
A subset of the C++ programming language, which is intended for embedded systems programming. The fact that performance and portability are particularly important in embedded systems development was considered when defining the language.

Embedded system
A combination of hardware and software, designed for a specific purpose. Embedded systems are often part of a larger system or product.

Emulator
An emulator is a hardware device that performs emulation of one or more derivatives of a processor family. An emulator can often be used instead of the actual microcontroller and connects directly to the printed circuit board—where the microcontroller would have been connected—via a connecting device. An emulator always behaves exactly as the processor it emulates, and is used when debugging requires all systems actuators, or when debugging device drivers.

Enumeration
A type which includes in its definition an exhaustive list of possible values for variables of that type. Common examples include Boolean, which takes values from the list [true, false], and day-of-week which takes values [Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday]. Enumerated types are a feature of typed languages, including C and Ada.

Characters, (fixed-size) integers, and even floating-point types might be (but are not usually) considered to be (large) enumerated types.

Exceptions
An exception is an interrupt initiated by the processor hardware, or hardware that is tightly coupled with the processor, for instance, a memory management unit (MMU). The exception signals a violation of the rules of the architecture (access to protected memory), or an extreme error condition (division by zero).

Do not confuse this use of the word exception with the term *exception* used in the C++ language (but not in Embedded C++).

Expensive
As in *expensive memory access*. An expensive memory access either requires many cycles to perform, or many bytes of code to implement. An expensive memory access is said to have a high cost. See *Memory access cost*.

Extended keywords
Non-standard keywords in C and C++. These usually control the definition and declaration of objects (that is, data and functions). See also *Keywords*.

F

Format specifiers
Used to specify the format of strings sent by library functions such as printf. In the following example, the function call contains one format string with one format specifier, %c, that prints the value of a as a single ASCII character:

```
printf("a = %c", a);
```

G

General options
Parameters you can specify to change the default behavior of all tools that are included in the IAR Embedded Workbench IDE.

Generic pointers
Pointers that have the ability to point to all different memory types in, for example, a microcontroller based on the Harvard architecture.

H

Harvard architecture
A microcontroller based on the Harvard architecture has separate data and instruction buses. This allows execution to occur in parallel. As an instruction is being fetched, the current instruction is executing on the data bus. Once the current instruction is complete, the next instruction is ready to go. This theoretically allows for much faster execution than a von Neumann architecture, but there is some added silicon complexity. Compare *von Neumann architecture*.

Heap memory
The heap is a pool of memory in a system that is reserved for dynamic memory allocation. An application can request parts of the heap for its own use; once memory has been allocated from the heap it remains valid until it is explicitly released back to the heap by the application. This type of memory is

useful when the number of objects is not known until the application executes. Note that this type of memory is risky to use in systems with a limited amount of memory or systems that are expected to run for a very long time.

Heap size
Total size of memory that can be dynamically allocated.

Host
The computer that communicates with the target processor. The term is used to distinguish the computer on which the debugger is running from the microcontroller the embedded application you develop runs on.

I

IDE (integrated development environment)
A programming environment with all necessary tools integrated into one single application.

Include file
A text file which is included into a source file. This is often performed by the preprocessor.

Inline assembler
Assembler language code that is inserted directly between C statements.

Inlining
An optimization that replaces function calls with the body of the called function. This optimization increases the execution speed and can even reduce the size of the generated code.

Instruction mnemonics
A word or acronym used in assembler language to represent a machine instruction. Different processors have different instruction sets and therefore use a different set of mnemonics to represent them, such as, ADD, BR (branch), BLT (branch if less than), MOVE, LDR (load register).

Interrupt vector
A small piece of code that will be executed, or a pointer that points to code that will be executed when an interrupt occurs.

Interrupt vector table
A table containing interrupt vectors, indexed by interrupt type. This table contains the processor's mapping between interrupts and interrupt service routines and must be initialized by the programmer.

Interrupts
In embedded systems, the use of interrupts is a method of detecting external events immediately, for example a timer overflow or the pressing of a button.

Interrupts are asynchronous events that suspend normal processing and temporarily divert the flow of control through an "interrupt handler" routine. Interrupts can be caused by both hardware (I/O, timer, machine check) and software (supervisor, system call or trap instruction). Compare *Trap*.

Intrinsic
An adjective describing native compiler objects, properties, events, and methods.

Intrinsic functions
1. Function calls that are directly expanded into specific sequences of machine code. 2. Functions called by the compiler for internal purposes (that is, floating point arithmetic etc.).

K

Key bindings
Key shortcuts for menu commands used in the IAR Embedded Workbench IDE.

Keywords
A fixed set of symbols built into the syntax of a programming language. All keywords used in a language are reserved—they cannot be used as identifiers (in other words, user-defined objects such as variables or procedures). See also *Extended keywords*.

L

L-value
A value that can be found on the left side of an assignment and thus be changed. This includes plain variables and de-referenced pointers. Expressions like (x + 10) cannot be assigned a new value and are therefore not L-values.

Language extensions
Target-specific extensions to the C language.

Library
See *Runtime library*.

Linker command file
A file used by the IAR XLINK Linker. It contains command line options which specify the locations where the memory segments can be placed, thereby assuring that your application fits on the target chip.

Because many of the chip-specific details are specified in the linker command file and not in the source code, the linker command file also helps to make the code portable.

In particular, the linker specifies the placement of segments, the stack size, and the heap size.

Local variable
See *Auto variables*.

Location counter
See *Program location counter (PLC)*.

Logical address
See *Virtual address (logical address)*.

M

MAC (Multiply and accumulate)
A special instruction, or on-chip device, that performs a multiplication together with an addition. This is very useful when performing signal processing where many filters and transforms have the form:

$$y_j = \sum_{i=0}^{N} c_i \cdot x_{i+j}$$

The accumulator of the MAC usually has a higher precision (more bits) than normal registers. See also *Digital signal processor (DSP)*.

Macro
1. Assembler macros are user-defined sets of assembler lines that can be expanded later in the source file by referring to the given macro name. Parameters will be substituted if referred to.

2. C macro. A text substitution mechanism used during preprocessing of source files. Macros are defined using the #define preprocessing directive. The replacement text of each macro is then substituted for any occurrences of the macro name in the rest of the translation unit.

3. C-SPY macros are programs that you can write to enhance the functionality of the IAR C-SPY Debugger. A typical application of C-SPY macros is to associate them with breakpoints; when such a breakpoint is hit, the macro is run and can for example be used to simulate peripheral devices, to evaluate complex conditions, or to output a trace.

The C-SPY macro language is like a simple dialect of C, but is less strict with types.

Mailbox
A mailbox in an RTOS is a point of communication between two or more tasks. One task can send messages to another task by placing the message in the mailbox of the other task. Mailboxes are also known as message queues or message ports.

Memory access cost
The cost of a memory access can be in clock cycles, or in the number of bytes of code needed to perform the access. A memory which requires large instructions or many instructions is said to have a higher access cost than a memory which can be accessed with few, or small instructions.

Memory area
A region of the memory.

Memory bank
The smallest unit of continuous memory in banked memory. One memory bank at a time is visible in a microcontroller's physical address space.

Memory map
A map of the different memory areas available to the microcontroller.

Memory model
Specifies the memory hierarchy and how much memory the system can handle. Your application must use only one memory model at a time, and the same model must be used by all user modules and all library modules.

Microcontroller
A microprocessor on a single integrated circuit intended to operate as an embedded system. As well as a CPU, a microcontroller typically includes small amounts of RAM, PROM, timers, and I/O ports.

Microprocessor
A CPU contained on one (or a small number of) integrated circuits. A single-chip microprocessor can include other components such as memory, memory management, caches, floating-point unit, I/O ports and timers. Such devices are also known as microcontrollers.

Module
The basic unit of linking. A module contains definitions for symbols (exports) and references to external symbols (imports). When compiling C/C++, each translation unit produces one module. In assembler, each source file can produce more than one module.

N

Nested interrupts
A system where an interrupt can be interrupted by another interrupt is said to have nested interrupts.

Non-banked memory
Has a single storage location for each memory address in a microcontroller's physical address space.

Non-initialized memory
Memory that can contain any value at reset, or in the case of a soft reset, can remember the value it had before the reset.

Non-volatile storage
Memory devices such as battery-backed RAM, ROM, magnetic tape and magnetic disks that can retain data when electric power is shut off. Compare *Volatile storage*.

NOP
No operation. This is an instruction that does not perform anything, but is used to create a delay. In pipelined architectures, the NOP instruction can be used for synchronizing the pipeline. See also *Pipeline*.

O

Operator
A symbol used as a function, with infix syntax if it has two arguments (+, for example) or prefix syntax if it has only one (for instance, bitwise negation, ~). Many languages use operators for built-in functions such as arithmetic and logic.

Operator precedence
Each operator has a precedence number assigned to it that determines the order in which the operator and its operands are evaluated. The highest precedence operators are evaluated first. Use parentheses to group operators and operands to control the order in which the expressions are evaluated.

P

Parameter passing
See *Calling convention*.

Peripheral
A hardware component other than the processor, for example memory or an I/O device.

Pipeline
A structure that consists of a sequence of stages through which a computation flows. New operations can be initiated at the start of the pipeline even though other operations are already in progress through the pipeline.

Pointer
An object that contains an address to another object of a specified type.

#pragma
During compilation of a C/C++ program, the #pragma preprocessing directive causes the compiler to behave in an implementation-defined manner. This can include, for example, producing output on the console, changing the declaration of a subsequent object, changing the optimization level, or enabling/disabling language extensions.

Pre-emptive multitasking
An RTOS task is allowed to run until a higher priority process is activated. The higher priority task might become active as the result of an interrupt. The term preemptive indicates that although a task is allotted to run a given length of time (a timeslice), it might lose the processor at any time. Each time an interrupt occurs, the task scheduler looks for the highest priority task that is active and switches to that task. If the located task is different from the task that was executing before the interrupt, the previous task is suspended at the point of interruption.

Compare *Round Robin*.

Preprocessing directives
A set of directives that are executed before the parsing of the actual code is started.

Preprocessor
See *C-style preprocessor*.

Processor variant
The different chip setups that the compiler supports. See *Derivative*.

Program counter (PC)
A special processor register that is used to address instructions. Compare *Program location counter (PLC)*.

Program location counter (PLC)
Used in the IAR Assembler to denote the code address of the current instruction. The PLC is represented by a special symbol (typically $) that can be used in arithmetic expressions. Also called simply location counter (LC).

PROM
Programmable Read-Only Memory. A type of ROM that can be programmed only once.

Project
The user application development project.

Project options
General options that apply to an entire project, for example the target processor that the application will run on.

Q

Qualifiers
See *Type qualifiers*.

R

R-value
A value that can be found on the right side of an assignment. This is just a plain value. See also *L-value*.

Real-time operating system (RTOS)
An operating system which guarantees the latency between an interrupt being triggered and the interrupt handler starting, as well as how tasks are scheduled. An RTOS is typically much smaller than a normal desktop operating system. Compare *Real-time system*.

Real-time system
A computer system whose processes are time-sensitive. Compare *Real-time operating system (RTOS)*.

Register constant
A register constant is a value that is loaded into a dedicated processor register when the system is initialized. The compiler can then generate code that assumes that the constants are present in the dedicated registers.

Register
A small on-chip memory unit, usually just one or a few bytes in size, which is particularly efficient to access and therefore often reserved to function as a temporary storage area during program execution.

Register locking
Register locking means that the compiler can be instructed that some processor registers shall not be used during normal code generation. This is useful in a number of situations. For example, some parts of a system might be written in assembler language to gain speed. These parts might be given dedicated processor registers. Or the register might be used by an operating system, or by other third-party software.

Register variables
Typically, register variables are local variables that have been placed in registers instead of on the (stack) frame of the function. Register variables are much more efficient than other variables because they do not require memory accesses, so the compiler can use shorter/faster instructions when working with them. See also *Auto variables*.

Relocatable segments
Segments that have no fixed location in memory before linking.

Reset
A reset is a restart from the initial state of a system. A reset can originate from hardware (hard reset), or from software (soft reset). A hard reset can usually not be distinguished from the power-on condition, which a soft reset can be.

ROM-monitor
A piece of embedded software that has been designed specifically for use as a debugging tool. It resides in the ROM of the evaluation board chip and communicates with a debugger via a serial port or network connection. The ROM-monitor provides a set of primitive commands to view and modify memory locations and registers, create and remove breakpoints, and execute your application. The debugger combines these primitives to fulfill higher-level requests like program download and single-step.

Round Robin
Task scheduling in an operating system, where all tasks have the same priority level and are executed in turn, one after the other. Compare *Pre-emptive multitasking*.

RTOS
See *Real-time operating system (RTOS)*.

Runtime library
A collection of useful routines, stored as an object file, that can be linked into any application.

Runtime model attributes
A mechanism that is designed to prevent modules that are not compatible to be linked into an application. A runtime attribute is a pair constituted of a named key and its corresponding value.

Two modules can only be linked together if they have the same value for each key that they both define.

S

Saturated mathematics
Most, if not all, C and C++ implementations use mod–2^N 2-complement-based mathematics where an overflow wraps the value in the value domain, that is, (127 + 1) = -128. Saturated mathematics, on the other hand, does *not* allow wrapping in the value domain, for instance, (127 + 1) = 127, if 127 is the upper limit. Saturated mathematics is often used in signal processing, where an overflow condition would have been fatal if value wrapping had been allowed.

Scheduler
The part of an RTOS that performs task-switching. It is also responsible for selecting which task that should be allowed to run. There are many different scheduling algorithms, but most of them are either based on static scheduling (performed at compile-time), or on dynamic scheduling (where the actual choice of which task to run next is taken at runtime, depending on the state of the system at the time of the task-switch). Most real-time systems use static scheduling, because it makes it possible to prove that the system will not violate the real-time requirements.

Scope
The section of an application where a function or a variable can be referenced by name. The scope of an item can be limited to file, function, or block.

Segment
A chunk of data or code that should be mapped to a physical location in memory. The segment can either be placed in RAM (read-and-writeable memory) or in ROM (read-only memory).

Segment map
A set of segments and their locations.

Semaphore
A semaphore is a type of flag that is used for guaranteeing exclusive access to resources. The resource can be a hardware port, a configuration memory, or a set of variables. If several different tasks have to access the same resource, the parts of the code (the critical sections) that access the resource have to be made exclusive for every task. This is done by obtaining the semaphore that protects that resource, thus blocking all other tasks from it. If another task wishes to use the resource, it also has to obtain the semaphore. If the semaphore is already in use, the second task has to wait until the semaphore is released. After the semaphore is released, the second task is allowed to execute and can obtain the semaphore for its own exclusive access.

Severity level
The level of seriousness of the diagnostic response from the assembler, compiler, or debugger, when it notices that something is wrong. Typical severity levels are remarks, warnings, errors, and fatal errors. A remark just points to a possible problem, while a fatal error means that the programming tool exits without finishing.

Short addressing
Many microcontrollers have special addressing modes for efficient access to internal RAM and memory mapped I/O. Short addressing is therefore provided as an extended feature by many compilers for embedded systems. See also *Data pointers*.

Side effect
An expression in C or C++ is said to have a side-effect if it changes the state of the system. Examples are assignments to a variable, or using a variable with the post-increment operator. The C and C++ standards state that a variable that is subject to a side-effect should not be used more that once in an expression. As an example, this statement violates that rule:

```
*d++ = *d;
```

Signal
Signals provide event-based communication between tasks. A task can wait for one or more signals from other tasks. Once a task receives a signal it waits for, execution continues. A task in an RTOS that waits for a signal does not use any processing time, which allows other tasks to execute.

Simulator
A debugging tool that runs on the host and behaves as similar to the target processor as possible. A simulator is used to debug the application when the hardware is unavailable, or not needed for proper debugging. A simulator is usually not connected to any physical peripheral devices. A simulated processor is often slower, or even much slower, than the real hardware.

Single stepping
Executing one instruction or one C statement at a time in the debugger.

Skeleton code
An incomplete code framework that allows the user to specialize the code.

Special function register (SFR)
A register that is used to read and write to the hardware components of the microcontroller.

Stack frames
Data structures containing data objects as preserved registers, local variables, and other data objects that need to be stored temporary for a particular scope (usually a function).

Earlier compilers usually had a fixed size and layout on a stack frame throughout a complete function, while modern compilers might have a very dynamic layout and size that can change anywhere and anytime in a function.

Stack segments
The segment or segments that reserve space for the stack(s). Most processors use the same stack for calls and parameters, but some have separate stacks.

Statically allocated memory
This kind of memory is allocated once and for all at link-time, and remains valid all through the execution of the application. Variables that are either global or declared static are allocated this way.

Static object
An object whose memory is allocated at link-time and is created during system startup (or at first use). Compare *Dynamic object*.

Static overlay
Instead of using a dynamic allocation scheme for parameters and auto variables, the linker allocates space for parameters and auto variables at link time. This generates a worst-case scenario of stack usage, but might be preferable for small chips with expensive stack access or no stack access at all.

Structure value
A collecting names for structs and unions. A struct is a collection of data object placed sequentially in memory (possibly with pad bytes between them). A union is a collection of data sharing the same memory location.

Symbol
A name that represents a register, an absolute value, or a memory address (relative or absolute).

Symbolic location
A location that uses a symbolic name because the exact address is unknown.

T

Target
1. An architecture. 2. A piece of hardware. The particular embedded system you are developing the application for. The term is usually used to distinguish the system from the host system.

Task (thread)
A task is an execution thread in a system. Systems that contain many tasks that execute in parallel are called multitasking systems. Because a processor only executes one instruction stream at the time, most systems implement some sort of task-switch mechanism (often called context switch) so that all tasks get their share of processing time. The process of determining which task that should be allowed to run next is called scheduling. Two common scheduling methods are *Pre-emptive multitasking* and *Round Robin*.

Tentative definition
A variable that can be defined in multiple files, provided that the definition is identical and that it is an absolute variable.

Terminal I/O
A simulated terminal window in the IAR C-SPY Debugger.

Timeslice
The (longest) time an RTOS allows a task to run without running the task-scheduling algorithm. It is possible that a task will be allowed to execute during several consecutive timeslices before being switched out. It is also possible that a task will not be allowed to use its entire time slice, for example if, in a preemptive system, a higher priority task is activated by an interrupt.

Timer
A peripheral that counts independent of the program execution.

Translation unit
A source file together with all the header files and source files included via the preprocessor directive #include, with the exception of the lines skipped by conditional preprocessor directives such as #if and #ifdef.

Trap
A trap is an interrupt initiated by inserting a special instruction into the instruction stream. Many systems use traps to call operating system functions. Another name for trap is software interrupt.

Type qualifiers
In standard C/C++, const or volatile. IAR compilers usually add target-specific type qualifiers for memory and other type attributes.

U

UBROF (Universal Binary Relocatable Object Format)
File format produced by the IAR Systems programming tools.

V

Virtual address (logical address)
An address that needs to be translated by the compiler, linker or the runtime system into a physical memory address before it is used. The virtual address is the address seen by the application, which can be different from the address seen by other parts of the system.

Virtual space
An IAR Embedded Workbench Editor feature which allows you to place the insertion point outside of the area where there are actual characters.

Volatile storage
Data stored in a volatile storage device is not retained when the power to the device is turned off. In order to preserve data during a power-down cycle, you should store it in non-volatile storage. This should not be confused with the C keyword volatile. Compare *Non-volatile storage*.

von Neumann architecture
A computer architecture where both instructions and data are transferred over a common data channel. Compare *Harvard architecture*.

W

Watchpoints
Watchpoints keep track of the values of C variables or expressions in the C-SPY Watch window as the application is being executed.

X

XAR options
The set of commands that control how the IAR XAR Library Builder operates.

XLIB options
The set of commands that control how the IAR XLIB Librarian operates.

XLINK options

Parameters you can specify to change the default behavior of the IAR XLINK Linker.

Z

Zero-overhead loop

A loop in which the loop condition, including branching back to the beginning of the loop, does not take any time at all. This is usually implemented as a special hardware feature of the processor and is not available in all architectures.

Zone

Different processors have widely differing memory architectures. *Zone* is the term C-SPY uses for a named memory area. For example, on processors with separately addressable code and data memory there would be at least two zones. A processor with an intricate banked memory scheme might have several zones.

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Symbols

Numerics