8051 IAR Embedded Workbench™

User Guide

for the

8051 Family of Microcontrollers

COPYRIGHT NOTICE

DISCLAIMER

TRADEMARKS

EDITION NOTICE

First edition: January 2001

Part number: U8051-1

Contents

Tables

Figures

Preface

Welcome to the 8051 IAR Embedded Workbench™ User Guide. The purpose of this guide is to help you fully utilize the features in the 8051 IAR Embedded Workbench with its integrated Windows development tools for the 8051 microcontroller. The IAR Embedded Workbench is a very powerful Integrated Development Environment (IDE) that allows you to develop and manage a complete embedded application project.

The user guide includes comprehensive information about installation and product overviews, as well as tutorials that can help you get started.

Who should read this guide

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

- The C programming language
- The IAR 8051 assembly language
- The architecture and instruction set of the 8051 microcontroller (refer to the chip manufacturer's documentation for information about the 8051 architecture and instruction set)
- Windows 95/98/2000 or Windows NT menus, windows, and dialog boxes.

Refer to the *8051 IAR C Compiler Reference Guide*, *8051 IAR Assembler Reference Guide*, and *IAR XLINK Linker™ and IAR XLIB Librarian™ Reference Guide* for more information about the development tools incorporated in the IAR Embedded Workbench.

How to use this guide

If you are new to using this product, we suggest that you read *Part 1: The IAR development tools* to give you a complete overview of the tools and the functions that the IAR Embedded Workbench can offer.

If you already have had some experience using the IAR Embedded Workbench, but need refreshing on how to work with the IAR development tools, *Part 2: Tutorials* is a good place to begin.

If you are an experienced user and need this guide as more of a reference, see the reference sections in *Part 3: The IAR Embedded Workbench* and *Part 4: The C-SPY simulator*.

What this guide contains

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

Part 1: The IAR development tools

This section provides a general overview of all the IAR development tools so that you can become familiar with all the products' functions and features.

- *Introduction* lists the system requirements, explains how to run the IAR Embedded Workbench with the IAR C-SPY Debugger, describes the directory structure and the type of files it contains, and includes an overview of the documentation supplied with the IAR development tools.
- *The IAR Embedded Workbench* provides a brief summary and lists the features offered in each of the IAR Systems development tools—IAR Embedded Workbench™, IAR C Compiler, IAR Assembler, IAR XLINK Linker ™, IAR XLIB Librarian™ and IAR C-SPY® Debugger—for the 8051 microcontroller.
- *The project model* describes how you can organize a project using the IAR Embedded Workbench by specifying the different target versions of the application that you want to build, creating groups to a particular target, and keeping track of source files. The chapter also explains how configuration options are managed and related to the project.

Part 2: Tutorials

These tutorials allow you to have hands-on training in order to help you get started with using the tools.

- *IAR Embedded Workbench tutorial* guides you through setting up a new project, compiling the program, examining the list file, linking the program and debugging it. The tutorial demonstrates a typical development cycle using the IAR Embedded Workbench, the 8051 IAR C Compiler, and the IAR XLINK Linker™. There is also a brief introduction about the IAR C-SPY Debugger.
- *Compiler tutorials* demonstrates how to utilize the features in the IAR Embedded Workbench and IAR C-SPY Debugger to develop a series of typical programs for the 8051 IAR C Compiler. The first tutorial shows how to utilize 8051 peripherals with IAR C Compiler features. In the second tutorial, the tutorial project is modified by adding an interrupt handler and then run using the C-SPY interrupt system in conjunction with complex breakpoints and macros.
- *Assembler tutorials* illustrates how you use the IAR Embedded Workbench and the IAR C-SPY Debugger to develop machine-code programs by using some of the most important features of the 8051 IAR Assembler. This chapter also introduces you to the IAR XLIB Librarian™, which helps to maintain files of library modules.

- *Advanced tutorials* is divided into several tutorials that explore some of the more advanced features of the IAR C-SPY Debugger, such as how to define complex breakpoints, profile the application, display code coverage, and debug in disassembly mode. The tutorials also illustrate how you can effectively use both code written for the 8051 IAR C Compiler and code written for the 8051 IAR Assembler in the same project.
- *ROM-monitor tutorial* shows how to set up options in the IAR Embedded Workbench before running a project in the ROM-monitor.

Part 3: The IAR Embedded Workbench

This section describes how to specify options in the IAR Embedded Workbench. The options are divided by category:

- *General options* specifies target processor and memory module options, as well as set up output directories.
- *Compiler options* specifies compiler options for language, code, optimizations, output, list file, preprocessor, and diagnostics.
- *Assembler options* shows how to set assembler options for code generation, debugging, preprocessor, and list file generation.
- *XLINK options* shows how to set XLINK options for output, defining symbols, diagnostics, list generation, setting up of the include path for linker command files, input, and processing.
- *C-SPY options* shows how to choose setup, serial communication, and ROM-monitor options for all tools or a specified tool.
- *IAR Embedded Workbench reference* contains a detailed reference for the IAR Embedded Workbench, such as details about the graphical user interface.

Part 4: The C-SPY simulator

This section goes in depth about the IAR C-SPY® Debugger, which is a powerful interactive debugger for embedded applications.

- *Introduction to C-SPY* gives an overview of the functions for debugging projects provided in this tool.
- *C-SPY expressions* defines the syntax of the expressions and variables used in C-SPY macros and gives examples about how to use macros in debugging.
- *C-SPY macros* lists the built-in system macros supplied with the IAR C-SPY Debugger.
- *Device description file* describes the purpose of the device description file.
- *C-SPY reference* provides detailed reference information about the C-SPY graphical user interface.
- *C-SPY command line options* gives information about how you set C-SPY options from the command line and lists a summary of command line options and setup macros.

Part 5: C-SPY for the 8051 ROM-monitor

- *Introduction to the ROM-monitor* describes the additional features of the C-SPY ROM-monitor version, as well as the differences between the ROM-monitor and simulator versions of the 8051 C-SPY debugger.
- *Controlling user applications* describes the additional options in the Embedded Workbench, and the command-line options used by the ROM-monitor version of C-SPY.
- *The ROM-monitor boards* explains how to avoid conflicts between the ROM-monitor and your own programs.
- *The ROM-monitor program* gives a brief, target independent, description of how a ROM-monitor works.
- *Advanced topics* describes more advanced use of the ROM-monitor, including executing transparent commands and compiling a modified ROM-monitor.
- *Diagnostic messages* lists the error and warning messages that the 8051 C-SPY ROM-monitor version can produce.

Other documentation

The complete set of IAR Systems development tools for the 8051 microcontroller are described in a series of guides. For information about:

- Configuring and programming for the 8051 IAR C Compiler, refer to the *8051 IAR C Compiler Reference Guide*
- Programming for the 8051 IAR Assembler, refer to the *8051 IAR Assembler Reference Guide*
- Using the IAR XLINK Linker and the IAR XLIB Librarian, refer to the *IAR XLINK Linker™ and IAR XLIB Librarian™ Reference Guide*
- Using the IAR C Library, refer to the *IAR C Library Functions Reference Guide*.

All of these guides are delivered in PDF format on the installation media. Some of them are also delivered as printed books.

Document conventions

This book uses the following typographic conventions:

Style	Used for
`computer`	Text that you enter or that appears on the screen.
`parameter`	A label representing the actual value you should type as part of a command.
`[option]`	An optional part of a command.
`{a | b | c}`	Alternatives in a command.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
reference	A cross-reference within or to another guide.
🛠	Identifies instructions specific to the IAR Embedded Workbench versions of the IAR development tools.
>_	Identifies instructions specific to the command line versions of the IAR development tools.

Table 1: Typographic conventions used in this guide

Part 1: The IAR development tools

This part of the 8051 IAR Embedded Workbench™ User Guide includes the following chapters:

- Introduction
- The IAR Embedded Workbench
- The project model.

Introduction

To help you get started, this chapter lists the system requirements and shows you how to run the IAR Embedded Workbench. It also describes which directories are created 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*, which is delivered with the product, for information about how to install and register the IAR products.

Included in this package

The IAR Systems development tools package for the 8051 microcontroller contains the following items:

- Installation media
- *QuickStart Card*
- User documentation in printed or in PDF format. For detailed information about what guides are included, see *Documentation*, page 7.

System requirements

The IAR Systems development tools for the 8051 microcontroller run under Windows 95/98/2000, or Windows NT 4 or later.

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

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

Running the program

RUNNING THE IAR EMBEDDED WORKBENCH

Select the **Start** button in the taskbar and select **Programs**. Select **IAR Systems** in the menu. Then select **IAR Embedded Workbench for 8051** and **IAR Embedded Workbench** to run the `ew23.exe` program which is located in the installation root directory.

Note: The product version number may become updated in a future release. Refer to the `readme.htm` file for the most recent product information.

RUNNING THE IAR C-SPY DEBUGGER

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

You can also start C-SPY from the **Programs** menu. Select **IAR Systems** in the menu. Then select **IAR Embedded Workbench for 8051** and **IAR C-SPY** to run the `cw23.exe` program which is located in the installation root directory.

It is also possible to start C-SPY by using the Windows **Run...** command, specifying options. See *C-SPY command line options*, page 247, for additional information about command line options.

UPGRADING TO A NEW VERSION

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

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

UNINSTALLING THE PRODUCTS

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

Directory structure

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

THE ROOT DIRECTORY

The `iar systems\ew23\` directory is the root directory created by the default installation procedure. The executable files for the IAR Embedded Workbench and the IAR C-SPY Debugger are located here.

The root directory also contains the `8051` directory, where all product-specific subdirectories are located.

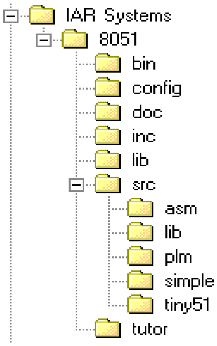


Figure 1: Directory structure

If you already have an `ew23.exe` file installed, the installation program will suggest to use its root directory also for the installation of the 8051 IAR development tools.

THE BIN DIRECTORY

The `bin` subdirectory contains executable files such as `exe` and `dll` files, the C-SPY driver, and the 8051 help files.

THE CONFIG DIRECTORY

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

THE DOC DIRECTORY

The `doc` subdirectory contains read-me files (`*.htm` or `*.txt`) with recent additional information about the 8051 tools. It is recommended that you read all of these files before proceeding. The directory also contains online versions (PDF format) of this user guide, and of the 8051 reference guides.

THE INC DIRECTORY

The `inc` subdirectory holds include files, such as the header files for the standard C library, as well as a specific header file defining special function registers (SFRs).

THE LIB DIRECTORY

The `lib` subdirectory holds library modules used by the compiler.

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

THE LICENSE DIRECTORY

The `license` subdirectory holds the IAR Systems License Manager utility.

THE SRC DIRECTORY

The `src\asm` contains assembly source files for testing and trying out the assembler.

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

The `src\plm` subdirectory contains example files for running the compiler together with PL/M functions.

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

The `src\tiny51` subdirectory contains source files for the Tiny51 real-time kernel for the 8051 microcontroller.

THE TUTOR DIRECTORY

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

File types

The 8051 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
a03	Target program	XLINK	EPROM, C-SPY, etc.
c	C program source	Text editor	Compiler
d03	Target program with debug information	XLINK	C-SPY and other symbolic debuggers
ddf	Device description file	Text editor	C-SPY
h	C header source	Text editor	Compiler `#include`
i	Preprocessed code	Compiler	Compiler

Table 2: File types

Ext.	Type of file	Output from	Input to
inc	Assembler header	Text editor	Assembler `#include` file
lst	List	Compiler and assembler	–
mac	C-SPY macro definition	Text editor	C-SPY
prj	IAR Embedded Workbench project	IAR Embedded Workbench	IAR Embedded Workbench
r03	Object module	Compiler and assembler	XLINK and XLIB
s03	Assembler program source	Text editor	Assembler
xcl	Extended command	Text editor	XLINK
xlb	Librarian 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 `cfg` are created dynamically when you install and run the IAR Embedded Workbench tools. These files contain information about your configuration and other settings.

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

Documentation

This section briefly describes the information that is available in the 8051 user and reference guides, in the online help, and on the Internet.

USER AND REFERENCE GUIDES

The 8051 user and reference guides are all available in PDF format and can easily be accessed from the **Help** menu in the IAR Embedded Workbench or from the `readme.htm` file. Hypertext links are implemented throughout the documents and you can easily find what you are looking for via the table of contents or index.

8051 IAR Embedded Workbench™ User Guide

This guide.

8051 IAR C Compiler Reference Guide

This guide provides reference information about the 8051 IAR 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 available data types
- The run-time libraries
- The IAR language extensions.

8051 IAR Assembler Reference Guide

This guide provides reference information about the 8051 IAR Assembler. This includes details of the assembler source format and reference information about the assembler operators, directives, mnemonics, and diagnostics.

IAR XLINK Linker™ and IAR XLIB Librarian™ Reference Guide

This guide contains introductory and reference information about the IAR XLINK Linker and the IAR XLIB Librarian. It describes the linker functions, and provides details about segment control and the XLINK output formats. This guide is particularly useful when you assign memory locations.

IAR C Library Functions Reference Guide

This guide describes the functions of the C library that is provided with the product.

ONLINE HELP

From the **Help** menu in the IAR Embedded Workbench and the IAR C-SPY Debugger, you can access the 8051 online documentation. Context-sensitive help is also available via the F1 key in the IAR Embedded Workbench and C-SPY windows and dialog boxes.

RECENT INFORMATION

We recommend that you read the `readme.htm` file for recent information that may not be included in the user guides. This file is available from the **Start** menu in Windows, and contains links to readme files for all components in the product package. The readme files are located in the `\8051\Doc` directory.

IAR ON THE WEB

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

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

The IAR Embedded Workbench

The IAR Embedded Workbench™ is a powerful Integrated Development Environment (IDE) that allows you to develop and manage complete embedded application projects. It is a true 32-bit Windows environment with all the features you would expect to find in your daily workplace.

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

The framework

The IAR Embedded Workbench is the framework where all necessary tools are seamlessly integrated. Support for a large number of microprocessors and microcontrollers can be added into the IAR Embedded Workbench; This allows you to change processors for the next project yet still work within the same, familiar environment.

The IAR Embedded Workbench also promotes a useful working methodology, thereby significantly reducing development time which is achieved using IAR tools. We refer to this concept as: "Different Architectures. One Solution". The IAR Embedded Workbench is available for a large number of microcontrollers and microprocessors in the 8-, 16-, and 32-bit segments. It provides an easy-to-learn and highly efficient development environment with maximum code inheritance capabilities and comprehensive and specific target support.

INTEGRATED TOOLS

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

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

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

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

IAR Embedded Workbench

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

FEATURES

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

General features

The IAR Embedded Workbench provides the following general features:

- Runs under Windows 95/98/2000, or Windows NT 4 or later.
- Intuitive user interface, taking advantage of Windows 95/98/2000 features.
- Hierarchical project representation.
- Full integration between the IAR Embedded Workbench tools and editor.
- Binary File Editor with multi-level undo and redo.
- Support of drag-and-drop features.

The IAR Embedded Workbench editor

The IAR Embedded Workbench Editor provides the following features:

- Syntax of C programs shown using text styles and colors.
- Powerful search and replace commands, including multi-file search.
- Direct jump to context from error listing.
- Parenthesis matching.
- Automatic indentation.
- Multi-level undo and redo for each window.

Compiler and assembler projects

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

- Projects build in the background, allowing simultaneous editing.

- Options can be set globally, on groups of source files, or on individual source files.
- The Make utility recompiles, reassembles, and links files only when necessary.
- Generic and 8051-specific optimization techniques produce very efficient machine code.

Documentation

The 8051 IAR Embedded Workbench is documented in the *8051 IAR Embedded Workbench™ User Guide* (this guide). There is also context-sensitive help and hypertext versions of the user documentation available online.

IAR C Compiler

The IAR C Compiler for the 8051 microcontroller offers the standard features of the C language, plus many extensions designed to take advantage of the 8051-specific facilities.

The 8051 IAR C Compiler is integrated with other IAR Systems software for the 8051 microcontroller. It is supplied with the IAR 8051 Assembler, with which it shares linker and librarian manager tools.

FEATURES

The following section describes the features of the 8051 IAR C Compiler.

Language facilities

- Conformance to the ISO/ANSI standard for a free-standing environment.
- Standard library of functions applicable to embedded systems, with source code optionally available.
- IEEE-compatible floating-point arithmetic.
- Object code can be linked with assembly routines.
- Interrupt functions can be written in C.
- Powerful extensions for 8051-specific features, including efficient I/O.
- Long identifiers—up to 255 significant characters.
- Up to 32,000 external symbols.

Type checking

- External references are type-checked at link time.
- Fast compilation by memory-based design which avoids temporary files or overlays.
- Linkage of user code with assembly routines.

- Extensive type checking at compile time.
- Extensive module interface type checking at link time.
- LINT-like checking of program source.
- Link-time inter-module consistency checking of the run-time module.
- Maximum compatibility with other IAR Systems compilers.

Code generation

- Selectable optimization for code size or execution speed.
- Comprehensive output options, including relocatable object code, assembly source code, and C list files with optional assembler mnemonics.
- Easy-to-understand error and warning messages.
- Compatibility with IAR C-SPY (see *8051 IAR C Compiler Reference Guide*).
- Generation of fully PROMable code without language restrictions.
- Support for PL/M.

Target support

- Flexible variable allocation.
- `#pragma` directives to maintain portability while using processor-specific extensions.
- Tiny, small, compact, medium, large, and banked memory models.
- Interrupt functions require no assembly language.
- Intrinsic functions.

Documentation

The 8051 IAR C Compiler is documented in the *8051 IAR C Compiler Reference Guide.*

IAR Assembler

The 8051 IAR Assembler is a powerful relocating macro assembler with a versatile set of directives.

FEATURES

The 8051 IAR Assembler provides the following features:

- Integration with other IAR Systems software for the 8051 microcontroller.
- Built-in C language preprocessor.
- Extensive set of assembler directives and expression operators.
- Conditional assembly.
- Powerful recursive macro facilities supporting the Intel/Motorola style.

- List file with augmented cross-reference output.
- Number of symbols and program size limited only by available memory.
- Support for complex expressions with external references.
- Up to 256 relocatable segments per module.
- 255 significant characters in symbol names.
- 32-bit arithmetic.

Documentation

The 8051 IAR Assembler is documented in the *8051 IAR Assembler Reference Guide*.

IAR XLINK Linker

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

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

The final output produced by the IAR XLINK Linker is an absolute, target-executable object file that can be downloaded to the 8051 microcontroller or to a hardware emulator.

FEATURES

The IAR XLINK Linker offers the following important features:

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

Documentation

The IAR XLINK Linker is documented in the *IAR XLINK Linker™ and IAR XLIB Librarian™ Reference Guide*, which is delivered in PDF format on the installation media.

IAR XLIB Librarian

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

FEATURES

The IAR XLIB Librarian provides the following features:

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

Documentation

The IAR XLIB Librarian is documented in the *IAR XLINK Linker™ and IAR XLIB Librarian™ Reference Guide*, which is delivered in PDF format on the installation media.

IAR C-SPY Debugger

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

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

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

FEATURES

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

General

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

- Intuitive user interface, taking advantage of Windows 95/98/2000 features.
- Source and disassembly mode debugging.
- C and assembly source debugging.
- Fast simulator.
- Log file option.
- Powerful macro language.
- Complex code and data breakpoints.
- Memory validation.
- Interrupt simulation.
- UBROF, INTEL-EXTENDED, and Motorola input formats supported.

High-level-language debugging

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

Assembler-level debugging

- Full support for auto and register variables.
- Built-in assembler/disassembler.

Documentation

The IAR C-SPY Debugger is documented in the *8051 IAR Embedded Workbench™ User Guide* (this guide). There is also context-sensitive help available online.

VERSIONS

The IAR C-SPY Debugger for the 8051 microcontroller is currently available in simulator and ROM-monitor versions.

Simulator version

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

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

ROM-monitor version

The ROM-monitor version of C-SPY provides a low-cost solution to debugging. It is available for standard evaluation boards and can be modified for customer hardware. It allows true real-time debugging at a low cost.

If you are using the ROM-monitor version of C-SPY, refer to the *Part 5: C-SPY for the 8051 ROM-monitor* for additional information.

The project model

This chapter briefly discusses the project model used by the IAR Embedded Workbench. It covers how projects are organized and how you specify targets, groups, source files, and options so that you can better handle different versions of your applications.

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

Developing projects

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

HOW PROJECTS ARE ORGANIZED

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

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

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

Targets

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

Groups

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

Source files

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

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

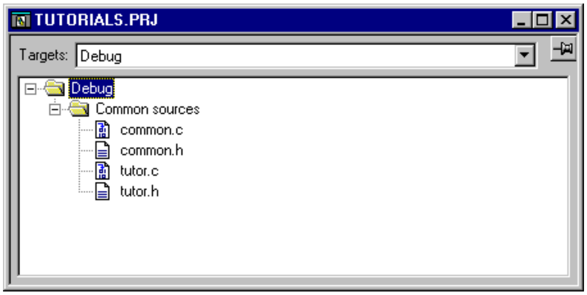


Figure 2: Project window

SETTING OPTIONS

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

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

For an example where different options are set on file level, see *Tutorial 8*, page 81. For information about how to set options, see the chapters *Compiler options* and *Assembler options* in *Part 3: The IAR Embedded Workbench* in this guide.

BUILDING A PROJECT

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

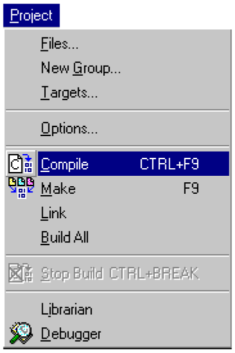


Figure 3: Compile command in the Project menu

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

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

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

TESTING THE CODE

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

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

SAMPLE APPLICATIONS

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

A basic application

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

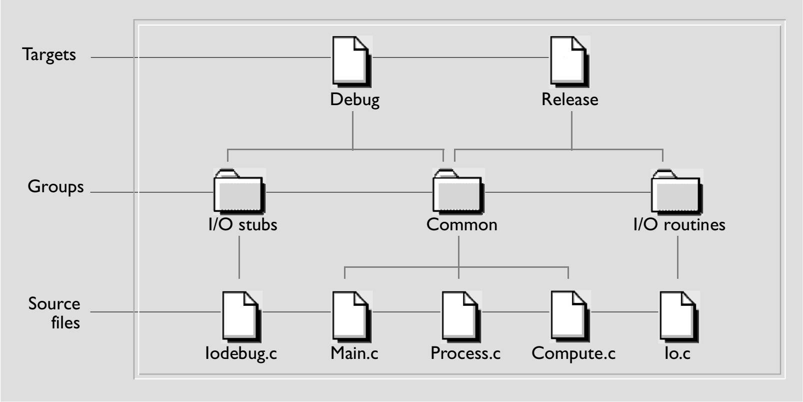


Figure 4: Basic application

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

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

A more complex project

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

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

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

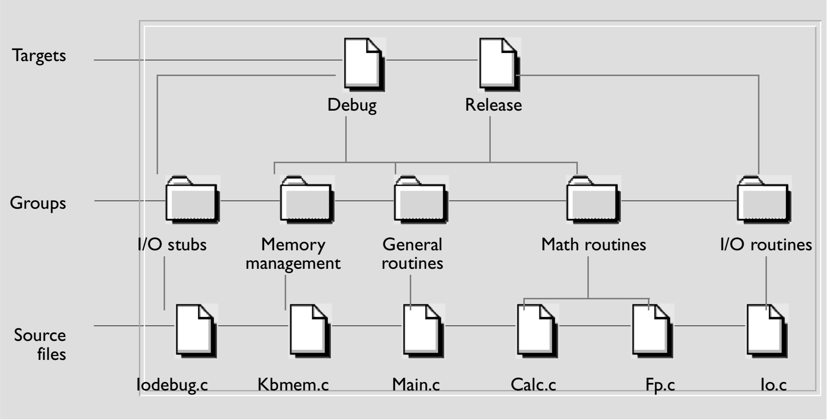


Figure 5: Complex application

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

Part 2: Tutorials

This part of the 8051 IAR Embedded Workbench™ User Guide contains the following chapters:

- IAR Embedded Workbench tutorial
- Compiler tutorials
- Assembler tutorials
- Advanced tutorials
- ROM-monitor tutorial.

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

Notice that it may be helpful to return to the chapter *The project model* while running the tutorials. Moreover, remember that the first tutorial, the IAR Embedded Workbench tutorial, contains descriptions of many basic procedures. Even if you are only using the IAR Assembler, you should read through the IAR Embedded Workbench tutorial first.

IAR Embedded Workbench tutorial

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

The following steps demonstrate a typical development cycle:

- Creating a project, adding source files to it, and specifying target options
- Compiling the program, examining the list file, and linking the program
- Running the program in the IAR C-SPY Debugger.

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

Tutorial 1

We recommend that you create a specific directory where you can store all your project files, for example the `8051\projects` directory.

CREATING A NEW PROJECT

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

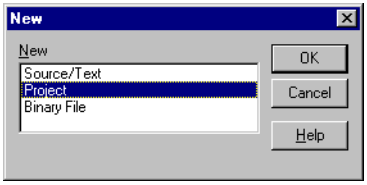
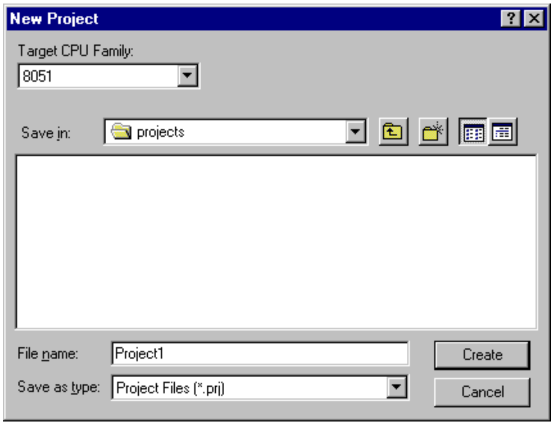


Figure 6: Creating a new project

The **Help** button provides access to information about the IAR Embedded Workbench.

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

Enter `Project1` in the **File name** box, and set the **Target CPU Family** to **8051**. Specify where you want to place your project files, for example in a `projects` directory:



Target CPU Family: 8051

Save in: projects

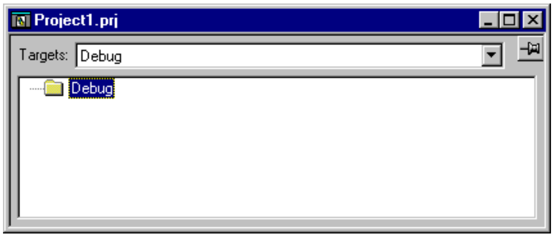
File name: Project1

Save as type: Project Files (*.prj)

Figure 7: New Project dialog box

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

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



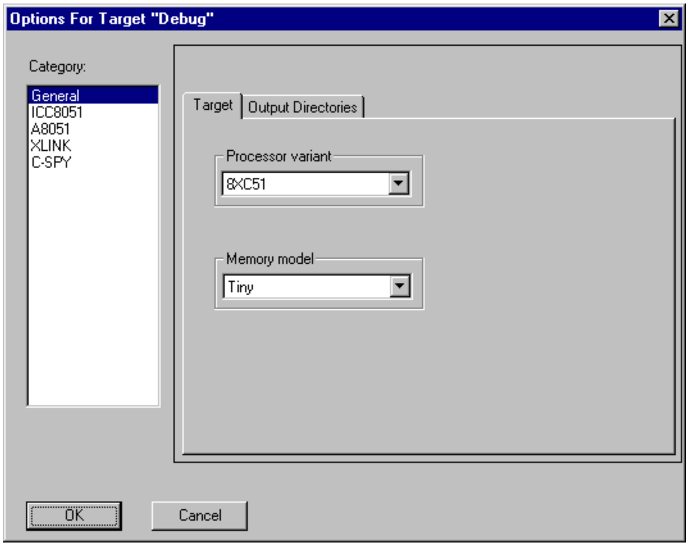
Targets: Debug

Figure 8: Project window

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

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

In this tutorial we use the default settings. Make sure that the **Processor variant** is set to **8XC51** and that the **Memory model** is set to **Tiny**:



Processor variant: 8XC51

Memory model: Tiny

Figure 9: Target settings

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

THE SOURCE FILES

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

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

The tutor.c program

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

```
#include "tutor.h"

int call_count;

/*
    Increase the 'call_count' variable.
    Get and print the associated Fibonacci number.
*/
void do_foreground_process(void)
{
  unsigned int fib;
  ++call_count;
  fib = get_fibonacci( call_count );
  put_value( fib );
}


/*
    Main program for tutor1.
    Prints the Fibonacci numbers.
*/
void main(void)
 {
 call_count=0;

 init_fibonacci();

 while (  call_count < MAX_FIBONACCI )
   do_foreground_process();
 }
```

The common.c program

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

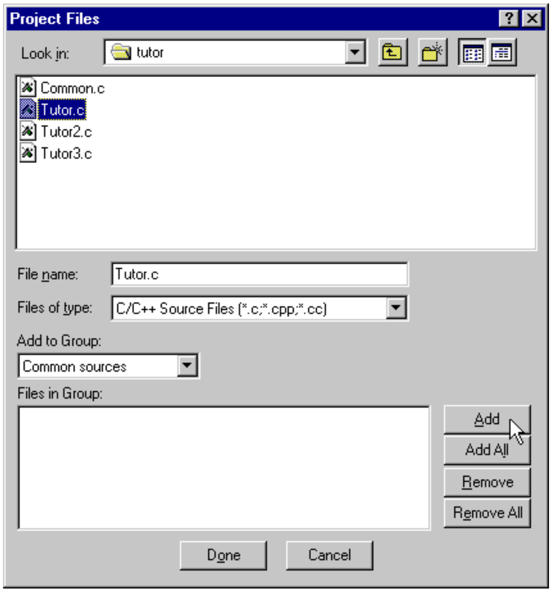
```
#include <stdio.h>
#include "common.h"


static unsigned int fibonacci[MAX_FIBONACCI];

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

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

  for ( i=2 ; i<MAX_FIBONACCI ; ++i)
```

```
    fibonacci[i] = fibonacci[i-1] + fibonacci[i-2];
}

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

  return fibonacci[index];
}

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

  p = buf;
  *p++ = 0;

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

  *p++ = '\n';

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

ADDING FILES TO THE PROJECT

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

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



Look in: tutor

File name: Tutor.c

Files of type: C/C++ Source Files (*.c;*.cpp;*.cc)

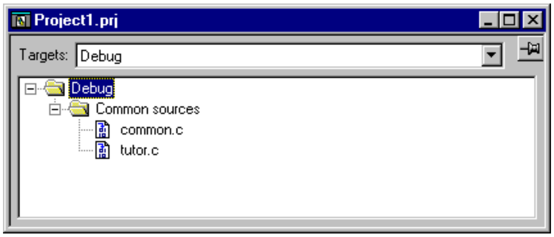
Add to Group: Common sources

Figure 10: Adding files to project

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

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

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



Targets: Debug

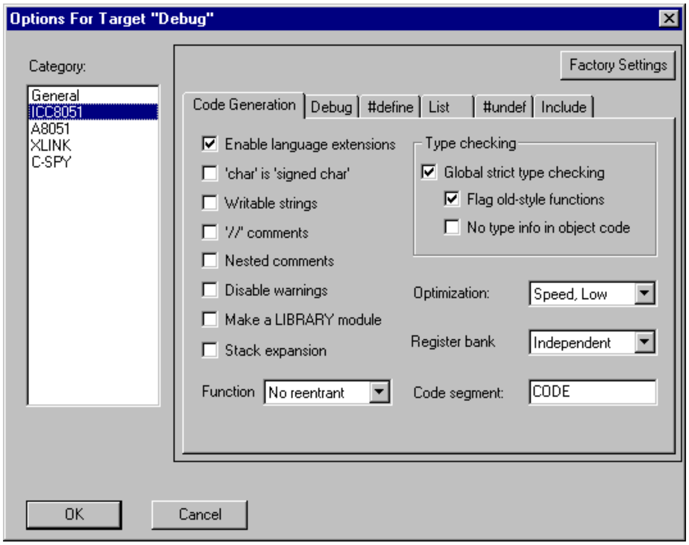
Figure 11: Displaying files in the Project window

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

SETTING COMPILER OPTIONS

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

Select the **Debug** folder icon in the Project window, choose **Options…** from the **Project** menu, and select **ICC8051** in the **Category** list to display the IAR C Compiler options pages:



☑ Enable language extensions
☐ 'char' is 'signed char'
☐ Writable strings
☐ '//' comments
☐ Nested comments
☐ Disable warnings
☐ Make a LIBRARY module
☐ Stack expansion
Function: No reentrant

Type checking
☑ Global strict type checking
☑ Flag old-style functions
☐ No type info in object code

Optimization: Speed, Low
Register bank: Independent
Code segment: CODE

Figure 12: Setting compiler options

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

Page	Options
Code Generation	Enable language extensions Global strict type checking Flag old-style functions Optimizations, Size: Low
Debug	Generate debug information File references Debuggable register handling
List	List file Insert mnemonics

Table 3: Tutorial 1 compiler options

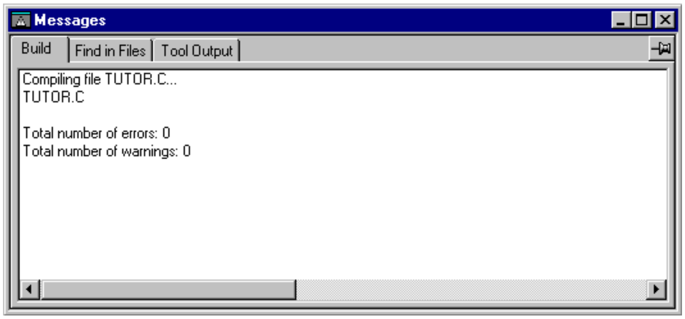
When you have made these selections, click **OK** to set the options you have specified.

COMPILING THE TUTOR.C AND COMMON.C FILES

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

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

The progress will be displayed in the Messages window:



```
Compiling file TUTOR.C...
TUTOR.C

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

Figure 13: Compilation message

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

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

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

- In the `list` directory, your list files from the `Debug` target will be placed. The list files have the extension `lst` and will be located here.
- In the `obj` directory, the object files from the compiler and the assembler will be placed. These files have the extension `r03` and will be used as input to the linker.
- In the `exe` directory, you will find the executable files. These files have the extension `d03` and will be used as input to the IAR C-SPY Debugger. Notice that this directory will be empty until you have linked the object files.

VIEWING THE LIST FILE

Open the list file `tutor.lst` by selecting **Open…** from the **File** menu, and selecting `tutor.lst` from the `debug\list` directory to examine the list file.

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

```
##########################################################################
 IAR 8051 C-Compiler VN.nnX/WIN
      Compile time  =  dd/Mmm/yyyy  hh:mm:ss
      Target option =  8051
      Memory model  =  tiny
      Source file   =  c:\iar\ew23\8051\tutor\tutor.c
      List file     =  c:\iar\ew23\8051\projects\debug\list\tutor.lst
      Object file   =  c:\iar\ew23\8051\projects\debug\obj\tutor.r03
      Command line  =  -v0 -mt -OC:\IAR\EW23\8051\Projects\Debug\Obj\ -e
                       -gA -z3 -RCODE
                       -LC:\IAR\EW23\8051\Projects\Debug\List\ -q -t8
                       -IC:\IAR\EW23\8051\inc\ -r0r
                       C:\IAR\EW23\8051\tutor\Tutor.c
                          Copyright 1999 IAR Systems. All rights reserved.
##########################################################################
```

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

```
   \   0000                    NAME    tutor(16)
   \   0000                    RSEG    CODE(0)
   \   0000                    RSEG    D_UDATA(0)
   \   0000                    PUBLIC  call_count
   \   0000                    PUBLIC  do_foreground_process
   \   0000                    $DEFFN
do_foreground_process(2,0,0,0,32768,0,0,0),get_fibonacci,put_value
   \   0000                    EXTERN  get_fibonacci
   \   0000                    $DEFFN  get_fibonacci(32769,0,0,0)
   \   0000                    EXTERN  init_fibonacci
   \   0000                    $DEFFN  init_fibonacci(32768,0,0,0)
   \   0000                    PUBLIC  main
   \   0000                    $DEFFN
main(0,0,0,0,32768,0,0,0),init_fibonacci,do_foreground_process
   \   0000                    EXTERN  put_value
   \   0000                    $DEFFN  put_value(32770,0,0,0)
   \   0000                    EXTERN  ?CL8051T_5_50_L17
   \   0000                    RSEG    CODE
   \   0000            do_foreground_process:
      1          #include "tutor.h"
      2
      3          int call_count;
      4
      5          /*
      6              Increase the ’call_count’ variable.
      7              Get and print the associated Fibonacci number.
```

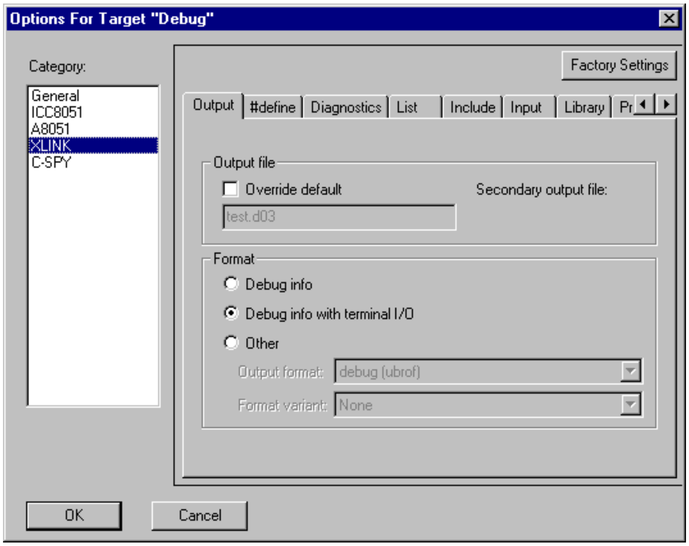
```
  8           */
  9           void do_foreground_process(void)
 10           {
 11             unsigned int fib;
 12             ++call_count;
\   0000  0501              INC     call_count+1
\   0002  E501              MOV     A,call_count+1
\   0004  7002              JNZ     ?0003
\   0006  0500              INC     call_count
\   0008            ?0003:
 13             fib = get_fibonacci( call_count );
\   0008  AC01              MOV     R4,call_count+1
\   000A  120000            LCALL   $REFFN get_fibonacci
 14             put_value( fib );
\   000D  8C00              MOV     $LOCBD do_foreground_process+1,R4
\   000F  8D00              MOV     $LOCBD do_foreground_process,R5
\   0011  120000            LCALL   $REFFN put_value
 15           }
\   0014  22                RET
\   0015            main:
 16
 17
 18           /*
 19               Main program for tutor1.
 20               Prints the Fibonacci numbers.
 21           */
 22           void main(void)
 23            {
 24            call_count=0;
\   0015  E4                CLR     A
\   0016  F501              MOV     call_count+1,A
\   0018  F500              MOV     call_count,A
 25
 26            init_fibonacci();
\   001A  120000            LCALL   $REFFN init_fibonacci
\   001D            ?0001:
 27
 28            while (  call_count < MAX_FIBONACCI )
\   001D  C3                CLR     C
\   001E  E501              MOV     A,call_count+1
\   0020  940A              SUBB    A,#10
\   0022  E500              MOV     A,call_count
\   0024  6480              XRL     A,#128
\   0026  9480              SUBB    A,#128
\   0028  5005              JNC     ?0000
\   002A            ?0002:
 29              do_foreground_process();
\   002A  120000          LCALL   $REFFN do_foreground_process
  30           }
\   002D  80EE            SJMP    ?0001
\   002F          ?0000:
\   002F  22              RET
\   0000                  RSEG    D_UDATA
\   0000          call_count:
\   0002                  DS      2
\   0002                  END
```

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

```
Errors: none
Warnings: none
Code size: 48
Constant size: 0
Static variable size: Data(2) Idata(0) Bit(0) Xdata(0) Pdata(0) Bdata(0)
```

LINKING THE TUTOR.C PROGRAM

First set up the options for the IAR XLINK Linker™. Select the **Debug** folder icon in the Project window and choose **Options…** from the **Project** menu. Then select **XLINK** in the **Category** list to display the XLINK options pages:



☐ Override default

test.d03

○ Debug info

● Debug info with terminal I/O

○ Other

Output format: debug (ubrof)

Format variant: None

Figure 14: Tutorial 1 XLINK options

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

Page	Options
Output	Debug info with terminal I/O
List	Generate linker listing Segment map Module map

Table 4: Tutorial 1 XLINK options

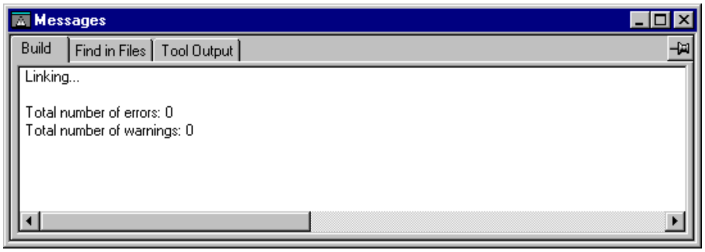
If you want to examine the linker command file, use a suitable text editor, such as the IAR Embedded Workbench editor, or print a copy of the file, and verify that the entries match your requirements.

The definitions in the linker command file are not permanent; they can be altered later on to suit your project if the original choice proves to be incorrect, or less than optimal. For more information about linker command files, see the *Configuration* chapter in the *8051 IAR C Compiler Reference Guide.*

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

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

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



```
Linking...

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

Figure 15: Linking message

The result of the linking is a code file `project1.d03` with debug information and a map file `project1.map`.

Viewing the map file

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

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

Viewing the build tree

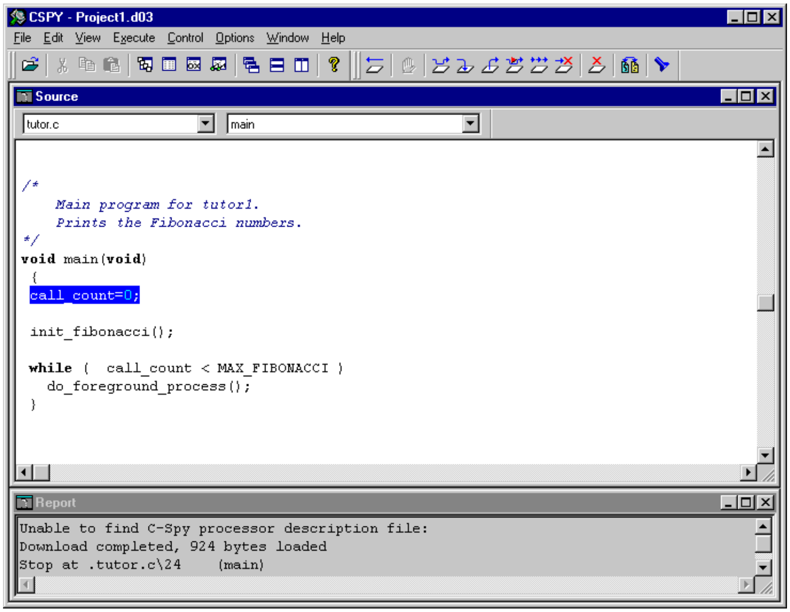
In the Project window, press the right mouse button and select **Save as Text...** from the pop-up menu that appears. This creates a text file that allows you to conveniently examine the options for each level of the project.

Notice that the text file will contain the command line equivalents to the options that you have specified in the IAR Embedded Workbench. The command line options are described in the *8051 IAR C Compiler Reference Guide* and *8051 IAR Assembler Reference Guide*, respectively.

RUNNING THE PROGRAM

Now we will run the `project1.d03` program using the IAR C-SPY Debugger to watch variables, set a breakpoint, and print the program output in the Terminal I/O window.

Choose **Debugger** from the **Project** menu in the IAR Embedded Workbench. Alternatively, click the C-SPY button in the toolbar. The following C-SPY window will be opened for this file:



```
/*
    Main program for tutor1.
    Prints the Fibonacci numbers.
*/
void main(void)
 {
 call_count=0;

 init_fibonacci();

 while (  call_count < MAX_FIBONACCI )
   do_foreground_process();
 }
```

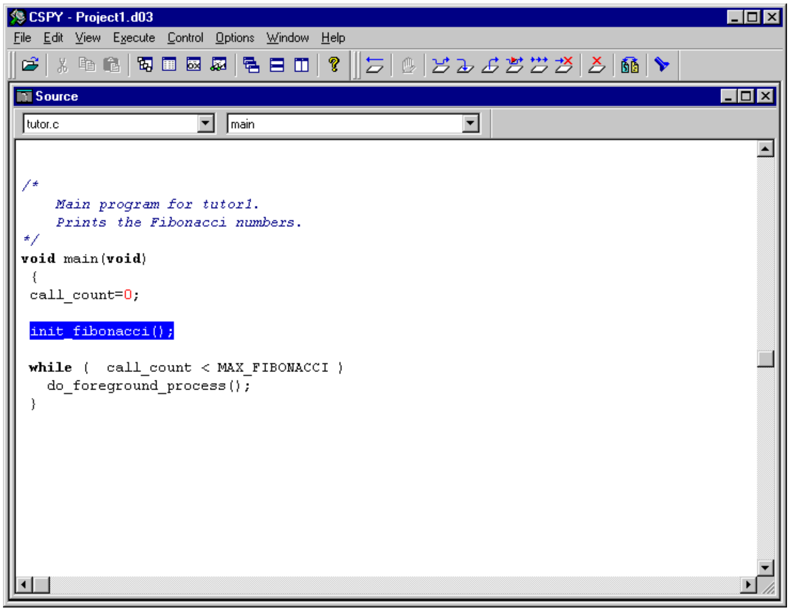
```
Unable to find C-Spy processor description file:
Download completed, 924 bytes loaded
Stop at .tutor.c\24     (main)
```

Figure 16: Starting C-SPY

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

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

Execute one step by choosing **Step** from the **Execute** menu. Alternatively, click the **Step** button in the toolbar; at source level **Step** executes one source statement at a time. The current position should be the call to the `init_fibonacci` function:



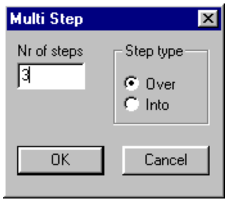
```
/*
    Main program for tutor1.
    Prints the Fibonacci numbers.
*/
void main(void)
 {
 call_count=0;

 init_fibonacci();

 while (  call_count < MAX_FIBONACCI )
   do_foreground_process();
 }
```

Figure 17: Stepping in C-SPY

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

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

To step three times at once, choose **Multi Step** from the **Execute** menu and enter 3.

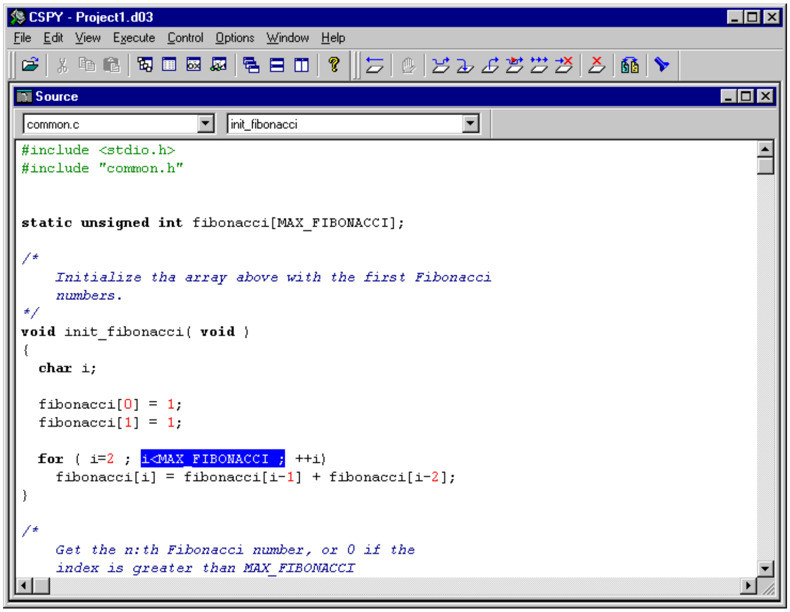


Nr of steps: 3

Step type: ● Over ○ Into

Figure 18: Tutorial 1 Multi Step dialog box

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



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


static unsigned int fibonacci[MAX_FIBONACCI];

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

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

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

/*
    Get the n:th Fibonacci number, or 0 if the
    index is greater than MAX_FIBONACCI
```

Figure 19: Multi step in C-SPY

WATCHING VARIABLES

C-SPY allows you to set watchpoints on C variables or expressions, to allow you to keep track of their values as you execute the program. You can watch variables in a number of ways; for example, you can watch a variable by pointing at it in the Source window with the mouse pointer, or by opening the Locals window.

Alternatively, you can open the QuickWatch window from the pop-up menu that appears when you press the right mouse button in the Source window.

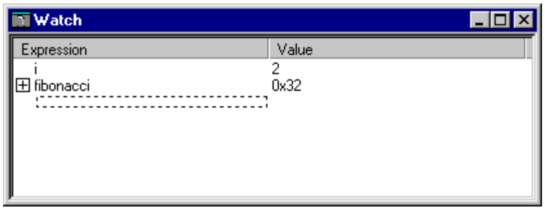
Here we will use the Watch window. Choose **Watch** from the **Window** menu to open the Watch window, or click the **Watch Window** button in the toolbar. If necessary, resize and rearrange the windows so that the Watch window is visible.

Setting a watchpoint

Set a watchpoint on the variable `i` using the following procedure:

Select the dotted rectangle, then click and briefly hold the left mouse button. In the entry field that appears when you release the button, type `i` and press the Enter key.

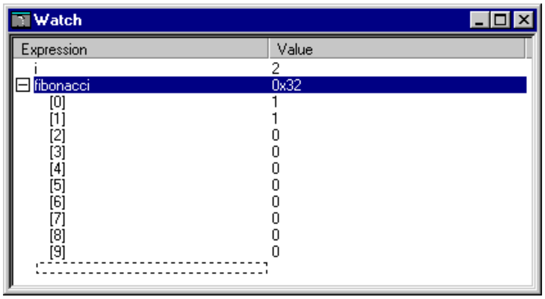
You can also drag and drop a variable in the Watch window. Select the `fibonacci` array in the `init_fibonacci` function. When `fibonacci` is marked, drag and drop it in the Watch window which will show the current value of `i` and `fibonacci`:



Expression	Value
i	2
⊞ fibonacci	0x32

Figure 20: Watching variables in C-SPY

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



Expression	Value
i	2
⊟ fibonacci	0x32
[0]	1
[1]	1
[2]	0
[3]	0
[4]	0
[5]	0
[6]	0
[7]	0
[8]	0
[9]	0

Figure 21: Expanded view of variables in C-SPY

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

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

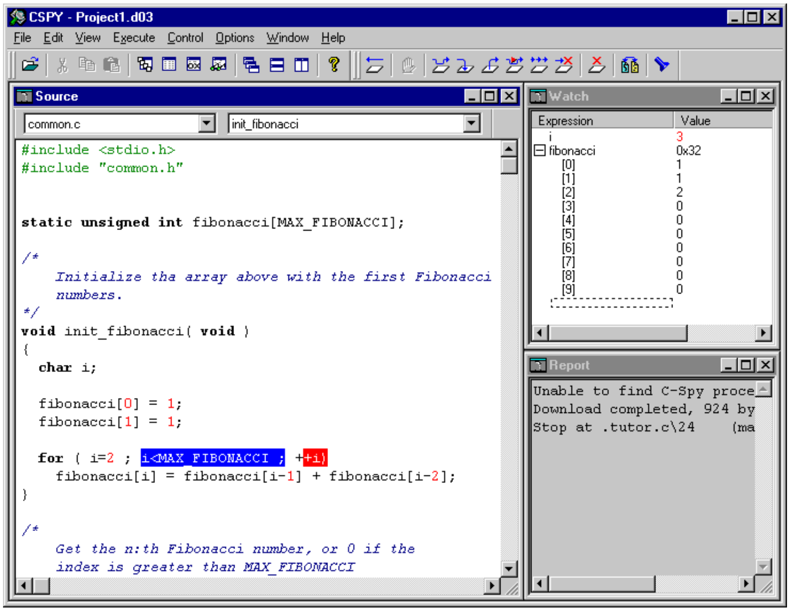
SETTING BREAKPOINTS

You can set breakpoints at C function names or line numbers, or at assembler symbols or addresses. The most convenient way is usually to set breakpoints interactively, simply by positioning the cursor in a statement and then choosing the **Toggle Breakpoint** command. For additional information, see *Toggle breakpoint*, page 230.

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

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

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



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


static unsigned int fibonacci[MAX_FIBONACCI];

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

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

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

/*
    Get the n:th Fibonacci number, or 0 if the
    index is greater than MAX_FIBONACCI
```

Expression	Value
i	3
⊟ fibonacci	0x32
[0]	1
[1]	1
[2]	2
[3]	0
[4]	0
[5]	0
[6]	0
[7]	0
[8]	0
[9]	0

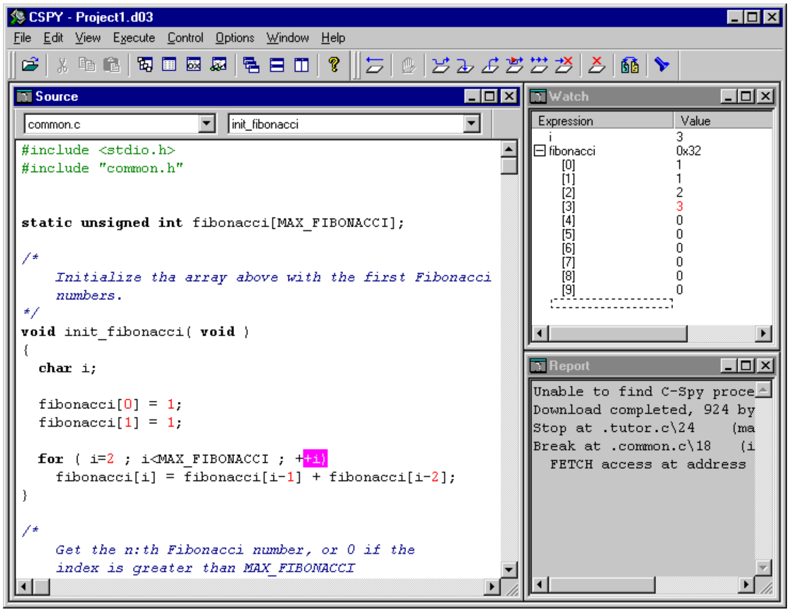
```
Unable to find C-Spy proce
Download completed, 924 by
Stop at .tutor.c\24     (ma
```

Figure 22: Setting breakpoints

EXECUTING UP TO A BREAKPOINT

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

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



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


static unsigned int fibonacci[MAX_FIBONACCI];

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

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

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

/*
    Get the n:th Fibonacci number, or 0 if the
    index is greater than MAX_FIBONACCI
```

Expression	Value
i	3
⊟ fibonacci	0x32
[0]	1
[1]	1
[2]	2
[3]	3
[4]	0
[5]	0
[6]	0
[7]	0
[8]	0
[9]	0

```
Unable to find C-Spy proce
Download completed, 924 by
Stop at .tutor.c\24     (ma
Break at .common.c\18   (i
  FETCH access at address
```

Figure 23: Executing up to a breakpoint

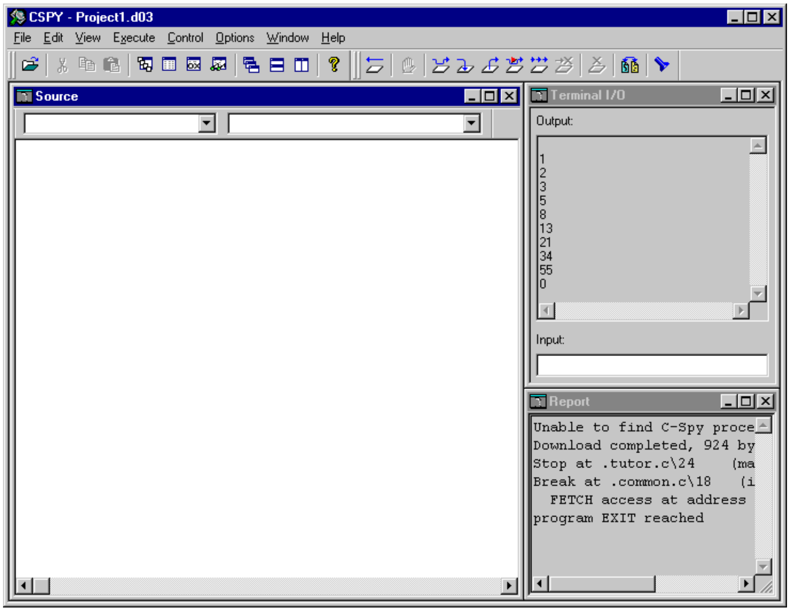
Remove the breakpoint by selecting **Edit breakpoints...** from the **Control** menu. Alternatively, click the right mouse button to display a pop-up menu and select **Edit breakpoints...**. In the **Breakpoints** dialog box, select the breakpoint in the **Breakpoints** list and click **Clear**. Then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

CONTINUING EXECUTION

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

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

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



```
1
2
3
5
8
13
21
34
55
0
```

```
Unable to find C-Spy proce
Download completed, 924 by
Stop at .tutor.c\24     (ma
Break at .common.c\18    (i
  FETCH access at address
program EXIT reached
```

Figure 24: Reaching program EXIT in C-SPY

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

EXITING FROM C-SPY

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

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

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

Compiler tutorials

This chapter introduces you to some of the IAR C Compiler's 8051-specific features:

- Tutorial 2 demonstrates how to utilize 8051 peripherals with the IAR C Compiler features. The #pragma language directive allows us to use the 8051-specific language extensions. The program will be extended to handle polled I/O. Finally, we run the program in C-SPY and create virtual registers.
- In Tutorial 3, we modify the tutorial project by adding an interrupt handler. The system is extended to handle the real-time interrupt using the 8051 IAR C Compiler intrinsics and keywords. Finally, we run the program using the C-SPY interrupt system in conjunction with complex breakpoints and macros.

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

Tutorial 2

This IAR C Compiler tutorial will demonstrate how to simulate the 8051 Universal Asynchronous Receiver/Transmitter (UART) using the IAR C Compiler features.

THE TUTOR2.C SERIAL PROGRAM

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

```
#include <stdio.h>
#include "tutor2.h"

/***********************************
*        Start of code             *
***********************************/
int call_count;

void do_foreground_process(void)
{
  unsigned int fib;

/*
  wait for receive data
*/
  if ( !receive_ok() )
    putchar( '.' );
  else
  {
    fib = get_fibonacci( call_count );
    call_count++;
    put_value( fib );
    clear_interupt();
  }
}

void main(void)
{
/*
   Initialize controll SFR's
*/
  init_cntr();

  init_fibonacci();

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

COMPILING AND LINKING THE TUTOR2.C SERIAL PROGRAM

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

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

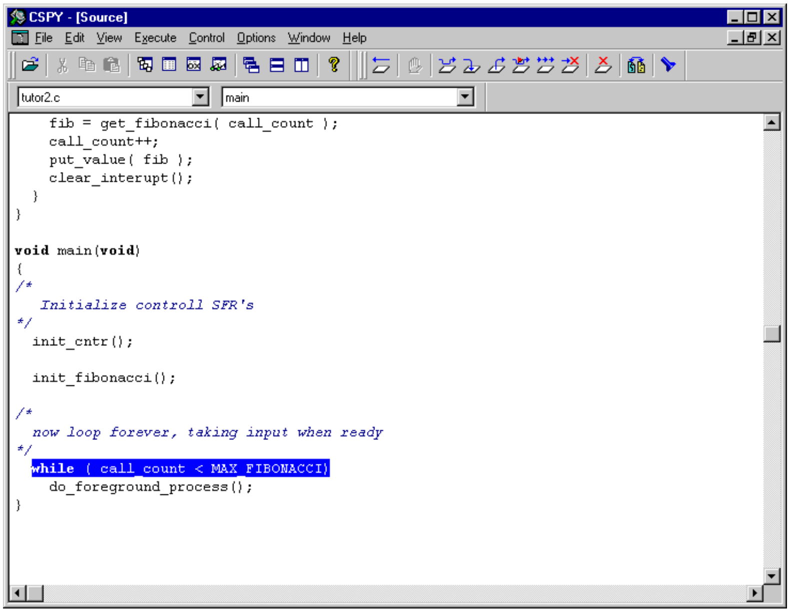
Select **Options...** from the **Project** menu. Select **ICC8051** from the **Categories** box. In the **Code Generation** page, make sure that language extensions are enabled and that debug information will be generated.

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

RUNNING THE TUTOR2.C SERIAL PROGRAM

Start the IAR C-SPY Debugger and run the modified `project1` project. Step until you reach the while (`call_count < MAX_FIBONACCI`) loop, where the program waits for input.

Open the Terminal I/O window by selecting **Terminal I/O** from the **Window** menu. This is where the `tutor2` result will be displayed.



```
    fib = get_fibonacci( call_count );
    call_count++;
    put_value( fib );
    clear_interupt();
  }
}

void main(void)
{
/*
   Initialize controll SFR's
*/
  init_cntr();

  init_fibonacci();

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

Figure 25: Opening the Terminal I/O window

DEFINING VIRTUAL REGISTERS

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

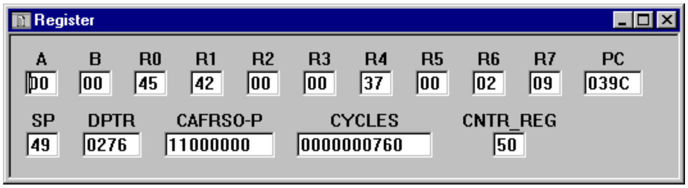
Choose **Settings...** from the **Options** menu. On the **Register Setup** page, click the **New** button to add a new register. Now the **Virtual Register** dialog box will appear.

Enter the following information in the dialog box:

Input field	Input	Description
Name	`CNTR_REG`	Virtual register name
Size	`1`	One byte
Base	`16`	Binary values
Address	`98`	Memory location (in hex)
Segment	`SFR`	Segment name

Table 5: Defining virtual registers

Then click **OK** in the dialog box. Mark the virtual register called CNTR-REG and click OK. Open the Register window from the **Window** menu, or select the **Register Window** button in the toolbar. `USR` should now be included in the list of registers. The current value of each bit in the serial interface register is shown:



A	B	R0	R1	R2	R3	R4	R5	R6	R7	PC
00	00	45	42	00	00	37	00	02	09	039C

SP	DPTR	CAFRSO-P	CYCLES	CNTR_REG
49	0276	11000000	0000000760	50

Figure 26: Tutorial 2 Register window

As you step through the program, you can enter new values into `CNTR_REG` in the Register window. When the first bit (`0x01`) is set, a new Fibonacci number will be printed, and when the bit is cleared, a period (`.`) will be printed instead.

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

Tutorial 3

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

In Embedded Workbench, open the `tutor3.c` file.

THE TUTOR3.C PROGRAM

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

```
#include <stdio.h>
#include "tutor3.h"
```

```
/**********************************
*        Start of code           *
**********************************/
int call_count;

void do_foreground_process(void)
{
  putchar ( '.' );
}

interrupt [TUTOR_INTERRUPT_VECTOR] void tutor_interrupt(void)
{
  unsigned int fib;

  fib = get_fibonacci( call_count );
  call_count++;
  put_value( fib );
  clear_interupt();
}

void main(void)
{
  /* Initialize comms channel */

  init_cntr();

  init_fibonacci();
/*  enable_interrupt(); */

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

The address for the interrupt handler and the actual interrupt vector `TUTOR_INTERRUPT_VECTOR` are defined in the header file `tutor3.h`.

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

```
interrupt [TUTOR_INTERRUPT_VECTOR] void tutor_interrupt(void)
```

The extended keywords are described in the *8051 IAR C Compiler Reference Guide*.

The interrupt handler will fetch the latest Fibonacci value from the `get_fibonacci` function. It will then print the value by using the `put_value` function.

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

THE C-SPY TUTOR3.MAC MACRO FILE

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

Initializing the system

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

A message is displayed in the C-SPY Report window so that we know that this macro has been executed. For additional information, see *Report window*, page 220.

Next, initialize the control registers `SCON` at addresses `0x98` to zero. Clear the enable serial port interrupt bit that is bit 4 at address `0xA8`. Continue to set up two data break points in the `SFR` and connect them to C-SPY macros that are also defined in the file `tutor3.mac`.

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

  //
  // Clear SCON
  //
  __writeMemoryByte(0, 0x98, "SFR");

  //
  // clear the ES bit from IE;
  //
  ie =  __readMemoryByte(0xA8, "SFR");
  ie = ie & 0xEF;
  __writeMemoryByte(ie, 0xA8, "SFR");

  //
  // if SCON or IE is altered, check if we should simulate an
interrupt
  //
  __setBreak("0xA8", "SFR", 1, 1, "", "TRUE", "W",
"_check_SCON_IE()");
  __setBreak("0x98", "SFR", 1, 1, "", "TRUE", "W",
"_check_SCON_IE()");
}
```

Generating interrupts

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

```
_InterruptID = __orderInterrupt("sio ti", (#CYCLES + 2000),
2000, 0, 100, 100);
```

The following parameters are used:

`sio ti`	Specifies which interrupt vector to use. Must be the same name as defined in the device definition file (.ddf files).
`#CYCLES`	Specifies the activation moment for the interrupt. The interrupt is activated when the cycle counter has passed this value. The interrupt activation time is calculated as an offset from the current cycle count `#CYCLES`.
`2000`	Specifies the repeat interval for the interrupt, measured in clock cycles.
`0`	Time variance
`100`	Latency
`100`	Specifies probability. Here it denotes 100%. We want the interrupt to occur at the given frequency. Another percentage could be used for simulating a more randomized interrupt behavior.

During execution, C-SPY will wait until the cycle counter has passed the activation time. Then it will, with 100% certainty, generate an interrupt approximately every 2000 cycles.

Using breakpoints to simulate incoming values

Check the `SCON` and IE registers to see if any new value is set. This is done by setting a breakpoint at the address of the `SCON` and IE `SFR` registers and connecting a user-defined macro to them. Here we use the `__setBreak` system macro.

The following parameters are used:

`0xA8`	Receive buffer address.
SFR	The memory segment where this address is found. The segments `IDATA`, `CODE`, SFR, and `XDATA` are valid for the 8051 product.
`1`	Length.
`1`	Count.
""	Denotes unconditional breakpoint.

`"TRUE"`	Condition type.
`"W"`	Execute the macro if a write occurs at the specific address.
`"check_SCON_IE()"`	The macro connected to the breakpoint.

During execution, when C-SPY detects a write to any of the two breakpoints set at addresses IE (`0xA8`) or `SCON` (`0x98`), it will start executing the macro `check_SCON_IE()`. Since this macro ends with a `resume` statement, C-SPY will then resume the simulation.

Resetting the system

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

```
execUserReset()
{
  var ie;
  message "execUserReset() called, cancelling all
interrupts\n";

  //
  // Clear SCON
  //
  __writeMemoryByte(0, 0x98, "SFR");

  //
  // clear the ES bit from IE;
  //
  ie =  __readMemoryByte(0xA8, "SFR");
  ie = ie & 0xEF;
  __writeMemoryByte(ie, 0xA8, "SFR");

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

Exiting the system

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

```
execUserExit()
{
  message "execUserExit called, cancelling all interrupts\n";

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

The interrupts are cleared and the input file is closed.

COMPILING AND LINKING THE TUTOR3.C PROGRAM

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

Select the **Debug** folder in the project window and then choose **Options...** from the **Project** menu. In the **General** category, select **8XC51** and the **Small** memory model.

In the **ICC8051** category, make sure that the following options are enabled:

Page	Option
Language	Enable language extensions
Debug	Generate debug information

Table 6: Tutorial 3 compiler options

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

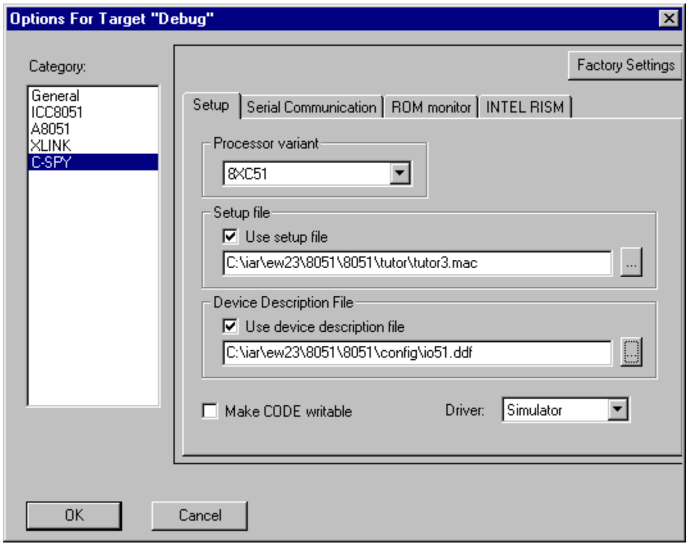
RUNNING THE TUTOR3.C INTERRUPT PROGRAM

To run the `tutor3.c` program, first specify the macro and device description file to be used. The device description file defines `SFR`'s and interrupt vectors. In this tutorial, use the `io51.ddf` file to set up the `SFR` window and the standard 8051 interrupt system. Read more about the device description file on page 132 and in the *Device description file* chapter in *Part 4: The C-SPY simulator* in this guide.

To set the options, select **Options** from the **Project** menu.

Select **C-SPY** in the **General** category. Select the **Setup** page and check the **Use Setup file** option. Browse for the `tutor3.mac` file in your `tutor` folder.

Check the **Use device description file** option and browse for the `io51.ddf` file in the `config` folder. Click **OK** in the **Options** dialog box.



Processor variant: 8XC51

Setup file

☑ Use setup file

C:\iar\ew23\8051\8051\tutor\tutor3.mac

Device Description File

☑ Use device description file

C:\iar\ew23\8051\8051\config\io51.ddf

☐ Make CODE writable

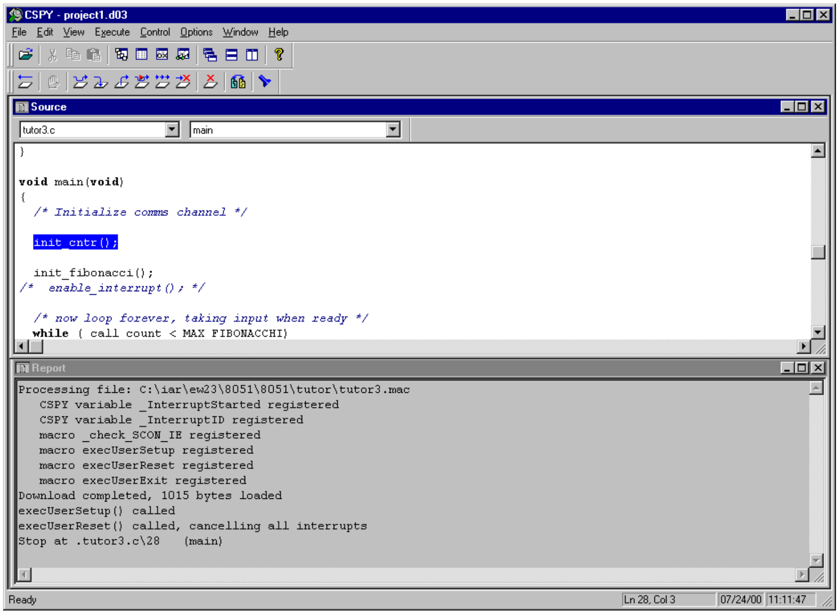
Driver: Simulator

Figure 27: Specifying the setup macro and DDF file

If you use the IAR C-SPY Debugger without using the IAR Embedded Workbench, the macro file can be specified via the command line option `-f` and the device description file with the command line option `-p`; for additional information, see the chapter *C-SPY command line options*, page 247.

Note: Macro files can also be loaded via the **Options** menu in the IAR C-SPY Debugger, see *Load Macro…*, page 243. Due to its contents, the `tutor3.mac` file cannot, however, be used in this manner because the `execUserSetup` macro will not be activated until you load the `project1.d03` file.

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



```
}

void main(void)
{
  /* Initialize comms channel */

  init_cntr();

  init_fibonacci();
/*  enable_interrupt(); */

  /* now loop forever, taking input when ready */
  while ( call count < MAX FIBONACCHI)
```

```
Processing file: C:\iar\ew23\8051\8051\tutor\tutor3.mac
   CSPY variable _InterruptStarted registered
   CSPY variable _InterruptID registered
   macro _check_SCON_IE registered
   macro execUserSetup registered
   macro execUserReset registered
   macro execUserExit registered
Download completed, 1015 bytes loaded
execUserSetup() called
execUserReset() called, cancelling all interrupts
Stop at .tutor3.c\28   (main)
```

Figure 28: C-SPY Source and Report windows

The Report window displays the registered macros.

If warning or error messages should also appear in the Report window, make sure that the breakpoint has been set and that the interrupt has been registered.

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

Now you have two breakpoints in `SFR` memory. To inspect the details of the interrupt, open the **Interrupt** dialog box by selecting **Interrupt...** from the **Control** menu. At this point there is no interrupt set.

In the Source window, make sure that `tutor3.c` is selected in the **Source file** box. Then select the `tutor_interrupt` function in the **Function** box.

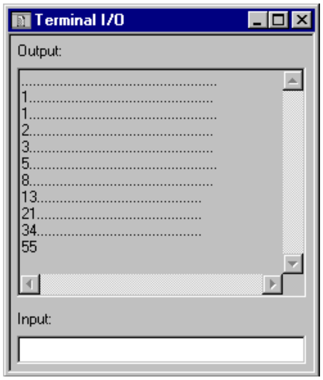
Place the cursor on the `get_fibonacci()` statement in the `tutor_interrupt` function. Set a breakpoint by selecting **Toggle Breakpoint** from the **Control** menu, or clicking the **Toggle Breakpoint** button in the toolbar. Alternatively, use the pop-up menu.

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

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

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

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



```
................................................
1..............................................
1...............................................
2..............................................
3..............................................
5...............................................
8..............................................
13.........................................
21.........................................
34.........................................
55
```

Figure 29: Terminal I/O window

Assembler tutorials

These tutorials illustrate how you can use the IAR Embedded Workbench™ to develop a series of simple machine-code programs for the 8051 microcontroller:

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

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

Tutorial 4

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

CREATING A NEW PROJECT

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

Set up the target options in the **General** category to suit the processor and memory model. In this tutorial we use the default settings. Make sure that the **Processor variant** is set to **8XC51** and that the **Memory model** is set to **Tiny**.

The procedure is described in *Creating a new project*, page 27.

THE FIRST.S03 PROGRAM

The first assembler tutorial program is a simple count loop which counts up the registers `R0` and `R1` in binary-coded decimal. A copy of the program `first.s03` is provided with the product.

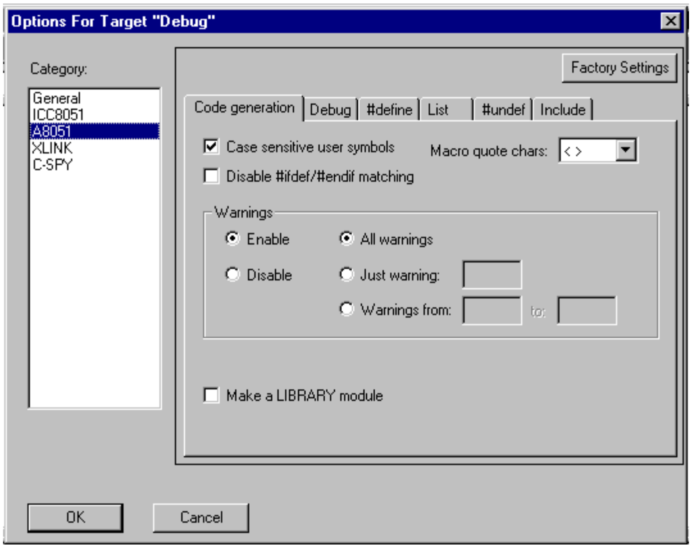
```
        NAME    first

        ORG     0
        LJMP    main

        RSEG    CODE
main    MOV     R0,#0
        MOV     R1,#0
loop    INC     R1
        MOV     A,R1
        DA      A
        MOV     R1,A
        MOV     A,R0
        ADDC    A,#0
        DA      A
        MOV     R0,A
        JNC     loop
        RET

        END
```

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

Add the program to the `Project2` project. Choose **Files…** from the **Project** menu to display the **Project Files** dialog box. Locate the file `first.s03` in the **Tutor** folder and click **Add** to add it to the **Common Sources** group. You now have a source file which is ready to assemble.

ASSEMBLING THE PROGRAM

Now you should set up the assembler options for the project. Select the **Debug** folder icon in the Project window, choose **Options…** from the **Project** menu, and select **A8051** in the **Category** list to display the assembler options pages.



☑ Case sensitive user symbols

Macro quote chars: < >

☐ Disable #ifdef/#endif matching

● Enable ○ Disable

● All warnings

○ Just warning:

○ Warnings from: to:

☐ Make a LIBRARY module

Figure 30: Tutorial 4 Assembler code generation options

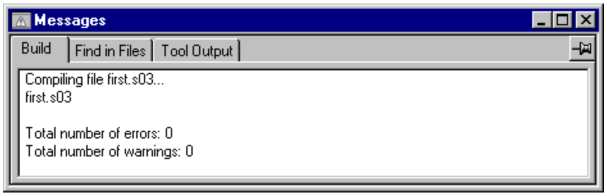
Make sure that the following options are selected on the appropriate pages of the **Options** dialog box:

Page	Option
Debug	Generate debug information File references
List	List file, Include header, Include listing, Macro expansion

Table 7: Tutorial 4 assembler options

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

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



```
Compiling file first.s03...
first.s03

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

Figure 31: Messages window after assembling the file

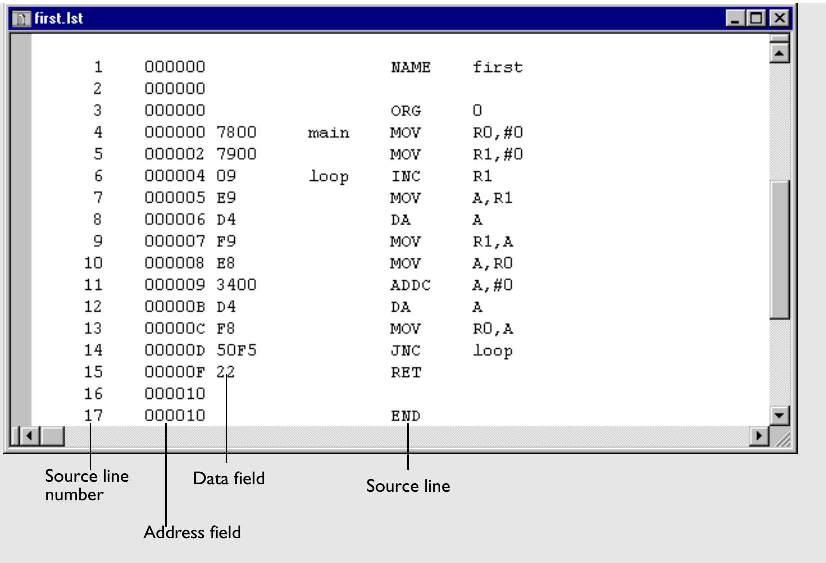
The listing is created in a file `first.lst` in the folder specified in the **General** options page; by default this is `Debug\list`. Open the list file by choosing **Open…** from the **File** menu, and selecting `first.lst` from the appropriate folder.

VIEWING THE FIRST.LST LIST FILE

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

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

The format of the listing is as follows:



```
 1    000000                   NAME    first
 2    000000
 3    000000                   ORG     0
 4    000000 7800      main    MOV     R0,#0
 5    000002 7900              MOV     R1,#0
 6    000004 09        loop    INC     R1
 7    000005 E9                MOV     A,R1
 8    000006 D4                DA      A
 9    000007 F9                MOV     R1,A
10    000008 E8                MOV     A,R0
11    000009 3400              ADDC    A,#0
12    00000B D4                DA      A
13    00000C F8                MOV     R0,A
14    00000D 50F5              JNC     loop
15    00000F 22                RET
16    000010
17    000010                   END
```

Figure 32: Assembler list file

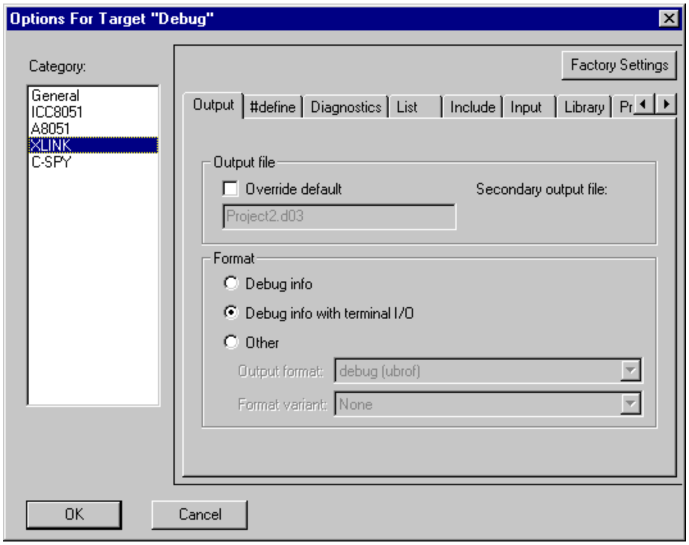
If you make any errors when writing a program, these will be displayed on the screen during the assembly and will be listed in the list file. If this happens, return to the editor by double-clicking on the error message. Check carefully through the source code to locate and correct all the mistakes,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

Assuming that the source assembled successfully, the file `first.r03`, will also be created, containing the linkable object code.

LINKING THE PROGRAM

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

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



☐ Override default

Project2.d03

Secondary output file:

○ Debug info

● Debug info with terminal I/O

○ Other

Output format: debug (ubrof)

Format variant: None

Figure 33: XLINK output options

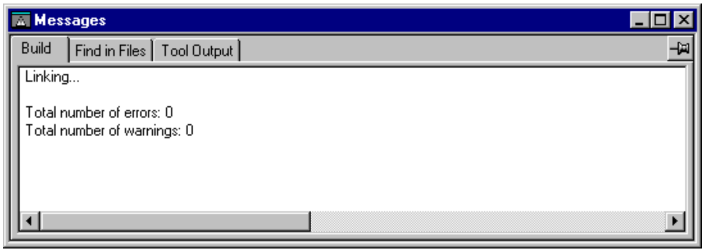
Specify the following XLINK options:

Page	**Option**
Output	Debug info with terminal I/O
Include	Override default XCL file name select the asm.xcl in the config folder.
Library	Override the default library and remove the library name so that it is left blank. The C library will not be used for the assembler project.

Table 8: Tutorial 4 XLINK options

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

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



```
Linking...

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

Figure 34: Messages window after linking the file

The code will be placed in a file `project2.d03`.

RUNNING THE PROGRAM

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

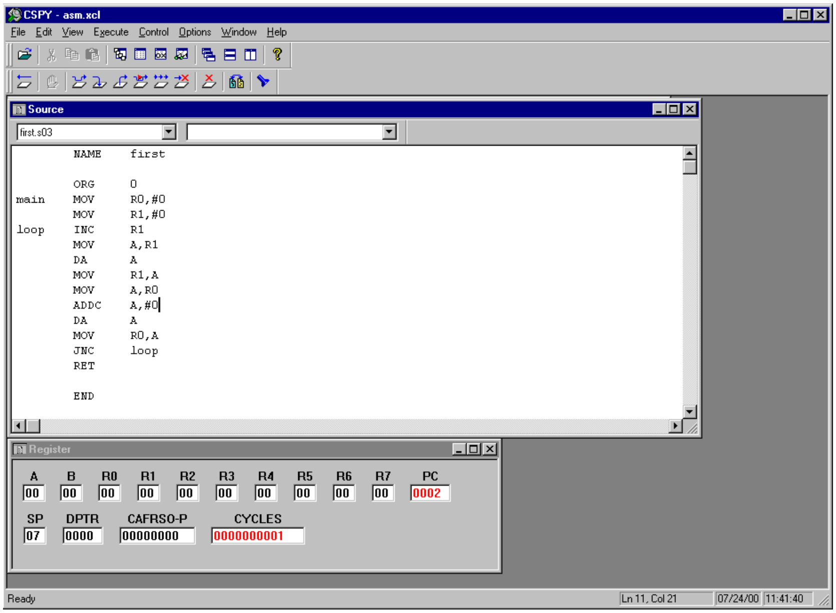
A warning message will be displayed in the Report window.

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

In this example, set a breakpoint on the `ADDC A,#0` instruction within the loop.

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

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



```
        NAME    first

        ORG     0
main    MOV     R0,#0
        MOV     R1,#0
loop    INC     R1
        MOV     A,R1
        DA      A
        MOV     R1,A
        MOV     A,R0
        ADDC    A,#0
        DA      A
        MOV     R0,A
        JNC     loop
        RET

        END
```

A	B	R0	R1	R2	R3	R4	R5	R6	R7	PC
00	00	00	00	00	00	00	00	00	00	0002

SP	DPTR	CAFRSO-P	CYCLES
07	0000	00000000	0000000001

Ln 11, Col 21 07/24/00 11:41:40

Figure 35: Registers counting when executing program

Tutorial 5

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

USING LIBRARIES

If you are working on a large project you will soon accumulate a collection of useful routines that are used by several of your programs. To avoid having to assemble a routine each time the routine is needed, you can store such routines as object files, i.e., assembled but not linked. A collection of routines in a single object file is referred to as a *library*. It is recommended that you use library files to create collections of related routines, such as a device driver.

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

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

THE MAIN.S03 PROGRAM

The following listing shows the `main.s03` program. A copy of the program is provided with the product.

```
       NAME      main

       EXTERN    rightshift

       RSEG      CODE
main   MOV       A,start
       MOV       B,#4
       LCALL     rightshift
       RET

       RSEG      DATA
start  DS        11

       END       main
```

This simply uses a routine called `rightshift` to shift the contents of `start` to the right. The first byte of the array `start` is moved to register A and the `rightshift` routine is called to shift it to the right by four places as specified by the contents of register `B`.

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

THE LIBRARY ROUTINES

The following two library routines will form a separately assembled library. It consists of the `rightshift` routine called by `main`, and a corresponding `leftshift` routine, both of which operate on the contents of the register A by repeatedly shifting it to the right or left. The number of shifts performed is controlled by decrementing register B to zero. The file containing these library routines is called `shifts.s03`, and a copy is provided with the product.

```
       MODULE    rightshift
       PUBLIC    rightshift
       RSEG      CODE
rightshift:
       RR        A
       DJNZ      B,rightshift
       RET
       ENDMOD

       MODULE    leftshift
       PUBLIC    leftshift
       RSEG      CODE
leftshift:
       RL        A
       DJNZ      B,leftshift
       RET

       END
```

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

The `PUBLIC` directive makes the `rightshift` and `leftshift` entry addresses public to other modules.

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

CREATING A NEW PROJECT

Create a new project called `Project3`. Add the files `main.s03` and `shifts.s03` to the new project.

Then set up the target options to suit the project. Make sure that the **Processor variant** is set to **8XC51** and that the **Memory model** is set to **Tiny**.

The procedure is described in *Creating a new project*, page 27.

ASSEMBLING AND LINKING THE SOURCE FILES

To assemble and link the `main.s03` and `shifts.s03` source files, you must exclude the `CSTARTUP` initialization module in the default run-time library.

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

Page	Option
Include	Override the default XCL file name and browse to find the `asm.xcl` file in the `config` folder.
Library	Override the default library and remove the library name so that it is left blank. The C library will not be used for the assembler project.

Table 9: Tutorial 5 XLINK options

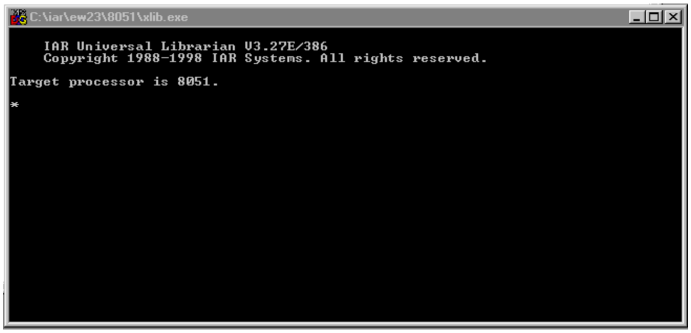
To assemble and link the `main.s03` and the `shifts.s03` files , select **Make** from the **Project** menu. Alternatively, click the **Make** button in the toolbar.

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

USING THE IAR XLINK LIBRARIAN

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

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



```
    IAR Universal Librarian V3.27E/386
    Copyright 1988-1998 IAR Systems. All rights reserved.

Target processor is 8051.

*
```

Figure 36: XLIB window

You can now enter XLIB commands at the `*` prompt.

Giving XLIB options

Extract the modules you want from `shifts.r03` into a library called `math.r03`. To do this enter the command:

`FETCH-MODULES`

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

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

Table 10: XLIB FETCH-MODULES parameters

This creates the file `math.r03` which contains the code for the `rightshift` and `leftshift` routines.

You can confirm this by typing:

```
LIST-MODULES
```

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

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

Table 11: XLIB LIST-MODULES parameters

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

Finally, leave the librarian by typing:

```
EXIT
```

or

```
QUIT
```

Then press Enter.

Try out the new `math.r03` library:

1. Remove the `shift.s03` assembler file from the project by choosing **Files...** from the **Project** menu.
2. Select the `shift.s03` file and click the **Remove** button.
3. Click **Done** to exit from the dialog box.

To include the new math library:

1. Choose XLINK from the **Category** list.
2. Select the **Library** page.
3. Type `math.r03` in the **Override default library** field.
4. This library is not included in the standard library directory, but it is found in another directory. On the **Include** page, specify it in the edit box by typing `’$TOOLKIT_DIR$\PROJECTS\DEBUG\OBJ\'` just below the `'$TOOLKIT_DIR$\LIB\'` line.
5. Click **OK** and build the project.

Advanced tutorials

These tutorials explore some of the more advanced features of the IAR C-SPY Debugger, which are very useful when you work on larger projects.

- In Tutorial 6, we define complex breakpoints, profile the application, and display code coverage.
- Tutorial 7 shows how to debug in disassembly mode.
- Tutorial 8 demonstrates how to create a project containing both C and assembly language source files.

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

Tutorial 6

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

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

CREATING PROJECT4

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

Category	Page	Option
General	Target	Processor variant: 8XC51 (default) Memory model: Tiny (default)
ICC8051	Code Generation	Size optimization: Low
	Debug	Generate debug information (default)
	List	List file
XLINK	Output	Debug info with terminal I/O (default)

Table 12: Tutorial 6 options

Click **OK** to set the options.

Select **Make** from the **Project** menu, or click the **Make** button in the toolbar to compile and link the files. This creates the `project4.d03` file.

Start C-SPY to run the `project4.d03` program.

DEFINING COMPLEX BREAKPOINTS

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

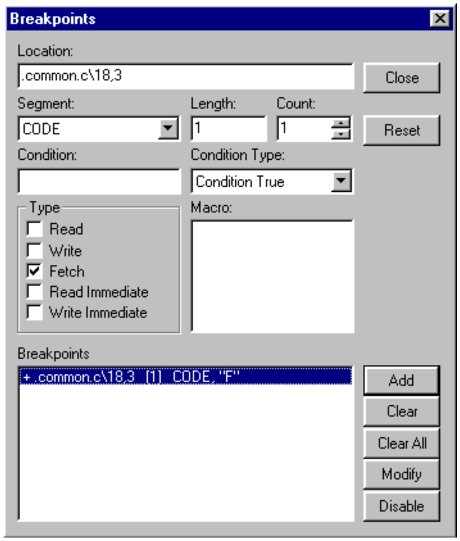
The file `project4.d03` should now be open in C-SPY. The current position should be the call to the `call_count=0` statement.

Execute a step and then click the **Step into** button to move to the `init_fibonacci` function. Set a breakpoint at the statement `++i`.

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

Choose **Edit Breakpoints…** from the **Control** menu to display the **Breakpoints** dialog box.

Select the breakpoint in the **Breakpoints** list to display information about the breakpoint you have defined:



Location: .common.c\18,3

Segment: CODE

Length: 1

Count: 1

Condition:

Condition Type: Condition True

Type

☐ Read
☐ Write
☑ Fetch
☐ Read Immediate
☐ Write Immediate

Macro:

Breakpoints

+ .common.c\18,3 [1] CODE, "F"

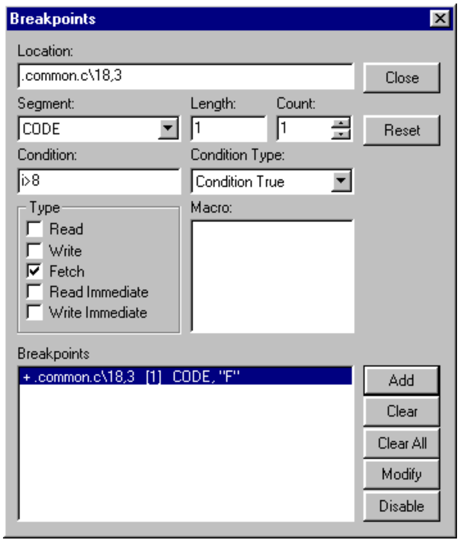
Figure 37: Displaying breakpoint information

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

Add a condition to the breakpoint using the following procedure:

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

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



Breakpoints

Location: .common.c\18,3

Segment: CODE

Length: 1

Count: 1

Condition: i>8

Condition Type: Condition True

Type
- ☐ Read
- ☐ Write
- ☑ Fetch
- ☐ Read Immediate
- ☐ Write Immediate

Macro:

Breakpoints

+ .common.c\18,3 [1] CODE, "F"

Figure 38: Modifying breakpoints

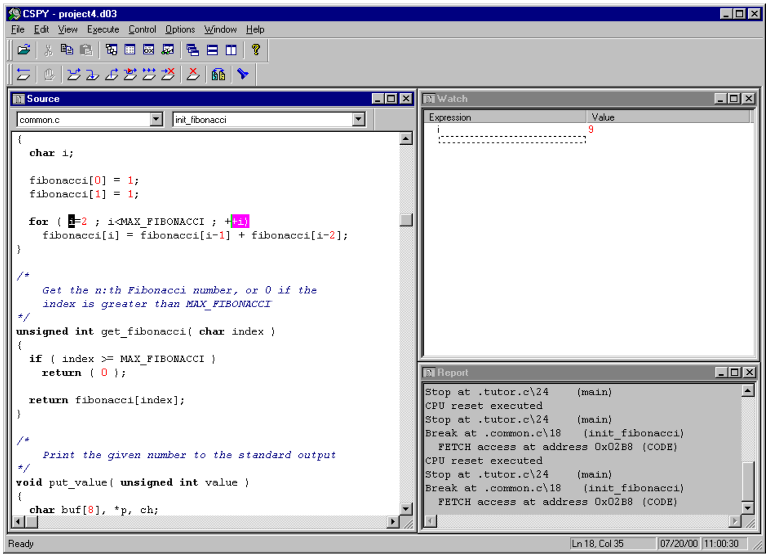
Finally, click the **Close** button to close the **Breakpoints** dialog box.

Open the Watch window and add the variable `i`. The procedure is described in *Watching variables*, page 42.

Position the Source, Watch, and Report windows conveniently.

EXECUTING UNTIL A CONDITION IS TRUE

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



```
{
  char i;

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

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

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

  return fibonacci[index];
}

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

Expression	Value
i	9

```
Stop at .tutor.c\24     (main)
CPU reset executed
Stop at .tutor.c\24     (main)
Break at .common.c\18    (init_fibonacci)
  FETCH access at address 0x02B8 (CODE)
CPU reset executed
Stop at .tutor.c\24     (main)
Break at .common.c\18    (init_fibonacci)
  FETCH access at address 0x02B8 (CODE)
```

Figure 39: Executing in C-SPY until a breakpoint condition is true

EXECUTING UP TO THE CURSOR

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

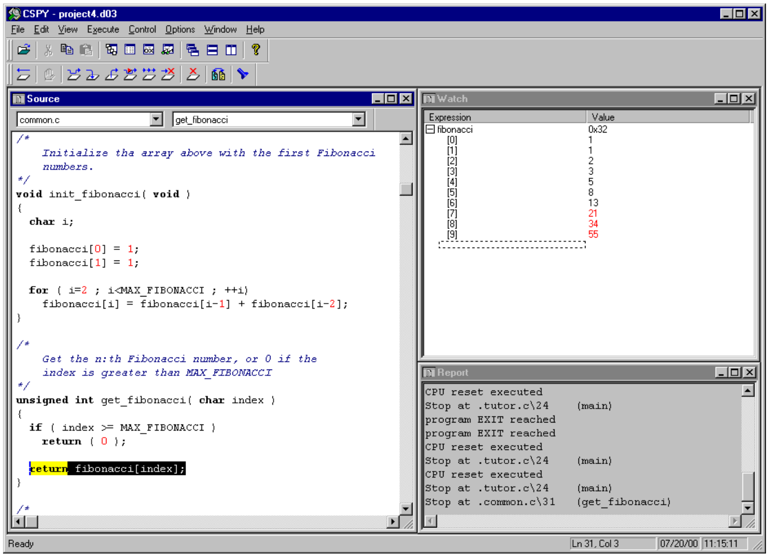
First remove the existing breakpoint. Use the **Edit Breakpoints...** command from the **Control** menu or from the pop-up menu to open the **Breakpoints** dialog box. Select the breakpoint and click on the **Clear** button.

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

Position the cursor in the Source window in the statement:

`return fibonacci[index];`

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



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

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

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


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

  return fibonacci[index];
}

/*
```

Expression	Value
fibonacci	0x32
[0]	1
[1]	1
[2]	2
[3]	3
[4]	5
[5]	8
[6]	13
[7]	21
[8]	34
[9]	55

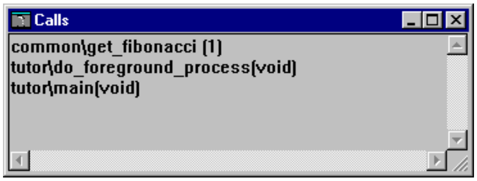
```
CPU reset executed
Stop at .tutor.c\24    (main)
program EXIT reached
program EXIT reached
CPU reset executed
Stop at .tutor.c\24    (main)
CPU reset executed
Stop at .tutor.c\24    (main)
Stop at .common.c\31   (get_fibonacci)
```

Figure 40: Executing in C-SPY up to the cursor

DISPLAYING FUNCTION CALLS

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

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



```
common\get_fibonacci (1)
tutor\do_foreground_process(void)
tutor\main(void)
```

Figure 41: Calls window

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

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

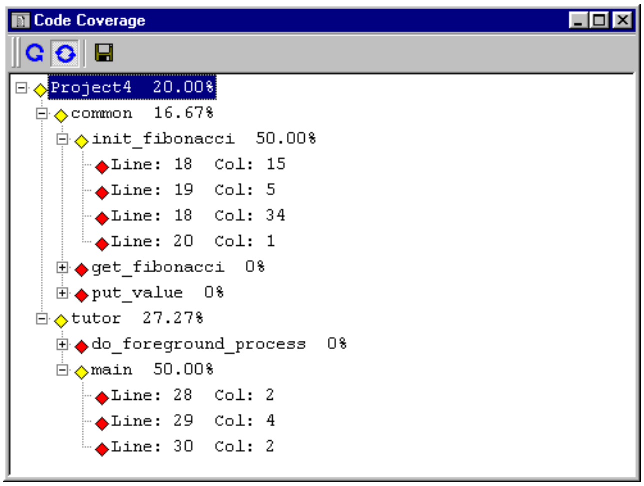
DISPLAYING CODE COVERAGE INFORMATION

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

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

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

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



```
Project4   20.00%
  common   16.67%
    init_fibonacci   50.00%
      Line: 18   Col: 15
      Line: 19   Col: 5
      Line: 18   Col: 34
      Line: 20   Col: 1
    get_fibonacci   0%
    put_value   0%
  tutor   27.27%
    do_foreground_process   0%
    main   50.00%
      Line: 28   Col: 2
      Line: 29   Col: 4
      Line: 30   Col: 2
```

Figure 42: Code coverage window

For additional information about the layout of the Code Coverage window, see *Code coverage window*, page 221.

PROFILING THE APPLICATION

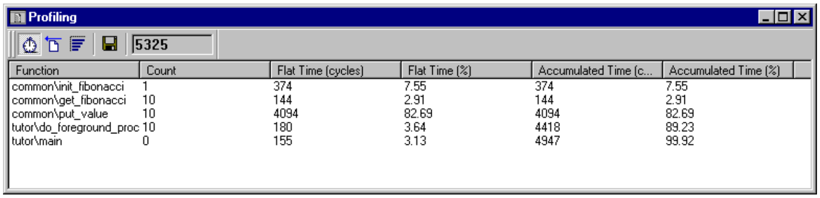
The profiling tool provides you with timing information on your application.

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

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

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

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



5325

Function	Count	Flat Time (cycles)	Flat Time (%)	Accumulated Time (c...	Accumulated Time (%)
common\init_fibonacci	1	374	7.55	374	7.55
common\get_fibonacci	10	144	2.91	144	2.91
common\put_value	10	4094	82.69	4094	82.69
tutor\do_foreground_proc	10	180	3.64	4418	89.23
tutor\main	0	155	3.13	4947	99.92

Figure 43: Tutorial 6 Profiling window

The Profiling window contains the following information:

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

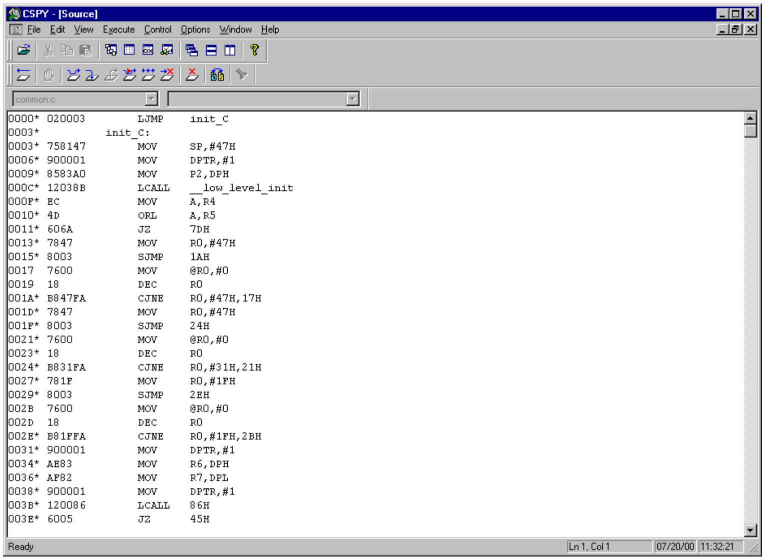
From the **Profiling** toolbar it is possible to display the profiling information graphically, to save the information to a file, or to start a new measurement. For additional information, see *Profiling window*, page 222.

Tutorial 7

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

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

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



```
0000* 020003         LJMP    init_C
0003*           init_C:
0003* 758147         MOV     SP,#47H
0006* 900001         MOV     DPTR,#1
0009* 8583A0         MOV     P2,DPH
000C* 12038B         LCALL   __low_level_init
000F* EC             MOV     A,R4
0010* 4D             ORL     A,R5
0011* 606A           JZ      7DH
0013* 7847           MOV     R0,#47H
0015* 8003           SJMP    1AH
0017  7600           MOV     @R0,#0
0019  18             DEC     R0
001A* B847FA         CJNE    R0,#47H,17H
001D* 7847           MOV     R0,#47H
001F* 8003           SJMP    24H
0021* 7600           MOV     @R0,#0
0023* 18             DEC     R0
0024* B831FA         CJNE    R0,#31H,21H
0027* 781F           MOV     R0,#1FH
0029* 8003           SJMP    2EH
002B  7600           MOV     @R0,#0
002D  18             DEC     R0
002E* B81FFA         CJNE    R0,#1FH,2BH
0031* 900001         MOV     DPTR,#1
0034* AE83           MOV     R6,DPH
0036* AF82           MOV     R7,DPL
0038* 900001         MOV     DPTR,#1
003B* 120086         LCALL   86H
003E* 6005           JZ      45H
```

Figure 44: Debugging in disassembly mode

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

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

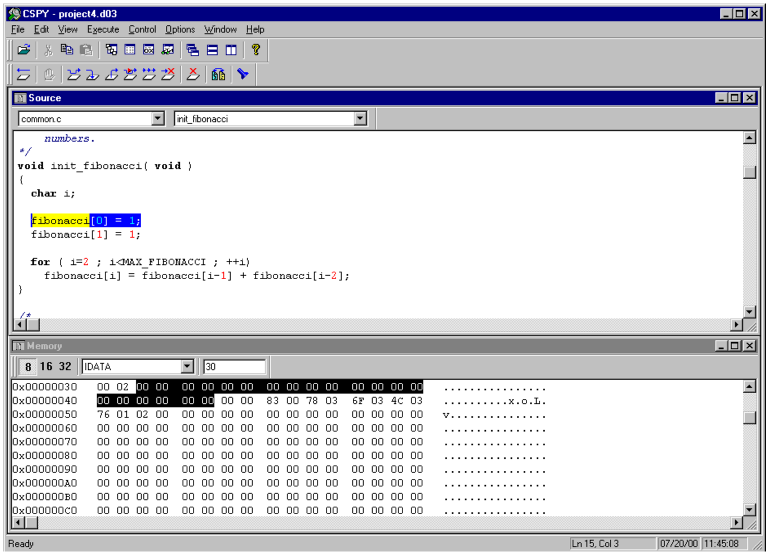
MONITORING MEMORY

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

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

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

Reset C-SPY and step into the `init_fibonacci` function. Select `fibonacci` from the file `common.c`. Then drag it from the Source window and drop it into the Memory window. The Memory window will show the contents of memory corresponding to `fibonacci`:

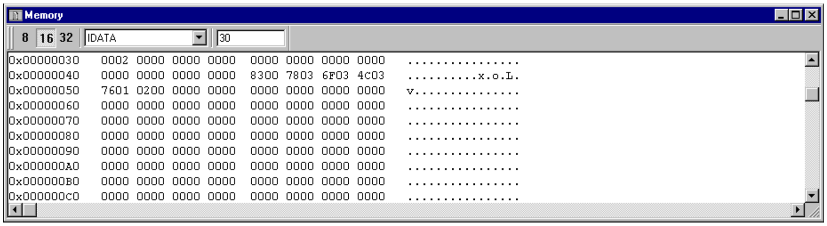


```
    numbers.
*/
void init_fibonacci( void )
{
  char i;

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

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

/*
```

```
0x00000030   00 02 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
0x00000040   00 00 00 00  00 00 00 00  83 00 78 03  6F 03 4C 03   ..........x.o.L.
0x00000050   76 01 02 00  00 00 00 00  00 00 00 00  00 00 00 00   v...............
0x00000060   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
0x00000070   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
0x00000080   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
0x00000090   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
0x000000A0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
0x000000B0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
0x000000C0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
```

Figure 45: Monitoring memory

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



```
0x00000030    0002 0000 0000 0000  0000 0000 0000 0000   ................
0x00000040    0000 0000 0000 0000  8300 7803 6F03 4C03   ..........x.o.L.
0x00000050    7601 0200 0000 0000  0000 0000 0000 0000   v...............
0x00000060    0000 0000 0000 0000  0000 0000 0000 0000   ................
0x00000070    0000 0000 0000 0000  0000 0000 0000 0000   ................
0x00000080    0000 0000 0000 0000  0000 0000 0000 0000   ................
0x00000090    0000 0000 0000 0000  0000 0000 0000 0000   ................
0x000000A0    0000 0000 0000 0000  0000 0000 0000 0000   ................
0x000000B0    0000 0000 0000 0000  0000 0000 0000 0000   ................
0x000000C0    0000 0000 0000 0000  0000 0000 0000 0000   ................
```

Figure 46: Displaying memory contents as 16-bit words

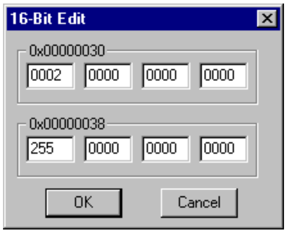
Step around a few times in the for loop and watch the `fibonacci` array in the Memory Window become initialized.

CHANGING MEMORY

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

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

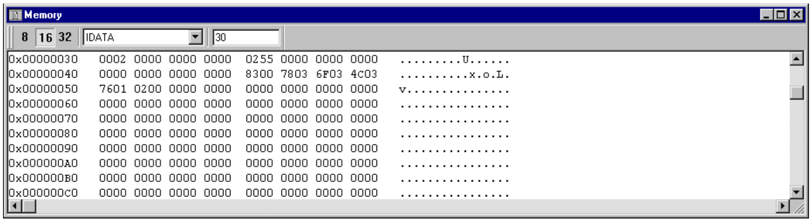
For example, if you want to write the number `0x255` in the first position in number in the `fibonacci` array, double-click that position in the Memory window and type `255` in the **16-Bit Edit** dialog box:



```
0x00000030
0002  0000  0000  0000
0x00000038
255   0000  0000  0000
```

Figure 47: Editing memory contents

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



```
0x00000030   0002 0000 0000 0000  0255 0000 0000 0000   .........U......
0x00000040   0000 0000 0000 0000  8300 7803 6F03 4C03   ..........x.o.L.
0x00000050   7601 0200 0000 0000  0000 0000 0000 0000   v...............
0x00000060   0000 0000 0000 0000  0000 0000 0000 0000   ................
0x00000070   0000 0000 0000 0000  0000 0000 0000 0000   ................
0x00000080   0000 0000 0000 0000  0000 0000 0000 0000   ................
0x00000090   0000 0000 0000 0000  0000 0000 0000 0000   ................
0x000000A0   0000 0000 0000 0000  0000 0000 0000 0000   ................
0x000000B0   0000 0000 0000 0000  0000 0000 0000 0000   ................
0x000000C0   0000 0000 0000 0000  0000 0000 0000 0000   ................
```

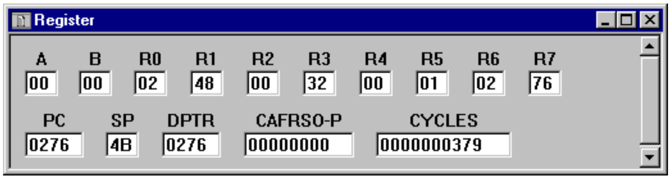
Figure 48: Displaying edited memory contents

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

MONITORING REGISTERS

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

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



A	B	R0	R1	R2	R3	R4	R5	R6	R7
00	00	02	48	00	32	00	01	02	76

PC	SP	DPTR	CAFRSO-P	CYCLES
0276	4B	0276	00000000	0000000379

Figure 49: Tutorial 7 Register window

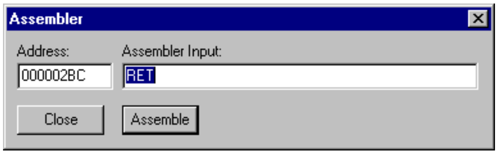
Select **Step** from the **Execute** menu, or click the **Step** button in the toolbar, to execute the next instructions, and watch how the values change in the Register window.

Then close the Register window.

CHANGING ASSEMBLER VALUES

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

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



Address: 000002BC

Assembler Input: RET

Figure 50: Assembler dialog box

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

Tutorial 8

In large projects it may be convenient to use source files written in both C and assembly language. In this tutorial we will demonstrate how they can be combined by substituting the file `common.c` with the assembler file `common.s03` and compiling the project.

CREATING A COMBINED COMPILER AND ASSEMBLER PROJECT

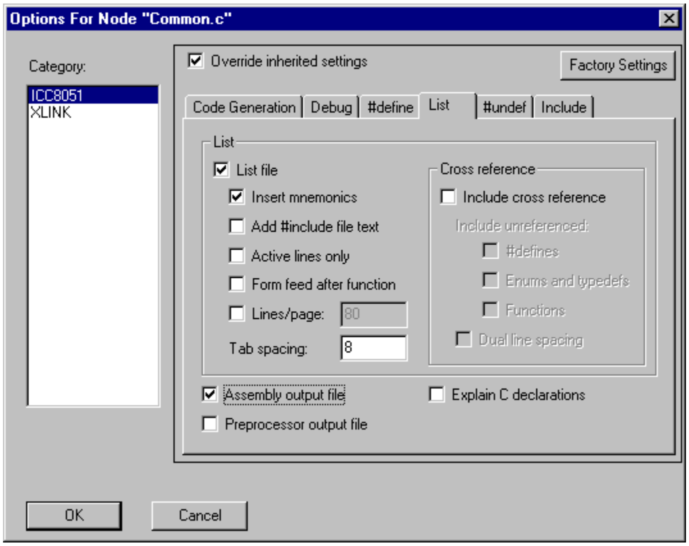
Return to or open `Project4` in the IAR Embedded Workbench. The project should contain the files `tutor.c` and `common.c`.

Now you should create the assembler file `common.s03`. In the Project window, select the file `common.c`. Then select **Options...** from the **Project** menu. You will notice that only the **ICC8051** and **XLINK** categories are available.

In the **ICC8051** category, select **Override inherited settings** and set the following options:

Page	Option
List	list file Insert mnemonics Assembler output file

Table 13: Tutorial 8 compiler options



Options For Node "Common.c"

Category: ICC8051, XLINK

☑ Override inherited settings

List

☑ List file

☑ Insert mnemonics

☐ Add #include file text

☐ Active lines only

☐ Form feed after function

☐ Lines/page: 80

Tab spacing: 8

Cross reference

☐ Include cross reference

Include unreferenced:

☐ #defines

☐ Enums and typedefs

☐ Functions

☐ Dual line spacing

☑ Assembly output file

☐ Explain C declarations

☐ Preprocessor output file

Figure 51: Tutorial 8 Compiler list file options

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

Compile each of the files. To see how the C code is represented in assembly language, open the file `common.s03` that was created from the file `common.c`. It is located in the `debug\list` directory.

Now modify `Project4` by removing the file `common.c` and adding the file `common.s03` instead. Then select **Make** from the **Project** menu to relink `Project4`.

Start C-SPY to run the `project4.d03` program and see that it behaves like in the previous tutorials.

ROM-monitor tutorial

This chapter describes how to specify the settings needed in the IAR Embedded Workbench™ to run the ROM-monitor version of the IAR C-SPY® Debugger.

Tutorial 9

We recommend that you create a specific directory where you can store all your project files, for example `\8051\projects`.

GETTING STARTED

Create a new project, `Project5`, and add the source files `demo.c` and `demo_two.c` to it. The procedure is described in detail in *Tutorial 1*, page 27.

The demo.c program

The `demo.c` program is a simple program that uses only standard C facilities. It uses iteration to calculate 2^{13}, and then prints out the fourth character in a char array. A copy of the program is provided with the product.

```
#include "stdio.h"
#include "defns.h"
void demo_two(int i);

int d,w;

int main(int i)
  {
    for (i = 0, d = 1; i < TWO_POWER; i++)
      d *= 2;
    printf("2 to the power of %d is %d\n",
                 TWO_POWER, d);
    demo_two(3);
  }
```

The demo_two.c program

The `demo_two.c` program, which is also provided with the product, contains the definition of the `demo_two` routine:

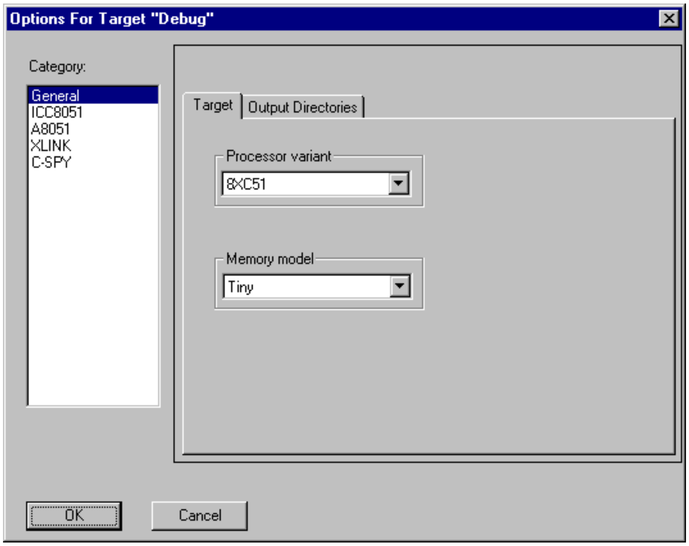
```
#include "stdio.h"

char  array[10] = "abcd";
```

```
void demo_two(int i)
  {
    char *cp;

    cp = &array[i];
    printf ("%c\n", *cp);
  }
```

SETTING TARGET AND COMPILER OPTIONS

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

In this tutorial we use the default **Target** settings—the 8XC51 processor variant and the tiny memory model:



Processor variant: 8XC51

Memory model: Tiny

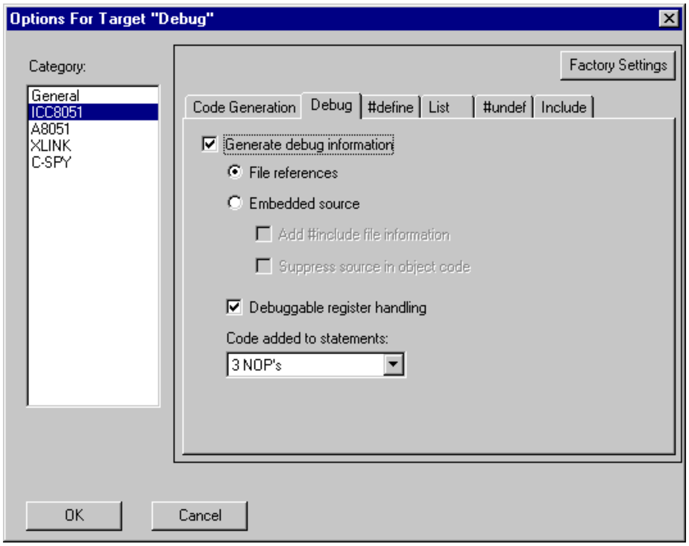
Figure 52: Target settings

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

Select **ICC8051** in the **Category** list to display the IAR C Compiler options pages. Keep the default settings; in addition, specify the following compiler options from the appropriate pages of the **Options** dialog box:

Page	Options
Code Generation	Optimizations, Size: Low
Debug	Code added to statements: 3 NOP's
List	List file Insert mnemonics

Table 14: Tutorial 9 compiler options



☑ Generate debug information
● File references
○ Embedded source
☐ Add #include file information
☐ Suppress source in object code
☑ Debuggable register handling
Code added to statements: 3 NOP's

Figure 53: Setting compiler options for the ROM-monitor

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

COMPILING THE DEMO.C AND DEMO_TWO.C FILES

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

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

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

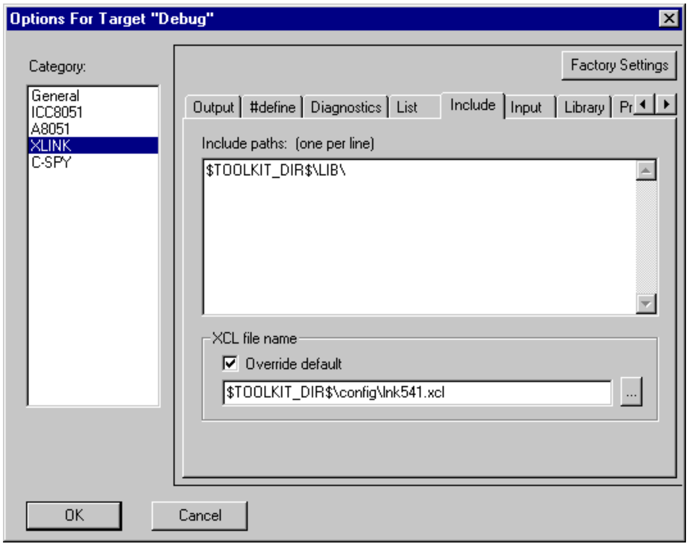
SETTING XLINK OPTIONS

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

On the **List** page, select the **Generate linker listing**, **Segment map**, and **Memory map** options.

Select the **Include** page. Under **XCL filename**, select **Override default** and specify the appropriate linker command file for your project, for example:

```
$TOOLKIT_DIR$\config\lnk541.xcl
```



Options For Target "Debug"

Category: XLINK

Include paths: (one per line)

```
$TOOLKIT_DIR$\LIB\
```

XCL file name

☑ Override default

$TOOLKIT_DIR$\config\lnk541.xcl

Figure 54: Tutorial 9 XLINK options

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

LINKING THE PROJECT

Now you should link the object file to generate code that can be debugged. Choose **Link** from the **Project** menu.

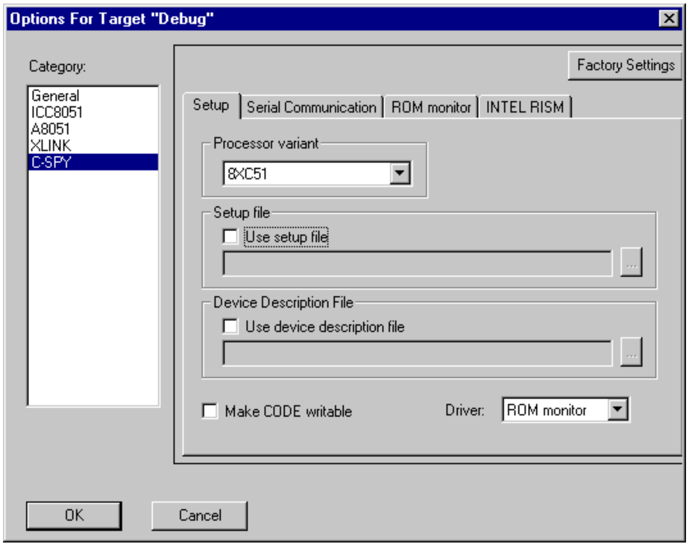
The result of the linking is a code file `project5.d03` with debug information and a map file `project5.map`.

Note: The file `demo.d03` is included in the product and can be run as directly in the ROM-monitor. To start the ROM-monitor with the appropriate command line settings, see the chapter *C-SPY command line options* in *Part 4: The C-SPY simulator* in this guide.

SETTING C-SPY OPTIONS

You must configure the ROM-monitor before running the program. Select the **Debug** folder icon in the Project window and choose **Options...** from the **Project** menu. Then select **C-SPY** in the **Category** list to display the C-SPY options pages.

In the **Setup** page, select **ROM-monitor** from the **Driver** drop-down list:



Processor variant: 8XC51

Setup file: ☐ Use setup file

Device Description File: ☐ Use device description file

☐ Make CODE writable

Driver: ROM monitor

Figure 55: Selecting the ROM-monitor driver

Select the **Serial Communication** and **ROM monitor** pages to view the available alternatives. The exact settings depend on which evaluation board you are using; for example, the C541 board uses a baud rate of 19200.

Then click **OK** to leave the **Options** dialog box.

RUNNING THE PROGRAM

Choose **Debugger** from the **Project** menu in the IAR Embedded Workbench. Alternatively, click the **C-SPY** button in the toolbar. Open the Terminal I/O window in C-SPY.

To run the program at full speed, select **Go** from the **Execute** menu, or click the **Go** button in the toolbar.

Click on the **Stop** button to stop execution.

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

Part 3: The IAR Embedded Workbench

This part of the 8051 IAR Embedded Workbench™ User Guide contains the following chapters:

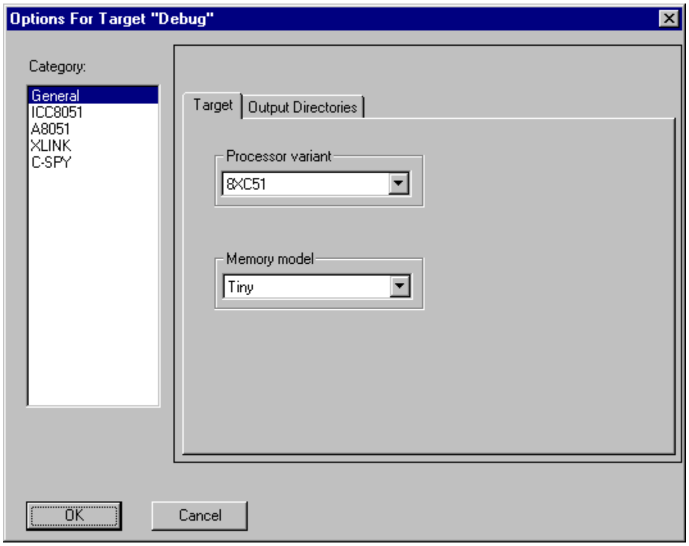
- General options
- Compiler options
- Assembler options
- XLINK options
- C-SPY options
- IAR Embedded Workbench reference.

General options

This chapter describes how to set general options in the IAR Embedded Workbench™. These include how to specify the target processor and memory model, as well as how to set up output directories.

Setting general options

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



Processor variant: 8XC51

Memory model: Tiny

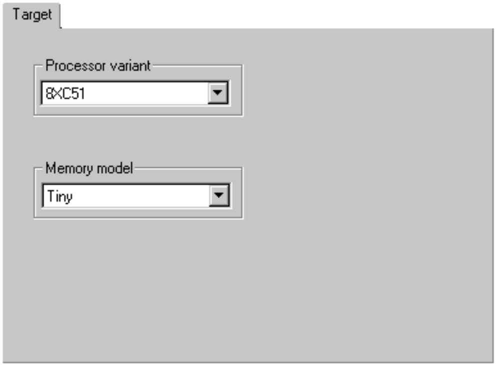
Figure 56: Setting general options

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

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

Target

The **Target** options in the **General** category specify the target processor and memory model for the 8051 IAR C Compiler and Assembler.



Processor variant: 8XC51

Memory model: Tiny

Figure 57: Target options

PROCESSOR VARIANT

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

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

What you chose as a processor variant determines the availability of memory model options.

For a description of the available options, see the *Configuration* chapter in the *8051 IAR C Compiler Reference Guide*.

MEMORY MODEL

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

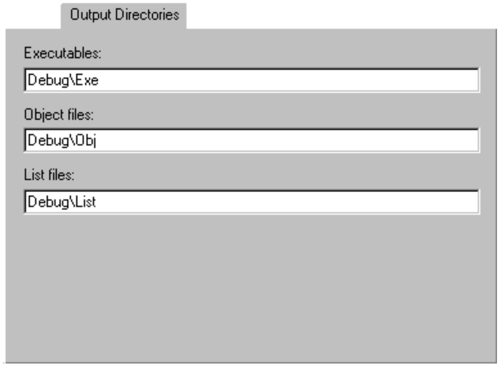
Select the memory model for your project from the drop-down list.

What you chose as a processor variant determines the availability of memory model options.

For a description of the available options, see the *Configuration* chapter in the *8051 IAR C Compiler Reference Guide*.

Output directories

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



Executables: Debug\Exe

Object files: Debug\Obj

List files: Debug\List

Figure 58: Output directories

Executables

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

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

Object files

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

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

List files

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

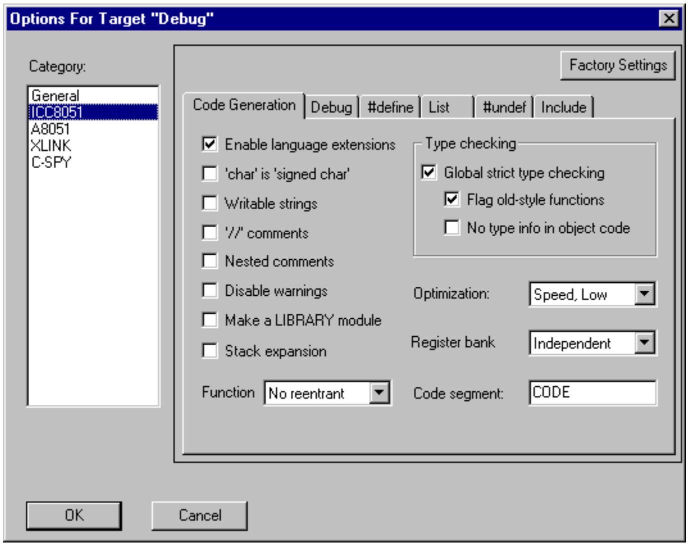
Enter the name of the directory where you want to save list files for the project.

Compiler options

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

Setting compiler options

To set compiler options in the IAR Embedded Workbench, select **Options…** from the **Project** menu to display the **Options** dialog box. Select **ICC8051** in the **Category** list to display the compiler options pages:



Category:
- General
- ICC8051
- A8051
- XLINK
- C-SPY

☑ Enable language extensions
☐ 'char' is 'signed char'
☐ Writable strings
☐ '//' comments
☐ Nested comments
☐ Disable warnings
☐ Make a LIBRARY module
☐ Stack expansion

Type checking
☑ Global strict type checking
☑ Flag old-style functions
☐ No type info in object code

Optimization: Speed, Low
Register bank: Independent
Function: No reentrant
Code segment: CODE

Figure 59: Compiler options

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

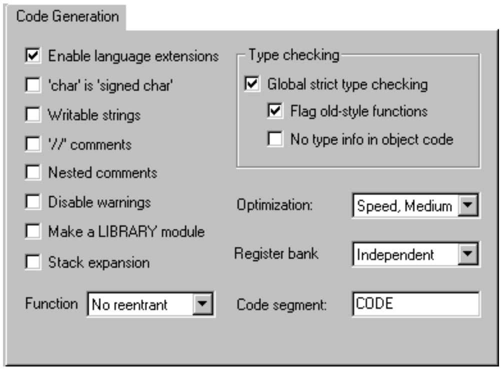
Notice that compiler options can be specified on a target level, group level, or file level. When options are set on the group or file level, you can choose to override settings inherited from a higher level.

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

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

Code generation

The **Code Generation** options determine the interpretation of the source program and the generation of object code.



Code Generation

☑ Enable language extensions
☐ 'char' is 'signed char'
☐ Writable strings
☐ '//' comments
☐ Nested comments
☐ Disable warnings
☐ Make a LIBRARY module
☐ Stack expansion

Function: No reentrant

Type checking
☑ Global strict type checking
  ☑ Flag old-style functions
  ☐ No type info in object code

Optimization: Speed, Medium

Register bank: Independent

Code segment: CODE

Figure 60: Compiler code generation options

ENABLE LANGUAGE EXTENSIONS

Use this option to enable target-dependent extensions to the C language. If your source code contains language extensions, this option must be selected.

‘CHAR’ IS ‘SIGNED CHAR’

Use this option to make the `char` type equivalent to `signed char`.

By default the compiler interprets the `char` type as `unsigned char`. To make the compiler interpret the `char` type as `signed char` instead, for example for compatibility with a different compiler, use this option.

WRITABLE STRINGS

Use this option to make the compiler treat string literals and other constants as initialized variables.

By default string literals and constants are compiled as read-only. If you want to be able to write to them, use this option which causes them to be compiled as writable variables.

Note: Arrays initialized with strings (i.e. `char c[] = ` *`string`*) are always compiled as initialized variables, and are not affected by the **Writable strings** option.

‘//’ COMMENTS

Use this option to enable comments in C++ style, i.e. comments introduced by ‘`//`’ and extending to the end of the line.

For compatibility reasons, the compiler normally does not accept C++ style comments. If your source includes C++ style comments, you must use this option to make them accepted.

NESTED COMMENTS

Use this option to enable nested comments.

By default the compiler treats nested comments as a fault and issues a warning when it encounters one, resulting for example from a failure to close a comment. If you want to use nested comments, for example to comment out sections of code that include comments, use this option to disable this warning.

DISABLE WARNINGS

Use this option to disable all warnings issued by the compiler.

By default the compiler issues standard warning messages, and any additional warning messages enabled with the **Global strict type checking** option.

MAKE A LIBRARY MODULE

By default the compiler produces a program module ready for linking with the IAR XLINK Linker.

Use this option if you instead want a library module for inclusion in a library with the IAR XLIB Librarian.

STACK EXPANSION

Use this option if you want to force the compiler to store function return addresses in external (XDATA) memory.

FUNCTION

Without this option, simple recursive functions will work correctly. However, mutual recursion may not function as expected because local variables are stored in fixed locations rather than on a stack.

TYPE CHECKING

Global strict type checking

Use this option to make the compiler check type information throughout the source.

Sometimes there are conditions in the source code that indicate possible programming faults but which for compatibility reasons the compiler and linker normally ignore. To cause the compiler and linker to issue a warning each time they encounter such a condition, use this option.

Flag old-style functions

By default the **Global strict type checking** option does not warn of old-style K&R functions. To enable such warnings, use the **Flag old-style functions** option.

No type info in object code

By default the **Global strict type checking** option includes type information in the object module which allows the linker to issue type check warnings. This information increases the size of the module and the link time. To exclude this information and disable the linker type check warnings, use the **No type info in object code** option.

When linking multiple modules, notice that objects in a module compiled without type information, i.e. without the **Global strict type checking** option, are considered typeless. Hence there will never be any warning of a type mismatch from a declaration from a module compiled without type information, even if the module with a corresponding declaration has been compiled with type information.

The conditions checked by the **Global strict type checking** option are:

- Calls to undeclared functions.
- Undeclared K&R formal parameters.
- Missing return values in non-void functions.
- Unreferenced local or formal parameters.
- Unreferenced `goto` labels.
- Unreachable code.
- Unmatching or varying parameters to K&R functions.
- `#undef` of unknown symbols.
- Valid but ambiguous initializers.
- Constant array indexing out of range.

OPTIMIZATION

By default the compiler optimizes for minimum size at level 3 (see the table below).

Use this option to make the compiler optimize for size or speed at the selected level:

Level	Option	Description
0	None	No optimization.
3	Low	Fully debuggable.
6	Medium	Some constructs not debuggable.
8	High	Heavy optimization can make the program flow hard to follow during debug.
9	Full	Full optimization

Table 15: Optimization options

REGISTER BANK

This option allows the generation of register bank-dependent code. The parameter specifies the register bank 0 to 3. The default is 0.

The options are as follows:

Option	Command line
Independent	
Register Bank 0	-h0
Register Bank 1	-h1
Register Bank 2	-h2
Register Bank 3	-h3

Table 16: Register bank options

CODE SEGMENT

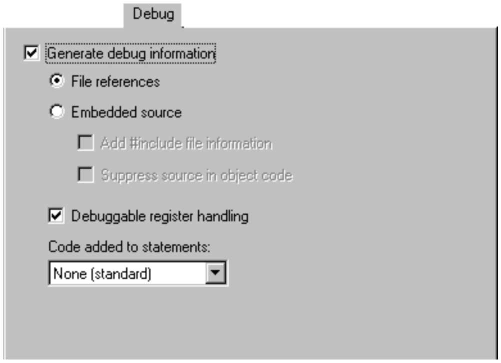
Use this option to specify the name of the code segment.

By default the compiler places executable code in the segment named `CODE` which, also by default, the linker places at a variable address.

If you want to be able to specify an explicit address for the code, you use this option to specify a special code segment name which you can then assign to a fixed address in the linker command file.

Debug

The **Debug** options determine the level of debug information included in the object code.



☑ Generate debug information

● File references

○ Embedded source

☐ Add #include file information

☐ Suppress source in object code

☑ Debuggable register handling

Code added to statements: None (standard)

Figure 61: Compiler debugging options

GENERATE DEBUG INFORMATION

Use this option to make the compiler include additional information required by C-SPY and other symbolic debuggers in the object modules.

Normally the compiler does not include debugging information, for code efficiency. To make code debuggable with C-SPY, you simply use this option with the **File references** option, which is selected by default. This adds source file references, symbol debug information, and other debug information to the object file.

To make code debuggable with debuggers that read the UBROF5 object file format only, it is necessary to use the **Embedded source** option to include the full source file into object code. It will then be possible to generate the UBROF 5 object file format from the linker at a later stage.

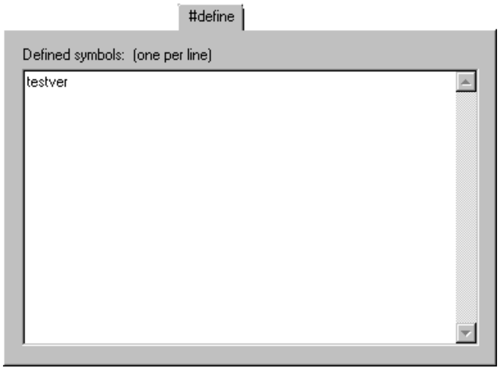
The option **Add #include file information** will insert include files into the copied source as well, giving the possibility to debug code statements inside include files. A side effect is that the source line number is the global (=total) line count so far in the copied source. The option **Suppress source in object file** will give the same line count but will not embed the source files into the object file.

Normally, the compiler tries to put locals as register variables. To suppress the use of register variables, use **Debuggable register handling**.

Use the **Code added to statements** option to add a `NOP` in front of every C statement. Only use this option if your debugging tools specifically require you to do so.

#define

The **#define** option allows you to define symbols for use by the C compiler.



Defined symbols: (one per line)

testver

Figure 62: Compiler #define options

DEFINED SYMBOLS

Defines a symbol with the name *`symb`* and the value *`xx`*. If no value is specified, `1` is used.

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

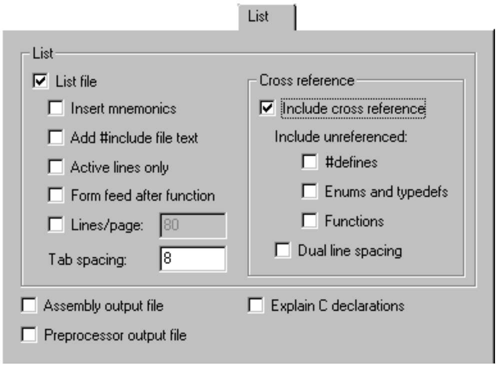
The **Defined symbols** option is useful for conveniently specifying a value or choice that would otherwise be specified in the source file.

For example, you could arrange your source to produce either the test or production version of your program depending on whether the symbol `testver` was defined. To do this you would use include sections such as:

```
#ifdef  testver
   ...  ; additional code lines for test version only
#endif
```

List

The **List** options determine whether a listing is produced, and the type of information included in the listing.



List

List

☑ List file

☐ Insert mnemonics

☐ Add #include file text

☐ Active lines only

☐ Form feed after function

☐ Lines/page: 80

Tab spacing: 8

Cross reference

☑ Include cross reference

Include unreferenced:

☐ #defines

☐ Enums and typedefs

☐ Functions

☐ Dual line spacing

☐ Assembly output file

☐ Explain C declarations

☐ Preprocessor output file

Figure 63: Compiler list file options

LIST FILE

Insert mnemonics

Use this option to make the listing include generated assembly lines.

By default the compiler does not include the generated assembly lines in the listing. If you want these to be included, for example to be able to check the efficiency of code generated by a particular statement, use this option.

Add #include file text

Use this option to make the listing include `#include` files.

Normally the listing does not include `#include` files, since they usually contain only header information that would waste space in the listing. If they for example contain function definitions or preprocessed lines, use this option to have them included in the listing.

Active lines only

Use this option to make the compiler list only active source lines.

Normally the compiler lists all source lines. To save listing space by eliminating inactive lines, such as those in false `#if` structures, you use this option.

Form feed after function

Use this option to generate a form-feed after each listed function in the assembly listing.

By default the listing simply starts each function on the next line. To cause each function to appear at the top of a new page, you would include this option.

Form-feeds are never generated for functions that are not listed, for example, as in `#include` files.

Lines/page

This option causes the listing to be formatted into pages, and specifies the number of lines per page in the range 10 to 150.

By default the listing is not formatted into pages. To format it into pages with a form feed at every page, you use this option.

TAB SPACING

Use this option to set the number of character positions per tab stop to *`n`*, which must be in the range 2 to 9.

By default the listing is formatted with a tab spacing of 8 characters. If you want a different tab spacing, you set it with this option.

CROSS REFERENCE

Use this option to include a cross-reference list in the listing.

By default the compiler does not include global symbols in the listing. To include at the end of the listing a list of all variable objects, and all functions, `#define` statements, `enum` statements, and `typedef` statements that are referenced, you use this option.

You can also choose to use dual-line spacing in the listing.

ASSEMBLY OUTPUT FILE

Use this option to generate assembler source code to *`filename`*`.s03`.

By default the compiler does not generate an assembler source. To send assembler source to the file with the same name as the source leafname but with the extension `s03`, use this option.

If you also select the options **List file** and **Insert mnemonics**, the C source lines are included in the assembly source file as comments.

PREPROCESSOR OUTPUT FILE

Use this option to generate preprocessor output to `filename.i`.

By default the compiler does not generate preprocessor output. To send preprocessor output to the file with the same name as the source leafname but with the extension `i`, use this option.

EXPLAIN C DECLARATIONS

Use this option to include an English description of each C declaration in the file.

This may aid, for example in the investigation of error messages. The following example shows a C declaration and its description:

The declaration:

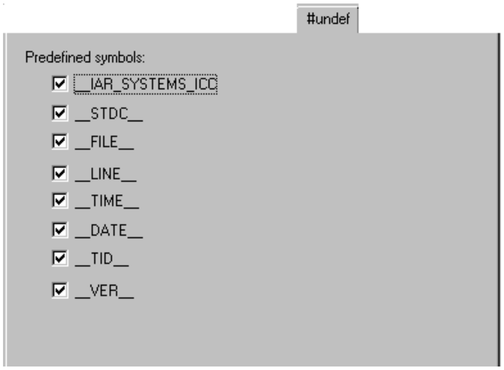
```
void (* signal(int __sig, void (* func) ())) (int);
```

gives the description:

```
Identifier: signal
storage class: extern
  prototyped near_func function returning
    near - near_func code pointer to
      prototyped near_func function returning
        near - void
      and having following parameter(s):
        storage class: auto
        near - int
  and having following parameter(s):
    storage class: auto
    near - int
    storage class: auto
    near - near_func code pointer to
      near_func function returning
        near - void
```

#undef

The **#undef** options allow you to undefine the predefined symbols.



Predefined symbols:

☑ __IAR_SYSTEMS_ICC

☑ __STDC__

☑ __FILE__

☑ __LINE__

☑ __TIME__

☑ __DATE__

☑ __TID__

☑ __VER__

Figure 64: Compiler #undef options

UNDEFINE SYMBOL

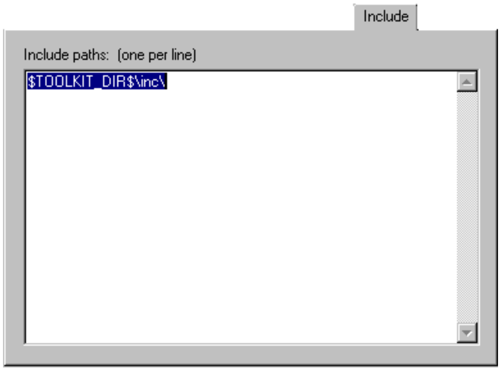
Use this option to remove the definition of the named symbol.

By default the compiler provides various predefined symbols. If you want to remove one of these, for example to avoid a conflict with a symbol of your own with the same name, you use this option.

For a list of the predefined symbols, see the *8051 IAR C Compiler Reference Guide*.

Include

The **Include** option allows you to define include paths for the C compiler.



$TOOLKIT_DIR$\inc\

Figure 65: Compiler Include path options

INCLUDE PATHS

Use this option to add a path to the list of `#include` file paths.

By default the compiler searches for include files only in the source directory (if the filename is enclosed in quotes as opposed to angle brackets), the `C_INCLUDE` paths, and finally the current directory. If you have placed `#include` files in another directory, you must use this option to inform the compiler of that directory.

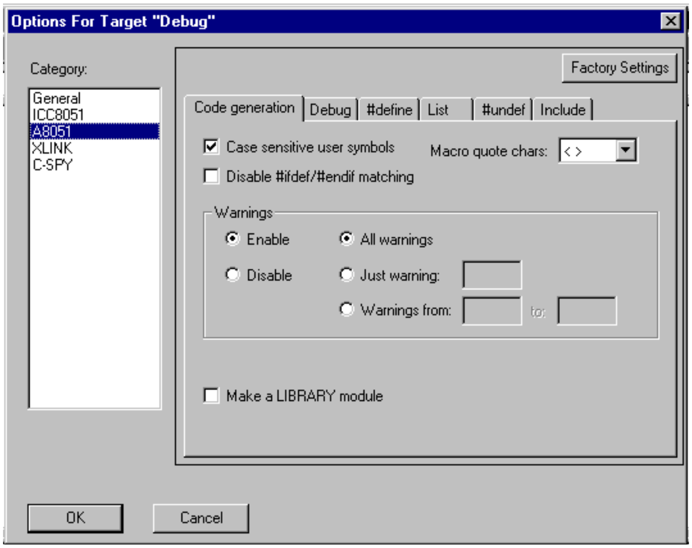
Enter the full file path of your `#include` files or use an argument variable, see *Configure tools…*, page 161 and onwards.

Assembler options

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

Setting assembler options

To set assembler options in the IAR Embedded Workbench, choose **Options…** from the **Project** menu to display the **Options** dialog box. Then select **A8051** in the **Category** list to display the assembler options pages:



☑ Case sensitive user symbols
Macro quote chars: < >
☐ Disable #ifdef/#endif matching

Warnings
● Enable
○ Disable
● All warnings
○ Just warning:
○ Warnings from: to:

☐ Make a LIBRARY module

Figure 66: Assembler options

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

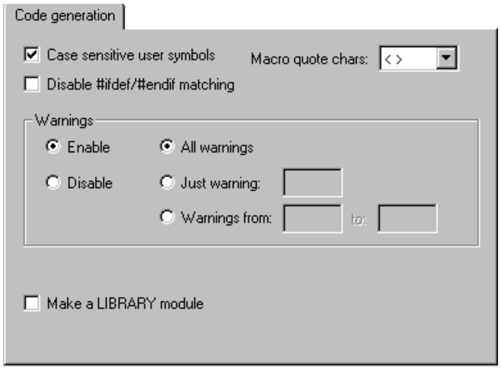
Notice that assembler options can be specified on a target level, a group level, or a file level. When options are set on the group or file level, you can choose to override settings inherited from a higher level.

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

The following sections give full descriptions of each assembler option.

Code generation

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



☑ Case sensitive user symbols

Macro quote chars: < >

☐ Disable #ifdef/#endif matching

● Enable

○ Disable

● All warnings

○ Just warning:

○ Warnings from:

☐ Make a LIBRARY module

Figure 67: Assembler code generation options

CASE SENSITIVE USER SYMBOLS

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

DISABLE #IFDEF/#ENDIF MATCHING (-D)

This option allows unmatched `#ifdef...#endif` statements to be used without causing an error.

The checks for `#ifdef...#endif` matching are performed for each module, and a `#endif` outside modules will therefore normally generate an error message. Use this option to turn checking off.

This allows you to write constructs such as:

```
#ifdef  Version1

   MODULE M1
   NOP
   ENDMOD

#endif
   MODULE M2
   .
   .
   .
```

WARNINGS

The assembler displays a warning message when it finds an element of the source that is legal, but probably the result of a programming error (see the *8051 IAR Assembler Reference Guide*).

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

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

MAKE A LIBRARY MODULE

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

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

MACRO QUOTE CHARS

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

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

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

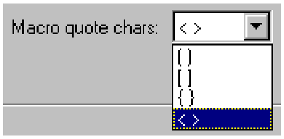
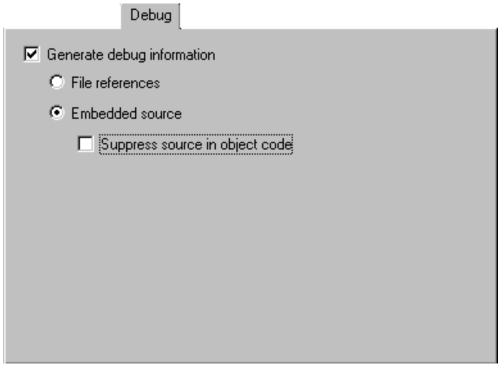


Figure 68: Selecting macro quote characters

Debug

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



☑ Generate debug information

○ File references

● Embedded source

☐ Suppress source in object code

Figure 69: Assembler debugging options

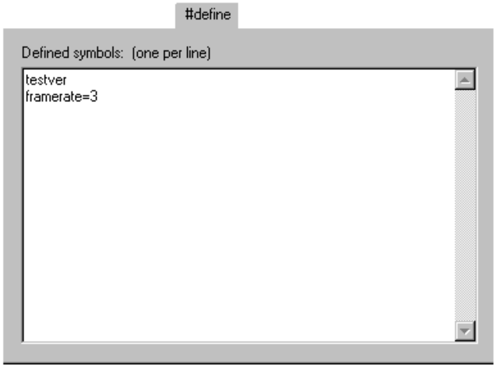
GENERATE DEBUG INFORMATION

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

When you select this option to generate debug information, **File references** is selected by default. If you instead want to include the entire source file into the object file, select **Embedded source**.

#define

The **#define** option allows you to define symbols in addition to the predefined symbols in the 8051 Assembler.



testver
framerate=3

Figure 70: Assembler #define options

#define

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

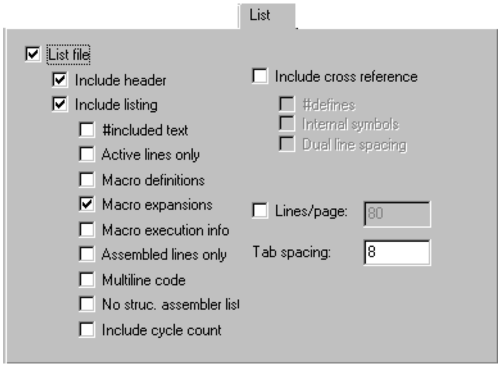
Enter the symbols you want to define in the **#define** page, one per line.

- For example, you could arrange your source to produce either the test or production version of your program depending on whether the symbol `testver` was defined. To do this you would use include sections such as:
  ```
  #ifdef  testver
  ...  ; additional code lines for test version only
  #endif
  ```
  You would then define the symbol `testver` in the Debug target but not in the Release target.
- Alternatively, your source might use a variable that you need to change often, for example `framerate`. You would leave the variable undefined in the source and use the **#define** option to specify a value for the project, for example **framerate=3**.

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

List

The **List** options are used to cause the assembler to generate a listing, to select the contents of the listing, and to generate other listing-type output.



List

☑ List file

☑ Include header

☑ Include listing

☐ #included text

☐ Active lines only

☐ Macro definitions

☑ Macro expansions

☐ Macro execution info

☐ Assembled lines only

☐ Multiline code

☐ No struc. assembler lis

☐ Include cycle count

☐ Include cross reference

☐ #defines

☐ Internal symbols

☐ Dual line spacing

☐ Lines/page: 80

Tab spacing: 8

Figure 71: Assembler list file options

LIST FILE

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

Note: If you want to save the list file in another directory than the default directory for list files, use the **Output Directories** option in the **General** category; see *Output directories*, page 93, for additional information.

When **List file** is selected the following list options become available:

Option	Description
Include header	Includes a header in the listing.
Include listing	Includes the body of the listing.
#included text	Includes `#include` files in the listing.
Active lines only	Includes only active lines in the listing.
Macro definitions	Includes macro definitions in the listing.
Macro expansions	Includes macro expansions in the listing.
Macro execution info	Prints macro execution information on every call of a macro.
Assembled lines only	Excludes lines in false conditional assembly sections from the listing.

Table 17: Assembler list file options

Option	Description
Multiline code	Lists the code generated by directives on several lines, if necessary.
No structured assembler list	Excludes structured assembly sections from the listing.
Include cycle count	Includes the cycle count.

Table 17: Assembler list file options (continued)

INCLUDE CROSS-REFERENCE

The **Include cross-reference** option causes the assembler to generate a cross-reference table at the end of the listing. For an example, see *Listing format* in the *8051 IAR Assembler Reference Guide* for details.

LINES/PAGE

The default number of lines per page is 44 for the assembler listing. Use this option to set the number of lines per page, within the range 10 to 150.

Use the **Lines/page** option to set the number of lines per page for the assembler listing.

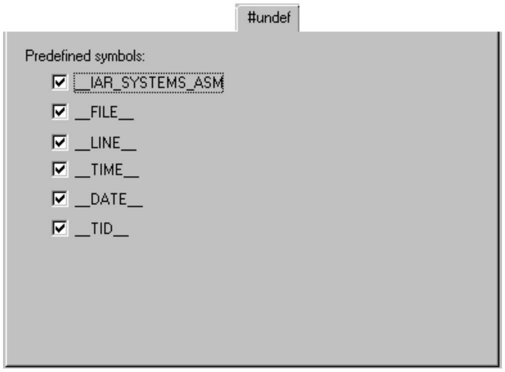
TAB SPACING

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.

Enter your preferred number of character positions per tab stop.

#undef

The **#undef** option allows you to undefine the predefined symbols provided in the assembler.



Predefined symbols:

☑ __IAR_SYSTEMS_ASM

☑ __FILE__

☑ __LINE__

☑ __TIME__

☑ __DATE__

☑ __TID__

Figure 72: Assembler #undef options

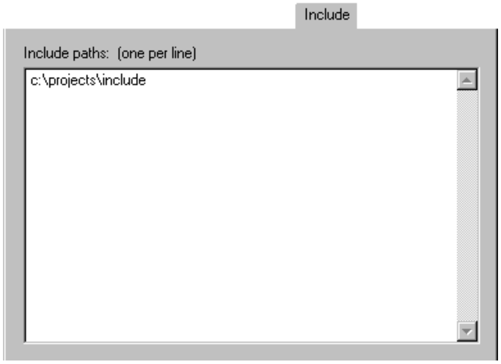
#UNDEF

By default, the assembler provides certain predefined symbols; see the *8051 IAR Assembler Reference Guide* for more information. The **#undef** option allows you to undefine such a predefined symbol to make its name available for your own use through a subsequent **#define** option or source definition.

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

Include

The **Include** option allows you to define the include path for the assembler.



Include paths: (one per line)

c:\projects\include

Figure 73: Assembler Include path options

INCLUDE

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

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

See the *8051 IAR Assembler Reference Guide* for information about the `#include` directive.

Note: By default the assembler searches for `#include` files also in the paths specified in the `A8051_INC` environment variable. We do not, however, recommend the use of environment variables in the IAR Embedded Workbench.

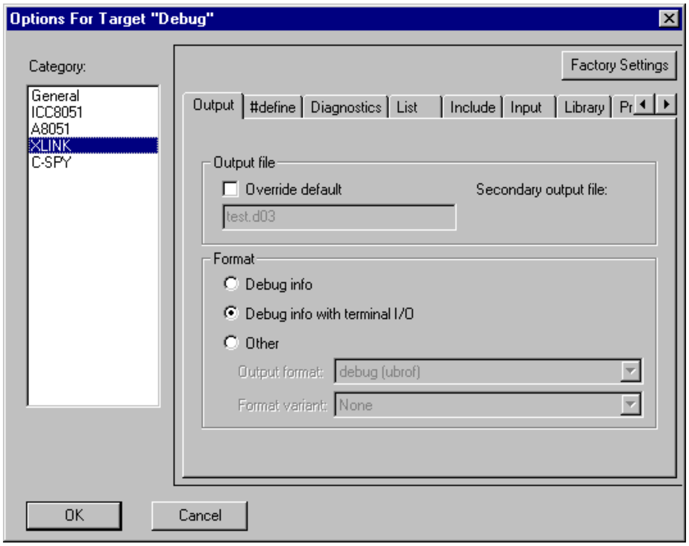
XLINK options

This chapter describes how to set XLINK options and gives reference information about the options available in the IAR Embedded Workbench. XLINK options allow you to control the operation of the IAR XLINK Linker™.

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

Setting XLINK options

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



Output file

☐ Override default

test.d03

Secondary output file:

Format

○ Debug info

● Debug info with terminal I/O

○ Other

Output format: debug (ubrof)

Format variant: None

Figure 74: XLINK options

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

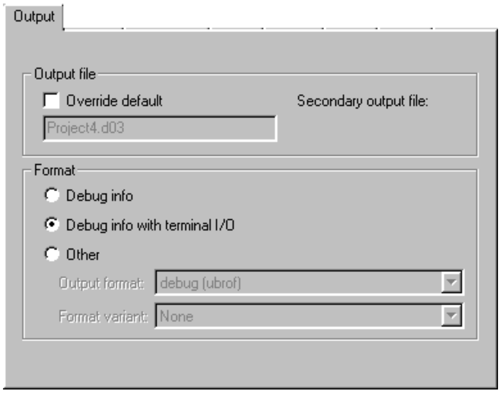
Notice that XLINK options can be specified on a target level, a group level, or a file level. When options are set on the group or file level, you can choose to override settings inherited from a higher level.

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

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

Output

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



Output file

☐ Override default

Secondary output file:

Project4.d03

Format

○ Debug info

● Debug info with terminal I/O

○ Other

Output format: debug (ubrof)

Format variant: None

Figure 75: 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 name `project.d03`. If a name is supplied without a file type, the default file type for the selected output format (see *Output format*, page 119) will be used.

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

Override default

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

FORMAT

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

Debug info

Use this option to create an output file in **debug (ubrof)** format, with a `d03` extension, to be used with the IAR C-SPY® Debugger.

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

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

Debug info with terminal I/O

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

Output format

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

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

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

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

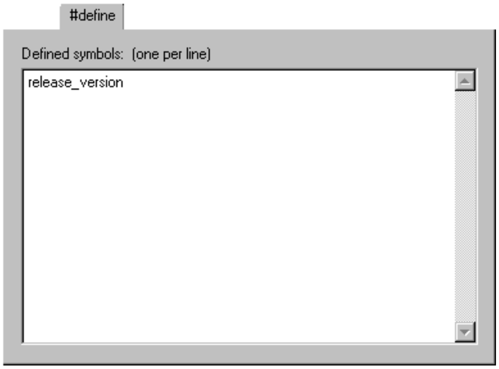
Format variant

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

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

#define

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



Defined symbols: (one per line)

release_version

Figure 76: XLINK defined symbols options

DEFINE SYMBOL

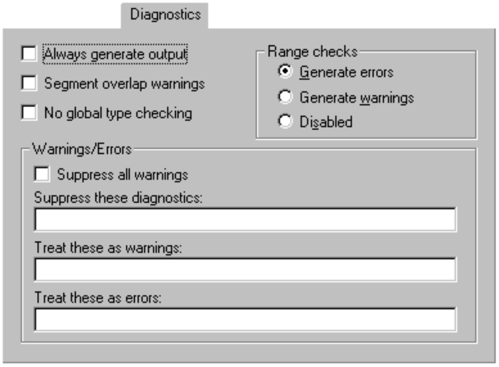
Use **Define symbol** to define absolute symbols at link time. This is especially useful for configuration purposes.

Any number of symbols can be defined in a linker command file. The symbol(s) defined in this manner will be located in a special module called `?ABS_ENTRY_MOD`, which is generated by the linker.

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

Diagnostics

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



Diagnostics

☐ Always generate output
☐ Segment overlap warnings
☐ No global type checking

Range checks
● Generate errors
○ Generate warnings
○ Disabled

Warnings/Errors
☐ Suppress all warnings
Suppress these diagnostics:
Treat these as warnings:
Treat these as errors:

Figure 77: 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 program should not need this option, there may be occasions where it is helpful.

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

RANGE CHECKS

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

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

Table 18: XLINK range check options

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

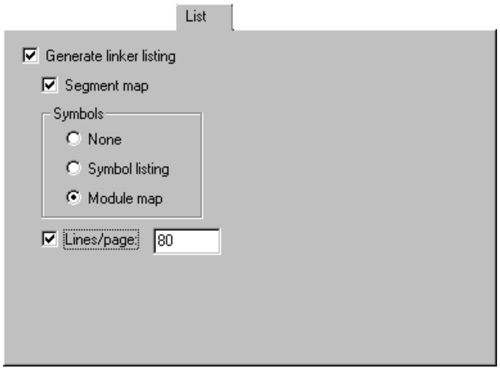
WARNINGS/ERRORS

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

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

List

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



☑ Generate linker listing
  ☑ Segment map
  Symbols
  ○ None
  ○ Symbol listing
  ● Module map
  ☑ Lines/page: 80

Figure 78: XLINK list file options

GENERATE LINKER LISTING

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

Segment map

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

Symbols

The following options are available:

Option	Description
None	Symbols will be excluded from the linker listing.
Symbol listing	An abbreviated list of every entry (global symbol) in every module. This entry map is useful for quickly finding the address of a routine or data element.
Module map	A list of all segments, local symbols, and entries (public symbols) for every module in the program.

Table 19: XLINK list file 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.

Include

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

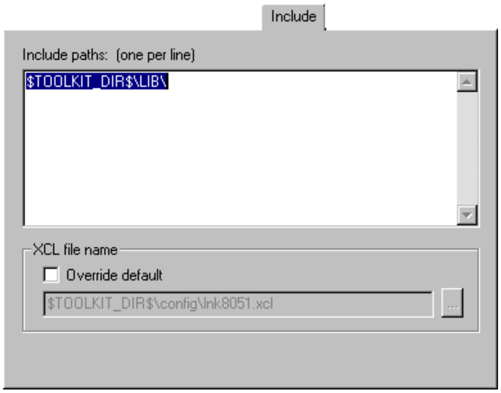

Include
Include paths: (one per line)
$TOOLKIT_DIR$\LIB\
XCL file name
☐ Override default
$TOOLKIT_DIR$\config\lnk8051.xcl
...

Figure 79: XLINK include files options

INCLUDE PATHS

Specifies a path name to be searched for object files.

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

To make products more portable, use the argument variable `$TOOLKIT_DIR$\lib\` for the `lib` subdirectory of the active product (that is, standard system `#include` files) and `$PROJ_DIR$\lib\` for the `lib` subdirectory of the current project directory. For an overview of the argument variables, see *Table 31*, page 162.

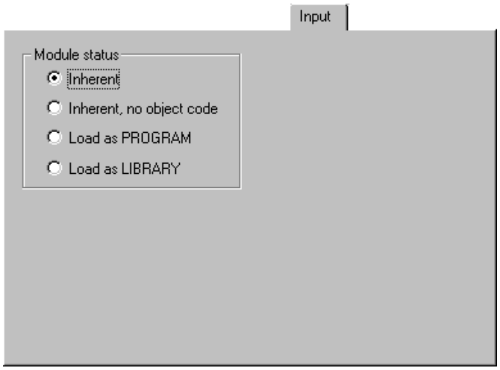
XCL FILENAME

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

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

Input

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



● Inherent
○ Inherent, no object code
○ Load as PROGRAM
○ Load as LIBRARY

Figure 80: XLINK input files options

MODULE STATUS

Inherent

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

Inherent, no object code

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

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

Load as PROGRAM

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

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

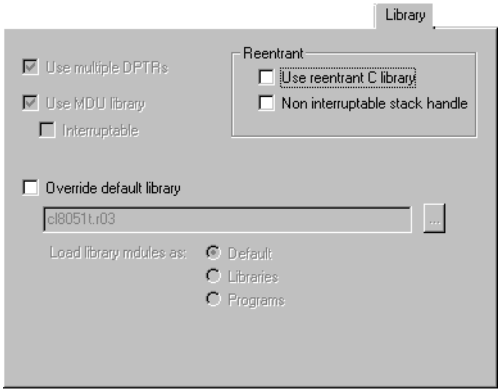
Load as LIBRARY

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

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

Library

The **Library** options allow you to set the library options and override the default linker command file.



☑ Use multiple DPTRs

☑ Use MDU library

☐ Interruptable

☐ Use reentrant C library

☐ Non interruptable stack handle

☐ Override default library

cl8051t.r03

Load library mdules as: ● Default ○ Libraries ○ Programs

Figure 81: XLINK library options

USE MULTIPLE DPTRS

This option will include a library supporting multiple DPTRs. The library used is determined by the processor selected in the **General** category.

Microcontroller	Library
8X517	cl517str.r03
8X320	cl320str.r03

Table 20: Libraries supporting multiple DPTRs

Also see the *Configuration* chapter in the *8051 IAR C Compiler Reference Guide*.

USE MDU LIBRARY

This option will include a library supporting the extended math unit (MDU) in the 8X517 microcontroller. The option includes `cl8517.r03` if **Interruptable** is not selected. If **Interruptable** is selected, `cl8517i.r03` is included.

Also see the *Configuration* chapter in the *8051 IAR C Compiler Reference Guide*.

REENTRANT

Use the **Use reentrant C library** option to specify that the linker should produce reentrant code.

The library used is as follows:

Memory model	Non-reentrant	Reentrant
Tiny	cl8051t.r03	cl8051tr.r03
Small	cl8051s.r03	cl8051sr.r03
Compact	cl8051c.r03	cl8051cr.r03
Medium	cl8051m.r03	cl8051mr.r03
Large	cl8051l.r03	cl8051lr.r03
Banked	cl8051b.r03	cl8051br.r03

Table 21: Non-reentrant and reentrant C library files

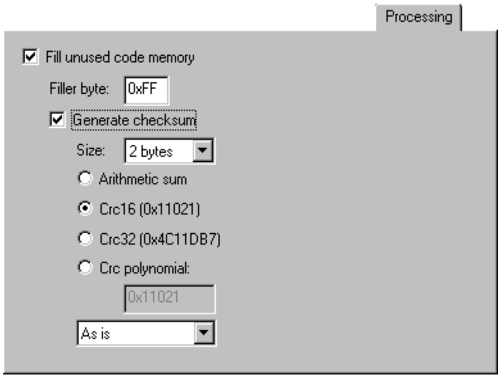
Use the **Non-interruptable stack handle** option to load the library `move_xsp.r03` before the standard library; this disables interrupts before calls to reentrant functions, and enables them again afterwards, to reduce the stack requirements. Note that the `move_xsp` library affects all reentrant functions, both user-defined and system functions.

OVERRIDE DEFAULT LIBRARY

A default library file is selected automatically for the memory model you specify in the **General** category. You can override this by selecting **Override default library name**, and then specifying an alternative library file. You can also specify how to load the library modules.

Processing

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



☑ Fill unused code memory

Filler byte: 0xFF

☑ Generate checksum

Size: 2 bytes

○ Arithmetic sum

● Crc16 (0x11021)

○ Crc32 (0x4C11DB7)

○ Crc polynomial:

0x11021

As is

Figure 82: 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 `hexvalue`. The linker can introduce gaps either because of alignment restriction, or at the end of ranges given in segment placement options.

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

Filler byte

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

Generate checksum

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

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

One of the following algorithms can be used:

Algorithms	Description
Arithmetic sum	Simple arithmetic sum.
Crc16	CRC16, generating polynomial 0x11021 (default)
Crc32	CRC32, generating polynomial 0x4C11DB7.
Crc polynomial	CRC with a generating polynomial of *`hexvalue`*.

Table 22: XLINK checksum algorithms

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

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

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

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

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

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

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

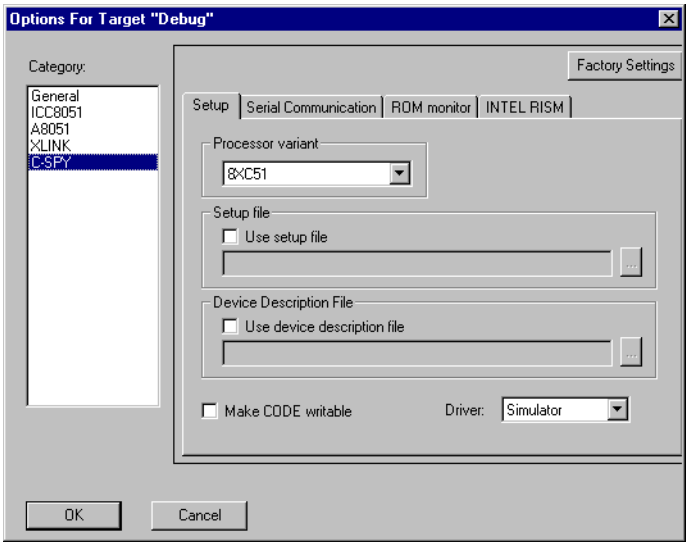
C-SPY options

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

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

Setting C-SPY options

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



Processor variant: 8XC51

Setup file: ☐ Use setup file

Device Description File: ☐ Use device description file

☐ Make CODE writable

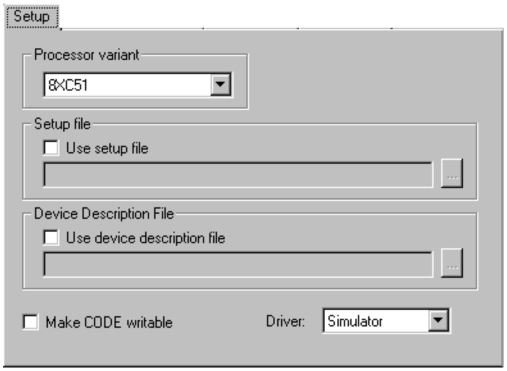
Driver: Simulator

Figure 83: C-SPY options

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

Setup

The **Setup** options specify the processor variant, the setup file, and device description file to be used.



Processor variant: 8XC51

☐ Use setup file

☐ Use device description file

☐ Make CODE writable

Driver: Simulator

Figure 84: C-SPY setup options

PROCESSOR VARIANT

In the **General** category, select the processor type to use. Here you you should select the variant closest to the actual target you will be using.

SETUP FILE

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

DEVICE DESCRIPTION FILE

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

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

MAKE CODE WRITABLE

This option will make C-SPY use the external memory as code memory. Segments that are linked as `CODE` will be placed in the external memory, and C-SPY will execute code from external memory instead of from a ROM memory. To use external data with this option, link external data at addresses not used by the `CODE` segments.

DRIVER

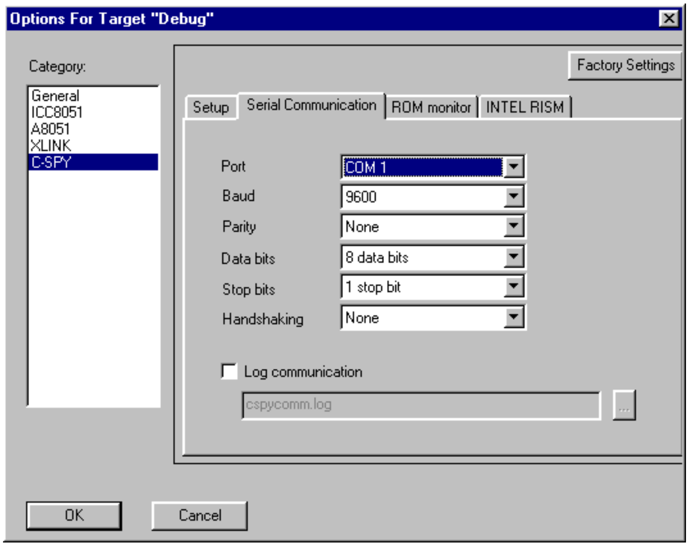
Selects one of the following drivers for use with C-SPY:

C-SPY version	Driver
Simulator	`s8051.cdr`
ROM monitor	`r8051.cdr`
Intel RISM	`r8051i.cdr`

Table 23: C-SPY driver options

Serial Communication

In the **Serial Communication** page the serial port to be used with the ROM-monitor or the Intel RISM is set up.



Port: COM 1
Baud: 9600
Parity: None
Data bits: 8 data bits
Stop bits: 1 stop bit
Handshaking: None

☐ Log communication
cspycomm.log

Figure 85: C-SPY serial communication options

C-SPY tries connecting with the selected baud rate when making the first contact with the ROM-monitor board.

If this option has not been given, C-SPY will try using the COM1 port at 9600 baud. The evaluation board must of course support the requested baud rate.

Parameters	Description
`PORT`	One of the supported ports: `COM1`, `COM2`, `COM3`, `COM4`
`BAUD`	One of following speeds: `300`, `600`, `1200`, `2400`, `4800`, `9600`, `19200`, `38400`, `57600`, `115200` (default `9600`)
`PARITY`	Only `N` (None) is allowed.
`DATA BITS`	Only `8` is allowed.
`STOP BITS`	`1` or `2` stop bits (default 1).
`HANDSHAKING`	`NONE` or `RTSCTS` (default `NONE`).

Table 24: C-SPY serial communication options

For trouble-shooting purposes, there is a possibility to log all characters sent between C-SPY and the ROM-monitor to a file. If you check the **Log communication** option, the file `cspycomm.log` will be created in the current working directory.

ROM monitor

The ROM-monitor page contains all the options specific to the ROM-monitor. The options are only available when the **ROM-monitor** driver has been selected on the **Setup** option page. See also *Part 5: C-SPY for the 8051 ROM-monitor*.

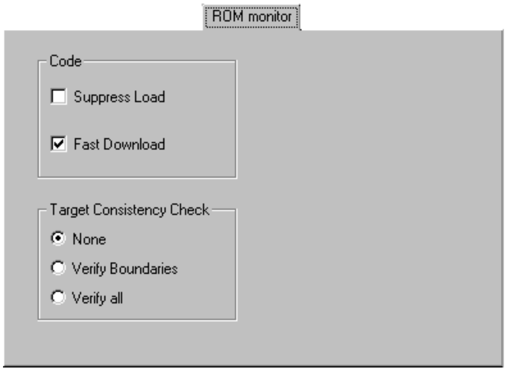


Figure 86: C-SPY ROM-monitor options

CODE

C-SPY uses a special fast download mode that runs with very little protocol overhead. The ROM-monitor must be fast enough to handle the incoming stream using full error checking on the memory or it will fail.

If fast download fails, a warning message is given by C-SPY. The download will then restart using a slower—but safer—communication protocol.

Suppress load

This option disables the downloading of code which can be time-consuming, but creates C-SPY tables internally. This command is useful if you need to exit C-SPY for a while and continue without loading code. The implicit RESET performed by C-SPY at startup is not disabled though.

Fast download

This option enables fast downloading of user code. This option is checked by default, which means that fast downloading is enabled. If you uncheck it, downloading will take more time since the error-free protocol is used. However, this should only be necessary if the ROM-monitor is not fast enough to process the data stream, or due to an insufficiently shielded communication cable.

TARGET CONSISTENCY CHECK

None

This option, which is the default, disables all target consistency checking.

Verify boundaries

This option verifies that the memory on the ADB is writable and mapped in a consistent way. A warning message will be generated if there are any problems during download.

Verify all

This option verifies download. Similar to the **Verify boundaries** option, but checks every byte after loading to verify that the hardware (ADB) is OK.

Intel RISM

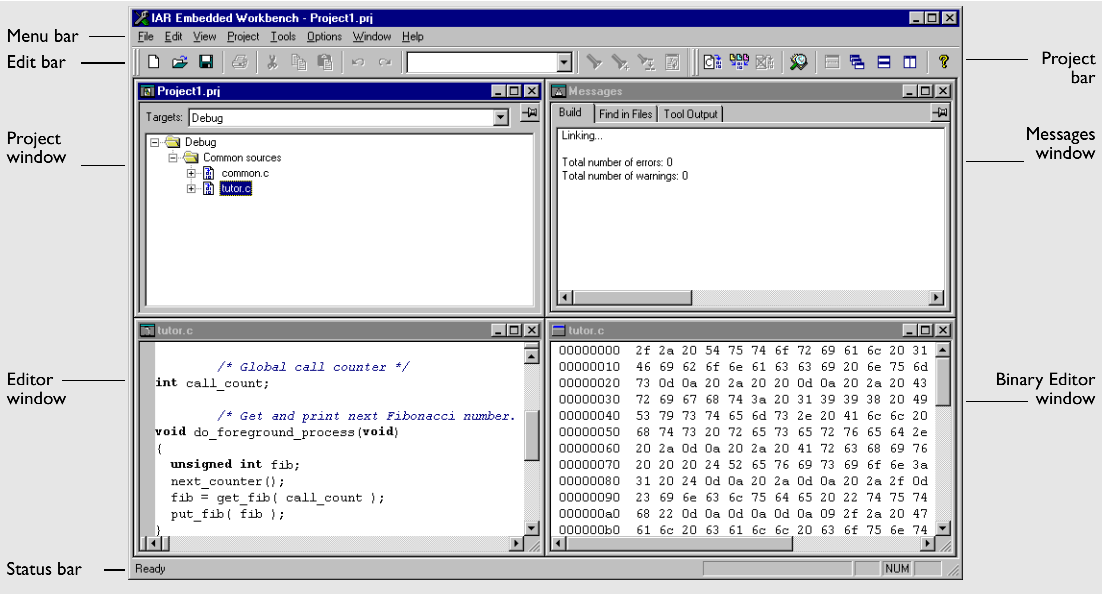
The 8051 IAR Embedded Workbench also supports the Intel RISM ROM-monitor, which is described in a separate document supplied with the product.

IAR Embedded Workbench reference

This chapter contains detailed descriptions about the windows, menus, menu commands, and their components which are found in the IAR Embedded Workbench.

The IAR Embedded Workbench window

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



```
Linking...

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

```c

        /* Global call counter */
int call_count;

        /* Get and print next Fibonacci number.
void do_foreground_process(void)
{
  unsigned int fib;
  next_counter();
  fib = get_fib( call_count );
  put_fib( fib );
}
```

```
00000000  2f 2a 20 54 75 74 6f 72 69 61 6c 20 31
00000010  46 69 62 6f 6e 61 63 63 69 20 6e 75 6d
00000020  73 0d 0a 20 2a 20 20 0d 0a 20 2a 20 43
00000030  72 69 67 68 74 3a 20 31 39 39 38 20 49
00000040  53 79 73 74 65 6d 73 2e 20 41 6c 6c 20
00000050  68 74 73 20 72 65 73 65 72 76 65 64 2e
00000060  20 2a 0d 0a 20 2a 20 41 72 63 68 69 76
00000070  20 20 20 24 52 65 76 69 73 69 6f 6e 3a
00000080  31 20 24 0d 0a 20 2a 0d 0a 20 2a 2f 0d
00000090  23 69 6e 63 6c 75 64 65 20 22 74 75 74
000000a0  68 22 0d 0a 0d 0a 0d 0a 09 2f 2a 20 47
000000b0  61 6c 20 63 61 6c 6c 20 63 6f 75 6e 74
```

Figure 87: IAR Embedded Workbench window

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

MENU BAR

Gives access to the IAR Embedded Workbench menus.

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

Table 25: IAR Embedded Workbench menu bar

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

TOOLBARS

The IAR Embedded Workbench window contains two toolbars:

- The edit bar.
- The project bar.

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

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

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

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

Edit bar

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

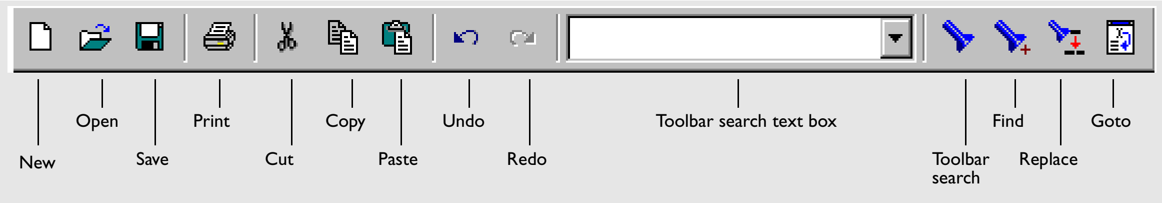
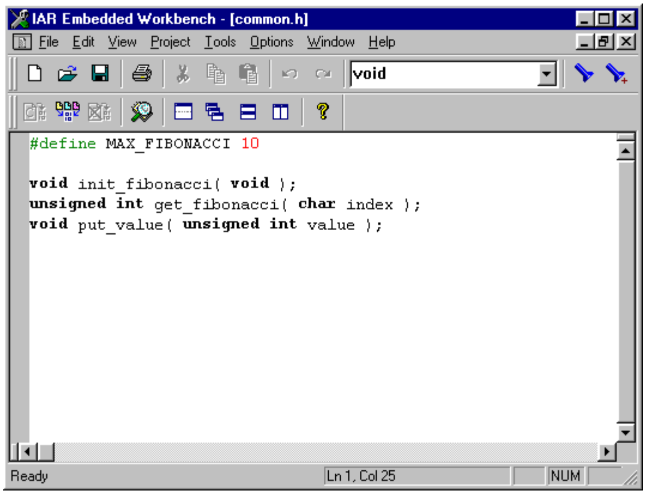


Figure 88: IAR Embedded Workbench edit bar

Toolbar search

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



```
#define MAX_FIBONACCI 10

void init_fibonacci( void );
unsigned int get_fibonacci( char index );
void put_value( unsigned int value );
```

Figure 89: Toolbar search

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

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

Project bar

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

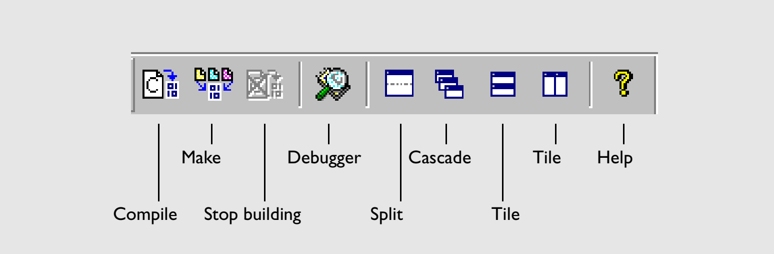


Figure 90: IAR Embedded Workbench project bar

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

PROJECT WINDOW

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

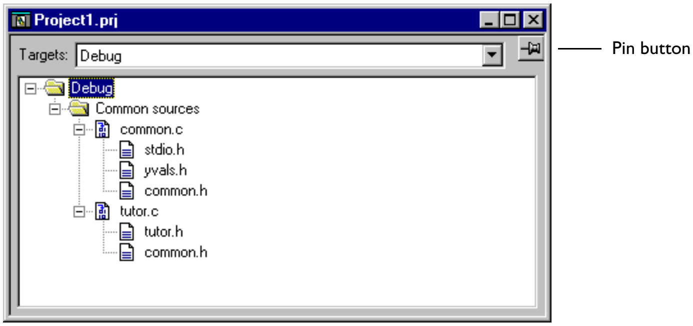


Figure 91: Project window

Pressing the right mouse button in the Project window displays a pop-up menu which gives you convenient access to several useful commands. **Save As Text...** allows you to save a description of the project, including all options that you have specified.

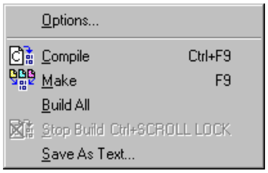


Figure 92: Project window pop-up menu

Pin button

The **Pin** button, in the top right corner of the Project window, allows you to pin the window to the desktop so that it is not affected by the **Tile** or **Cascade** commands on the **Window** menu.

Targets

The top node in the tree shows the current target. You can change the target by choosing a different target from the **Targets** drop-down list box at the top of the Project window. Each target corresponds to a different version of your project that you want to compile or assemble. For example, you might have a target called **Debug**, which includes debugging code, and one called **Release**, with the debugging code omitted.

You can expand the tree by double-clicking on the target icon, or by clicking on the plus sign icon, to display the groups included in this target.

Groups

Groups are used for collecting together related source files. Each group may be included in one or more targets, and a source file can be present in one or more groups.

Source files

You can expand each group by double-clicking on its icon, or by clicking on the plus sign icon, to show the list of source files it contains.

Once a project has been successfully built any include files are displayed in the structure below the source file that included them.

Note: The include files associated with a particular source file may depend on which target the source file appears in, since preprocessor or directory options may affect which include files are associated with a particular source file.

Editing a file

To edit a source or include file, double-click its icon in the Project window tree display.

Moving a source file between groups

You can move a source file between two groups by dragging its icon between the group icons in the Project window tree display.

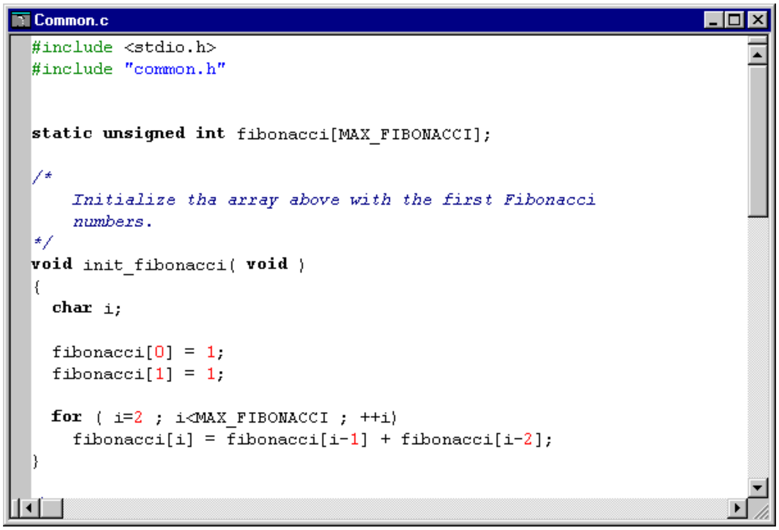
Removing items from a project

To remove an item from a project, click on it to select it, and then press Delete.

To remove a file from a project you can also use the **Project Files** dialog box, displayed by choosing **Files…** from the **Project** menu.

EDITOR WINDOW

Source files are displayed in the Editor window. The IAR Embedded Workbench editor automatically recognizes the syntax of C programs, and displays the different components of the program in different text styles.



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


static unsigned int fibonacci[MAX_FIBONACCI];

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

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

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

Figure 93: Editor window

The following table shows the default styles used for each component of a C program:

Item	Style
Default	Black plain
Keyword	Black bold
Strings	Blue
Preprocessor	Green
Integer (dec)	Red
Integer (oct)	Magenta
Integer (hex)	Magenta
Real	Blue
C++ comment: //	Dark blue italic
C comment: /*...*/	Dark blue italic

Table 26: Editor syntax coloring

To change these styles choose **Settings…** from the **Options** menu, and then select the **Colors and Fonts** page in the **Settings** dialog box, see *Colors and Fonts*, page 168.

Auto indent

The editor automatically indents a line to the same indent as the previous line, making it easy to lay out programs in a structured way.

Matching brackets

When the cursor is next to a bracket you can automatically find the matching bracket by choosing **Match Brackets** from the **Edit** menu.

Read-only and modification indicators

The name of the open source file is displayed in the Editor window title bar.

If a file is a read-only file, the text `(Read Only)` appears after the file name, for example `Common (Read Only)`.

When a file has been modified after it was last saved, an asterisk appears after the window title, for example `Common *`.

Editor options

The IAR Embedded Workbench editor provides a number of special features, each of which can be enabled or disabled independently in the **Editor** page of the **Settings** dialog box. For more information see *Settings…*, page 164.

Editor key summary

Use the following keys and key combinations for moving the insertion point:

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

Table 27: Editor keyboard commands for cursor navigation

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 28: 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

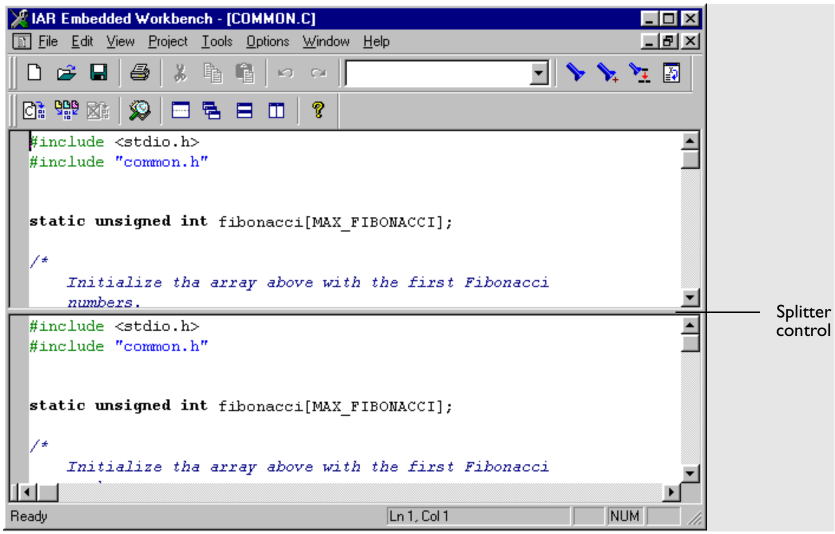
Table 29: Editor keyboard commands for selecting text

To select	Press
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 29: Editor keyboard commands for selecting text (continued)

Splitting the Editor window into panes

You can split the Editor window horizontally or vertically into multiple panes, to allow you to look at two different parts of the same source file at once, or cut and paste text between two different parts. To split the window drag the appropriate **splitter** control to the middle of the window:



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


static unsigned int fibonacci[MAX_FIBONACCI];

/*
    Initialize tha array above with the first Fibonacci
    numbers.
```

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


static unsigned int fibonacci[MAX_FIBONACCI];

/*
    Initialize tha array above with the first Fibonacci
```

Splitter control

Figure 94: Splitting the Editor window

To revert to a single pane double-click the appropriate splitter control, or drag it back to the end of the scroll bar. You can also split a window into panes using the **Split** command on the **Window** menu.

STATUS BAR

Displays the status information, and the state of the modifier keys.

As you are editing in the Editor window the status bar shows the current line and column number containing the cursor, and the Caps Lock, Num Lock, and Overwrite status:

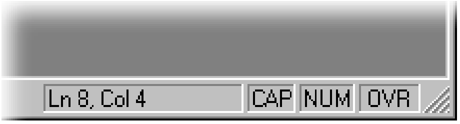
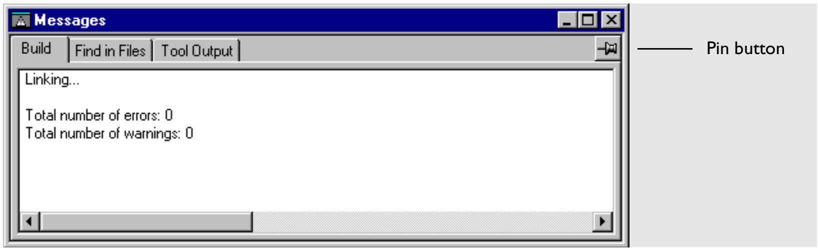


Figure 95: Editor window status bar

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

MESSAGES WINDOW

The Messages window shows the output from different IAR Embedded Workbench commands. The window is divided into multiple pages and you select the appropriate page by clicking on the corresponding tab.



```
Linking...

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

Figure 96: Messages window

Pressing the right mouse button in the Messages window displays a pop-up menu which allows you to save the contents of the window as a text file.

Figure 97: Save As... pop-up menu

To specify the level of output to the Messages window, select the **Make Control** page in the Settings window. See *Make Control*, page 169.

Pin button

The **Pin** button, in the top right corner of the Messages window, allows you to pin the window to the desktop so that it is not affected by the **Tile** or **Cascade** commands on the **Window** menu.

Build

Build shows the messages generated when building a project. Double-clicking a message in the Build panel opens the appropriate file for editing, with the cursor at the correct position.

Find in Files

Find in Files displays the output from the **Find in Files…** command on the **Edit** menu. Double-clicking an entry in the panel opens the appropriate file with the cursor positioned at the correct location.

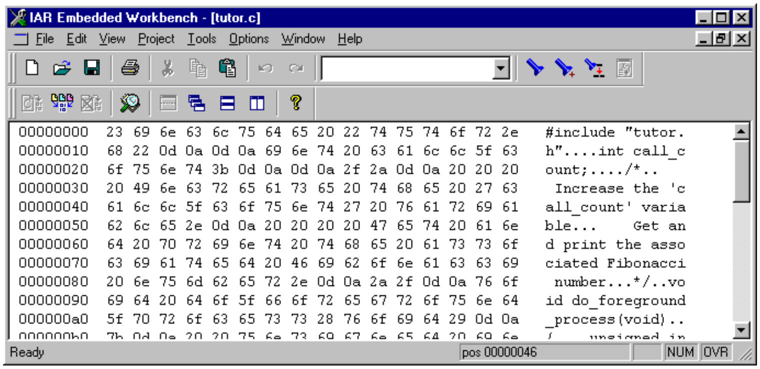
Tool Output

Tool Output displays any messages output by user-defined tools in the **Tools** menu.

BINARY EDITOR WINDOW

The Binary Editor window displays and allows you to edit the contents of a binary file. The data is displayed in hexadecimal format, with its **ASCII** equivalent to the right of each line. You can edit the contents by inserting or overwriting data.

To open the Binary Editor window, choose **Binary Editor…** from the **Tools** menu.



```
00000000  23 69 6e 63 6c 75 64 65 20 22 74 75 74 6f 72 2e   #include "tutor.
00000010  68 22 0d 0a 0d 0a 69 6e 74 20 63 61 6c 6c 5f 63   h"....int call_c
00000020  6f 75 6e 74 3b 0d 0a 0d 0a 2f 2a 0d 0a 20 20 20   ount;..../*..
00000030  20 49 6e 63 72 65 61 73 65 20 74 68 65 20 27 63    Increase the 'c
00000040  61 6c 6c 5f 63 6f 75 6e 74 27 20 76 61 72 69 61   all_count' varia
00000050  62 6c 65 2e 0d 0a 20 20 20 20 47 65 74 20 61 6e   ble...    Get an
00000060  64 20 70 72 69 6e 74 20 74 68 65 20 61 73 73 6f   d print the asso
00000070  63 69 61 74 65 64 20 46 69 62 6f 6e 61 63 63 69   ciated Fibonacci
00000080  20 6e 75 6d 62 65 72 2e 0d 0a 2a 2f 0d 0a 76 6f    number...*/..vo
00000090  69 64 20 64 6f 5f 66 6f 72 65 67 72 6f 75 6e 64   id do_foreground
000000a0  5f 70 72 6f 63 65 73 73 28 76 6f 69 64 29 0d 0a   _process(void)..
```

Figure 98: Binary Editor window

File menu

The **File** menu provides commands for opening projects and source files, saving and printing, and exiting from the IAR Embedded Workbench.

The menu also includes a numbered list of the most recently opened files to allow you to open one by selecting its name from the menu.

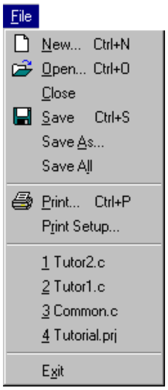


Figure 99: File menu

NEW...

Displays the following dialog box to allow you to specify whether you want to create a new project, or a new text file:

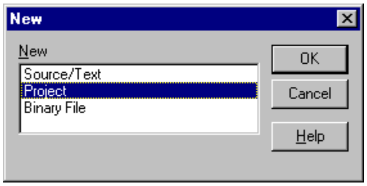
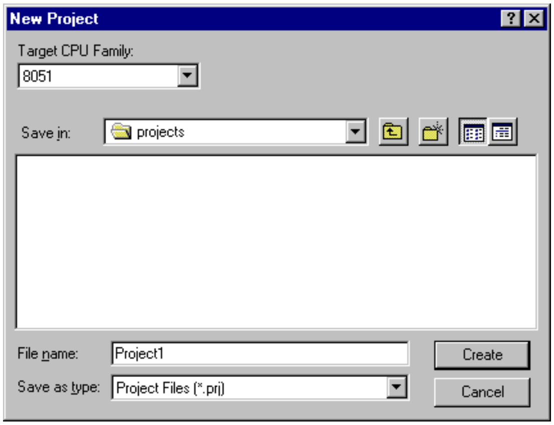


Figure 100: New dialog box

Choosing **Source/Text** opens a new Editor window to allow you to enter a text file.

Choosing **Project** displays the following dialog box to allow you to specify a name for the project and the target CPU family:



Target CPU Family: 8051

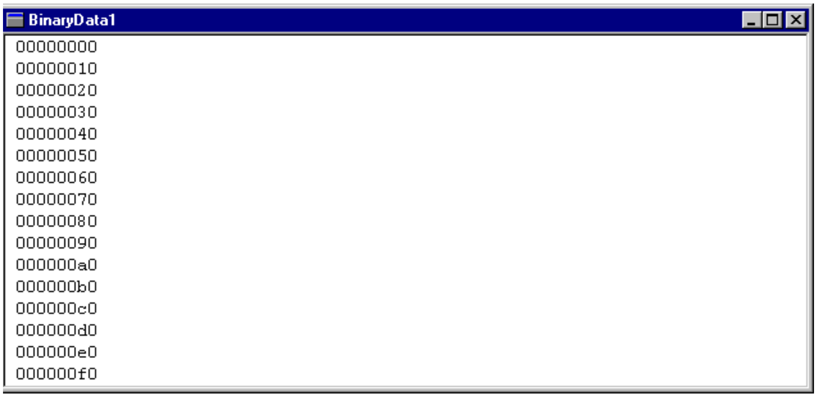
File name: Project1

Save as type: Project Files (*.prj)

Figure 101: New Project dialog box

The project will then be displayed in a new Project window. By default new projects are created with two targets, **Release** and **Debug**.

Selecting **Binary File** opens the Binary Editor window, allowing you to enter binary data:



```
00000000
00000010
00000020
00000030
00000040
00000050
00000060
00000070
00000080
00000090
000000a0
000000b0
000000c0
000000d0
000000e0
000000f0
```

Figure 102: Binary Editor window

Note: The Binary Editor starts in overwrite mode.

OPEN...

Displays a standard **Open** dialog box to allow you to select a text or project file to open. Opening a new project file automatically saves and closes any currently open project.

CLOSE

Closes the active window.

You will be warned if a text document has changed since it was last saved, and given the opportunity to save it before closing. Projects are saved automatically.

SAVE

Saves the current text or project document.

SAVE AS...

Displays the standard **Save As** dialog box to allow you to save the active document with a different name.

SAVE ALL

Saves all open text documents.

PRINT...

Displays the standard **Print** dialog box to allow you to print a text document.

PRINT SETUP...

Displays the standard **Print Setup** dialog box to allow you to set up the printer before printing.

EXIT

Exits from the IAR Embedded Workbench. You will be asked whether to save any changes to text windows before closing them. Changes to the project are saved automatically.

Edit menu

The **Edit** menu provides commands for editing and searching in Editor windows.

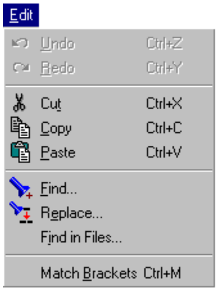


Figure 103: Edit menu

UNDO

Undoes the last edit made to the current Editor window.

REDO

Redoes the last **Undo** in the current Editor window.

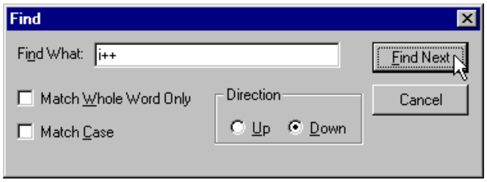
You can undo and redo an unlimited number of edits independently in each Editor window.

CUT, COPY, PASTE

Provide the standard Windows functions for editing text within Editor windows and dialog boxes.

FIND...

Displays the following dialog box to allow you to search for text within the current Editor window:


Find What: i++

☐ Match Whole Word Only

☐ Match Case

Direction: ○ Up ● Down

Figure 104: Find dialog box

Enter the text to search for in the **Find What** text box.

Select **Match Whole Word Only** to find the specified text only if it occurs as a separate word. Otherwise `int` will also find `print`, `sprintf` etc.

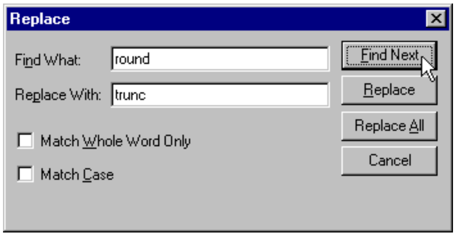
Select **Match Case** to find only occurrences that exactly match the case of the specified text. Otherwise specifying `int` will also find `INT` and `Int`.

Select **Up** or **Down** to specify the direction of the search.

Choose **Find Next** to find the next occurrence of the text you have specified.

REPLACE...

Allows you to search for a specified string and replace each occurrence with another string.



Find What: round

Replace With: trunc

☐ Match Whole Word Only

☐ Match Case

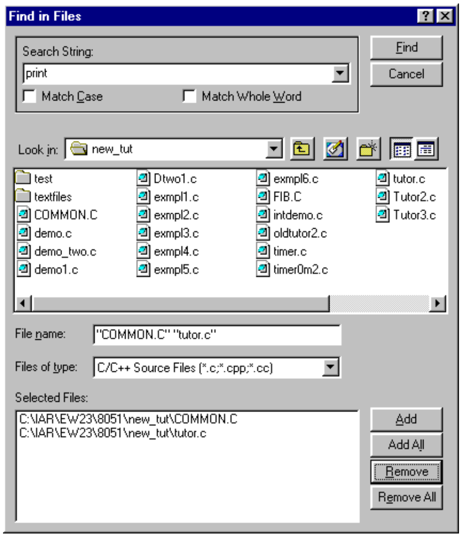
Figure 105: Replace dialog box

Enter the text to replace each found occurrence in the **Replace With** box. The other options are identical to those for **Find…**.

Choose **Find Next** to find the next occurrence, and **Replace** to replace it with the specified text. Alternatively choose **Replace All** to replace all occurrences in the current Editor window.

FIND IN FILES...

Allows you to search for a specified string in multiple text files. The following dialog box allows you to specify the criteria for the search:



Search String: print

☐ Match Case ☐ Match Whole Word

Look in: new_tut

File name: "COMMON.C" "tutor.c"

Files of type: C/C++ Source Files (*.c;*.cpp;*.cc)

Selected Files:
C:\IAR\EW23\8051\new_tut\COMMON.C
C:\IAR\EW23\8051\new_tut\tutor.c

Figure 106: Find in Files dialog box

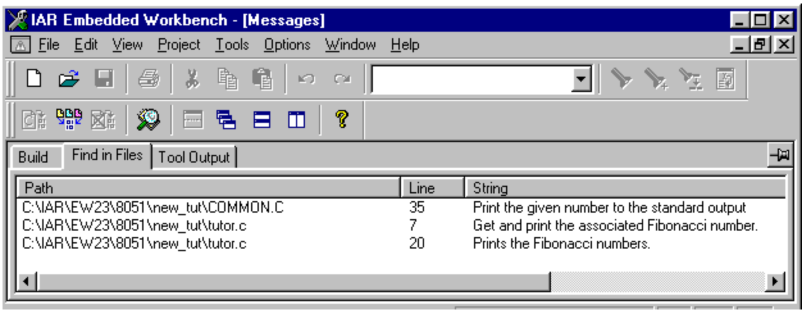
Specify the string you want to search for in the **Search String** text box, or select a string you have previously searched for from the drop-down list box.

Select **Match Whole Word** or **Match Case** to restrict the search to the occurrences that match as a whole word or match exactly in case, respectively.

Select each file you want to search in the **File Name** list, and choose **Add** to add it to the **Selected Files** list.

You can add all the files in the **File Name** list by choosing **Add All**, or you can select multiple files using the Shift and Ctrl keys and choose **Add** to add the files you have selected. Likewise you can remove files from the **Selected Files** list using the **Remove** and **Remove All** buttons.

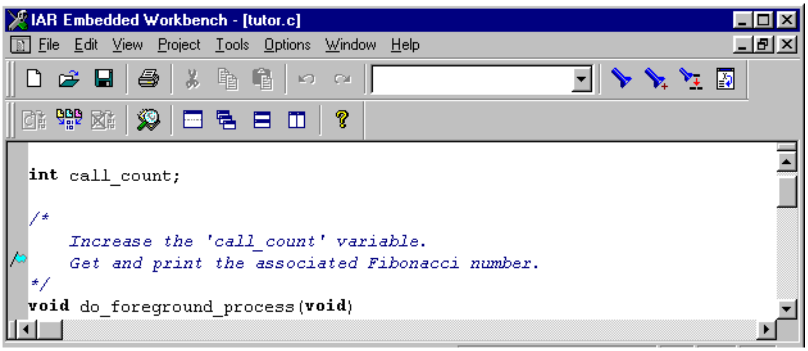
When you have selected the files you want to search choose **Find** to proceed with the search. All the matching occurrences are listed in the Messages window. You can then very simply edit each occurrence by double-clicking it:


IAR Embedded Workbench - [Messages]
File Edit View Project Tools Options Window Help
Build
Find in Files
Tool Output

Path	Line	String
C:\IAR\EW23\8051\new_tut\COMMON.C	35	Print the given number to the standard output
C:\IAR\EW23\8051\new_tut\tutor.c	7	Get and print the associated Fibonacci number.
C:\IAR\EW23\8051\new_tut\tutor.c	20	Prints the Fibonacci numbers.

Figure 107: Messages window displaying found strings

Double-clicking an item opens the corresponding file in an Editor window with the cursor positioned at the start of the line containing the specified text:


IAR Embedded Workbench - [tutor.c]
File Edit View Project Tools Options Window Help

```
int call_count;

/*
    Increase the 'call_count' variable.
    Get and print the associated Fibonacci number.
*/
void do_foreground_process(void)
```

Figure 108: Editor window displaying found string

MATCH BRACKETS

If the cursor is positioned next to a bracket this command moves the cursor to the matching bracket, or beeps if there is no matching bracket.

View menu

The commands on the **View** menu allow you to change the information displayed in the IAR Embedded Workbench window.

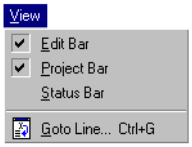


Figure 109: View menu

EDIT BAR

Toggles the edit bar on and off.

PROJECT BAR

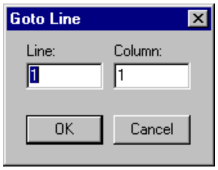
Toggles the project bar on and off.

STATUS BAR

Toggles the status bar on and off.

GOTO LINE...

Displays the following dialog box to allow you to move the cursor to a specified line and column in the current Editor window:



Line: 1

Column: 1

Figure 110: Goto Line dialog box

Project menu

The **Project** menu provides commands for adding files to a project, creating groups, specifying project options, and running the IAR Systems development tools on the current project.

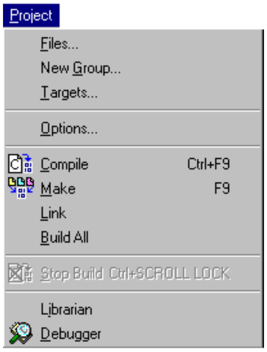
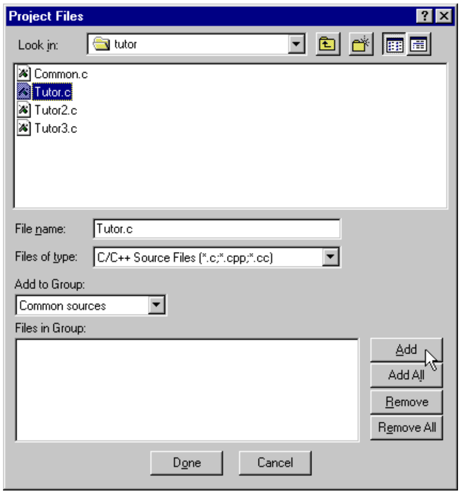


Figure 111: Project menu

FILES...

Displays the following dialog box to allow you to edit the contents of the current project:



Look in: tutor

File name: Tutor.c

Files of type: C/C++ Source Files (*.c;*.cpp;*.cc)

Add to Group: Common sources

Figure 112: Project Files dialog box

The **Add to Group** drop-down list box shows all the groups included in the current target. Select the one you want to edit, and the files currently in that group are displayed in the **Files in Group** list at the bottom of the dialog box.

The upper part of the **Project Files** dialog box is a standard file dialog box, to allow you to locate and select the files you want to add to each particular group.

Adding files to a group

To add files to the currently displayed group select them using the standard file controls in the upper half of the dialog box and choose the **Add** button, or choose **Add All** to add all the files in the **File Name** list box.

Removing files from a group

To remove files from the currently displayed group select them in the **Files in Group** list and choose **Remove**, or choose **Remove All** to remove all the files from the group.

You can use the **Project Files** dialog box to make changes to several groups. Choosing **Done** will then apply all the changes to the project. Alternatively, choosing **Cancel** will discard all the changes and leave the project unaffected.

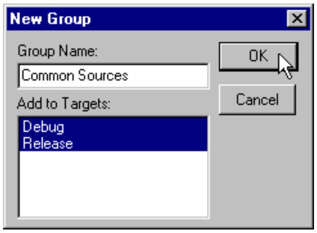
Source file paths

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

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

NEW GROUP...

Displays the following dialog box to allow you to create a new group:



Group Name: Common Sources

Add to Targets:
Debug
Release

Figure 113: New Group dialog box

Specify the name of the group you want to create in the **Group Name** text box. Select the targets to which you want to add the new group in the **Add to Targets** list. By default the group is added to all targets.

TARGETS...

Displays the following dialog box to allow you to create new targets, and display or change the groups included in each target:

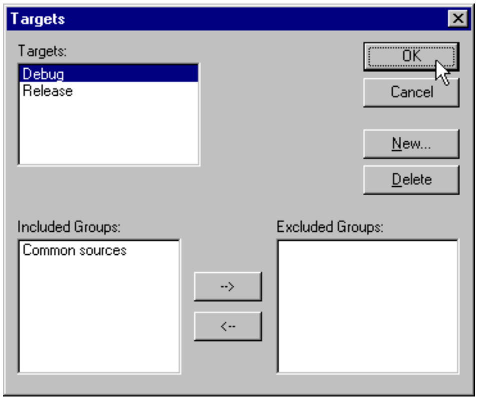


Figure 114: Targets dialog box

To create a new target, select **New...** and enter a name for the new target.

To delete a target, select it and click **Delete**.

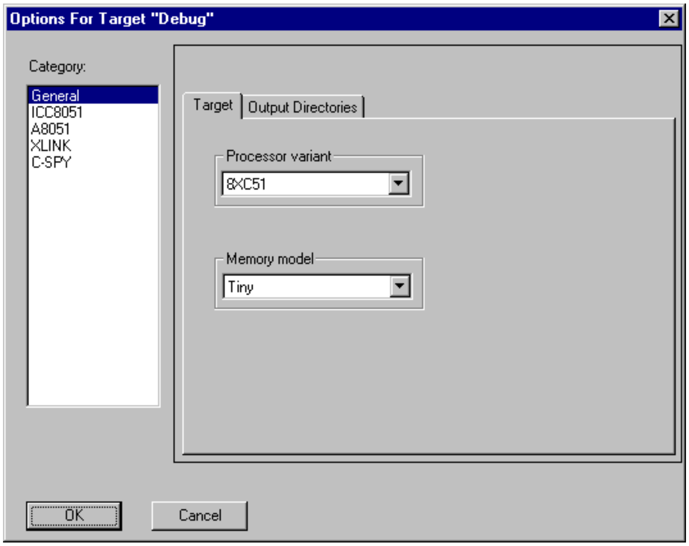
To view the groups included in a target select it in the **Targets** list.

The groups are shown in the **Included Groups** list, and you can add or remove groups using the arrow buttons.

OPTIONS...

Displays the **Options** dialog box to allow you to set directory and compiler options on the selected item in the Project window.

You can set options on the entire target, on a group of files, or on an individual file.



Processor variant: 8XC51

Memory model: Tiny

Figure 115: Options dialog box

The **Category** list allows you to select which set of options you want to modify. The options available in the **Category** list will depend on the tools installed in your IAR Embedded Workbench, and will typically include the following options:

Category	**Description**	**Refer to the chapter**
General	General options	*General options*
ICC8051	8051 Compiler options	*Compiler options*
A8051	8051 Assembler options	*Assembler options*
XLINK	IAR XLINK Linker™ options	*XLINK options*
C-SPY	IAR C-SPY® Debugger options	*C-SPY options*

Table 30: Option categories in IAR Embedded Workbench

Selecting a category displays one or more pages of options for that component of the IAR Embedded Workbench.

For more detailed information about the tools installed, see the *8051 IAR C Compiler Reference Guide*, the *8051 IAR Assembler Reference Guide* and the *IAR XLINK Linker™ and IAR XLIB Librarian™ Reference Guide*.

COMPILE

Compiles or assembles the currently active file or project as appropriate.

You can compile a file or project by selecting its icon in the Project window and choosing **Compile**. Alternatively, you can compile a file in the Editor window provided it is a member of the current target.

MAKE

Brings the current target up to date by compiling, assembling, and linking only the files that have changed since last build.

LINK

Explicitly relinks the current target.

BUILD ALL

Rebuilds and relinks all files in the current target.

STOP BUILD

Stops the current build operation.

LIBRARIAN

Starts the IAR XLIB Librarian™ to allow you to perform operations on library modules in library files.

DEBUGGER

Starts the IAR C-SPY Debugger so that you can debug the project object file.

You can specify the version of C-SPY to run in the **Debug** options for the target. If necessary a Make will be performed before running C-SPY to ensure that the project is up to date.

Tools menu

The **Tools** menu is a user-configurable menu to which you can add tools for use with the IAR Embedded Workbench.

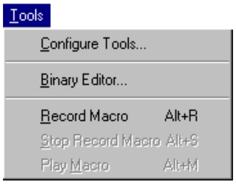
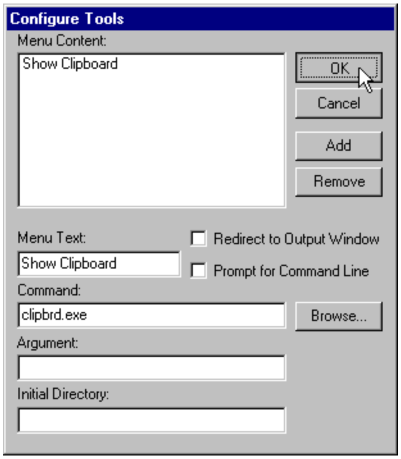


Figure 116: Tools menu

CONFIGURE TOOLS...

Configure Tools... displays the following dialog box to allow you to specify a user-defined tool to add to the menu:



Menu Content: Show Clipboard

Menu Text: Show Clipboard

☐ Redirect to Output Window

☐ Prompt for Command Line

Command: clipbrd.exe

Argument:

Initial Directory:

Figure 117: Configure Tools dialog box

Specify the text for the menu item in the **Menu Text** box, and the command to be run when you select the item in the **Command** text box. Alternatively, choose **Browse** to display a standard file dialog box to allow you to locate an executable file on disk and add its path to the **Command** text box.

Specify the argument for the command in the **Argument** text box, or select **Prompt for Command Line** to display a prompt for the command line argument when the command is selected from the **Tools** menu.

Variables can be used in the arguments, allowing you to set up useful tools such as interfacing to a command line revision control system, or running an external tool on the selected file.

The following argument variables can be used:

Variable	Description
`$CUR_DIR$`	Current directory
`$CUR_LINE$`	Current line
`$EW_DIR$`	Directory of the IAR Embedded Workbench, for example `c:\program files\iar systems\ew23`
`$EXE_DIR$`	Directory for executable output
`$FILE_DIR$`	Directory of active file, no file name
`$FILE_FNAME$`	File name of active file without path
`$FILE_PATH$`	Full path of active file (in Editor, Project, or Message window)
`$LIST_DIR$`	Directory for list output
`$OBJ_DIR$`	Directory for object output
`$PROJ_DIR$`	Project directory
`$PROJ_FNAME$`	Project file name without path
`$PROJ_PATH$`	Full path of project file
`$TARGET_DIR$`	Directory of primary output file
`$TARGET_FNAME$`	Filename without path of primary output file
`$TARGET_PATH$`	Full path of primary output file
`$TOOLKIT_DIR$`	Directory of the active product, for example `c:\program files\iar systems\ew23\8051`

Table 31: Argument variables

The **Initial Directory** text box allows you to specify an initial working directory for the tool.

Select **Redirect to Output Window** to display any console output from the tool in the Tools window.

Note: Tools that require user input or make special assumptions regarding the console that they execute in, will *not* work if you set this option.

When you have specified the command you want to add choose **Add** to add it to the **Menu Content** list. You can remove a command from the **Tools** menu by selecting it in this list and choosing **Remove**.

To confirm the changes you have made to the **Tools** menu and close the dialog box choose **OK**.

The menu items you have specified will then be displayed in the **Tools** menu:

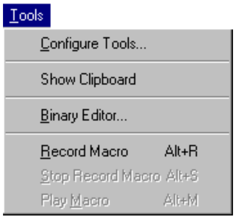


Figure 118: Customized Tools menu

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, and the command line command or batch file name in the **Argument** text box.

The command shells are specified as follows:

System	Command shell
Windows 95/98	`command.com`
Windows NT/2000	`cmd.exe` (recommended) or `command.com`

Table 32: Command shells

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

`/C` *`name`*

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

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

For example, to add the command **Backup** to the **Tools** menu to make a copy of the entire `project` directory to a network drive, you would specify **Command** as `command` and **Argument** as:

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

or

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

BINARY EDITOR…

Opens the Binary Editor window where you can edit a file in hexadecimal format. It displays a standard file dialog box allowing you to select a file. For more information, see *Binary Editor window*, page 147.

RECORD MACRO

Allows you to record a sequence of editor keystrokes as a macro.

STOP RECORD MACRO

Ends the recording of a macro.

PLAY MACRO

Replays the macro you have recorded.

Options menu

The **Settings…** command on the **Options** menu allows you to customize the IAR Embedded Workbench according to your own requirements.

Figure 119: Options menu (IAR Embedded Workbench)

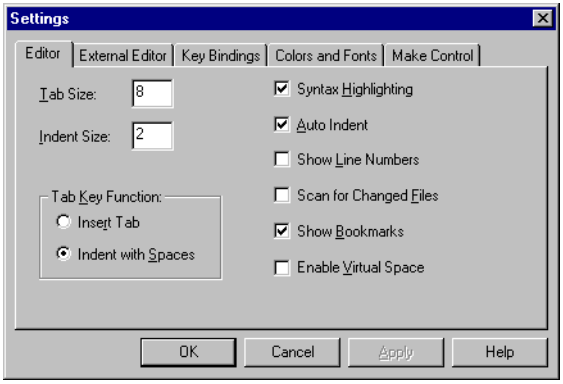
SETTINGS…

Displays the **Settings** dialog box to allow you to customize the IAR Embedded Workbench.

Select the feature you want to customize by clicking the **Editor**, **External Editor**, **Key Bindings**, **Colors and Fonts**, or **Make Control** tabs.

Editor

The **Editor** page allows you to change the editor options:



Tab Size: 8

Indent Size: 2

Tab Key Function:
○ Insert Tab
● Indent with Spaces

☑ Syntax Highlighting
☑ Auto Indent
☐ Show Line Numbers
☐ Scan for Changed Files
☑ Show Bookmarks
☐ Enable Virtual Space

Figure 120: Editor settings

It provides the following options:

Option	Description
Tab Size	Specifies the number of character spaces corresponding to each tab.
Indent Size	Specifies the number of character spaces to be used for indentation.
Tab Key Function	Specifies how the tab key is used.
Syntax Highlighting	Displays the syntax of C programs in different text styles.
Auto Indent	When you insert a line, the new line will automatically have the same indentation as the previous line.
Show Line Number	Displays line numbers in the Editor window.
Scan for Changed Files	The editor will check if files have been modified by some other tool and automatically reload them. If a file has been modified in the IAR Embedded Workbench, you will be prompted first.
Show Bookmarks	Displays compiler errors and Find in Files... search results.
Enable Virtual Space	Allows the cursor to move outside the text area.

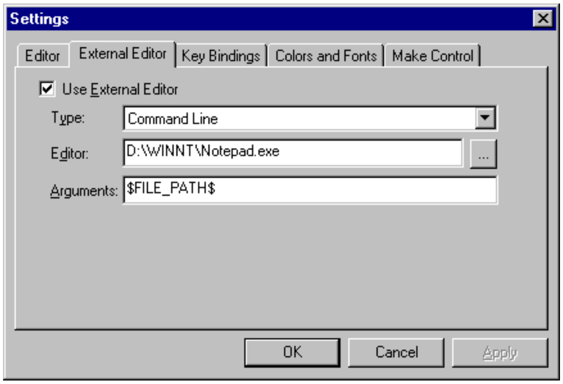
Table 33: Editor settings

For more information about the IAR Embedded Workbench Editor, see *Editor window*, page 142.

External Editor

The **External Editor** page allows you to specify an external editor.

An external editor can be called either by passing command line parameters or by using DDE (Windows Dynamic Data Exchange).



☑ Use External Editor

Type: Command Line

Editor: D:\WINNT\Notepad.exe

Arguments: $FILE_PATH$

Figure 121: Specifying external command-line editor

Selecting **Type: Command Line** will call the external editor by passing command line parameters. Provide the file name and path of your external editor in the **Editor** field. Then specify the command line to pass to the editor in the **Arguments** field.

Note: Variables can be used in arguments. See Table 31, *Argument variables*, page 162, for information about the argument variables that are available.

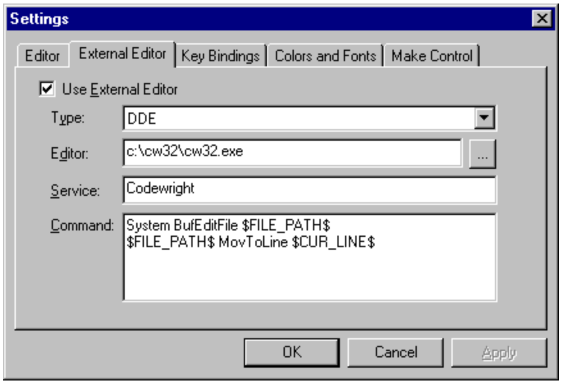
Selecting **Type: DDE** will call the external editor by using DDE. Provide the file name and path of your external editor in the **Editor** field.

Specify the DDE service name used by the editor in the **Service** field. Then specify a sequence of command strings to send to the editor in the **Command** field.

The command strings should be entered as:

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

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



☑ Use External Editor

Type: DDE

Editor: c:\cw32\cw32.exe

Service: Codewright

Command:

```
System BufEditFile $FILE_PATH$
$FILE_PATH$ MovToLine $CUR_LINE$
```

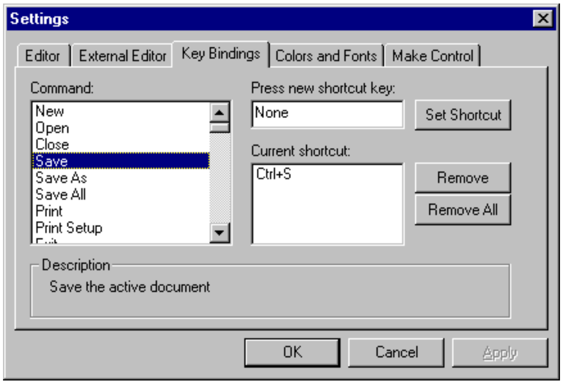
Figure 122: External editor DDE settings

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

Note: Variables can be used in the arguments. See Table 31, *Argument variables*, page 162, for more information about the argument variables that are available.

Key Bindings

The **Key Bindings** page displays the shortcut keys used for each of the menu options, and allows you to change them:



Command: Save

Press new shortcut key: None

Current shortcut: Ctrl+S

Description: Save the active document

Figure 123: Specifying key bindings

Select the command you want to edit in the **Command** list. Any currently defined shortcut keys are shown in the **Current shortcut** list.

To add a shortcut key to the command click in the **Press new shortcut key** box and type the key combination you want to use. Then click **Set Shortcut** to add it to the **Current shortcut** list. You will not be allowed to add it if it is already used by another command.

To remove a shortcut key select it in the **Current shortcut** list and click **Remove**, or click **Remove All** to remove all the command's shortcut keys.

Then choose **OK** to use the new key bindings you have defined and the menus will be updated to show the shortcuts you have defined.

You can set up more than one shortcut for a command, but only one will be displayed in the menu.

Colors and Fonts

The **Colors and Fonts** page allows you to specify the colors and fonts used for text in the Editor windows, and the font used for text in the other windows.

The panel shows a list of the C syntax elements you can customize in the Editor window:

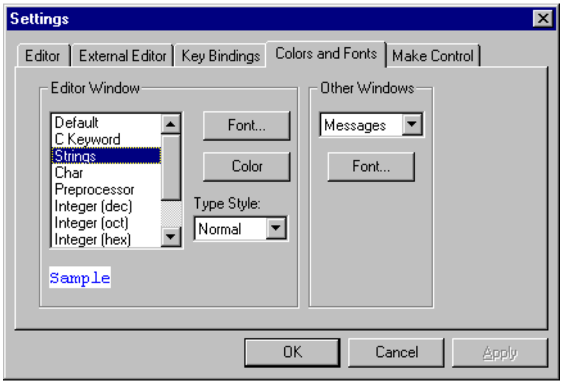


Figure 124: Specifying Editor window colors and fonts

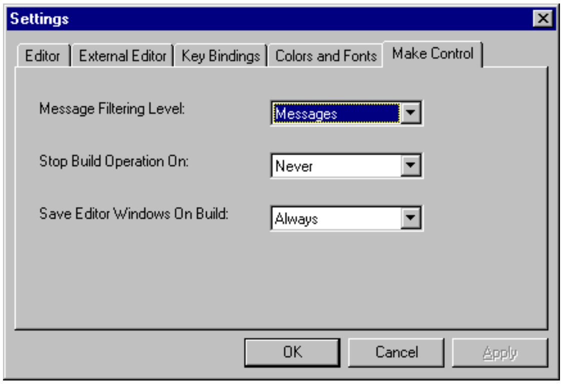
To specify the style used for each element of C syntax in the Editor window, select the item you want to define from the **Editor Window** list. The current setting is shown in the **Sample** box below the list box.

You can choose a text color by clicking **Color** and a font by clicking **Font...**. You can also choose the type style from the **Type Style** drop-down list.

Then choose **OK** to use the new styles you have defined, or **Cancel** to revert to the previous styles.

Make Control

The **Make Control** page allows you to set options for Make and Build:



Message Filtering Level: Messages

Stop Build Operation On: Never

Save Editor Windows On Build: Always

Figure 125: Make Control settings

The following table gives the options, and the alternative settings for each option:

Option	Setting
Message Filtering Level	All: Show all messages. Include compiler and linker information. Messages: Show messages, warnings, and errors. Warnings: Show warnings and errors. Errors: Show errors only.
Stop Build Operation On	Never: Do not Stop. Warnings: Stop on warnings and errors. Errors: Stop on errors.
Save Editor Windows On Build	Always: Always save before Make or Build. Ask: Prompt before saving. Never: Do not save.

Table 34: Make Control settings

Window menu

The commands on the **Window** menu allow you to manipulate the Workbench windows and change their arrangement on the screen.

The last section of the **Window** menu lists the windows currently open on the screen, and allows you to activate one by selecting it.

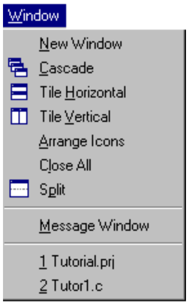


Figure 126: Window menu

NEW WINDOW

Opens a new window for the current file.

CASCADE, TILE HORIZONTAL, TILE VERTICAL

Provide the standard Windows functions for arranging the IAR Embedded Workbench windows on the screen.

ARRANGE ICONS

Arranges minimized window icons neatly at the bottom of the IAR Embedded Workbench window.

CLOSE ALL

Closes all open windows.

SPLIT

Allows you to split an Editor window horizontally into two panes to allow you to see two parts of a file simultaneously.

MESSAGE WINDOW

Opens the Messages window that displays messages and text output from the IAR Embedded Workbench commands.

Help menu

Provides help about the IAR Embedded Workbench.

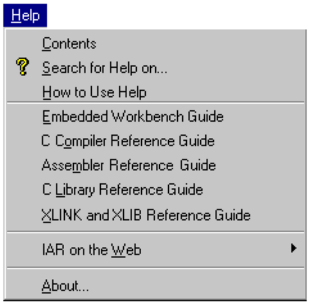

Help
Contents
Search for Help on...
How to Use Help
Embedded Workbench Guide
C Compiler Reference Guide
Assembler Reference Guide
C Library Reference Guide
XLINK and XLIB Reference Guide
IAR on the Web
About...

Figure 127: Help menu

CONTENTS

Displays the Contents page for help about the IAR Embedded Workbench.

SEARCH FOR HELP ON...

Allows you to search for help on a keyword.

HOW TO USE HELP

Displays help about using help.

EMBEDDED WORKBENCH GUIDE

Provides access to an online version of this user guide, available in Acrobat® Reader format.

C COMPILER REFERENCE GUIDE

Provides access to an online version of the *8051 IAR C Compiler Reference Guide*, available in Acrobat® Reader format.

ASSEMBLER GUIDE

Provides access to an online version of the *8051 IAR Assembler Reference Guide*, available in Acrobat® Reader format.

XLINK AND XLIB GUIDE

Provides access to a PDF version of the *IAR XLINK Linker™ and IAR XLIB Librarian™ Reference Guide*, available only in Acrobat® Reader format.

C LIBRARY REFERENCE GUIDE

Provides access to the C library documentation, which is available in Acrobat® Reader format.

IAR ON THE WEB

Allows you to browse the home page, news page, and FAQ (frequently asked questions) page of the IAR website, and to contact IAR Technical Support.

ABOUT...

Displays the version numbers of the user interface and of the 8051 IAR Embedded Workbench.

Part 4: The C-SPY simulator

This part of the 8051 IAR Embedded Workbench™ User Guide contains the following chapters:

- Introduction to C-SPY
- C-SPY expressions
- C-SPY macros
- Device description file
- C-SPY reference
- C-SPY command line options.

Introduction to C-SPY

The IAR C-SPY Debugger® is a powerful interactive debugger for embedded applications. This chapter gives an overview of the functions for debugging projects provided in this tool.

For information about the C-SPY options available in the IAR Embedded Workbench, see the chapter *C-SPY options* in *Part 3: The IAR Embedded Workbench*. For information about the available command line options, see the *C-SPY command line options* chapter.

Debugging projects

DISASSEMBLY AND SOURCE MODE DEBUGGING

C-SPY allows you to switch between C or assembler source and disassembly mode debugging as required.

Wherever source code is available, the source mode debugging displays the source program, and you can execute the program one statement at a time while monitoring the values of variables and data structures. Source mode debugging provides the quickest and easiest way of developing your application, without having to worry about how the compiler or assembler has implemented the code.

Disassembly mode debugging displays a mnemonic assembler listing of your program based on actual memory contents rather than source code, and lets you execute the program exactly one assembler instruction at a time. Disassembly mode debugging lets you focus on the critical sections of your application, and provides you with precise control over the simulated hardware.

During both source and disassembly mode debugging you can display the registers and memory, and change their contents.

If the file that you are debugging changes on the disk, you will be prompted to reload the file.

Source window

As you debug an application the source or disassembly source is displayed in a Source window, with the next source or disassembly statement to be executed highlighted.

You can navigate quickly to a particular file or function in the source code by selecting its name from the file or function box at the top of the Source window.

For convenience the Source window uses colors and text styles to identify key elements of the syntax. For example, by default C keywords are displayed in bold and constants in red. However, the colors and font styles are fully configurable, so that you can change them to whatever you find most convenient. See *Window Settings*, page 239 for additional information.

PROGRAM EXECUTION

C-SPY provides a flexible range of options for executing the target program.

The **Go** command continues execution from the current position until a breakpoint or program exit is reached. You can also execute up to a selected point in the program, without having to set a breakpoint, with the **Go to Cursor** command. Alternatively, you can execute out of a C function with the **Go Out** command.

Program execution is indicated by a flashing **Stop** command button in the debug toolbar. While the program is executing you may stop it either by clicking on the **Stop** button or by pressing the Escape key. You may also use any command accelerator associated with the **Stop** command.

Single stepping

The **Step** and **Step Into** commands allow you to execute the program a statement or instruction at a time. **Step Into** continues stepping inside function or subroutine calls whereas **Step** executes each function call in a single step.

The **Autostep** command steps repeatedly, and the **Multi Step** command lets you execute a specified number of steps before stopping.

Breakpoints

You can set breakpoints in the program being debugged using the **Toggle Breakpoint** command. Statements or instructions at which breakpoints are set are shown highlighted in the Source window listing.

Alternatively the **Edit Breakpoints** command allows you to define and alter complex breakpoints, including break conditions. You can optionally specify a macro which will perform certain actions when the breakpoint is encountered.

For detailed information, see *Edit Breakpoints…*, page 230.

Interrupt simulation

C-SPY includes an interrupt system allowing you to optionally simulate the execution of interrupts when debugging with C-SPY. The interrupt system can be turned on or off as required either with a system macro or by using the **Interrupt** dialog box and loading it into the simulator. The interrupt system is activated by default, but if it is not required, it can be turned off to speed up instruction set simulation.

For the latest information about how to set up the interrupt simulation, see the `cs8051.htm` file.

The interrupt system has the following features:

- Interrupts, single or periodical, can be set up so that they are generated based on the cycle counter value.
- C-SPY provides interrupt support suitable for the 8051 processor variants.
- By combining an interrupt with a data breakpoint, you can simulate peripheral devices, such as a serial port.

Note: The C-SPY interrupt system uses the cycle counter as a clock to determine when an interrupt will be raised in the simulator. If you change the system cycle counter, it affects your interrupt orders. When changing the cycle counter, you have two alternatives:

1. Save the old cycle counter value in a macro variable to reconstruct your interrupt orders and cancel the current waiting interrupt orders. This allows you to reconstruct the situation as the old cycle counter is saved.
2. Change the cycle counter without saving the old value. Your previously ordered interrupts are not modified according to the new cycle counter value—they remain as before. This approach does not allow you to reconstruct the situation.

Performing a C-SPY reset will reset the cycle counter. If there are repeatable orders, they remain but with an adjusted event time which is their repeat value. All other interrupt orders are cleared.

Example

The cycle counter is 123456.

A repeatable order that raises an interrupt every 4000 cycles is ordered.

A single order is about to raise an order at 123500 cycles.

After a system reset the repeatable interrupt order remains and will raise an interrupt every 4000 cycles with the first interrupt at 4000 cycles. The single order is removed.

Interrupts are ordered by giving a vector. You should be familiar with the `8XC51` derivatives interrupt system. To let the simulator generate an interrupt, specify the interrupt vector. This is done with the vector name defined in the Device Description file.

C function information

C-SPY keeps track of the active functions and their local variables, and a list of the function calls can be displayed in the Calls window. You can also trace functions during program execution using the **Trace** command and tracing information is displayed in the Report window. For additional information, see *Trace*, page 238.

You can use the **Quick Watch** command to examine the value of any local, global, or static variable that is in scope. You can monitor the value of a macro, variable, or expression in the Watch window as you step through the program.

Viewing and editing memory and registers

You can display the contents of the processor registers in the Register window, and specified areas of memory in the Memory window.

The Register window allows you to edit the content of any register, and the register is automatically updated to reflect the change.

The Memory window can display the contents of memory in groups of 8, 16, or 32 bits, and you can double-click any memory address to edit the contents of memory at that address.

Terminal I/O

C-SPY can simulate terminal input and output using the Terminal I/O window.

Macro language

C-SPY includes a powerful internal macro language, to allow you to define complex sets of actions to be performed; for example, calculating the stack depth or when breakpoints are encountered. The macro language includes conditional and loop constructs, and you can use variables and expressions.

Tutorial 3, page 50, shows how C-SPY macros can be used.

Profiling

The profiling tool provides you with timing information on your application. This is useful for identifying the most time-consuming parts of the code and optimizing your program.

Code coverage

The code coverage tool can be used for identifying unused code in an application, as well as providing you with code coverage status at different stages during execution.

When displaying source at assembler level, every assembler instruction that has been executed is marked with an `*` (asterisk). This information is updated when disassembled source is read into the C-SPY source buffer. This means that an assemble statement may not be marked with `*` immediately after it has been executed.

C-SPY expressions

In addition to the C symbols defined in your program, C-SPY® allows you to define C-SPY variables and macros and use them when evaluating expressions. Expressions that are built with these components are called C-SPY expressions and can be used in the Watch and QuickWatch windows and in C-SPY macros.

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

Expression syntax

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

- C symbols
- Assembler symbols—that is, CPU register names and assembler labels
- C-SPY variables and C-SPY macros; see the chapter *C-SPY macros*.

C SYMBOLS

C symbols can be referenced by their names or using an extended C-SPY format which allows you to reference symbols outside the current scope.

Expression	What it means
`i`	C variable `i` in the current scope or C-SPY variable `i`.
`\i`	C variable `i` in the current function.
`\func\i`	C variable `i` in the function `func`.
`mod\func\i`	C variable `i` in the function `func` in the module `mod`.

Table 35: C-SPY C symbols expressions

Note: When using the module name to reference a C symbol, the module name must be a valid C identifier or it must be encapsulated in backquotes ` (ASCII character 0x60), for example:

```
nice_module_name\func\i
`very strange () module + - name`\func\i
```

In case of a name conflict, C-SPY variables have a higher precedence than C variables. Extended C-SPY format can be used for solving such ambiguities.

Examples of valid C-SPY expressions are:

```
i = my_var * my_mac() + #asm_label
another_mac(2, my_var)
mac_var = another_module\another_func\my_var
```

ASSEMBLER SYMBOLS

Assembler symbols can be used in C expressions if they are preceded by `#`. These symbols can be assembler labels or CPU register names.

Example	What it does
`#pc++`	Increments the value of the program counter.
`myptr = #main`	Sets `myptr` to point to label `main`.

Table 36: 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 encapsulate the label in backquotes ` (ASCII character 0x60). For example:

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

Table 37: Handling name conflicts between h/w registers and assembler labels

FORMAT SPECIFIERS

The following format specifiers can be used in the **Display Format** drop-down list in the **Symbol Properties** dialog box (see *Inspecting expression properties*, page 219) and in a macro `message` statement:

Macro message specifier	Description
`%b`	Binary format
`%c`	Char format
%d	Signed decimal format
%f	Float format `[-]ddd.ddd`
`%o`	Unsigned octal format
%p	Pointer format
`%s`	String format
%u	Unsigned decimal format
`%x`	Unsigned hexadecimal format
`%X`	Unsigned hexadecimal format (capital letters)

Table 38: C-SPY expressions format specifiers

The precision for the default float format is seven or 15 decimals for four and eight byte floats.

Strings with format `%s` are printed in quotation marks. If no `NULL` character (`'\0'`) is found within 1000 characters, the printout will stop without a final quotation mark.

C-SPY macros

The IAR C-SPY® Debugger provides comprehensive macro capabilities allowing you to automate the debugging process and to simulate peripheral devices. Macros can be used in conjunction with complex breakpoints and interrupt simulation to perform a wide variety of tasks.

This chapter deals with how to use and set up C-SPY macros. At the end of the chapter, there is comprehensive reference information for each built-in system macro provided with C-SPY. Tutorial 3 in the *Compiler tutorials* chapter gives an example of how the C-SPY macros can be used.

Using C-SPY macros

C-SPY allows you to define both *macro variables* (global or local) and *macro functions*. In addition, several predefined system macro variables and macro functions are provided; they return information about the system status, and perform complex tasks such as opening and closing files, and file I/O operations. System macro names start with double underscore and are reserved names.

Note: To view the available macros, select **Load macro...** from the **Options** menu. The available macros will be displayed in the **Macro Files** dialog box, under **Registered Macros**. You can select whether to view all macros, the predefined system macros, or the user-defined macros; see also *Load Macro…*, page 243.

Defining macros

To define a macro variable or macro function, you should first create a text file containing its definition. You can use any suitable text editor, such as the IAR Embedded Workbench™ Editor. Then you should register the macro file. There are several ways to do this:

- You can register a macro by choosing **Load Macro…** from the **Options** menu. For more information, see *Load Macro…*, page 243.
- In the IAR Embedded Workbench, you can specify which setup file to use with a project. See *Setting C-SPY options*, page 131, for more information.
- When starting C-SPY with the Windows **Run...** command, you can use the `-f` command line option to specify the setup file. See *C-SPY command line options*, page 247, for more information.
- Macros can also be registered using the system macro `__registerMacroFile`. This macro allows you to register macro files from other macros. This means that you can dynamically select which macro files to register, depending on the run-time conditions. For more information, see *__registerMacroFile*, page 201.

Executing C-SPY macros

You can assign values to a macro variable, or execute a macro function, using the **Quick Watch…** command on the **Control** menu, or from within another C-SPY macro including setup macros. For details of the setup macros, see *C-SPY setup macros*, page 189.

A macro can also be executed if it is associated with a breakpoint that is activated.

MACRO VARIABLES

A macro variable is a variable defined and allocated outside the user program space. It can then be used in a C-SPY expression.

The command to define one or more macro variables has the following form:

```
var nameList;
```

where *`nameList`* is a list of C-SPY variable names separated by commas.

A macro variable defined outside a macro body has global scope, and retains its value and type through the whole debugging session. A macro variable defined within a macro body is created when its definition is executed and deallocated on return from the macro.

By default a macro variable is initialized to signed integer 0. When a C-SPY variable is assigned a value in an expression its type is also converted to the type of the operand. For example:

Expression	**What it means**
`myvar = 3.5`	`myvar` is now type `float`, value `3.5`.
`myvar = (int*)i`	`myvar` is now type pointer to `int`, and the value is the same as `i`.

Table 39: Examples of C-SPY macro variables

A complex type (`struct` or `union`) cannot be assigned to a macro variable but a macro variable can contain an address to such an object.

MACRO FUNCTIONS

C-SPY macro functions consist of a series of C-SPY variable definitions and macro statements which are executed when the macro is called. An unlimited number of parameters can be passed to a macro, and macros can return a value on exit.

A C-SPY macro has the following form:

```
macroName (parameterList)
{
  macroBody
}
```

where `parameterList` is a list of macro formal names separated by commas, and `macroBody` is any series of C-SPY variable definitions and C-SPY statements.

Type checking is not performed on the values passed to the macro parameters. When an `array`, `struct`, or `union` is passed, only its address is passed.

MACRO STATEMENTS

The following C-SPY macro statements are accepted:

Expressions

```
expression;
```

Conditional statements

```
if (expression)
  statement
```

```
if (expression)
  statement
else
  statement
```

Loop statements

```
for (init_expression; cond_expression; after_expression)
  statement
```

```
while (expression)
  statement
```

```
do
  statement
while (expression);
```

Return statements

```
return;
```

```
return (expression);
```

If the return value is not explicitly set by default, `signed int 0` is returned.

Blocks

```
{
  statement1
  statement2
  .
  .
  .
  statementN
}
```

In the above example, *`expression`* means a C-SPY expression; statements are expected to behave in the same way as corresponding C statements would do.

Printing messages

The `message` statement allows you to print messages while executing a macro. Its definition is as follows:

```
message argList;
```

where *`argList`* is a list of C-SPY expressions or strings separated by commas. The value of expression arguments or strings are printed to the Report window.

It is possible to override the default display format of an element in *`argList`* by suffixing it with a `:` followed by a format specifier, for example:

```
message int1:%X, int2;
```

This will print `int1` in hexadecimal format and `int2` in default format (decimal for an integer type).

Displaying contents of complex objects

To display the contents of complex objects (structures, unions and arrays) when executing a quick expression, watch expression or message macro command, the expression should be prefixed with the `@` character.

Example

The message `"My structure: ", my_struct, "\n";`

will print the address of the structure `my_struct` while

```
message "My structure: ", @my_struct, "\n";
```

will print a list of all structure members and their values.

Resume statement

The `resume` statement allows you to resume execution of a program after a breakpoint is encountered. For example, specifying:

```
resume;
```

in a breakpoint macro will resume execution after the breakpoint.

Error handling in macros

Two types of errors can occur while a macro is being executed:

- Stop errors, which stop execution. Stop errors are caused by mismatched macro parameter types, missing parameters, illegal addresses when setting a breakpoint or map, or illegal interrupt vectors when setting up an interrupt. They are handled by the C-SPY error handler, and execution stops with an appropriate error message.
- Minor errors, which cause the macro to return an error number. Minor errors are caused by actions such as failing to open a file, or cancelling a non-existing interrupt. You can test for minor errors by checking the value returned by the system macro; zero indicates successful execution, any other value is a C-SPY error number.

C-SPY setup macros

The setup macros are reserved macro names that will be called by C-SPY at specific stages during execution. To use them you should create and register a macro with the name specified in the following table:

Macro	Description
`execUserExit()`	Called each time the program is about to exit. Implement this macro to save status data, etc.
`execUserInit()`	Called before communication with the target system is established. If you have not already chosen the processor option using the IAR Embedded Workbench or the C-SPY command line option, you can use this macro. It can also be used for performing other initialization, for example, port initialization for the emulator and ROM-monitor variants. Notice that since there is still no code loaded, you cannot, for example, set a breakpoint from this macro.
`execUserPreload()`	Called after communication with the target system is established but before downloading the target program. Implement this macro to initialize memory locations and/or registers which are vital for loading data properly.

Table 40: C-SPY setup macros

Macro	Description
`execUserReset()`	Called each time the reset command is issued. Implement this macro to set up and restore data.
`execUserSetup()`	Called once after the target program is downloaded. Implement this macro to set up the memory map, breakpoints, interrupts, register macro files, etc.
`execUserTrace()`	Called each time C-SPY issues a trace printout (when the `Trace` command is active).

Table 40: C-SPY setup macros (continued)

Descriptions of system macros

The following sections provide reference information for each of the C-SPY system macros.

`__autoStep` `__autoStep(`*`delay`*`)`

Parameters

`delay`	Delay between steps in tenth of a second (integer)

Return value

`int 0`

Description

Steps continuously, with selectable time delay, until a breakpoint or the program exit is detected. For additional information, see *Autostep...*, page 228.

Example

```
__autoStep(12);
```

`__calls` `__calls(`*`mode`*`)`

Parameters

`mode`	Predefined string, one of: "`ON`" turns calls mode on "`OFF`" turns calls mode off

Return value

`int 0`

Description

Toggles calls mode on or off. For additional information, see *Calls window*, page 217.

Example

```
__calls("ON");
```

`__cancelAllInterrupts` `__cancelAllInterrupts()`

Return value

`int 0`

Description

Cancels all ordered interrupts. For additional information, see *Interrupt…*, page 236.

Example

```
__cancelAllInterrupts();
```

`__cancelInterrupt` `__cancelInterrupt(`*`interrupt_id`*`)`

Parameters

`interrupt_id`	The value returned by the corresponding `__orderInterrupt` macro call (unsigned long)

Return value

Result	Value
Successful	`int 0`
Unsuccessful	Non-zero error number

Table 41: __cancelInterrupt return values

Description

Cancels an interrupt. For additional information, see *Interrupt…*, page 236.

Example

`__cancelInterrupt(`*`interrupt_id`*`)`

`__clearAllBreaks` `__clearAllBreaks()`

Return value

`int 0`

Description

Clears all user-defined breakpoints. For additional information, see *Edit Breakpoints…*, page 230.

Example

`__clearAllBreaks();`

`__clearAllMaps` `__clearAllMaps()`

Return value

`int 0`

Description

Clears all user-defined memory mappings. For additional information, see *Memory Map…*, page 233.

Example

`__clearAllMaps();`

`__clearBreak` `__clearBreak(`*`address, segment, access`*`)`

Parameters

`address`	The breakpoint location (string)
`segment`	The memory segment name (string), one of: `CODE`, `IDATA`, `XDATA`, or `SFR`

access	The memory access type (string); concatenation of any of “`R`”, “`W`”, “`F`”, “`I`”, or “`O`”:

`R`	Read
`W`	Write
`F`	Fetch
`I`	Read immediate
`O`	Write immediate

Return value

Result	Value
Successful	`int 0`
Unsuccessful	Non-zero error number

Table 42: __clearBreak return values

Description

Clears a given breakpoint. For additional information, see *Edit Breakpoints…*, page 230.

Example

The following example shows how a line in the source file is specified as *address*:

```
__clearBreak(".demo\\12", "CODE", "F");
```

The following example shows how *address* is specified in hexadecimal notation:

```
__clearBreak("0x1300", "CODE", "F");
```

`__clearMap` `__clearMap(`*`address, segment`*`)`

Parameters

address	The address (integer)
segment	The memory segment name (string), one of: `CODE`, `IDATA`, `XDATA`, or `SFR`

Return value

Result	Value
Successful	`int 0`
Unsuccessful	Non-zero error number

Table 43: __clearMap return values

Description

Clears a given memory mapping. For additional information, see *Memory Map…*, page 233.

Example

```
__clearMap(1234, "CODE");
```

`__closeFile` `__closeFile(filehandle)`

Parameters

`filehandle`	The macro variable used as filehandle by the `__openFile` macro

Return value

`int 0.`

Description

Closes a file previously opened by `__openFile`.

Example

```
__closeFile(filehandle);
```

`__disableInterrupts` `__disableInterrupts()`

Return value

Result	Value
Successful	`int 0`
Unsuccessful	Non-zero error number

Table 44: __disableInterrupts return values

Description

Disables the generation of interrupts. For additional information, see *Interrupt…*, page 236.

Example

```
__disableInterrupts();
```

`__enableInterrupts` `__enableInterrupts()`

Return value

Result	Value
Successful	`int 0`
Unsuccessful	Non-zero error number

Table 45: __enableInterrupts return values

Description

Enables the generation of interrupts. For additional information, see *Interrupt…*, page 236.

Example

```
__enableInterrupts();
```

`__getLastMacroError` `__getLastMacroError()`

Return value

Value of the last system macro error code.

Description

Returns the last macro error code (excluding stop errors).

Example

```
__getLastMacroError();
```

`__go` `__go()`

Return value

`int 0`

Description

Starts execution. For additional information, see *Go*, page 229.

Example

```
__go();
```

`__multiStep` `__multiStep(`*`kindOf`*`, `*`noOfSteps`*`)`

Parameters

`kindOf`	Predefined string, one of:
	`"OVER"` does not enter C functions or assembler subroutines
	`"INTO"` enters C functions or assembler subroutines
`noOfSteps`	Number of steps to execute (integer)

Return value

`int 0`

Description

Executes a sequence of steps. For additional information, see *Multi Step…*, page 228.

Example

```
__multiStep("INTO", 12);
```

`__openFile` `__openFile(`*`filehandle, filename, access`*`)`

Parameters

`filehandle`	The macro variable to contain the file handle
`filename`	The filename as a string

`access`	The access type (string); one of the following: `"r"` ASCII read `"rb"` Binary read `"w"` ASCII write `"wb"` Binary write

Return value

Result	Value
Successful	`int 0`
Unsuccessful	Non-zero error number

Table 46: __openFile return values

Description

Opens a file for I/O operations.

Example

```
var filehandle;
__openFile(filehandle, "C:\\TESTDIR\\TEST.TST", "r");
```

`__orderInterrupt`

```
__orderInterrupt(address, activation_time, repeat_interval,
      jitter, latency, probability)
```

Parameters

`address`	The interrupt vector (string)
`activation_time`	The activation time in cycles (integer)
`repeat_interval`	The periodicity in cycles (integer)
`jitter`	The timing variation range (integer between 0 and 100)
`latency`	The latency (integer)
`probability`	The probability in percent (integer between 0 and 100)

Return value

The macro returns an interrupt identifier (`unsigned long`).

Description

Generates an interrupt. For additional information, see *Interrupt…*, page 236.

Example

```
__orderInterrupt ("0x03", 5000, 1500, 50, 0, 75);
```

`__printLastMacroError` `__printLastMacroError()`

Return value

`int 0`

Description

Prints the last system macro error message (excluding stop errors) to the Report window.

Example

```
__printLastMacroError();
```

`__processorOption` `__processorOption(`*`procOption`*`)`

Parameters

`procOption`	The processor option given in the same way it would have been given on the command line (string)

Return value

`int 0`

Description

Sets a given processor option. This macro can only be called from the `execUserInit()` macro. For additional information, see *-v*, page 251.

Example

```
__processorOption("-v1");
```

`__readFile` `__readFile(`*`filehandle`*`)`

Parameters

`filehandle`	The macro variable used as the filehandle by the `__openFile` macro

Return value

The return value depends on the access type of the file.

In ASCII mode a series of hex digits, delimited by space, are read and converted to an `unsigned long`, which is returned by the macro.

In binary mode one byte is read and returned.

Description

Reads from a file.

When the end of the file is reached, the file is rewound and a message is printed in the Report window. For additional information, see *Report window*, page 220.

Example

Assuming a file was opened with the `r` access type containing the following data:

```
1234 56 78
```

calls to `__readFile()` would return the numeric values `0x1234`, `0x56`, and `0x78`.

`__readFileGuarded` `__readFileGuarded(`*`filehandle, errorstatus`*`)`

Parameters

`filehandle`	The macro variable used as the file handle by the `__openFile` macro
`errorstatus`	A C-SPY variable to contain the error status

Return value

Result	**Value**
Successful	The value read
Unsuccessful	`-1L`

Table 47: __readFileGuarded return values

Description

Reads from a file. This macro works in exactly the same way as `__readFile`, except that when the end of the file is encountered `-1L` is returned, and the value of *`errorstatus`* is set to the corresponding error number.

Example

`__readFileGuarded(`*`filehandle, errorstatus`*`)`

`__readMemoryByte` `__readMemoryByte(`*`address, segment`*`)`

Parameters

`address`	The memory address (integer)
`segment`	The memory segment name (string), one of: `CODE`, `IDATA`, `XDATA`, or `SFR`

Return value

The macro returns the value from memory.

Description

Reads one byte from a given memory location.

Example

```
__readMemoryByte(0x2000, "CODE");
```

`__realtime` `__realtime(`*`what`*`)`

Parameters

`what`	Predefined string, one of: `"ON"` turns real-time mode on `"OFF"` turns real-time mode off

Return value

`int 0`

Description

Toggles real-time mode on or off. Notice that real-time mode only applies to the emulator and ROM-monitor versions of C-SPY. For additional information, see *Realtime*, page 238.

Example

```
__realtime("OFF");
```

`__registerMacroFile` `__registerMacroFile(`*`filename`*`)`

Parameters

`filename`	A file containing the macros to be registered (string)

Return value

`int 0`

Description

Registers macros from a specified macro file. For additional information, see *Load Macro…*, page 243.

Example

```
__registerMacroFile("c://testdir//macro.mac");
```

`__reset` `__reset()`

Return value

`int 0`

Description

Resets the target processor. For additional information, see *Reset*, page 229.

Example

```
__reset();
```

`__rewindFile` `__rewindFile(`*`filehandle`*`)`

Parameters

`filehandle`	The macro variable used as filehandle by the `__openFile` macro

Return value

`int 0`

Description

Rewinds the file previously opened by `__openFile`.

Example

```
__rewindFile(filehandle);
```

`__setBreak` `__setBreak(`*`address, segment, length, count, condition, cond_type, access, macro`*`)`

Parameters

`address`	The address in memory or any expression that evaluates to a valid address, for example a function or variable name. A `.` (period) must precede a code breakpoint (string).
`segment`	The memory segment name (string), one of: `CODE`, `IDATA`, `XDATA`, or `SFR`
`length`	The number of bytes to be covered by the breakpoint (integer)
`count`	The number of times that a breakpoint condition must be fulfilled before a break occurs (integer)
`condition`	The breakpoint condition (string)
`cond_type`	The condition type; either “`CHANGED`” or “`TRUE`” (string)

access	The memory access type (string); concatenation of any of “`R`”, “`W`”, “`F`”, “`I`”, or “`O`”.	
	`R`	Read
	`W`	Write
	`F`	Fetch
	`I`	Read immediate
	`O`	Write immediate
macro	The expression to be executed after the breakpoint is accepted (string)	

Return value

Result	Value
Successful	`int 0`
Unsuccessful	Non-zero error number

Table 48: __setBreak return values

Description

Sets a given breakpoint.

Examples

The following example shows a *code* breakpoint:

```
__setBreak(".demo\\12", "CODE", 1, 3, "d>16", "TRUE", "RF",
"afterMacro ()");
```

The following example shows a *data* breakpoint:

```
__setBreak("0x32", "IDATA", 1, 1, "", "TRUE", "I",
"_readTCNT()");
```

For additional information, see *Edit Breakpoints…*, page 230.

`__setMap` `__setMap(`*`address, segment, length, type`*`)`

Parameters

address	The start location (hex value)

`segment`	The memory segment name (string), one of: `CODE`, `IDATA`, `XDATA`, or `SFR`
`length`	The number of bytes to be covered by mapping (integer)
`type`	The memory mapping type; “`G`” (guarded) or “`P`” (protected) (string)

Return value

`int 0`

Description

Sets a given memory mapping. For additional information, see *Memory Map…*, page 233.

Example

```
__setMap(1234, "CODE", 1000, "G");
```

`__step` `__step(`*`kindOf`*`)`

Parameters

`kindOf`	Predefined string, one of: `"OVER"` does not enter C functions or assembler subroutines `"INTO"` enters C functions or assembler subroutines

Return value

`int 0`

Description

Executes the next statement or instruction. For additional information, see *Step*, page 228.

Example

```
__step("OVER");
```

`__writeFile` `__writeFile(`*`filehandle, value`*`)`

Parameters

`filehandle`	The macro variable used as the file handle set by the `__openFile` macro
`value`	The value to be written to the file. *`value`* is written using a format depending on with what access type the file was opened. In ASCII mode the value is written to the file as a string of hex digits corresponding to *`value`*. In binary mode the lowest byte of *`value`* is written as a binary byte

Return value

`int 0`

Description

Writes to a file.

Example

`__writeFile(`*`filehandle`*`, 123);`

`__writeMemoryByte` `__writeMemoryByte(`*`value, address, segment`*`)`

Parameters

`value`	The value to be written (integer)
`address`	The memory address (integer)
`segment`	The memory segment name (string), one of: `CODE`, `IDATA`, `XDATA`, or `SFR`

Return value

`int 0`

Description

Writes one byte to a given memory location.

Example

```
__writeMemoryByte(0xFF, 0x2000, "CODE");
```

Device description file

This chapter contains detailed information about the Device Description File (DDF) which is used by the IAR C-SPY Debugger to set up the SFR window and the interrupt simulation system. The DDF file consists mainly of two different sections, the [Sfr] section and the [INTERRUPT VECTOR] section.

SFR window setup

This section describes the `SFR` setup syntax for C-SPY for the 8051 microcontroller. For more examples, see the `*.ddf` files in the `\8051\config` directory in the program's root directory.

Syntax

```
SfrN = name, segment name, address[:bit[-endbit]], size[,
attribute][, attribute]...[; comment]
```

Note: This must be on a single line.

Restrictions

- The section under which the `SFR` definitions will exist is [`Sfr`] with contiguously enumerated keys `Sfr0` - `SfrN`.
- Each `SFR` can carry attributes.
- The `SFR` name is case sensitive.
- The address can be specified either as a hex or a decimal number.
- The bit-range (address field extension) is specified using decimal numbers.
- The size (# memory units, not bytes) can be specified as a hex or a decimal number.
- The base attribute must be specified as a decimal number (2, 8, 10, or 16, where 16 is default).
- The readable and writable mask attributes are specified using hex masks, where the bits correspond to true (1) and false (0) (def = all read/writable (all 1:s)).

Example

```
[Sfr]
Sfr0 = FOO, Memory, 0x0010, 2, readable(0x0)
Sfr1 = BAR, Memory, 0x0012, 1, base(10), writable(0x0)
Sfr2 = NIB, Memory, 0x0014:0-3, 1, base(2)
...
```

`readable(0x0)` specifies that reading from any of the bits in this SFR is not allowed, and `writable(0x0)` that writing to any of the bits is not allowed.

SFR GROUPS

In order to logically tie related SFRs together, groups should be specified under the section [`SfrGroupInfo`].

Syntax

```
GroupN=group name[,sfr name][,sfr name][,sfr name]... [;
comment]
[GroupN:1=[sfr name][,sfr name]... ] [; comment]
```

Example

```
[SfrGroupInfo]
Group0   = Timer0,FOO,BAR,NIB
Group1   = Timer1,SOMESFR,ANOTHERSFR,...
Group1:1 = MORESFR,...
...
```

Interrupt system simulation

To enable the interrupt system you have to set up the properties for the interrupts and load it into the C-SPY simulator.

You can define the interrupt system in two ways:

- Write your own interrupt system according to the specification described under *Interrupt system syntax* below.
- Use one of the predefined interrupt systems set up in the `*.ddf` (device description file) file. See the `\8051\config` directory for available device description files.

LOADING THE DEVICE DESCRIPTION FILE

In the IAR Embedded Workbench, you specify the device description file when setting options for your project. On the **Setup** page in the **C-SPY** category, check **Use device decription file** and enter the appropriate file name.

Load the device description file with the `-p` option.

Example

```
CW8051 demo.d03 -p c:\iar\ew23\8051\config\io51.ddf
```

INTERRUPT SYSTEM SYNTAX

To be able to use the interrupt simulation, you must specify what interrupts to simulate and how to simulate them.

This is done in the `[INTERRUPT VECTORS]` section of the device description file. If a global enable bit is used, this is specified first in the `[INTERRUPT VECTORS]` section. The global enable bit is specified with a `BIT-ADDRESS` (see *Syntax* below). After this optional global enable bit, all interrupts used can be specified.

Syntax

`Name`	Name of the interrupt. If the name contains space, enclose the string inside "".
`Vector`	Interrupt vector. The vector address for the interrupt.
`Pending`	Pending bit. The bit indicating that an interrupt occurred.
`Enable`	Enable bit(s) The local enable bit/bits used to enable/disable the interrupt.
`Priority`	Priority bit(s). The priority bit/bits. If more than one priority bit is used the priority is set low -> high (left -> right).
`Ctrl`	A control string consisting of two characters. The first character decides if the interrupt is active High or Low [H|L]. The second character decides if the interrupt should be cleared by Software or Hardware [S|H]
`BIT-ADDRESS`	Is a byte/bit pair enclosed in '<' and '>', separated by a comma ','. Example: <0x88,0x02> is bit 1 at address 88H.

Example

Below is a short sample from a `*.ddf` file. For more examples, see the `*.ddf` files in the `\8051\config` directory.

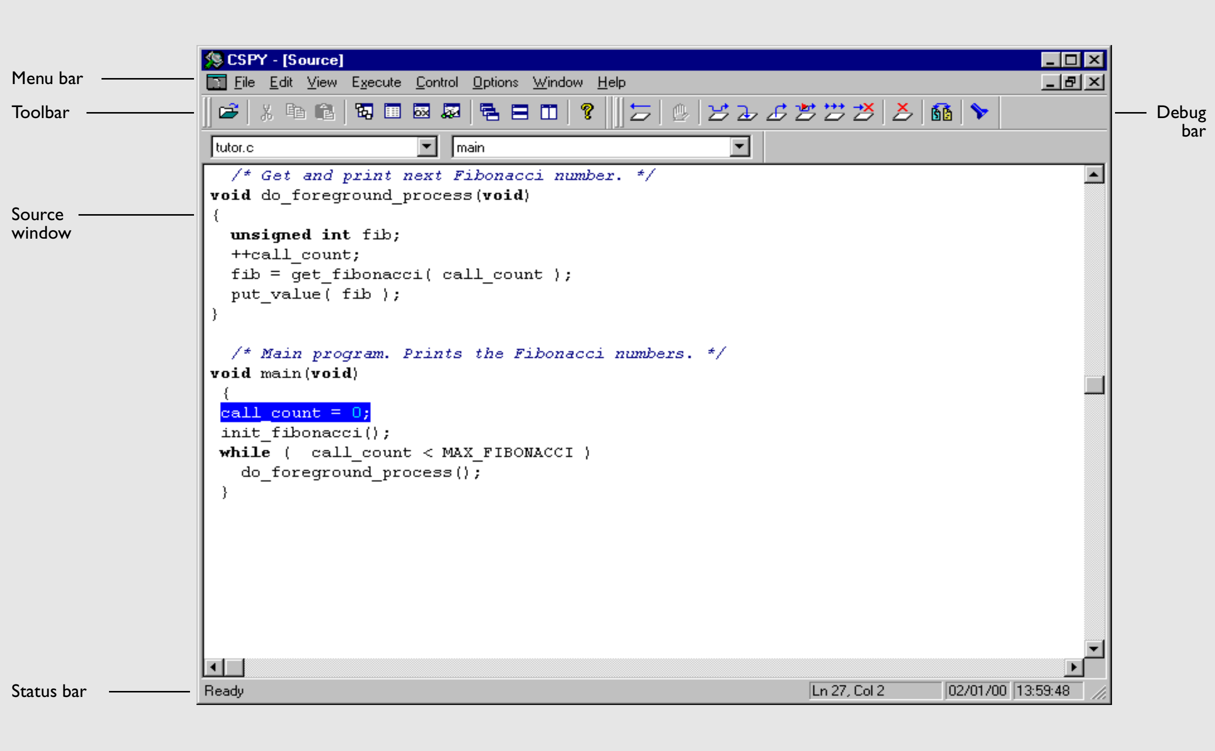
```
[INTERRUPT VECTORS]
//
// Global enable at address 0xA8, bit 7.
<0xa8,0x80>
//
// Name  Vector  Pending      Enable       Priority(low--high)  Ctrl
//
//  -------------------  External Interrupt 0  -------------------
extern0  0x03    <0x88,0x02>  <0xa8,0x01>  <0xb8,0x01>          HS
//
//  ---------------------  Timer 0 Overflow  ---------------------
timer0   0x0B    <0x88,0x20>  <0xa8,0x02>  <0xb8,0x02>          HH
```

C-SPY reference

This chapter contains detailed descriptions about the windows, menus, menu commands, and their components found in the IAR C-SPY® Debugger.

The C-SPY window

The following illustration shows the main C-SPY window:



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

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

Figure 128: C-SPY window

TYPES OF C-SPY WINDOWS

The following windows are available in C-SPY:

- Source window
- Watch window
- Report window
- Register window
- SFR window
- Profiling window
- Terminal I/O window
- Locals window
- Memory window
- Calls window
- Code Coverage window.

These windows are described in greater detail on the following pages.

MENU BAR

Gives access to the C-SPY menus:

Menu	Description
File	The File menu provides commands for opening and closing files, and exiting from C-SPY.
Edit	The Edit menu provides commands for use with the Source window.
View	The View menu provides commands to allow you to select which windows are displayed in the C-SPY window.
Execute	The Execute menu provides commands for executing and debugging the source program. Most of the commands are also available as icon buttons in the debug bar.
Control	The Control menu provides commands allowing you to control the execution of the program.
Options	The commands on the Options menu allow you to change the configuration of your C-SPY environment, register and display macros.
Window	The Window menu lets you select or open C-SPY windows and control the order and arrangement of the windows.
Help	The Help menu provides help about C-SPY.

Table 49: C-SPY menus

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

TOOLBAR AND DEBUG BAR

The toolbar and debug bar provide buttons for the most frequently used commands on the menus. You can move each bar to a different position in the C-SPY window, or convert it to a floating palette, by dragging it with the mouse.

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

Toolbar

This diagram shows the command corresponding to each of the toolbar buttons:

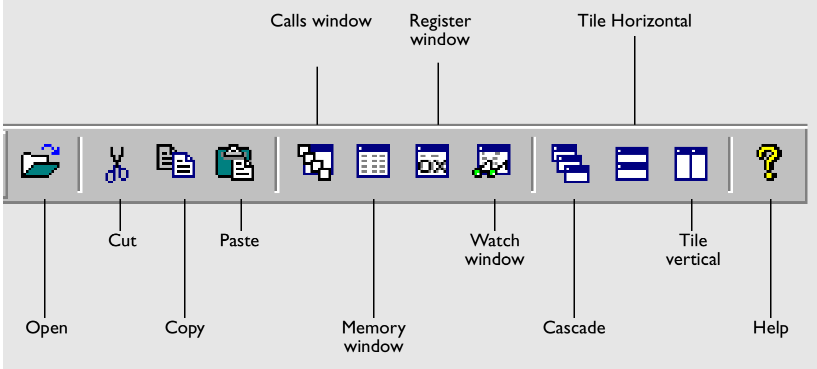


Figure 129: C-SPY toolbar

You can choose whether the toolbar is displayed by using the **Toolbar** command on the **View** menu.

Debug bar

The following diagram shows the command corresponding to each button:

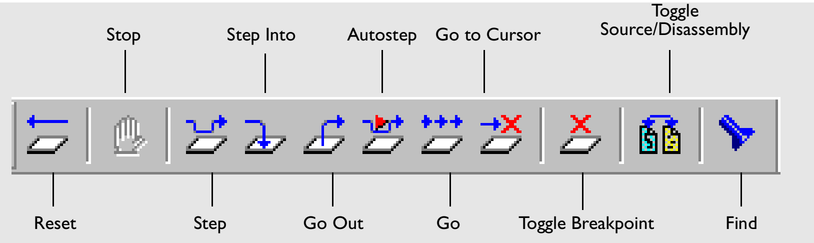


Figure 130: C-SPY debug bar

Use the **Debug Bar** command on the **View** menu to toggle the debug bar on and off.

SOURCE WINDOW

The C-SPY Source window shows the source program being debugged, as either C or assembler source code or disassembled program code. You can switch between source mode and disassembly mode by choosing **Toggle Source/Disassembly** from the **View** menu, or by clicking the **Toggle Source/Disassembly** button in the debug bar. Clicking the right mouse button in the Source window displays a pop-up menu:

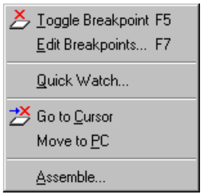
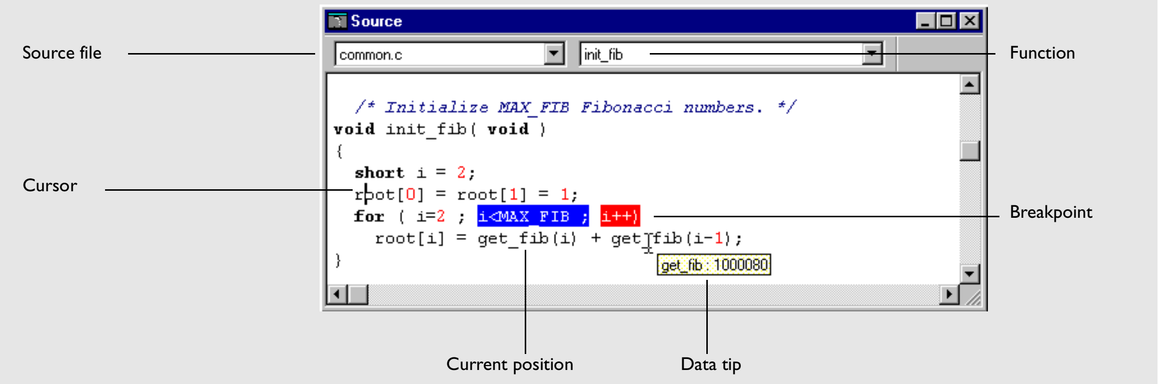


Figure 131: Source window pop-up menu

When you start C-SPY, the first executable statement in the `main` function will be displayed in the Source window. If the Source window is initially blank, no `main` function has been found and the program starts in a low-level assembly module, assembled without debug information, so there is no corresponding source code.

Source file and function

The **Source file** and **Function** boxes show the name of the current source file and function displayed in the Source window, and allow you to move to a different module or function by selecting a name from the corresponding drop-down list. The following types of highlighting are used in the Source window:



```
  /* Initialize MAX_FIB Fibonacci numbers. */
void init_fib( void )
{
  short i = 2;
  root[0] = root[1] = 1;
  for ( i=2 ; i<MAX_FIB ; i++)
    root[i] = get_fib(i) + get_fib(i-1);
}
```

Figure 132: Highlighting in C-SPY Source window

Current position

The current position indicates the next C statement or assembler instruction to be executed, and is highlighted in blue.

Cursor

Any statement in the Source window can be selected by clicking on it with the mouse pointer. The selected statement is indicated by the cursor.

Alternatively, you can move the cursor using the navigation keys.

The **Go to Cursor** command on the **Execute** menu will execute the program from the current position up to the statement containing the cursor.

Breakpoint

C statements or assembler instructions at which breakpoints have been set are highlighted in the Source window.

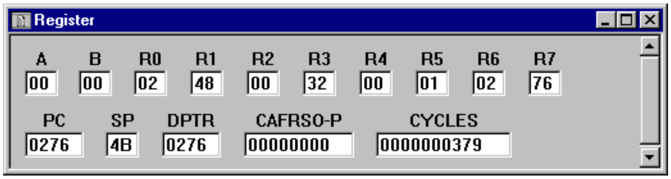
To set a breakpoint choose **Toggle Breakpoint...** from the **Control** menu or click the **Toggle Breakpoint** button in the toolbar.

Data tip

If you position the mouse pointer over a function, variable, or constant name in the C source shown in the Source window, the function start address or the current value of the variable or constant is shown below the mouse pointer.

REGISTER WINDOW

The Register window gives a continuously updated display of the contents of the processor registers, and allows you to edit them. When a value changes it becomes highlighted.



A	B	R0	R1	R2	R3	R4	R5	R6	R7
00	00	02	48	00	32	00	01	02	76

PC	SP	DPTR	CAFRSO-P	CYCLES
0276	4B	0276	00000000	0000000379

Figure 133: Register window

To change the contents of a register, edit the corresponding text box. The register will be updated when you tab to the next register or press Enter.

You can configure the registers displayed in the Register window using the **Register Setup** page in the **Settings** dialog box.

Note: If the contents of a register changes during execution, the change will be highlighted in this window.

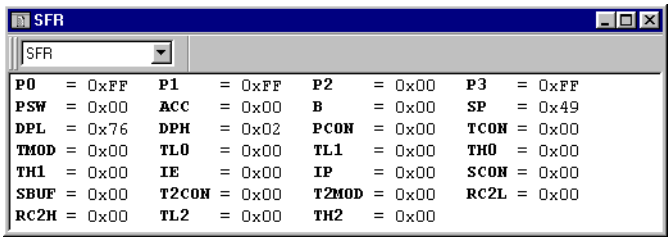
SFR WINDOW

The SFR window allows you to view and edit the contents of the special function registers.

In order to make this window available, you must specify a device description file (`ddf`) with SFR definitions when you start C-SPY. Use the option **Device description file** in the IAR Embedded Workbench as described on page 132, or the command line option `-p`, page 249, to specify the device description file.

The contents of the window is controlled by the SFR settings; see *Settings…*, page 239, for additional information.

Use the drop-down list to select which group of SFRs to display:



```
P0   = 0xFF  P1    = 0xFF  P2    = 0x00  P3   = 0xFF
PSW  = 0x00  ACC   = 0x00  B     = 0x00  SP   = 0x49
DPL  = 0x76  DPH   = 0x02  PCON  = 0x00  TCON = 0x00
TMOD = 0x00  TL0   = 0x00  TL1   = 0x00  TH0  = 0x00
TH1  = 0x00  IE    = 0x00  IP    = 0x00  SCON = 0x00
SBUF = 0x00  T2CON = 0x00  T2MOD = 0x00  RC2L = 0x00
RC2H = 0x00  TL2   = 0x00  TH2   = 0x00
```

Figure 134: SFR window

To edit the contents of an SFR, double-click its current value and type a new value. Then press Enter. The new contents will be highlighted, both in the SFR window and in the Memory window. If C-SPY during execution changes the memory or SFR contents, the change will be highlighted in this window.

Note: The value `0x--` signifies a write-only register.

MEMORY WINDOW

The Memory window gives a continuously updated display of a specified block of memory and allows you to edit it.

If C-SPY during execution changes the memory or SFR contents, the change will be highlighted in this window.

Choose **8**, **16**, or **32** to display the memory contents in blocks of bytes, words, or long words.

Clicking the right mouse button in the Memory window displays a pop-up menu which gives you access to several useful commands.

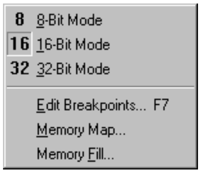
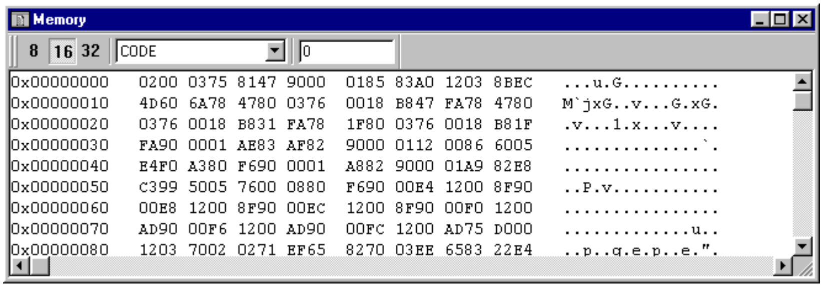


Figure 135: Memory window pop-up menu

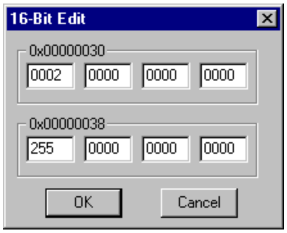
To edit the contents of memory, double-click the address or value you want to edit:



```
0x00000000    0200 0375 8147 9000  0185 83A0 1203 8BEC   ...u.G..........
0x00000010    4D60 6A78 4780 0376  0018 B847 FA78 4780   M`jxG..v...G.xG.
0x00000020    0376 0018 B831 FA78  1F80 0376 0018 B81F   .v...1.x...v....
0x00000030    FA90 0001 AE83 AF82  9000 0112 0086 6005   ..............`.
0x00000040    E4F0 A380 F690 0001  A882 9000 01A9 82E8   ................
0x00000050    C399 5005 7600 0880  F690 00E4 1200 8F90   ..P.v...........
0x00000060    00E8 1200 8F90 00EC  1200 8F90 00F0 1200   ................
0x00000070    AD90 00F6 1200 AD90  00FC 1200 AD75 D000   .............u..
0x00000080    1203 7002 0271 EF65  8270 03EE 6583 22E4   ..p..q.e.p..e.".
```

Figure 136: Editing memory in C-SPY

The following dialog box then allows you to edit the memory:



0x00000030: 0002 0000 0000 0000

0x00000038: 255 0000 0000 0000

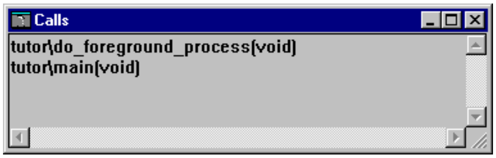
Figure 137: Memory edit dialog box

CALLS WINDOW

Displays the C call stack. Each entry has the format:

`module\function(values)`

where `values` is a list of the parameter values, or `void` if the function does not take any parameters.



```
tutor\do_foreground_process(void)
tutor\main(void)
```

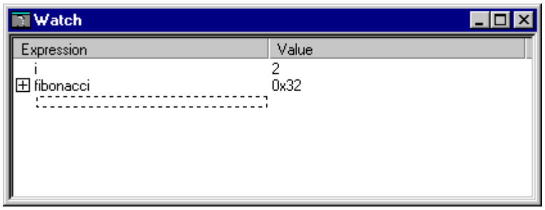
Figure 138: Calls window

STATUS BAR

Shows help text, and the position of the cursor in the Source window. Use the **Status Bar** command on the **View** menu to toggle the status bar on and off.

WATCH WINDOW

Allows you to monitor the values of C expressions or variables:



Expression	Value
i	2
fibonacci	0x32

Figure 139: C-SPY Watch window

Viewing the contents of an expression

To view the contents of an expression such as an array, a structure or a union, click the plus sign icon to expand the tree structure.

Adding an expression to the Watch window

To add an expression to the Watch window, click in the dotted rectangle, then hold down and release the mouse button. Alternatively, click the right mouse button in the Watch window and choose **Add** from the pop-up menu.

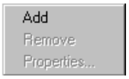


Figure 140: Add command on C-SPY Watch window pop-up menu

Then type the expression and press Enter.

You can also drag and drop an expression from the Source window.

Inspecting expression properties

Select an expression in the Watch window and choose **Properties...** from the pop-up menu.

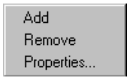
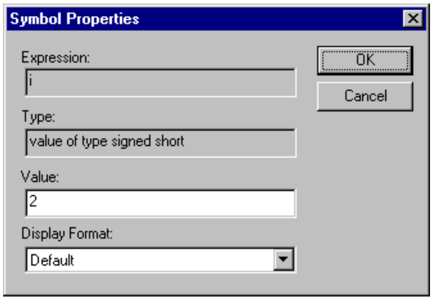


Figure 141: Properties... command on C-SPY Watch window pop-up menu

You can then edit the value of the expression and change the display format in the **Symbol Properties** dialog box:



Expression: i

Type: value of type signed short

Value: 2

Display Format: Default

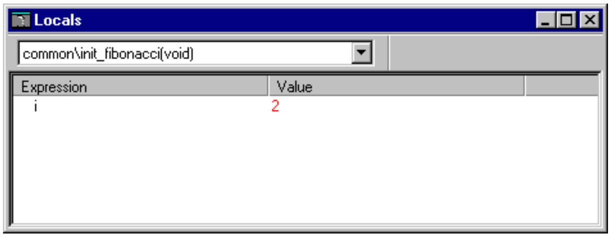
Figure 142: Symbol Properties dialog box

Removing an expression

Select the expression and press the Delete key, or choose **Remove** from the pop-up menu. When a value changes it becomes highlighted.

LOCALS WINDOW

Automatically displays the local variables and their parameters:



common\init_fibonacci(void)

Expression	Value
i	2

Figure 143: Locals window

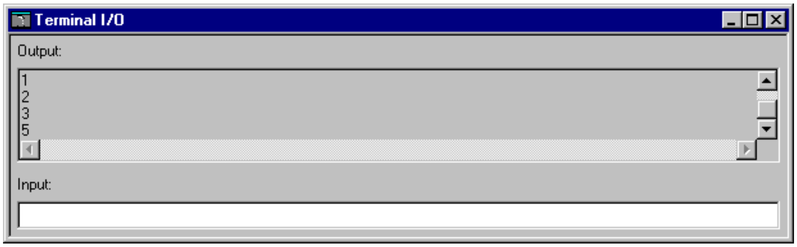
Editing the value of a local variable

To change the value of a local variable, click the right mouse button, and choose **Properties...** from the pop-up menu.

You can then change the value or display format in the **Symbol Properties** dialog box.

TERMINAL I/O WINDOW

Allows you to enter input to your program, and display output from it.



```
1
2
3
5
```

Figure 144: Terminal I/O window

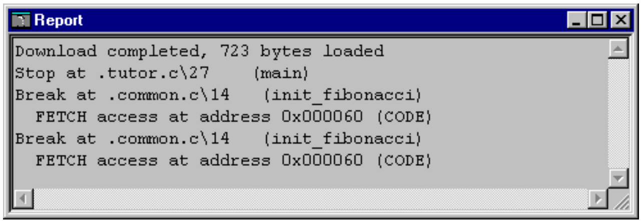
To use this window, you need to link the program with the option **Debug info with terminal I/O**. C-SPY will then direct `stdout` and `stderr` to this window. The window will only be available if your program uses the terminal I/O functions in the C library.

If the Terminal I/O window is open, C-SPY will write output to it, and read input from it.

If the Terminal I/O window is closed, C-SPY will open it automatically when input is required, but not for output.

REPORT WINDOW

Displays debugger output, such as diagnostic messages and trace information.

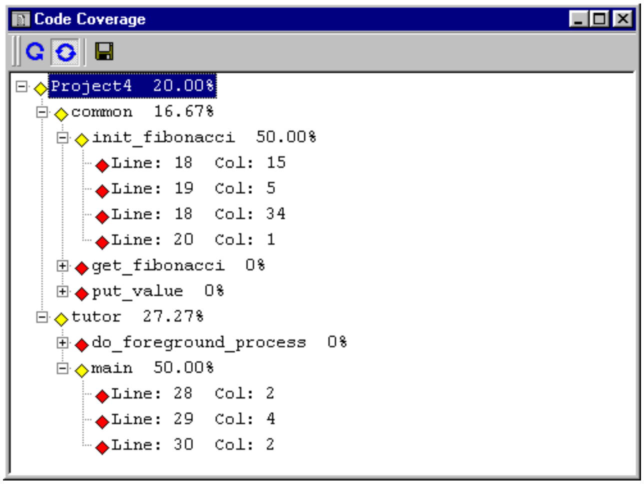


```
Download completed, 723 bytes loaded
Stop at .tutor.c\27    (main)
Break at .common.c\14   (init_fibonacci)
  FETCH access at address 0x000060 (CODE)
Break at .common.c\14   (init_fibonacci)
  FETCH access at address 0x000060 (CODE)
```

Figure 145: Report window

CODE COVERAGE WINDOW

Reports the current code coverage status. The report includes all modules and functions and the statements that have not yet been executed.



```
Project4   20.00%
  common   16.67%
    init_fibonacci   50.00%
      Line: 18   Col: 15
      Line: 19   Col: 5
      Line: 18   Col: 34
      Line: 20   Col: 1
    get_fibonacci   0%
    put_value   0%
  tutor   27.27%
    do_foreground_process   0%
    main   50.00%
      Line: 28   Col: 2
      Line: 29   Col: 4
      Line: 30   Col: 2
```

Figure 146: Code coverage window

The code coverage information is displayed in a tree structure, showing the program, module, function, and statement levels. The plus sign and minus sign icons allow you to expand and collapse the structure.

The percentage displayed at the end of every line shows the amount of code that has been covered so far. In addition, the following colors are used for giving you an overview of the current status on all levels:

- Red signifies that 0% of the code has been covered.
- Yellow signifies that some of the code has been covered.
- Green signifies that 100% of the code has been covered.

When a statement has been executed, it is removed from this window.

When the contents becomes dimmed and an asterisk (*) appears in the title bar, this indicates that C-SPY has continued to execute and that the Code Coverage window needs to be refreshed because the displayed information is no longer up to date.

Double-clicking on a statement line in the Code Coverage window displays that statement as current position in the Source window, which becomes the active window.

Clicking the right mouse button in the Code Coverage window displays a pop-up menu that gives you access to several useful commands.

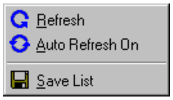
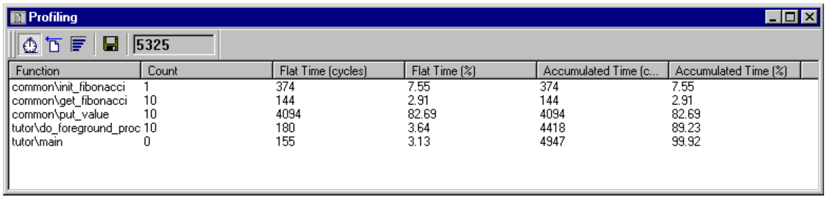


Figure 147: Code coverage pop-up window

The code coverage information is reset when the processor is reset.

PROFILING WINDOW

Displays profiling information:



Function	Count	Flat Time (cycles)	Flat Time (%)	Accumulated Time (c...	Accumulated Time (%)
common\init_fibonacci	1	374	7.55	374	7.55
common\get_fibonacci	10	144	2.91	144	2.91
common\put_value	10	4094	82.69	4094	82.69
tutor\do_foreground_proc	10	180	3.64	4418	89.23
tutor\main	0	155	3.13	4947	99.92

Figure 148: Profiling window

Clicking on the column header sorts the complete list depending on column. Double-clicking on an item in the **Function** column automatically displays the function in the Source window.

The information in the colums **Flat time** and **Accumulated time** can be displayed either as digits or as column diagrams. Flat time signifies the time in a function *excluding* child functions, while accumulated time signifies time in a function *including* its child functions.

Clicking the right mouse button in the Profiling window displays a pop-up menu which gives you access to several useful commands.

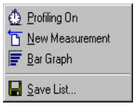


Figure 149: Profiling window pop-up menu

The following diagram shows the commands corresponding to the Profiling bar buttons:

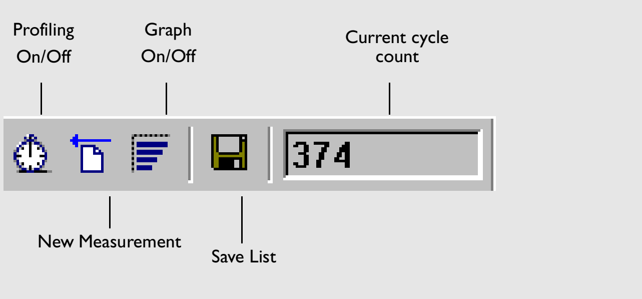


Figure 150: Profiling window buttons

Profiling On/Off

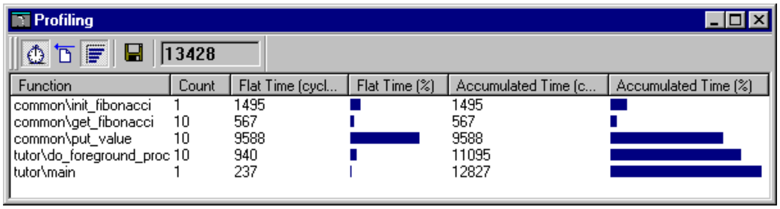
Switches profiling on and off during execution. Alternatively, use the **Profiling** command on the **Control** menu to toggle profiling on and off.

New Measurement

Starts a new measurement. By clicking on the icon the values displayed are reset to zero.

Graph On/Off

Displays the relative numbers as graph or numbers.



Function	Count	Flat Time (cycl...	Flat Time (%)	Accumulated Time (c...	Accumulated Time (%)
common\init_fibonacci	1	1495		1495	
common\get_fibonacci	10	567		567	
common\put_value	10	9588		9588	
tutor\do_foreground_proc	10	940		11095	
tutor\main	1	237		12827	

Figure 151: Profiling window Graph On/Off button

Save List

Saves list to file.

Current Cycle Count

Displays the current value of the cycle counter.

File menu

The **File** menu provides commands for opening and closing files, and exiting from C-SPY.

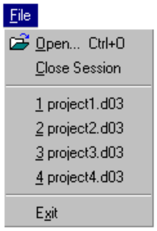


Figure 152: File menu (C-SPY)

OPEN...

Displays a standard **Open** dialog box to allow you to select a program file to debug.

If another file is already open it will be closed first.

Note: When you choose a file to be loaded via using the **Open** command or from the list of recently used files, a dialog box is displayed showing driver and session options to be used. You may modify them if a file to be loaded needs another set of options.

Observe that no driver option or input file name should be specified in the **Options** field.

CLOSE SESSION

Closes the current C-SPY session.

RECENT FILES

Displays a list of the files most recently opened, and allows you to select one to open it.

EXIT

Exits from C-SPY.

Edit menu

The **Edit** menu provides commands for use with the Source window.

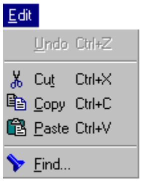


Figure 153: Edit menu (C-SPY)

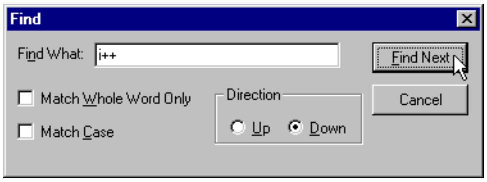
UNDO, CUT, COPY, PASTE

Provides the usual Windows editing features for editing text in some of the windows and dialog boxes.

FIND...

Allows you to search for text in the Source window.

This dialog box allows you to specify the text to search for:



Find What: i++

☐ Match Whole Word Only

☐ Match Case

Direction

○ Up ● Down

Figure 154: Find dialog box (C-SPY)

Enter the text you want to search for in the **Find What** text box.

Select **Match Whole Word Only** to find the specified text only if it occurs as a separate word. Otherwise `int` will also find `print`, `sprintf` etc.

Select **Match Case** to find only occurrences that exactly match the case of the specified text. Otherwise specifying `int` will also find `INT` and `Int`.

Select **Up** or **Down** to specify the direction to search.

Choose **Find Next** to start searching. The source pointer will be moved to the next occurrence of the specified text.

View menu

The **View** menu provides commands to allow you to select which windows are displayed in the C-SPY window.

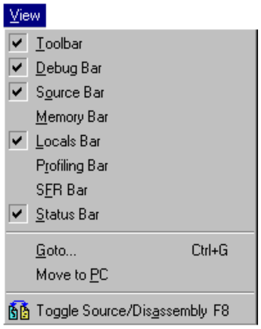


Figure 155: View menu (C-SPY)

TOOLBAR

Toggles on or off the display of the toolbar.

DEBUG BAR

Toggles on or off the display of the debug bar.

SOURCE BAR

Toggles on or off the Source window toolbar.

MEMORY BAR

Toggles on or off the Memory window toolbar.

LOCALS BAR

Toggles on or off the Locals window toolbar.

PROFILING BAR

Toggles on or off the Profiling window toolbar.

SFR BAR

Toggles on or off the SFR window toolbar, provided that you have specified the device description file (`ddf`) to be used. This file contains information about the SFRs, such as I/O registers (SFR) definitions, vector, and control register definitions.

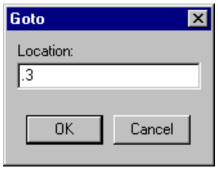
For additional information, see *Device description file*, page 132, and the chapter *Device description file*.

STATUS BAR

Toggles on or off the display of the status bar, along the bottom of the C-SPY window.

GOTO...

Displays the following dialog box to allow you to move the source pointer to a specified location in the Source window.



Location: .3

Figure 156: Goto dialog box

To go to a specified source line, prefix the line number with a period (`.`). For example:

Location	Description
`.12`	Moves to line 12 in current file.
`.tutor.c\12`	Moves to line 12 in file `tutor.c`.
`main`	Moves to function `main` in current scope.
`0x1000`	Moves to address `0x1000` (C-level debugging)
`1000`	Moves to address `1000` (assembly-level debugging)

Table 50: Examples of using Goto... command to move to a specified source line

MOVE TO PC

Moves the source pointer to the current program counter (PC) position in the Source window.

TOGGLE SOURCE/DISASSEMBLY

Switches between source and disassembly mode debugging.

Execute menu

The **Execute** menu provides commands for executing and debugging the source program. Most of the commands are also available as icon buttons in the debug bar.

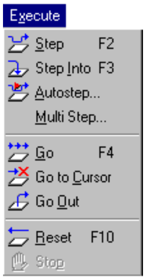


Figure 157: Execute menu

STEP

Executes the next statement or instruction, without entering C functions or assembler subroutines.

STEP INTO

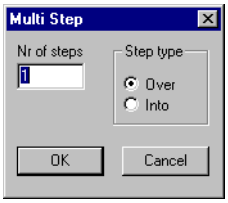
Executes the next statement or instruction, entering C functions or assembler subroutines.

AUTOSTEP...

Steps continuously, with a selectable time delay, until a breakpoint or program exit is detected.

MULTI STEP...

Allows you to execute a specified number of **Step** or **Step Into** commands. This option displays the following dialog box to allow you to specify the number of steps:


Multi Step

Nr of steps: 1

Step type:
● Over
○ Into

Figure 158: Multi Step dialog box

Select **Over** to step over C functions or assembler subroutines, or **Into** to step into each C function or assembler subroutine. Then choose **OK** to execute the steps.

GO

Executes from the current statement or instruction until a breakpoint or program exit is reached.

GO TO CURSOR

Executes from the current statement or instruction up to a selected statement or instruction.

GO OUT

Executes from the current statement up to the statement after the call to the current function.

RESET

Resets the target processor.

STOP

Stops program execution or automatic stepping.

Control menu

The **Control** menu provides commands to allow you to define breakpoints and change the memory mapping.

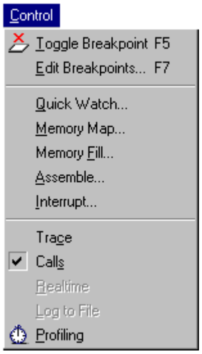


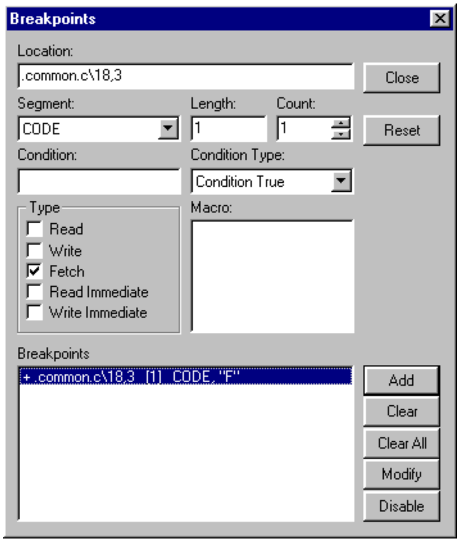
Figure 159: Control menu

TOGGLE BREAKPOINT

Toggles on or off a breakpoint at the statement or instruction containing the cursor in the Source window. This command is also available as an icon button in the debug bar.

EDIT BREAKPOINTS...

Displays the following dialog box which shows the currently defined breakpoints, and allows you to edit them or define new breakpoints:



Location: .common.c\18,3
Segment: CODE
Length: 1
Count: 1
Condition:
Condition Type: Condition True

Type
- ☐ Read
- ☐ Write
- ☑ Fetch
- ☐ Read Immediate
- ☐ Write Immediate

Breakpoints
+ .common.c\18,3 [1] CODE, "F"

Figure 160: Breakpoints dialog box

This dialog box lists the breakpoints you have set with the **Toggle Breakpoint** command, and allows you to define, modify, or remove breakpoints with break conditions.

To define a new breakpoint, enter the characteristics of the breakpoint you want to define and choose **Add**.

To modify an existing breakpoint, select it in the **Breakpoints** list and choose one of the following buttons:

Choose this	To do this
Clear	Remove the selected breakpoint.
Clear All	Removes all the breakpoints in the list.

Table 51: Modifying existing breakpoints

Choose this	To do this
Modify	Modifies the breakpoint to the settings you select.
Disable/Enable	Toggles the breakpoint on or off. Enabled breakpoints are prefixed with a + in the Breakpoints list.

Table 51: Modifying existing breakpoints (continued)

For each breakpoint you can define the following characteristics:

Location

The address in memory or any expression that evaluates to a valid address, for example a function or variable name.

When setting a *code* breakpoint, you can specify a location in the C source program with the formats `.source\line` or `.line`. Notice that the line must start with a `.` (period) which indicates the code breakpoint. For example, `.common.c\12` sets a breakpoint at the first statement on line 12 in the source file `common.c`.

When setting a *data* breakpoint, enter the name of a variable or any expression that evaluates to a valid memory location. For example, `my_var` refers to the location of the variable `my_var`, and `arr[3]` refers to the third element of the array `arr`.

Note: You cannot set a breakpoint on a variable that does not have a constant address in memory.

Segment

The memory segment in which the location or address belongs.

Length

The number of bytes to be guarded by the breakpoint.

Count

The number of times that the breakpoint condition must be fulfilled before a break takes place. Click **Reset** to reset this to 1.

Condition

A valid expression conforming to C-SPY expression syntax.

Condition type	Description
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 condition expression has changed.

Table 52: Breakpoint conditions

Note: The condition is evaluated only when the breakpoint is encountered.

For an example of a breakpoint with a condition, see *Defining complex breakpoints* in Tutorial 6, page 72.

For information about the C-SPY expressioin syntax, see the *C-SPY expressions* chapter in this guide.

Type

Specifies the type of memory access guarded by the breakpoint:

Type	Description
Read	Read from location.
Write	Write to location.
Fetch	Fetch opcode from location.
Read Immediate	Read from location, immediate break.
Write Immediate	Write to location, immediate break.

Table 53: Breakpoint types

The **Read**, **Write**, and **Fetch** breakpoint types never break execution within a single assembler instruction. **Read** and **Write** breakpoints are recorded and reported after the instruction is completed. If a **Fetch** breakpoint is detected on the first byte of an instruction, it will be reported before the instruction is executed; otherwise the breakpoint is reported after the instruction is completed.

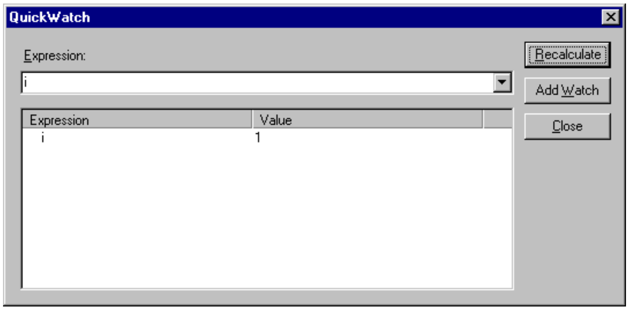
The **Read Immediate** and **Write Immediate** breakpoint types are only applicable to simulators and will cause a break as soon as encountered, even in the middle of executing an instruction. Execution will automatically continue, and the only action is to execute the associated macro. They are provided to allow you to simulate the behavior of a port. For example, you can set a **Read Immediate** breakpoint at a port address, and assign a macro to the breakpoint that reads a value from a file and writes it to the port location.

Macro

An expression to be executed once the breakpoint is activated.

QUICK WATCH…

Allows you to watch the value of a variable or expression and to execute macros. Displays the following dialog box to allow you to specify the expression to watch:



Expression: i

Expression	Value
i	1

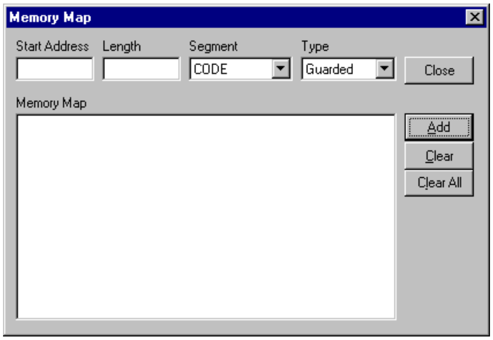
Figure 161: Watch window

Enter the C-SPY variable or expression you want to evaluate in the **Expression** box. Alternatively, you can select an expression you have previously watched from the drop-down list. For additional information, see *Expression syntax*, page 181.

Choose **Recalculate** to evaluate the expression, or **Add Watch** to evaluate the expression and add it to the Watch window. Choose **Close** to exit this dialog box.

MEMORY MAP…

C-SPY allows the simulation of non-existing and read-only memory by the use of memory maps. A memory map is a specified memory area with an access type attached to it, either no memory or read-only memory. The **Memory Map** dialog box allows you to define memory maps:



Segment: CODE

Type: Guarded

Figure 162: Memory Map dialog box

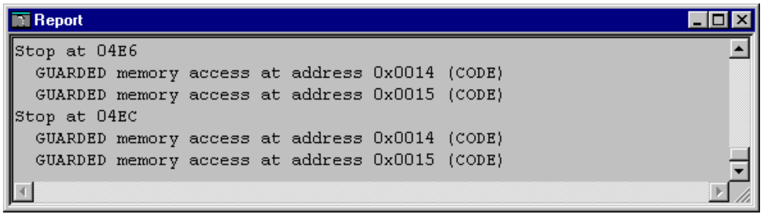
To define a new memory map, enter the **Start Address**, **Length**, and **Segment** and choose the **Type** according to the following table:

Type	Description
Guarded	Simulates addresses with no memory by flagging all accesses as illegal.
Protected	Simulates ROM memory by flagging all write accesses to this address as illegal.

Table 54: Memory map types

To delete an existing memory map, select it in the **Memory Map** list and choose **Clear**.

If a memory access occurs that violates the access type of that memory map, C-SPY will regard this access as illegal and display it in the Report window:



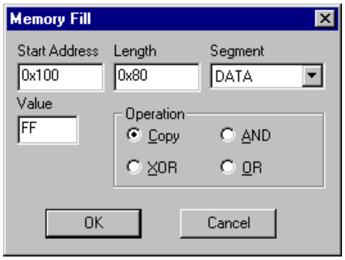
```
Stop at 04E6
  GUARDED memory access at address 0x0014 (CODE)
  GUARDED memory access at address 0x0015 (CODE)
Stop at 04EC
  GUARDED memory access at address 0x0014 (CODE)
  GUARDED memory access at address 0x0015 (CODE)
```

Figure 163: Illegal access reported in Report window

MEMORY FILL…

Allows you to fill a specified area of memory with a value.

The following dialog box is displayed to allow you to specify the area to fill:



Start Address: 0x100
Length: 0x80
Segment: DATA
Value: FF
Operation: ● Copy

Figure 164: Memory Fill dialog box

Enter the **Start Address** and **Length** in hexadecimal notation, and select the segment type from the **Segment** drop-down list.

Enter the **Value** to be used for filling each memory location and select the logical operation. The default is **Copy**, but you may choose one of the following operations:

Operation	Description
Copy	The Value will be copied to the specified memory area.
AND	An AND operation will be performed between the Value and the existing contents of memory before writing the result to memory.
XOR	An XOR operation will be performed between the Value and the existing contents of memory before writing the result to memory.
OR	An OR operation will be performed between the Value and the existing contents of memory before writing the result to memory.

Table 55: Memory fill operations

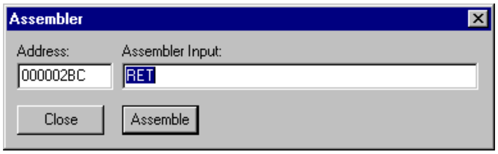
Finally choose **OK** to proceed with the memory fill.

ASSEMBLE...

Displays the assembler mnemonic for a machine-code instruction, and allows you to modify it and assemble it into memory.

Assemble... is only available in disassembly mode. If you are debugging in source mode, choose **Toggle Source/Disassembly** from the **View** menu to change mode.

Then double-click a line in the Source window, or position the cursor in the line and choose **Assemble...**. This dialog box shows the address and assembler instruction at that address:



Address: 000002BC

Assembler Input: RET

Figure 165: Assembler dialog box (C-SPY)

To modify the instruction, edit the text in the **Assembler Input** field and click **Assemble**.

You can also enter an address in the **Address** field and then press Tab to display the assembler instruction at that address.

INTERRUPT...

The interrupt simulation can be used in conjunction with macros and complex breakpoints to simulate interrupt-driven ports. For example, to simulate port input, first specify an interrupt that will cause the appropriate interrupt handler to be called. Then set a breakpoint at the entry of the interrupt-handler routine, and associate it with a macro that sets up the input data by reading it from a file or by generating it using an appropriate algorithm.

Note: C-SPY only polls for interrupts between instructions, regardless of how many cycles an instruction takes.

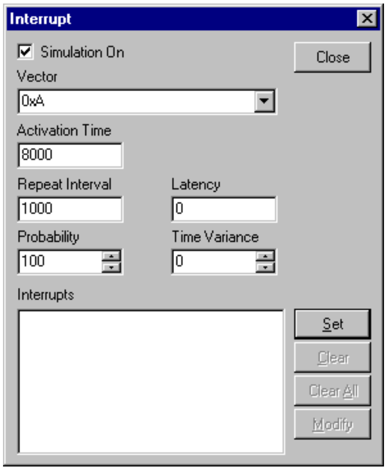
The C-SPY interrupt system uses the cycle counter as a clock to determine when an interrupt should be raised in the simulator. Changing the cycle counter will affect any interrupts you have set up in the **Interrupt** dialog box.

Performing a C-SPY reset will reset the cycle counter, and any interrupt orders with a fixed activation time will be cleared. For example, consider the case where the cycle counter is 123456, a repeatable order raises an interrupt every 4000 cycles, and a single order is about to raise an interrupt at 123500 cycles.

After a system reset the repeatable interrupt order remains and will raise an interrupt every 4000 cycles, with the first interrupt at 4000 cycles. The single order is removed.

For an example where interrupts are used, see *Tutorial 3*, page 50.

The **Interrupt...** command displays the following dialog box to allow you to configure C-SPY's interrupt simulation:



☑ Simulation On

Vector: 0xA

Activation Time: 8000

Repeat Interval: 1000

Latency: 0

Probability: 100

Time Variance: 0

Figure 166: Interrupt dialog box

To define a new interrupt enter the characteristics of the interrupt you want to simulate and choose **Set**.

To edit an existing interrupt select it in the **Interrupts** list and choose **Modify** to display or edit its characteristics, or **Clear** to delete it. Notice that deleting an interrupt does not remove any pending interrupt from the system.

For each interrupt you can define the following characteristics:

Vector

The interrupt vector table for a specific derivative must be defined in the device definitions file (`ddf`) which you specify using the C-SPY option **Use description file** in the IAR Embedded Workbench; see page 132 for additional information.

An interrupt vector can be selected from the drop-down list in the **Interrupt** dialog box.

For example, to generate an external interrupt, select the vector `external0`.

The `external0` is the same name as the `interrupt` name that is specified in the DDF-file.

Activation Time

The time, in cycles, after which the specified type of interrupt can be generated.

Repeat Interval

The periodicity of the interrupt in cycles.

Latency

Describes how long, in cycles, the interrupt remains pending until removed if it has not been processed. Latency is not implemented for the 8051 microcontroller.

Probability

The probability, in percent, that the interrupt will actually appear in a period.

Time Variance

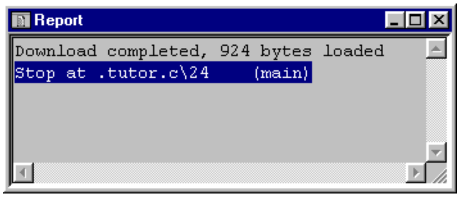
A timing variation range, as a percentage of the repeat interval, in which the interrupt may occur for a period. For example, if the repeat interval is 100 and the variance 5%, the interrupt may occur anywhere between T=95 and T=105, to simulate a variation in the timing.

Simulation On/Off

Enables or disables interrupt simulation. If the interrupt is disabled the definition remains but no interrupts will be generated.

TRACE

Toggles trace mode on or off. When trace mode is on, each step and function call is listed in the Report window:



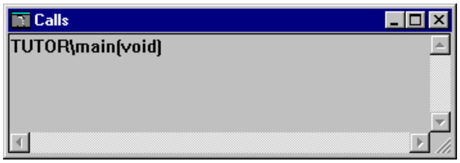
```
Download completed, 924 bytes loaded
Stop at .tutor.c\24     (main)
```

Figure 167: Report window with trace mode on

Note: The trace information will be reduced if calls mode is off.

CALLS

Toggles calls mode on or off. Toggling calls mode off does not affect the recognition of the program exit breakpoint. When calls mode is on, the function calls are listed in the Calls window:



```
TUTOR\main(void)
```

Figure 168: Calls window with calls mode on

REALTIME

Reserved for the emulator and ROM-monitor versions of C-SPY.

LOG TO FILE

Toggles writing to the log file on or off. When log file mode is on, the contents of the Report window are logged to a file. Choose **Select Log File…** from the **Options** menu to enable the log file function.

PROFILING

Toggles profiling on or off. For further information regarding **Profiling**, see *Profiling window*, page 222.

Options menu

The commands on the **Options** menu allow you to change the configuration of your C-SPY environment, and register macros.

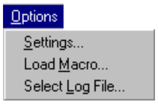


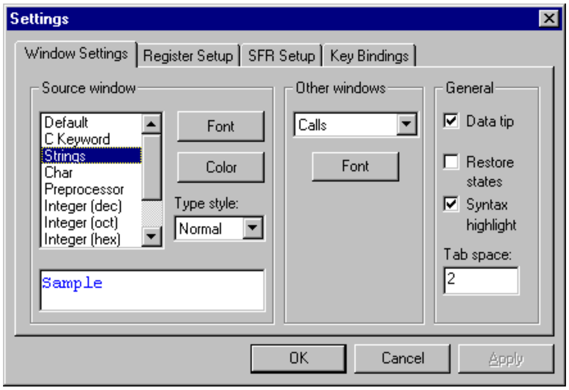
Figure 169: Options menu (C-SPY)

SETTINGS...

Displays the **Settings** dialog box to allow you to define the colors and fonts used in the windows, and to set up registers.

Window Settings

Allows you to specify the colors and fonts used for text in the Source window, the font used for text in other windows, and several general settings:



Type style: Normal

Calls

☑ Data tip

☐ Restore states

☑ Syntax highlight

Tab space: 2

Figure 170: Window Settings tab in Settings dialog box

To specify the style used for each element of C syntax in the Source window, select the item you want to define from the **Source window** list. The current setting is shown by the **Sample** below the list box.

You can choose a text color by clicking **Color**, and a font by clicking **Font**. You can also choose the type style from the **Type Style** drop-down list.

To specify the font used in other windows, choose a window from the drop-down list and click **Font**.

You can also specify the following general settings:

Settings	Description
Data tip	Shows the current value or function start address when the mouse pointer is moved over a variable constant, or function name in the Source window.
Restore states	Restores breakpoints, memory maps, and interrupts between sessions.
Syntax highlight	Highlights the syntax of C programs in the Source window.
Tab space	Specifies the number of spaces used for expanding tabs.

Table 56: General window settings

Then choose **OK** to use the new styles you have defined, or **Cancel** to revert to the previous styles.

Register Setup

Allows you to specify which registers to be displayed in the Register window and to define virtual registers.

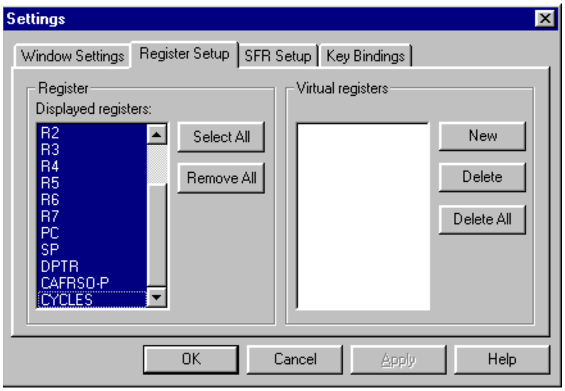


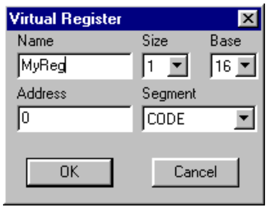
Figure 171: Register Setup tab in Settings dialog box

To specify which registers are displayed in the Register window, select them in the **Displayed Register** list and click **OK**.

Click **Select All** or **Remove All** to select or deselect all the registers.

The **Virtual registers** field allows you to specify any memory locations to be displayed in the Register window, in addition to the standard registers.

To define virtual registers, click **New** to display the **Virtual Register** dialog box:



Name: MyReg
Size: 1
Base: 16
Address: 0
Segment: CODE

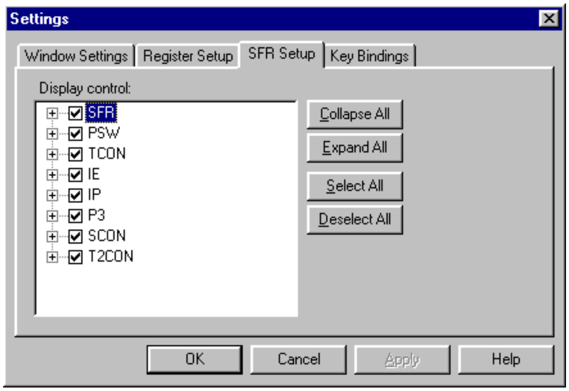
Figure 172: Virtual Register dialog box

Enter the **Name** and **Address** for the virtual register, and select the **Size** in bytes, **Base**, and **Segment** from the drop-down lists. Then choose **OK** to define the register. It will be displayed in the Register window after the standard registers you have selected.

For information about the processor-specific symbols, see *Assembler symbols*, page 182.

SFR Setup

The special function registers are categorized into groups, and the **Display control** tree structure allows you to select the registers or groups of registers to be displayed in the SFR window; see *SFR window*, page 216, for additional information.



Display control:
- ☑ SFR
- ☑ PSW
- ☑ TCON
- ☑ IE
- ☑ IP
- ☑ P3
- ☑ SCON
- ☑ T2CON

Figure 173: SFR Setup tab in Settings dialog box

To view the contents of a group, click on its plus-sign icon or select the **Expand All** button to view the contents of all groups. To hide the contents of a group, click on its minus-sign icon or select the **Collapse All** button to hide the contents of all groups.

To select a register or a group of registers, click on its check box. When you select a group, all registers in that group become selected. If you want to deselect one or more registers from a selected group, first expand the group and then uncheck each register that should not be displayed in the SFR window.

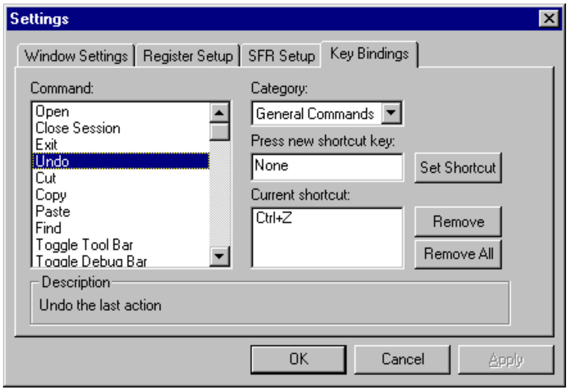
The **Select All** button selects all registers in all groups, and the **Deselect All** button deselects all groups and the registers in them.

Choose **OK** to set the display of SFRs.

Note: The **SFR Setup** page is available only if you have specified the device description file (`*.ddf`) to be used. This file includes necessary information about the SFRs. For additional information, see *Device description file*, page 132, and the chapter *Device description file* in this guide.

Key Bindings

Displays the shortcut keys used for each of the menu options, and allows you to change them:



Category: General Commands

Press new shortcut key: None

Current shortcut: Ctrl+Z

Description

Undo the last action

Figure 174: Key Bindings tab in Settings dialog box

To define a shortcut key select the command category from the **Category** list, and then select the command you want to edit in the **Command** list. Any currently defined shortcut keys are shown in the **Current shortcut** list.

To add a shortcut key to the command click in the **Press new shortcut key** box and type the key combination you want to use. Then click **Set Shortcut** to add it to the **Current shortcut** list. You will not be allowed to add it if it is already used by another command.

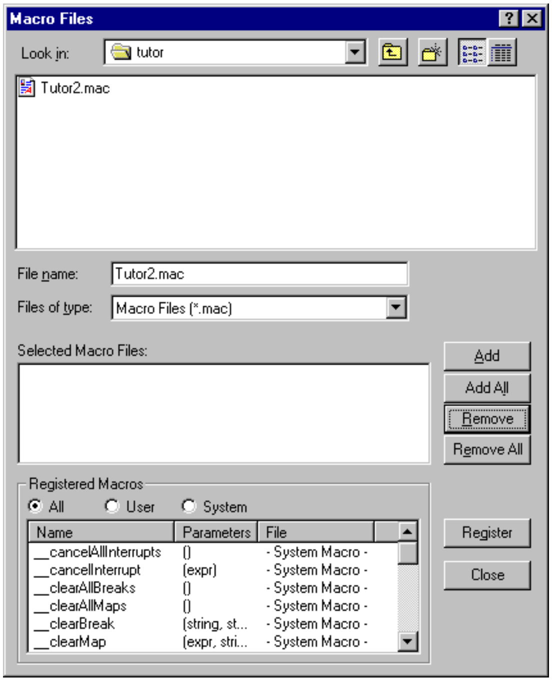
To remove a shortcut key select it in the **Current shortcut** list and click **Remove**, or click **Remove All** to remove all shortcut keys for a command.

Then choose **OK** to use the key bindings you have defined, and the menus will be updated to show the new shortcuts.

You can set up more than one shortcut for a command, but only one will be displayed in the menu.

LOAD MACRO...

Displays the following dialog box, to allow you to specify a list of files from which to read macro definitions into C-SPY:



Look in: tutor

File name: Tutor2.mac

Files of type: Macro Files (*.mac)

Registered Macros

● All ○ User ○ System

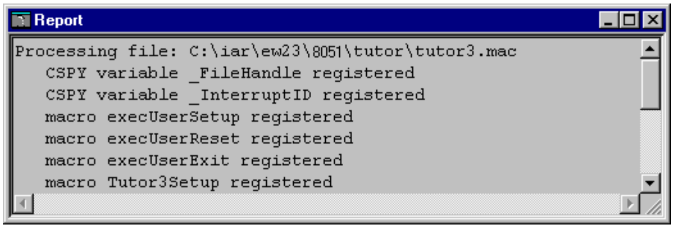
Name	Parameters	File
__cancelAllInterrupts	()	- System Macro -
__cancelInterrupt	(expr)	- System Macro -
__clearAllBreaks	()	- System Macro -
__clearAllMaps	()	- System Macro -
__clearBreak	(string, st...	- System Macro -
__clearMap	(expr, stri...	- System Macro -

Figure 175: Macro Files dialog box

Select the macro definition files you want to use in the file selection list, and click **Add** to add them to the **Selected Macro Files** list or **Add All** to add all the listed files.

You can remove files from the **Selected Macro Files** list using **Remove** or **Remove All**.

Once you have selected the macro definition files you want to use click **Register** to register them, replacing any previously defined macros or variables. The macros are listed in the Report window as they are registered:



```
Processing file: C:\iar\ew23\8051\tutor\tutor3.mac
   CSPY variable _FileHandle registered
   CSPY variable _InterruptID registered
   macro execUserSetup registered
   macro execUserReset registered
   macro execUserExit registered
   macro Tutor3Setup registered
```

Figure 176: Listing of registered macros in Report window

Registered macros are also displayed in the scroll window under **Registered Macros**.

Clicking on either **Name** or **File** under **Registered Macros** displays the column contents sorted by macro names or by file. Clicking once again sorts the contents in the reverse order. Selecting **All** displays all macros, selecting **User** displays all user macros, and selecting **System** displays all system macros.

Double-clicking on a user-defined macro in the **Name** column automatically opens the file in Microsoft Notepad, where it is available for editing.

Click **Close** to exit the **Macro Files** window.

SELECT LOG FILE...

Allows you to log input and output from C-SPY to a file. This command displays a standard **Save As** dialog box to allow you to select the name and the location of the log file.

Browse to a suitable folder and type in a filename; the default extension is `log`. Then click **Save** to select the specified file.

Choose **Log to File** in the **Control** menu to turn on or off the logging to the file.

Window menu

The first section of the **Window** menu contains commands to let you control the order and arrangement of the C-SPY windows.

The central section of the menu lists each of the C-SPY windows. Select a menu command to open the corresponding window.

The last section of the menu lists the open windows. Selecting a window makes it the active window.

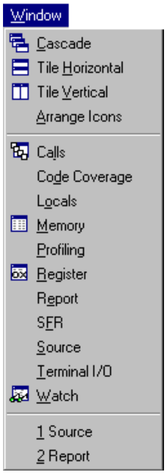


Figure 177: Window menu (C-SPY)

CASCADE

Rearranges the windows in a cascade on the screen.

TILE HORIZONTAL

Tiles the windows horizontally in the main C-SPY window.

TILE VERTICAL

Tiles the windows vertically in the main C-SPY window.

ARRANGE ICONS

Tidies minimized window icons in the main C-SPY window.

Help menu

Provides help about C-SPY.

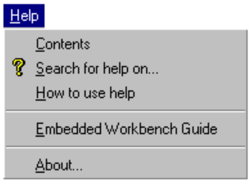


Figure 178: Help menu (C-SPY)

CONTENTS

Displays the Contents page for help about C-SPY.

SEARCH FOR HELP ON...

Allows you to search for help on a keyword.

HOW TO USE HELP

Displays help about using help.

EMBEDDED WORKBENCH GUIDE

Provides access to a hypertext version of this user guide.

ABOUT...

Displays the version number of the IAR C-SPY Debugger user interface and of the C-SPY driver for the 8051.

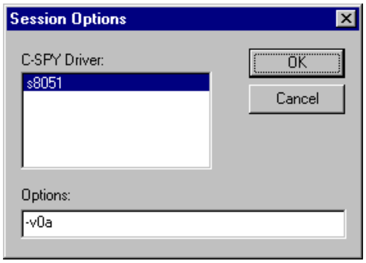
C-SPY command line options

This chapter gives information about how you set C-SPY options from the command line and lists a summary of command line options.

Normally, you specify the C-SPY options among the other project options in the IAR Embedded Workbench; see the *C-SPY options* in *Part 3: The IAR Embedded Workbench* chapter in this guide for detailed information.

Setting C-SPY options from the command line

You can specify C-SPY options when you start the IAR C-SPY Debugger, `cw23.exe`, with the Windows **Run…** command or from the command line. When you run C-SPY outside the IAR Embedded Workbench, the following dialog box will appear when you open a project:



C-SPY Driver: s8051

Options: -v0a

Figure 179: Session Options dialog box

Note: If you specify a valid input file name and a driver (and any other session options) on the command line, C-SPY loads the input file automatically. Otherwise, if no driver is specified (and there is more than one installed), a dialog box appears and asks you to choose one and C-SPY continues. If no input file is given, C-SPY goes into standby mode. In this case, you must load the input file from the **File** menu to continue.

USING MACRO FILES FOR OPTIONS

There is a possibility to use the macro `execUserInit()` to tell C-SPY what options to use. Define this in a C-SPY macro file:

```
execUserInit()
{
__processorOption("-spCOM1:9600,N,8,1,NONE");
__processorOption("-c1");
}
```

Note that the macro `execUserInit()` cannot contain the `__transparent()` macro. This can be done in `execUserPreload()` since it is called after the communication has been initialized.

Summary of command line options

The following command line options are available:

Option	Description
`-c1`	Verifies memory after downloading to ROM-monitor.
`-c2`	Verifies every byte after downloading to ROM-monitor.
`-d` *`driver`*	Selects C-SPY driver.
`-f` *`file`*	Specifies setup file.
`-n`	Suppresses downloading of code to ROM-monitor.
`-p` *`file`*	Loads device description file.
`-rz`	Disables fast download to the ROM-monitor.
`-sp`	Specifies the serial communication for the ROM-monitor.
`-sl`	Creates log file for the ROM-monitor communication.
`-v{0|1}`	Specifies processor option.
`-x`	Allows C-SPY to use a medium memory model where XDATA and CODE are mapped to the same memory.

Table 57: Summary of C-SPY command line options

Descriptions of C-SPY command line options

The following section gives details about the C-SPY command line options.

`-c1` `-c1`

Use this option to verify that the memory on the EVB is writable and mapped in a consistent way. A warning message will be generated if there are any problems during download.

`-c2` `-c2`

This option is similar to the `-c1` option but it checks every byte after downloading to verify that there are no hardware problems.

`-d` `-d` *`driver`*

Use this option to select the appropriate driver for use with C-SPY:

Driver	C-SPY version
s8051.cdr	Simulator
r8051.cdr	ROM monitor
r8051i.cdr	INTEL RISM monitor

Table 58: C-SPY supported drivers

Example

To debug `project1.d03` with the simulator driver:

```
cw23 -d s8051 project1.d03
```

`-f` `-f` *`file`*

Use this option to register the contents of the specified macro file in the C-SPY startup sequence. If no extension is specified, the extension `mac` is assumed.

Example

To register `watchdog.mac` at startup when debugging `watchdog.d03`:

```
cw23 -f watchdog.mac watchdog.d03
```

`-n` `-n`

This option disables the downloading of code, which can be time-consuming; instead it creates C-SPY tables internally. This command is useful if you need to exit C-SPY for a while and continue without loading code. The implicit `RESET` performed by C-SPY at startup is not disabled, though.

`-p` `-p` *`file`*

Use this option to load a device description file which contains various device specific information such as I/O registers (SFR) definitions, vector, and control register definitions.

Example

To use the `io051.ddf` description file, enter the following command:

```
cw23 -p io051.ddf
```

`-rz` `-rz`

Fast downloading of user code is enabled by default in the ROM-monitor version of C-SPY. If you use the `-rz` option, downloading will take more time since the error-free protocol is used. However, this should only be necessary if the ROM-monitor is not fast enough to process the data stream, or due to an insufficiently shielded communication cable.

If fast download fails constantly, there are a couple of things you can do:

- Lower the transmission speed (if possible).
- Use RTS/CTS handshaking between C-SPY and the ROM-monitor.
- Disable fast downloads. On the command line use the `-rz` option.

The precompiled ROM-monitor initializes the serial port to 9600 baud.

Your ROM-monitor might manage higher speeds. If the CPU has a clock rate of 16MHz or more, 19200 baud should work properly. This gives a download speed that peaks at about 2Kbytes per second.

`-sp` `-sp`*`port`*`[:`*`baud`*`[:`*`parity`*`[,`*`bits`*`[,`*`stop`*`[,`*`handshaking`*`]]]]]`

Parameters	**Description**
port	One of the supported ports: `COM1`, `COM2`, `COM3`, `COM4`
baud	One of following speeds: `300`, `600`, `1200`, `2400`, `4800`, `9600`, `19200`, `38400`, `57600`, `115200` (default `9600`)
parity	Only `N` (None) is allowed.
bits	Only `8` data bits are allowed.
stop	`1` or `2` stop bits (default 1).
handshaking	`NONE` or `RTSCTS` (default `NONE`).

Table 59: Serial port command line options

If this option has not been given, C-SPY will try using the COM1 port at 9600 baud. Your evaluation board must of course support the requested baud rate.

Example

To use COM1 at 38400 baud, add the following to the C-SPY command line:

```
-spCOM1:38400,N,8,1,NONE
```

`-s1` `-s1` *`logfile`*

Use this option to log the communication between C-SPY and the ROM-monitor to the specified log file. This can be particularly useful for trouble-shooting purposes.

`-v` `-v{`*`option`*`}`

Specifies the processor variant as follows:

Option	Description
-v0	Max 256 byte data, 8 Kbyte code
-v1	Max 64 Kbyte data, 8 Kbyte code

Table 60: Available processor type options

`-x` `-x`

This option allows C-SPY to use a medium memory model where `XDATA` and `CODE` are mapped to the same memory.

Part 5: C-SPY for the 8051 ROM-monitor

This part of the 8051 IAR Embedded Workbench™ User Guide contains the following chapters:

- Introduction to the ROM-monitor
- Controlling user applications
- The ROM-monitor boards
- The ROM-monitor program
- Advanced topics
- Diagnostic messages.

These chapters assume that you already have some working knowledge of the evaluation board (EVB) you are using.

You should read this part in conjunction with Part 4: The C-SPY simulator in this guide.

Introduction to the ROM-monitor

This chapter helps you get started using the 8051 ROM-monitor and describes the differences between the ROM-monitor and simulator versions of the 8051 C-SPY debugger. It assumes that you already have working knowledge of the evaluation board you are using.

The C-SPY ROM-monitor

The IAR C-SPY ROM-monitor consists of the IAR C-SPY Debugger and a target monitor program which comes in the following different versions:

Target monitor program	Description
`Mon517.a03`	The EM31 ROM-monitor board, 32Kbytes PROM, 4800 baud, @ 12 MHz
`Mon504.a03`	The KitCon-504 board, 128Kbytes Flash, 19200 baud, @ 40 MHz, 9600 baud @ 20 MHz
`Mon505.a03`	The KitCon-504 board, 128Kbytes Flash, 9600 baud @ 8 MHz.
`Mon515.a03`	The KitCon-504 board, 128K Flash, 9600 baud @ 10 MHz.
`Mon541.a03`	The KitCon-541 board, 128 KBytes Flash, @12 MHz
`Monem31.a03`	The MCB-517 board, 16/32Kbytes PROM, 4800 baud, @ 12 MHz

Table 61: ROM-monitor versions

The target monitor programs are available in the `\8051\src\rom` directory.

In order to be able to control the execution of your application from C-SPY, the ROM-monitor must be installed on your evaluation board.

The ROM-monitor file can be burned directly into an EPROM, which replaces the monitor supplied with your board. To be able to do this you must use an appropriate programming adapter, typically supplied with the evaluation board, or use a separate EPROM programmer.

The ROM-monitor can also be downloaded into flash ROM.

All files are in INTEL-format.

Differences between the ROM-monitor and simulator versions of C-SPY

The following table summarizes the key differences between the ROM-monitor and simulator versions of C-SPY:

C-SPY ROM-Monitor	C-SPY simulator
Only OP-fetch breakpoints.	OP-fetch and data breakpoints.
Execution in real-time.	Not real time.
Real interrupts.	Simulated interrupts.
No cycle counter.	Cycle counter.

Table 62: Differences between C-SPY ROM-monitor and C-SPY simulator

The ROM-monitor program

This section gives a brief, target independent, description of how a ROM-monitor works.

Communication

The ROM-monitor starts as soon as the EVB has been reset. It first tries to set up a new connection with C-SPY by repeatingly sending out four characters with the hexadecimal values `c0 1a e5 c0` until getting an answer from C-SPY.

When the connection is established C-SPY starts controlling the ROM-monitor with options like **read/write memory**, **read/write registers**, **go**, **single step** and **reset**.

All commands and data sent between C-SPY and the ROM-monitor are put into packets. Bad or lost packets are retransmitted until the receiver acknowledges them. Every time C-SPY has no ordinary commands to send to the ROM-monitor, it sends a test command to check if the ROM-monitor is still active or if it has been reset.

Execution of user code

When the ROM-monitor is running, all interrupts are disabled. The ROM-monitor keeps the user registers in its working area and uses its own stack.

Before executing the user program, all registers must be restored. The ROM-monitor copies some or all of the user registers to the user stack. After that it pops them back into the registers, restores the interrupts and makes a subroutine return to the user code. Some space must be left on the stack to allow for calls to and returns from the ROM-monitor.

Entering the ROM-monitor from user code is usually done through a breakpoint or an interrupt. The breakpoint consists of an instruction written to the address, in the user code, where the execution should be stopped. User instructions, overwritten by breakpoints, are kept and stored by C-SPY.

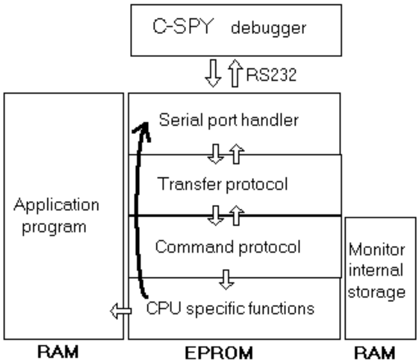
When single stepping, temporary breakpoint(s) are put where the next instruction will begin. Overwritten code is stored by the ROM-monitor and restored after executing the instruction. Interrupts are always disabled during single stepping. On some CPUs a trace bit can be set instead of putting breakpoints in the code. The type of breakpoint instruction depends on the CPU and can either be an instruction that causes an interrupt or it can be a subroutine call to the ROM-monitor.

An interrupt is also used to stop user programs if no breakpoints can be reached.

This interrupt handles incoming characters received from the serial port during execution of user code. On some EVBs this type of interrupt is used and on others a periodical interrupt from a timer must be used instead.

C-SPY does not send any characters to the ROM-monitor when user code is running. When **Stop** is pressed a stop character is sent from C-SPY. The interrupt handler will detect it and jump to the ROM-monitor.

Most CPUs disable interrupts at reset. The ROM-monitor does not enable any interrupts. Therefore the Stop interrupt has to be turned on by the user. In the 8051 this is done by setting the ES and EA bits in the IE `sfr` register.



The transfer and command protocols are delivered as library functions. The serial port handler and parts of the CPU specific functions are delivered in source code.

Controlling user applications

This chapter describes how to control applications using the ROM-monitor.

Breakpoints

Code breakpoints are `LCALL` instructions, which call the ROM-monitor. The size of the `LCALL` is 3 bytes and this may cause problems in cases where jumps to instructions are in the middle of a covering breakpoint.

```
       1000 E4    CLR A
       1001 7810  MOV R0, #10h<- breakpoint is set here
LOOP:  1003 26    ADD A, @R0
       1004 D8FD  DJNZ R0, LOOP<- PC points here
```

In the example code above, the bytes containing the values 78, 10, 26 are replaced with the 3-byte breakpoint/`LCALL`. A go in C-SPY, from the current PC position will result in something else than `ADD A, @R0`.

Situations like this cannot be detected by C-SPY and you should be aware of this when you set your own breakpoints.

Breakpoints set by C-SPY are always at the beginning of C statements and C functions. To ensure that these are correct you should always compile your application modules with the **Code added to statements: 3 NOPs** (`-r1=3`) option, to tell the compiler to insert three extra NOPs at the beginning of each statement.

When stepping at assembler level, C-SPY checks if you are stepping to an instruction that would, when running, be replaced with the last two bytes of a breakpoint, and then gives a warning. By stepping through the assembler code you can trace badly placed breakpoints.

C-SPY also checks that no breakpoints overlap each other.

The Control C feature

To stop a running program you click on the **Stop** button in C-SPY. A stop character will be sent to the ROM-monitor. The character is detected by an interrupt routine that stops the programs and enters the ROM-monitor. This will only work if interrupts have been turned on by your application or from C-SPY.

Note that the interrupts are turned off by default.

By default the ROM-monitor boards are configured to use the internal serial port for the communication with C-SPY.

To enable the internal serial port interrupt from C-SPY, do this in **Quick Watch** dialog box:

```
ES=1
EA=1
```

The following example shows how to enable this interrupt in your program:

```
#include <io51.h>
void enable_control_c(void)
{
ES=1;/* Enable internal serial port interrupt */
EA=1;/* Enable all enabled interrupts */
}
```

Each time the ROM-monitor starts execution of the application code it puts the serial interrupt vector into the application code. If the internal serial port interrupt is used by the ROM-monitor, a `LJMP` *`xxx`* vector is written to `0023h`.

Single stepping

When single stepping at assembler level all interrupts are turned off (if the ROM-monitor allows all interrupts to be turned off).The monitor temporarily sets out a breakpoint where the instruction ends and starts running the user application.

Debugging in real time

This section describes restrictions to C programs in a real-time environment.

To ensure that the program is executed in real time set **Realtime ON** in the **Control** menu. When real-time checking is enabled C-SPY does not execute the following commands in the **Execute** menu that require the CPU to be halted: **Step**, **Step Into**, **Autostep**, and **Go Out** (and their toolbar button equivalents).

If the C function stack, **Calls**, is enabled, and the Terminal I/O window is open, the following warnings will be displayed:

```
Warning[20]: Calls disabled in realtime
Warning[21]: Termio disabled in realtime
```

The C function stack and terminal I/O will automatically be turned off until the **Realtime** command is set back to **OFF**.

Real-time checking is off by default.

DEBUG OPTIONS

The code must be compiled with the **Generate debug information - Code added to statements: 3 NOP's** (`-r1=3`) option. This option inserts 3 NOP instructions at the beginning of each C statement to reserve space for the 3-bytes long break instruction. This is to make sure that two statements are not closer than three bytes to each other. Otherwise breakpoint overlap errors will occur during the debugging.

CPU HALT

When C-SPY is executed in non-real-time mode (default) the ROM-monitor CPU will often be temporarily halted for information exchange with C-SPY. This is however not seen by the user.

Note that this affects the interrupt timing, and interrupts that occur during the halt periods could be lost. The execution speed will also be reduced.

Only the **Go** and **Goto Cursor** commands with the C function stack disabled (**Calls OFF**) execute the program without halting the CPU, with full interrupt support.

Any other execution command will enable the C function stack which halts the user application.

Note that communication through the Terminal I/O window also temporarily halts the ROM-monitor CPU even when executing the **Go** or **Goto Cursor** commands (or their toolbar button equivalents) on the **Execute** menu.

PROCESSOR SHARING

If a program consists of several functions that are switched on and off from a realtime executive, or similar processor sharing system, only modules from one task at a time should be compiled with debug. Otherwise there is a potential risk that the C function stack will contain incorrect information.

Debugging interrupts

If the ROM-monitor EPROM covers the exception vector table, all vectors are normally remapped to the beginning of the user RAM area. The extra overhead added is an `LJMP` instruction that takes 2 cycles. This allows you to install exception vectors in RAM.

For all ROM-monitors, all unused vectors are filled with breakpoints that gives control to the ROM-monitor if you failed to install a vector properly. This is done every time the application is downloaded.

To see how the ROM-monitor handles interrupt vectors:

1. Download an application that installs at least one exception vector.
2. Look at the exception vector table using the Memory window.
3. Find out where the vector you are interested in points.
4. Make sure the source window displays assembly instructions. If not, press the **Toggle C/Assembler** button.
5. Choose **Goto...** from the **View** menu, then type the addresses where the exception vectors you are interested in points. There should be a `LJMP` instruction that points to your interrupt handler, otherwise it should be the beginning of your interrupt handler.

Resolving problems with the ROM-monitor

This section includes suggestions for resolving problems when debugging with C-SPY in conjunction with a ROM-monitor.

VERIFYING THE DOWNLOAD

Use the **Target consistency check** options on the ROM-monitor page of the C-SPY options: **Verify boundaries** (`-c1`) or **Verify all** (`-c2`) to verify that the EVB memory is writable and mapped in a consistent way. The **Verify all** option verifies every byte after loading, and so is considerably slower but more thorough.

For more information, see the chapters *C-SPY options* and *C-SPY command line options*, respectively.

CHECKING FOR PROBLEMS

Reload the application with the **Suppress load** (`-n`) and **Verify all** (`-c2`) options set on the **ROM-monitor** page of the C-SPY options. This will verify whether the memory contents have changed, or whether the program is self-modifying.

For more information, see the chapters *C-SPY options* and *C-SPY command line options*, respectively.

For problems concerning the operation of the EVB, refer to the documentation supplied with it, or contact your hardware distributor.

POSSIBLE PROBLEMS

The following list gives details of the most common problems.

STOP only works from time to time

Normally, characters are polled from the serial port. But when C-SPY sends the stop command while the application is running, the application must be interrupted.

Sometimes, however, an interrupt is not generated correctly. Then verify that interrupts are enabled in the CPU and that the serial port is set to generate interrupts.

Warnings about write failure during load

ROM-monitors on EVBs, with the exception vectors in ROM, use a remapped interrupt table where your interrupt vectors can be entered. If interrupts are used in the user application and the ROM-monitor uses such a table, the table must be correctly installed. If the exception vectors reside in the ROM-monitor, the vectors must point into the USER RAM-based area where the actual vectors are installed (see the manual for your EVB). Use the **#define INTERRUPT_JUMP_TABLE** option, rebuild the monitor and put the remapped interrupt table at a safe place in RAM.

The application can also be linked to a place where there is no RAM. An incorrect linker command file usually causes this. If you only have a few errors, the probable cause is some interrupt vectors being loaded into ROM. If there are more than just a few errors, your program is probably being loaded into ROM. In either case, give the linker command file a closer look.

No progress in normal code when a periodic interrupt is running

Running with **Calls ON** selected in the **Control** menu (default) will steal a lot of time when C- SPY and the ROM-monitor are communicating. Try running with **Calls OFF** if the application still runs too slow, choosing **Realtime ON** will quicken things up.

Terminal I/O window cannot be opened

The application has probably been linked with a real `putchar()/getchar()`. Use the standard library and remove any special `putchar()/getchar()` from the XLINK command line or `xcl` file. The Terminal I/O window operates with a special version of `putchar()/getchar()` which has its I/O redirected to the Terminal I/O window.

Use the XLINK **Debug info with terminal I/O** (`-rt`) option to enable the Terminal I/O window.

Monitor works, but application won't run

The application is probably linked to some illegal code area (like the interrupt table). Do not use the original `xcl` file delivered with the C compiler without changing the start addresses of CODE and DATA segments to somewhere outside the interrupt table.

Monitor crashes when running application

If the monitor crashes the board will need a reset. This happens when the application writes data to restricted memory areas (like the monitor's RAM or the application code).

Stack overflow or exception vectors that are not properly installed may also be the problem.

No contact with the monitor

If the serial cable is damaged or of the wrong type, C-SPY might use the wrong port or the wrong speed on the computer when trying to communicate with the ROM-monitor.

For more information, see the chapters *C-SPY options* and *C-SPY command line options*, respectively.

The monitor might also have been linked to the wrong address.

The application might also be compiled to use another type of serial port or another address to the serial port on the EVB.

Advanced topics

This chapter describes more advanced use of the ROM-monitor, such as executing transparent commands, adapting the ROM-monitor, switching memory layout, building a new ROM-monitor, and compiling a modified ROM-monitor from the IAR Embedded Workbench.

Executing transparent commands

If the ROM-monitor has been extended with transparent commands (see *Writing transparent commands*, page 266) these can easily be executed. The reason for adding ROM-monitor commands is that some operations are best performed when executed on the board. This way we are not limited by the transfer speed between the computer and the evaluation board (EVB). Let us assume that we have defined the transparent command `stringSearch` *`text`*, returning the addresses where the text is found in memory. This command would take too long to execute if it could only operate through memory reads. By running the command on the EVB, we only have to send the address of each match encountered in memory.

HOW TO EXECUTE A TRANSPARENT COMMAND

Follow these steps to execute a transparent command:

- In C-SPY, select **Quick Watch...** in the **Control** menu.

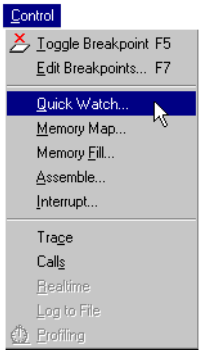
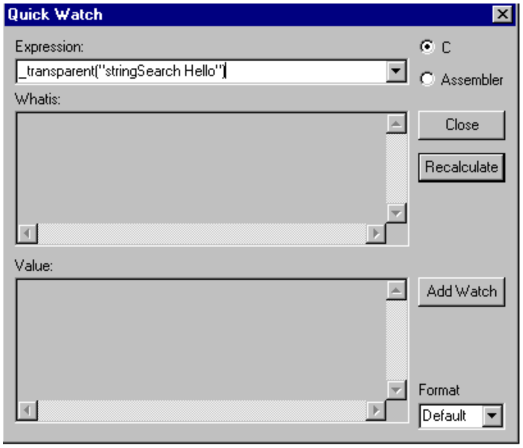


Figure 180: Quick Watch command in the Control menu

2 Use the predefined macro `__transparent(`*`commandstring`*`)` to send your transparent command to the ROM-monitor. Click the **Recalculate** button; the macro will send the string argument as a transparent command to the ROM-monitor where it will be interpreted and executed.



Expression: _transparent("stringSearch Hello")

● C
○ Assembler

Whatis:

Value:

Format: Default

Figure 181: Quick Watch window

The output will be presented in the Report window, where you can examine it by scrolling it up and down.

The `__transparent()` macro can also be called from the C-SPY setup macros defined in a macro file. Note that the macro `execUserInit()` cannot contain the `__transparent()` macro because the communication channel has not been opened. Macros used for initialization can be put in the macro `execUserPreload()` instead, since this macro is called after the communication has been set up.

Writing transparent commands

The ROM-monitor works well as it is with C-SPY, but you may want to add even more functionality to it. You can do this with transparent commands as described in *Executing transparent commands*, page 265.

There are two useful functions, `init_transparent()` and `transparent()` which can be found in the file `transp.c`.

void init_transparent(boolean_R20 cold_start)

This function is called during ROM-monitor initialization. If you have state variables or something that needs to be initialized you must do it explicitly in this function as the ROM-monitor uses a minimized C-startup that does not initialize any variables.

void transparent(unsigned char leave)

Every transparent command typed in C-SPY ends up here. The parameter `leave` will always be `0` when called from the IAR Embedded Workbench. If `leave` differs from `0`, do not try to interpret any command, as there is none. If leave is `0`, there is a transparent command available, you can read it as shown in `transp.c`. How the command is interpreted and handled is your decision.

Output is transmitted to C-SPY using `easy_put_string_R10` and you must end the output by sending the string `#*`.

When writing transparent commands, the following functions in the ROM-monitor can be of interest:

```
boolean_R20 poke_byte_R20(call_address_R20 caddr, char b,
unsigned char type)
```

Writes a byte to memory. `caddr` is the address to write at, `b` is the byte to be written and `type` is the memory type as defined in the file `monitor.h`, you can normally put `Monitor_MEM_TYPE_IRAM` for IDATA memory.

This should return `0`.

```
int peek_byte_R20(call_address_R20 caddr, unsigned char type)
```

Reads a byte from memory and returns it to the caller. If the byte was not readable, `-1` is returned.

CPU registers are kept in a global variable called `cpu_register_R20`, refer to the file `r20.h` for more information.

Addresses in the ROM-monitor are of the type `address_R20` that is a union defined in the file `r20.h`. This union is very small, but as the compiler does not handle small unions as efficiently as integral numbers it is converted to a number when doing function calls. This number is of the type `call_address_R20`. The conversion is done using the `CAST()` macro defined in `r20.h`. This is a simple example function that adds two to an address and passes it to another function:

```
call_address_r20 inc2(call_address_R20 caddr)
{
  address_R20 addr;
  CAST(addr) = caddr;  /* Make addr valid */
  addr.c.c0 += 2;   /* inc 8 bits address */
  bar(CAST(addr));      /* call function */
  return CAST(addr);    /* return incremented address */
}
```

When incrementing addresses, you should use `INC_ADDRESS_R20` defined in the file `r20.h`.

Note that the bytes not used in `caddr` and `addr` are undefined in all monitor functions. If IDATA memory is used, the byte in `addr.c.c1` is undefined.

PROTECTED MEMORY

The ROM-monitor tests memory before writing to it by inverting the bits to make sure that it is writable. Some memory locations, especially memory mapped I/O are vulnerable to such attempts and should be protected. There are three different verification functions in `config.c` for this:

Function	Description
`verify_read_CONFIG`	Use this to read protect C-SPY from reading the memory. Useful if you have created a simple I/O by decoding the bus and a read cause undesirable events. By default all reads are permitted.
`verify_write_CONFIG`	Use this to write-protect memory. Can be useful if you do not want to be able to write to I/O at all from C-SPY or for similar reasons as for the previous function. By default all writes are permitted.
`verify_test_CONFIG`	Use this to prevent the ROM-monitor from testing the memory by inverting the bits. Highly recommended on memory-mapped I/O.

Table 63: Verification functions

USING ADDRESS MASKS

If you are using an incomplete address bus, not all addresses exist and certain addresses are "translated” to other addresses. You can tell the ROM-monitor which address lines are in use for a particular memory type in the function `address_mask_CONFIG`. However, no harm is done if you omit this.

Adapting the ROM-monitor

This section describes how you can adapt the 8051 EVB ROM-monitor to your own hardware.

The first step is to modify your hardware to the ROM-monitor. The entire application must reside in RAM to be debuggable. You may have to modify your hardware so that there is enough RAM.

ROM-MONITOR MEMORY USE

The size of the ROM-monitor is about 13Kbytes, but you should reserve some extra Kbytes for future versions. It will however never be larger than 16Kbytes (use a 16Kbytes EPROM) and can be located anywhere in the `CODE` memory. The ROM-monitor needs 2Kbytes RAM in `XDATA` memory and 16 bytes of user `CODE` memory.

HARDWARE RESET

The ROM-monitor takes control during reset and prepares for running its C code by setting up the stack in the file `conf.s03`. The first function is called `monmain` and resides in the file `r10.c` that is not delivered in source form. `monmain` will immediately call `init_all_CONFIG` that initializes everything.

At reset the 3 first bytes of the ROM-monitor program memory must be mapped to `0000h`. These contain a `LJMP` to the beginning of the ROM-monitor.

At reset the ROM-monitor EPROM on the MCB-517 is also located at `0000h` until the `LJMP` to the right memory location has been executed. As soon as an instruction is read from the right position of the ROM-monitor above `8000h`, the ROM-monitor EPROM at `0000h` is automatically replaced with the user code and XDATA RAM until next reset.

On EVB:s with switched memory like the EM31 ROM-monitor, reset should switch the ROM-monitor EPROM to CODE and user code memory to XDATA.

The ROM-monitors for the KITCON-504, -505 and -515 sets up the memory map after reset. The code inside `#ifdef MONSK5` sets the EVB's control registers.

SERIAL COMMUNICATION

Both the MCB-517 and EM ROM-monitor board use the internal serial port. The functions that handle the communication for the internal port can be found in the file `sc0.c`.

The baud rate for the internal port is defined in the beginning of `sc0.c`. The timer 1 used with the internal serial port cannot generate higher baud rate than 4800 when using a 12.000 MHz crystal. If the crystal is replaced with one on 11.059 MHz speeds of 9600 and 19200 can also be generated.

The serial code resides in the files `sc0.c` for the EM and MCB517, `sc504.c` for the C504, `sc505.c` for C505 and `sc515.c` for the C515 board.

The ROM-monitor comes with all these files for the built-in serial port on these boards. It can use its own receive interrupt vector for handling `Stop` commands. See the example in *Setting the interrupt vector*, page 270.

WRITING YOUR OWN SERIAL PORT DRIVER

If you want to use another serial port than the ones mentioned above, you can write a serial I/O module for your port. Make a copy of `io.c` and rename it to something suitable for your serial port. Add the code needed to initialize the port, read a character, write a character and the interrupt poll routine and possibly some others.

`void init_IO(boolean_R20 cold_start)`
Initializes the serial port to 8 data bits, no parity, one stop bit and the proper speed.

`void forbid_serial_interrupt_IO(void)`
This function is called when the control comes back to the ROM-monitor after a `GO`. You can forget this function unless you need to add code to `permit_serial_interrupt_IO()`.

`void permit_serial_interrupt_IO(void)`
The ROM-monitor uses polled I/O and receive interrupts for `Stop` commands. The ROM-monitor will read all (expected) bytes from the serial port. If the serial port latches the interrupt state and this causes problems for you, you should reset the hanging serial interrupt here before the ROM-monitor gives control to your application. Usually you can leave this function empty, and you can definitely leave it empty when trying to get the ROM-monitor up and running on your serial port.

`void put_char_IO(unsigned char c)`
Waits until it is possible to transmit a character and then transmits the given character.

`int get_char_IO(void)`
Tries to read a character from the serial port. Returns `EOF` if there is no character available at the serial port.

SETTING THE INTERRUPT VECTOR

To set up the interrupt vector for your serial port, proceed as follows. Make sure that the project top node was selected so that all files inherit this definition. If you use the SCI port with the SCI interrupt for the EVB board replace the existing `#define` in ICC8051 option in the **Project** menu to:

```
SERIAL_VECTOR=SERIAL_PORT_INTERRUPT
```

The vector addresses are defined in the file `monitor.h`.

SWITCHED MEMORY LAYOUT

The ROM-monitor also supports switched memory layouts like that on the EM ROM-monitor board. This allows you to have separate CODE and XDATA memory for your application.

When the ROM-monitor is running, it has user code memory always switched to XDATA. This makes reads and writes to user code memory easier to perform. The only thing mapped to CODE memory is the ROM-monitor EPROM. The user XDATA memory is disabled when user CODE memory is in XDATA. When the ROM-monitor wants to access user XDATA memory it temporary switches it back to XDATA. Using a latch can perform the switching by read or write to two different addresses. The latch can, of course, also be at one address, and it can be controlled by writing different values to it.

The example in the file `switch.inc` contains four macros with switching code working with the EM ROM-monitor. The four macros are written to work with two switches using two pairs of latch addresses. Using these switches requires that the `USE_SWITCH` is `#defined`.

The macro `SWITCH_TO_MON_MODE` switches ROM-monitor EPROM to CODE memory to CODE. This macro also contains the macro `SWITCH_TO_MON_XDATA` that switches user code memory to XDATA.

The macro `SWITCH_TO_RUN_MODE` switches user code memory to CODE. This macro also contains the macro `SWITCH_TO_USER_XDATA` that switches user code memory to XDATA.

The memory switches can be used as one or two. In the EM ROM-monitor there is only one hardware latch. It is set or reset by reading two different addresses in CODE memory. The two switches in the file `switch.inc` refers to the same pair of latch addresses.

When the latch is set user code memory is in CODE and user xdata memory is in XDATA. When the latch is reset the ROM-monitor is in CODE and user code memory is in XDATA.

To make it possible to get back to the ROM-monitor from user code or run user code, a part of the ROM-monitor code must always be accessible as CODE.

In the EM ROM-monitor the upper part of the ROM-monitor EPROM is always accessible as CODE. This ROM-monitor code also includes the functions for reading and writing user XDATA memory. If the switch is constructed to replace the whole ROM-monitor EPROM with user RAM, a part of the ROM-monitor must be copied to the user RAM. This is done by using the command `#define USE_COPY_RM_HIGH`.

BUILDING A NEW ROM-MONITOR

Before building the new ROM-monitor, the linker command file must be modified to suit your board.

Modifying the linker command file

The following example explains the most important lines in the linker command file. These are taken from the file `mon517.xcl`.

```
-Z(XDATA)C_ARGX, X_UDATA, X_IDATA, ECSTR, RF_XDATA,
XSTACK=07800
```

The ROM-monitor's 2 Kbytes of RAM are located from `7800h` to `7FFFh` (the last 16 bytes is SFR_PROG):

```
-Z(XDATA) SFR_PROG=07FF0
```

It is very important that `SFR_PROG` is writeable and executable. The ROM-monitor writes some instructions into this memory and then executes them to get access to the SFR registers. If you are using switched memory you should let `SFR_PROG` be at a position in the user CODE memory, which will be in the ROM-monitor's XDATA memory.

```
-Z(XDATA)JUMP_TABLE=8000
```

If `INTERRUPT_JUMP_TABLE` is defined, `JUMP_TABLE` defines where the remapped exception vector table starts.

```
-Z(CODE) INTVEC, RCODE, D_CDATA, I_CDATA, X_CDATA, C_ICALL,
C_RECFN, CSTR, CCSTR, CODE, CONST, CTABLE=8000
```

The ROM-monitor ROM starts at `8000h` (with a size of approx 13Kbytes).

The code of the segment `RM_HIGH` of the assembler routines in the file `conf.s03`, placed somewhere in the end of the EPROM, is used to get back to ROM-monitor. This code must always be accessable as CODE.

```
-Z(CODE) RM_HIGH=0BF00
```

The two pairs of addresses read by the ROM-monitor when switching memory.

No program code should be executed at these addresses if you use the type of memory switches written in the file `switch.inc`. The switching is not used with MCB-517, and the addresses are set to any value.

```
-DSW_MON_XDATA=0F000
-DSW_MON_CODE=0F000
-DSW_USER_XDATA=0F800
-DSW_USER_XDATA=0F800
```

If you are not using switched memory the macros in the file `switch.inc` can be excluded by not defining `USE_SWITCH`. The macros in `switch.inc` can be used anyway without any problems.

If you have an external serial communication chip, make sure its addresses do not overlap your `XDATA` RAM memory. If you are using switched memory you can use the switch to disable these addresses when running user code.

Compiling from the command line

Use the file `mkmon.bat` to compile and link the ROM-monitor. The new ROM-monitor will reside in the file `mon.a03`.

Compiling in the IAR Embedded Workbench

If the ROM-monitor has been modified in any way—for example, with a new serial communication routine or added transparent commands—it has to be re-built. For this purpose, a project file template has been included to reduce the amount of work involved. The name of this template is `mon.prj`; it is found in the `\src\rom` directory.

The project file template includes the files that typically need updating when the ROM-monitor is changed. Some minor modifications to the compiler and linker options are however required.

Open the predefined project file `mon.prj` provided in the `\8051\src\rom\` directory. This project includes the files required for the supported boards according to the following table:

Board	Files

C504	`transp.c`
	`config.c`
	`sc504.c`
	`conf.s03`

C505	`transp.c`
	`config.c`
	`sc505.c`
	`conf.s03`

C515	`transp.c`
	`config.c`
	`sc515.c`
	`conf.s03`

Table 64: Required files for ROM-monitor configuration

Board	Files

C541	`transp.c`
	`config.c`
	`sc541.c`
	`conf.s03`

EM31	`transp.c`
	`config.c`
	`sc0.c`
	`conf.s03`

MCB517	`transp.c`
	`config.c`
	`sc0.c`
	`conf.s03`

Table 64: Required files for ROM-monitor configuration (continued)

In this example we will assume that the board to be used is the PHYTEC C504 board. Save the project under a new name, for example `mymon504.prj`.

Select **mon504** in the **Target** check box to display the project options to see what is predefined and what has to be changed before building the ROM-monitor.

Make sure that the project-tree top node is selected, so that the changes will affect all project files:

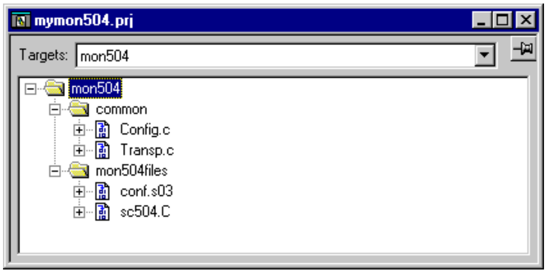
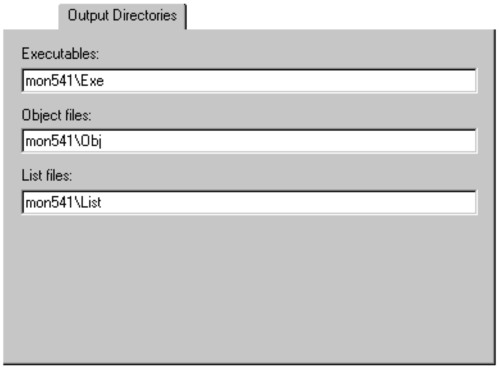


Figure 182: The mymon504 project

Open the options dialog by selecting **Options...** from the **Project** menu.

Start by looking at the category **General** to see what has to be changed.

On the **Output Directories** page, you will notice that the output will be located in the subdirectories of the `mon504` directory:



Executables: mon541\Exe

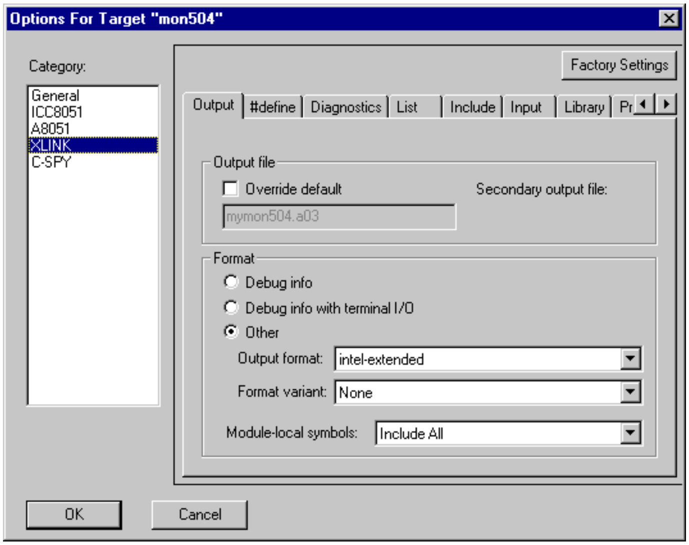
Object files: mon541\Obj

List files: mon541\List

Figure 183: The mon504 output directories

Change to the category **ICC8051** to see what is specific for the compiler. In the **Code Generation** page the size optimization level has been set to 9 (max). Leave it that way.

Select the **XLINK** category and the **Output** page. Select **Intel-extended** from the **Output format** drop-down list to generate code for your ROM-monitor:



☐ Override default

mymon504.a03

○ Debug info

○ Debug info with terminal I/O

● Other

Output format: intel-extended

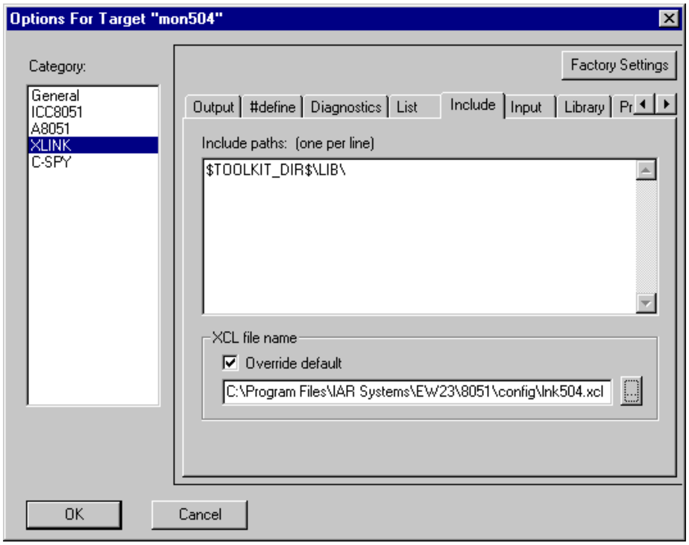
Format variant: None

Module-local symbols: Include All

Figure 184: Output page in XLINK options

Continue to the **Include** page in the **XLINK** category. Since the ROM-monitor has its preferred location, we must specify an `xcl` file written especially for the board in use.

Check the **Override default** box and browse your way to the linker command file that came with the 8051 ROM-monitor. In this example the `lnk504.xcl` is used:



Options For Target "mon504"

Category:

- General
- ICC8051
- A8051
- XLINK
- C-SPY

Include paths: (one per line)

$TOOLKIT_DIR$\LIB\

XCL file name

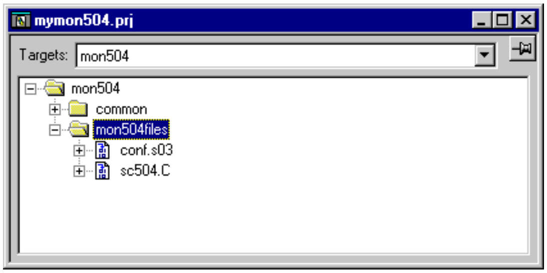
☑ Override default

C:\Program Files\IAR Systems\EW23\8051\config\lnk504.xcl

Figure 185: Specifying a linker command file for the ROM-monitor

Then click **OK** to save your settings.

In the project window, select the **mon504files** group to view the symbols that have been defined for theses files:



mymon504.prj

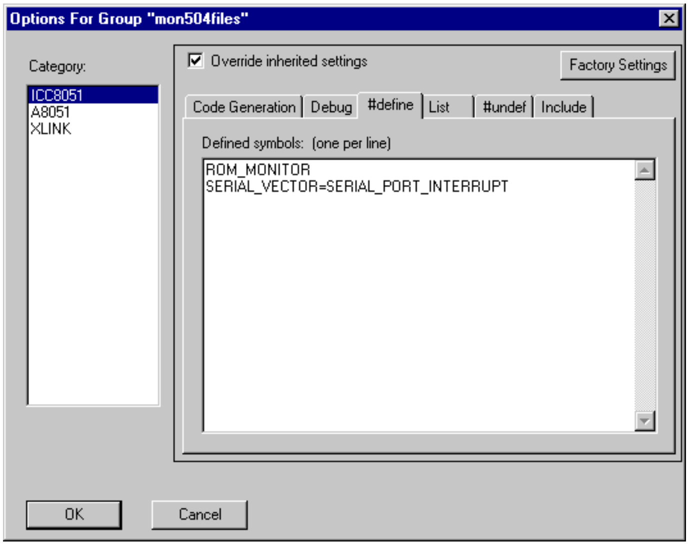
Targets: mon504

- mon504
  - common
  - mon504files
    - conf.s03
    - sc504.C

Figure 186: The mon504files group

Then select **Options** from the **Project** menu. In the **ICC8051** category, display the **#define** page. The following symbols should be defined or the ROM-monitor may stop working:

```
ROM_MONITOR
SERIAL_VECTOR=SERIAL_PORT_INTERRUPT
```



Defined symbols: (one per line)

```
ROM_MONITOR
SERIAL_VECTOR=SERIAL_PORT_INTERRUPT
```

Figure 187: Predefined symbols for the ROM-monitor

Select the **#define** page in the **A8051** category. The following symbols should be defined for the assembler:

```
SERIAL_VECTOR=SERIAL_PORT_INTERRUPT
MONSK5
```

Do not change these definitions, or it is likely that your ROM-monitor will stop working. Click **OK** to exit the dialog box.

Select **Make** from the **Project** menu to get the PROMable ROM-monitor. If you get warnings about labels or variables never being referenced, you can ignore them.

TESTING THE MODIFIED SERIAL COMMUNICATIONS

Compile and link the ROM-monitor with `LOOP_BACK` defined for the file `config.c`. You may find it easiest to remove the comments from `#define LOOP_BACK` early in `config.c`.

When your modified ROM-monitor compiles and links without errors, inspect the link map and make sure that it is all right. Download the ROM-monitor (`mon.a03`) to you board (burn an EPROM or emulate one) and connect some kind of terminal to it.

When your terminal is connected and set up properly to match the serial port of the ROM-monitor, reset the ROM-monitor. A single hash mark `#` should appear on the screen. Try typing on the terminal and the characters sent back should have their ASCII code incremented by 1, i.e. an `a` will come back as a `b`.

If the hash mark does not appear, use a logic probe, oscilloscope or interface tester to see if the ROM-monitor transmits something at reset. If it does, check the cable and make sure that the serial ports have been set up to match each other.

If the ROM-monitor does not send any characters at all, check the link file, `init_IO()`, `put_char_IO()` and your hardware.

If the hash mark appears but characters are not echoed back incremented by 1, check `get_char_IO()`. If the ROM-monitor echoes the same character back, the most likely problem is that you have local echo on your terminal. In that case, turn it off.

When your serial port code works, put the comment back around `LOOP_BACK` and recompile the ROM-monitor. Download the new ROM-monitor and enable the log file. The ROM-monitor should send 4 bytes every second, `C0 1A F5 C0` (hex). One way to supervise the communication is through a log file, which can be generated by selecting **Options...** from the **Project** menu and checking the **Log Communication** box on the **Serial communication** page. The file can then be examined in order to see what was sent.

This sequence is a connect sequence from the ROM-monitor that is sent when the ROM-monitor looks for C-SPY. C-SPY sends the same sequence when looking for the ROM-monitor.

If this sequence does not appear, the most likely cause is that you have placed a part of the ROM-monitor RAM area outside of RAM.

Connect C-SPY to the ROM-monitor and download the demo file `demo.d03` and verify that the ROM-monitor works as it should.

If you are planning to use the HOST PORT CONNECTOR on the board (connected to the internal serial port), the RxD and TxD pins are (and always have been) swapped compared to the usual connector (TERMINAL PORT CONNECTOR).

The ROM-monitor boards

Understanding the memory layout is important to avoid conflicts between the ROM-monitor and your own programs. This chapter gives an overview of the different boards available.

EM ROM-monitor

The EM ROM-monitor board has two memory layouts between which the ROM-monitor switches. One of them is used when user code is running (RUN mode) and one when the ROM-monitor is running (MONitor mode).

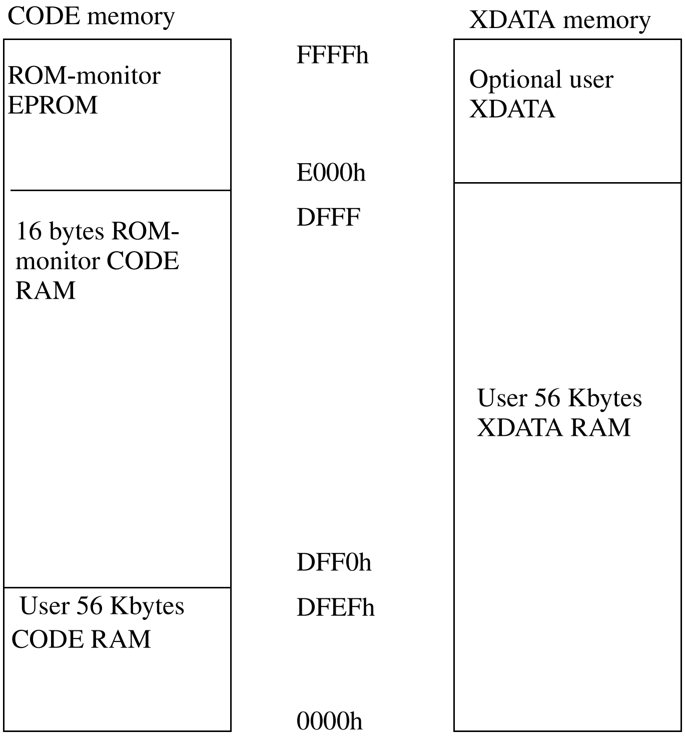
MEMORY MAPS IN RUN MODE

You need to know the RUN mode memory layout when you are linking your applications. The first 56 Kbytes of CODE memory can be used, except the last 16 bytes of these 56 Kbytes.

One of the CPU's interrupt vectors is used by the ROM-monitor for serial interrupt (when the **STOP** button is pressed). This vector is put into the user code memory by the ROM-monitor before running it. All other interrupt vectors can be used by your application.

The first 56 Kbytes of the XDATA memory is RAM, the remaining 8 Kbytes is not used.

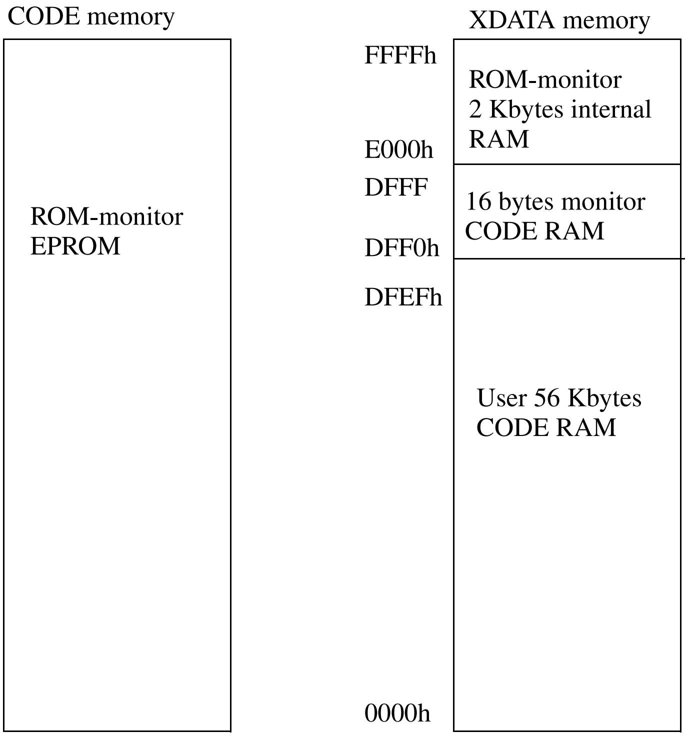
The part of the ROM-monitor, above 56 Kbytes, is also visible in RUN mode. This part contains the entrance to the ROM-monitor from a breakpoint or interrupt and the return from the ROM-monitor to user code.


CODE memory
XDATA memory
ROM-monitor EPROM
FFFFh
Optional user XDATA
E000h
DFFF
16 bytes ROM-monitor CODE RAM
User 56 Kbytes XDATA RAM
DFF0h
User 56 Kbytes CODE RAM
DFEFh
0000h

MEMORY MAPS IN MONITOR MODE

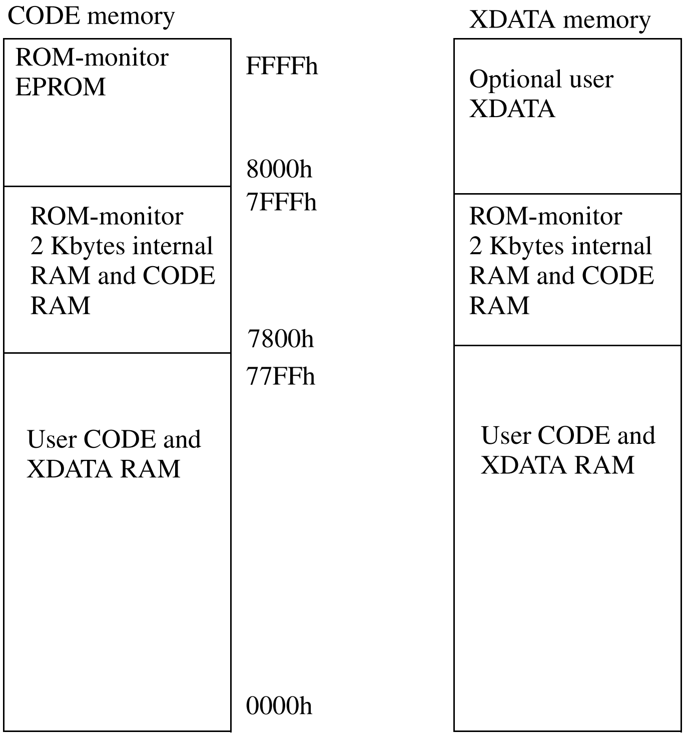
In monitor mode the ROM-monitor is mapped to all addresses in CODE memory. A 32 Kbytes EPROM will be at two places, a 16 Kbytes EPROM at four. The real ROM-monitor starts at `8000h` in the 32 Kbytes EPROM. The first instruction at `8000h`, a `LJMP` to `8003h`, will also be found as a reset vector at `0000h`.

In monitor mode user code memory is connected to XDATA. Above 56 Kbytes of XDATA there is RAM containing the ROM-monitor's internal data.



MCB-517 ROM-MONITOR

The MCB-517 board has only one memory layout. The same memory maps are used both when the ROM-monitor is running and when the user code is running.



The `XDATA` and `CODE` memory between `0000h-7FFFh` are connected to the same RAM. The first 30 Kbytes of `CODE` and `XDATA` are available for your application and the optional 32 Kbytes XDATA at `8000h`.

The 2 Kbytes RAM between `7800H-7FFFh` contains the ROM-monitor's internal data. Writes to this area may cause a ROM-monitor crash.

The ROM-monitor program starts at `8000h` in `CODE` memory. The first instruction at `8000h` is an LJMP instruction to `8003h`. At reset the EPROM resides at `0000h`. As soon as the LJMP has been executed and the CPU reads the next instruction above `8000h`, the lower 32 Kbytes will be replaced with RAM.

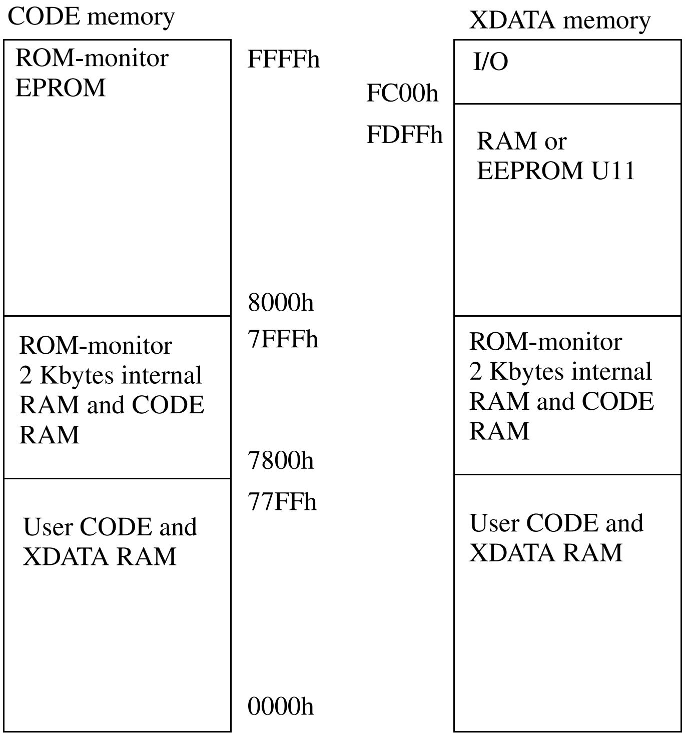
KITCON-504C, 505C AND 515C ROM-MONITOR

The KitCON-5xxC boards have only one memory layout. The same memory maps are used both when the ROM-monitor is running and when the user code is running. The `XDATA` and `CODE` memory between `0000h-7FFFh` are connected to the same RAM. The first 30 Kbytes of `CODE` and `XDATA` are available for your application and the optional 32 Kbytes `XDATA` at `8000h`.

The 2 Kbytes RAM between `7800h-7FFFh` contains the ROM-monitor's internal data. Writes to this area may cause a ROM-monitor crash.

The ROM-monitor program starts at `0000h` in CODE memory. The first instruction at `0000h` is an `LJMP` to `8003h`. At reset the FLASH PROM resides at `0000h-FFFFh`. As soon as the `LJMP` has been executed the following instructions will change the memory layout and the lower 32 Kbytes will be replaced with RAM.

The memory layout will be different depending on the way you set the EVB control registers in (see the file `conf.s03` in the `cw8051r` directory) and how you linked the monitor (see the `xcl` file) (see the KitCON-5xxC SIEMENS manuals to find out how to set these registers). Our setting method in `conf.s03` gives the memory layout shown above.



INTERNAL CPU MEMORY

The ROM-monitor needs only 6 bytes of the stack in internal memory. This DATA or IDATA is used when breakpoints or interrupts are encountered. Before using more of this memory the ROM-monitor saves the internal memory into its working area.

Diagnostic messages

This chapter lists the error and warning messages that the 8051 C-SPY ROM-monitor version can produce.

Warning messages

The warning messages of the ROM-monitor are referenced to by the number 40. The following table lists the warning messages:

0 **Memory write prevented by verify_write_CONFIG**
A memory write was prevented by the function `verify_write_CONFIG` in the file `config.c`.

1 **Memory read prevented by verify_read_CONFIG**
A memory read was prevented by the function `verify_read_CONFIG` in the file `config.c`.

2 **Fast download failed -- trying normal download**
The ROM-monitor is not quick enough to handle a constant stream of data and C-SPY switched to normal (slower) writes. You can disable fast download using the `-rz` switch on the command line. Hardware handshake or lowering the speed can also help. You should do something to get rid of this warning as the downloads go slower than they should.

3 **Memory read failed at address XXXX**
If the given address should be correct, check your hardware.

4 **Memory write failed at address XXXX**
If the given address should be correct, check your hardware. You have probably linked your application incorrectly if this happens during download.

5 **Memory write not recorded by memory**
No memory seems to be connected to this address.

6 **Breakpoint set failed at address XXXX**
If the given address should be correct, check your hardware. Your application must reside in RAM. Otherwise C-SPY will not be able to step it at assembly or C level. C-SPY also inserts breakpoints when you run with calls on.

7 **Failed to write SFR access instruction Into user CODE memory**
The ROM-monitor SFR_PROG segment is linked to wrong location or the hardware is wrong configured.

Error messages

The error messages of the ROM-monitor are referenced to by the number 130. The following table lists the error messages:

0 **Unable to step ROM**
You can only step in RAM memory since the ROM-monitor must patch in breakpoints when single stepping.

1 **Stack points outside writable memory**
The stack pointer has to be set up properly to read/writable memory or there is no use in trying to run the code. A temporary stack is allocated by the ROM-monitor for the application. This stack is very small so the first thing the application must do is to set up the stack properly.

2 **Instruction jumps to itself**
The ROM-monitor inserts a breakpoint at the location where the instruction ends. In this case the instruction points to itself and the ROM-monitor cannot execute the instruction.

3 **PC at illegal instruction**
Tried to step an illegal instruction.

Fatal error messages

The fatal error messages of the ROM-monitor are referenced to by the number 60. The following table lists the fatal error messages:

0 **Protocol version mismatch**
You are mixing a new C-SPY with an old ROM-monitor or vice versa.

1 **Illegal target ROM-monitor**
Use the correct C-SPY with the ROM-monitor.

2 **Unable to get extended error**
A protocol error occurred. Reset the ROM-monitor and restart C-SPY.

3 **Unable to read ROM-monitor memory**
A protocol error occurred. Reset the ROM-monitor and restart C-SPY.

4 **Unable to write registers**
A protocol error occurred. Reset the ROM-monitor and restart C-SPY.

5 **Unable to read registers**
A protocol error occurred. Reset the ROM-monitor and restart C-SPY.

6 **Unable to get protocol version**
A protocol error occurred. Reset the ROM-monitor and restart C-SPY.

7 **Unable to get Monitor status**
A protocol error occurred. Reset the ROM-monitor and restart C-SPY.

8 **Unable to get Sign on message**
A protocol error occurred. Reset the ROM-monitor and restart C-SPY.

9 **Transparent command failed**
A protocol error occurred. Reset the ROM-monitor and restart C-SPY. If you have added new transparent commands check the file `transp.c`.

10 **C-SPY address size not supported by Monitor**
You are running C-SPY in banked mode but the ROM-monitor does not support this.

11 **Failed to adjust Monitor address size**
A protocol error occurred. Reset the ROM-monitor and restart C-SPY.

12 **Unable to get address mask**
A protocol error occurred. Reset the ROM-monitor and restart C-SPY.