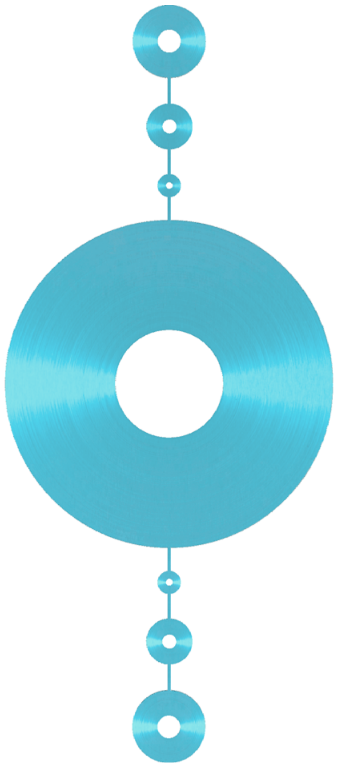
IAR Embedded Workbench®

C-SPY® Debugging Guide

for the 8051 Microcontroller Architecture



UCS8051-1

IAR SYSTEMS

EDITION NOTICE

First edition: April 2011

Part number: UCS8051-1

This guide applies to version 8.x of IAR Embedded Workbench® for 8051.

The *C-SPY® Debugging Guide for 8051* replaces all debugging information in the *IAR Embedded Workbench IDE User Guide* and the hardware debugger guides for 8051.

Internal reference: M9, Too6.1, tut2009.1, IJOA.

Brief contents

Contents

Tables

Figures

Preface

Welcome to the *C-SPY® Debugging Guide for 8051*. The purpose of this guide is to help you fully use the features in the IAR C-SPY® Debugger for debugging your application based on the 8051 microcontroller architecture.

Who should read this guide

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

- The C or C++ programming language
- Application development for embedded systems
- The architecture and instruction set of the 8051 microcontroller (refer to the chip manufacturer's documentation)
- The operating system of your host computer.

For more information about the other development tools incorporated in the IDE, refer to their respective documentation, see *Other documentation*, page 23.

How to use this guide

If you are new to using IAR Embedded Workbench, we suggest that you first read the guide *Getting Started with IAR Embedded Workbench®* for an overview of the tools and the features that the IDE offers.

If you already have had some experience using IAR Embedded Workbench, but need refreshing on how to work with the IAR Systems development tools, the tutorials which you can find in the IAR Information Center is a good place to begin. The process of managing projects and building, as well as editing, is described in the *IDE Project Management and Building Guide*, whereas information about how to use C-SPY for debugging is described in this guide.

This guide describes a number of *topics*, where each topic section contains an introduction which also covers concepts related to the topic. This will give you a good understanding of the features in C-SPY. Furthermore, the topic section provides procedures with step-by-step descriptions to help you use the features. Finally, each topic section gives all relevant reference information.

We also recommend the Glossary which you can find in the *IDE Project Management and Building Guide* if you should encounter any unfamiliar terms in the IAR Systems user and reference guides.

What this guide contains

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

- *The IAR C-SPY Debugger* introduces you to the C-SPY debugger and to the concepts that are related to debugging in general and to C-SPY in particular. The chapter also introduces the various C-SPY drivers. The chapter briefly shows the difference in functionality that the various C-SPY drivers provide.
- *Getting started using C-SPY* helps you get started using C-SPY, which includes setting up, starting, and adapting C-SPY for target hardware.
- *Executing your application* describes the conceptual differences between source and disassembly mode debugging, the facilities for executing your application, and finally, how you can handle terminal input and output.
- *Working with variables and expressions* describes the syntax of the expressions and variables used in C-SPY, as well as the limitations on variable information. The chapter also demonstrates the various methods for monitoring variables and expressions.
- *Using breakpoints* describes the breakpoint system and the various ways to set breakpoints.
- *Monitoring memory and registers* shows how you can examine memory and registers.
- *Collecting and using trace data* describes how you can inspect the program flow up to a specific state using trace data.
- *Using the profiler* describes how the profiler can help you find the functions in your application source code where the most time is spent during execution.
- *Code coverage* describes how the code coverage functionality can help you verify whether all parts of your code have been executed, thus identifying parts which have not been executed.
- *Simulating interrupts* contains detailed information about the C-SPY interrupt simulation system and how to configure the simulated interrupts to make them reflect the interrupts of your target hardware.
- *Using C-SPY macros* describes the C-SPY macro system, its features, the purposes of these features, and how to use them.
- *The C-SPY Command Line Utility—cspybat* describes how to use C-SPY in batch mode.

- *Debugger options* describes the options you must set before you start the C-SPY debugger.
- *Additional information on C-SPY drivers* describes menus and features provided by the C-SPY drivers not described in any dedicated topics.
- *Target-adapting the ROM-monitor* describes how you can easily adapt the generic ROM-monitor provided with IAR Embedded Workbench to suit a device that does not have an existing debug solution supported by IAR Systems.

Other documentation

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

USER AND REFERENCE GUIDES

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

- System requirements and information about how to install and register the IAR Systems products, refer to the booklet *Quick Reference* (available in the product box) and the *Installation and Licensing Guide*.
- Getting started using IAR Embedded Workbench and the tools it provides, see the guide *Getting Started with IAR Embedded Workbench®*
- Using the IDE for project management and building, see the *IDE Project Management and Building Guide*
- Programming for the IAR C/C++ Compiler for 8051, see the *IAR C/C++ Compiler Reference Guide for 8051*
- Using the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian, see the *IAR Linker and Library Tools Reference Guide.*
- Programming for the IAR Assembler for 8051, see the *IAR Assembler Reference Guide for 8051.*
- Using the IAR DLIB Library, see the *DLIB Library Reference information*, available in the online help system.
- Using the IAR CLIB Library, see the *IAR C Library Functions Reference Guide*, available in the online help system.
- Porting application code and projects created with a previous version of the IAR Embedded Workbench for 8051, see the *IAR Embedded Workbench® migration guides for 8051.*

- Developing safety-critical applications using the MISRA C guidelines, see the *IAR Embedded Workbench® MISRA C:2004 Reference Guide* or the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*.

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

THE ONLINE HELP SYSTEM

The context-sensitive online help contains:

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

WEB SITES

Recommended web sites:

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

Document conventions

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

When referring to a directory in your product installation, for example `8051\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench 6.`*`n`*`\8051\doc`.

TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:

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

Table 1: Typographic conventions used in this guide

NAMING CONVENTIONS

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

Brand name	Generic term
IAR Embedded Workbench® for 8051	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for 8051	the IDE
IAR C-SPY® Debugger for 8051	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for 8051	the compiler

Table 2: Naming conventions used in this guide

Brand name	Generic term
IAR Assembler™ for 8051	the assembler
IAR XLINK Linker™	XLINK, the linker
IAR XAR Library Builder™	the library builder
IAR XLIB Librarian™	the librarian
IAR DLIB Library™	the DLIB library
IAR CLIB Library™	the CLIB library

Table 2: Naming conventions used in this guide (Continued)

In this guide, *8051 microcontroller* refers to all microcontrollers compatible with the 8051 microcontroller architecture.

The IAR C-SPY Debugger

This chapter introduces you to the IAR C-SPY® Debugger and to the concepts that are related to debugging in general and to C-SPY in particular. The chapter also introduces the various C-SPY drivers. More specifically, this means:

- Introduction to C-SPY
- Debugger concepts
- C-SPY drivers overview
- The IAR C-SPY Simulator
- The C-SPY hardware debugger drivers.

Introduction to C-SPY

This section covers these topics:

- An integrated environment
- General C-SPY debugger features
- RTOS awareness.

AN INTEGRATED ENVIRONMENT

C-SPY is a high-level-language debugger for embedded applications. It is designed for use with the IAR Systems compilers and assemblers, and is completely integrated in the IDE, providing development and debugging within the same application. This will give you possibilities such as:

- Editing while debugging. During a debug session, you can make corrections directly in the same source code window that is used for controlling the debugging. Changes will be included in the next project rebuild.
- Setting breakpoints at any point during the development cycle. You can inspect and modify breakpoint definitions also when the debugger is not running, and breakpoint definitions flow with the text as you edit. Your debug settings, such as watch properties, window layouts, and register groups will be preserved between your debug sessions.

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

GENERAL C-SPY DEBUGGER FEATURES

Because IAR Systems provides an entire toolchain, the output from the compiler and linker can include extensive debug information for the debugger, resulting in good debugging possibilities for you.

C-SPY offers these general features:

- Source and disassembly level debugging

  C-SPY allows you to switch between source and disassembly debugging as required, for both C or C++ and assembler source code.
- Single-stepping on a function call level

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

  The C-SPY breakpoint system lets you set breakpoints of various kinds in the application being debugged, allowing you to stop at locations of particular interest. For example, you set breakpoints to investigate whether your program logic is correct or to investigate how and when the data changes.
- Monitoring variables and expressions

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

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

  The compiler generates extensive call stack information. This allows the debugger to show, without any runtime penalty, the complete stack of function calls wherever the program counter is. You can select any function in the call stack, and for each function you get valid information for local variables and available registers.

- Powerful macro system

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

Additional general C-SPY debugger features

This list shows some additional features:

- Threaded execution keeps the IDE responsive while running the target application
- Automatic stepping
- The source browser provides easy navigation to functions, types, and variables
- Extensive type recognition of variables
- Configurable registers (CPU and peripherals) and memory windows
- Graphical stack view with overflow detection
- Support for code coverage and function level profiling
- The target application can access files on the host PC using file I/O
- UBROF, Intel-extended, and Motorola input formats supported
- Optional terminal I/O emulation.

RTOS AWARENESS

C-SPY supports real-time OS aware debugging.

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

Debugger concepts

This section introduces some of the concepts and terms that are related to debugging in general and to C-SPY in particular. This section does not contain specific information related to C-SPY features. Instead, you will find such information in each chapter of this part of the documentation. The IAR Systems user documentation uses the terms described in this section when referring to these concepts.

C-SPY AND TARGET SYSTEMS

You can use C-SPY to debug either a software target system or a hardware target system.

This figure gives an overview of C-SPY and possible target systems:

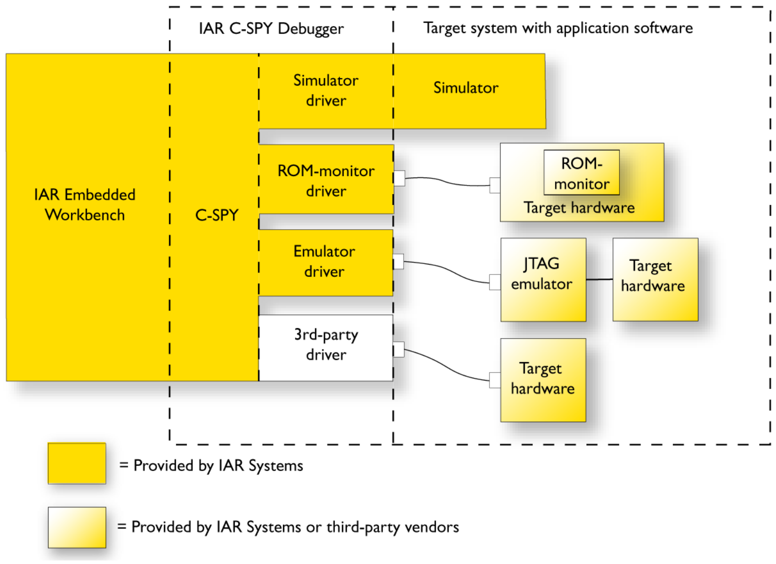


Figure 1: C-SPY and target systems

THE DEBUGGER

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

THE TARGET SYSTEM

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

THE APPLICATION

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

C-SPY DEBUGGER SYSTEMS

C-SPY consists of both a general part which provides a basic set of debugger features, and a target-specific back end. The back end consists of two components: a processor module—one for every microcontroller, which defines the properties of the microcontroller, and a *C-SPY driver*. The C-SPY driver is the part that provides communication with and control of the target system. The driver also provides the user interface—menus, windows, and dialog boxes—to the functions provided by the target system, for instance, special breakpoints. Typically, there are three main types of C-SPY drivers:

- Simulator driver
- ROM-monitor drivers
- Emulator drivers.

C-SPY is available with a simulator driver, and depending on your product package, optional drivers for hardware debugger systems. For an overview of the available C-SPY drivers and the functionality provided by each driver, see *C-SPY drivers overview*, page 32.

THE ROM-MONITOR PROGRAM

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

THIRD-PARTY DEBUGGERS

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

C-SPY PLUGIN MODULES

C-SPY is designed as a modular architecture with an open SDK that can be used for implementing additional functionality to the debugger in the form of plugin modules. These modules can be seamlessly integrated in the IDE.

Plugin modules are provided by IAR Systems, or can be supplied by third-party vendors. Examples of such modules are:

- Code Coverage and the Stack window, both integrated in the IDE.
- The various C-SPY drivers for debugging using certain debug systems.
- RTOS plugin modules for support for real-time OS aware debugging.
- C-SPYLink that bridges IAR visualSTATE and IAR Embedded Workbench to make true high-level state machine debugging possible directly in C-SPY, in addition to

the normal C level symbolic debugging. For more information, refer to the documentation provided with IAR visualSTATE.

For more information about the C-SPY SDK, contact IAR Systems.

C-SPY drivers overview

At the time of writing this guide, the IAR C-SPY Debugger for the 8051 microcontrollers is available with drivers for these target systems and evaluation boards:

- Simulator
- Texas instruments CCxxxx evaluation boards and the CC debugger
- FS2 System Navigator for CAST 8051, Mentor Graphics M8051EW, and processors from Handshake Solutions and NXP Semiconductors
- Infineon's DAS (Device Access Server) protocol for debugging all XC8xx devices
- Nordic Semiconductor's nRFGo development platform
- IAR ROM-monitor (including prebuilt ROM-monitors for NXP 93x, Analog Devices ADu 84x, and Texas Instruments MSC 1211 evaluation boards and a template project for building your own ROM-monitor)
- Analog Devices' ADuC8xx and ADe development boards
- Silicon Laboratories' USB and serial debug adapters for C8051Fxxx MCUs.

Note: In addition to the drivers supplied with the IAR Embedded Workbench, you can also load debugger drivers supplied by a third-party vendor; see *Third-Party Driver*, page 283.

DIFFERENCES BETWEEN THE C-SPY DRIVERS

This table summarizes the key differences between the C-SPY drivers:

Feature	Simulator	TI	FS2	Infineon	Nordic Semi	ROM-monitor	Analog Devices	Silabs
Code breakpoints	Unlimited	4	x[1,3]	4	x[2]	x[3]	Unlimited	4
Data breakpoints	Unlimited	—	x[1,3]	x	—	x[3]	—	x
Execution in real time	—	x	x	x	x	x	x	x
Zero memory footprint	x	x	x[4]	x[4]	x	—	x[4]	x
Simulated interrupts	x	—	—	—	—	—	—	—
Real interrupts	—	x	x	x	x	x	x	x
Interrupt logging	x	—	—	—	—	—	—	—

Table 3: Driver differences

Feature	Simulator	TI	FS2	Infineon	Nordic Semi	ROM-monitor	Analog Devices	Silabs
Live watch	—	—	—	—	—	—	—	—
Cycle counter	x	—	—	—	—	—	—	—
Code coverage	x	—	—	—	—	—	—	—
Data coverage	x	—	—	—	—	—	—	—
Function/instruction profiler	x	—	—	—	—	—	—	—
Trace	x	—	—	—	—	—	—	—

Table 3: Driver differences (Continued)

1 The data breakpoints are shared with the fetch hardware breakpoints.
2 Hardware breakpoints only.
3 Depends on the target system.
4 Only when software breakpoints are disabled.

The IAR C-SPY Simulator

The C-SPY Simulator simulates the functions of the target processor entirely in software, which means that you can debug the program logic long before any hardware is available. Because no hardware is required, it is also the most cost-effective solution for many applications.

FEATURES

In addition to the general features in C-SPY, the simulator also provides:

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

SELECTING THE SIMULATOR DRIVER

Before starting C-SPY, you must choose the simulator driver:

1. In the IDE, choose **Project>Options** and click the **Setup** tab in the **Debugger** category.
2. Choose **Simulator** from the **Driver** drop-down list.

The C-SPY hardware debugger drivers

The C-SPY hardware debugger drivers are automatically installed during the installation of IAR Embedded Workbench.

Using the C-SPY hardware debugger driver as an interface, C-SPY can connect to the third-party hardware debugger. Many of the 8051 microcontrollers have built-in, on-chip debug support. Because the hardware debugger logic is built into the microcontroller or located within the boot loader, no ordinary ROM-monitor program or extra specific hardware is needed to make the debugging work.

When a debugging session is started, your application is automatically downloaded and programmed into flash memory. You can disable this feature, if necessary.

INSTALLING EXTRA SOFTWARE

For these drivers, you might need to install extra software:

- Texas Instruments
- FS2 System Navigator
- Infineon
- Nordic Semiconductor
- Analog Devices.

See the release notes for these drivers, available from the Information Center.

COMMUNICATION OVERVIEW

There are two main communication setups, depending on the type of target system.

Overview of a target system with a debug probe

Some target systems have an emulator, a debug probe or a debug adapter connected between the host computer and the evaluation board:

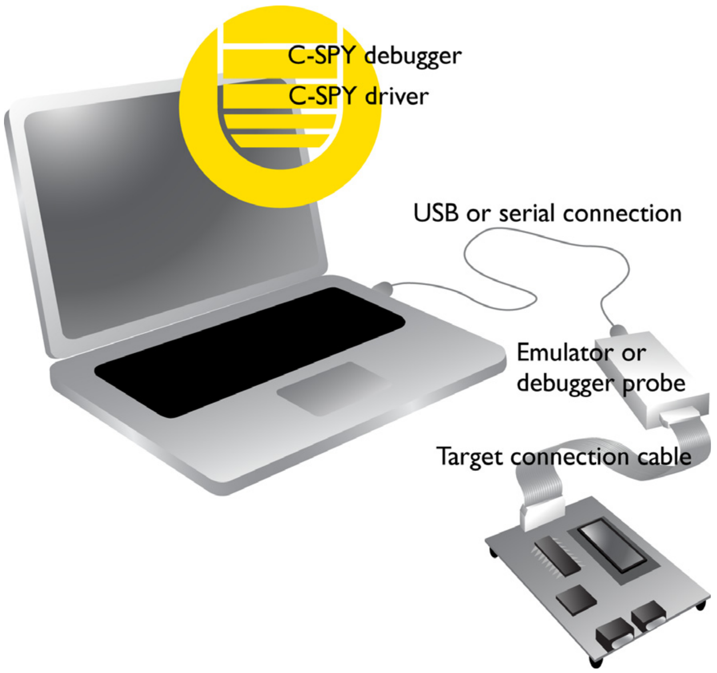


Figure 2: C-SPY driver communication overview with a debug probe

Overview of a target system without a debug probe

Some target systems have all debugger functionality located on the target board itself:

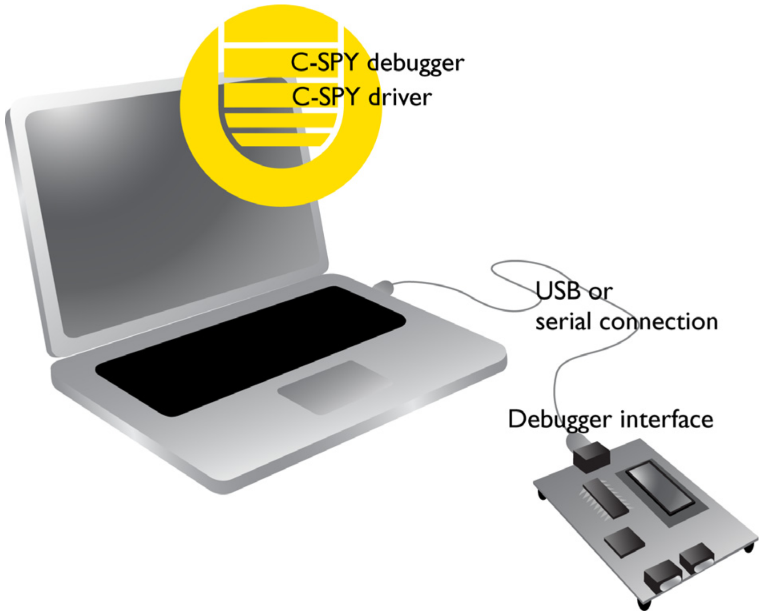


Figure 3: C-SPY driver communication overview without a debug probe

HARDWARE INSTALLATION

For information about the hardware installation, see the documentation supplied with the target system from the manufacturer. The following power-up sequence is recommended to ensure proper communication between the target board, the emulator or debug probe (if there is one), and C-SPY:

1. Power up the target board.
2. Power up the emulator or debug probe, if there is one.
3. Start the C-SPY debugging session.

THE C-SPY INFINEON DRIVER

The C-SPY Infineon driver works as an interface to the DAS server from Infineon. The C-SPY driver connects to a DAS server, which in turn connects to the target system.

THE IAR C-SPY ROM-MONITOR DRIVER

There are still devices that lack on-chip debug support. For these, a ROM-monitor provides a working debug solution. It has a small memory footprint, it occupies only 4 Kbytes of non-volatile memory and uses 256 bytes in xdata memory and 5–7 bytes in idata memory.

IAR Embedded Workbench comes with a set of ready-made ROM-monitors for some devices. In addition, a generic ROM-monitor framework is provided, which you can adapt for your own target board.

Before you can use the IAR C-SPY ROM-monitor driver, you must make sure that ROM-monitor firmware is located on the target board. A template for creating firmware is available from the **Create New Project** dialog box. There are firmware images for some devices in the `8051\src\rom\monitor_image` directory. The source code for these images is located in the `8051\src\rom` directory.

A set of ready-made ROM-monitors for some devices are located in the directory `8051\src\rom`. For a generic ROM-monitor framework which you can adapt for your own target board, see the chapter *Target-adapting the ROM-monitor*, page 293.

Getting started using C-SPY

This chapter helps you get started using C-SPY®. More specifically, this means:

- Setting up C-SPY
- Starting C-SPY
- Adapting for target hardware
- Running example projects
- Reference information on starting C-SPY.

Setting up C-SPY

This section describes the steps involved for setting up C-SPY.

More specifically, you will get information about:

- Setting up for debugging
- Executing from reset
- Using a setup macro file
- Selecting a device description file
- Loading plugin modules.

SETTING UP FOR DEBUGGING

1. Before you start C-SPY, choose **Project>Options>Debugger>Setup** and select the C-SPY driver that matches your debugger system: simulator or a hardware debugger system.
2. In the **Category** list, select the appropriate C-SPY driver and make your settings.

   For information about these options, see *Debugger options*, page 261.
3. Click **OK**.

4 Choose **Tools>Options>Debugger** to configure:

- The debugger behavior
- The debugger's tracking of stack usage.

For more information about these options, see the *IDE Project Management and Building Guide*.

See also *Adapting for target hardware*, page 44.

EXECUTING FROM RESET

The **Run to** option—available on the **Debugger>Setup** page—specifies a location you want C-SPY to run to when you start the debugger as well as after each reset. C-SPY will place a temporary breakpoint at this location and all code up to this point is executed before stopping at the location.

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

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

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

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

USING A SETUP MACRO FILE

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

For more information about setup macro files and functions, see *Briefly about setup macro functions and files, page 178*. For an example of how to use a setup macro file, see the chapter *Initializing target hardware before C-SPY starts*, page 45.

To register a setup macro file:

1 Before you start C-SPY, choose **Project>Options>Debugger>Setup**.

2 Select **Use macro file** and type the path and name of your setup macro file, for example `Setup.mac`. If you do not type a filename extension, the extension `mac` is assumed.

SELECTING A DEVICE DESCRIPTION FILE

C-SPY uses device description files to handle device-specific information.

A default device description file is automatically used based on your project settings. If you want to override the default file, you must select your device description file. Device description files are provided in the `8051\config` directory and they have the filename extension `ddf`.

For more information about device description files, see *Adapting for target hardware*, page 44.

To override the default device description file:

1 Before you start C-SPY, choose **Project>Options>Debugger>Setup**.

2 Enable the use of a device description file and select a file using the **Device description file** browse button.

LOADING PLUGIN MODULES

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

For more information, see *Plugins*, page 266.

Starting C-SPY

When you have set up the debugger, you are ready to start a debug session; this section describes the steps involved.

More specifically, you will get information about:

- Starting the debugger
- Loading executable files built outside of the IDE
- Starting a debug session with source files missing
- Loading multiple images.

STARTING THE DEBUGGER

You can choose to start the debugger with or without loading the current project.

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

To start C-SPY without reloading the current project, click the **Debug without Downloading** button. Alternatively, choose **Project>Debug without Downloading**.

LOADING EXECUTABLE FILES BUILT OUTSIDE OF THE IDE

You can also load C-SPY with an application that was built outside the IDE, for example applications built on the command line. To load an externally built executable file and to set build options you must first create a project for it in your workspace.

To create a project for an externally built file:

1 Choose **Project>Create New Project**, and specify a project name.

2 To add the executable file to the project, choose **Project>Add Files** and make sure to choose **All Files** in the **Files of type** drop-down list. Locate the executable file.

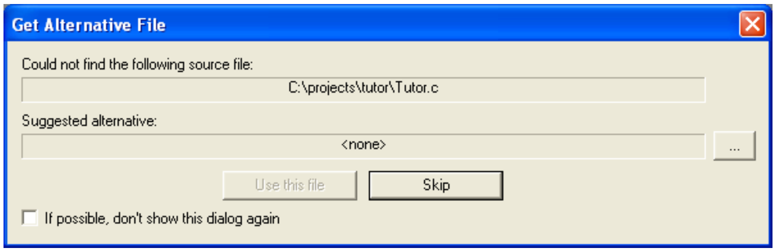
3 To start the executable file, click the **Download and Debug** button. The project can be reused whenever you rebuild your executable file.

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

STARTING A DEBUG SESSION WITH SOURCE FILES MISSING

Normally, when you use the IAR Embedded Workbench IDE to edit source files, build your project, and start the debug session, all required files are available and the process works as expected.

However, if C-SPY cannot automatically find the source files, for example if the application was built on another computer, the **Get Alternative File** dialog box is displayed:



Could not find the following source file:
C:\projects\tutor\Tutor.c
Suggested alternative:
<none>
If possible, don't show this dialog again

Figure 4: Get Alternative File dialog box

Typically, you can use the dialog box like this:

- The source files are not available: Click **If possible, don't show this dialog again** and then click **Skip**. C-SPY will assume that there simply is no source file available. The dialog box will not appear again, and the debug session will not try to display the source code.
- Alternative source files are available at another location: Specify an alternative source code file, click **If possible, don't show this dialog again**, and then click **Use this file**. C-SPY will assume that the alternative file should be used. The dialog box will not appear again, unless a file is needed for which there is no alternative file specified and which cannot be located automatically.

If you restart the IAR Embedded Workbench IDE, the **Get Alternative File** dialog box will be displayed again once even if you have clicked **If possible, don't show this dialog again**. This gives you an opportunity to modify your previous settings.

For more information, see *Get Alternative File dialog box*, page 53.

LOADING MULTIPLE IMAGES

Normally, a debuggable application consists of exactly one file that you debug. However, you can also load additional debug files (images). This means that the complete program consists of several images.

Typically, this is useful if you want to debug your application in combination with a prebuilt ROM image that contains an additional library for some platform-provided features. The ROM image and the application are built using separate projects in the IAR Embedded Workbench IDE and generate separate output files.

If more than one image has been loaded, you will have access to the combined debug information for all the loaded images. In the Images window you can choose whether you want to have access to debug information for one image or for all images.

To load additional images at C-SPY startup:

1. Choose **Project>Options>Debugger>Images** and specify up to three additional images to be loaded. For more information, see *Images*, page 265.
2. Start the debug session.

To load additional images at a specific moment:

Use the `__loadImage` system macro and execute it using either one of the methods described in *Procedures for using C-SPY macros*, page 179.

To display a list of loaded images:

Choose **Images** from the **View** menu. The Images window is displayed, see *Images window*, page 52.

Adapting for target hardware

This section provides information about how to describe the target hardware to C-SPY, and how you can make C-SPY initialize the target hardware before your application is downloaded to memory.

More specifically, you will get information about:

- Modifying a device description file
- Initializing target hardware before C-SPY starts.

MODIFYING A DEVICE DESCRIPTION FILE

C-SPY uses device description files provided with the product to handle several of the target-specific adaptations, see *Selecting a device description file*, page 41. They contain device-specific information such as:

- Memory information for device-specific memory zones, see *C-SPY memory zones*, page 114
- Definitions of interrupt vectors, SFR banked registers, memory-mapped peripheral units, device-specific CPU registers, and groups of these
- Definitions for device-specific interrupts, which makes it possible to simulate these interrupts in the C-SPY simulator; see *Simulating interrupts*, page 161.

Normally, you do not need to modify the device description file. However, if the predefinitions are not sufficient for some reason, you can edit the file. The syntax is described in the files. Note, however, that the format of these descriptions might be updated in future upgrade versions of the product.

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

For information about how to load a device description file, see *Selecting a device description file*, page 41.

INITIALIZING TARGET HARDWARE BEFORE C-SPY STARTS

If your hardware uses external memory that must be enabled before code can be downloaded to it, C-SPY needs a macro to perform this action before your application can be downloaded. For example:

1 Create a new text file and define your macro function. For example, a macro that enables external SDRAM might look like this:

```
/* Your macro function. */
enableExternalSDRAM()
{
  __message "Enabling external SDRAM\n";
  __writeMemory32( /* Place your code here. */ );
  /* And more code here, if needed. */
}

/* Setup macro determines time of execution. */
execUserPreload()
{
  enableExternalSDRAM();
}
```

Because the built-in `execUserPreload` setup macro function is used, your macro function will be executed directly after the communication with the target system is established but before C-SPY downloads your application.

2 Save the file with the filename extension `mac`.

3 Before you start C-SPY, choose **Project>Options>Debugger** and click the **Setup** tab.

4 Select the option **Use Setup file** and choose the macro file you just created.

Your setup macro will now be loaded during the C-SPY startup sequence.

Running example projects

IAR Embedded Workbench comes with example applications. You can use these examples to get started using the development tools from IAR Systems or simply to verify that contact has been established with your target board. You can also use the examples as a starting point for your application project.

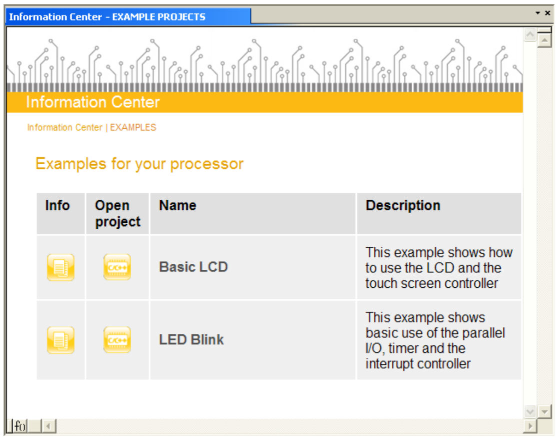
You can find the examples in the `8051\examples` directory. The examples are ready to be used as is. They are supplied with ready-made workspace files, together with source code files and all other related files.

RUNNING AN EXAMPLE PROJECT

To run an example project:

1 Choose **Help>Information Center** and click **EXAMPLE PROJECTS**.

2 Browse to the example that matches the specific evaluation board or starter kit you are using.


Information Center - EXAMPLE PROJECTS
Information Center
Information Center | EXAMPLES
Examples for your processor

Info	Open project	Name	Description
		Basic LCD	This example shows how to use the LCD and the touch screen controller
		LED Blink	This example shows basic use of the parallel I/O, timer and the interrupt controller

Figure 5: Example applications

Click the **Open Project** button.

3 In the dialog box that appears, choose a destination folder for your project location. Click **Select** to confirm your choice.

4 The available example projects are displayed in the workspace window. Select one of the projects, and if it is not the active project (highlighted in bold), right-click it and choose **Set As Active** from the context menu.

5 To view the project settings, select the project and choose **Options** from the context menu. Verify the settings for **Device**, **CPU core**, and **Debugger>Setup>Driver**. As for other settings, the project is set up to suit the target system you selected.

For more information about the C-SPY options and how to configure C-SPY to interact with the target board, see *Debugger options*, page 261.

Click **OK** to close the project **Options** dialog box.

6. To compile and link the application, choose **Project>Make** or click the **Make** button.
7. To start C-SPY, choose **Project>Debug** or click the **Download and Debug** button. If C-SPY fails to establish contact with the target system, see *Resolving problems*, page 289.
8. Choose **Debug>Go** or click the **Go** button to start the application.

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

Reference information on starting C-SPY

This section gives reference information about these windows and dialog boxes:

- *C-SPY Debugger main window*, page 47
- *Images window*, page 52
- *Get Alternative File dialog box*, page 53

See also:

- Tools options for the debugger in the *IDE Project Management and Building Guide*.

C-SPY Debugger main window

When you start the debugger, these debugger-specific items appear in the main IAR Embedded Workbench IDE window:

- A dedicated **Debug** menu with commands for executing and debugging your application
- Depending on the C-SPY driver you are using, a driver-specific menu, often referred to as the ***Driver menu*** in this documentation. Typically, this menu contains menu commands for opening driver-specific windows and dialog boxes.
- A special debug toolbar
- Several windows and dialog boxes specific to C-SPY.

The C-SPY main window might look different depending on which components of the product installation you are using.

Menu bar

These menus are available when C-SPY is running:

Debug	Provides commands for executing and debugging the source application, see *Debug menu*, page 49. Most of the commands are also available as icon buttons on the debug toolbar.
Simulator	Provides access to the dialog boxes for setting up interrupt simulation and memory access checking. This menu is only available when the C-SPY Simulator is selected, see *Simulator menu*, page 286.
Texas Instruments Emulator	Provides commands specific to the C-SPY Texas Instruments driver. This menu is only available when the driver is selected, see *Texas Instruments Emulator menu*, page 287.
FS2 Emulator	Provides access to the **Breakpoints Usage** dialog box for the C-SPY FS2 driver, see *Breakpoint Usage dialog box*, page 102. This menu is only available when the driver is selected.
Infineon Emulator	Provides commands specific to the C-SPY Infineon driver. This menu is only available when the driver is selected, see *Infineon Emulator menu*, page 288.
Nordic Semiconductor Emulator	Provides access to the **Breakpoints Usage** dialog box for the C-SPY Nordic Semiconductor driver, see *Breakpoint Usage dialog box*, page 102. This menu is only available when the driver is selected.
ROM-Monitor Debugger	Provides access to the **Breakpoints Usage** dialog box for the C-SPY ROM-monitor driver, see *Breakpoint Usage dialog box*, page 102. This menu is only available when the driver is selected.
Analog Devices ROM-Monitor	Provides access to the **Breakpoints Usage** dialog box for the C-SPY Analog Devices driver, see *Breakpoint Usage dialog box*, page 102. This menu is only available when the driver is selected.
Silabs Emulator	Provides commands specific to the C-SPY Silabs driver. This menu is only available when the driver is selected, see *Silabs Emulator menu*, page 289.

Debug menu

The **Debug** menu is available when C-SPY is running. The **Debug** menu provides commands for executing and debugging the source application. Most of the commands are also available as icon buttons on the debug toolbar.

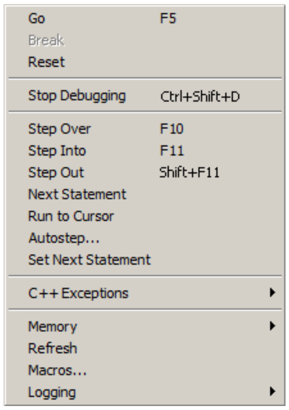


Figure 6: Debug menu

These commands are available:

⏩	**Go** F5	Executes from the current statement or instruction until a breakpoint or program exit is reached.
✋	**Break**	Stops the application execution.
⟲	**Reset**	Resets the target processor.
✕	**Stop Debugging** Ctrl+Shift+D	Stops the debugging session and returns you to the project manager.
↷	**Step Over** F10	Executes the next statement, function call, or instruction, without entering C or C++ functions or assembler subroutines.
↓	**Step Into** F11	Executes the next statement or instruction, or function call, entering C or C++ functions or assembler subroutines.
↑	**Step Out** Shift+F11	Executes from the current statement up to the statement after the call to the current function.
→	**Next Statement**	Executes directly to the next statement without stopping at individual function calls.

Run to Cursor	Executes from the current statement or instruction up to a selected statement or instruction.
Autostep	Displays a dialog box where you can customize and perform autostepping, see *Autostep settings dialog box*, page 71.
Set Next Statement	Moves the program counter directly to where the cursor is, without executing any source code. Note, however, that this creates an anomaly in the program flow and might have unexpected effects.
C++ Exceptions> Break on Throw	This menu command is not supported by your product package.
C++ Exceptions> Break on Uncaught Exception	This menu command is not supported by your product package.
Memory>Save	Displays a dialog box where you can save the contents of a specified memory area to a file, see *Memory Save dialog box*, page 120.
Memory>Restore	Displays a dialog box where you can load the contents of a file in Intel-extended or Motorola s-record format to a specified memory zone, see *Memory Restore dialog box*, page 121.
Refresh	Refreshes the contents of all debugger windows. Because window updates are automatic, this is needed only in unusual situations, such as when target memory is modified in ways C-SPY cannot detect. It is also useful if code that is displayed in the Disassembly window is changed.
Macros	Displays a dialog box where you can list, register, and edit your macro files and functions, see *Using the Macro Configuration dialog box*, page 181.
Logging>Set Log file	Displays a dialog box where you can choose to log the contents of the Debug Log window to a file. You can select the type and the location of the log file. You can choose what you want to log: errors, warnings, system information, user messages, or all of these. See *Log File dialog box*, page 70.
Logging> Set Terminal I/O Log file	Displays a dialog box where you can choose to log simulated target access communication to a file. You can select the destination of the log file. See *Terminal I/O Log File dialog box*, page 68.

C-SPY windows

Depending on the C-SPY driver you are using, these windows specific to C-SPY are available when C-SPY is running:

- C-SPY Debugger main window
- Disassembly window
- Memory window
- Symbolic Memory window
- Register window
- Watch window
- Locals window
- Auto window
- Live Watch window
- Quick Watch window
- Statics window
- Call Stack window
- Trace window
- Function Trace window
- Timeline window
- Terminal I/O window
- Code Coverage window
- Function Profiler window
- Images window
- Stack window
- Symbols window.

Additional windows are available depending on which C-SPY driver you are using.

Editing in C-SPY windows

You can edit the contents of the Memory, Symbolic Memory, Register, Auto, Watch, Locals, Statics, Live Watch, and Quick Watch windows.

Use these keyboard keys to edit the contents of these windows:

Enter	Makes an item editable and saves the new value.
Esc	Cancels a new value.

In windows where you can edit the **Expression** field, you can specify the number of elements to be displayed in the field by adding a semicolon followed by an integer. For example, to display only the three first elements of an array named `myArray`, or three elements in sequence starting with the element pointed to by a pointer, write:

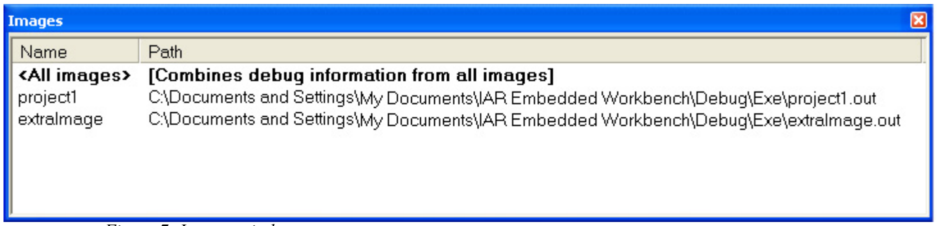
```
myArray;3
```

Optionally, add a comma and another integer that specifies which element to start with. For example, to display elements 10–14, write:

```
myArray;5,10
```

Images window

The Images window is available from the **View** menu.



Name	Path
<All images>	**[Combines debug information from all images]**
project1	C:\Documents and Settings\My Documents\IAR Embedded Workbench\Debug\Exe\project1.out
extraImage	C:\Documents and Settings\My Documents\IAR Embedded Workbench\Debug\Exe\extraImage.out

Figure 7: Images window

The Images window lists all currently loaded images (debug files).

Normally, a source application consists of exactly one image that you debug. However, you can also load additional images. This means that the complete debuggable unit consists of several images.

Display area

This area lists the loaded images in these columns:

Name	The name of the loaded image.
Path	The path to the loaded image.

C-SPY can either use debug information from all of the loaded images simultaneously, or from one image at a time. Double-click on a row to show information only for that image. The current choice is highlighted.

Context menu

This context menu is available:

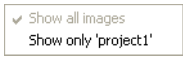


Figure 8: Images window context menu

These commands are available:

Show all images	Shows debug information for all loaded debug images.
Show only *image*	Shows debug information for the selected debug image.

Related information

For related information, see:

- *Loading multiple images*, page 43
- *Images*, page 265
- *__loadImage*, page 197.

Get Alternative File dialog box

The **Get Alternative File** dialog box is displayed if C-SPY cannot automatically find the source files to be loaded, for example if the application was built on another computer.

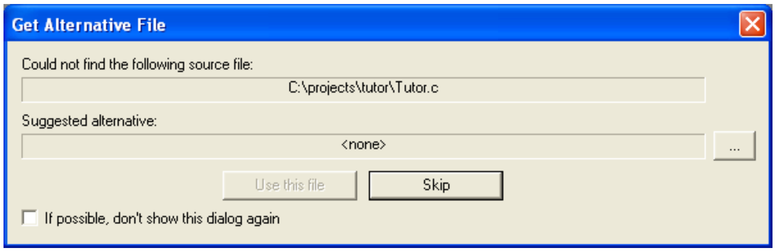

Get Alternative File
Could not find the following source file:
C:\projects\tutor\Tutor.c
Suggested alternative:
<none>
...
Use this file
Skip
If possible, don't show this dialog again

Figure 9: Get Alternative File dialog box

Could not find the following source file

The missing source file.

Suggested alternative

Specify an alternative file.

Use this file

After you have specified an alternative file, **Use this file** establishes that file as the alias for the requested file. Note that after you have chosen this action, C-SPY will automatically locate other source files if these files reside in a directory structure similar to the first selected alternative file.

The next time you start a debug session, the selected alternative file will be preloaded automatically.

Skip

C-SPY will assume that the source file is not available for this debug session.

If possible, don't show this dialog again

Instead of displaying the dialog box again for a missing source file, C-SPY will use the previously supplied response.

Related information

For related information, see *Starting a debug session with source files missing*, page 42.

Executing your application

This chapter contains information about executing your application in C-SPY®. More specifically, this means:

- Introduction to application execution
- Reference information on application execution.

Introduction to application execution

This section covers these topics:

- Briefly about application execution
- Source and disassembly mode debugging
- Single stepping
- Running the application
- Highlighting
- Call stack information
- Terminal input and output
- Debug logging.

BRIEFLY ABOUT APPLICATION EXECUTION

C-SPY allows you to monitor and control the execution of your application. By single-stepping through it, and setting breakpoints, you can examine details about the application execution, for example the values of variables and registers. You can also use the call stack to step back and forth in the function call chain.

The terminal I/O and debug log features let you interact with your application.

You can find commands for execution on the **Debug** menu and on the toolbar.

SOURCE AND DISASSEMBLY MODE DEBUGGING

C-SPY allows you to switch between source mode and disassembly mode debugging as needed.

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

Disassembly mode debugging lets you focus on the critical sections of your application, and provides you with precise control of the application code. You can open a disassembly window which displays a mnemonic assembler listing of your application based on actual memory contents rather than source code, and lets you execute the application exactly one machine instruction at a time.

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

SINGLE STEPPING

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

- **Step Into**
- **Step Over**
- **Next Statement**
- **Step Out**.

Using the **Autostep settings** dialog box, you can automate the single stepping. For more information, see *Autostep settings dialog box*, page 71.

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

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

Step Into

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

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

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

Step Over

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

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

Next Statement

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

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

Step Out

When inside the function, you can—if you wish—use the **Step Out** command to step out of it before it reaches the exit. This will take you directly to the statement immediately after the function call:

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

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

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

RUNNING THE APPLICATION

Go

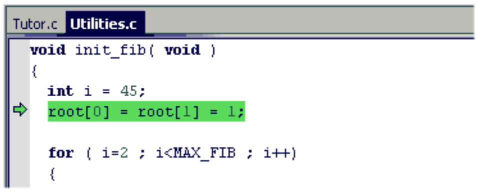
The **Go** command continues execution from the current position until a breakpoint or program exit is reached.

Run to Cursor

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

HIGHLIGHTING

At each stop, C-SPY highlights the corresponding C or C++ source or instruction with a green color, in the editor and the Disassembly window respectively. In addition, a green arrow appears in the editor window when you step on C or C++ source level, and in the Disassembly window when you step on disassembly level. This is determined by which of the windows is the active window. If none of the windows are active, it is determined by which of the windows was last active.



```
void init_fib( void )
{
  int i = 45;
  root[0] = root[1] = 1;

  for ( i=2 ; i<MAX_FIB ; i++)
  {
```

Figure 10: C-SPY highlighting source location

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

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

CALL STACK INFORMATION

The compiler generates extensive backtrace information. This allows C-SPY to show, without any runtime penalty, the complete function call chain at any time.

Typically, this is useful for two purposes:

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

The Call Stack window shows a list of function calls, with the current function at the top. When you inspect a function in the call chain, the contents of all affected windows are updated to display the state of that particular call frame. This includes the editor, Locals, Register, Watch and Disassembly windows. A function would normally not make use of all registers, so these registers might have undefined states and be displayed as dashes (---).

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

For your convenience, it is possible to select a function in the call stack and click the **Run to Cursor** command to execute to that function.

Assembler source code does not automatically contain any backtrace information. To see the call chain also for your assembler modules, you can add the appropriate `CFI` assembler directives to the assembler source code. For further information, see the *IAR Assembler Reference Guide for 8051*.

TERMINAL INPUT AND OUTPUT

Sometimes you might have to debug constructions in your application that use `stdin` and `stdout` without an actual hardware device for input and output. The Terminal I/O window lets you enter input to your application, and display output from it. You can also direct terminal I/O to a file, using the **Terminal I/O Log Files** dialog box.

This facility is useful in two different contexts:

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

For more information, see *Terminal I/O window*, page 67 and *Terminal I/O Log File dialog box*, page 68.

DEBUG LOGGING

The Debug Log window displays debugger output, such as diagnostic messages, macro-generated output, event log messages, and information about trace.

It can sometimes be convenient to log the information to a file where you can easily inspect it. The two main advantages are:

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

Reference information on application execution

This section gives reference information about these windows and dialog boxes:

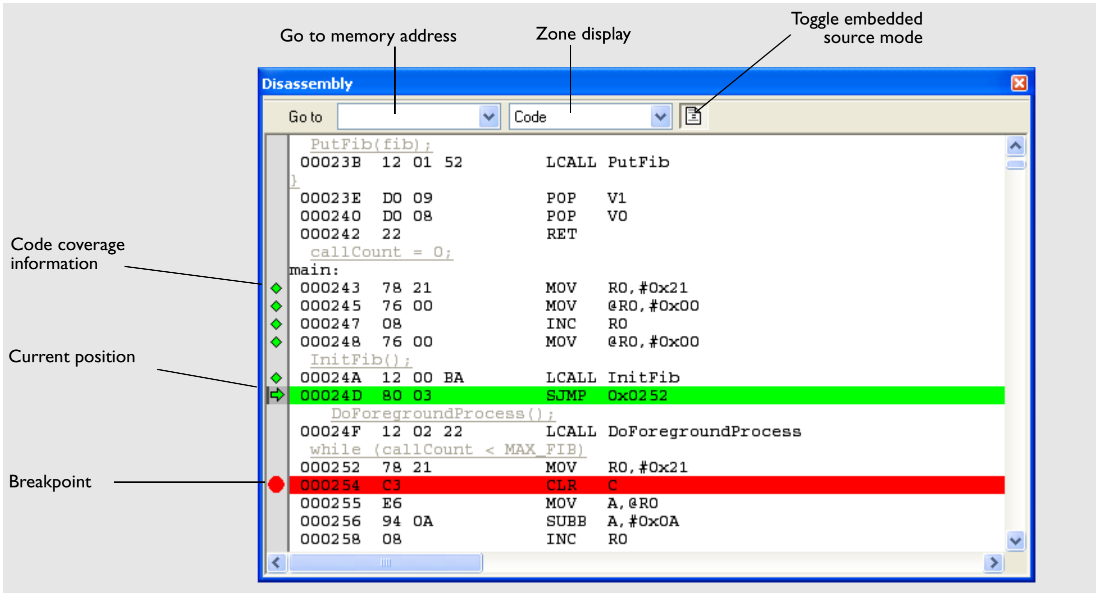
- *Disassembly window*, page 61
- *Call Stack window*, page 65
- *Terminal I/O window*, page 67
- *Terminal I/O Log File dialog box*, page 68
- *Debug Log window*, page 69
- *Log File dialog box*, page 70
- *Report Assert dialog box*, page 71

- *Autostep settings dialog box*, page 71.

See also Terminal I/O options in *IDE Project Management and Building Guide*.

Disassembly window

The C-SPY Disassembly window is available from the **View** menu.



```
 PutFib(fib);
000023B  12 01 52        LCALL PutFib
}
00023E  D0 09            POP   V1
000240  D0 08            POP   V0
000242  22               RET
 callCount = 0;
main:
000243  78 21            MOV   R0,#0x21
000245  76 00            MOV   @R0,#0x00
000247  08               INC   R0
000248  76 00            MOV   @R0,#0x00
 InitFib();
00024A  12 00 BA         LCALL InitFib
00024D  80 03            SJMP  0x0252
  DoForegroundProcess();
00024F  12 02 22         LCALL DoForegroundProcess
 while (callCount < MAX_FIB)
000252  78 21            MOV   R0,#0x21
000254  C3               CLR   C
000255  E6               MOV   A,@R0
000256  94 0A            SUBB  A,#0x0A
000258  08               INC   R0
```

Figure 11: C-SPY Disassembly window

This window shows the application being debugged as disassembled application code.

To change the default color of the source code in the Disassembly window:

1. Choose **Tools>Options>Debugger**.
2. Set the default color using the **Source code coloring in disassembly window** option.

To view the corresponding assembler code for a function, you can select it in the editor window and drag it to the Disassembly window.

Toolbar

The toolbar contains:

Go to	The location you want to view. This can be a memory address, or the name of a variable, function, or label.
Zone display	Lists the available memory zones to display, see *C-SPY memory zones*, page 114.
Toggle Mixed-Mode	Toggles between displaying only disassembled code or disassembled code together with the corresponding source code. Source code requires that the corresponding source file has been compiled with debug information.

Display area

The display area shows the disassembled application code.

This area contains these graphic elements:

Green highlight	Indicates the current position, that is the next assembler instruction to be executed. To move the cursor to any line in the Disassembly window, click the line. Alternatively, move the cursor using the navigation keys.
Yellow highlight	Indicates a position other than the current position, such as when navigating between frames in the Call Stack window or between items in the Trace window.
Red dot	Indicates a breakpoint. Double-click in the gray left-side margin of the window to set a breakpoint. For more information, see *Using breakpoints*, page 89.
Green diamond	Indicates code that has been executed—that is, code coverage.

If instruction profiling has been enabled from the context menu, an extra column in the left-side margin appears with information about how many times each instruction has been executed.

Context menu

This context menu is available:

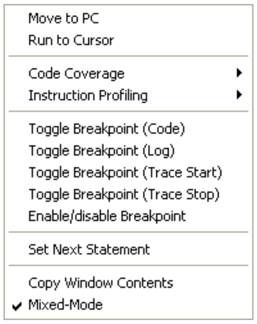


Figure 12: Disassembly window context menu

Note: The contents of this menu are dynamic, which means it might look different depending on your product package.

These commands are available:

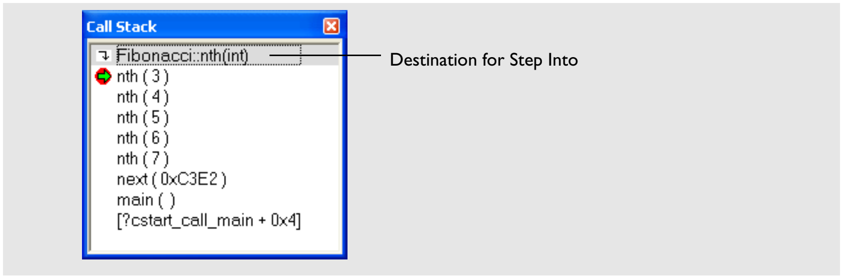
Move to PC	Displays code at the current program counter location.
Run to Cursor	Executes the application from the current position up to the line containing the cursor.
Code Coverage	Displays a submenu that provides commands for controlling code coverage. This command is only enabled if the driver you are using supports it. **Enable**, toggles code coverage on or off. **Show**, toggles the display of code coverage on or off. Executed code is indicated by a green diamond. **Clear**, clears all code coverage information.

Instruction Profiling	Displays a submenu that provides commands for controlling instruction profiling. This command is only enabled if the driver you are using supports it. **Enable**, toggles instruction profiling on or off. **Show**, toggles the display of instruction profiling on or off. For each instruction, the left-side margin displays how many times the instruction has been executed. **Clear**, clears all instruction profiling information.
Toggle Breakpoint (Code)	Toggles a code breakpoint. Assembler instructions and any corresponding label at which code breakpoints have been set are highlighted in red. For more information, see *Code breakpoints dialog box*, page 103.
Toggle Breakpoint (Log)	Toggles a log breakpoint for trace printouts. Assembler instructions at which log breakpoints have been set are highlighted in red. For more information, see *Log breakpoints dialog box*, page 105.
Toggle Breakpoint (Trace Start)	Toggles a Trace Start breakpoint. When the breakpoint is triggered, the trace data collection starts. Note that this menu command is only available if the C-SPY driver you are using supports trace. For more information, see *Trace Start breakpoints dialog box*, page 143.
Toggle Breakpoint (Trace Stop)	Toggles a Trace Stop breakpoint. When the breakpoint is triggered, the trace data collection stops. Note that this menu command is only available if the C-SPY driver you are using supports trace. For more information, see *Trace Stop breakpoints dialog box*, page 144.
Enable/Disable Breakpoint	Enables and Disables a breakpoint. If there is more than one breakpoint at a specific line, all those breakpoints are affected by the **Enable/Disable** command.
Edit Breakpoint	Displays the breakpoint dialog box to let you edit the currently selected breakpoint. If there is more than one breakpoint on the selected line, a submenu is displayed that lists all available breakpoints on that line.
Set Next Statement	Sets the program counter to the address of the instruction at the insertion point.
Copy Window Contents	Copies the selected contents of the Disassembly window to the clipboard.

Mixed-Mode Toggles between showing only disassembled code or disassembled code together with the corresponding source code. Source code requires that the corresponding source file has been compiled with debug information.

Call Stack window

The Call stack window is available from the **View** menu.



```
Fibonacci::nth(int)
nth ( 3 )
nth ( 4 )
nth ( 5 )
nth ( 6 )
nth ( 7 )
next ( 0xC3E2 )
main ( )
[?cstart_call_main + 0x4]
```

Figure 13: Call Stack window

This window displays the C function call stack with the current function at the top. To inspect a function call, double-click it. C-SPY now focuses on that call frame instead.

If the next **Step Into** command would step to a function call, the name of the function is displayed in the grey bar at the top of the window. This is especially useful for implicit function calls, such as C++ constructors, destructors, and operators.

Display area

Provided that the command **Show Arguments** is enabled, each entry in the display area has the format:

`function(values)`

where `(values)` is a list of the current value of the parameters, or empty if the function does not take any parameters.

Context menu

This context menu is available:

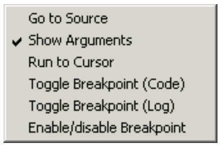


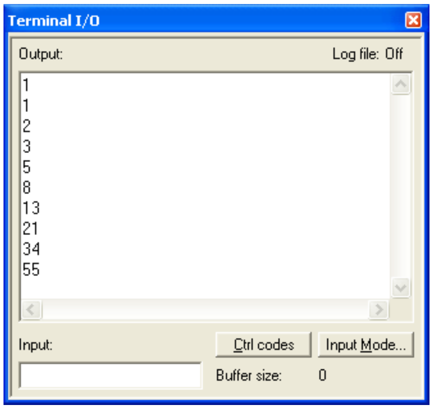
Figure 14: Call Stack window context menu

These commands are available:

Go to Source	Displays the selected function in the Disassembly or editor windows.
Show Arguments	Shows function arguments.
Run to Cursor	Executes until return to the function selected in the call stack.
Toggle Breakpoint (Code)	Toggles a code breakpoint.
Toggle Breakpoint (Log)	Toggles a log breakpoint.
Enable/Disable Breakpoint	Enables or disables the selected breakpoint.

Terminal I/O window

The Terminal I/O window is available from the **View** menu.



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

Figure 15: Terminal I/O window

Use this window to enter input to your application, and display output from it.

To use this window, you must:

1 Link your application with the option **With I/O emulation modules**.

C-SPY will then direct `stdin`, `stdout` and `stderr` to this window. If the Terminal I/O window is closed, C-SPY will open it automatically when input is required, but not for output.

Input

Type the text that you want to input to your application.

Ctrl codes

Opens a menu for input of special characters, such as `EOF` (end of file) and `NUL`.

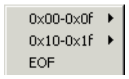
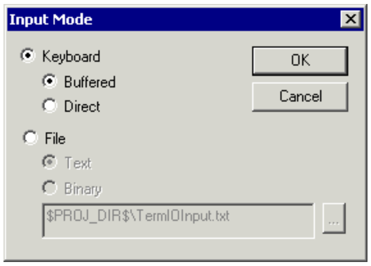


Figure 16: Ctrl codes menu

Input Mode

Opens the **Input Mode** dialog box where you choose whether to input data from the keyboard or from a file.



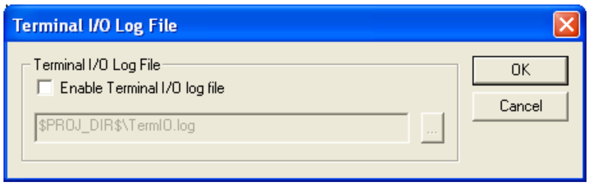
(● Keyboard) File
(● Buffered) Direct
(● Text) Binary
$PROJ_DIR$\TermIOInput.txt

Figure 17: Input Mode dialog box

For reference information about the options available in this dialog box, see Terminal I/O options in *IDE Project Management and Building Guide*.

Terminal I/O Log File dialog box

The **Terminal I/O Log File** dialog box is available by choosing **Debug>Logging>Set Terminal I/O Log File**.



Terminal I/O Log File
☐ Enable Terminal I/O log file
$PROJ_DIR$\TermIO.log

Figure 18: Terminal I/O Log File dialog box

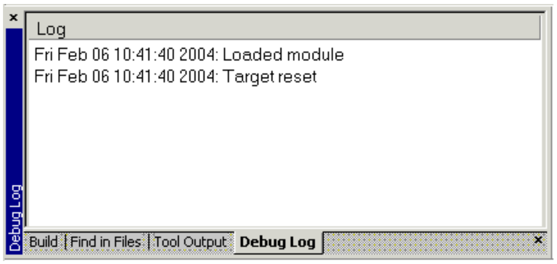
Use this dialog box to select a destination log file for terminal I/O from C-SPY.

Terminal IO Log Files

Controls the logging of terminal I/O. To enable logging of terminal I/O to a file, select **Enable Terminal IO log file** and specify a filename. The default filename extension is `log`. A browse button is available for your convenience.

Debug Log window

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



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

Figure 19: Debug Log window (message window)

This window displays debugger output, such as diagnostic messages, macro-generated output, event log messages, and information about trace. This output is only available when C-SPY is running. When opened, this window is, by default, grouped together with the other message windows, see *IDE Project Management and Building Guide*.

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

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

Context menu

This context menu is available:

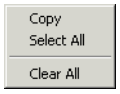


Figure 20: Debug Log window context menu

These commands are available:

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

Log File dialog box

The **Log File** dialog box is available by choosing **Debug>Logging>Set Log File**.

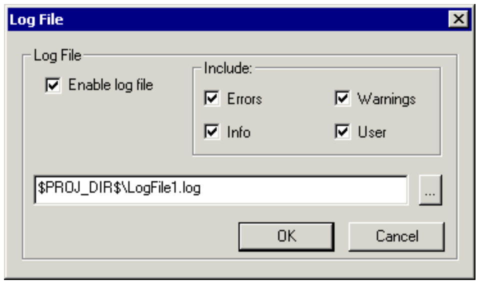


Figure 21: Log File dialog box

Use this dialog box to log output from C-SPY to a file.

Enable Log file

Enables or disables logging to the file.

Include

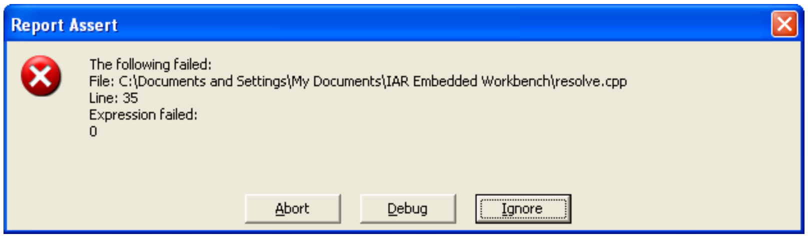
The information printed in the file is, by default, the same as the information listed in the Log window. To change the information logged, choose between:

Errors	C-SPY has failed to perform an operation.
Warnings	An error or omission of concern.
Info	Progress information about actions C-SPY has performed.
User	Messages from C-SPY macros, that is, your messages using the `__message` statement.

Use the browse button, to override the default file and location of the log file (the default filename extension is `log`).

Report Assert dialog box

The **Report Assert dialog box** appears if you have a call to the `assert` function in your application source code, and the assert condition is false. In this dialog box you can choose how to proceed.



```
The following failed:
File: C:\Documents and Settings\My Documents\IAR Embedded Workbench\resolve.cpp
Line: 35
Expression failed:
0
```

Figure 22: Report Assert dialog box

Abort

The application stops executing and the runtime library function `abort`, which is part of your application on the target system, will be called. This means that the application itself terminates its execution.

Debug

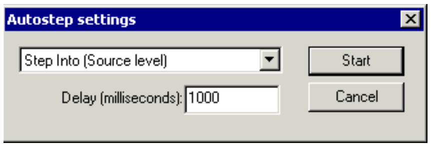
C-SPY stops the execution of the application and returns control to you.

Ignore

The assertion is ignored and the application continues to execute.

Autostep settings dialog box

The **Autostep settings** dialog box is available from the **Debug** menu.



Step Into (Source level)

Delay (milliseconds): 1000

Figure 23: Autostep settings dialog box

Use this dialog box to customize autostepping.

The drop-down menu lists the available step commands.

Delay

Specify the delay between each step in milliseconds.

Working with variables and expressions

This chapter describes how variables and expressions can be used in C-SPY®. More specifically, this means:

- Introduction to working with variables and expressions
- Reference information on working with variables and expressions.

Introduction to working with variables and expressions

This section covers these topics:

- Briefly about working with variables and expressions
- C-SPY expressions
- Limitations on variable information
- Viewing assembler variables.

BRIEFLY ABOUT WORKING WITH VARIABLES AND EXPRESSIONS

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

- Tooltip watch—in the editor window—provides the simplest way of viewing the value of a variable or more complex expressions. Just point at the variable with the mouse pointer. The value is displayed next to the variable.
- The Auto window displays a useful selection of variables and expressions in, or near, the current statement. The window is automatically updated when execution stops.
- The Locals window displays the local variables, that is, auto variables and function parameters for the active function. The window is automatically updated when execution stops.
- The Watch window allows you to monitor the values of C-SPY expressions and variables. The window is automatically updated when execution stops.
- The Live Watch window repeatedly samples and displays the values of expressions while your application is executing. Variables in the expressions must be statically located, such as global variables.

- The Statics window displays the values of variables with static storage duration. The window is automatically updated when execution stops.
- The Quick Watch window gives you precise control over when to evaluate an expression.
- The Symbols window displays all symbols with a static location, that is, C/C++ functions, assembler labels, and variables with static storage duration, including symbols from the runtime library.
- The Trace-related windows let you inspect the program flow up to a specific state. For more information, see *Collecting and using trace data*, page 133.

Details about using these windows

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

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

For text that is too wide to fit in a column—in any of the these windows, except the Trace window—and thus is truncated, just point at the text with the mouse pointer and tooltip information is displayed.

Right-click in any of the windows to access the context menu which contains additional commands. Convenient drag-and-drop between windows is supported, except for in the Locals window and the Quick Watch window where it is not relevant.

C-SPY EXPRESSIONS

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

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

Expressions that are built with these types of symbols are called C-SPY expressions and there are several methods for monitoring these in C-SPY. Examples of valid C-SPY expressions are:

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

C/C++ symbols

C symbols are symbols that you have defined in the C source code of your application, for instance variables, constants, and functions (functions can be used as symbols but cannot be executed). C symbols can be referenced by their names. Note that C++ symbols might implicitly contain function calls which are not allowed in C-SPY symbols and expressions.

Assembler symbols

Assembler symbols can be assembler labels or register names. That is, general purpose registers, such as `R0`–`R7`, and special purpose registers, such as the program counter and the status register. If a device description file is used, all memory-mapped peripheral units, such as I/O ports, can also be used as assembler symbols in the same way as the CPU registers. See *Modifying a device description file*, page 44.

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

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

Table 4: C-SPY assembler symbols expressions

In case of a name conflict between a hardware register and an assembler label, hardware registers have a higher precedence. To refer to an assembler label in such a case, you must enclose the label in back quotes ` (ASCII character 0x60). For example:

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

Table 5: Handling name conflicts between hardware registers and assembler labels

Which processor-specific symbols are available by default can be seen in the Register window, using the `CPU Registers` register group. See *Register window*, page 128.

C-SPY macro functions

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

For information about C-SPY macro functions and how to use them, see *Briefly about the macro language*, page 179.

C-SPY macro variables

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

For information about C-SPY macro variables and how to use them, see *Reference information on the macro language*, page 185.

Using sizeof

According to standard C, there are two syntactical forms of `sizeof`:

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

The former is for types and the latter for expressions.

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

LIMITATIONS ON VARIABLE INFORMATION

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

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

Effects of optimizations

The compiler is free to optimize the application software as much as possible, as long as the expected behavior remains. The optimization can affect the code so that debugging might be more difficult because it will be less clear how the generated code relates to the source code. Typically, using a high optimization level can affect the code in a way that will not allow you to view a value of a variable as expected.

Consider this example:

```
myFunction()
{
 int i = 42;
 ...
 x = computer(i); /* Here, the value of i is known to C-SPY */
 ...
}
```

From the point where the variable `i` is declared until it is actually used, the compiler does not need to waste stack or register space on it. The compiler can optimize the code, which means that C-SPY will not be able to display the value until it is actually used. If you try to view the value of a variable that is temporarily unavailable, C-SPY will display the text:

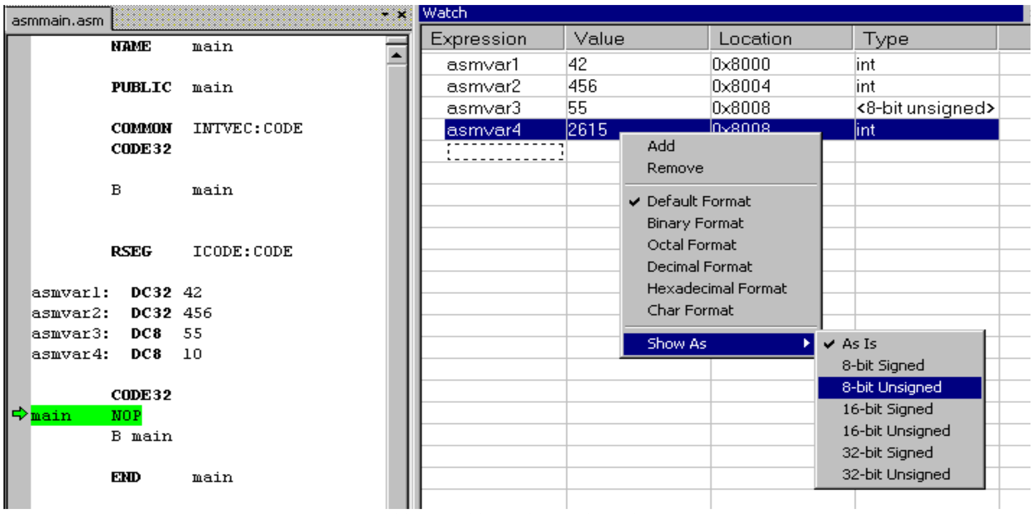
```
Unavailable
```

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

VIEWING ASSEMBLER VARIABLES

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

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



```
            NAME    main

            PUBLIC  main

            COMMON  INTVEC:CODE
            CODE32

            B       main


            RSEG    ICODE:CODE

asmvar1:    DC32  42
asmvar2:    DC32  456
asmvar3:    DC8   55
asmvar4:    DC8   10

            CODE32
main        NOP
            B main

            END     main
```

Expression	Value	Location	Type
asmvar1	42	0x8000	int
asmvar2	456	0x8004	int
asmvar3	55	0x8008	<8-bit unsigned>
asmvar4	2615	0x8008	int

Figure 24: Viewing assembler variables in the Watch window

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

Reference information on working with variables and expressions

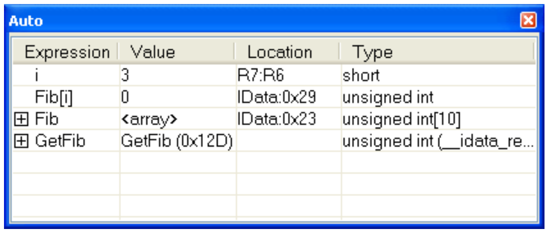
This section gives reference information about these windows and dialog boxes:

- *Auto window*, page 79
- *Locals window*, page 79
- *Watch window*, page 80
- *Live Watch window*, page 81
- *Statics window*, page 82
- *Select Statics dialog box*, page 84
- *Quick Watch window*, page 85
- *Symbols window*, page 86
- *Resolve Symbol Ambiguity dialog box*, page 87.

For trace-related reference information, see *Reference information on trace*, page 136.

Auto window

The Auto window is available from the **View** menu.



Expression	Value	Location	Type
i	3	R7:R6	short
Fib[i]	0	IData:0x29	unsigned int
⊞ Fib	<array>	IData:0x23	unsigned int[10]
⊞ GetFib	GetFib (0x12D)		unsigned int (__idata_re...

Figure 25: Auto window

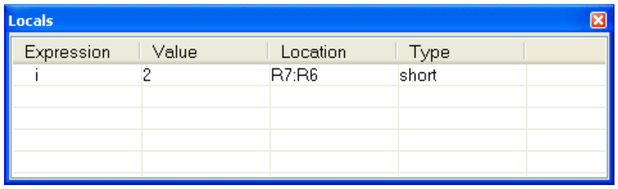
This window displays a useful selection of variables and expressions in, or near, the current statement. Every time execution in C-SPY stops, the values in the Auto window are recalculated. Values that have changed since the last stop are highlighted in red.

Context menu

For more information about the context menu, see *Watch window*, page 80.

Locals window

The Locals window is available from the **View** menu.



Expression	Value	Location	Type
i	2	R7:R6	short

Figure 26: Locals window

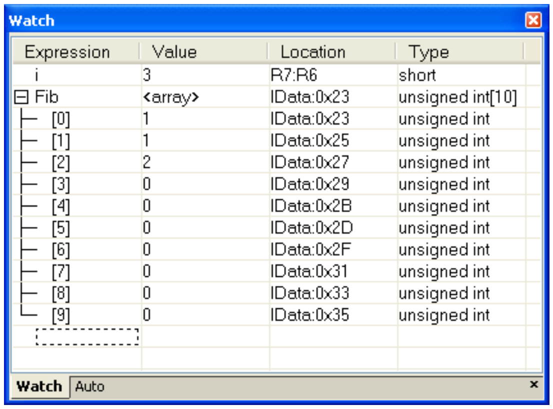
This window displays the local variables and parameters for the current function. Every time execution in C-SPY stops, the values in the Locals window are recalculated. Values that have changed since the last stop are highlighted in red.

Context menu

For more information about the context menu, see *Watch window*, page 80.

Watch window

The Watch window is available from the **View** menu.



Expression	Value	Location	Type
i	3	R7:R6	short
⊟ Fib	<array>	IData:0x23	unsigned int[10]
[0]	1	IData:0x23	unsigned int
[1]	1	IData:0x25	unsigned int
[2]	2	IData:0x27	unsigned int
[3]	0	IData:0x29	unsigned int
[4]	0	IData:0x2B	unsigned int
[5]	0	IData:0x2D	unsigned int
[6]	0	IData:0x2F	unsigned int
[7]	0	IData:0x31	unsigned int
[8]	0	IData:0x33	unsigned int
[9]	0	IData:0x35	unsigned int

Figure 27: Watch window

Use this window to monitor the values of C-SPY expressions or variables. You can view, add, modify, and remove expressions. Tree structures of arrays, structs, and unions are expandable, which means that you can study each item of these.

Every time execution in C-SPY stops, the values in the Watch window are recalculated. Values that have changed since the last stop are highlighted in red.

Context menu

This context menu is available:

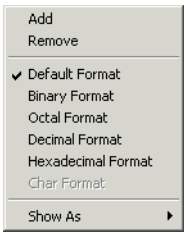


Figure 28: Watch window context menu

These commands are available:

Add	Adds an expression.
Remove	Removes the selected expression.
Default Format, **Binary Format**, **Octal Format**, **Decimal Format**, **Hexadecimal Format**, **Char Format**	Changes the display format of expressions. The display format setting affects different types of expressions in different ways, see Table 6, *Effects of display format setting on different types of expressions*. Your selection of display format is saved between debug sessions.
Show As	Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers. For more information, see *Viewing assembler variables*, page 77.

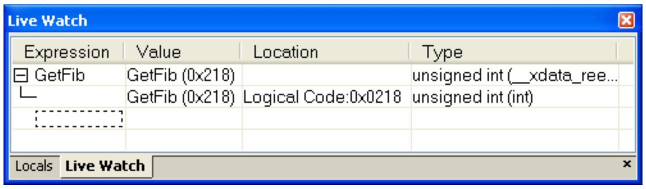
The display format setting affects different types of expressions in these ways:

Type of expression	Effects of display format setting
Variable	The display setting affects only the selected variable, not other variables.
Array element	The display setting affects the complete array, that is, the same display format is used for each array element.
Structure field	All elements with the same definition—the same field name and C declaration type—are affected by the display setting.

Table 6: Effects of display format setting on different types of expressions

Live Watch window

The Live Watch window is available from the **View** menu.



Expression	Value	Location	Type
⊟ GetFib	GetFib (0x218)		unsigned int (__xdata_ree...
└	GetFib (0x218)	Logical Code:0x0218	unsigned int (int)

Figure 29: Live Watch window

This window repeatedly samples and displays the value of expressions while your application is executing. Variables in the expressions must be statically located, such as global variables.

This window can only be used for hardware target systems supporting this feature.

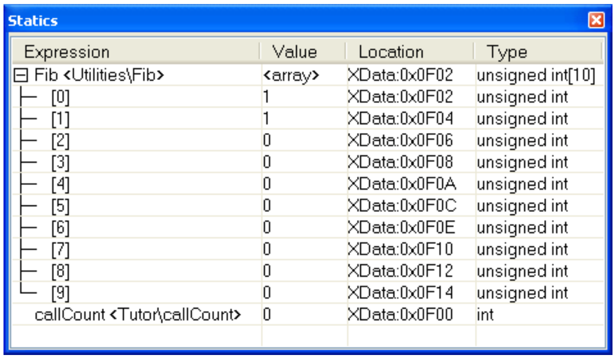
Context menu

For more information about the context menu, see *Watch window*, page 80.

In addition, the menu contains the **Options** command, which opens the **Debugger** dialog box where you can set the **Update interval** option. The default value of this option is 1000 milliseconds, which means the **Live Watch** window will be updated once every second during program execution.

Statics window

The Statics window is available from the **View** menu.



Expression	Value	Location	Type
⊟ Fib <Utilities\Fib>	<array>	XData:0x0F02	unsigned int[10]
[0]	1	XData:0x0F02	unsigned int
[1]	1	XData:0x0F04	unsigned int
[2]	0	XData:0x0F06	unsigned int
[3]	0	XData:0x0F08	unsigned int
[4]	0	XData:0x0F0A	unsigned int
[5]	0	XData:0x0F0C	unsigned int
[6]	0	XData:0x0F0E	unsigned int
[7]	0	XData:0x0F10	unsigned int
[8]	0	XData:0x0F12	unsigned int
[9]	0	XData:0x0F14	unsigned int
callCount <Tutor\callCount>	0	XData:0x0F00	int

Figure 30: Statics window

This window displays the values of variables with static storage duration, typically that is variables with file scope but also static variables in functions and classes. Note that `volatile` declared variables with static storage duration will not be displayed.

Every time execution in C-SPY stops, the values in the Statics window are recalculated. Values that have changed since the last stop are highlighted in red.

Display area

This area contains these columns:

Expression	The name of the variable. The base name of the variable is followed by the full name, which includes module, class, or function scope. This column is not editable.
Value	The value of the variable. Values that have changed are highlighted in red. This column is editable.
Location	The location in memory where this variable is stored.
Type	The data type of the variable.

Context menu

This context menu is available:

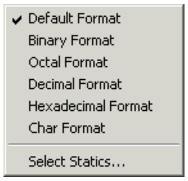


Figure 31: Statics window context menu

These commands are available:

Default Format, **Binary Format**, **Octal Format**, **Decimal Format**, **Hexadecimal Format**, **Char Format**	Changes the display format of expressions. The display format setting affects different types of expressions in different ways, see Table 6, *Effects of display format setting on different types of expressions*. Your selection of display format is saved between debug sessions.
Select Statics	Displays a dialog box where you can select a subset of variables to be displayed in the Statics window, see *Select Statics dialog box*, page 84.

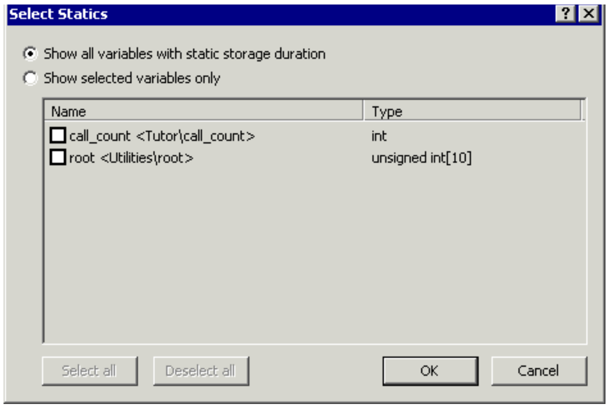
The display format setting affects different types of expressions in these ways:

Type of expression	Effects of display format setting
Variable	The display setting affects only the selected variable, not other variables.
Array element	The display setting affects the complete array, that is, the same display format is used for each array element.
Structure field	All elements with the same definition—the same field name and C declaration type—are affected by the display setting.

Table 7: Effects of display format setting on different types of expressions

Select Statics dialog box

The **Select Statics** dialog box is available from the context menu in the Statics window.


Select Statics
Show all variables with static storage duration
Show selected variables only
Select all
Deselect all
OK
Cancel

Name	Type
☐ call_count <Tutor\call_count>	int
☐ root <Utilities\root>	unsigned int[10]

Figure 32: Select Statics dialog box

Use this dialog box to select which variables should be displayed in the Statics window.

Show all variables with static storage duration

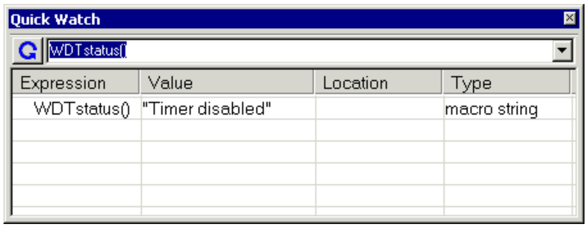
Makes *all* variables be displayed in the Statics window, including new variables that are added to your application between debug sessions.

Show selected variables only

Selects which variables to be displayed in the Statics window. Note that if you add a new variable to your application between two debug sessions, this variable will not automatically be displayed in the Statics window. Select the checkbox next to a variable to make that variable be displayed. Alternatively, click **Select All**.

Quick Watch window

The Quick Watch window is available from the **View** menu and from the context menu in the editor window.



Expression	Value	Location	Type
WDTstatus()	"Timer disabled"		macro string

Figure 33: Quick Watch window

Use this window to watch the value of a variable or expression and evaluate expressions at a specific point in time.

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

To evaluate an expression:

1. In the editor window, right-click on the expression you want to examine and choose Quick Watch from the context menu that appears.
2. The expression will automatically appear in the Quick Watch window.

Alternatively:

1. In the Quick Watch window, type the expression you want to examine in the **Expressions** text box.
2. Click the **Recalculate** button to calculate the value of the expression.

For an example, see *Executing macros using Quick Watch*, page 183.

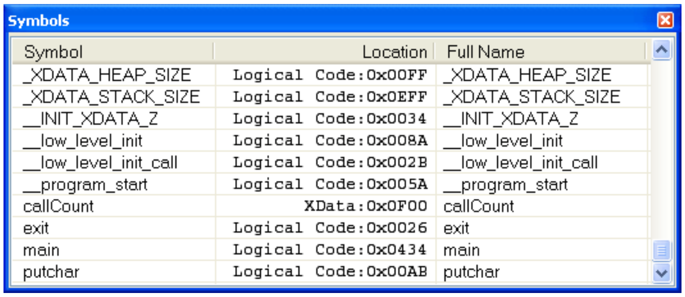
Context menu

For more information about the context menu, see *Watch window*, page 80.

In addition, the menu contains the **Add to Watch window** command, which adds the selected expression to the Watch window.

Symbols window

The Symbols window is available from the **View** menu.



Symbol	Location	Full Name
_XDATA_HEAP_SIZE	Logical Code:0x00FF	_XDATA_HEAP_SIZE
_XDATA_STACK_SIZE	Logical Code:0x0EFF	_XDATA_STACK_SIZE
__INIT_XDATA_Z	Logical Code:0x0034	__INIT_XDATA_Z
__low_level_init	Logical Code:0x008A	__low_level_init
__low_level_init_call	Logical Code:0x002B	__low_level_init_call
__program_start	Logical Code:0x005A	__program_start
callCount	XData:0x0F00	callCount
exit	Logical Code:0x0026	exit
main	Logical Code:0x0434	main
putchar	Logical Code:0x00AB	putchar

Figure 34: Symbols window

This window displays all symbols with a static location, that is, C/C++ functions, assembler labels, and variables with static storage duration, including symbols from the runtime library.

Display area

This area contains these columns:

Symbol	The symbol name.
Location	The memory address.
Full name	The symbol name; often the same as the contents of the Symbol column but differs for example for C++ member functions.

Click the column headers to sort the list by symbol name, location, or full name.

Context menu

This context menu is available:

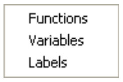


Figure 35: Symbols window context menu

These commands are available:

Functions	Toggles the display of function symbols on or off in the list.
Variables	Toggles the display of variables on or off in the list.
Labels	Toggles the display of labels on or off in the list.

Resolve Symbol Ambiguity dialog box

The **Resolve Symbol Ambiguity** dialog box appears, for example, when you specify a symbol in the Disassembly window to go to, and there are several instances of the same symbol due to templates or function overloading.

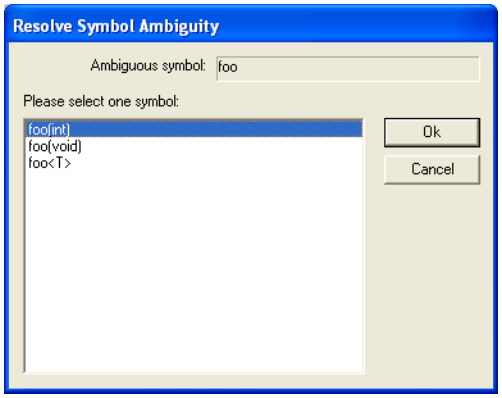


Figure 36: Resolve Symbol Ambiguity dialog box

Ambiguous symbol

Indicates which symbol that is ambiguous.

Please select one symbol

A list of possible matches for the ambiguous symbol. Select the one you want to use.

Using breakpoints

This chapter describes breakpoints and the various ways to define and monitor them. More specifically, this means:

- Introduction to setting and using breakpoints
- Procedures for setting breakpoints
- Reference information on breakpoints.

Introduction to setting and using breakpoints

This section introduces breakpoints.

These topics are covered:

- Reasons for using breakpoints
- Briefly about setting breakpoints
- Breakpoint types
- Breakpoint icons
- Breakpoints in the C-SPY simulator
- Breakpoints in the C-SPY hardware drivers
- Breakpoint consumers.

REASONS FOR USING BREAKPOINTS

C-SPY® lets you set various types of breakpoints in the application you are debugging, allowing you to stop at locations of particular interest. You can set a breakpoint at a *code* location to investigate whether your program logic is correct, or to get trace printouts. In addition to code breakpoints, and depending on what C-SPY driver you are using, additional breakpoint types might be available. For example, you might be able to set a *data* breakpoint, to investigate how and when the data changes.

You can let the execution stop under certain *conditions*, which you specify. You can also let the breakpoint trigger a *side effect*, for instance executing a C-SPY macro function, by transparently stopping the execution and then resuming. The macro function can be defined to perform a wide variety of actions, for instance, simulating hardware behavior.

All these possibilities provide you with a flexible tool for investigating the status of your application.

BRIEFLY ABOUT SETTING BREAKPOINTS

You can set breakpoints in many various ways, allowing for different levels of interaction, precision, timing, and automation. All the breakpoints you define will appear in the Breakpoints window. From this window you can conveniently view all breakpoints, enable and disable breakpoints, and open a dialog box for defining new breakpoints. The **Breakpoint Usage** dialog box also lists all internally used breakpoints, see *Breakpoint consumers*, page 94.

Breakpoints are set with a higher precision than single lines, using the same mechanism as when stepping; for more information about the precision, see *Single stepping*, page 56.

You can set breakpoints while you edit your code even if no debug session is active. The breakpoints will then be validated when the debug session starts. Breakpoints are preserved between debug sessions.

Note: For most hardware debugger systems it is only possible to set breakpoints when the application is not executing.

BREAKPOINT TYPES

Depending on the C-SPY driver you are using, C-SPY supports different types of breakpoints.

Code breakpoints

Code breakpoints are used for code locations to investigate whether your program logic is correct or to get trace printouts. Code breakpoints are triggered when an instruction is fetched from the specified location. If you have set the breakpoint on a specific machine instruction, the breakpoint will be triggered and the execution will stop, before the instruction is executed.

Log breakpoints

Log breakpoints provide a convenient way to add trace printouts without having to add any code to your application source code. Log breakpoints are triggered when an instruction is fetched from the specified location. If you have set the breakpoint on a specific machine instruction, the breakpoint will be triggered and the execution will temporarily stop and print the specified message in the C-SPY Debug Log window.

Trace breakpoints

Trace Start and Stop breakpoints start and stop trace data collection—a convenient way to analyze instructions between two execution points.

Data breakpoints

Data breakpoints are primarily useful for variables that have a fixed address in memory. If you set a breakpoint on an accessible local variable, the breakpoint is set on the corresponding memory location. The validity of this location is only guaranteed for small parts of the code. Data breakpoints are triggered when data is accessed at the specified location. The execution will usually stop directly after the instruction that accessed the data has been executed.

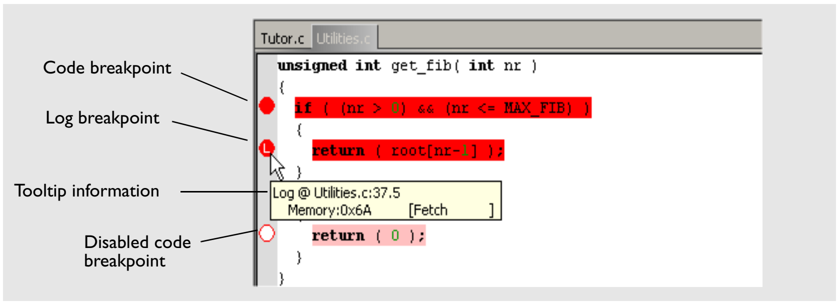
Immediate breakpoints

The C-SPY Simulator lets you set *immediate* breakpoints, which will halt instruction execution only temporarily. This allows a C-SPY macro function to be called when the simulated processor is about to read data from a location or immediately after it has written data. Instruction execution will resume after the action.

This type of breakpoint is useful for simulating memory-mapped devices of various kinds (for instance serial ports and timers). When the simulated processor reads from a memory-mapped location, a C-SPY macro function can intervene and supply appropriate data. Conversely, when the simulated processor writes to a memory-mapped location, a C-SPY macro function can act on the value that was written.

BREAKPOINT ICONS

A breakpoint is marked with an icon in the left margin of the editor window, and the icon varies with the type of breakpoint:



```
unsigned int get_fib( int nr )
{
  if ( (nr > 0) && (nr <= MAX_FIB) )
  {
    return ( root[nr-1] );
  }
    return ( 0 );
  }
}
```

```
Log @ Utilities.c:37.5
  Memory:0x6A        [Fetch        ]
```

Figure 37: Breakpoint icons

If the breakpoint icon does not appear, make sure the option **Show bookmarks** is selected, see Editor options in the *IDE Project Management and Building Guide*.

Just point at the breakpoint icon with the mouse pointer to get detailed tooltip information about all breakpoints set on the same location. The first row gives user breakpoint information, the following rows describe the physical breakpoints used for implementing the user breakpoint. The latter information can also be seen in the **Breakpoint Usage** dialog box.

Note: The breakpoint icons might look different for the C-SPY driver you are using.

BREAKPOINTS IN THE C-SPY SIMULATOR

The C-SPY simulator supports all breakpoint types and you can set an unlimited amount of breakpoints.

BREAKPOINTS IN THE C-SPY HARDWARE DRIVERS

Using the C-SPY drivers for hardware debugger systems you can set various breakpoint types. The amount of breakpoints you can set depends on the number of *hardware breakpoints* available on the target system or—if the driver and the device support them—whether your target system uses *software breakpoints*, in which case the number of breakpoints you can set is unlimited.

A software breakpoint instruction temporarily replaces the application code with an instruction that hands the execution over to the driver. There are several ways of doing this, for example:

- Inserting an `LCALL #monitor` instruction, if you are using the IAR ROM-monitor
- Using an exception operation code, for example `0xA5`, to halt the execution.

This table summarizes the characteristics of breakpoints for the different target systems:

C-SPY hardware driver	Code and Log breakpoints	Data breakpoints
Texas Instruments		
using 4 hardware breakpoints*	4	—
FS2 System Navigator		
using 2 hardware breakpoints*	2	2
using software breakpoints†	Unlimited	—
Infineon		
using 4 hardware breakpoints	4	4
using software breakpoints	Unlimited	—
Nordic Semiconductor		
using 2 hardware breakpoints*	2	2
ROM-monitor – depends on the device		
Analog Devices		

Table 8: Available breakpoints in C-SPY hardware drivers

C-SPY hardware driver	Code and Log breakpoints	Data breakpoints
using software breakpoints	Unlimited	—
Silicon Laboratories		
using 4 hardware breakpoints	4	4

Table 8: Available breakpoints in C-SPY hardware drivers (Continued)

*** The number of available hardware breakpoints depends on the target system you are using.**
† For code in RAM.

If the driver and the device support software breakpoints and they are enabled, the debugger will first use any available hardware breakpoints before using software breakpoints. Exceeding the number of available hardware breakpoints, if software breakpoints do not exist or are not enabled, causes the debugger to single step. This will significantly reduce the execution speed. For this reason you must be aware of the different breakpoint consumers.

Padding for safe insertion of breakpoint instruction(s)

When using the `LCALL` instruction as a code breakpoint, for example as in the IAR C-SPY ROM-monitor, padding with extra memory space might be needed to avoid overwriting application memory. In an assembler program this padding must be done manually. In C programs you can use the compiler option `--rom_mon_bp_padding`. See the *IAR C/C++ Compiler Reference Guide for 8051* for reference information about this option.

To set the equivalent option in the IAR Embedded Workbench IDE, choose **Project>Options>C/C++ Compiler>Code>Padding for ROM-monitor breakpoints**.

Using this option makes it possible to set a breakpoint on every C statement.

Breakpoints in flash memory

When you set a software breakpoint in flash memory, the driver must flash the page(s) containing the breakpoint instruction byte(s) once.

If you set a conditional breakpoint, the driver must flash the page(s) every time the breakpoint is evaluated to check if the condition is met.

Every step you take at C level forces the driver to temporarily set breakpoints on each possible endup statement.

Note: The Analog Devices driver will cache breakpoints, and it will not flash the page until the execution has started.

BREAKPOINTS AND INTERRUPTS

If an interrupt becomes active when C-SPY is processing a breakpoint, C-SPY will stop at the first instruction of the interrupt service routine.

BREAKPOINT CONSUMERS

A debugger system includes several consumers of breakpoints.

User breakpoints

The breakpoints you define in the breakpoint dialog box or by toggling breakpoints in the editor window often consume one physical breakpoint each, but this can vary greatly. Some user breakpoints consume several physical breakpoints and conversely, several user breakpoints can share one physical breakpoint. User breakpoints are displayed in the same way both in the **Breakpoint Usage** dialog box and in the Breakpoints window, for example `Data @[R] callCount`.

C-SPY itself

C-SPY itself also consumes breakpoints. C-SPY will set a breakpoint if:

- The debugger option **Run to** has been selected, and any step command is used. These are temporary breakpoints which are only set when the debugger system is running. This means that they are not visible in the Breakpoints window.
- The linker option **With I/O emulation modules** has been selected.

These types of breakpoint consumers are displayed in the **Breakpoint Usage** dialog box, for example, `C-SPY Terminal I/O & libsupport module`.

In the CLIB runtime environment, C-SPY will set a breakpoint if:

- the library functions `putchar` and `getchar` are used (low-level routines used by functions like `printf` and `scanf`)
- the application has an `exit` label.

You can disable the setting of system breakpoints on the `putchar` and `getchar` functions and on the `exit` label; see *Exclude system breakpoints on*, page 264.

In the DLIB runtime environment, C-SPY will set a system breakpoint on the `__DebugBreak` label.

C-SPY plugin modules

For example, modules for real-time operating systems can consume additional breakpoints. Specifically, by default, the Stack window consumes one physical breakpoint.

To disable the breakpoint used by the Stack window:

1 Choose **Tools>Options>Stack**.

2 Deselect the **Stack pointer(s) not valid until program reaches:** ***label*** option.

You can also disable the Stack window entirely by disabling the Stack plugin on the **Project>Options>Debugger>Plugins** page.

Procedures for setting breakpoints

This section gives you step-by-step descriptions about how to set and use breakpoints.

More specifically, you will get information about:

- Various ways to set a breakpoint
- Toggling a simple code breakpoint
- Setting breakpoints using the dialog box
- Setting a data breakpoint in the Memory window
- Setting breakpoints using system macros
- Useful breakpoint hints.

VARIOUS WAYS TO SET A BREAKPOINT

You can set a breakpoint in various ways:

- Using the **Toggle Breakpoint** command toggles a code breakpoint. This command is available both from the **Tools** menu and from the context menus in the editor window and in the Disassembly window.
- Right-clicking in the left-side margin of the editor window or the Disassembly window toggles a code breakpoint.
- Using the **New Breakpoints** dialog box and the **Edit Breakpoints** dialog box available from the context menus in the editor window, Breakpoints window, and in the Disassembly window. The dialog boxes give you access to all breakpoint options.
- Setting a data breakpoint on a memory area directly in the Memory window.
- Using predefined system macros for setting breakpoints, which allows automation.

The different methods offer different levels of simplicity, complexity, and automation.

TOGGLING A SIMPLE CODE BREAKPOINT

Toggling a code breakpoint is a quick method of setting a breakpoint. The following methods are available both in the editor window and in the Disassembly window:

- Double-click in the gray left-side margin of the window
- Place the insertion point in the C source statement or assembler instruction where you want the breakpoint, and click the **Toggle Breakpoint** button in the toolbar
- Choose **Edit>Toggle Breakpoint**
- Right-click and choose **Toggle Breakpoint** from the context menu.

SETTING BREAKPOINTS USING THE DIALOG BOX

The advantage of using a breakpoint dialog box is that it provides you with a graphical interface where you can interactively fine-tune the characteristics of the breakpoints. You can set the options and quickly test whether the breakpoint works according to your intentions.

All breakpoints you define using a breakpoint dialog box are preserved between debug sessions.

To set a new breakpoint:

You can open the dialog box from the context menu available in the editor window, Breakpoints window, and in the Disassembly window.

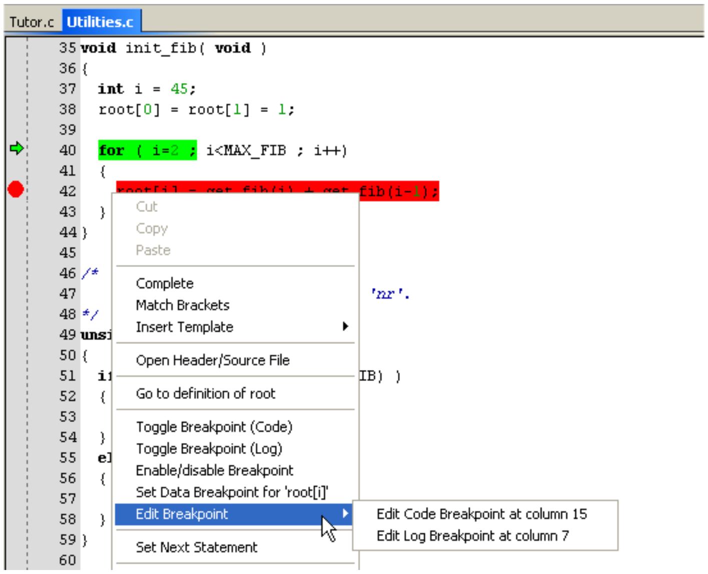
1. Choose **View>Breakpoints** to open the Breakpoints window.
2. In the Breakpoints window, right-click, and choose **New Breakpoint** from the context menu.
3. On the submenu, choose the breakpoint type you want to set.

   Depending on the C-SPY driver you are using, different breakpoint types are available.
4. In the breakpoint dialog box that appears, specify the breakpoint settings and click **OK**.

   The breakpoint is displayed in the Breakpoints window.

To modify an existing breakpoint:

1. In the Breakpoints window, editor window, or in the Disassembly window, select the breakpoint you want to modify and right-click to open the context menu.



```
void init_fib( void )
{
  int i = 45;
  root[0] = root[1] = 1;

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

/*
                                   'nr'.
*/
unsi
{
  i                                IB) )
  {

  }
  el
  {

  }
}

```

Figure 38: Modifying breakpoints via the context menu

If there are several breakpoints on the same source code line, the breakpoints will be listed on a submenu.

2. On the context menu, choose the appropriate command.

3. In the breakpoint dialog box that appears, specify the breakpoint settings and click **OK**.

   The breakpoint is displayed in the Breakpoints window.

SETTING A DATA BREAKPOINT IN THE MEMORY WINDOW

You can set breakpoints directly on a memory location in the Memory window. Right-click in the window and choose the breakpoint command from the context menu that appears. To set the breakpoint on a range, select a portion of the memory contents.

The breakpoint is not highlighted in the Memory window; instead, you can see, edit, and remove it using the Breakpoints window, which is available from the **View** menu. The breakpoints you set in the Memory window will be triggered for both read and write accesses. All breakpoints defined in this window are preserved between debug sessions.

Note: Setting breakpoints directly in the Memory window is only possible if the driver you use supports this.

SETTING BREAKPOINTS USING SYSTEM MACROS

You can set breakpoints not only in the breakpoint dialog box but also by using built-in C-SPY system macros. When you use system macros for setting breakpoints, the breakpoint characteristics are specified as macro parameters.

Macros are useful when you have already specified your breakpoints so that they fully meet your requirements. You can define your breakpoints in a macro file, using built-in system macros, and execute the file at C-SPY startup. The breakpoints will then be set automatically each time you start C-SPY. Another advantage is that the debug session will be documented, and that several engineers involved in the development project can share the macro files.

Note: If you use system macros for setting breakpoints, you can still view and modify them in the Breakpoints window. In contrast to using the dialog box for defining breakpoints, all breakpoints that are defined using system macros are removed when you exit the debug session.

These breakpoint macros are available:

C-SPY macro for breakpoints	Simulator	TI	FS2	Infineon	Nordic Semi	ROM-monitor	Analog Devices	Silabs
`__setCodeBreak`	x	x	x	x	x	x	x	x
`__setDataBreak`	x	—	x	x	—	x	—	x
`__setLogBreak`	x	x	x	x	x	x	x	x
`__setSimBreak`	x	—	—	—	—	—	—	—
`__setTraceStartBreak`	x	—	—	—	—	—	—	—
`__setTraceStopBreak`	x	—	—	—	—	—	—	—
`__clearBreak`	x	x	x	x	x	x	x	x

Table 9: C-SPY macros for breakpoints

For information about each breakpoint macro, see *Reference information on C-SPY system macros*, page 191.

Setting breakpoints at C-SPY startup using a setup macro file

You can use a setup macro file to define breakpoints at C-SPY startup. Follow the procedure described in *Registering and executing using setup macros and setup files*, page 182.

USEFUL BREAKPOINT HINTS

Below are some useful hints related to setting breakpoints.

Tracing incorrect function arguments

If a function with a pointer argument is sometimes incorrectly called with a `NULL` argument, you might want to debug that behavior. These methods can be useful:

- Set a breakpoint on the first line of the function with a condition that is true only when the parameter is `0`. The breakpoint will then not be triggered until the problematic situation actually occurs. The advantage of this method is that no extra source code is needed. The drawback is that the execution speed might become unacceptably low.
- You can use the `assert` macro in your problematic function, for example:

  ```
  int MyFunction(int * MyPtr)
  {
    assert(MyPtr != 0); /* Assert macro added to your source
                           code. */
    /* Here comes the rest of your function. */
  }
  ```

  The execution will break whenever the condition is true. The advantage is that the execution speed is only very slightly affected, but the drawback is that you will get a small extra footprint in your source code. In addition, the only way to get rid of the execution stop is to remove the macro and rebuild your source code.
- Instead of using the `assert` macro, you can modify your function like this:

  ```
  int MyFunction(int * MyPtr)
  {
    if(MyPtr == 0)
       MyDummyStatement; /* Dummy statement where you set a
                            breakpoint. */
    /* Here comes the rest of your function. */
  }
  ```

  You must also set a breakpoint on the extra dummy statement, so that the execution will break whenever the condition is true. The advantage is that the execution speed is only very slightly affected, but the drawback is that you will still get a small extra footprint in your source code. However, in this way you can get rid of the execution stop by just removing the breakpoint.

Performing a task and continuing execution

You can perform a task when a breakpoint is triggered and then automatically continue execution.

You can use the **Action** text box to associate an action with the breakpoint, for instance a C-SPY macro function. When the breakpoint is triggered and the execution of your application has stopped, the macro function will be executed. In this case, the execution will not continue automatically.

Instead, you can set a condition which returns `0` (false). When the breakpoint is triggered, the condition—which can be a call to a C-SPY macro that performs a task—is evaluated and because it is not true, execution continues.

Consider this example where the C-SPY macro function performs a simple task:

```
__var my_counter;

count()
{
  my_counter += 1;
  return 0;
}
```

To use this function as a condition for the breakpoint, type `count()` in the **Expression** text box under **Conditions**. The task will then be performed when the breakpoint is triggered. Because the macro function `count` returns `0`, the condition is false and the execution of the program will resume automatically, without any stop.

Reference information on breakpoints

This section gives reference information about these windows and dialog boxes:

- *Breakpoints window*, page 101
- *Breakpoint Usage dialog box*, page 102
- *Code breakpoints dialog box*, page 103
- *Log breakpoints dialog box*, page 105
- *Data breakpoints dialog box*, page 106
- *Immediate breakpoints dialog box*, page 108
- *Enter Location dialog box*, page 109
- *Resolve Source Ambiguity dialog box*, page 111.

See also:

- *Reference information on C-SPY system macros*, page 191
- *Reference information on trace*, page 136.

Breakpoints window

The Breakpoints window is available from the **View** menu.

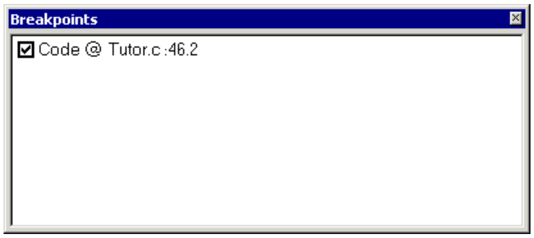


Figure 39: Breakpoints window

The Breakpoints window lists all breakpoints you define.

Use this window to conveniently monitor, enable, and disable breakpoints; you can also define new breakpoints and modify existing breakpoints.

Display area

This area lists all breakpoints you define. For each breakpoint, information about the breakpoint type, source file, source line, and source column is provided.

Context menu

This context menu is available:

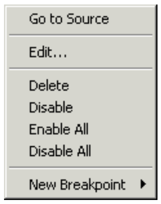


Figure 40: Breakpoints window context menu

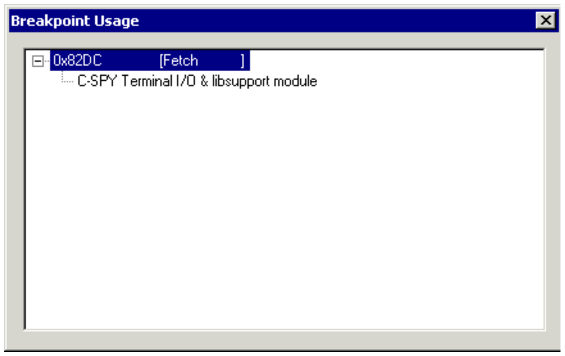
These commands are available:

Go to Source	Moves the insertion point to the location of the breakpoint, if the breakpoint has a source location. Double-click a breakpoint in the Breakpoints window to perform the same command.

Edit	Opens the breakpoint dialog box for the breakpoint you selected.
Delete	Deletes the breakpoint. Press the Delete key to perform the same command.
Enable	Enables the breakpoint. The check box at the beginning of the line will be selected. You can also perform the command by manually selecting the check box. This command is only available if the breakpoint is disabled.
Disable	Disables the breakpoint. The check box at the beginning of the line will be deselected. You can also perform this command by manually deselecting the check box. This command is only available if the breakpoint is enabled.
Enable All	Enables all defined breakpoints.
Disable All	Disables all defined breakpoints.
New Breakpoint	Displays a submenu where you can open the breakpoint dialog box for the available breakpoint types. All breakpoints you define using this dialog box are preserved between debug sessions.

Breakpoint Usage dialog box

The **Breakpoint Usage** dialog box is available from the menu specific to the C-SPY driver you are using.



```
0x82DC        [Fetch       ]
    C-SPY Terminal I/O & libsupport module
```

Figure 41: Breakpoint Usage dialog box

The **Breakpoint Usage** dialog box lists all breakpoints currently set in the target system, both the ones you have defined and the ones used internally by C-SPY. The format of the items in this dialog box depends on the C-SPY driver you are using.

The dialog box gives a low-level view of all breakpoints, related but not identical to the list of breakpoints displayed in the Breakpoints window.

C-SPY uses breakpoints when stepping. If your target system has a limited number of hardware breakpoints and is not using software breakpoints, exceeding the number of available hardware breakpoints will cause the debugger to single step. This will significantly reduce the execution speed. Therefore, in a debugger system with a limited amount of hardware breakpoints, you can use the **Breakpoint Usage** dialog box for:

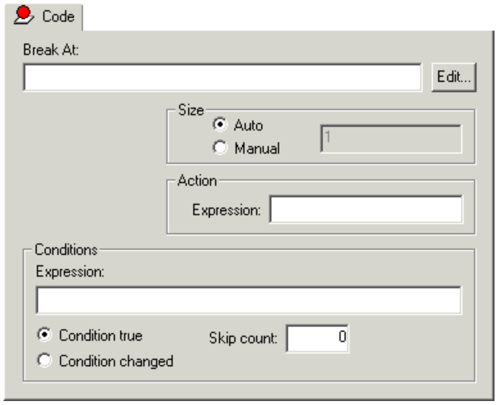
- Identifying all breakpoint consumers
- Checking that the number of active breakpoints is supported by the target system
- Configuring the debugger to use the available breakpoints in a better way, if possible.

Display area

For each breakpoint in the list, the address and access type are displayed. Each breakpoint in the list can also be expanded to show its originator.

Code breakpoints dialog box

The **Code** breakpoints dialog box is available from the context menu in the editor window, Breakpoints window, and in the Disassembly window.


Code

Break At:

Size: [✓ Auto] Manual 1

Action Expression:

Conditions Expression:

[✓ Condition true] Condition changed

Skip count: 0

Figure 42: Code breakpoints dialog box

Use the **Code** breakpoints dialog box to set a code breakpoint.

Note: The **Code** breakpoints dialog box depends on the C-SPY driver you are using. This figure reflects the C-SPY simulator. For information about support for breakpoints in the C-SPY driver you are using, see *Breakpoints in the C-SPY hardware drivers*, page 92.

Break At

Specify the location of the breakpoint in the text box. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 109.

Size

Determines whether there should be a size—in practice, a range—of locations where the breakpoint will trigger. Each fetch access to the specified memory range will trigger the breakpoint. Select how to specify the size:

Auto	The size will be set automatically, typically to 1.
Manual	Specify the size of the breakpoint range in the text box

Action

Determines whether there is an action connected to the breakpoint. Specify an expression, for instance a C-SPY macro function, which is evaluated when the breakpoint is triggered and the condition is true.

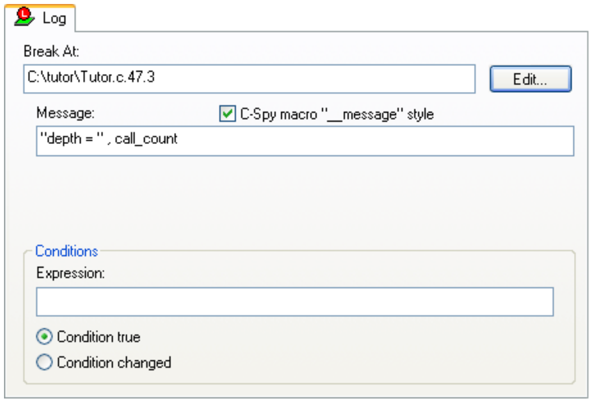
Conditions

Specify simple or complex conditions:

Expression	Specify a valid expression conforming to the C-SPY expression syntax.
Condition true	The breakpoint is triggered if the value of the expression is true.
Condition changed	The breakpoint is triggered if the value of the expression has changed since it was last evaluated.
Skip count	The number of times that the breakpoint condition must be fulfilled before the breakpoint starts triggering. After that, the breakpoint will trigger every time the condition is fulfilled.

Log breakpoints dialog box

The **Log** breakpoints dialog box is available from the context menu in the editor window, Breakpoints window, and in the Disassembly window.



Break At:
C:\tutor\Tutor.c.47.3

Message: ☑ C-Spy macro "__message" style
"depth = " , call_count

Expression:

(●) Condition true
( ) Condition changed

Figure 43: Log breakpoints dialog box

Use the **Log** breakpoints dialog box to set a log breakpoint.

Note: The **Log** breakpoints dialog box depends on the C-SPY driver you are using. This figure reflects the C-SPY simulator. For information about support for breakpoints in the C-SPY driver you are using, see *Breakpoints in the C-SPY hardware drivers*, page 92.

Break At

Specify the location of the breakpoint. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 109.

Message

Specify the message you want to be displayed in the C-SPY Debug Log window. The message can either be plain text, or—if you also select the option **C-SPY macro "__message" style**—a comma-separated list of arguments.

C-SPY macro "__message" style

Select this option to make a comma-separated list of arguments specified in the Message text box be treated exactly as the arguments to the C-SPY macro language statement `__message`, see *Formatted output*, page 188.

Conditions

Specify simple or complex conditions:

Expression	Specify a valid expression conforming to the C-SPY expression syntax.
Condition true	The breakpoint is triggered if the value of the expression is true.
Condition changed	The breakpoint is triggered if the value of the expression has changed since it was last evaluated.

Data breakpoints dialog box

The **Data** breakpoints dialog box is available from the context menu in the editor window, Breakpoints window, the Memory window, and in the Disassembly window.

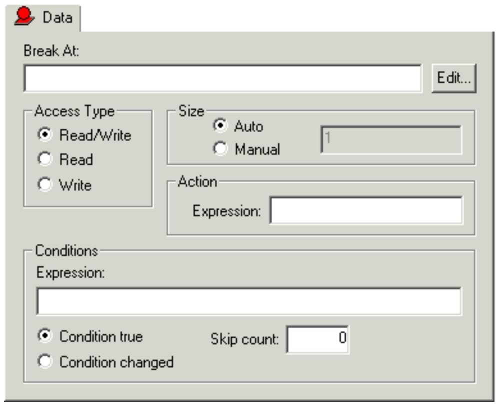


Figure 44: Data breakpoints dialog box

Use the **Data** breakpoints dialog box to set a data breakpoint. Data breakpoints never stop execution within a single instruction. They are recorded and reported after the instruction is executed.

Note: The **Data** breakpoints dialog box depends on the C-SPY driver you are using. This figure reflects the C-SPY simulator. For information about support for breakpoints in the C-SPY driver you are using, see *Breakpoints in the C-SPY hardware drivers*, page 92.

Break At

Specify the location for the breakpoint in the **Break At** text box. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 109.

Access Type

Selects the type of memory access that triggers data breakpoints:

Read/Write	Reads from or writes to location.
Read	Reads from location.
Write	Writes to location.

Size

Determines whether there should be a size—in practice, a range—of locations where the breakpoint will trigger. Each fetch access to the specified memory range will trigger the breakpoint. For data breakpoints, this can be useful if you want the breakpoint to be triggered on accesses to data structures, such as arrays, structs, and unions. Select between two different ways to specify the size:

Auto	The size will automatically be based on the type of expression the breakpoint is set on. For example, if you set the breakpoint on a 12-byte structure, the size of the breakpoint will be 12 bytes.
Manual	Specify the size of the breakpoint range in the text box.

Action

Determines whether there is an action connected to the breakpoint. Specify an expression, for instance a C-SPY macro function, which is evaluated when the breakpoint is triggered and the condition is true.

Conditions

Specify simple or complex conditions:

Expression	Specify a valid expression conforming to the C-SPY expression syntax.
Condition true	The breakpoint is triggered if the value of the expression is true.

Condition changed	The breakpoint is triggered if the value of the expression has changed since it was last evaluated.
Skip count	The number of times that the breakpoint condition must be fulfilled before a break occurs (integer).

Immediate breakpoints dialog box

The **Immediate** breakpoints dialog box is available from the context menu in the editor window, Breakpoints window, the Memory window, and in the Disassembly window.

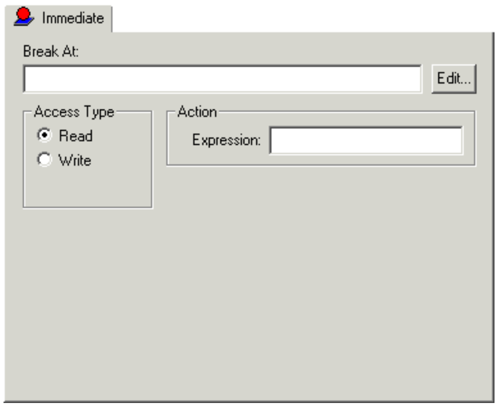


Figure 45: Immediate breakpoints dialog box

In the C-SPY simulator, use the **Immediate** breakpoints dialog box to set an immediate breakpoint. Immediate breakpoints do not stop execution at all; they only suspend it temporarily.

Break At

Specify the location for the breakpoint in the **Break At** text box. Alternatively, click the **Edit** button to open the **Enter Location** dialog box; see *Enter Location dialog box*, page 109.

Access Type

Selects the type of memory access that triggers immediate breakpoints:

Read	Reads from location.
Write	Writes to location.

Action

Determines whether there is an action connected to the breakpoint. Specify an expression, for instance a C-SPY macro function, which is evaluated when the breakpoint is triggered and the condition is true.

Enter Location dialog box

The **Enter Location** dialog box is available from the breakpoints dialog box, either when you set a new breakpoint or when you edit a breakpoint.

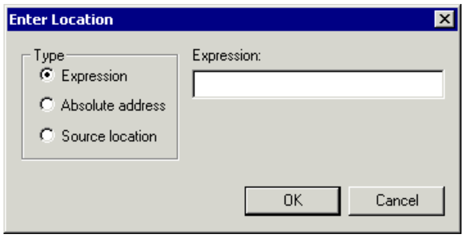


Figure 46: Enter Location dialog box

Use the **Enter Location** dialog box to specify the location of the breakpoint.

Note: This dialog box looks different depending on the **Type** you select.

Type

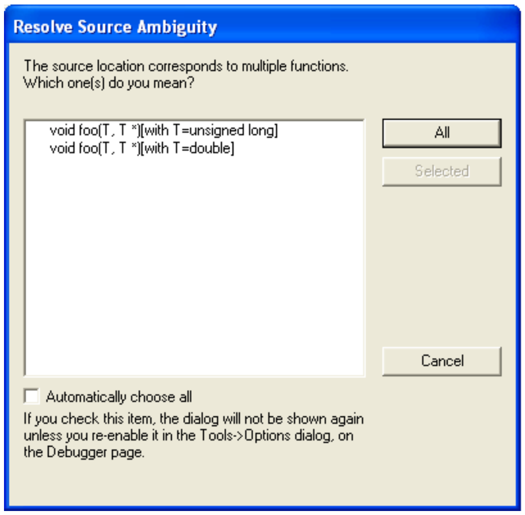
Selects the type of location to be used for the breakpoint:

Expression	Any expression that evaluates to a valid address, such as a function or variable name. Code breakpoints are set on functions, for example `main`. Data breakpoints are set on variable names. For example, `my_var` refers to the location of the variable `my_var`, and `arr[3]` refers to the location of the third element of the array `arr`. For static variables declared with the same name in several functions, use the syntax `my_func::my_static_variable` to refer to a specific variable. For more information about C-SPY expressions, see *C-SPY expressions*, page 74.

Absolute address	An absolute location on the form `zone:hexaddress` or simply `hexaddress` (for example `Memory:0x42`). `zone` refers to C-SPY memory zones and specifies in which memory the address belongs.
Source location	A location in your C source code using the syntax: `{filename}.row.column.` `filename` specifies the filename and full path. `row` specifies the row in which you want the breakpoint. `column` specifies the column in which you want the breakpoint. For example, `{C:\src\prog.c}.22.3` sets a breakpoint on the third character position on line 22 in the source file `Utilities.c`. Note that the Source location type is usually meaningful only for code breakpoints.

Resolve Source Ambiguity dialog box

The **Resolve Source Ambiguity** dialog box appears, for example, when you try to set a breakpoint on inline functions or templates, and the source location corresponds to more than one function.


Resolve Source Ambiguity

The source location corresponds to multiple functions.
Which one(s) do you mean?

void foo(T, T *)[with T=unsigned long]
void foo(T, T *)[with T=double]

Automatically choose all

If you check this item, the dialog will not be shown again unless you re-enable it in the Tools->Options dialog, on the Debugger page.

Figure 47: Resolve Source Ambiguity dialog box

To resolve a source ambiguity, perform one of these actions:

- In the text box, select one or several of the listed locations and click **Selected**.
- Click **All**.

All

The breakpoint will be set on all listed locations.

Selected

The breakpoint will be set on the source locations that you have selected in the text box.

Cancel

No location will be used.

Automatically choose all

Determines that whenever a specified source location corresponds to more than one function, all locations will be used.

Note that this option can also be specified in the **IDE Options** dialog box, see Debugger options in the *IDE Project Management and Building Guide*.

Monitoring memory and registers

This chapter describes how to use the features available in C-SPY® for examining memory and registers. More specifically, this means information about:

- Introduction to monitoring memory and registers
- Reference information on memory and registers.

Introduction to monitoring memory and registers

This section covers these topics:

- Briefly about monitoring memory and registers
- C-SPY memory zones
- Stack display
- Memory access checking.

BRIEFLY ABOUT MONITORING MEMORY AND REGISTERS

C-SPY provides many windows for monitoring memory and registers, each of them available from the **View** menu:

- The Memory window

  Gives an up-to-date display of a specified area of memory—a memory zone—and allows you to edit it. Different colors are used for indicating data coverage along with execution of your application. You can fill specified areas with specific values and you can set breakpoints directly on a memory location or range. You can open several instances of this window, to monitor different memory areas. The content of the window can be regularly updated while your application is executing.
- The Symbolic memory window

  Displays how variables with static storage duration are laid out in memory. This can be useful for better understanding memory usage or for investigating problems caused by variables being overwritten, for example by buffer overruns.

- The Stack window

  Displays the contents of the stack, including how stack variables are laid out in memory. In addition, some integrity checks of the stack can be performed to detect and warn about problems with stack overflow. For example, the Stack window is useful for determining the optimal size of the stack. You can open several instances of this window, each showing different stacks or different display modes of the same stack.
- The Register window

  Gives an up-to-date display of the contents of the processor registers and SFRs, and allows you to edit them. Due to the large amount of registers—memory-mapped peripheral unit registers and CPU registers—it is inconvenient to show all registers concurrently in the Register window. Instead you can divide registers into *register groups*. You can choose to load either predefined register groups or define your own application-specific groups. You can open several instances of this window, each showing a different register group.

To view the memory contents for a specific variable, simply drag the variable to the Memory window or the Symbolic memory window. The memory area where the variable is located will appear.

Reading the value of some registers might influence the runtime behavior of your application. For example, reading the value of a UART status register might reset a pending bit, which leads to the lack of an interrupt that would have processed a received byte. To prevent this from happening, make sure that the Register window containing any such registers is closed when debugging a running application.

C-SPY MEMORY ZONES

In C-SPY, the term *zone* is used for a named memory area. A memory address, or *location*, is a combination of a zone and a numerical offset into that zone. By default, the 8051 architecture has four zones—`Code`, `XData`, `IData`, and `SFR`—that cover the whole 8051 memory range.

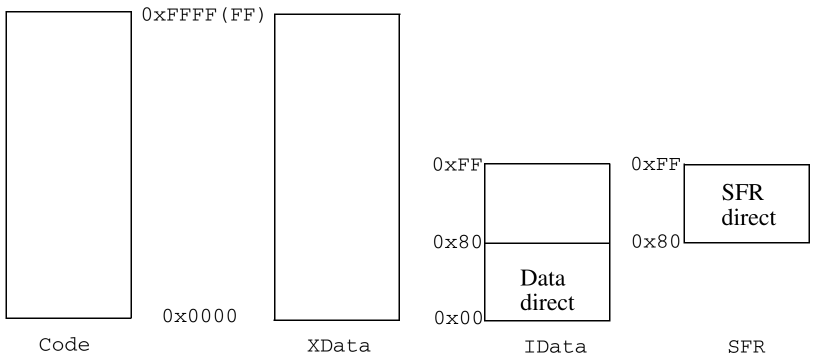


Figure 48: Zones in C-SPY

Memory zones are used in several contexts, most importantly in the Memory and Disassembly windows. Use the **Zone** box in these windows to choose which memory zone to display.

STACK DISPLAY

The Stack window displays the contents of the stack, overflow warnings, and it has a graphical stack bar. These can be useful in many contexts. Some examples are:

- Investigating the stack usage when assembler modules are called from C modules and vice versa
- Investigating whether the correct elements are located on the stack
- Investigating whether the stack is restored properly
- Determining the optimal stack size
- Detecting stack overflows.

For microcontrollers with multiple stacks, you can select which stack to view.

Stack usage

When your application is first loaded, and upon each reset, the memory for the stack area is filled with the dedicated byte value `0xCD` before the application starts executing. Whenever execution stops, the stack memory is searched from the end of the stack until a byte with a value different from `0xCD` is found, which is assumed to be how far the stack has been used. Although this is a reasonably reliable way to track stack usage, there is no guarantee that a stack overflow is detected. For example, a stack *can* incorrectly grow outside its bounds, and even modify memory outside the stack area, without actually modifying any of the bytes near the stack range. Likewise, your application might modify memory within the stack area by mistake.

The Stack window cannot detect a stack overflow when it happens, but can only detect the signs it leaves behind. However, when the graphical stack bar is enabled, the functionality needed to detect and warn about stack overflows is also enabled.

Note: The size and location of the stack is retrieved from the definition of the segment holding the stack, made in the linker configuration file. If you, for some reason, modify the stack initialization made in the system startup code, `cstartup`, you should also change the segment definition in the linker configuration file accordingly; otherwise the Stack window cannot track the stack usage. For more information about this, see the *IAR C/C++ Compiler Reference Guide for 8051*.

MEMORY ACCESS CHECKING

The C-SPY simulator can simulate various memory access types of the target hardware and detect illegal accesses, for example a read access to write-only memory. If a memory access occurs that does not agree with the access type specified for the specific memory area, C-SPY will regard this as an illegal access. Also, a memory access to memory which is not defined is regarded as an illegal access. The purpose of memory access checking is to help you to identify any memory access violations.

The memory areas can either be the zones predefined in the device description file, or memory areas based on the segment information available in the debug file. In addition to these, you can define your own memory areas. The access type can be read and write, read-only, or write-only. You cannot map two different access types to the same memory area. You can check for access type violation and accesses to unspecified ranges. Any violations are logged in the Debug Log window. You can also choose to have the execution halted.

Reference information on memory and registers

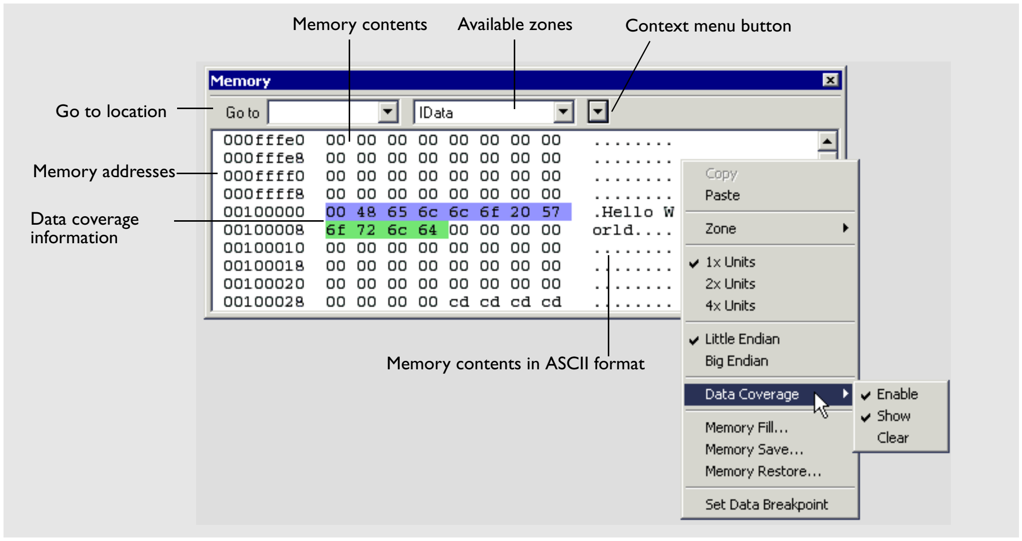
This section gives reference information about these windows and dialog boxes:

- *Memory window*, page 117
- *Memory Save dialog box*, page 120
- *Memory Restore dialog box*, page 121
- *Fill dialog box*, page 122
- *Symbolic Memory window*, page 123
- *Stack window*, page 125
- *Register window*, page 128
- *Memory Access Setup dialog box*, page 129

- *Edit Memory Access dialog box*, page 131.

Memory window

The Memory window is available from the **View** menu.



```
000fffe0  00 00 00 00 00 00 00 00   ........
000fffe8  00 00 00 00 00 00 00 00   ........
000ffff0  00 00 00 00 00 00 00 00   ........
000ffff8  00 00 00 00 00 00 00 00   ........
00100000  00 48 65 6c 6c 6f 20 57   .Hello W
00100008  6f 72 6c 64 00 00 00 00   orld....
00100010  00 00 00 00 00 00 00 00   ........
00100018  00 00 00 00 00 00 00 00   ........
00100020  00 00 00 00 00 00 00 00   ........
00100028  00 00 00 00 cd cd cd cd   ........
```

Figure 49: Memory window

This window gives an up-to-date display of a specified area of memory—a memory zone—and allows you to edit it. You can open several instances of this window, which is very convenient if you want to keep track of several memory or register zones, or monitor different parts of the memory.

To view the memory corresponding to a variable, you can select it in the editor window and drag it to the Memory window.

Toolbar

The toolbar contains:

Go to	The location you want to view. This can be a memory address, or the name of a variable, function, or label.
Zone display	Selects a memory zone to display, see *C-SPY memory zones*, page 114.

Context menu button	Displays the context menu, see *Context menu*, page 119.
Update Now	Updates the content of the Memory window while your application is executing. This button is only enabled if the C-SPY driver you are using has access to the target system memory while your application is executing.
Live Update	Updates the contents of the Memory window regularly while your application is executing. This button is only enabled if the C-SPY driver you are using has access to the target system memory while your application is executing. To set the update frequency, specify an appropriate frequency in the **IDE Options>Debugger** dialog box.

Display area

The display area shows the addresses currently being viewed, the memory contents in the format you have chosen, and—provided that the display mode is set to **1x Units**—the memory contents in ASCII format. You can edit the contents of the display area, both in the hexadecimal part and the ASCII part of the area.

Data coverage is displayed with these colors:

Yellow	Indicates data that has been read.
Blue	Indicates data that has been written
Green	Indicates data that has been both read and written.

Note: Data coverage is not supported by all C-SPY drivers. Data coverage is supported by the C-SPY Simulator.

Context menu

This context menu is available:

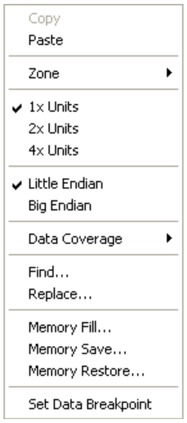


Figure 50: Memory window context menu

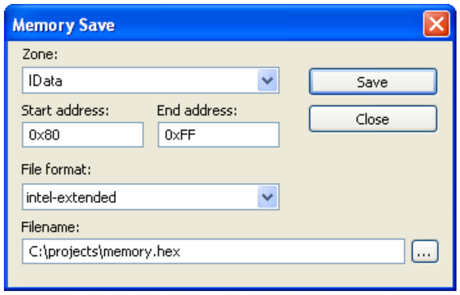
These commands are available:

Copy, **Paste**	Standard editing commands.
Zone	Selects a memory zone to display, see *C-SPY memory zones*, page 114.
1x Units	Displays the memory contents in units of 8 bits.
2x Units	Displays the memory contents in units of 16 bits.
4x Units	Displays the memory contents in units of 32 bits.
Little Endian	Displays the contents in little-endian byte order.
Big Endian	Displays the contents in big-endian byte order.
Data Coverage	Choose between: **Enable** toggles data coverage on or off. **Show** toggles between showing or hiding data coverage. **Clear** clears all data coverage information. These commands are only available if your C-SPY driver supports data coverage.

Find	Displays a dialog box where you can search for text within the Memory window; read about the **Find** dialog box in the *IDE Project Management and Building Guide*.
Replace	Displays a dialog box where you can search for a specified string and replace each occurrence with another string; read about the **Replace** dialog box in the *IDE Project Management and Building Guide*.
Memory Fill	Displays a dialog box, where you can fill a specified area with a value, see *Fill dialog box*, page 122.
Memory Save	Displays a dialog box, where you can save the contents of a specified memory area to a file, see *Memory Save dialog box*, page 120.
Memory Restore	Displays a dialog box, where you can load the contents of a file in Intel-hex or Motorola s-record format to a specified memory zone, see *Memory Restore dialog box*, page 121.
Set Data Breakpoint	Sets breakpoints directly in the Memory window. The breakpoint is not highlighted; you can see, edit, and remove it in the Breakpoints dialog box. The breakpoints you set in this window will be triggered for both read and write access. For more information, see *Setting a data breakpoint in the Memory window*, page 97.

Memory Save dialog box

The **Memory Save** dialog box is available by choosing **Debug>Memory>Save** or from the context menu in the Memory window.



Zone: IData

Start address: 0x80

End address: 0xFF

File format: intel-extended

Filename: C:\projects\memory.hex

Figure 51: Memory Save dialog box

Use this dialog box to save the contents of a specified memory area to a file.

Zone

Selects a memory zone.

Start address

Specify the start address of the memory range to be saved.

End address

Specify the end address of the memory range to be saved.

File format

Selects the file format to be used, which is Intel-extended by default.

Filename

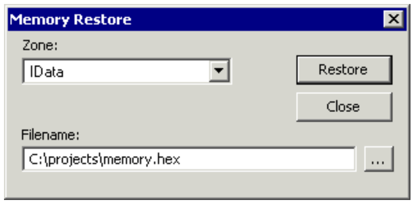
Specify the destination file to be used; a browse button is available for your convenience.

Save

Saves the selected range of the memory zone to the specified file.

Memory Restore dialog box

The **Memory Restore** dialog box is available by choosing **Debug>Memory>Restore** or from the context menu in the Memory window.



Zone: IData

Filename: C:\projects\memory.hex

Figure 52: Memory Restore dialog box

Use this dialog box to load the contents of a file in Intel-extended or Motorola S-record format to a specified memory zone.

Zone

Selects a memory zone.

Filename

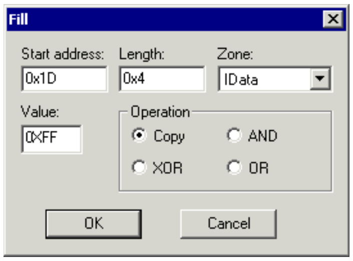
Specify the file to be read; a browse button is available for your convenience.

Restore

Loads the contents of the specified file to the selected memory zone.

Fill dialog box

The **Fill** dialog box is available from the context menu in the Memory window.



Start address: 0x1D
Length: 0x4
Zone: IData
Value: 0XFF
Operation: [✓ Copy] AND XOR OR

Figure 53: Fill dialog box

Use this dialog box to fill a specified area of memory with a value.

Start address

Type the start address—in binary, octal, decimal, or hexadecimal notation.

Length

Type the length—in binary, octal, decimal, or hexadecimal notation.

Zone

Selects a memory zone.

Value

Type the 8-bit value to be used for filling each memory location.

Operation

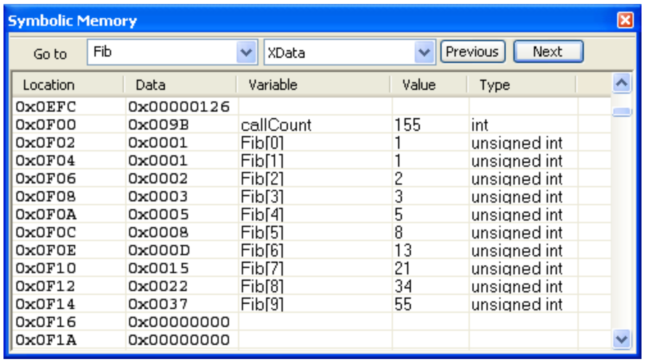
These are the available memory fill operations:

Copy	**Value** will be copied to the specified memory area.
AND	An AND operation will be performed between **Value** and the existing contents of memory before writing the result to memory.

XOR	An `XOR` operation will be performed between **Value** and the existing contents of memory before writing the result to memory.
OR	An `OR` operation will be performed between **Value** and the existing contents of memory before writing the result to memory.

Symbolic Memory window

The Symbolic Memory window is available from the **View** menu when the debugger is running.



Location	Data	Variable	Value	Type
0x0EFC	0x00000126			
0x0F00	0x009B	callCount	155	int
0x0F02	0x0001	Fib[0]	1	unsigned int
0x0F04	0x0001	Fib[1]	1	unsigned int
0x0F06	0x0002	Fib[2]	2	unsigned int
0x0F08	0x0003	Fib[3]	3	unsigned int
0x0F0A	0x0005	Fib[4]	5	unsigned int
0x0F0C	0x0008	Fib[5]	8	unsigned int
0x0F0E	0x000D	Fib[6]	13	unsigned int
0x0F10	0x0015	Fib[7]	21	unsigned int
0x0F12	0x0022	Fib[8]	34	unsigned int
0x0F14	0x0037	Fib[9]	55	unsigned int
0x0F16	0x00000000			
0x0F1A	0x00000000			

Figure 54: Symbolic Memory window

This window displays how variables with static storage duration, typically variables with file scope but also static variables in functions and classes, are laid out in memory. This can be useful for better understanding memory usage or for investigating problems caused by variables being overwritten, for example buffer overruns. Other areas of use are spotting alignment holes or for understanding problems caused by buffers being overwritten.

To view the memory corresponding to a variable, you can select it in the editor window and drag it to the Symbolic Memory window.

Toolbar

The toolbar contains:

Go to	The memory location or symbol you want to view.

Zone display	Selects a memory zone to display, see *C-SPY memory zones*, page 114.
Previous	Highlights the previous symbol in the display area.
Next	Highlights the next symbol in the display area.

Display area

This area contains these columns:

Location	The memory address.
Data	The memory contents in hexadecimal format. The data is grouped according to the size of the symbol. This column is editable.
Variable	The variable name; requires that the variable has a fixed memory location. Local variables are not displayed.
Value	The value of the variable. This column is editable.
Type	The type of the variable.

There are several different ways to navigate within the memory space:

- Text that is dropped in the window is interpreted as symbols
- The scroll bar at the right-side of the window
- The toolbar buttons **Next** and **Previous**
- The toolbar list box **Go to** can be used for locating specific locations or symbols.

Note: Rows are marked in red when the corresponding value has changed.

Context menu

This context menu is available:

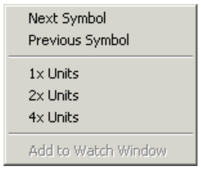


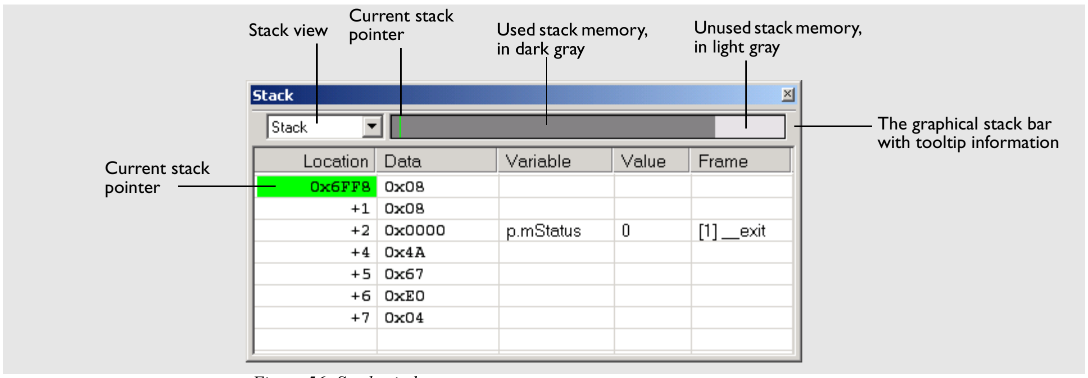
Figure 55: Symbolic Memory window context menu

These commands are available:

Next Symbol	Highlights the next symbol in the display area.
Previous Symbol	Highlights the previous symbol in the display area.
1x Units	Displays the memory contents in units of 8 bits. This applies only to rows which do not contain a variable.
2x Units	Displays the memory contents in units of 16 bits.
4x Units	Displays the memory contents in units of 32 bits.
Add to Watch Window	Adds the selected symbol to the Watch window.

Stack window

The Stack window is available from the **View** menu.


Stack view
Current stack pointer
Used stack memory, in dark gray
Unused stack memory, in light gray
The graphical stack bar with tooltip information
Current stack pointer
Stack
Stack

Location	Data	Variable	Value	Frame
0x6FF8	0x08			
+1	0x08			
+2	0x0000	p.mStatus	0	[1] __exit
+4	0x4A			
+5	0x67			
+6	0xE0			
+7	0x04			

Figure 56: Stack window

This window is a memory window that displays the contents of the stack. In addition, some integrity checks of the stack can be performed to detect and warn about problems with stack overflow. For example, the Stack window is useful for determining the optimal size of the stack.

To view the graphical stack bar:

1. Choose **Tools>Options>Stack**.
2. Select the option **Enable graphical stack display and stack usage**.

You can open several Stack windows, each showing a different stack—if several stacks are available—or the same stack with different display settings.

Note: By default, this window uses one physical breakpoint. For more information, see *Breakpoint consumers*, page 94.

For information about options specific to the Stack window, see the *IDE Project Management and Building Guide.*

Toolbar

Stack	Selects which stack to view. This applies to microcontrollers with multiple stacks.

The graphical stack bar

Displays the state of the stack graphically.

The left end of the stack bar represents the bottom of the stack, in other words, the position of the stack pointer when the stack is empty. The right end represents the end of the memory space reserved for the stack. The graphical stack bar turns red when the stack usage exceeds a threshold that you can specify.

When the stack bar is enabled, the functionality needed to detect and warn about stack overflows is also enabled.

Place the mouse pointer over the stack bar to get tooltip information about stack usage.

Display area

This area contains these columns:

Location	Displays the location in memory. The addresses are displayed in increasing order. If a stack grows toward high addresses, the top of that stack will consequently be located at the bottom of the window. The address referenced by the stack pointer, in other words the top of the stack, is highlighted in a green color.
Data	Displays the contents of the memory unit at the given location. From the Stack window context menu, you can select how the data should be displayed; as a 1-, 2-, or 4-byte group of data.
Variable	Displays the name of a variable, if there is a local variable at the given location. Variables are only displayed if they are declared locally in a function, and located on the stack and not in registers.
Value	Displays the value of the variable that is displayed in the **Variable** column.

Frame	Displays the name of the function that the call frame corresponds to.

Context menu

This context menu is available:

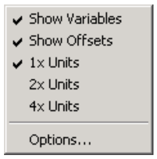


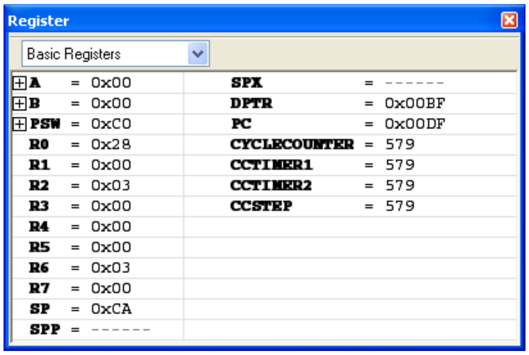
Figure 57: Stack window context menu

These commands are available:

Show variables	Displays separate columns named **Variables**, **Value**, and **Frame** in the Stack window. Variables located at memory addresses listed in the Stack window are displayed in these columns.
Show offsets	Displays locations in the **Location** column as offsets from the stack pointer. When deselected, locations are displayed as absolute addresses.
1x Units	Displays data in the **Data** column as single bytes.
2x Units	Displays data in the **Data** column as 2-byte groups.
4x Units	Displays data in the **Data** column as 4-byte groups.
Options	Opens the **IDE Options** dialog box where you can set options specific to the Stack window, see the *IDE Project Management and Building Guide*.

Register window

The Register window is available from the **View** menu.



```
Register
Basic Registers
⊞A    = 0x00        SPX           = ------
⊞B    = 0x00        DPTR          = 0x00BF
⊞PSW  = 0xC0        PC            = 0x00DF
 R0   = 0x28        CYCLECOUNTER  = 579
 R1   = 0x00        CCTIMER1      = 579
 R2   = 0x03        CCTIMER2      = 579
 R3   = 0x00        CCSTEP        = 579
 R4   = 0x00
 R5   = 0x00
 R6   = 0x03
 R7   = 0x00
 SP   = 0xCA
 SPP  = ------
```

Figure 58: Register window

This window gives an up-to-date display of the contents of the processor registers and special function registers, and allows you to edit their contents. Optionally, you can choose to load either predefined register groups or to define your own application-specific groups

You can open several instances of this window, which is very convenient if you want to keep track of different register groups.

To enable predefined register groups:

1. Select a device description file that suits your device, see *Selecting a device description file*, page 41.
2. The register groups appear in the Register window, provided that they are defined in the device description file. Note that the available register groups are also listed on the **Register Filter** page.

To define application-specific register groups, read about register filter options in the *IDE Project Management and Building Guide*.

Toolbar

CPU Registers	Selects which register group to display, by default CPU Registers. Additional register groups are predefined in the device description files that make all SFR registers available in the register window. The device description file contains a section that defines the special function registers and their groups.

Display area

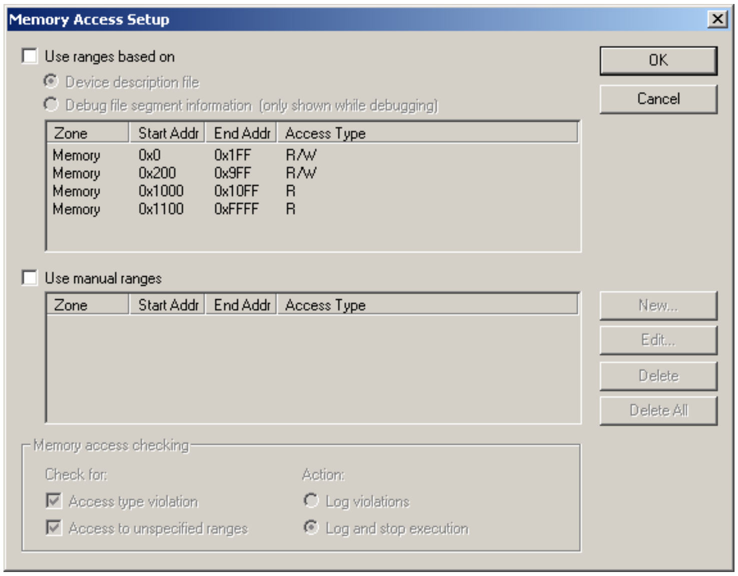
Displays registers and their values. Every time C-SPY stops, a value that has changed since the last stop is highlighted. To edit the contents of a register, click it, and modify the value.

Some registers are expandable, which means that the register contains interesting bits or subgroups of bits.

To change the display format, change the **Base** setting on the **Register Filter** page—available by choosing **Tools>Options**.

Memory Access Setup dialog box

The **Memory Access Setup** dialog box is available from the **Simulator** menu.



Zone	Start Addr	End Addr	Access Type
Memory	0x0	0x1FF	R/W
Memory	0x200	0x9FF	R/W
Memory	0x1000	0x10FF	R
Memory	0x1100	0xFFFF	R

Figure 59: Memory Access Setup dialog box

This dialog box lists all defined memory areas, where each column in the list specifies the properties of the area. In other words, the dialog box displays the memory access setup that will be used during the simulation.

Note: If you enable both the **Use ranges based on** and the **Use manual ranges** option, memory accesses are checked for all defined ranges.

For information about the columns and the properties displayed, see *Edit Memory Access dialog box*, page 131.

Use ranges based on

Selects any of the predefined alternatives for the memory access setup. Choose between:

Device description file	Loads properties from the device description file.
Debug file segment information	Properties are based on the segment information available in the debug file. This information is only available while debugging. The advantage of using this option, is that the simulator can catch memory accesses outside the linked application.

Use manual ranges

Specify your own ranges manually via the **Edit Memory Access** dialog box. To open this dialog box, choose **New** to specify a new memory range, or select a memory zone and choose **Edit** to modify it. For more information, see *Edit Memory Access dialog box*, page 131.

The ranges you define manually are saved between debug sessions.

Memory access checking

Check for determines what to check for;

- Access type violation
- Access to unspecified ranges.

Action selects the action to be performed if an access violation occurs; choose between:

- Log violations
- Log and stop execution.

Any violations are logged in the Debug Log window.

Buttons

These buttons are available:

New	Opens the **Edit Memory Access** dialog box, where you can specify a new memory range and attach an access type to it, see *Edit Memory Access dialog box*, page 131.

Edit	Opens the **Edit Memory Access** dialog box, where you can edit the selected memory area. See *Edit Memory Access dialog box*, page 131.
Delete	Deletes the selected memory area definition.
Delete All	Deletes all defined memory area definitions.

Note: Except for the OK and Cancel buttons, buttons are only available when the option **Use manual ranges** is selected.

Edit Memory Access dialog box

The **Edit Memory Access** dialog box is available from the **Memory Access Setup** dialog box.

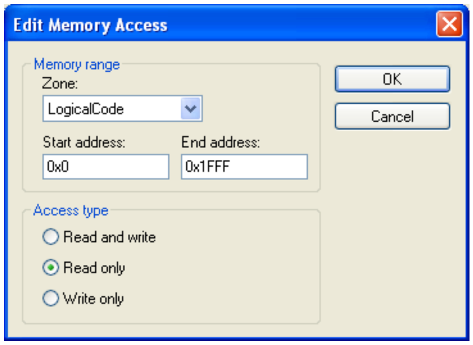


Figure 60: Edit Memory Access dialog box

Use this dialog box to specify the memory ranges, and assign an access type to each memory range, for which you want to detect illegal accesses during the simulation.

Memory range

Defines the memory area for which you want to check the memory accesses:

Zone	Selects a memory zone, see *C-SPY memory zones*, page 114.
Start address	Specify the start address for the address range, in hexadecimal notation.
End address	Specify the end address for the address range, in hexadecimal notation.

Access type

Selects an access type to the memory range; choose between:

- **Read and write**
- **Read only**
- **Write only**.

Collecting and using trace data

This chapter gives you information about collecting and using trace data in C-SPY®. More specifically, this means:

- Introduction to using trace
- Procedures for using trace
- Reference information on trace.

Introduction to using trace

This section introduces trace.

These topics are covered:

- Reasons for using trace
- Briefly about trace
- Requirements for using trace.

See also:

- *Using the profiler*, page 149.

REASONS FOR USING TRACE

By using trace, you can inspect the program flow up to a specific state, for instance an application crash, and use the trace data to locate the origin of the problem. Trace data can be useful for locating programming errors that have irregular symptoms and occur sporadically.

BRIEFLY ABOUT TRACE

Your target system must be able to generate trace data. Once generated, C-SPY can collect it and you can visualize and analyze the data in various windows and dialog boxes.

Trace features in C-SPY

In C-SPY, you can use the trace-related windows Trace, Function Trace, Timeline, and Find in Trace. In the C-SPY simulator, you can also use the Trace Expressions window. Depending on your C-SPY driver, you can set various types of trace breakpoints to control the collection of trace data.

In addition, several other features in C-SPY also use trace data, features such as the Profiler, Code coverage, and Instruction profiling.

REQUIREMENTS FOR USING TRACE

The C-SPY simulator supports trace-related functionality, and there are no specific requirements.

Trace data cannot be collected from the hardware debugger systems.

Procedures for using trace

This section gives you step-by-step descriptions about how to collect and use trace data.

More specifically, you will get information about:

- Getting started with trace in the C-SPY simulator
- Trace data collection using breakpoints
- Searching in trace data
- Browsing through trace data.

GETTING STARTED WITH TRACE IN THE C-SPY SIMULATOR

To collect trace data using the C-SPY simulator, no specific build settings are required.

To get started using trace:

1 After you have built your application and started C-SPY, choose **Simulator>Trace** to open the Trace window, and click the **Activate** button to enable collecting trace data.

2 Start the execution. When the execution stops, for instance because a breakpoint is triggered, trace data is displayed in the Trace window. For more information about the window, see *Trace window*, page 137.

TRACE DATA COLLECTION USING BREAKPOINTS

A convenient way to collect trace data between two execution points is to start and stop the data collection using dedicated breakpoints. Choose between these alternatives:

- In the editor or Disassembly window, position your insertion point, right-click, and toggle a **Trace Start** or **Trace Stop** breakpoint from the context menu.
- In the Breakpoints window, choose **Trace Start** or **Trace Stop**.
- The C-SPY system macros `__setTraceStartBreak` and `__setTraceStopBreak` can also be used.

For more information about these breakpoints, see *Trace Start breakpoints dialog box*, page 143 and *Trace Stop breakpoints dialog box*, page 144, respectively.

SEARCHING IN TRACE DATA

When you have collected trace data, you can perform searches in the collected data to locate the parts of your code or data that you are interested in, for example, a specific interrupt or accesses of a specific variable.

You specify the search criteria in the **Find in Trace** dialog box and view the result in the Find in Trace window.

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

To search in your trace data:

1 In the Trace window toolbar, click the **Find** button.

2 In the **Find in Trace** dialog box, specify your search criteria.

Typically, you can choose to search for:

- A specific piece of text, for which you can apply further search criteria
- An address range
- A combination of these, like a specific piece of text within a specific address range.

For more information about the different options, see *Find in Trace dialog box*, page 146.

3 When you have specified your search criteria, click **Find**. The Find in Trace window is displayed, which means you can start analyzing the trace data. For more information, see *Find in Trace window*, page 147.

BROWSING THROUGH TRACE DATA

To follow the execution history, simply look and scroll in the Trace window. Alternatively, you can enter *browse mode*.

To enter browse mode, double-click an item in the Trace window, or click the **Browse** toolbar button.

The selected item turns yellow and the source and disassembly windows will highlight the corresponding location. You can now move around in the trace data using the up and down arrow keys, or by scrolling and clicking; the source and Disassembly windows will be updated to show the corresponding location. This is like stepping backward and forward through the execution history.

Double-click again to leave browse mode.

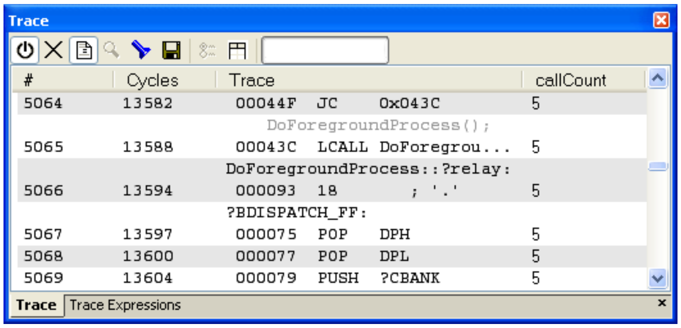
Reference information on trace

This section gives reference information about these windows and dialog boxes:

- *Trace window*, page 137
- *Function Trace window*, page 138
- *Timeline window*, page 139
- *Trace Start breakpoints dialog box*, page 143
- *Trace Stop breakpoints dialog box*, page 144
- *Trace Expressions window*, page 145
- *Find in Trace dialog box*, page 146
- *Find in Trace window*, page 147.

Trace window

The Trace window is available from the **Simulator** menu.



```
#       Cycles    Trace                                  callCount
5064    13582      00044F   JC      0x043C               5
                      DoForegroundProcess();
5065    13588      00043C   LCALL  DoForegrou...         5
                  DoForegroundProcess::?relay:
5066    13594      000093   18           ; '.'           5
                  ?BDISPATCH_FF:
5067    13597      000075   POP     DPH                  5
5068    13600      000077   POP     DPL                  5
5069    13604      000079   PUSH    ?CBANK               5
```

Figure 61: Trace window

This window displays a collected sequence of executed machine instructions. In addition, the window can display trace data for expressions.

Trace toolbar

The toolbar in the Trace window and in the Function trace window contains:

Enable/Disable	Enables and disables collecting and viewing trace data in this window. This button is not available in the Function trace window.
Clear trace data	Clears the trace buffer. Both the Trace window and the Function trace window are cleared.
Toggle source	Toggles the Trace column between showing only disassembly or disassembly together with the corresponding source code.
Browse	Toggles browse mode on or off for a selected item in the Trace window, see *Browsing through trace data*, page 136.
Find	Displays a dialog box where you can perform a search, see *Find in Trace dialog box*, page 146.
Save	Displays a standard **Save As** dialog box where you can save the collected trace data to a text file, with tab-separated columns.
Edit Settings	In the C-SPY simulator this button is not enabled.

Edit Expressions (C-SPY simulator only)	Opens the Trace Expressions window, see *Trace Expressions window*, page 145.

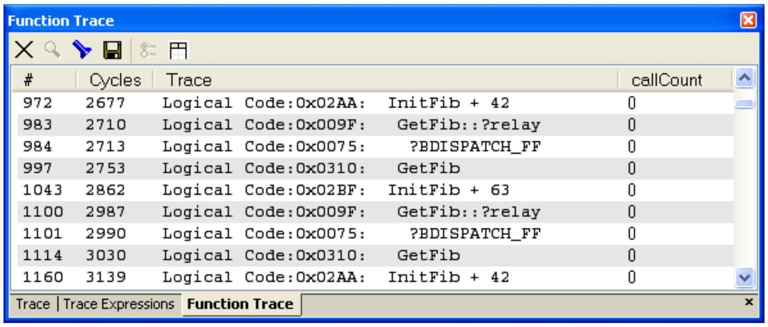
Display area

This area contains these columns:

#	A serial number for each row in the trace buffer. Simplifies the navigation within the buffer.
Cycles	The number of cycles elapsed to this point.
Trace	The collected sequence of executed machine instructions. Optionally, the corresponding source code can also be displayed.
Expression	Each expression you have defined to be displayed appears in a separate column. Each entry in the expression column displays the value *after* executing the instruction on the same row. You specify the expressions for which you want to collect trace data in the Trace Expressions window, see *Trace Expressions window*, page 145.

Function Trace window

The Function Trace window is available from the **Simulator** menu during a debug session.



```
#     Cycles  Trace                                         callCount
972   2677    Logical Code:0x02AA:  InitFib + 42            0
983   2710    Logical Code:0x009F:   GetFib::?relay         0
984   2713    Logical Code:0x0075:     ?BDISPATCH_FF        0
997   2753    Logical Code:0x0310:   GetFib                 0
1043  2862    Logical Code:0x02BF:  InitFib + 63            0
1100  2987    Logical Code:0x009F:   GetFib::?relay         0
1101  2990    Logical Code:0x0075:     ?BDISPATCH_FF        0
1114  3030    Logical Code:0x0310:   GetFib                 0
1160  3139    Logical Code:0x02AA:  InitFib + 42            0
```

Figure 62: Function Trace window

This window is available for the C-SPY simulator.

This window displays a subset of the trace data displayed in the Trace window. Instead of displaying all rows, the Function Trace window only shows trace data corresponding to calls to and returns from functions.

Toolbar

For information about the toolbar, see *Trace toolbar*, page 137.

Display area

For information about the columns in the display area, see *Display area*, page 138.

Timeline window

The Timeline window is available from the **Simulator** menu during a debug session.

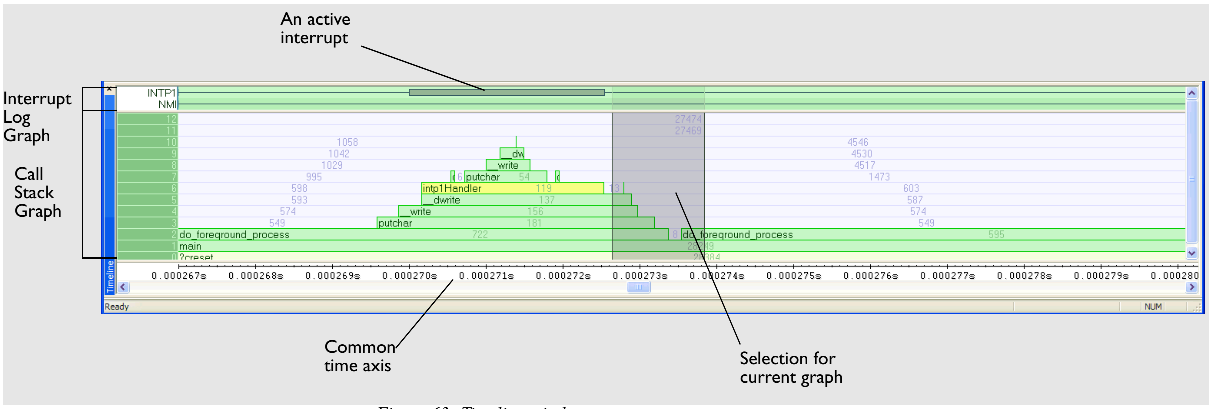


Figure 63: Timeline window

This window is available for the C-SPY simulator.

This window displays trace data for interrupt logs and for the call stack as graphs in relation to a common time axis.

Display area

The display area can be populated with an Interrupt Log Graph and a Call Stack Graph.

At the bottom of the window, there is a common time axis that uses seconds as the time unit.

Interrupt Log Graph

The Interrupt Log Graph displays interrupts reported by the C-SPY simulator. In other words, the graph provides a graphical view of the interrupt events during the execution of your application, where:

- The label area at the left end of the graph shows the names of the interrupts.
- The graph itself shows active interrupts as a thick green horizontal bar. This graph is a graphical representation of the information in the Interrupt Log window, see *Interrupt Log window*, page 173.

Call Stack Graph

The Call Stack Graph displays the sequence of calls and returns collected by trace. At the bottom of the graph you will usually find `main`, and above it, the functions called from `main`, and so on. The horizontal bars, which represent invocations of functions, use four different colors:

- Medium green for normal C functions with debug information
- Light green for functions known to the debugger only through an assembler label
- Medium or light yellow for interrupt handlers, with the same distinctions as for green.

The numbers represent the number of cycles spent in, or between, the function invocations.

Selection and navigation

Click and drag to select. The selection extends vertically over all graphs, but appears highlighted in a darker color for the selected graph. You can navigate backward and forward in the selected graph using the left and right arrow keys. Use the Home and End keys to move to the first or last relevant point, respectively. Use the navigation keys in combination with the Shift key to extend the selection.

Context menu

This context menu is available:

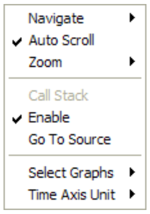


Figure 64: Timeline window context menu for the Call Stack Graph

Note: The context menu contains some commands that are common to all graphs and some commands that are specific to each graph. The figure reflects the context menu for the Call Stack Graph, which means that the menu looks slightly different for the other graphs.

These commands are available:

Navigate	All graphs	Commands for navigating over the graph(s); choose between: **Next** moves the selection to the next relevant point in the graph. Shortcut key: right arrow. **Previous** moves the selection backward to the previous relevant point in the graph. Shortcut key: left arrow. **First** moves the selection to the first data entry in the graph. Shortcut key: Home. **Last** moves the selection to the last data entry in the graph. Shortcut key: End. **End** moves the selection to the last data in any displayed graph, in other words the end of the time axis. Shortcut key: Ctrl+End.
Auto Scroll	All graphs	Toggles auto scrolling on or off. When on, the most recent collected data is automatically displayed.

Zoom	All graphs	Commands for zooming the window, in other words, changing the time scale; choose between: **Zoom to Selection** makes the current selection fit the window. Shortcut key: Return. **Zoom In** zooms in on the time scale. Shortcut key: +. **Zoom Out** zooms out on the time scale. Shortcut key: -. **10ns**, **100ns**, **1us**, etc makes an interval of 10 nanoseconds, 100 nanoseconds, 1 microsecond, respectively, fit the window. **1ms**, **10ms**, etc makes an interval of 1 millisecond or 10 milliseconds, respectively, fit the window. **10m**, **1h**, etc makes an interval of 10 minutes or 1 hour, respectively, fit the window.
Call Stack	Call Stack Graph	A heading that shows that the Call stack-specific commands below are available.
Interrupt	Interrupt Log Graph	A heading that shows that the Interrupt Log-specific commands below are available.
Enable	All graphs	Toggles the display of the graph on or off. If you disable a graph, that graph will be indicated as `OFF` in the Timeline window. If no trace data has been collected for a graph, *no data* will appear instead of the graph.
Go To Source	Common	Displays the corresponding source code in an editor window, if applicable.
Select Graphs	Common	Selects which graphs to be displayed in the Timeline window.
Time Axis Unit	Common	Selects the unit used in the time axis; choose between **Seconds** and **Cycles**.
Profile Selection	Common	Enables profiling time intervals in the Function Profiler window. Note that this command is only available if the C-SPY driver supports PC Sampling.

Trace Start breakpoints dialog box

The **Trace Start** dialog box is available from the context menu that appears when you right-click in the Breakpoints window.

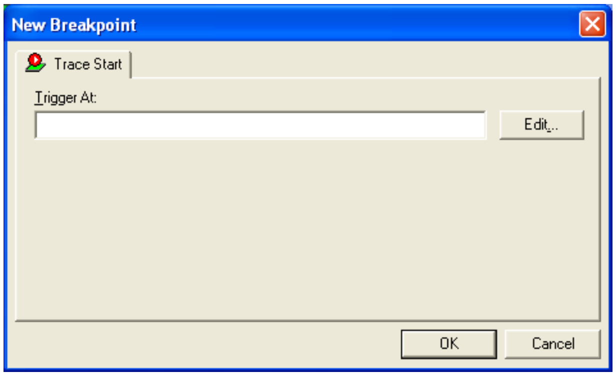


Figure 65: Trace Start breakpoints dialog box

This dialog box is available for the C-SPY simulator.

To set a Trace Start breakpoint:

1 In the editor or Disassembly window, right-click and choose **Trace Start** from the context menu.

Alternatively, open the Breakpoints window by choosing **View>Breakpoints**.

2 In the Breakpoints window, right-click and choose **New Breakpoint>Trace Start**.

Alternatively, to modify an existing breakpoint, select a breakpoint in the Breakpoints window and choose **Edit** on the context menu.

3 In the **Trigger At** text box, specify an expression, an absolute address, or a source location. Click **OK**.

4 When the breakpoint is triggered, the trace data collection starts.

Trigger At

Specify the location for the breakpoint in the text box. Alternatively, click the **Edit** browse button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 109.

Trace Stop breakpoints dialog box

The **Trace Stop** dialog box is available from the context menu that appears when you right-click in the Breakpoints window.

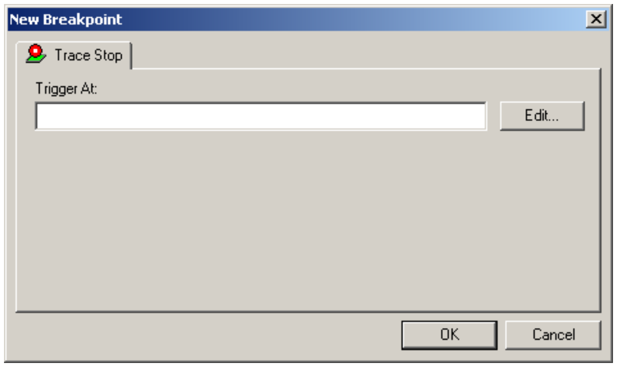


Figure 66: Trace Stop breakpoints dialog box

This dialog box is available for the C-SPY simulator.

To set a Trace Stop breakpoint:

1. In the editor or Disassembly window, right-click and choose **Trace Stop** from the context menu.

   Alternatively, open the Breakpoints window by choosing **View>Breakpoints**.

2. In the Breakpoints window, right-click and choose **New Breakpoint>Trace Stop**.

   Alternatively, to modify an existing breakpoint, select a breakpoint in the Breakpoints window and choose **Edit** on the context menu.

3. In the **Trigger At** text box, specify an expression, an absolute address, or a source location. Click **OK**.

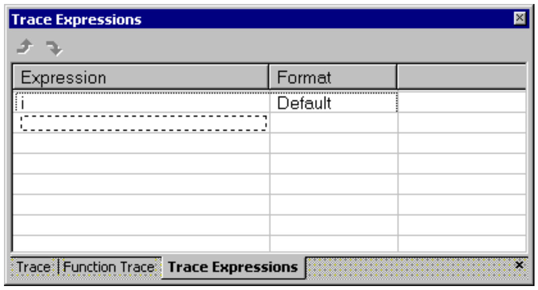
4. When the breakpoint is triggered, the trace data collection stops.

Trigger At

Specify the location for the breakpoint in the text box. Alternatively, click the **Edit** browse button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 109.

Trace Expressions window

The Trace Expressions window is available from the Trace window toolbar.



Expression	Format
i	Default

Figure 67: Trace Expressions window

This dialog box is available for the C-SPY simulator.

Use this window to specify, for example, a specific variable (or an expression) for which you want to collect trace data.

Toolbar

The toolbar buttons change the order between the expressions:

Arrow up	Moves the selected row up.
Arrow down	Moves the selected row down.

Display area

Use the display area to specify expressions for which you want to collect trace data:

Expression	Specify any expression that you want to collect data from. You can specify any expression that can be evaluated, such as variables and registers.
Format	Shows which display format that is used for each expression. Note that you can change display format via the context menu.

Each row in this area will appear as an extra column in the Trace window.

Find in Trace dialog box

The **Find in Trace** dialog box is available by clicking the **Find** button on the Trace window toolbar or by choosing **Edit>Find and Replace>Find**.

Note that the **Edit>Find and Replace>Find** command is context-dependent. It displays the **Find in Trace** dialog box if the Trace window is the current window or the **Find** dialog box if the editor window is the current window.

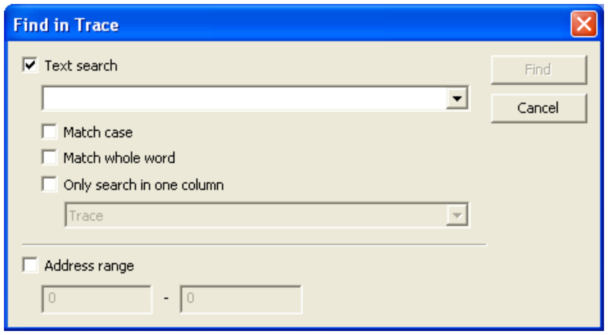


Figure 68: Find in Trace dialog box

This dialog box is available for the C-SPY simulator.

Use this dialog box to specify the search criteria for advanced searches in the trace data.

The search results are displayed in the Find in Trace window—available by choosing the **View>Messages** command, see *Find in Trace window*, page 147.

See also *Searching in trace data*, page 135.

Text search

Specify the string you want to search for. To specify the search criteria, choose between:

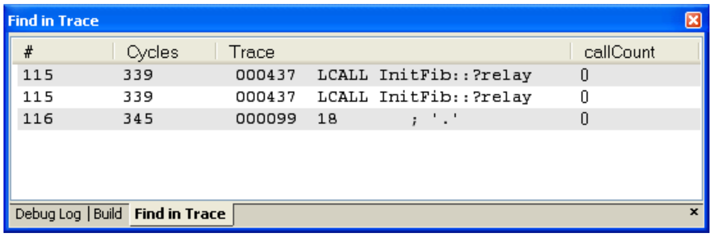
Match Case	Searches only for occurrences that exactly match the case of the specified text. Otherwise `int` will also find `INT` and `Int` and so on.
Match whole word	Searches only for the string when it occurs as a separate word. Otherwise `int` will also find `print`, `sprintf` and so on.
Only search in one column	Searches only in the column you selected from the drop-down list.

Address Range

Specify the address range you want to display or search. The trace data within the address range is displayed. If you also have specified a text string in the **Text search** field, the text string is searched for within the address range.

Find in Trace window

The Find in Trace window is available from the **View>Messages** menu. Alternatively, it is automatically displayed when you perform a search using the **Find in Trace** dialog box.



#	Cycles	Trace	callCount
115	339	000437 LCALL InitFib::?relay	0
115	339	000437 LCALL InitFib::?relay	0
116	345	000099 18 ; '.'	0

Figure 69: Find in Trace window

This dialog box is available for the C-SPY simulator.

This window displays the result of searches in the trace data. Double-click an item in the Find in Trace window to bring up the same item in the Trace window.

Before you can view any trace data, you must specify the search criteria in the **Find in Trace** dialog box, see *Find in Trace dialog box*, page 146.

For more information, see *Searching in trace data*, page 135.

Display area

The Find in Trace window looks like the Trace window and shows the same columns and data, but *only* those rows that match the specified search criteria.

Using the profiler

This chapter describes how to use the profiler in C-SPY®. More specifically, this means:

- Introduction to the profiler
- Procedures for using the profiler
- Reference information on the profiler.

Introduction to the profiler

This section introduces the profiler.

These topics are covered:

- Reasons for using the profiler
- Briefly about the profiler
- Requirements for using the profiler.

REASONS FOR USING THE PROFILER

Function profiling can help you find the functions in your source code where the most time is spent during execution. You should focus on those functions when optimizing your code. A simple method of optimizing a function is to compile it using speed optimization. Alternatively, you can move the data used by the function into the memory which uses the most efficient addressing mode. For detailed information about efficient memory usage, see the *IAR C/C++ Compiler Reference Guide for 8051*.

Instruction profiling can help you fine-tune your code on a very detailed level, especially for assembler source code. Instruction profiling can also help you to understand where your compiled C/C++ source code spends most of its time, and perhaps give insight into how to rewrite it for better performance.

BRIEFLY ABOUT THE PROFILER

Function profiling information is displayed in the Function Profiler window, that is, timing information for the functions in an application. Profiling must be turned on explicitly using a button on the window’s toolbar, and will stay enabled until it is turned off.

Instruction profiling information is displayed in the Disassembly window, that is, the number of times each instruction has been executed.

Profiling sources

The profiler can use different mechanisms, or *sources*, to collect profiling information. Depending on the available hardware features, one or more of the sources can be used for profiling:

- Trace (calls)

  The full instruction trace is analyzed to determine all function calls and returns. When the collected instruction sequence is incomplete or discontinuous, the profiling information is less accurate.
- Trace (flat)

  Each instruction in the full instruction trace or each PC Sample is assigned to a corresponding function or code fragment, without regard to function calls or returns. This is most useful when the application does not exhibit normal call/return sequences, such as when you are using an RTOS, or when you are profiling code which does not have full debug information.

REQUIREMENTS FOR USING THE PROFILER

The C-SPY simulator supports the profiler, and there are no specific requirements for using the profiler.

Profiling is not supported by the hardware debuggers.

Procedures for using the profiler

This section gives you step-by-step descriptions about how to use the profiler.

More specifically, you will get information about:

- Getting started using the profiler on function level
- Getting started using the profiler on instruction level.

GETTING STARTED USING THE PROFILER ON FUNCTION LEVEL

To display function profiling information in the Function Profiler window:

1 Make sure you build your application using these options:

Category	Setting
C/C++ Compiler	Output>Generate debug information
Linker	Output>Format>Debug information for C-SPY

Table 10: Project options for enabling the profiler

2 When you have built your application and started C-SPY, choose **Simulator>Function Profiler** to open the Function Profiler window, and click the **Enable** button to turn on the profiler. Alternatively, choose **Enable** from the context menu that is available when you right-click in the Function Profiler window.

3 Start executing your application to collect the profiling information.

4 Profiling information is displayed in the Function Profiler window. To sort, click on the relevant column header.

5 When you start a new sampling, you can click the **Clear** button—alternatively, use the context menu—to clear the data.

GETTING STARTED USING THE PROFILER ON INSTRUCTION LEVEL

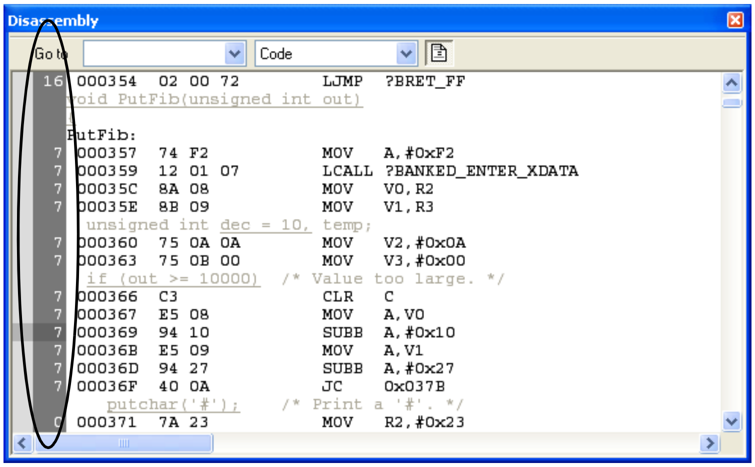
To display instruction profiling information in the Disassembly window:

1 When you have built your application and started C-SPY, choose **View>Disassembly** to open the Disassembly window, and choose **Enable** from the context menu that is available when you right-click in the Profiler window.

2 Make sure that the **Show** command on the context menu is selected, to display the profiling information.

3 Start executing your application to collect the profiling information.

4 When the execution stops, for instance because the program exit is reached or a breakpoint is triggered, you can view instruction level profiling information in the left-hand margin of the Disassembly window.



```
16 000354   02 00 72        LJMP   ?BRET_FF
   void PutFib(unsigned int out)
   {
   PutFib:
 7 000357   74 F2           MOV    A,#0xF2
 7 000359   12 01 07        LCALL  ?BANKED_ENTER_XDATA
 7 00035C   8A 08           MOV    V0,R2
 7 00035E   8B 09           MOV    V1,R3
    unsigned int dec = 10, temp;
 7 000360   75 0A 0A        MOV    V2,#0x0A
 7 000363   75 0B 00        MOV    V3,#0x00
    if (out >= 10000)  /* Value too large. */
 7 000366   C3              CLR    C
 7 000367   E5 08           MOV    A,V0
 7 000369   94 10           SUBB   A,#0x10
 7 00036B   E5 09           MOV    A,V1
 7 00036D   94 27           SUBB   A,#0x27
 7 00036F   40 0A           JC     0x037B
      putchar('#');    /* Print a '#'. */
 0 000371   7A 23           MOV    R2,#0x23
```

Figure 70: Instruction count in Disassembly window

For each instruction, the number of times it has been executed is displayed.

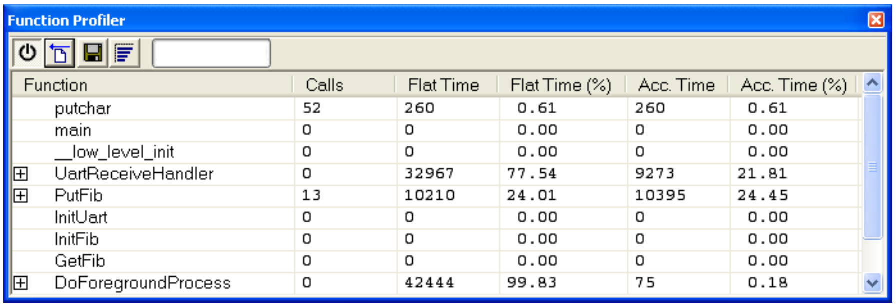
Reference information on the profiler

This section gives reference information about these windows and dialog boxes:

- *Function Profiler window*, page 153
- *Disassembly window*, page 61

Function Profiler window

The Function Profiler window is available from the **Simulator** menu.



Function Profiler

Function	Calls	Flat Time	Flat Time (%)	Acc. Time	Acc. Time (%)
putchar	52	260	0.61	260	0.61
main	0	0	0.00	0	0.00
__low_level_init	0	0	0.00	0	0.00
⊞ UartReceiveHandler	0	32967	77.54	9273	21.81
⊞ PutFib	13	10210	24.01	10395	24.45
InitUart	0	0	0.00	0	0.00
InitFib	0	0	0.00	0	0.00
GetFib	0	0	0.00	0	0.00
⊞ DoForegroundProcess	0	42444	99.83	75	0.18

Figure 71: Function Profiler window

This window displays function profiling information.

Toolbar

The toolbar contains:

⏻	**Enable/Disable**	Enables or disables the profiler.
	Clear	Clears all profiling data.
	Save	Opens a standard **Save As** dialog box where you can save the contents of the window to a file, with tab-separated columns. Only non-expanded rows are included in the list file.
	Graphical view	Overlays the values in the percentage columns with a graphical bar.
	Progress bar	Displays a backlog of profiling data that is still being processed. If the rate of incoming data is higher than the rate of the profiler processing the data, a backlog is accumulated. The progress bar indicates that the profiler is still processing data, but also approximately how far the profiler has come in the process. Note that because the profiler consumes data at a certain rate and the target system supplies data at another rate, the amount of data remaining to be processed can both increase and decrease. The progress bar can grow and shrink accordingly.

Display area

The content in the display area depends on which source that is used for the profiling information:

- For the Trace (calls) source, the display area contains one line for each function compiled with debug information enabled. When some profiling information has been collected, it is possible to expand rows of functions that have called other functions. The child items for a given function list all the functions that have been called by the parent function and the corresponding statistics.
- For the Trace (flat) source, the display area contains one line for each C function of your application, but also lines for sections of code from the runtime library or from other code without debug information, denoted only by the corresponding assembler labels. Each executed `PC` address from trace data is treated as a separate sample and is associated with the corresponding line in the Profiling window. Each line contains a count of those samples.

More specifically, the display area provides information in these columns:

Function	All sources	The name of the profiled C function.
Calls	Trace (calls)	The number of times the function has been called.
Flat time	Trace (calls)	The time in cycles spent inside the function.
Flat time (%)	Trace (calls)	Flat time expressed as a percentage of the total time.
Acc. time	Trace (calls)	The time in cycles spent in this function and everything called by this function.
Acc. time (%)	Trace (calls)	Accumulated time expressed as a percentage of the total time.
PC Samples	Trace (flat)	The number of PC samples associated with the function.
PC Samples (%)	Trace (flat)	The number of PC samples associated with the function as a percentage of the total number of samples.

Context menu

This context menu is available:

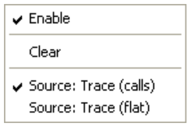


Figure 72: Function Profiler window context menu

These commands are available:

Enable	Enables the profiler. The system will collect information also when the window is closed.
Clear	Clears all profiling data.
Source	Selects which source to be used for the profiling information. Choose between: **Trace (calls)**—the instruction count for instruction profiling is only as complete as the collected trace data. **Trace (flat)**—the instruction count for instruction profiling is only as complete as the collected trace data.

Code coverage

This chapter describes the code coverage functionality in C-SPY®, which helps you verify whether all parts of your code have been executed. More specifically, this means:

- Introduction to code coverage
- Reference information on code coverage.

Introduction to code coverage

This section covers these topics:

- Reasons for using code coverage
- Briefly about code coverage
- Requirements for using code coverage.

REASONS FOR USING CODE COVERAGE

The code coverage functionality is useful when you design your test procedure to verify whether all parts of the code have been executed. It also helps you identify parts of your code that are not reachable.

BRIEFLY ABOUT CODE COVERAGE

The Code Coverage window reports the status of the current code coverage analysis. For every program, module, and function, the analysis shows the percentage of code that has been executed since code coverage was turned on up to the point where the application has stopped. In addition, all statements that have not been executed are listed. The analysis will continue until turned off.

REQUIREMENTS FOR USING CODE COVERAGE

Code coverage is not supported by the hardware debugger drivers. Code coverage is supported by the C-SPY Simulator.

Reference information on code coverage

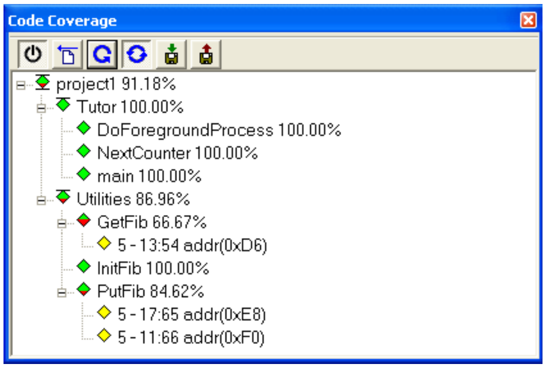
This section gives reference information about these windows and dialog boxes:

- *Code Coverage window*, page 158.

See also *Single stepping*, page 56.

Code Coverage window

The Code Coverage window is available from the **View** menu.



```
project1 91.18%
  Tutor 100.00%
    DoForegroundProcess 100.00%
    NextCounter 100.00%
    main 100.00%
  Utilities 86.96%
    GetFib 66.67%
      5 - 13:54 addr(0xD6)
    InitFib 100.00%
    PutFib 84.62%
      5 - 17:65 addr(0xE8)
      5 - 11:66 addr(0xF0)
```

Figure 73: Code Coverage window

This window reports the status of the current code coverage analysis. For every program, module, and function, the analysis shows the percentage of code that has been executed since code coverage was turned on up to the point where the application has stopped. In addition, all statements that have not been executed are listed. The analysis will continue until turned off.

An asterisk (*) in the title bar indicates that C-SPY has continued to execute, and that the Code Coverage window must be refreshed because the displayed information is no longer up to date. To update the information, use the **Refresh** command.

To get started using code coverage:

- Before using the code coverage functionality you must build your application using these options:

Category	Setting
C/C++ Compiler	Output>Generate debug information

Table 11: Project options for enabling code coverage

Category	Setting
Linker	Output>Debug information for C-SPY
Debugger	Plugins>Code Coverage

Table 11: Project options for enabling code coverage

2 After you have built your application and started C-SPY, choose **View>Code Coverage** to open the Code Coverage window.

3 Click the **Activate** button, alternatively choose **Activate** from the context menu, to switch on code coverage.

4 Start the execution. When the execution stops, for instance because the program exit is reached or a breakpoint is triggered, click the **Refresh** button to view the code coverage information.

Display area

The code coverage information is displayed in a tree structure, showing the program, module, function, and statement levels. The window displays only source code that was compiled with debug information. Thus, startup code, exit code, and library code is not displayed in the window. Furthermore, coverage information for statements in inlined functions is not displayed. Only the statement containing the inlined function call is marked as executed. The plus sign and minus sign icons allow you to expand and collapse the structure.

These icons give you an overview of the current status on all levels:

Red diamond	Signifies that 0% of the modules or functions has been executed.
Green diamond	Signifies that 100% of the modules or functions has been executed.
Red and green diamond	Signifies that some of the modules or functions have been executed.
Yellow diamond	Signifies a statement that has not been executed.

The percentage displayed at the end of every program, module, and function line shows the amount of statements that has been covered so far, that is, the number of executed statements divided with the total number of statements.

For statements that have not been executed (yellow diamond), the information displayed is the column number range and the row number of the statement in the source window, followed by the address of the step point:

<*column_start*>-<*column_end*>:*row address*.

A statement is considered to be executed when one of its instructions has been executed. When a statement has been executed, it is removed from the window and the percentage is increased correspondingly.

Double-clicking a statement or a function in the Code Coverage window displays that statement or function as the current position in the source window, which becomes the active window. Double-clicking a module on the program level expands or collapses the tree structure.

Context menu

This context menu is available:

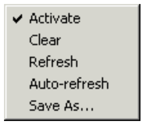


Figure 74: Code coverage window context menu

These commands are available:

	Activate	Switches code coverage on and off during execution.
	Clear	Clears the code coverage information. All step points are marked as not executed.
	Refresh	Updates the code coverage information and refreshes the window. All step points that have been executed since the last refresh are removed from the tree.
	Auto-refresh	Toggles the automatic reload of code coverage information on and off. When turned on, the code coverage information is reloaded automatically when C-SPY stops at a breakpoint, at a step point, and at program exit.
	Save As	Saves the current code coverage result in a text file.
	Save session	Saves your code coverage session data to a `*.dat` file. This is useful if you for some reason must abort your debug session, but want to continue the session later on. This command is available on the toolbar.
	Restore session	Restores previously saved code coverage session data. This is useful if you for some reason must abort your debug session, but want to continue the session later on. This command is available on the toolbar.

Simulating interrupts

By simulating interrupts, you can debug the program logic long before any hardware is available. This chapter contains detailed information about the C-SPY® interrupt simulation system and how to configure the simulated interrupts to make them reflect the interrupts of your target hardware. More specifically, this means:

- Introduction to interrupt simulation
- Procedures for simulating interrupts
- Reference information on simulating interrupts.

Introduction to interrupt simulation

This section introduces the C-SPY interrupt simulation system.

This section covers these topics:

- Reasons for using the interrupt simulation system
- Briefly about the interrupt simulation system
- Interrupt characteristics
- Interrupt simulation states
- C-SPY system macros for interrupts
- Target-adapting the interrupt simulation system.

See also:

- *Reference information on C-SPY system macros*, page 191
- *Using breakpoints*, page 89
- The *IAR C/C++ Compiler Reference Guide for 8051*.

REASONS FOR USING THE INTERRUPT SIMULATION SYSTEM

Simulated interrupts let you test the logic of your interrupt service routines and debug the interrupt handling in the target system. If you use simulated interrupts in conjunction with C-SPY macros and breakpoints, you can compose a complex simulation of, for instance, interrupt-driven peripheral devices.

BRIEFLY ABOUT THE INTERRUPT SIMULATION SYSTEM

The C-SPY Simulator includes an interrupt simulation system where you can simulate the execution of interrupts during debugging. You can configure the interrupt simulation system so that it resembles your hardware interrupt system.

The interrupt system has the following features:

- Simulated interrupt support for the 8051 microcontroller
- Single-occasion or periodical interrupts based on the cycle counter
- Predefined interrupts for various devices
- Configuration of hold time, probability, and timing variation
- State information for locating timing problems
- Configuration of interrupts using a dialog box or a C-SPY system macro—that is, one interactive and one automating interface. In addition, you can instantly force an interrupt.
- A log window which continuously displays events for each defined interrupt.

All interrupts you define using the **Interrupt Setup** dialog box are preserved between debug sessions, unless you remove them. A forced interrupt, on the other hand, exists only until it has been serviced and is not preserved between sessions.

The interrupt simulation system is activated by default, but if not required, you can turn off the interrupt simulation system to speed up the simulation. To turn it off, use either the **Interrupt Setup** dialog box or a system macro.

INTERRUPT CHARACTERISTICS

The simulated interrupts consist of a set of characteristics which lets you fine-tune each interrupt to make it resemble the real interrupt on your target hardware. You can specify a *first activation time*, a *repeat interval*, a *hold time*, a *variance*, and a *probability.*

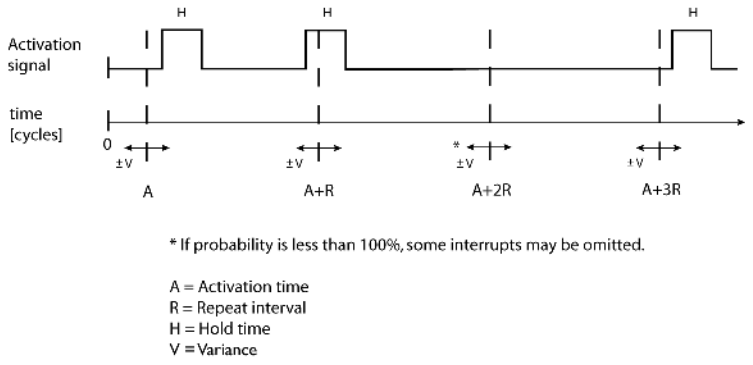


Figure 75: Simulated interrupt configuration

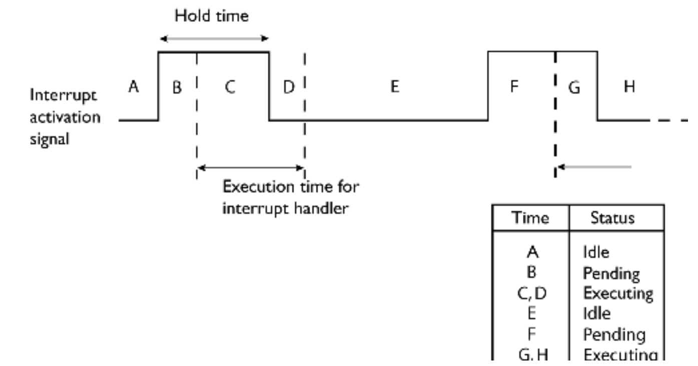
The interrupt simulation system uses the cycle counter as a clock to determine when an interrupt should be raised in the simulator. You specify the *first activation time*, which is based on the cycle counter. C-SPY will generate an interrupt when the cycle counter has passed the specified activation time. However, interrupts can only be raised between instructions, which means that a full assembler instruction must have been executed before the interrupt is generated, regardless of how many cycles an instruction takes.

To define the periodicity of the interrupt generation you can specify the *repeat interval* which defines the amount of cycles after which a new interrupt should be generated. In addition to the repeat interval, the periodicity depends on the two options *probability*—the probability, in percent, that the interrupt will actually appear in a period—and *variance*—a time variation range as a percentage of the repeat interval. These options make it possible to randomize the interrupt simulation. You can also specify a *hold time* which describes how long the interrupt remains pending until removed if it has not been processed. If the hold time is set to *infinite*, the corresponding pending bit will be set until the interrupt is acknowledged or removed.

INTERRUPT SIMULATION STATES

The interrupt simulation system contains status information that you can use for locating timing problems in your application. The **Interrupt Setup** dialog box displays the available status information. For an interrupt, these states can be displayed: *Idle*, *Pending*, *Executing*, *Executed*, *Removed*, *Rejected*, or *Expired*.

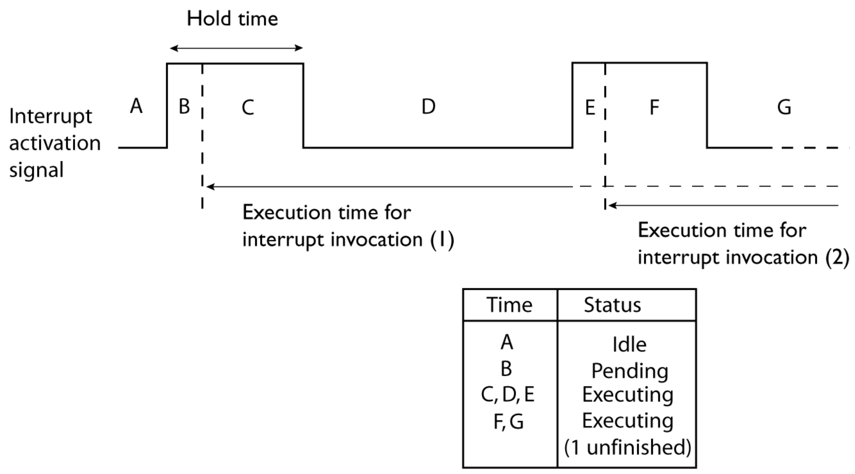
For a repeatable interrupt that has a specified repeat interval which is longer than the execution time, the status information at different times can look like this:


Hold time
Interrupt activation signal
A
B
C
D
E
F
G
H
Execution time for interrupt handler

Time	Status
A	Idle
B	Pending
C, D	Executing
E	Idle
F	Pending
G, H	Executing

Figure 76: Simulation states - example 1

If the interrupt repeat interval is shorter than the execution time, and the interrupt is reentrant (or non-maskable), the status information at different times can look like this:


Hold time
Interrupt activation signal
A
B
C
D
E
F
G
Execution time for interrupt invocation (1)
Execution time for interrupt invocation (2)

Time	Status
A	Idle
B	Pending
C, D, E	Executing
F, G	Executing (1 unfinished)

Figure 77: Simulation states - example 2

In this case, the execution time of the interrupt handler is too long compared to the repeat interval, which might indicate that you should rewrite your interrupt handler and make it faster, or that you should specify a longer repeat interval for the interrupt simulation system.

C-SPY SYSTEM MACROS FOR INTERRUPTS

Macros are useful when you already have sorted out the details of the simulated interrupt so that it fully meets your requirements. If you write a macro function containing definitions for the simulated interrupts, you can execute the functions automatically when C-SPY starts. Another advantage is that your simulated interrupt definitions will be documented if you use macro files, and if you are several engineers involved in the development project you can share the macro files within the group.

The C-SPY Simulator provides these predefined system macros related to interrupts:

`__enableInterrupts`

`__disableInterrupts`

`__orderInterrupt`

`__cancelInterrupt`

`__cancelAllInterrupts`

`__popSimulatorInterruptExecutingStack`

The parameters of the first five macros correspond to the equivalent entries of the **Interrupts** dialog box.

For more information about each macro, see *Reference information on C-SPY system macros*, page 191.

TARGET-ADAPTING THE INTERRUPT SIMULATION SYSTEM

The interrupt simulation system is easy to use. However, to take full advantage of the interrupt simulation system you should be familiar with how to adapt it for the processor you are using.

The behavior of the interrupt simulation resembles the hardware—the main difference is that the simulation does not have interrupt priority. This means that the execution of an interrupt is dependent on the status of the global interrupt enable bit. The execution of maskable interrupts is also dependent on the status of the individual interrupt enable bits.

To perform these actions for various devices, the interrupt system must have detailed information about each available interrupt. Except for default settings, this information is provided in the device description files. The default settings are used if no device description file has been specified.

For information about device description files, see *Selecting a device description file*, page 41.

Procedures for simulating interrupts

This section gives you step-by-step descriptions about how to use the interrupt simulation system.

More specifically, you will get information about:

- Simulating a simple interrupt
- Simulating an interrupt in a multi-task system.

See also:

- *Registering and executing using setup macros and setup files*, page 182 for details about how to use a setup file to define simulated interrupts at C-SPY startup
- The tutorial *Simulating an interrupt* in the Information Center.

SIMULATING A SIMPLE INTERRUPT

This example demonstrates the method for simulating a timer interrupt. However, the procedure can also be used for other types of interrupts.

To simulate and debug an interrupt:

1 Assume this simple application which contains an interrupt service routine for a timer, which increments a tick variable. The main function sets the necessary status registers. The application exits when 100 interrupts have been generated.

```
#include <io8051.h>
#include <stdio.h>
#include <intrinsics.h>

volatile int ticks = 0;
void main (void)
{
  /* Setup timer 0 */
  TCON_bit.TF0 = 0;
  TCON_bit.TR0 = 1; /* Start timer 0 */

  /* 16 bit timer mode */
  TMOD_bit.M00 = 1;
  TMOD_bit.M10 = 0;

  /* Set the time with 16 bits. To get a repeat interval of 2000
  cycles we loads TLH0:TL0 with 0xFFFF - 1 - 2000 = 0xF82E. */
  TL0 = 0x2E;
  TH0 = 0xF8;
  IE_bit.ET0 = 1;              /* Enable timer 0 interrupts */
  __enable_interrupt();       /* Enable interrupts */
  while (ticks < 100);        /* Endless loop */
  printf("Done\n");
}

/* Timer interrupt service routine */
#pragma vector = TF0_int
__interrupt void basic_timer(void)
{
  ticks += 1;
  TCON_bit.TF0 = 0;
}
```

2 Add your interrupt service routine to your application source code and add the file to your project.

3 Choose **Project>Options>Debugger>Setup** and select a device description file. The device description file contains information about the interrupt that C-SPY needs to be able to simulate it. Use the **Use device description file** browse button to locate the `io8051.ddf` file in the `config\devices\_generic` directory.

4 Build your project and start the simulator.

5 Choose **Simulator>Interrupt Setup** to open the **Interrupts Setup** dialog box. Select the **Enable interrupt simulation** option to enable interrupt simulation. Click **New** to open the **Edit Interrupt** dialog box. For the Timer example, verify these settings:

Option	Settings
Interrupt	TF0_int
First activation	4000
Repeat interval	2000
Hold time	10
Probability (%)	100
Variance (%)	0

Table 12: Timer interrupt settings

Click **OK**.

6 Execute your application. If you have enabled the interrupt properly in your application source code, C-SPY will:

- Generate an interrupt when the cycle counter has passed 4000
- Continuously repeat the interrupt after approximately 2000 cycles.

7 To watch the interrupt in action, choose **Simulator>Interrupt Log** to open the Interrupt Log window.

SIMULATING AN INTERRUPT IN A MULTI-TASK SYSTEM

If you are using interrupts in such a way that the normal instruction used for returning from an interrupt handler is not used, for example in an operating system with task-switching, the simulator cannot automatically detect that the interrupt has finished executing. The interrupt simulation system will work correctly, but the status information in the **Interrupt Setup** dialog box might not look as you expect. If too many interrupts are executing simultaneously, a warning might be issued.

To simulate a normal interrupt exit:

1 Set a code breakpoint on the instruction that returns from the interrupt function.

2 Specify the `__popSimulatorInterruptExecutingStack` macro as a condition to the breakpoint.

When the breakpoint is triggered, the macro is executed and then the application continues to execute automatically.

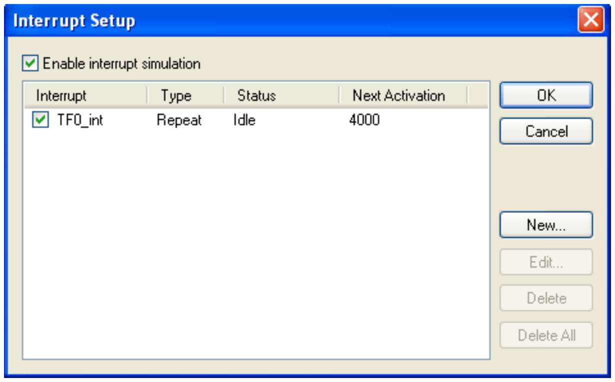
Reference information on simulating interrupts

This section gives reference information about these windows and dialog boxes:

- *Interrupt Setup dialog box*, page 169
- *Edit Interrupt dialog box*, page 171
- *Forced Interrupt window*, page 172
- *Interrupt Log window*, page 173.

Interrupt Setup dialog box

The **Interrupt Setup** dialog box is available by choosing **Simulator>Interrupt Setup**.



✓ Enable interrupt simulation

Interrupt	Type	Status	Next Activation
✓ TFO_int	Repeat	Idle	4000

Figure 78: Interrupt Setup dialog box

This dialog box lists all defined interrupts. Use this dialog box to enable or disable the interrupt simulation system, as well as to enable or disable individual interrupts.

Enable interrupt simulation

Enables or disables interrupt simulation. If the interrupt simulation is disabled, the definitions remain but no interrupts are generated. Note that you can also enable and disable installed interrupts individually by using the check box to the left of the interrupt name in the list of installed interrupts.

Display area

This area contains these columns:

Interrupt	Lists all interrupts. Use the checkbox to enable or disable the interrupt.

Type	Shows the type of the interrupt. The type can be one of: **Forced**, a single-occasion interrupt defined in the Forced Interrupt Window. **Single**, a single-occasion interrupt. **Repeat**, a periodically occurring interrupt. For repeatable interrupts there might be additional information about how many interrupts of the same type that are simultaneously executing (`*n* executing`). If *n* is larger than one, there is a reentrant interrupt in your interrupt simulation system that never finishes executing, which might indicate that there is a problem in your application.
Status	Shows the state of the interrupt: **Idle**, the interrupt activation signal is low (deactivated). **Pending**, the interrupt activation signal is active, but the interrupt has not been acknowledged yet by the interrupt handler. **Executing**, the interrupt is currently being serviced, that is the interrupt handler function is executing. **Executed**, this is a single-occasion interrupt and it has been serviced. **Removed**, the interrupt has been removed, but because the interrupt is currently executing it is visible until it is finished. **Rejected**, the interrupt has been rejected because the necessary interrupt registers are not set up to accept the interrupt. **Expired**, this is a single-occasion interrupt which was not serviced while the interrupt activation signal was active.
Next Activation	Shows the next activation time in cycles.

Buttons

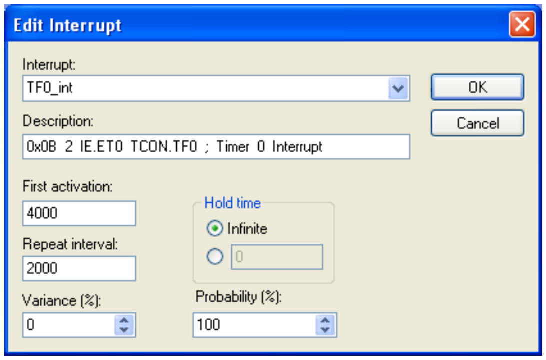
These buttons are available:

New	Opens the **Edit Interrupt** dialog box, see *Edit Interrupt dialog box*, page 171.
Edit	Opens the **Edit Interrupt** dialog box, see *Edit Interrupt dialog box*, page 171.
Delete	Removes the selected interrupt.
Delete All	Removes all interrupts.

Note: You can only edit or remove non-forced interrupts.

Edit Interrupt dialog box

The **Edit Interrupt** dialog box is available from the **Interrupt Setup** dialog box.


Edit Interrupt

Interrupt: TF0_int

Description: 0x0B 2 IE.ET0 TCON.TF0 ; Timer 0 Interrupt

First activation: 4000

Repeat interval: 2000

Variance (%): 0

Hold time: [✓ Infinite] / 0

Probability (%): 100

Figure 79: Edit Interrupt dialog box

Use this dialog box to interactively fine-tune the interrupt parameters. You can add the parameters and quickly test that the interrupt is generated according to your needs.

Interrupt

Selects the interrupt that you want to edit. The drop-down list contains all available interrupts. Your selection will automatically update the **Description** box. The list is populated with entries from the device description file that you have selected.

Description

A description of the selected interrupt, if available. The description is retrieved from the selected device description file and consists of a string describing the vector address, priority, enable bit, pending bit, and a description string, separated by space characters. For interrupts specified using the system macro `__orderInterrupt`, the **Description** box is empty.

First activation

Specify the value of the cycle counter after which the specified type of interrupt will be generated.

Repeat interval

Specify the periodicity of the interrupt in cycles.

Variance %

Selects a timing variation range, as a percentage of the repeat interval, in which the interrupt might occur for a period. For example, if the repeat interval is 100 and the variance 5%, the interrupt might occur anywhere between T=95 and T=105, to simulate a variation in the timing.

Hold time

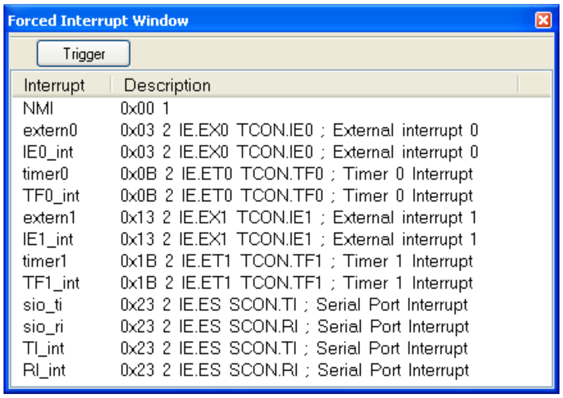
Specify how long, in cycles, the interrupt remains pending until removed if it has not been processed. If you select **Infinite**, the corresponding pending bit will be set until the interrupt is acknowledged or removed.

Probability %

Selects the probability, in percent, that the interrupt will actually occur within the specified period.

Forced Interrupt window

The **Forced Interrupt** window is available from the **Simulator** menu.



Interrupt	Description
NMI	0x00 1
extern0	0x03 2 IE.EX0 TCON.IE0 ; External interrupt 0
IE0_int	0x03 2 IE.EX0 TCON.IE0 ; External interrupt 0
timer0	0x0B 2 IE.ET0 TCON.TF0 ; Timer 0 Interrupt
TF0_int	0x0B 2 IE.ET0 TCON.TF0 ; Timer 0 Interrupt
extern1	0x13 2 IE.EX1 TCON.IE1 ; External interrupt 1
IE1_int	0x13 2 IE.EX1 TCON.IE1 ; External interrupt 1
timer1	0x1B 2 IE.ET1 TCON.TF1 ; Timer 1 Interrupt
TF1_int	0x1B 2 IE.ET1 TCON.TF1 ; Timer 1 Interrupt
sio_ti	0x23 2 IE.ES SCON.TI ; Serial Port Interrupt
sio_ri	0x23 2 IE.ES SCON.RI ; Serial Port Interrupt
TI_int	0x23 2 IE.ES SCON.TI ; Serial Port Interrupt
RI_int	0x23 2 IE.ES SCON.RI ; Serial Port Interrupt

Figure 80: Forced Interrupt window

Use this window to force an interrupt instantly. This is useful when you want to check your interrupt logistics and interrupt routines.

The hold time for a forced interrupt is infinite, and the interrupt exists until it has been serviced or until a reset of the debug session.

To force an interrupt:

1 Enable the interrupt simulation system, see *Interrupt Setup dialog box*, page 169.

2 Double-click the interrupt in the Forced Interrupt window, or select the interrupt and click **Trigger**.

Display area

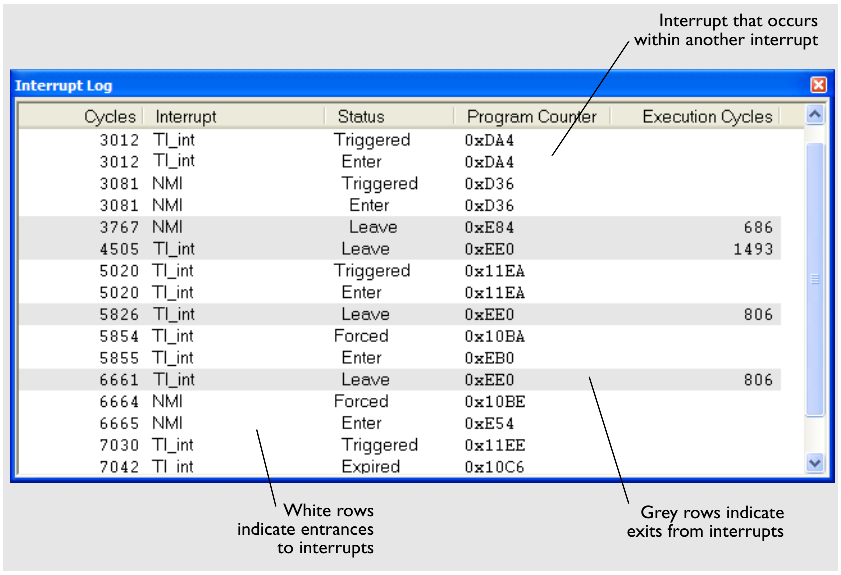
Lists all available interrupts and their definitions. The information is retrieved from the selected device description file. See this file for a detailed description.

Trigger

Triggers the interrupt you selected in the display area.

Interrupt Log window

The **Interrupt Log** window is available from the **Simulator** menu.


Interrupt that occurs within another interrupt
Interrupt Log
White rows indicate entrances to interrupts
Grey rows indicate exits from interrupts

Cycles	Interrupt	Status	Program Counter	Execution Cycles
3012	TI_int	Triggered	0xDA4	
3012	TI_int	Enter	0xDA4	
3081	NMI	Triggered	0xD36	
3081	NMI	Enter	0xD36	
3767	NMI	Leave	0xE84	686
4505	TI_int	Leave	0xEE0	1493
5020	TI_int	Triggered	0x11EA	
5020	TI_int	Enter	0x11EA	
5826	TI_int	Leave	0xEE0	806
5854	TI_int	Forced	0x10BA	
5855	TI_int	Enter	0xEB0	
6661	TI_int	Leave	0xEE0	806
6664	NMI	Forced	0x10BE	
6665	NMI	Enter	0xE54	
7030	TI_int	Triggered	0x11EE	
7042	TI_int	Expired	0x10C6	

Figure 81: Interrupt Log window

This window displays runtime information about the interrupts that you have activated in the **Edit Interrupts** dialog box or forced via the Forced Interrupt window. The information is useful for debugging the interrupt handling in the target system.

When the Interrupt Log window is open, it is updated continuously at runtime.

Note: There is a limit on the number of saved logs. When this limit is exceeded, the entries in the beginning of the buffer are erased.

For information about how to get a graphical view of the interrupt events during the execution of your application, see *Timeline window*, page 139.

Display area

This area contains these columns:

Time	The time for the interrupt entrance, based on an internally specified clock frequency. This column is available when you have selected **Show time** from the context menu.
Cycles	The number of cycles from the start of the execution until the event. This column is available when you have selected **Show cycles** from the context menu.
Interrupt	The interrupt as defined in the device description file.
Status	Shows the event status of the interrupt: **Triggered**, the interrupt has passed its activation time. **Forced**, the same as Triggered, but the interrupt was forced from the Forced Interrupt window. **Enter**, the interrupt is currently executing. **Leave**, the interrupt has been executed. **Expired**, the interrupt hold time has expired without the interrupt being executed. **Rejected**, the interrupt has been rejected because the necessary interrupt registers were not set up to accept the interrupt.
Program Counter	The value of the program counter when the event occurred.
Execution Time/Cycles	The time spent in the interrupt, calculated using the Enter and Leave timestamps. This includes time spent in any subroutines or other interrupts that occurred in the specific interrupt.

Interrupt Log window context menu

This context menu is available in the Interrupt Log window and the Interrupt Log Summary window:

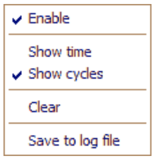


Figure 82: Interrupt Log window context menu

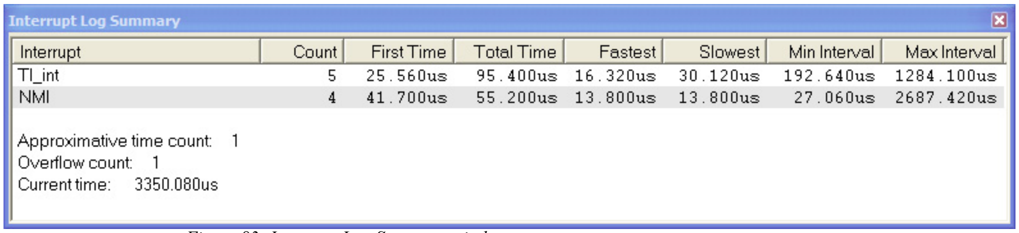
Note: The commands are the same in each window, but they only operate on the specific window.

These commands are available:

Enable	Enables the logging system. The system will log information also when the window is closed.
Show time	Displays the **Time** column in the Interrupt Log window.
Show cycles	Displays the **Cycles** column in the Interrupt Log window.
Clear	Deletes the log information. Note that this will happen also when you reset the debugger.
Save to log file	Displays a dialog box where you can select the destination file for the log information. The entries in the log file are separated by `TAB` and `LF`. An `X` in the Approx column indicates that the time stamp is an approximation.

Interrupt Log Summary window

The Interrupt Log Summary window is available from the **Simulator** menu.



Interrupt	Count	First Time	Total Time	Fastest	Slowest	Min Interval	Max Interval
TI_int	5	25.560us	95.400us	16.320us	30.120us	192.640us	1284.100us
NMI	4	41.700us	55.200us	13.800us	13.800us	27.060us	2687.420us

Approximative time count: 1
Overflow count: 1
Current time: 3350.080us

Figure 83: Interrupt Log Summary window

This window displays a summary of logs of entrances and exits to and from interrupts.

Display area

Each row in this area displays some statistics about the specific interrupt based on the log information in these columns:

Interrupt*	The type of interrupt that occurred.
Count	The number of times the interrupt occurred.
First time	The first time the interrupt was executed.
Total time**	The accumulated time spent in the interrupt.
Fastest**	The fastest execution of a single interrupt of this type.
Slowest**	The slowest execution of a single interrupt of this type.
Max interval†	The longest time between two interrupts of this type.

*** At the bottom of the column, the current time or cycles is displayed—the number of cycles or the execution time since the start of execution. Overflow count and approximative time count is always zero.**
**** Calculated in the same way as for the Execution time/cycles in the Interrupt Log window.**
† The interval is specified as the time interval between the entry time for two consecutive interrupts.

Context menu

See *Interrupt Log window context menu*, page 175.

Using C-SPY macros

C-SPY® includes a comprehensive macro language which allows you to automate the debugging process and to simulate peripheral devices.

This chapter describes the C-SPY macro language, its features, for what purpose these features can be used, and how to use them. More specifically, this means:

- Introduction to C-SPY macros
- Procedures for using C-SPY macros
- Reference information on the macro language
- Reference information on reserved setup macro function names
- Reference information on C-SPY system macros.

Introduction to C-SPY macros

This section covers these topics:

- Reasons for using C-SPY macros
- Briefly about using C-SPY macros
- Briefly about setup macro functions and files
- Briefly about the macro language.

REASONS FOR USING C-SPY MACROS

You can use C-SPY macros either by themselves or in conjunction with complex breakpoints and interrupt simulation to perform a wide variety of tasks. Some examples where macros can be useful:

- Automating the debug session, for instance with trace printouts, printing values of variables, and setting breakpoints.
- Hardware configuring, such as initializing hardware registers.
- Feeding your application with simulated data during runtime.

- Simulating peripheral devices, see the chapter *Simulating interrupts*. This only applies if you are using the simulator driver.
- Developing small debug utility functions, for instance reading I/O input from a file, see the file `setupsimple.mac` located in the directory `\8051\tutor\`.

BRIEFLY ABOUT USING C-SPY MACROS

To use C-SPY macros, you should:

- Write your macro variables and functions and collect them in in one or several *macro files*
- Register your macros
- Execute your macros.

For registering and executing macros, there are several methods to choose between. Which method you choose depends on which level of interaction or automation you want, and depending on at which stage you want to register or execute your macro.

BRIEFLY ABOUT SETUP MACRO FUNCTIONS AND FILES

There are some reserved *setup macro function names* that you can use for defining macro functions which will be called at specific times, such as:

- Once after communication with the target system has been established but before downloading the application software
- Once after your application software has been downloaded
- Each time the reset command is issued
- Once when the debug session ends.

To define a macro function to be called at a specific stage, you should define and register a macro function with one of the reserved names. For instance, if you want to clear a specific memory area before you load your application software, the macro setup function `execUserPreload` should be used. This function is also suitable if you want to initialize some CPU registers or memory-mapped peripheral units before you load your application software.

You should define these functions in a *setup macro file*, which you can load before C-SPY starts. Your macro functions will then be automatically registered each time you start C-SPY. This is convenient if you want to automate the initialization of C-SPY, or if you want to register multiple setup macros.

For more information about each setup macro function, see *Reference information on reserved setup macro function names*, page 190.

BRIEFLY ABOUT THE MACRO LANGUAGE

The syntax of the macro language is very similar to the C language. There are:

- *Macro statements*, which are similar to C statements.
- *Macro functions*, which you can define with or without parameters and return values.
- Predefined built-in *system macros*, similar to C library functions, which perform useful tasks such as opening and closing files, setting breakpoints, and defining simulated interrupts.
- *Macro variables*, which can be global or local, and can be used in C-SPY expressions.
- *Macro strings*, which you can manipulate using predefined system macros.

For more information about the macro language components, see *Reference information on the macro language*, page 185.

Example

Consider this example of a macro function which illustrates the various components of the macro language:

```
__var oldVal;
CheckLatest(val)
{
 if (oldval != val)
 {
  __message "Message: Changed from ", oldval, " to ", val, "\n";
  oldval = val;
 }
}
```

Note: Reserved macro words begin with double underscores to prevent name conflicts.

Procedures for using C-SPY macros

This section gives you step-by-step descriptions about how to register and execute C-SPY macros.

More specifically, you will get information about:

- Registering C-SPY macros—an overview
- Executing C-SPY macros—an overview
- Using the Macro Configuration dialog box
- Registering and executing using setup macros and setup files

- Executing macros using Quick Watch
- Executing a macro by connecting it to a breakpoint.

For more examples using C-SPY macros, see:

- The tutorial *Simulating an interrupt* in the Information Center
- *Initializing target hardware before C-SPY starts*, page 45.

REGISTERING C-SPY MACROS—AN OVERVIEW

C-SPY must know that you intend to use your defined macro functions, and thus you must *register* your macros. There are various ways to register macro functions:

- You can register macros interactively in the **Macro Configuration** dialog box, see *Using the Macro Configuration dialog box*, page 181.
- You can register macro functions during the C-SPY startup sequence, see *Registering and executing using setup macros and setup files*, page 182.
- You can register a file containing macro function definitions, using the system macro `__registerMacroFile`. This means that you can dynamically select which macro files to register, depending on the runtime conditions. Using the system macro also lets you register multiple files at the same moment. For information about the system macro, see *__registerMacroFile*, page 204.

Which method you choose depends on which level of interaction or automation you want, and depending on at which stage you want to register your macro.

EXECUTING C-SPY MACROS—AN OVERVIEW

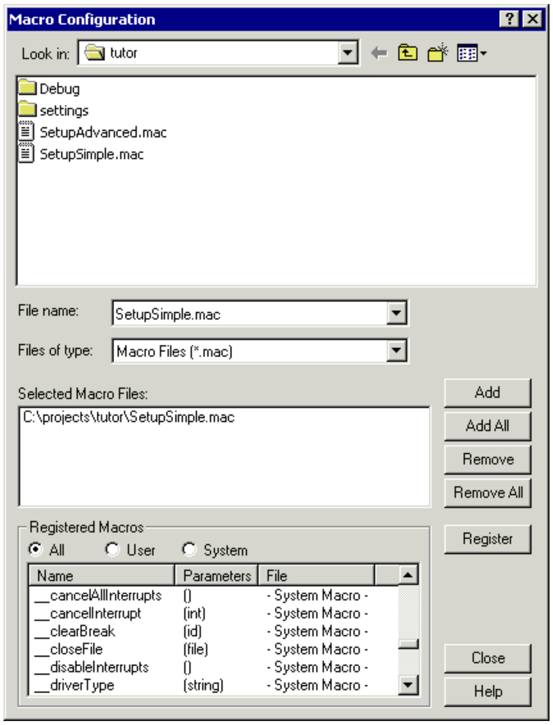
There are various ways to execute macro functions:

- You can execute macro functions during the C-SPY startup sequence and at other predefined stages during the debug session by defining setup macro functions in a setup macro file, see *Registering and executing using setup macros and setup files*, page 182.
- The Quick Watch window lets you evaluate expressions, and can thus be used for executing macro functions. For an example, see *Executing macros using Quick Watch*, page 183.
- A macro can be connected to a breakpoint; when the breakpoint is triggered the macro is executed. For an example, see *Executing a macro by connecting it to a breakpoint*, page 184.

Which method you choose depends on which level of interaction or automation you want, and depending on at which stage you want to execute your macro.

USING THE MACRO CONFIGURATION DIALOG BOX

The **Macro Configuration** dialog box is available by choosing **Debug>Macros**.



Look in: tutor

File name: SetupSimple.mac

Files of type: Macro Files (*.mac)

C:\projects\tutor\SetupSimple.mac

[✓ All] User System

Name	Parameters	File
__cancelAllInterrupts	()	- System Macro -
__cancelInterrupt	(int)	- System Macro -
__clearBreak	(id)	- System Macro -
__closeFile	(file)	- System Macro -
__disableInterrupts	()	- System Macro -
__driverType	(string)	- System Macro -

Figure 84: Macro Configuration dialog box

Use this dialog box to list, register, and edit your macro files and functions. The dialog box offers you an interactive interface for registering your macro functions which is convenient when you develop macro functions and continuously want to load and test them.

Macro functions that have been registered using the dialog box are deactivated when you exit the debug session, and will not automatically be registered at the next debug session.

To register a macro file:

- Select the macro files you want to register in the file selection list, and click **Add** or **Add All** to add them to the **Selected Macro Files** list. Conversely, you can remove files from the **Selected Macro Files** list using **Remove** or **Remove All**.

2 Click **Register** to register the macro functions, replacing any previously defined macro functions or variables. Registered macro functions are displayed in the scroll list under **Registered Macros**.

Note: System macros cannot be removed from the list, they are always registered.

To list macro functions:

1 Select **All** to display all macro functions, select **User** to display all user-defined macros, or select **System** to display all system macros.

2 Click either **Name** or **File** under **Registered Macros** to display the column contents sorted by macro names or by file. Clicking a second time sorts the contents in the reverse order.

To modify a macro file:

Double-click a user-defined macro function in the **Name** column to open the file where the function is defined, allowing you to modify it.

REGISTERING AND EXECUTING USING SETUP MACROS AND SETUP FILES

It can be convenient to register a macro file during the C-SPY startup sequence. To do this, specify a macro file which you load before starting the debugger. Your macro functions will be automatically registered each time you start the debugger.

If you use the reserved setup macro function names to define the macro functions, you can define exactly at which stage you want the macro function to be executed.

To define a setup macro function and load it during C-SPY startup:

1 Create a new text file where you can define your macro function.

For example:

```
execUserSetup()
{
 ...
 __registerMacroFile("MyMacroUtils.mac");
 __registerMacroFile("MyDeviceSimulation.mac");

}
```

This macro function registers the additional macro files `MyMacroUtils.mac` and `MyDeviceSimulation.mac`. Because the macro function is defined with the function name `execUserSetup`, it will be executed directly after your application has been downloaded.

2 Save the file using the filename extension `mac`.

3 Before you start C-SPY, choose **Project>Options>Debugger>Setup**. Select **Use Setup file** and choose the macro file you just created.

The macros will now be registered during the C-SPY startup sequence.

EXECUTING MACROS USING QUICK WATCH

The Quick Watch window lets you dynamically choose when to execute a macro function.

1 Consider this simple macro function that checks the status of a watchdog timer interrupt enable bit:

```
WDTstatus()
{
  if ((WDT & 0x01) != 0)        /* Checks the status of WDT */
    return "Timer enabled";     /* C-SPY macro string used */
  else
    return "Timer disabled";    /* C-SPY macro string used */
}
```

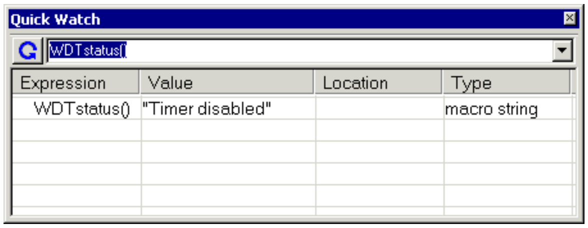
2 Save the macro function using the filename extension `mac`.

3 To register the macro file, choose **Debug>Macros**. The **Macro Configuration** dialog box appears.

4 Locate the file, click **Add** and then **Register**. The macro function appears in the list of registered macros.

5 Choose **View>Quick Watch** to open the Quick Watch window, type the macro call `WDTstatus()` in the text field and press Return,

Alternatively, in the macro file editor window, select the macro function name `WDTstatus()`. Right-click, and choose **Quick Watch** from the context menu that appears.



Expression	Value	Location	Type
WDTstatus()	"Timer disabled"		macro string

Figure 85: Quick Watch window

The macro will automatically be displayed in the Quick Watch window.

For more information, see *Quick Watch window*, page 85.

EXECUTING A MACRO BY CONNECTING IT TO A BREAKPOINT

You can connect a macro to a breakpoint. The macro will then be executed when the breakpoint is triggered. The advantage is that you can stop the execution at locations of particular interest and perform specific actions there.

For instance, you can easily produce log reports containing information such as how the values of variables, symbols, or registers change. To do this you might set a breakpoint on a suspicious location and connect a log macro to the breakpoint. After the execution you can study how the values of the registers have changed.

To create a log macro and connect it to a breakpoint:

1 Assume this skeleton of a C function in your application source code:

```
int fact(int x)
{
 ...
}
```

2 Create a simple log macro function like this example:

```
logfact()
{
 __message "fact(" ,x, ")";
}
```

The `__message` statement will log messages to the Log window.

Save the macro function in a macro file, with the filename extension `mac`.

3 To register the macro, choose **Debug>Macros** to open the **Macro Configuration** dialog box and add your macro file to the list **Selected Macro Files**. Click **Register** and your macro function will appear in the list **Registered Macros**. Close the dialog box.

4 To set a code breakpoint, click the **Toggle Breakpoint** button on the first statement within the function `fact` in your application source code. Choose **View>Breakpoints** to open the Breakpoints window. Select your breakpoint in the list of breakpoints and choose the **Edit** command from the context menu.

5 To connect the log macro function to the breakpoint, type the name of the macro function, `logfact()`, in the **Action** field and click **Apply**. Close the dialog box.

6 Execute your application source code. When the breakpoint is triggered, the macro function will be executed. You can see the result in the Log window.

Note that the expression in the **Action** field is evaluated only when the breakpoint causes the execution to really stop. If you want to log a value and then automatically continue execution, you can either:

- Use a Log breakpoint, see *Log breakpoints dialog box*, page 105
- Use the **Condition** field instead of the **Action** field. For an example, see *Performing a task and continuing execution*, page 99.

7 You can easily enhance the log macro function by, for instance, using the `__fmessage` statement instead, which will print the log information to a file. For information about the `__fmessage` statement, see *Formatted output*, page 188.

For an example where a serial port input buffer is simulated using the method of connecting a macro to a breakpoint, see the tutorial *Simulating an interrupt* in the Information Center.

Reference information on the macro language

This section gives reference information on the macro language:

- *Macro functions*, page 185
- *Macro variables*, page 186
- *Macro strings*, page 186
- *Macro statements*, page 187
- *Formatted output*, page 188.

MACRO FUNCTIONS

C-SPY macro functions consist of C-SPY variable definitions and macro statements which are executed when the macro is called. An unlimited number of parameters can be passed to a macro function, and macro functions can return a value on exit.

A C-SPY macro has this form:

```
macroName (parameterList)
{
  macroBody
}
```

where *`parameterList`* is a list of macro parameters separated by commas, and *`macroBody`* is any series of C-SPY variable definitions and C-SPY statements.

Type checking is neither performed on the values passed to the macro functions nor on the return value.

MACRO VARIABLES

A macro variable is a variable defined and allocated outside your application. It can then be used in a C-SPY expression, or you can assign application data—values of the variables in your application—to it. For more information about C-SPY expressions, see *C-SPY expressions*, page 74.

The syntax for defining one or more macro variables is:

```
__var nameList;
```

where `nameList` is a list of C-SPY variable names separated by commas.

A macro variable defined outside a macro body has global scope, and it exists throughout the whole debugging session. A macro variable defined within a macro body is created when its definition is executed and destroyed on return from the macro.

By default, macro variables are treated as signed integers and initialized to 0. When a C-SPY variable is assigned a value in an expression, it also acquires the type of that expression. For example:

Expression	What it means
`myvar = 3.5;`	`myvar` is now type `float`, value `3.5`.
`myvar = (int*)i;`	`myvar` is now type pointer to `int`, and the value is the same as `i`.

Table 13: Examples of C-SPY macro variables

In case of a name conflict between a C symbol and a C-SPY macro variable, C-SPY macro variables have a higher precedence than C variables. Note that macro variables are allocated on the debugger host and do not affect your application.

MACRO STRINGS

In addition to C types, macro variables can hold values of *macro strings*. Note that macro strings differ from C language strings.

When you write a string literal, such as `"Hello!"`, in a C-SPY expression, the value is a macro string. It is not a C-style character pointer `char*`, because `char*` must point to a sequence of characters in target memory and C-SPY cannot expect any string literal to actually exist in target memory.

You can manipulate a macro string using a few built-in macro functions, for example `__strFind` or `__subString`. The result can be a new macro string. You can concatenate macro strings using the `+` operator, for example `str + "tail"`. You can also access individual characters using subscription, for example `str[3]`. You can get the length of a string using `sizeof(str)`. Note that a macro string is not NULL-terminated.

The macro function `__toString` is used for converting from a NULL-terminated C string in your application (`char*` or `char[]`) to a macro string. For example, assume this definition of a C string in your application:

```
char const *cstr = "Hello";
```

Then examine these macro examples:

```
__var str;          /* A macro variable */
str = cstr          /* str is now just a pointer to char */
sizeof str          /* same as sizeof (char*), typically 2 or 4 */
str = __toString(cstr,512)  /* str is now a macro string */
sizeof str          /* 5, the length of the string */
str[1]              /* 101, the ASCII code for 'e' */
str += " World!"    /* str is now "Hello World!" */
```

See also *Formatted output*, page 188.

MACRO STATEMENTS

Statements are expected to behave in the same way as the corresponding C statements would do. The following C-SPY macro statements are accepted:

Expressions

```
expression;
```

For more information about C-SPY expressions, see *C-SPY expressions*, page 74.

Conditional statements

```
if (expression)
  statement

if (expression)
  statement
else
  statement
```

Loop statements

```
for (init_expression; cond_expression; update_expression)
  statement

while (expression)
  statement
```

```
do
  statement
while (expression);
```

Return statements

```
return;

return expression;
```

If the return value is not explicitly set, `signed int 0` is returned by default.

Blocks

Statements can be grouped in blocks.

```
{
  statement1
  statement2
  .
  .
  .
  statementN
}
```

FORMATTED OUTPUT

C-SPY provides various methods for producing formatted output:

`__message argList;`	Prints the output to the Debug Log window.
`__fmessage file, argList;`	Prints the output to the designated file.
`__smessage argList;`	Returns a string containing the formatted output.

where *`argList`* is a comma-separated list of C-SPY expressions or strings, and *`file`* is the result of the `__openFile` system macro, see *__openFile*, page 199.

To produce messages in the Debug Log window:

```
var1 = 42;
var2 = 37;
__message "This line prints the values ", var1, " and ", var2,
" in the Log window.";
```

This produces this message in the Log window:

```
This line prints the values 42 and 37 in the Log window.
```

To write the output to a designated file:

```
__fmessage myfile, "Result is ", res, "!\n";
```

To produce strings:

```
myMacroVar = __smessage 42, " is the answer.";
```

`myMacroVar` now contains the string `"42 is the answer."`.

Specifying display format of arguments

To override the default display format of a scalar argument (number or pointer) in *`argList`*, suffix it with a `:` followed by a format specifier. Available specifiers are:

`%b`	for binary scalar arguments
`%o`	for octal scalar arguments
`%d`	for decimal scalar arguments
`%x`	for hexadecimal scalar arguments
`%c`	for character scalar arguments

These match the formats available in the Watch and Locals windows, but number prefixes and quotes around strings and characters are not printed. Another example:

```
__message "The character '", cvar:%c, "' has the decimal value
", cvar;
```

Depending on the value of the variables, this produces this message:

```
The character 'A' has the decimal value 65
```

Note: A character enclosed in single quotes (a character literal) is an integer constant and is not automatically formatted as a character. For example:

```
__message 'A', " is the numeric value of the character ",
'A':%c;
```

would produce:

```
65 is the numeric value of the character A
```

Note: The default format for certain types is primarily designed to be useful in the Watch window and other related windows. For example, a value of type `char` is formatted as 'A' (0x41), while a pointer to a character (potentially a C string) is formatted as 0x8102 "Hello", where the string part shows the beginning of the string (currently up to 60 characters).

When printing a value of type `char*`, use the `%x` format specifier to print just the pointer value in hexadecimal notation, or use the system macro `__toString` to get the full string value.

Reference information on reserved setup macro function names

There are reserved setup macro function names that you can use for defining your setup macro functions. By using these reserved names, your function will be executed at defined stages during execution. For more information, see *Briefly about setup macro functions and files, page 178*.

This table summarizes the reserved setup macro function names:

Macro	Description
`execUserPreload`	Called after communication with the target system is established but before downloading the target application. Implement this macro to initialize memory locations and/or registers which are vital for loading data properly.
`execUserSetup`	Called once after the target application is downloaded. Implement this macro to set up the memory map, breakpoints, interrupts, register macro files, etc.
`execUserPreReset`	Called each time just before the reset command is issued. Implement this macro to set up any required device state.
`execUserReset`	Called each time just after the reset command is issued. Implement this macro to set up and restore data.
`execUserExit`	Called once when the debug session ends. Implement this macro to save status data etc.

Table 14: C-SPY setup macros

If you define interrupts or breakpoints in a macro file that is executed at system start (using `execUserSetup`) we strongly recommend that you also make sure that they are removed at system shutdown (using `execUserExit`). An example is available in `SetupSimple.mac`, see *Simulating an interrupt* in the Information Center.

The reason for this is that the simulator saves interrupt settings between sessions and if they are not removed they will get duplicated every time `execUserSetup` is executed again. This seriously affects the execution speed.

Reference information on C-SPY system macros

This section gives reference information about each of the C-SPY system macros.

This table summarizes the pre-defined system macros:

Macro	Description
`__cancelAllInterrupts`	Cancels all ordered interrupts
`__cancelInterrupt`	Cancels an interrupt
`__clearBreak`	Clears a breakpoint
`__closeFile`	Closes a file that was opened by `__openFile`
`__delay`	Delays execution
`__disableInterrupts`	Disables generation of interrupts
`__driverType`	Verifies the driver type
`__enableInterrupts`	Enables generation of interrupts
`__evaluate`	Interprets the input string as an expression and evaluates it.
`__isBatchMode`	Checks if C-SPY is running in batch mode or not.
`__loadImage`	Loads an image.
`__memoryRestore`	Restores the contents of a file to a specified memory zone
`__memorySave`	Saves the contents of a specified memory area to a file
`__openFile`	Opens a file for I/O operations
`__orderInterrupt`	Generates an interrupt
`__popSimulatorInterruptExecutingStack`	Informs the interrupt simulation system that an interrupt handler has finished executing
`__readFile`	Reads from the specified file
`__readFileByte`	Reads one byte from the specified file
`__readMemory8`, `__readMemoryByte`	Reads one byte from the specified memory location
`__readMemory16`	Reads two bytes from the specified memory location
`__readMemory32`	Reads four bytes from the specified memory location
`__registerMacroFile`	Registers macros from the specified file
`__resetFile`	Rewinds a file opened by `__openFile`

Table 15: Summary of system macros

Macro	Description
`__setCodeBreak`	Sets a code breakpoint
`__setDataBreak`	Sets a data breakpoint
`__setLogBreak`	Sets a log breakpoint
`__setSimBreak`	Sets a simulation breakpoint
`__setTraceStartBreak`	Sets a trace start breakpoint
`__setTraceStopBreak`	Sets a trace stop breakpoint
`__sourcePosition`	Returns the file name and source location if the current execution location corresponds to a source location
`__strFind`	Searches a given string for the occurrence of another string
`__subString`	Extracts a substring from another string
`__targetDebuggerVersion`	Returns the version of the target debugger
`__toLower`	Returns a copy of the parameter string where all the characters have been converted to lower case
`__toString`	Prints strings
`__toUpper`	Returns a copy of the parameter string where all the characters have been converted to upper case
`__unloadImage`	Unloads a debug image.
`__writeFile`	Writes to the specified file
`__writeFileByte`	Writes one byte to the specified file
`__writeMemory8`, `__writeMemoryByte`	Writes one byte to the specified memory location
`__writeMemory16`	Writes a two-byte word to the specified memory location
`__writeMemory32`	Writes a four-byte word to the specified memory location

Table 15: Summary of system macros (Continued)

__cancelAllInterrupts

Syntax	`__cancelAllInterrupts()`
Return value	`int 0`
Description	Cancels all ordered interrupts.
Applicability	This system macro is only available in the C-SPY Simulator.

__cancelInterrupt

Syntax `__cancelInterrupt(`*`interrupt_id`*`)`

Parameter

`interrupt_id`	The value returned by the corresponding `__orderInterrupt` macro call (unsigned long)

Return value

Result	Value
Successful	`int 0`
Unsuccessful	Non-zero error number

Table 16: __cancelInterrupt return values

Description	Cancels the specified interrupt.
Applicability	This system macro is only available in the C-SPY Simulator.

__clearBreak

Syntax `__clearBreak(`*`break_id`*`)`

Parameter

`break_id`	The value returned by any of the set breakpoint macros

Return value	`int 0`
Description	Clears a user-defined breakpoint.
See also	*Using breakpoints*, page 89.

__closeFile

Syntax `__closeFile(`*`fileHandle`*`)`

Parameter

`fileHandle`	The macro variable used as filehandle by the `__openFile` macro

Return value `int 0`

Description Closes a file previously opened by `__openFile`.

__delay

Syntax `__delay(`*`value`*`)`

Parameter

`value`	The number of milliseconds to delay execution

Return value `int 0`

Description Delays execution the specified number of milliseconds.

__disableInterrupts

Syntax `__disableInterrupts()`

Return value

Result	Value
Successful	`int 0`
Unsuccessful	Non-zero error number

Table 17: __disableInterrupts return values

Description Disables the generation of interrupts.

Applicability This system macro is only available in the C-SPY Simulator.

__driverType

Syntax `__driverType(*driver_id*)`

Parameter

driver_id	A string corresponding to the driver you want to check for. Choose one of these: `"sim"` corresponds to the simulator driver. `"emu_cc"` corresponds to the Texas Instruments driver. `"emu_fs2"` corresponds to the FS2 System Navigator driver. `"emu_if"` corresponds to the Infineon driver. `"emu_ns"` corresponds to the Nordic Semiconductor driver. `"rom"` corresponds to the ROM-monitor driver. `"rom_ad2"` corresponds to the Analog Devices driver. `"rom_sl"` corresponds to the Silicon Laboratories driver.

Return value

Result	Value
Successful	`1`
Unsuccessful	`0`

Table 18: __driverType return values

Description Checks to see if the current C-SPY driver is identical to the driver type of the *driver_id* parameter.

Example

```
__driverType("sim")
```

If the simulator is the current driver, the value `1` is returned. Otherwise `0` is returned.

__enableInterrupts

Syntax `__enableInterrupts()`

Return value

Result	Value
Successful	`int 0`
Unsuccessful	Non-zero error number

Table 19: __enableInterrupts return values

Description Enables the generation of interrupts.

Applicability

This system macro is only available in the C-SPY Simulator.

__evaluate

Syntax

`__evaluate(`*`string, valuePtr`*`)`

Parameter

`string`	Expression string
`valuePtr`	Pointer to a macro variable storing the result

Return value

Result	Value
Successful	`int 0`
Unsuccessful	`int 1`

Table 20: __evaluate return values

Description

This macro interprets the input string as an expression and evaluates it. The result is stored in a variable pointed to by *`valuePtr`*.

Example

This example assumes that the variable `i` is defined and has the value 5:

```
__evaluate("i + 3", &myVar)
```

The macro variable `myVar` is assigned the value 8.

__isBatchMode

Syntax

```
__isBatchMode()
```

Return value

Result	Value
True	`int 1`
False	`int 0`

Table 21: __isBatchMode return values

Description

This macro returns True if the debugger is running in batch mode, otherwise it returns False.

__loadImage

Syntax

```
__loadImage(path, offset, debugInfoOnly)
```

Parameter

`path`	A string that identifies the path to the image to download. The path must either be absolute or use argument variables. For information about argument variables, see the *IDE Project Management and Building Guide*.
`offset`	An integer that identifies the offset to the destination address for the downloaded image.
`debugInfoOnly`	A non-zero integer value if no code or data should be downloaded to the target system, which means that C-SPY will only read the debug information from the debug file. Or, 0 (zero) for download.

Return value

Value	Result
Non-zero integer number	A unique module identification.
`int 0`	Loading failed.

Table 22: __loadImage return values

Description

Loads an image (debug file).

Example 1

Your system consists of a ROM library and an application. The application is your active project, but you have a debug file corresponding to the library. In this case you can add this macro call in the `execUserSetup` macro in a C-SPY macro file, which you associate with your project:

```
__loadImage(ROMfile, 0x8000, 1);
```

This macro call loads the debug information for the ROM library *`ROMfile`* without downloading its contents (because it is presumably already in ROM). Then you can debug your application together with the library.

Example 2

Your system consists of a ROM library and an application, but your main concern is the library. The library needs to be programmed into flash memory before a debug session. While you are developing the library, the library project must be the active project in the IDE. In this case you can add this macro call in the `execUserSetup` macro in a C-SPY macro file, which you associate with your project:

```
__loadImage(ApplicationFile, 0x8000, 0);
```

The macro call loads the debug information for the application and downloads its contents (presumably into RAM). Then you can debug your library together with the application.

See also

Images, page 265 and *Loading multiple images*, page 43.

__memoryRestore

Syntax

`__memoryRestore(zone, filename)`

Parameters

zone	The memory zone name (string); for a list of available zones, see *C-SPY memory zones*, page 114.
filename	A string that specifies the file to be read. The filename must include a path, which must either be absolute or use argument variables. For information about argument variables, see the *IDE Project Management and Building Guide*.

Return value

`int 0`

Description

Reads the contents of a file and saves it to the specified memory zone.

Example

```
__memoryRestore("Code", "c:\\temp\\saved_memory.hex");
```

See also

Memory Restore dialog box, page 121.

__memorySave

Syntax

`__memorySave(start, stop, format, filename)`

Parameters

start	A string that specifies the first location of the memory area to be saved
stop	A string that specifies the last location of the memory area to be saved

format	A string that specifies the format to be used for the saved memory. Choose between: `intel-extended` `motorola` `motorola-s19` `motorola-s28` `motorola-s37`.
filename	A string that specifies the file to write to. The filename must include a path, which must either be absolute or use argument variables. For information about argument variables, see the *IDE Project Management and Building Guide*.

Return value

`int 0`

Description

Saves the contents of a specified memory area to a file.

Example

```
__memorySave("XData:0x00", "XData:0xFF", "intel-extended",
"c:\\temp\\saved_memory.hex");
```

See also

Memory Save dialog box, page 120.

__openFile

Syntax

`__openFile(`*`filename, access`*`)`

Parameters

filename	The file to be opened. The filename must include a path, which must either be absolute or use argument variables. For information about argument variables, see the *IDE Project Management and Building Guide*.

`access` The access type (string).

These are mandatory but mutually exclusive:

`"a"`	append, new data will be appended at the end of the open file
`"r"`	read
`"w"`	write

These are optional and mutually exclusive:

`"b"`	binary, opens the file in binary mode
`"t"`	ASCII text, opens the file in text mode

This access type is optional:

`"+"` together with `r`, `w`, or `a`; `r+` or `w+` is *read* and *write*, while `a+` is *read* and *append*

Return value

Result	Value
Successful	The file handle
Unsuccessful	An invalid file handle, which tests as False

Table 23: __openFile return values

Description

Opens a file for I/O operations. The default base directory of this macro is where the currently open project file (`*.ewp`) is located. The argument to `__openFile` can specify a location relative to this directory. In addition, you can use argument variables such as `$PROJ_DIR$` and `$TOOLKIT_DIR$` in the path argument.

Example

```
__var myFileHandle;         /* The macro variable to contain */
                            /* the file handle */
myFileHandle = __openFile("$PROJ_DIR$\\Debug\\Exe\\test.tst",
"r");
if (myFileHandle)
{
    /* successful opening */
}
```

See also

For information about argument variables, see the *IDE Project Management and Building Guide*.

__orderInterrupt

Syntax

```
__orderInterrupt(specification, first_activation,
                 repeat_interval, variance, infinite_hold_time,
                 hold_time, probability)
```

Parameters

specification	The interrupt (string). The specification can either be the full specification used in the device description file (`ddf`) or only the name. In the latter case the interrupt system will automatically get the description from the device description file.
first_activation	The first activation time in cycles (integer)
repeat_interval	The periodicity in cycles (integer)
variance	The timing variation range in percent (integer between 0 and 100)
infinite_hold_time	1 if infinite, otherwise 0.
hold_time	The hold time (integer)
probability	The probability in percent (integer between 0 and 100)

Return value

The macro returns an interrupt identifier (`unsigned long`).

If the syntax of *specification* is incorrect, it returns -1.

Description

Generates an interrupt.

Applicability

This system macro is only available in the C-SPY Simulator.

Example

This example generates a repeating interrupt using an infinite hold time first activated after 4000 cycles:

```
__orderInterrupt( "TF0_int", 4000, 2000, 0, 1, 0, 100 );
```

__popSimulatorInterruptExecutingStack

Syntax

```
__popSimulatorInterruptExecutingStack(void)
```

Return value

This macro has no return value.

Description

Informs the interrupt simulation system that an interrupt handler has finished executing, as if the normal instruction used for returning from an interrupt handler was executed.

This is useful if you are using interrupts in such a way that the normal instruction for returning from an interrupt handler is not used, for example in an operating system with task-switching. In this case, the interrupt simulation system cannot automatically detect that the interrupt has finished executing.

Applicability

This system macro is only available in the C-SPY Simulator.

See also

Simulating an interrupt in a multi-task system, page 168.

__readFile

Syntax

`__readFile(`*`fileHandle`*`, `*`valuePtr`*`)`

Parameters

`fileHandle`	A macro variable used as filehandle by the `__openFile` macro
`valuePtr`	A pointer to a variable

Return value

Result	Value
Successful	0
Unsuccessful	Non-zero error number

Table 24: __readFile return values

Description

Reads a sequence of hexadecimal digits from the given file and converts them to an `unsigned long` which is assigned to the *`value`* parameter, which should be a pointer to a macro variable.

Example

```
__var number;
if (__readFile(myFileHandle, &number) == 0)
{
  // Do something with number
}
```

__readFileByte

Syntax `__readFileByte(`*`fileHandle`*`)`

Parameter

`fileHandle`	A macro variable used as filehandle by the `__openFile` macro

Return value `-1` upon error or end-of-file, otherwise a value between `0` and `255`.

Description Reads one byte from a file.

Example

```
__var byte;
while ( (byte = __readFileByte(myFileHandle)) != -1 )
{
  /* Do something with byte */
}
```

__readMemory8, __readMemoryByte

Syntax

```
__readMemory8(address, zone)
__readMemoryByte(address, zone)
```

Parameters

`address`	The memory address (integer)
`zone`	The memory zone name (string); for more information about available zones, see *C-SPY memory zones*, page 114.

Return value The macro returns the value from memory.

Description Reads one byte from a given memory location.

Example

```
__readMemory8(0x0108, "XData");
```

__readMemory16

Syntax `__readMemory16(`*`address`*`, `*`zone`*`)`

Parameters

`address`	The memory address (integer)

zone	The memory zone name (string); for more information about available zones, see *C-SPY memory zones*, page 114.

Return value

The macro returns the value from memory.

Description

Reads a two-byte word from a given memory location.

Example

```
__readMemory16(0x0108, "XData");
```

__readMemory32

Syntax

`__readMemory32(`*`address`*`, `*`zone`*`)`

Parameters

address	The memory address (integer)
zone	The memory zone name (string); for more information about available zones, see *C-SPY memory zones*, page 114.

Return value

The macro returns the value from memory.

Description

Reads a four-byte word from a given memory location.

Example

```
__readMemory32(0x0108, "XData");
```

__registerMacroFile

Syntax

`__registerMacroFile(`*`filename`*`)`

Parameter

filename	A file containing the macros to be registered (string). The filename must include a path, which must either be absolute or use argument variables. For information about argument variables, see the *IDE Project Management and Building Guide*.

Return value

`int 0`

Description

Registers macros from a setup macro file. With this function you can register multiple macro files during C-SPY startup.

Example	`__registerMacroFile("c:\\testdir\\macro.mac");`
See also	*Registering and executing using setup macros and setup files*, page 182.

__resetFile

Syntax `__resetFile(`*`fileHandle`*`)`

Parameter

`fileHandle`	A macro variable used as filehandle by the `__openFile` macro

Return value `int 0`

Description Rewinds a file previously opened by `__openFile`.

__setCodeBreak

Syntax `__setCodeBreak(`*`location, count, condition, cond_type, action`*`)`

Parameters

`location`	A string with a location description. Choose between: A C-SPY expression, whose value evaluates to a valid address, such as a function, for example `main`. For more information about C-SPY expressions, see *C-SPY expressions*, page 74. An absolute address, on the form *`zone`*`:`*`hexaddress`* or simply *`hexaddress`* (for example `Code:42`). *`zone`* refers to C-SPY memory zones and specifies in which memory the address belongs. A source location in your C source code, using the syntax `{`*`filename`*`}.`*`row`*`.`*`col`*. For example`{D:\\src\\prog.c}.22.3` sets a breakpoint on the third character position on row 22 in the source file `prog.c`. Note that the Source location type is usually meaningful only for code breakpoints.
`count`	The number of times that a breakpoint condition must be fulfilled before a break occurs (integer)
`condition`	The breakpoint condition (string)

cond_type	The condition type; either “`CHANGED`” or “`TRUE`” (string)
action	An expression, typically a call to a macro, which is evaluated when the breakpoint is detected

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	0

Table 25: __setCodeBreak return values

Description

Sets a code breakpoint, that is, a breakpoint which is triggered just before the processor fetches an instruction at the specified location.

Examples

```
__setCodeBreak("{D:\\src\\prog.c}.12.9", 3, "d>16", "TRUE",
"ActionCode()");
```

This example sets a code breakpoint on the label `main` in your source:

```
__setCodeBreak("main", 0, "1", "TRUE", "");
```

See also

Using breakpoints, page 89.

__setDataBreak

Syntax

```
__setDataBreak(location, count, condition, cond_type, access,
               action)
```

Parameters

location	A string with a location description. Choose between: A C-SPY expression, whose value evaluates to a valid address, such as a variable name. For example, `my_var` refers to the location of the variable `my_var`, and `arr[3]` refers to the location of the third element of the array `arr`. For static variables declared with the same name in several functions, use the syntax `my_func::my_static_variable` to refer to a specific variable. For more information about C-SPY expressions, see *C-SPY expressions*, page 74. An absolute address, on the form *`zone`*`:`*`hexaddress`* or simply *`hexaddress`* (for example `XData:42`). *`zone`* refers to C-SPY memory zones and specifies in which memory the address belongs. A source location in your C source code, using the syntax `{`*`filename`*`}.`*`row`*`.`*`col`*. For example`{D:\\src\\prog.c}.22.3` sets a breakpoint on the third character position on row 22 in the source file `prog.c`. Note that the Source location type is usually meaningful only for code breakpoints.
count	The number of times that a breakpoint condition must be fulfilled before a break occurs (integer)
condition	The breakpoint condition (string)
cond_type	The condition type; either `"CHANGED"` or `"TRUE"` (string)
access	The memory access type: `"R"` for read, `"W"` for write, or `"RW"` for read/write
`action`	An expression, typically a call to a macro, which is evaluated when the breakpoint is detected

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	0

Table 26: __setDataBreak return values

Description

Sets a data breakpoint, that is, a breakpoint which is triggered directly after the processor has read or written data at the specified location.

Applicability

This system macro is only available in the C-SPY Simulator.

Example

```
__var brk;
brk = __setDataBreak("XData:0x4710", 3, "d>6", "TRUE",
      "W", "ActionData()");
...
__clearBreak(brk);
```

See also

Using breakpoints, page 89.

__setLogBreak

Syntax

`__setLogBreak(`*`location, message, mesg_type, condition,`*
               `cond_type``)`

Parameters

`location`	A string with a location description. Choose between: A C-SPY expression, whose value evaluates to a valid address, such as a function, for example `main`. For more information about C-SPY expressions, see *C-SPY expressions*, page 74. An absolute address, on the form *`zone`*`:`*`hexaddress`* or simply *`hexaddress`* (for example `Code:42`). *`zone`* refers to C-SPY memory zones and specifies in which memory the address belongs. A source location in your C source code, using the syntax `{`*`filename`*`}.`*`row`*`.`*`col`*. For example`{D:\\src\\prog.c}.22.3` sets a breakpoint on the third character position on row 22 in the source file `prog.c`. Note that the Source location type is usually meaningful only for code breakpoints.

message	The message text
msg_type	The message type; choose between: `TEXT`, the message is written word for word. `ARGS`, the message is interpreted as a comma-separated list of C-SPY expressions or strings.
condition	The breakpoint condition (string)
cond_type	The condition type; either `"CHANGED"` or `"TRUE"` (string)

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. The same value must be used when you want to clear the breakpoint.
Unsuccessful	0

Table 27: __setLogBreak return values

Description

Sets a log breakpoint, that is, a breakpoint which is triggered when an instruction is fetched from the specified location. If you have set the breakpoint on a specific machine instruction, the breakpoint will be triggered and the execution will temporarily halt and print the specified message in the C-SPY Debug Log window.

Example

```
__var logBp1;
__var logBp2;

logOn()
{
  logBp1 = __setLogBreak ("{C:\\temp\\Utilities.c}.23.1",
    "\"Entering trace zone at :\", #PC:%X", "ARGS", "1", "TRUE");
  logBp2 = __setLogBreak ("{C:\\temp\\Utilities.c}.30.1",
    "Leaving trace zone...", "TEXT", "1", "TRUE");
}

logOff()
{
  __clearBreak(logBp1);
  __clearBreak(logBp2);
}
```

See also

Formatted output, page 188 and *Using breakpoints*, page 89.

__setSimBreak

Syntax `__setSimBreak(`*`location, access, action`*`)`

Parameters

`location`	A string with a location description. Choose between:
	A C-SPY expression, whose value evaluates to a valid address, such as a variable name. For example, `my_var` refers to the location of the variable `my_var`, and `arr[3]` refers to the location of the third element of the array `arr`. For static variables declared with the same name in several functions, use the syntax `my_func::my_static_variable` to refer to a specific variable. For more information about C-SPY expressions, see *C-SPY expressions*, page 74.
	An absolute address, on the form *`zone`*`:`*`hexaddress`* or simply *`hexaddress`* (for example `XData:42`). *`zone`* refers to C-SPY memory zones and specifies in which memory the address belongs.
	A source location in your C source code, using the syntax `{`*`filename`*`}.`*`row.col`*. For example`{D:\\src\\prog.c}.22.3` sets a breakpoint on the third character position on row 22 in the source file `prog.c`. Note that the Source location type is usually meaningful only for code breakpoints.
`access`	The memory access type: `"R"` for read or `"W"` for write
`action`	An expression, typically a call to a macro function, which is evaluated when the breakpoint is detected

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	`0`

Table 28: __setSimBreak return values

Description Use this system macro to set *immediate* breakpoints, which will halt instruction execution only temporarily. This allows a C-SPY macro function to be called when the processor is about to read data from a location or immediately after it has written data. Instruction execution will resume after the action.

This type of breakpoint is useful for simulating memory-mapped devices of various kinds (for instance serial ports and timers). When the processor reads at a memory-mapped location, a C-SPY macro function can intervene and supply the appropriate data. Conversely, when the processor writes to a memory-mapped location, a C-SPY macro function can act on the value that was written.

Applicability

This system macro is only available in the C-SPY Simulator.

__setTraceStartBreak

Syntax

`__setTraceStartBreak(location)`

Parameters

`location`	A string with a location description. Choose between:

A C-SPY expression, whose value evaluates to a valid address, such as a function, for example `main`. For more information about C-SPY expressions, see *C-SPY expressions*, page 74.

An absolute address, on the form *`zone`*`:`*`hexaddress`* or simply *`hexaddress`* (for example `Code:42`). *`zone`* refers to C-SPY memory zones and specifies in which memory the address belongs.

A source location in your C source code, using the syntax `{`*`filename`*`}.`*`row`*`.`*`col`*. For example`{D:\\src\\prog.c}.22.3` sets a breakpoint on the third character position on row 22 in the source file `prog.c`. Note that the Source location type is usually meaningful only for code breakpoints.

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. The same value must be used when you want to clear the breakpoint.
Unsuccessful	0

Table 29: __setTraceStartBreak return values

Description

Sets a breakpoint at the specified location. When that breakpoint is triggered, the trace system is started.

Applicability

This system macro is only available in the C-SPY Simulator.

Example

```
__var startTraceBp;
__var stopTraceBp;

traceOn()
{
  startTraceBp = __setTraceStartBreak
    ("{C:\\TEMP\\Utilities.c}.23.1");
  stopTraceBp = __setTraceStopBreak
    ("{C:\\temp\\Utilities.c}.30.1");
}

traceOff()
{
  __clearBreak(startTraceBp);
  __clearBreak(stopTraceBp);
}
```

See also *Using breakpoints*, page 89.

__setTraceStopBreak

Syntax `__setTraceStopBreak(`*`location`*`)`

Parameters

`location` A string with a location description. Choose between:

A C-SPY expression, whose value evaluates to a valid address, such as a function, for example `main`. For more information about C-SPY expressions, see *C-SPY expressions*, page 74.

An absolute address, on the form *`zone`*`:`*`hexaddress`* or simply *`hexaddress`* (for example `Code:42`). *`zone`* refers to C-SPY memory zones and specifies in which memory the address belongs.

A source location in your C source code, using the syntax `{`*`filename`*`}.`*`row`*`.`*`col`*. For example`{D:\\src\\prog.c}.22.3` sets a breakpoint on the third character position on row 22 in the source file `prog.c`. Note that the Source location type is usually meaningful only for code breakpoints.

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. The same value must be used when you want to clear the breakpoint.
Unsuccessful	`int 0`

Table 30: __setTraceStopBreak return values

Description

Sets a breakpoint at the specified location. When that breakpoint is triggered, the trace system is stopped.

Applicability

This system macro is only available in the C-SPY Simulator.

Example

See *__setTraceStartBreak*, page 211.

See also

Using breakpoints, page 89.

__sourcePosition

Syntax

`__sourcePosition(`*`linePtr, colPtr`*`)`

Parameters

`linePtr`	Pointer to the variable storing the line number
`colPtr`	Pointer to the variable storing the column number

Return value

Result	Value
Successful	Filename string
Unsuccessful	Empty (`""`) string

Table 31: __sourcePosition return values

Description

If the current execution location corresponds to a source location, this macro returns the filename as a string. It also sets the value of the variables, pointed to by the parameters, to the line and column numbers of the source location.

__strFind

Syntax

`__strFind(macro`*`String, pattern, position`*`)`

Parameters

`macroString`	The macro string to search in

pattern	The string pattern to search for
position	The position where to start the search. The first position is `0`

Return value

The position where the pattern was found or `-1` if the string is not found.

Description

This macro searches a given string for the occurrence of another string.

Example

```
__strFind("Compiler", "pile", 0)  = 3
__strFind("Compiler", "foo", 0)   = -1
```

See also

Macro strings, page 186.

__subString

Syntax

`__subString(`*`macroString`*`, `*`position`*`, `*`length`*`)`

Parameters

macroString	The macro string from which to extract a substring
position	The start position of the substring. The first position is 0.
length	The length of the substring

Return value

A substring extracted from the given macro string.

Description

This macro extracts a substring from another string.

Example

```
__subString("Compiler", 0, 2)
```

The resulting macro string contains `Co`.

```
__subString("Compiler", 3, 4)
```

The resulting macro string contains `pile`.

See also

Macro strings, page 186.

__targetDebuggerVersion

Syntax

`__targetDebuggerVersion`

Return value

A string that represents the version number of the C-SPY debugger processor module.

Description

This macro returns the version number of the C-SPY debugger processor module.

Example

```
__var toolVer;
toolVer = __targetDebuggerVersion();
__message "The target debugger version is, ", toolVer;
```

__toLower

Syntax

`__toLower(`*`macroString`*`)`

Parameter

`macroString`	Any macro string

Return value

The converted macro string.

Description

This macro returns a copy of the parameter string where all the characters have been converted to lower case.

Example

`__toLower("IAR")`

The resulting macro string contains `iar`.

`__toLower("Mix42")`

The resulting macro string contains `mix42`.

See also

Macro strings, page 186.

__toString

Syntax

`__toString(C`*`_string, maxlength`*`)`

Parameter

`C_string`	Any null-terminated C string
`maxlength`	The maximum length of the returned macro string

Return value

Macro string.

Description

This macro is used for converting C strings (`char*` or `char[]`) into macro strings.

Example

Assuming your application contains this definition:

```
char const * hptr = "Hello World!";
```

this macro call:

```
__toString(hptr, 5)
```

would return the macro string containing `Hello`.

See also

Macro strings, page 186.

__toUpper

Syntax

`__toUpper(`*`macroString`*`)`

Parameter

`macroString` is any macro string.

Return value

The converted string.

Description

This macro returns a copy of the parameter *`macroString`* where all the characters have been converted to upper case.

Example

```
__toUpper("string")
```

The resulting macro string contains `STRING`.

See also

Macro strings, page 186.

__unloadImage

Syntax

`__unloadImage(`*`module_id`*`)`

Parameter

`module_id`	An integer which represents a unique module identification, which is retrieved as a return value from the corresponding `__loadImage` C-SPY macro.

Return value

Value	Result
`module_id`	A unique module identification (the same as the input parameter).
`int 0`	The unloading failed.

Table 32: __unloadImage return values

Description

Unloads debug information from an already downloaded image.

See also	*Loading multiple images*, page 43 and *Images*, page 265.

__writeFile

Syntax	`__writeFile(`*`fileHandle, value`*`)`
Parameters	
`fileHandle`	A macro variable used as filehandle by the `__openFile` macro
`value`	An integer
Return value	`int 0`
Description	Prints the integer value in hexadecimal format (with a trailing space) to the file *`file`*.

Note: The `__fmessage` statement can do the same thing. The `__writeFile` macro is provided for symmetry with `__readFile`.

__writeFileByte

Syntax	`__writeFileByte(`*`fileHandle`*`, `*`value`*`)`
Parameters	
`fileHandle`	A macro variable used as filehandle by the `__openFile` macro
`value`	An integer in the range `0-255`
Return value	`int 0`
Description	Writes one byte to the file *`fileHandle`*.

__writeMemory8, __writeMemoryByte

Syntax

```
__writeMemory8(value, address, zone)
__writeMemoryByte(value, address, zone)
```

Parameters

`value`	The value to be written (integer)
`address`	The memory address (integer)

zone	The memory zone name (string); for more information about available zones, see *C-SPY memory zones*, page 114.

Return value `int 0`

Description Writes one byte to a given memory location.

Example
```
__writeMemory8(0x2F, 0x8020, "XData");
```

__writeMemory16

Syntax `__writeMemory16(`*`value, address, zone`*`)`

Parameters

value	The value to be written (integer)
address	The memory address (integer)
zone	The memory zone name (string); for more information about available zones, see *C-SPY memory zones*, page 114.

Return value `int 0`

Description Writes two bytes to a given memory location.

Example
```
__writeMemory16(0x2FFF, 0x8020, "XData");
```

__writeMemory32

Syntax `__writeMemory32(`*`value, address, zone`*`)`

Parameters

value	The value to be written (integer)
address	The memory address (integer)
zone	The memory zone name (string); for more information about available zones, see *C-SPY memory zones*, page 114.

Return value `int 0`

Description Writes four bytes to a given memory location.

Example

```
__writeMemory32(0x5555FFFF, 0x8020, "XData");
```

The C-SPY Command Line Utility—cspybat

This chapter describes how you can execute C-SPY® in batch mode, using the C-SPY Command Line Utility—cspybat.exe. More specifically, this means:

- Using C-SPY in batch mode
- Summary of C-SPY command line options
- Reference information on C-SPY command line options.

Using C-SPY in batch mode

You can execute C-SPY in batch mode if you use the command line utility `cspybat`, installed in the directory `common\bin`.

INVOCATION SYNTAX

The invocation syntax for `cspybat` is:

```
cspybat processor_DLL driver_DLL debug_file [cspybat_options]
        --backend driver_options
```

Note: In those cases where a filename is required—including the DLL files—you are recommended to give a full path to the filename.

Parameters

The parameters are:

Parameter	Description
processor_DLL	The processor-specific DLL file; available in `8051\bin`.
driver_DLL	The C-SPY driver DLL file; available in `8051\bin`.
debug_file	The object file that you want to debug (filename extension `d51`).
cspybat_options	The command line options that you want to pass to `cspybat`. Note that these options are optional. For information about each option, see *Reference information on C-SPY command line options*, page 228.

Table 33: cspybat parameters

Parameter	Description
`--backend`	Marks the beginning of the parameters to the C-SPY driver; all options that follow will be sent to the driver. Note that this option is mandatory.
`driver_options`	The command line options that you want to pass to the C-SPY driver. Note that some of these options are mandatory and some are optional. For information about each option, see *Reference information on C-SPY command line options*, page 228.

Table 33: cspybat parameters (Continued)

Example

This example starts `cspybat` using the simulator driver:

```
EW_DIR\common\bin\cspybat EW_DIR\8051\bin\8051proc.dll
EW_DIR\8051\bin\8051sim.dll PROJ_DIR\myproject.d51 --plugin
EW_DIR\8051\bin\8051bat.dll --backend -B
--proc_set_exit_breakpoint --proc_set_putchar_breakpoint
-proc_set_getchar_breakpoint --proc_core plain --proc_code_model
near --proc_nr_virtual_regs 8 -proc_pdata_bank_reg_addr 0xB3
--proc_dptr_nr_of 1 --proc_data_model small -p
EW_DIR\8051\config\devices\_generic\io8051.ddf --proc_driver sim
```

where *`EW_DIR`* is the full path of the directory where you have installed IAR Embedded Workbench

and where *`PROJ_DIR`* is the path of your project directory.

OUTPUT

When you run `cspybat`, these types of output can be produced:

- Terminal output from `cspybat` itself

  All such terminal output is directed to `stderr`. Note that if you run `cspybat` from the command line without any arguments, the `cspybat` version number and all available options including brief descriptions are directed to `stdout` and displayed on your screen.

- Terminal output from the application you are debugging

  All such terminal output is directed to `stdout`, provided that you have used the `--plugin` option. See *--plugin*, page 241.

- Error return codes

  `cspybat` return status information to the host operating system that can be tested in a batch file. For *successful*, the value `int 0` is returned, and for *unsuccessful* the value `int 1` is returned.

USING AN AUTOMATICALLY GENERATED BATCH FILE

When you use C-SPY in the IDE, C-SPY generates a batch file `*projectname*.cspy.bat` every time C-SPY is initialized. You can find the file in the directory `$PROJ_DIR$\settings`. This batch file contains the same settings as in the IDE, and with minimal modifications, you can use it from the command line to start `cspybat`. The file also contains information about required modifications.

Summary of C-SPY command line options

GENERAL CSPYBAT OPTIONS

`--backend`	Marks the beginning of the parameters to be sent to the C-SPY driver (mandatory).
`--code_coverage_file`	Enables the generation of code coverage information and places it in a specified file.
`--cycles`	Specifies the maximum number of cycles to run.
`--download_only`	Downloads a code image without starting a debug session afterwards.
`--macro`	Specifies a macro file to be used.
`--plugin`	Specifies a plugin file to be used.
`--silent`	Omits the sign-on message.
`--timeout`	Limits the maximum allowed execution time.

OPTIONS AVAILABLE FOR ALL C-SPY DRIVERS

`-B`	Enables batch mode (mandatory).
`--core`	Specifies the core to be used.
`--nr_of_extra_images`	Specifies that extra debug images will be downloaded.
`-p`	Specifies the device description file to be used.
`--proc_code_model`	Specifies the code model.
`--proc_codebank_end`	Specifies the end address of the banked area.
`--proc_codebank_mask`	Sets the bank register as the active bits.

`--proc_codebank_reg`	Specifies the SFR address for the code bank register.
`--proc_codebank_start`	Specifies the start address of the banked area.
`--proc_core`	Specifies the core type.
`--proc_data_addr_24`	Enables the use of 24 bits wide data addresses.
`--proc_data_model`	Specifies the data model.
`--proc_DPHn`	Specifies the SFR address for the `DPH` registers.
`--proc_DPLn`	Specifies the SFR address for the `DPL` registers.
`--proc_dptr_DPS`	Specifies the SFR address of the `DPTR` select register.
`--proc_dptr_mask`	Specifies the active bits in the `DPTR` select register.
`--proc_dptr_nr_of`	Specifies the number of `DPTR`s on the device.
`--proc_dptr_switch_method`	Specifies the method to change the `DPTR` select register.
`--proc_dptr_visibility`	Specifies the type of `DPTR` visibility in the SFR area.
`--proc_DPXn`	Specifies the SFR address for the `DPX` registers.
`--proc_driver`	Specifies which driver to use.
`--proc_exclude_exit_breakpoint`	Disables the breakpoint on the `exit` label.
`--proc_exclude_getchar_breakpoint`	Disables the breakpoint on the `getchar` function.
`--proc_exclude_putchar_breakpoint`	Disables the breakpoint on the `putchar` function.
`--proc_extended_stack`	Specifies the address of the extended stack.
`--proc_nr_virtual_regs`	Specifies the number of virtual registers.
`--proc_pdata_bank_ext_reg_addr`	Specifies the address of the `MOVX@R0` instructions on devices with a 24-bit address bus.
`--proc_pdata_bank_reg_addr`	Specifies the address of the `MOVX@R0` instructions on devices with an 8- or 16-bit address bus.
`--proc_silent`	Sets silent operation.

OPTIONS AVAILABLE FOR THE SIMULATOR DRIVER

`--disable_interrupts`	Disables the interrupt simulation.
`--mapu`	Activates memory access checking.
`--sim_guard_stacks`	Stops execution of the simulator if your application attempts to write outside any of the stacks.

OPTIONS AVAILABLE FOR THE TEXAS INSTRUMENTS DRIVER

`--boot_lock`	Locks the boot sector.
`--communication_logfile`	Logs communication between C-SPY and the target system.
`--debug_lock`	Locks the debug interface.
`--erase_flash`	Erases all flash memory before download.
`--lock_bits`	Protects the downloaded code against read/write accesses.
`--lock_flash`	Locks the flash memory.
`--number_of_banks`	Sets the number of memory banks on the device.
`--reduce_speed`	Slows down communication between your host and the target board.
`--retain_memory`	Makes sure only the changed, new, or updated pages will be downloaded to flash.
`--retain_pages`	Makes certain pages remain untouched during download.
`--stack_overflow`	Enables stack overflow warnings.
`--suppress_download`	Suppresses download of your application to flash memory.
`--verify_download`	Verifies that the program data has been correctly transferred.

OPTIONS AVAILABLE FOR THE FS2 DRIVER

`--fs2_configuration`	Specifies the core to be used.
`--fs2_flash_cfg_entry`	Specifies how to write to the flash memory.
`--fs2_flash_in_code`	Specifies where program flash memory is located.
`--fs2_ram_in_code`	Specifies the location for program code in RAM.
`--suppress_download`	Suppresses download of your application to flash memory.
`--verify_download`	Verifies that the program data has been correctly transferred.

OPTIONS AVAILABLE FOR THE INFINEON DRIVER

`--connect_to`	Specifies which DAS debug port to connect to.
`--erase_flash`	Erases all flash memory before download.
`--key_noN`	Specifies the key value for DAS server security key number *N*.
`--server_address`	Specifies the address for the server on which the DAS server software is running.
`--server_name`	Specifies the type of DAS server to connect to.
`--software_breakpoints`	Enables the use of software breakpoints.
`--suppress_download`	Suppresses download of your application to flash memory.
`--verify_download`	Verifies that the program data has been correctly transferred.

OPTIONS AVAILABLE FOR THE NORDIC SEMICONDUCTOR DRIVER

`--suppress_download`	Suppresses download of your application to flash memory.
`--verify_download`	Verifies that the program data has been correctly transferred.

OPTIONS AVAILABLE FOR THE ROM-MONITOR DRIVER

`--drv_communication_log`	Logs communication between C-SPY and the ROM-monitor firmware.
`--rom_serial_port`	Specifies communication options for the ROM-monitor driver.
`--suppress_download`	Suppresses download of your application to flash memory.
`--toggle_DTR`	Toggles the DTR signal on the target board whenever the debugger is reset.
`--toggle_RTS`	Toggles the RTS signal on the target board whenever the debugger is reset.
`--verify_all`	Verifies that the program data has been correctly transferred.

OPTIONS AVAILABLE FOR THE ANALOG DEVICES DRIVER

`--ADe_protocol`	Specifies the use of an ADe device.
`--baud_rate`	Specifies the communication speed between C-SPY and the evaluation board.
`--core_clock_frequency`	Specifies the default CPU clock frequency.
`--erase_data_flash`	Erases the data flash area during download.
`--handshake_at_9600`	Handshakes at 9600 baud.
`--serial_port`	Specifies the port to be used for contact with the evaluation board.
`--suppress_download`	Suppresses download of your application to flash memory.
`--verify_all`	Verifies that the program data has been correctly transferred.

OPTIONS AVAILABLE FOR THE SILABS DRIVER

`--baud_rate`	Specifies the communication speed between C-SPY and the evaluation board.

`--devices_after`	Specifies the number of devices in the chain after the device to be debugged.
`--devices_before`	Specifies the number of devices in the chain before the device to be debugged.
`--drv_silabs_page_size`	Selects the size of the flash page.
`--multiple_devices`	Specifies that more than one device is connected to the same JTAG interface.
`--power_target`	Provides power to the target hardware.
`--preserve_hex_files`	Preserves hexadecimal files when flashing the device.
`--registers_after`	Specifies the number of JTAG registers in the chain after the device to be debugged.
`--registers_before`	Specifies the number of JTAG registers in the chain before the device to be debugged.
`--serial_port`	Specifies the port to be used for contact with the evaluation board.
`--silabs_2wire_interface`	Specifies the interface to the Silabs 2-wire debugging interface.
`--suppress_download`	Suppresses download of your application to flash memory.
`--usb_interface`	Specifies the download interface to USB.
`--verify_all`	Verifies that the program data has been correctly transferred.

Reference information on C-SPY command line options

This section gives detailed reference information about each `cspybat` option and each option available to the C-SPY drivers.

--ADe_protocol

Syntax `--ADe_protocol`

Applicability The C-SPY Analog Devices driver.

Description

Specifies that you are debugging using an ADe device.

Project>Options>Debugger>Analog Devices>Download>ADe device protocol

-B

Syntax

`-B`

Applicability

All C-SPY drivers.

Description

Use this option to enable batch mode.

--backend

Syntax

`--backend {`*`driver options`*`}`

Parameters

`driver options`	Any option available to the C-SPY driver you are using.

Applicability

Sent to `cspybat` (mandatory).

Description

Use this option to send options to the C-SPY driver. All options that follow `--backend` will be passed to the C-SPY driver, and will not be processed by `cspybat` itself.

--baud_rate

Syntax

`--baud_rate` *`rate`*

Parameters

`rate` is a value corresponding to the communication speed that you want to set.

For the Analog Devices driver, *`rate`* can be one of these:

`2400`	Sets the communication speed to 2400 bps
`4800`	Sets the communication speed to 4800 bps
`9600`	Sets the communication speed to 9600 bps
`19200`	Sets the communication speed to 19200 bps
`38400`	Sets the communication speed to 38400 bps

`57600`	Sets the communication speed to 57600 bps
`115200`	Sets the communication speed to 115200 bps

For the Silabs driver, *`rate`* can be one of these:

`1`	Sets the communication speed to 115200 bps
`2`	Sets the communication speed to 57600 bps
`3`	Sets the communication speed to 38400 bps
`4`	Sets the communication speed to 9600 bps
`5`	Sets the communication speed to 2400 bps

Applicability

- The C-SPY Analog Devices driver (for both serial and USB communication)
- The C-SPY Silabs driver.

Description

This option specifies the communication speed between C-SPY and the evaluation board. For the Analog Devices driver, if the option `--handshake_at_9600` is not used the only available speed is `115200`.

See also

The option *--handshake_at_9600*, page 237.

Project>Options>Debugger>*Driver*>Serial Port>Baud rate

--boot_lock

Syntax

`--boot_lock`

Applicability

The C-SPY Texas Instruments driver.

Description

Use this option to protect your application on the microcontroller by locking the boot sector. To remove this lock, you must use the `--erase_flash` option.

Project>Options>Debugger>Texas Instruments>Download>Flash Lock Protection>Boot block lock

--code_coverage_file

Syntax	`--code_coverage_file` *`file`*
Parameters	*`file`* The name of the destination file for the code coverage information.
Applicability	Sent to `cspybat`.
Description	Use this option to enable the generation of code coverage information. The code coverage information will be generated after the execution has completed and you can find it in the specified file. Note that this option requires that the C-SPY driver you are using supports code coverage. If you try to use this option with a C-SPY driver that does not support code coverage, an error message will be directed to `stderr`.
See also	*Code coverage*, page 157.

--communication_logfile

Syntax	`--communication_logfile` *`path`*
Parameters	*`path`* Where the log file will be saved.
Applicability	The C-SPY Texas Instruments driver.
Description	Use this option to log communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the communication protocol is required. This log file can be useful if you intend to contact IAR Systems support for assistance.

Project>Options>Debugger>Texas Instruments>Target>Log communication

--connect_to

Syntax	`--connect_to jtag|usb`
Parameters	`jtag|usb` The type of debug port to connect to.

Applicability	The C-SPY Infineon driver.
Description	Use this option to specify which DAS debug port to connect to.

This option is not available in the IDE.

--core

Syntax	`--core {plain|pl|extended1|e1|extended2|e2}`
Parameters	

`plain|pl|extended1|e1|extended2|e2`	The core you are using. This option reflects the corresponding compiler option. For information about the cores, see the *IAR C/C++ Compiler Reference Guide for 8051*.

Applicability	All C-SPY drivers.
Description	Use this option to specify the core you are using.

--core_clock_frequency

Syntax	`--core_clock_frequency` *n*
Parameters	

n	The frequency, from 0 to 999999999 Hz.

Applicability	The C-SPY Analog Devices driver.
Description	Use this option if you have modified the hardware in such a way that the CPU clock frequency has changed.

Project>Options>Debugger>Analog Devices>Serial Port>Override default CPU clock frequency

--cycles

Syntax	`--cycles cycles`
Parameters	*cycles* The number of cycles to run.
Applicability	Sent to `cspybat`.
Description	Use this option to specify the maximum number of cycles to run. If the target program executes longer than the number of cycles specified, the target program will be aborted. Using this option requires that the C-SPY driver you are using supports a cycle counter, and that it can be sampled while executing.

--debug_lock

Syntax	`--debug_lock`
Applicability	The C-SPY Texas Instruments driver.
Description	Use this option to protect your application on the microcontroller from read and write accesses by locking the debug interface. To remove this lock, you must use the `--erase_flash` option.

Project>Options>Debugger>Texas Instruments>Download>Flash Lock Protection>Debug interface lock

--devices_after

Syntax	`--devices_after number`
Parameters	*number* The number of devices after the device to be debugged.
Applicability	The C-SPY Silabs driver.
Description	Use this option to specify the number of devices in the chain after the device to be debugged. This option must be specified when the option `--multiple_devices` is used.

See also	The option --*multiple_devices*, page 240.

Project>Options>Debugger>Silabs>Download>JTAG chain>Multiple devices>Devices>After

--devices_before

Syntax	`--devices_before number`
Parameters	*`number`* The number of devices before the device to be debugged.
Applicability	The C-SPY Silabs driver.
Description	Use this option to specify the number of devices in the chain before the device to be debugged. This option must be specified when the option `--multiple_devices` is used.
See also	The option --*multiple_devices*, page 240.

Project>Options>Debugger>Silabs>Download>JTAG chain>Multiple devices>Devices>Before

--disable_interrupts

Syntax	`--disable_interrupts`
Applicability	The C-SPY Simulator driver.
Description	Use this option to disable the interrupt simulation.

To set this option, choose **Simulator>Interrupt Setup** and deselect the **Enable interrupt simulation** option.

--download_only

Syntax	`--download_only`
Applicability	Sent to `cspybat`.
Description	Use this option to download the code image without starting a debug session afterwards.

To set related options, choose:

Project>Download

--drv_communication_log

Syntax	`--drv_communication_log` *`path`*
Parameters	

`path`	Where the log file will be saved.

Applicability	The C-SPY ROM-monitor driver.
Description	Use this option to log communication between C-SPY and the ROM-monitor firmware to a file.

Project>Options>Debugger>ROM-Monitor>Serial Port>Log communication

--drv_silabs_page_size

Syntax	`--drv_silabs_page_size {512|1024}`
Parameters	

`512|1024`	The flash page size in bytes.

Applicability	The C-SPY Silabs driver.
Description	Informs C-SPY of the size of the flash page.

Project>Options>Debugger>Silabs>Download>Flash page size

--erase_data_flash

Syntax	`--erase_data_flash`
Applicability	The C-SPY Analog Devices driver.
Description	Use this option to erase the data flash area during download.

Project>Options>Debugger>Analog Devices>Download>Erase data flash

--erase_flash

Syntax

```
--erase_flash
```

Applicability

- The C-SPY Texas Instruments driver
- The C-SPY Infineon driver.

Description

Use this option to erase all flash memory before download.

Project>Options>Debugger>*Driver*>Download>Erase flash

--fs2_configuration

Syntax

`--fs2_configuration` *`core`*

Parameters

`core` is the core type of your device. Choose between:

```
cast51-single-core
m8051ew-single-core
philips51-single-core
handshake51-single-core
```

Applicability

The C-SPY FS2 driver.

Description

Use this option to specify which core your device is.

Project>Options>Debugger>FS2 System Navigator>Target>Configuration

--fs2_flash_cfg_entry

Syntax

`--fs2_flash_cfg_entry` *`label`*

Parameters

`label`	The label for the entry in the `flash.cfg` file that describes how to program the flash memory of the device.

Applicability

The C-SPY FS2 driver.

Description

Use this option to describe to the debugger how to program the flash memory of the device.

Project>Options>Debugger>FS2 System Navigator>Target>Entry in flash.cfg

--fs2_flash_in_code

Syntax

`--fs2_flash_in_code` *`ranges`*

Parameters

`ranges`	One or more memory ranges separated by commas, like this: `0x0000-0x1111,0x2222-0x3333`.

Applicability

The C-SPY FS2 driver.

Description

Use this option to specify where the device has program flash memory.

Project>Options>Debugger>FS2 System Navigator>Target>Flash areas

--fs2_ram_in_code

Syntax

`--fs2_ram_in_code` *`ranges`*

Parameters

`ranges`	One or more memory ranges separated by commas, like this: `0x0000-0x1111,0x2222-0x3333`.

Applicability

The C-SPY FS2 driver.

Description

Use this option to specify where the device has program code in RAM memory, if your device supports code in RAM. This means that software breakpoints will be used in this memory area.

Project>Options>Debugger>FS2 System Navigator>Target>RAM areas

--handshake_at_9600

Syntax

`--handshake_at_9600`

Applicability

The C-SPY Analog Devices driver.

Description

Use this option to handshake at 9600 baud before initiating communication. This option must be used if you debug via the UART interface.

Project>Options>Debugger>Analog Devices>Download>UART debug mode

--key_no*N*

Syntax

`--key_noN value`

Parameters

N	The number of the key, from `1` to `4`.
value	The key value, `0x00000000`–`0x7FFFFFFF`.

Applicability

The C-SPY Infineon driver.

Description

The DAS server has security keys that can be used if they are enabled. The keys can be used to protect access to the device that is debugged. If used, specify the key value for each security key to connect to the server.

Project>Options>Debugger>Infineon>Target>Security keys>Key #*n*

--lock_bits

Syntax

`--lock_bits n`

Parameters

n	The bit range to lock, from `0` to `7`.

Applicability

The C-SPY Texas Instruments driver.

Description

Protects your application on the microcontroller. `--lock_bits 0` protects the whole flash memory from read/write accesses. If the parameter is `1`–`7`, a section of flash pages is protected. Exactly which section varies from device to device; see the documentation for your device. To remove this lock, you must use the `--erase_flash` option.

Project>Options>Debugger>Texas Instruments>Download>Boot block lock

--lock_flash

Syntax

```
--lock_flash
```

Applicability

The C-SPY Texas Instruments driver.

Description

Use this option to protect your application on the microcontroller by locking the flash memory. To remove this lock, you must use the `--erase_flash` option.

Project>Options>Debugger>Texas Instruments>Download>Lock flash memory

--macro

Syntax

```
--macro filename
```

Parameters

`filename`	The C-SPY macro file to be used (filename extension `mac`).

Applicability

Sent to `cspybat`.

Description

Use this option to specify a C-SPY macro file to be loaded before executing the target application. This option can be used more than once on the command line.

See also

Briefly about using C-SPY macros, page 178.

--mapu

Syntax

```
--mapu
```

Applicability

The C-SPY simulator driver.

Description

Specify this option to use the segment information in the debug file for memory access checking. During the execution, the simulator will then check for accesses to unspecified memory ranges. If any such access is found, the C function call stack and a message will be printed on `stderr` and the execution will stop.

See also

Memory access checking, page 116.

To set related options, choose:

Simulator>Memory Access Setup

--multiple_devices

Syntax `--multiple_devices`

Applicability The C-SPY Silabs driver.

Description Use this option to specify that more than one device is connected to the same JTAG interface. In this case, you must also specify `--devices_after`, `--devices_before`, `--registers_after`, and `--registers_before`.

See also --*devices_after*, page 233, --*devices_before*, page 234, --*registers_after*, page 252, and --*registers_before*, page 253.

Project>Options>Debugger>Silabs>Download>JTAG chain>Multiple devices

--nr_of_extra_images

Syntax `--nr_of_extra_images n`

Parameters

n	The number of extra images you want to download.

Applicability All C-SPY drivers.

Description Use this option to specify that extra debug images will be downloaded to the target system.

Project>Options>Debugger>Images

--number_of_banks

Syntax `--number_of_banks {1|2|4|8|16}`

Parameters

`1|2|4|8|16`	The number of banks.

Applicability The C-SPY Texas Instruments driver.

Description Informs C-SPY of the number of memory banks on the device.

Some Texas Instrument devices have built-in support for expanding the program memory. See the device hardware manual to see how much program memory (flash memory) your device has.

Project>Options>Debugger>Texas Instruments>Target>Number of banks

-p

Syntax `-p` *`filename`*

Parameters

`filename`	The device description file to be used.

Applicability All C-SPY drivers.

Description Use this option to specify the device description file to be used. This option is mandatory when you are using the Infineon driver.

See also *Selecting a device description file*, page 41.

--plugin

Syntax `--plugin` *`filename`*

Parameters

`filename`	The plugin file to be used (filename extension `dll`).

Applicability Sent to `cspybat`.

Description Certain C/C++ standard library functions, for example `printf`, can be supported by C-SPY—for example, the C-SPY Terminal I/O window—instead of by real hardware devices. To enable such support in `cspybat`, a dedicated plugin module called `8051bat.dll` located in the `8051\bin` directory must be used.

Use this option to include this plugin during the debug session. This option can be used more than once on the command line.

Note: You can use this option to include also other plugin modules, but in that case the module must be able to work with `cspybat` specifically. This means that the C-SPY plugin modules located in the `common\plugin` directory cannot normally be used with `cspybat`.

--power_target

Syntax	`--power_target`
Applicability	The C-SPY Silabs driver.
Description	Use this option to provide power to the target hardware even after the debug session has been closed. This option is only applicable when `--usb_interface` is used.
See also	*--usb_interface*, page 259.

Project>Options>Debugger>Silabs>Download>USB interface>Continuously power target

--preserve_hex_files

Syntax	`--preserve_hex_files`
Applicability	The C-SPY Silabs driver.
Description	Use this option to preserve the hexadecimal file generated during the download process.

To set this option, use **Project>Options>**Debugger>**Extra Options**.

--proc_code_model

Syntax `--proc_code_model {near|banked|banked_ext2|far}`

Parameters

`near`	Selects Near as the default code model.
`banked`	Selects Banked as the default code model.
`banked_ext2`	Selects Banked_ext2 as the default code model.
`far`	Selects Far as the default code model.

Applicability	All C-SPY drivers (mandatory).
Description	Use this option to specify the default code model.

Project>Options>General Options>Target>Code model

--proc_codebank_end

Syntax `--proc_codebank_end` *`address`*

Parameters

`address`	The end address of the banked area, from `0000` to `FFFF`.

Applicability All C-SPY drivers.

Description Use this option to specify the end address of the banked area. The end address must be specified when `--proc_code_model` is set to `banked`.

Project>Options>General Options>Code Bank>Bank end

--proc_codebank_mask

Syntax `--proc_codebank_mask` *`address`*

Parameters

`address`	The active bits in the bank register, from `0x80` to `0xFF`.

Applicability All C-SPY drivers.

Description Use this option to set the active bits in the bank register when `--proc_code_model` is set to `banked`.

Project>Options>General Options>Code bank>Register mask

--proc_codebank_reg

Syntax `--proc_codebank_reg` *`address`*

Parameters

`address`	The SFR address for the code bank register, from `0x80` to `0xFF`.

Applicability All C-SPY drivers.

Description

Use this option to specify the SFR address for the code bank register when `--proc_code_model` is set to `banked`.

Project>Options>General Options>Code Bank>Register address

--proc_codebank_start

Syntax

`--proc_codebank_start` *`address`*

Parameters

`address`	The start address of the banked area, from `0000` to `FFFF`.

Applicability

All C-SPY drivers.

Description

Use this option to specify the start address of the banked area. The start address must be specified when `--proc_code_model banked` is specified.

Project>Options>General Options>Code Bank>Bank start

--proc_core

Syntax

`--proc_core` *`core`*

Parameters

`core` is the type of 8051 core you are using. Choose between:

`plain`

`extended1`

`extended2`

Applicability

All C-SPY drivers (mandatory).

Description

Use this option to specify the instruction set extensions and other extensions that your application uses.

Project>Options>General Options>Target>CPU core

--proc_data_addr_24

Syntax `--proc_data_addr_24`

Applicability All C-SPY drivers.

Description Use this option to enable the use of 24-bit data addresses; mandatory when the device has a 24-bit address bus. If the option is omitted, the default is 16-bit addresses.

Project>Options>General Options>Data Pointer>Size

--proc_data_model

Syntax `--proc_data_model {tiny|small|large|far|generic}`

Parameters

`tiny`	Selects Tiny as the default data model.
`small`	Selects Small as the default data model.
`large`	Selects Large as the default data model.
`far`	Selects Far as the default data model.
`generic`	Selects Generic as the default data model.

Applicability All C-SPY drivers (mandatory).

Description Use this option to specify the default data model.

Project>Options>General Options>Target>Code model

--proc_DPH*n*

Syntax `--proc_DPHn address`

Parameters

n	The register number; *n* can be 1-7.
address	The address can be from `0x80` to `0xFF`.

Applicability All C-SPY drivers.

Description	Use this option to specify the SFR address of the high part of the `DPTR1`–`DPTR7` registers for your device. This option requires that `--proc_dptr_nr_of` is set to greater than `1` and that `--proc_dptr_visibility` is used.
See also	*--proc_dptr_nr_of*, page 247 and *--proc_dptr_visibility*, page 248

Project>Options>General Options>Data Pointer>DPTR addresses>Configure

--proc_DPL*n*

Syntax `--proc_DPHn address`

Parameters

n	The register number; *n* can be 1-7.
`address`	The address can be from `0x80` to `0xFF`.

Applicability All C-SPY drivers.

Description Use this option to specify the SFR address of the low part of the `DPTR1`–`DPTR7` registers for your device. This option requires `--proc_dptr_nr_of` to be set to greater than `1` and `--proc_dptr_visibility` to be set to `separate`.

See also *--proc_dptr_nr_of*, page 247 and *--proc_dptr_visibility*, page 248

Project>Options>General Options>Data Pointer>DPTR addresses>Configure

--proc_dptr_DPS

Syntax `--proc_dptr_DPS address`

Parameters

address	The SFR address of the DPTR select register, from `0x80` to `0xFF`.

Applicability All C-SPY drivers.

Description

Use this option to specify the SFR address of the DPTR select register that switches DPTRs on your device. This option requires `--proc_dptr_nr_of` to be set to greater than `1` and `--proc_dptr_visibility` to be set to `separate`.

Project>Options>General Options>Data Pointer>DPTR select>Select register

--proc_dptr_mask

Syntax

`--proc_dptr_mask` *`number`*

Parameters

`number`	The active bits in the DPTR select register, from `0x80` to `0xFF`.

Applicability

All C-SPY drivers.

Description

Use this option to specify the active bits in the DPTR select register.

Project>Options>General Options>Data Pointer>DPTR select>Set using XOR/AND>Mask

--proc_dptr_nr_of

Syntax

`--proc_dptr_nr_of` *`number`*

Parameters

`number`	The number of DPTRs on the device, from 1 to 8.

Applicability

All C-SPY drivers.

Description

Use this option to specify the number of DPTRs on the device. The default value is 1.

Project>Options>General Options>Data Pointer>Number of DPTRs

--proc_dptr_switch_method

Syntax

`--proc_dptr_switch_method {INC|XOR}`

Parameters

`INC|XOR`	The method to change the DPTR select register.

Applicability	All C-SPY drivers.
Description	Use this option to specify the method to change the DPTR select register. You can use the INC method if your device has the DPTR mask register in the least significant bit and is followed by a write-protected bit. You must specify the switch method if `--proc_dptr_nr_of` is greater than 1.

Project>Options>General Options>Data Pointer>DPTR select>Toggle using INC

Project>Options>General Options>Data Pointer>DPTR select>Set using XOR/AND

--proc_dptr_visibility

Syntax `--proc_dptr_visibility {separate|shadowed}`

Parameters

`separate|shadowed`	The type of DPTR visibility in the SFR area.

Applicability	All C-SPY drivers.
Description	Use this option to specify the number of DPTRs on the device. If all DPTRs share the same address for `DPL`, `DPH`, and `DPX` (if applicable), choose `shadowed`. Otherwise, choose `separate`. The visibility must be specified if `--proc_dptr_nr_of` is greater than 1.

Project>Options>General Options>Data Pointer>DPTR addresses

--proc_DPX*n*

Syntax `--proc_DPX`*`n address`*

Parameters

`n`	The register number; *`n`* can be `1`–`7`. Omit *`n`* when you are referring to the `DPX` register.
`address`	The address can be from `0x80` to `0xFF`.

Applicability All C-SPY drivers.

Description
Use this option to specify the SFR address for the `DPTR1`–`DPTR7` registers if your device has a 24-bit address bus. This option requires `--proc_dptr_nr_of` to be set to greater than `1` and `--proc_dptr_visibility` to be set to `separate`.

See also
--proc_dptr_nr_of, page 247 and *--proc_dptr_visibility*, page 248

Project>Options>General Options>Data Pointer>DPTR addresses>Configure

--proc_driver

Syntax
`--proc_driver {sim|ad|rom|silabs|chipcon|infineon|3rd_party}`

Parameters

`sim`	Specifies the simulator driver.
`ad`	Specifies the Analog Devices driver.
`rom`	Specifies the ROM-monitor driver.
`silabs`	Specifies the Silabs driver.
`chipcon`	Specifies the Texas Instruments driver.
`infineon`	Specifies the Infineon driver.
`3rd_party`	Specifies the third-party driver.

Applicability
All C-SPY drivers (mandatory).

Description
Use this option to specify the driver you are using.

Project>Options>Debugger>Setup>Driver

--proc_exclude_exit_breakpoint

Syntax
`--proc_exclude_exit_breakpoint`

Applicability
All C-SPY drivers.

Description
Use this option in the CLIB runtime environment to disable the system breakpoint on the `exit` label. Breakpoints are a critical resource in many hardware drivers. For more information, see *Breakpoint consumers*, page 94.

Project>Options>Debugger>Setup>Exclude system breakpoints on

--proc_exclude_getchar_breakpoint

Syntax	`--proc_exclude_getchar_breakpoint`
Applicability	All C-SPY drivers.
Description	Use this option in the CLIB runtime environment to disable the system breakpoint on the `getchar` function when your application is linked with I/O emulation modules. Breakpoints are a critical resource in many hardware drivers. For more information, see *Breakpoint consumers*, page 94.

Project>Options>Debugger>Setup>Exclude system breakpoints on

--proc_exclude_putchar_breakpoint

Syntax	`--proc_exclude_putchar_breakpoint`
Applicability	All C-SPY drivers.
Description	Use this option in the CLIB runtime environment to disable the system breakpoint on the `putchar` function when your application is linked with I/O emulation modules. Breakpoints are a critical resource in many hardware drivers. For more information, see *Breakpoint consumers*, page 94.

Project>Options>Debugger>Setup>Exclude system breakpoints on

--proc_extended_stack

Syntax

`--proc_extended_stack` *`address`*

Parameters

`address`	The address of the extended stack; `00-FFFFFF`.

Applicability

All C-SPY drivers.

Description

Use this option to specify the address of the extended stack if your application supports and uses an extended stack.

Project>Options>General Options>Target>Extended stack at

--proc_nr_virtual_regs

Syntax `--proc_nr_virtual_regs` *`number`*

Parameters

`number`	The number of virtual registers, from 8 to 32.

Applicability All C-SPY drivers.

Description Use this option to specify the number of virtual registers.

Project>Options>General Options>Target>Number of virtual registers

--proc_pdata_bank_ext_reg_addr

Syntax `--proc_pdata_bank_ext_reg_addr` *`address`*

Parameters

`address`	The address can be from `0x80` to `0xFF`.

Applicability All C-SPY drivers.

Description Use this option to specify the address of the `MOVX@R0` instructions on devices with a 24-bit address bus.

Project>Options>General Options>Data Pointer>Page register address>Bit 16-31

--proc_pdata_bank_reg_addr

Syntax `--proc_pdata_bank_reg_addr` *`address`*

Parameters

`address`	The address can be from `0x80` to `0xFF`.

Applicability All C-SPY drivers.

Description Use this option to specify the address of the `MOVX@R0` instructions on devices with a 16-bit address bus or the lower byte of the address on devices with a 24-bit address bus.

Project>Options>General Options>Data Pointer>Page register address>Bit 8-1

--proc_silent

Syntax `--proc_silent`

Applicability All C-SPY drivers.

Description Use this option to disable the output of messages during the debugging.

This option is not available in the IDE.

--reduce_speed

Syntax `--reduce_speed`

Applicability The C-SPY Texas Instruments driver.

Description Use this option to reduce the communication speed between your host computer and the target board. This can be very useful if you use a long cable or encounter communication problems or interference.

Project>Options>Debugger>Texas Instruments>Target>Reduce interface speed

--registers_after

Syntax `--registers_after number`

Parameters

number	Specifies the number of registers after the device to be debugged, `0`–*n*.

Applicability The C-SPY Silabs driver.

Description Use this option to specify the number of JTAG registers in the chain after the device to be debugged. This option must be specified when `--multiple_devices` is used.

See also *--multiple_devices*, page 240.

Project>Options>Debugger>Silabs>Download>JTAG chain>Multiple devices>Devices>After

--registers_before

Syntax	`--registers_before number`
Parameters	*number* — Specifies the number of registers before the device to be debugged, `0-n`.
Applicability	The C-SPY Silabs driver.
Description	Use this option to specify the number of JTAG registers in the chain before the device to be debugged. This option must be specified when the option `--multiple_devices` is used.
See also	--*multiple_devices*, page 240.

Project>Options>Debugger>Silabs>Download>JTAG chain>Multiple devices>Devices>Before

--retain_memory

Syntax	`--retain_memory`
Applicability	The C-SPY Texas Instruments driver.
Description	Use this option to make sure only the changed, new, or updated bytes are downloaded to flash memory, to save flash cycles.
See also	--*suppress_download*, page 257.

Project>Options>Debugger>Texas Instruments>Download>Retain unchanged pages

--retain_pages

Syntax	`--retain_pages n`
Parameters	*n* — The page number, `1-n`.

Applicability	The C-SPY Texas Instruments driver.
Description	Use this option to make sure that certain flash pages are not rewritten during download.

Project>Options>Debugger>Texas Instruments>Download>Retain flash pages

--rom_serial_port

Syntax `--rom_serial_port` *`port speed parity data stop handshake`*

Parameters

`port`	The communication port can be any port from `COM1` to `COM64`.
`speed`	The communication speed can be `300`, `600`, `1200`, `2400`, `4800`, `9600`, `19200`, `38400`, `57600`, or `115200` baud.
`parity`	The parity value must be `N`.
`data`	Only `8` data bits is supported.
`stop`	The stop bit can be `1` or `2`.
`handshake`	The handshaking value can be any of `NONE`, `NONELOW`, or `RTSCTS`.

Applicability The C-SPY ROM-monitor driver.

Description Use this option to specify the communication options for the ROM-monitor driver. C-SPY connects at 9600 baud and then changes to the communication speed of the selected serial port after making the first contact with the evaluation board. If these options have not been specified, C-SPY will try using the COM1 port.

Project>Options>Debugger>ROM-monitor>Serial Port

--serial_port

Syntax `--serial_port` *`port`*

Parameters *`port`* is the communication port that you want to use.

For the Silabs driver, choose between:

`1`	Sets the communication port to COM1.

`2`	Sets the communication port to COM2.
`3`	Sets the communication port to COM3.
`4`	Sets the communication port to COM4.

For the Analog Devices driver, choose between:

`COM1`	Sets the communication port to COM1.
`COM2`	Sets the communication port to COM2.
`COM3`	Sets the communication port to COM3.
`COM4`	Sets the communication port to COM4.
`COM5`	Sets the communication port to COM5.
`COM6`	Sets the communication port to COM6.
`COM7`	Sets the communication port to COM7.
`COM8`	Sets the communication port to COM8.
`COM9`	Sets the communication port to COM9.

Applicability

- The C-SPY Silabs driver
- The C-SPY Analog Devices driver.

Description

Use this option to specify the communication options for the driver. C-SPY tries to connect with the selected serial port when making the first contact with the evaluation board. If you do not specify a port, C-SPY will try using the COM1 port.

Project>Options>Debugger>*Driver*>Serial Port>Port

--server_address

Syntax

`--server_address` *`address`*

Parameters

`address`	The name or IP address of the connected server. Specify `localhost` if the server software is located on your host computer.

Applicability

The C-SPY Infineon driver.

Description

Use this option to specify the server on which the DAS server software is running.

Project>Options>Debugger>Infineon>Target>Server>Address

--server_name

Syntax

```
--server_name name
```

Parameters

name is the type of DAS server to connect to (case sensitive). Choose between:

```
"JTAG over USB Box"
"JTAG over Tantino"
"JTAG over USB Chip"
"UDAS"
```

Applicability

The C-SPY Infineon driver.

Description

Use this option to specify the type of DAS server. If the server is not specified, a dialog box will prompt you for a server name when the debug session starts.

Project>Options>Debugger>Infineon>Target>Server>Address

--silabs_2wire_interface

Syntax

```
--silabs_2wire_interface
```

Applicability

The C-SPY Silabs driver.

Description

The Silicon Labs C8051F3xx/F4xx/F5xx/F9xx devices use the Silabs 2-wire debugging interface (C2). Use this option to set the interface to the Silabs 2-wire debugging interface. You must specify this option to connect to any of these devices.

Project>Options>Debugger>Silabs>Download>SiLabs 2-wire (C2) interface

--silent

Syntax

```
--silent
```

Applicability

Sent to `cspybat`.

Description

Use this option to omit the sign-on message.

--sim_guard_stacks

Syntax	`--sim_guard_stacks`
Applicability	The C-SPY Simulator driver.
Description	Use this option to be alerted if the stack pointers are out of bounds.

Project>Options>Debugger>Simulator>Simulator>Guard stack pointers

--software_breakpoints

Syntax	`--software_breakpoints`
Applicability	The C-SPY Infineon driver.
Description	To extend the number of code breakpoints, software breakpoints can be used. Use this option to make C-SPY use software breakpoints for code breakpoint when you run out of hardware breakpoints.

Project>Options>Debugger>Infineon>Target>Software breakpoints

--stack_overflow

Syntax	`--stack_overflow`
Applicability	The C-SPY Texas Instruments driver.
Description	Use this option to enable IData stack overflow warnings. This is not a runtime check, but is done at the next stop, which means that it will not stop the execution if an IData stack overflow is encountered.

Project>Options>Debugger>Texas Instruments>Target>Enable stack overflow warning

--suppress_download

Syntax

```
--suppress_download
```

Applicability

- The C-SPY Texas Instruments driver
- The C-SPY FS2 driver

- The C-SPY Infineon driver
- The C-SPY Nordic Semiconductor driver
- The C-SPY ROM-monitor driver
- The C-SPY Silabs driver.

Description

Use this option to suppress download of your application to flash memory. If you already have your application in flash memory, select `--suppress_download`. If you do, it is highly recommended that you also use `--verify_download`.

See also

--*verify_download*, page 260 and --*retain_memory*, page 253.

Project>Options>Debugger>*Driver*>Download>Suppress download

--timeout

Syntax

`--timeout` *`milliseconds`*

Parameters

`milliseconds`	The number of milliseconds before the execution stops.

Applicability

Sent to `cspybat`.

Description

Use this option to limit the maximum allowed execution time.

This option is not available in the IDE.

--toggle_DTR

Syntax

`--toggle_DTR`

Applicability

The C-SPY ROM-monitor driver.

Description

Use this option to toggle the DTR signal on your target board whenever the debugger is reset. If the DTR signal is connected to the RESET pin on the microcontroller, toggling the signal will force a target hardware reset.

Project>Options>Debugger>ROM-Monitor>Serial Port>On Reset>Toggle DTR

--toggle_RTS

Syntax `--toggle_RTS`

Applicability The C-SPY ROM-monitor driver.

Description Use this option to toggle the RTS signal on your target board whenever the debugger is reset. If the RTS signal is connected to the RESET pin on the microcontroller, toggling the signal will force a target hardware reset.

Project>Options>Debugger>ROM-Monitor>Serial Port>On Reset>Toggle RTS

--usb_interface

Syntax `--usb_interface`

Applicability The C-SPY Silabs driver.

Description Use this option to specify the download interface as USB. Use this option if you are using a USB debugger adapter.

See also The option *--power_target*, page 242.

Project>Options>Debugger>Silabs>Download>USB interface

--verify_all

Syntax `--verify_all`

Applicability

- The C-SPY ROM-monitor driver
- The C-SPY Analog Devices driver
- The C-SPY Silabs driver.

Description Use this option to verify that the application data is correctly transferred from the driver to the device. This verification increases the programming sequence time.

Project>Options>Debugger>*Driver*>Download>Verify download

--verify_download

Syntax

```
--verify_download {read_back_memory|use_crc16}
```

Parameters

Parameters for the Texas Instruments driver only:

`read_back_memory`	Verifies a target by reading back memory.
`use_crc16`	Verifies a target using on-chip page CRC16.

Applicability

- The C-SPY Texas Instruments driver
- The C-SPY FS2 driver
- The C-SPY Infineon driver
- The C-SPY Nordic Semiconductor driver.

Description

Use this option to verify that the application data is correctly transferred from the driver to the device. This verification increases the programming sequence time, but the `read_back_memory` method increases the time overhead more than the `use_crc16` method.

Project>Options>Debugger>*Driver*>Download>Verify download

Debugger options

This chapter describes the C-SPY® options available in the IAR Embedded Workbench® IDE. More specifically, this means:

- Setting debugger options
- Reference information on general debugger options
- Reference information on C-SPY simulator options
- Reference information on C-SPY Texas Instruments driver options
- Reference information on C-SPY FS2 driver options
- Reference information on C-SPY Infineon driver options
- Reference information on C-SPY Nordic Semiconductor driver options
- Reference information on C-SPY ROM-monitor driver options
- Reference information on C-SPY Silabs driver options
- Reference information on C-SPY third-party driver options.

Setting debugger options

Before you start the C-SPY debugger, you must set some options—both C-SPY generic options and options required for the target system (C-SPY driver-specific options). This section gives detailed information about the options in the **Debugger** category.

To set debugger options in the IDE:

1. Choose **Project>Options** to display the **Options** dialog box.
2. Select **Debugger** in the **Category** list.

For more information about the generic options, see:

- *Setup*, page 263
- *Images*, page 265
- *Extra Options*, page 266

- *Plugins*, page 266.

3 On the **Setup** page, select the appropriate C-SPY driver from the **Driver** drop-down list.

4 To set the driver-specific options, select the appropriate driver from the **Category** list. Depending on which C-SPY driver you are using, different sets of option pages appear.

C-SPY driver	**Available options pages**
C-SPY simulator	*Setup options for the simulator*, page 268
C-SPY Texas Instruments driver	*Download options for Texas Instruments*, page 269 *Target options for Texas Instruments*, page 271
C-SPY FS2 System Navigator driver	*Download options for FS2*, page 272 *Target options for FS2*, page 273
C-SPY Infineon driver	*Download options for Infineon*, page 274 *Target options for Infineon*, page 275
C-SPY Nordic Semiconductor driver	*Download options for Nordic Semiconductor*, page 276
C-SPY ROM-monitor driver	*Download options for the ROM-monitor*, page 277 *Serial Port options for the ROM-monitor*, page 278
C-SPY Analog Devices driver	*Download options for Analog Devices*, page 279 *Serial Port options for Analog Devices*, page 280
C-SPY Silabs driver	*Download options for Silabs*, page 281 *Serial Port options for Silabs*, page 282
Third-party driver	*Third-Party Driver*, page 283.

Table 34: Options specific to the C-SPY drivers you are using

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

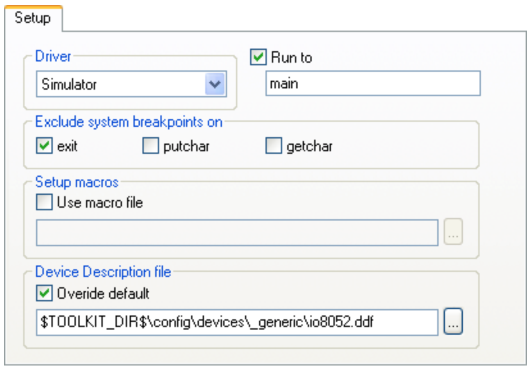
6 When you have set all the required options, click **OK** in the **Options** dialog box.

Reference information on general debugger options

This section gives reference information on general C-SPY debugger options.

Setup

The **Setup** options select the C-SPY driver, the setup macro file, and device description file to use, and specify which default source code location to run to.



Driver: Simulator

[✓] Run to: main

Exclude system breakpoints on: [✓] exit, [ ] putchar, [ ] getchar

[ ] Use macro file

[✓] Overide default

$TOOLKIT_DIR$\config\devices_generic\io8052.ddf

Figure 86: Debugger setup options

Driver

Selects the C-SPY driver for the target system you have:

Third-Party Driver

Texas Instruments

FS2 System Navigator

Infineon

Nordic Semiconductor

ROM-monitor

Analog Devices

Silabs

Simulator

Run to

Specifies the location C-SPY runs to when the debugger starts after a reset. By default, C-SPY runs to the `main` function.

To override the default location, specify the name of a different location you want C-SPY to run to. You can specify assembler labels or whatever can be evaluated as such, for example function names.

If the option is deselected, the program counter will contain the regular hardware reset address at each reset.

Exclude system breakpoints on

Controls the use of system breakpoints in the CLIB runtime environment. If the C-SPY Terminal I/O window is not required or if you do not need a breakpoint on the `exit` label, you can save hardware breakpoints by not reserving system breakpoints. Deselect or select the options **exit**, **putchar**, and **getchar**, respectively, if you want or do not want C-SPY to use system breakpoints for these. For more information, see *Breakpoint consumers*, page 94.

In the DLIB runtime environment, C-SPY will always set a system breakpoint on the `__DebugBreak` label. You cannot disable this behavior.

Setup macros

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

Device description file

A default device description file is selected automatically based on your project settings. To override the default file, select **Override default** and specify an alternative file. A browse button is available for your convenience.

For information about the device description file, see *Modifying a device description file*, page 44.

Device description files for each 8051 device are provided in the directory `8051\config\devices` and have the filename extension `ddf`.

Images

The **Images** options control the use of additional debug files to be downloaded.

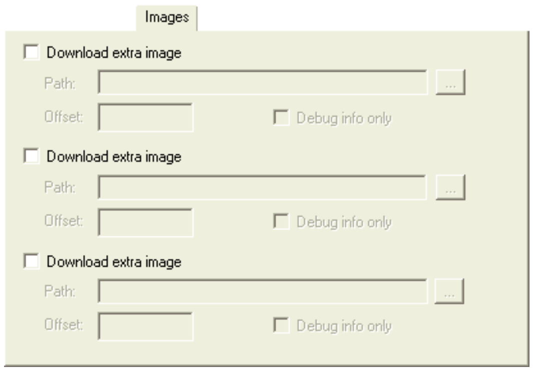


Figure 87: Debugger images options

Download extra Images

Controls the use of additional debug files to be downloaded:

Path	Specify the debug file to be downloaded. A browse button is available for your convenience.
Offset	Specify an integer that determines the destination address for the downloaded debug file.
Debug info only	Makes the debugger download only debug information, and not the complete debug file.

If you want to download more than three images, use the related C-SPY macro, see *__loadImage*, page 197.

For more information, see *Loading multiple images*, page 43.

Extra Options

The **Extra Options** page provides you with a command line interface to C-SPY.

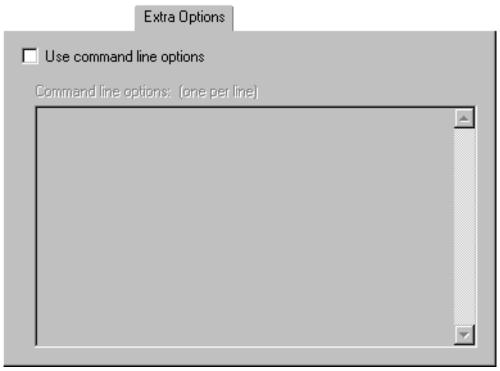


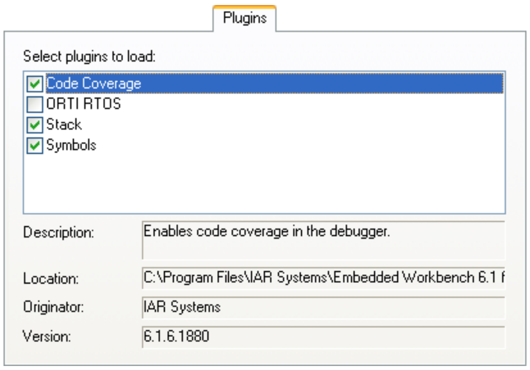
Figure 88: Debugger extra options

Use command line options

Specify additional command line arguments to be passed to C-SPY (not supported by the GUI).

Plugins

The **Plugins** options select the C-SPY plugin modules to be loaded and made available during debug sessions.



☑ Code Coverage
☐ ORTI RTOS
☑ Stack
☑ Symbols

Description: Enables code coverage in the debugger.
Location: C:\Program Files\IAR Systems\Embedded Workbench 6.1 f
Originator: IAR Systems
Version: 6.1.6.1880

Figure 89: Debugger plugin options

Select plugins to load

Selects the plugin modules to be loaded and made available during debug sessions. The list contains the plugin modules delivered with the product installation.

Description

Describes the plugin module.

Location

Informs about the location of the plugin module.

Generic plugin modules are stored in the `common\plugins` directory. Target-specific plugin modules are stored in the `8051\plugins` directory.

Originator

Informs about the originator of the plugin module, which can be modules provided by IAR Systems or by third-party vendors.

Version

Informs about the version number.

Reference information on C-SPY simulator options

This section gives reference information on C-SPY simulator options.

Setup options for the simulator

The simulator **Setup** options control the behavior of the C-SPY simulator.

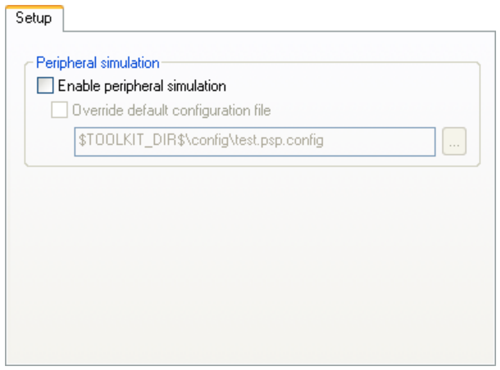

Setup
Peripheral simulation
Enable peripheral simulation
Override default configuration file
$TOOLKIT_DIR$\config\test.psp.config
...

Figure 90: Simulator setup options

Peripheral simulation

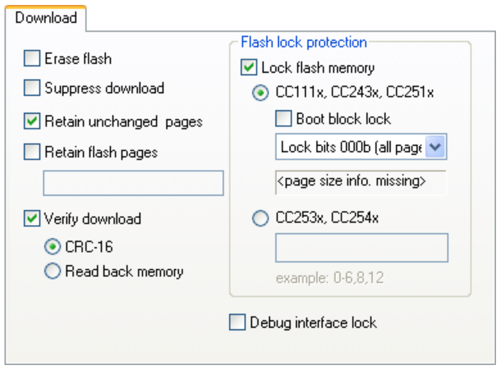
These options set up peripheral simulation, which requires a plugin from a third-party vendor. For information, see the PDF `EW_PeripheralSimulationGuide.pdf` in the *`EW_DIR`*`\8051\doc\` directory and the examples in the *`EW_DIR`*`\8051\plugins\simulation` directory.

Reference information on C-SPY Texas Instruments driver options

This section gives reference information on C-SPY Texas Instruments driver options.

Download options for Texas Instruments

The Texas Instruments **Download** options control the download.


Download

Flash lock protection

[ ] Erase flash

[ ] Suppress download

[✓] Retain unchanged pages

[ ] Retain flash pages

[✓] Verify download

(●) CRC-16

( ) Read back memory

[✓] Lock flash memory

(●) CC111x, CC243x, CC251x

[ ] Boot block lock

Lock bits 000b (all page

<page size info. missing>

( ) CC253x, CC254x

example: 0-6,8,12

[ ] Debug interface lock

Figure 91: Download options for Texas Instruments

Erase flash

Erases all flash memories before download.

Suppress download

Disables the downloading of code, while preserving the present content of the flash memory. This command is useful if you want to debug an application that already resides in target memory. The implicit reset performed by C-SPY at startup is not disabled, though.

If this option is combined with the **Verify download** option, the debugger will read back the code image from non-volatile memory and verify that it is identical to the debugged application.

Retain unchanged pages

Specifies that only changed, new, or updated pages are downloaded to flash memory, to save flash cycles.

Retain flash pages

Specify any flash pages that should not rewritten during download. Type the page numbers separated by commas, as page intervals, or a combination of these, like `0-6,8,12`.

Verify download

Verifies that the program data has been correctly transferred from the driver to the device. Choose between:

CRC-16	Verifies a target using on-chip page CRC16.
Read back memory	Verifies a target by reading back memory.

Lock flash memory

Protects your application resident in the flash memory of the device. Select the device family you are using and the parts of the flash memory you want to protect:

CC111x, CC243x, CC251x	Locks the flash memory of CC111x, CC243x, or CC251x devices.
Boot block lock	Locks the boot sector of CC111x, CC243x, or CC251x devices. Choose the lock bits from the drop-down menu. **Lock bits 000b** protects the whole flash memory, what the other options protect varies from device to device. The protected area is displayed under the drop-down menu.
CC253x, CC254x	Locks the flash memory of CC253x or CC254x devices. Type the flash pages you want to lock in the text box, separated by commas, as page intervals, or a combination of these, like `0-6,8,12`.

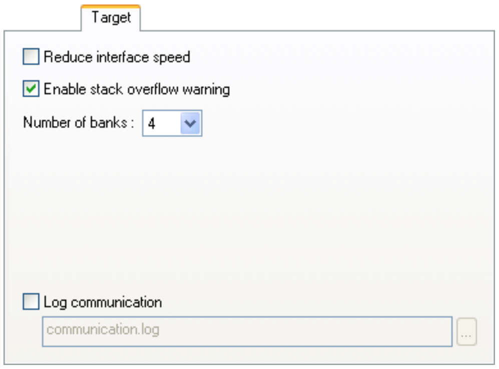
To remove these locks, you must select the **Erase flash** option.

Debug interface lock

Protects your application on the microcontroller from read and write accesses by locking the debug interface. To remove the lock, you must select the **Erase flash** option.

Target options for Texas Instruments

The Texas Instruments **Target** options control target-specific features of the Texas Instruments driver.



☐ Reduce interface speed

☑ Enable stack overflow warning

Number of banks : 4

☐ Log communication

communication.log

Figure 92: Texas Instruments target options

Reduce interface speed

Reduces the communication speed between your host computer and the target board. This can be very useful if you use a long cable or encounter communication problems or interference.

Enable stack overflow warning

Enables stack overflow warnings. This is not a runtime check, but is performed at the next stop, which means that it will not stop the execution if a stack overflow is encountered.

Number of banks

Specify the number of actual hardware memory banks on the device.

Log communication

Logs the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the communication protocol is required. This log file can be useful if you intend to contact IAR Systems support for assistance.

Reference information on C-SPY FS2 driver options

This section gives reference information on C-SPY FS2 driver options.

Download options for FS2

The FS2 **Download** options control the download.

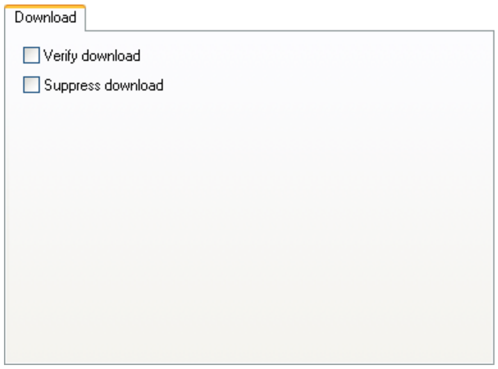


Figure 93: Download options for FS2

Verify download

Verifies that the downloaded code image can be read back from target memory with the correct contents.

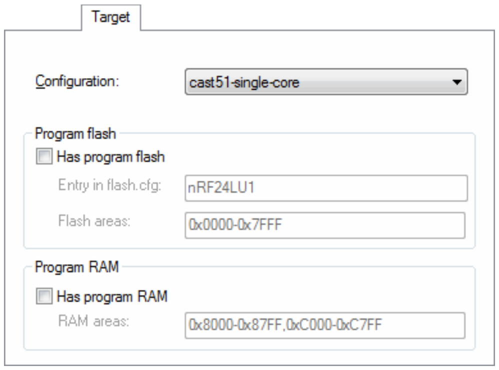
Suppress download

Disables the downloading of code, while preserving the present content of the flash memory. This command is useful if you want to debug an application that already resides in target memory. The implicit reset performed by C-SPY at startup is not disabled, though.

If this option is combined with the **Verify download** option, the debugger will read back the code image from non-volatile memory and verify that it is identical to the debugged application.

Target options for FS2

The FS2 **Target** options control target-specific features of the Infineon driver.



Configuration: cast51-single-core

Program flash
Has program flash
Entry in flash.cfg: nRF24LU1
Flash areas: 0x0000-0x7FFF

Program RAM
Has program RAM
RAM areas: 0x8000-0x87FF,0xC000-0xC7FF

Figure 94: FS2 target options

Configuration

Specify which core your device is. Choose between:

cast51-single-core

m8051ew-single-core

philips51-single-core

handshake51-single-core

Has program flash

Describes to the debugger how to program the flash memory of the device.

Entry in flash.cfg	The label for the entry in the `flash.cfg` file that describes how to program the flash memory of the device.
Flash areas	One or more memory ranges separated by commas, like this: `0x0000-0x1111,0x2222-0x3333`.

Has program RAM

Specify where the device has program code in RAM memory, if your device supports code in RAM. This means that software breakpoints will be used in this memory area.

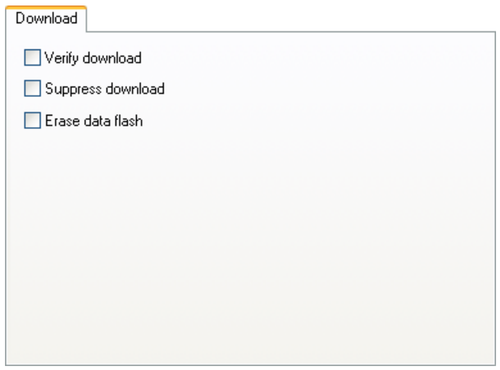
RAM areas	One or more memory ranges separated by commas, like this: `0x0000-0x1111,0x2222-0x3333`.

Reference information on C-SPY Infineon driver options

This section gives reference information on C-SPY Infineon driver options.

Download options for Infineon

The Infineon **Download** options control the download.



- [ ] Verify download
- [ ] Suppress download
- [ ] Erase data flash

Figure 95: Download options for Infineon

Verify download

Verifies that the downloaded code image can be read back from target memory with the correct contents.

Suppress download

Disables the downloading of code, while preserving the present content of the flash memory. This command is useful if you want to debug an application that already resides in target memory. The implicit reset performed by C-SPY at startup is not disabled, though.

If this option is combined with the **Verify download** option, the debugger will read back the code image from non-volatile memory and verify that it is identical to the debugged application.

Erase data flash

Erases the data flash area during download.

Target options for Infineon

The Infineon **Target** options control target-specific features of the Infineon driver.

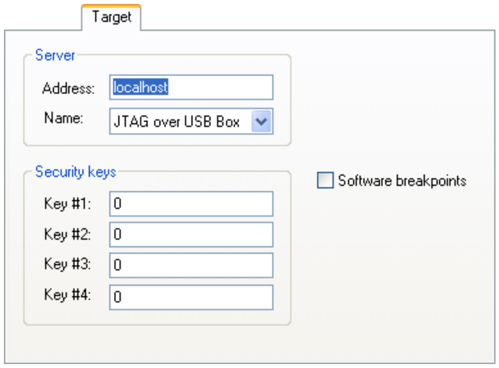


Figure 96: Infineon target options

Server

Specify the server on which the DAS server is running:

Address	The name or the IP address of the connected server. Specify `localhost` if the server is located on your host computer.
Name	Choose the DAS server to connect to.

Security keys

The DAS server has security keys which can be enabled and used to protect access to the device. If security keys are used, you must type the value for each key to connect to the server.

Software breakpoints

Enables software breakpoints, which increases the number of code breakpoints. If there are no hardware breakpoints available, software breakpoints will be used instead.

Note: Software breakpoints can only be used when the application is located in read/write memory. When you use this option, the breakpoints are implemented by a temporary substitution of the actual instruction. Before execution resumes, the original instruction will be restored. This generates some overhead.

Reference information on C-SPY Nordic Semiconductor driver options

This section gives reference information on C-SPY Nordic Semiconductor driver options.

Download options for Nordic Semiconductor

The Nordic Semiconductor **Download** options control the download.

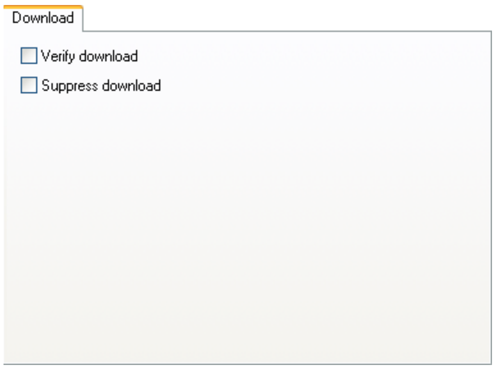


Figure 97: Download options for Nordic Semiconductor

Verify download

Verifies that the downloaded code image can be read back from target memory with the correct contents.

Suppress download

Disables the downloading of code, while preserving the present content of the flash memory. This command is useful if you want to debug an application that already resides in target memory. The implicit reset performed by C-SPY at startup is not disabled, though.

If this option is combined with the **Verify download** option, the debugger will read back the code image from non-volatile memory and verify that it is identical to the debugged application.

Reference information on C-SPY ROM-monitor driver options

This section gives reference information on C-SPY ROM-monitor driver options.

Download options for the ROM-monitor

The ROM-monitor **Download** options control the download.

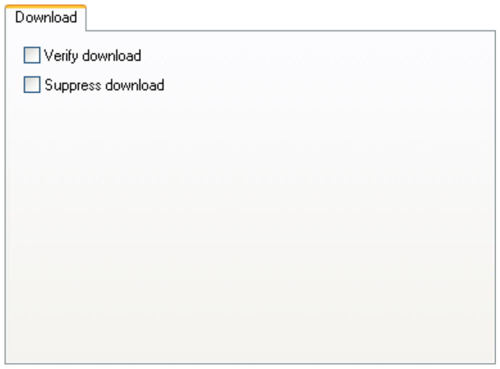


Figure 98: Download options for the ROM-monitor

Verify download

Verifies that the downloaded code image can be read back from target memory with the correct contents.

Suppress download

Disables the downloading of code, while preserving the present content of the flash memory. This command is useful if you want to debug an application that already resides in target memory. The implicit reset performed by C-SPY at startup is not disabled, though.

If this option is combined with the **Verify download** option, the debugger will read back the code image from non-volatile memory and verify that it is identical to the debugged application.

Serial Port options for the ROM-monitor

The ROM-monitor **Serial Port** options determine how the serial port should be used.

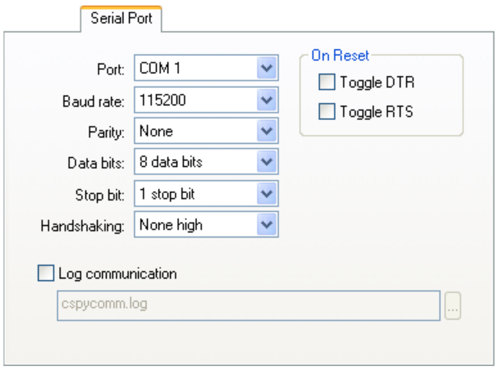


Figure 99: Serial port options for the ROM-monitor

Port

Selects one of the supported ports: `COM1`–`COM64`.

Baud rate

Selects one of these speeds: `2400`, `4800`, `9600`, `14400`, `19200`, `38400`, `57600`, or `115200` baud.

C-SPY always tries to connect at 9600 baud and then changes to the speed of the selected serial port when making the first contact with the evaluation board. If these options have not been specified, C-SPY will try using the COM1 port.

Parity

Selects the parity; `None`, `Even`, or `Odd`.

Data bits

Selects the number of data bits; only 8 data bits is allowed.

Stop bits

Selects the number of stop bits: `1` or `2`.

Handshaking

Selects the handshaking method; `None high`, `None low`, `RTSCTS`, or `XONXOFF`.

Toggle DTR

Toggles the DTR signal pin on the UART port when C-SPY resets the device.

Toggle RTS

Toggles the RTS signal pin on the UART port when C-SPY resets the device.

Log communication

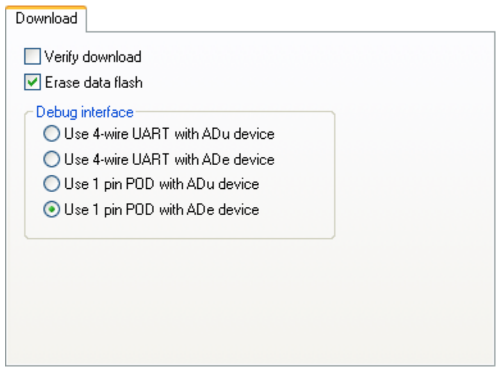
Logs the communication between C-SPY and the target system to the specified log file, which can be useful for troubleshooting purposes. The communication will be logged in the file `cspycomm.log` located in the current working directory. If required, use the browse button to locate a different file.

Reference information on C-SPY Analog Devices driver options

This section gives reference information on C-SPY Analog Devices driver options.

Download options for Analog Devices

The Analog Devices **Download** options control the download.



- [ ] Verify download
- [✓] Erase data flash

Debug interface
- ○ Use 4-wire UART with ADu device
- ○ Use 4-wire UART with ADe device
- ○ Use 1 pin POD with ADu device
- [✓ ● Use 1 pin POD with ADe device]

Figure 100: Download options for Analog Devices

Verify download

Verifies that the downloaded code image can be read back from target memory with the correct contents.

Erase data flash

Erases the data flash area during download.

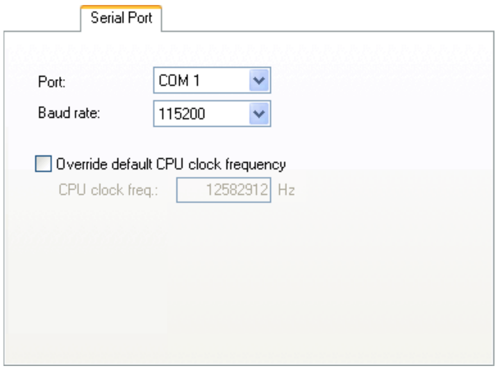
Debug interface

Specifies the communication method. Choose between:

- Use 4-wire UART with ADu device
- Use 4-wire UART with ADe device
- Use 1-pin POD with ADu device
- Use 1-pin POD with ADe device.

Serial Port options for Analog Devices

The Analog Devices **Serial Port** options determine how the serial port will be used.



Port: COM 1
Baud rate: 115200
☐ Override default CPU clock frequency
CPU clock freq.: 12582912 Hz

Figure 101: Serial port options for Analog Devices

Port

Selects one of the supported ports: `COM1`–`COM64`.

Baud rate

Selects one of the supported speeds: `2400`–`115200` baud. If a debug interface for an ADe device has been specified, the only available communication speed is `115200` baud.

C-SPY connects at 9600 baud and then changes to the speed of the selected port when making the first contact with the evaluation board.

Override default CPU clock frequency

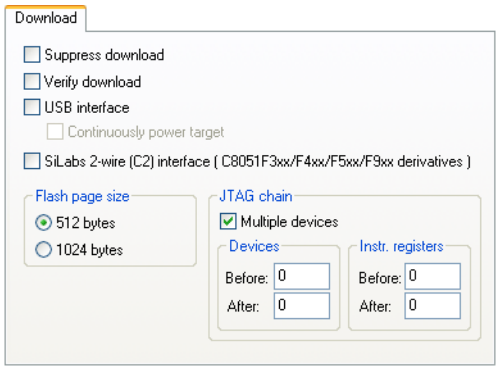
Specify the actual CPU clock frequency if you have modified the hardware in such a way that the clock frequency has changed.

Reference information on C-SPY Silabs driver options

This section gives reference information on C-SPY Silabs driver options.

Download options for Silabs

The Silabs **Download** options control the download.



[ ] Suppress download
[ ] Verify download
[ ] USB interface
[ ] Continuously power target
[ ] SiLabs 2-wire (C2) interface (C8051F3xx/F4xx/F5xx/F9xx derivatives)

Flash page size: [✓ 512 bytes] 1024 bytes

JTAG chain: [✓] Multiple devices

Devices — Before: 0, After: 0

Instr. registers — Before: 0, After: 0

Figure 102: Download options for Silabs

Suppress download

Disables the downloading of code, while preserving the present content of the flash memory. This command is useful if you want to debug an application that already resides in target memory. The implicit reset performed by C-SPY at startup is not disabled, though.

If this option is combined with the **Verify download** option, the debugger will read back the code image from non-volatile memory and verify that it is identical to the debugged application.

Verify download

Verifies that the downloaded code image can be read back from target memory with the correct contents.

USB interface

Specifies that you are using a USB debugger adapter.

Continuously power target

Provides power to the target hardware even after the debug session has been terminated.

Silabs 2-wire (C2) interface

The Silicon labs C8051F3*xx*/F4*xx*/F5*xx* devices use the Silabs 2-wire debugging interface (C2). You must select this option to connect to any of these devices.

Flash page size

Use this option to inform C-SPY of the size of the flash page of your device. Choose between 512 and 1024 bytes.

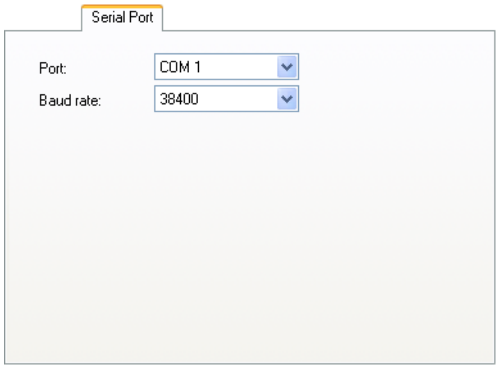
Multiple devices

Informs C-SPY that there is more than one device connected to the same JTAG interface. In this case, you must also specify:

Devices	The number of devices in the chain before and after the device to be debugged.
Instr. registers	The number of JTAG registers in the chain before and after the device to be debugged.

Serial Port options for Silabs

The Silabs **Serial Port** options determine how the serial port will be used.



Port: COM 1

Baud rate: 38400

Figure 103: Serial port options for Silabs

Port

Selects one of the supported ports: `COM1`–`COM64`. C-SPY connects with the selected serial port when making the first contact with the evaluation board. If you do not specify a port, C-SPY will try using the COM1 port.

Baud rate

Selects one of the supported speeds: `2400`–`115200` baud. C-SPY connects at 9600 baud and then changes to the speed of the selected port when making the first contact with the evaluation board.

Reference information on C-SPY third-party driver options

This section gives reference information on C-SPY third-party driver options.

Third-Party Driver

The **Third-Party Driver** options are used for loading any driver plugin provided by a third-party vendor. These drivers must be compatible with the C-SPY debugger driver specification.

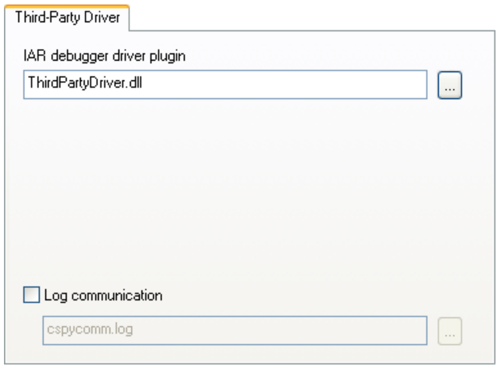


Figure 104: C-SPY Third-Party Driver options

You can set more options for the driver by using the **Project>Options>Debugger>Extra Options** page.

IAR debugger driver plugin

Specify the file path to the third-party driver plugin DLL file. A browse button is available for your convenience.

Log communication

Logs the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the interface is required. This option can be used if it is supported by the third-party driver.

Additional information on C-SPY drivers

This chapter gives additional reference information about some of the C-SPY® drivers, reference information not provided elsewhere in this documentation. More specifically, this means:

- Reference information on the C-SPY simulator
- Reference information on the C-SPY Texas Instruments driver
- Reference information on the C-SPY Infineon driver
- Reference information on the C-SPY Silabs driver
- Resolving problems.

Reference information on the C-SPY simulator

This section gives additional reference information about the C-SPY simulator, reference information not provided elsewhere in this documentation.

More specifically, this means the *Simulator menu*, page 286.

Simulator menu

When you use the simulator driver, the **Simulator** menu is added to the menu bar.

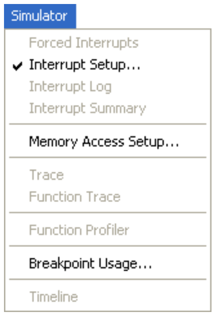


Figure 105: Simulator menu

These commands are available on the menu:

Forced Interrupts	Opens a window from where you can instantly trigger an interrupt, see *Forced Interrupt window*, page 172.
Interrupt Setup	Displays a dialog box where you can configure C-SPY interrupt simulation, see *Interrupt Setup dialog box*, page 169.
Interrupt Log	Opens a window which displays the status of all defined interrupts, see *Interrupt Log window*, page 173.
Interrupt Summary	Opens a window which displays a summary of the status of all defined interrupts, see *Interrupt Log Summary window*, page 175.
Memory Access Setup	Displays a dialog box to simulate memory access checking by specifying memory areas with different access types, see *Memory Access Setup dialog box*, page 129.
Trace	Opens a window which displays the collected trace data, see *Trace window*, page 137.
Function Trace	Opens a window which displays the trace data for function calls and function returns, see *Function Trace window*, page 138.
Function Profiler	Opens a window which shows timing information for the functions, see *Function Profiler window*, page 153.

Breakpoint Usage	Displays a dialog box which lists all active breakpoints, see *Breakpoint Usage dialog box*, page 102.
Timeline	Opens a window which shows trace data for interrupt logs and for the call stack, see *Timeline window*, page 139.

Reference information on the C-SPY Texas Instruments driver

This section gives additional reference information on the C-SPY Texas Instruments driver, reference information not provided elsewhere in this documentation.

More specifically, this means the *Texas Instruments Emulator menu*, page 287.

Texas Instruments Emulator menu

When you are using the C-SPY Texas Instruments driver, the **Texas Instruments Emulator** menu is added to the menu bar.

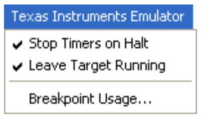


Figure 106: The Texas Instruments Emulator menu

These commands are available on the menu:

Stop Timers on Halt	Stops the timers when the execution is stopped.
Leave Target Running	Leaves the application running on the target hardware after the debug session has been terminated.
Breakpoint Usage	Displays a dialog box which lists all active breakpoints, see *Breakpoint Usage dialog box*, page 102.

Reference information on the C-SPY Infineon driver

This section gives additional reference information on the C-SPY Infineon driver, reference information not provided elsewhere in this documentation.

More specifically, this means:

- *Infineon Emulator menu*, page 288
- *Server Selection dialog box*, page 288.

Infineon Emulator menu

When you are using the C-SPY Infineon driver, the **Infineon Emulator** menu is added to the menu bar.

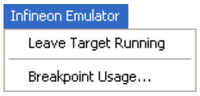


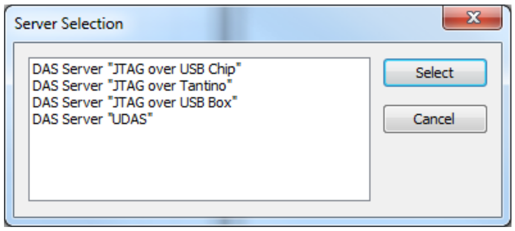
Figure 107: The Infineon Emulator menu

These commands are available on the menu:

Leave Target Running	Leaves the application running on the target hardware after the debug session has been terminated.
Breakpoint Usage	Displays a dialog box which lists all active breakpoints, see *Breakpoint Usage dialog box*, page 102.

Server Selection dialog box

The **Server Selection** dialog box is displayed if a debug session starts without a DAS server name specified on the **Project>Options>Infineon>Target** page.



DAS Server "JTAG over USB Chip"
DAS Server "JTAG over Tantino"
DAS Server "JTAG over USB Box"
DAS Server "UDAS"

Figure 108: Server Selection dialog box

Use this dialog box to specify the type of DAS server connection.

Reference information on the C-SPY Silabs driver

This section gives additional reference information on the C-SPY Silabs driver, reference information not provided elsewhere in this documentation.

More specifically, this means:

- *Silabs Emulator menu*, page 289
- *USB Device Selection dialog box*, page 289.

Silabs Emulator menu

When you are using the C-SPY Silabs driver, the **Silabs Emulator** menu is added to the menu bar.

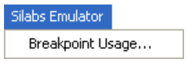


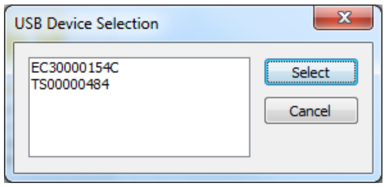
Figure 109: The Silabs Emulator menu

This command is available on the menu:

Breakpoint Usage	Displays a dialog box which lists all active breakpoints, see *Breakpoint Usage dialog box*, page 102.

USB Device Selection dialog box

The **USB Device Selection** dialog box is displayed if a debug session starts with more than one Silabs debug adapter connected to the host computer.



EC30000154C
TS00000484

Figure 110: USB Device Selection dialog box

Use this dialog box to select the device you want to use.

Resolving problems

Debugging using the C-SPY hardware debugger systems requires interaction between many systems, independent from each other. For this reason, setting up this debug system can be a complex task. If something goes wrong, it might at first be difficult to locate the cause of the problem.

This section includes suggestions for resolving the most common problems that can occur when debugging with the C-SPY hardware debugger systems.

For problems concerning the operation of the evaluation board, refer to the documentation supplied with it, or contact your hardware distributor.

WRITE FAILURE DURING LOAD

There are several possible reasons for write failure during load. The most common is that your application has been incorrectly linked:

- Check that you are using the correct linker configuration file.
- Check the contents of your linker configuration file and make sure that your application has not been linked to the wrong address

If you are using the IAR Embedded Workbench IDE, the linker configuration file is automatically selected based on your choice of device.

To choose a device:

1. Choose **Project>Options.**
2. Select the **General Options** category.
3. Click the **Target** tab.
4. Choose the appropriate device from the **Device** drop-down list.

To override the default linker configuration file:

1. Choose **Project>Options.**
2. Select the **Linker** category.
3. Click the **Config** tab.
4. Choose the appropriate linker configuration file in the **Linker configuration file** area.

NO CONTACT WITH THE TARGET HARDWARE

There are several possible reasons for C-SPY to fail to establish contact with the target hardware. Do this:

- Unplug and plug in the power supply to all debugger hardware
- Check the communication devices on your host computer
- Verify that the cable is properly plugged in and not damaged or of the wrong type
- Make sure that the evaluation board is supplied with sufficient power
- Check that the correct options for communication have been specified in the IAR Embedded Workbench IDE.

Examine the linker configuration file to make sure that the application has not been linked to the wrong address.

MONITOR WORKS, BUT APPLICATION WILL NOT RUN

Your application was probably linked to some illegal code area (like the interrupt table). Examine the linker command file and verify that the start addresses of `CODE` and `DATA` segments are correct.

Make sure you disable the watchdog timer if it is not used. Typically this should be done in the `__low_level_init` routine. Otherwise the application program will restart, which would lead to unexpected behavior.

NO CONTACT WITH THE MONITOR

There are several possible reasons if C-SPY fails to establish contact with the ROM-monitor firmware.

- The communication speed between C-SPY and the ROM-monitor might make the connection unreliable. Try a lower communication speed.
- A protocol error might have occurred. Try resetting your evaluation board and restart C-SPY.
- Check that the correct options for serial communication have been specified in the IAR Embedded Workbench IDE. See the corresponding sections for the appropriate driver.
- Verify that the serial cable is properly plugged in and not damaged or of the wrong type.

Target-adapting the ROM-monitor

This chapter describes how you can easily adapt the generic ROM-monitor provided with IAR Embedded Workbench® to suit a device that does not have an existing debug solution supported by IAR Systems.

This chapter also describes ROM-monitor functionality in detail.

Building your own ROM-monitor

There are a large number of 8051 devices on the market. The variety makes it difficult for one ROM-monitor firmware to support them all. Therefore, a full ROM-monitor project is included in IAR Embedded Workbench that you can use to customize the ROM-monitor for a specific target.

Building your own ROM-monitor in four easy steps:

1. Set up your ROM-monitor project.
2. Adapt the source files.
3. Build and download your ROM-monitor.
4. Debug the ROM-monitor.

Note: To download your ROM-monitor to the target board, refer to the chip vendor websites for information about suitable tools. When you have successfully downloaded the ROM-monitor, you can use it to debug your application via C-SPY. For information about required C-SPY options, see *Debugger options*, page 261.

SET UP YOUR ROM-MONITOR PROJECT

Choose **Project>Create new project** and select the ready-made project **ROM-monitor**. Click **OK** and choose a project name and a destination folder in the **Save As** dialog box.

This project will contain generic monitor files and files that must be edited. The generic files are located in `src\rom\common_src` and are not copied, while the files that must be edited are copied automatically to the project directory.

ADAPT THE SOURCE FILES

The ROM-monitor project contains many source and header files, but only a few need to be adapted to suit your target system:

`iotarget.h`	Includes the target-specific include file.
`chip_layout.h`	Holds target-specific definitions for special registers, etc.
`chip_layout.xcl`	Template linker command file for the ROM-monitor.
`uart_init.c`	UART initialization and baud rate function support.
`code_access.c`	Read and write functions for code memory.
`low_level_init.c`	Basic initialization code, executed early during system startup.
`high_level_init.c`	Additional initialization code called from the `main` function.

Setting up the chip_layout.h file

This is one of the most important files in the project, as it sets up the conditions for making the ROM-monitor work. Some of the sections of this file are highlighted here. There is more information available in the header file.

- Software and hardware breakpoints

  The ROM-monitor uses the `LCALL` instruction as a generic software breakpoint. There is also support for target-specific breakpoint instructions, such as `0xA5`. Set `SW_BP_TYPE` to either `BP_OF_LCALL_TYPE` or `BP_OF_A5_TYPE`. If no software breakpoints are to be used, set `SW_BP_TYPE` to `NO_SW_BP`.

  If required and if supported by the target board, the ROM-monitor can also handle code and data hardware breakpoints. In this case, you must set the symbols `CODE_HW_BP` and `DATA_HW_BP` to the number of breakpoints they support.

- Application bus width

  The application bus width controls the addressable memory area. It is either 16 or 24 bits depending on the target or the location of the application on the target.

- Remapped IData memory

  By default, the ROM-monitor will use `0x00-0x7F` in IData memory as working memory. When the ROM-monitor is running, application data located in this memory area will be stored in PData(XData)/IData memory. The symbol `MON_REMAP_IDATA_TO_MEM` controls which memory segment that is used for this copy.

- Special SFR registers

  Some SFR registers are needed by the ROM-monitor and would be overwritten if shared with your application. To avoid this, these registers are stored/restored by the monitor. For this purpose, an SFR information `struct` is used, where SFRs that need special care can be added.
- Flash memory information

  The flash page section defines the flash page properties such as page size and total memory size. When writing to memory, C-SPY will send the bytes to write at full speed via the UART. Depending on how long it takes to update the memory, a communication delay might have to be introduced to ensure that there is enough time to finish writing data to the flash memory. For this purpose, `FLASH_WRITE_DELAY` can be defined to insert a delay between each byte.

Setting up the serial communication—uart_init.c

You need to configure and initialize a UART to make the serial communication work between C-SPY and the ROM-monitor. One way to make sure that your UART setup works as intended is to create a small application that performs the initialization and then echoes any characters that the UART receives back to the transmitting device.

By doing this, you can debug your UART setup using a standard host computer and a terminal program (for example, Hyper Terminal) before you include it in your ROM-monitor. The file `uart_init.c` is available in your ROM-monitor project as a starting point:

- In the function `uart_init()`, set up the UART.
- By default, for each new debug session the communication starts at 9600 baud. However, it is recommended to include support for 9600, 38400, and 57600 baud right from the start. You can achieve this by modifying the function `set_baudrate()`.
- Use a standard hyper terminal on your host computer to verify the UART setup.
- Once the application is running, you can include your `uart_init.c` file in your ROM-monitor project.

Make sure also that the baud rates supported by the device are updated accordingly in the file `chip_layout.h`.

Note: To understand how to configure the UART on your device, refer to its documentation.

Setting up for code memory accesses—code_access.c

Most memory access methods are the same for all 8051-compatible devices, except for the method for writing code. Writing to code memory is divided into three functions, all defined in the `code_access.c` file:

`prepare_download()`	This function is called before download and should prepare the target system for code download. If the code is to be downloaded into flash memory, this function should erase the flash memory.
`erase_flash_page()`	This function is called when rewriting a page. For example, when writing a software breakpoint located in flash memory. This function is usually called by the `prepare_download` function when erasing memory before download.
`byte_write_code()`	This function writes one byte to code memory. Any overhead, such as reading back, erasing, and writing data is handled automatically by the C-SPY driver.

Setting up target-specific details—low_level_init.c, high_level_init.c

In the source files `low_level_init.c` and `high_level_init.c`, you can set up the target-specific details to be performed before system startup, such as enabling memory, initializing clocks, etc. The `low_level_init` function is executed after a hard reset from the `cstartup.s51` file before initializing variables. The `high_level_init` function is called each time the ROM-monitor is entered.

To read more about the system startup code, see *IAR C/C++ Compiler Reference Guide for 8051*.

DEBUG THE ROM-MONITOR

Debugging a ROM-monitor can sometimes be difficult, because of the number of subsystems involved (host computer, host debugger, ROM-monitor firmware, hardware, and user application). Therefore, you are strongly recommended to use the available hardware resources on the target system as a way to provide feedback during the debugging process.

For example, if there is an LCD on the board you can use it to display status messages, or if a second UART is available the same can be done to a terminal program running on a host computer. If there is a LED available, it can be used as simple `printf` functionality.

Note: Because neither interrupts nor buffers for incoming data are used in the communication between C-SPY and the ROM-monitor, the overhead introduced by making peripheral units log events during debugging can cause the ROM-monitor to lose its connection with C-SPY.

Divide the work needed to customize the ROM-monitor into small steps, and verify that each step works as intended before going on to the next step.

Debugging using the C-SPY simulator

You can also use the C-SPY simulator for debugging your ROM-monitor. You can make C-SPY simulate the UART and read the communication data from a file. There is a macro file `serialData.mac` included with the product that will set up the necessary interrupts and feed the simulated UART with data. There is also a UART data file `SerialData.txt` included, which contains some basic driver commands. When the macro is used, the data from the file will be sent as input to the monitor, triggering the different actions/calls. The files are located in `8051\src\rom`.

Note: In the macro file `serialData.mac`, the registers `SBUF` and `SCON` are used by default. Depending on the device you are using, you might need to modify the file by replacing these registers to make the macro simulate your target device.

BUILD AND DOWNLOAD YOUR ROM-MONITOR

After you have adapted the source code files, you can build your project. Note that it is a good idea to divide this work into small steps by verifying only one part of the ROM-monitor at a time. Repeat the work until you have successfully managed to build and download the complete ROM-monitor.

Make sure that you pay attention to the following issues when you build your ROM-monitor project:

- It can sometimes be difficult to write to target memory during download; typically, the function `byte_write_code` (writing to code) can cause problems. Therefore, you are recommended to initially select **Suppress download**. This way you can start to verify that the ROM-monitor's basic functions work, such as reading memory and registers, or single stepping. Once this is done, you can deselect **Suppress download** and download a test application.
- The ROM-monitor only uses 1 DPTR. Choose **Project>Options>General Options>Data Pointer** and select 1 from the **Number of DPTRs** drop-down list.
- When you link your application program, you must verify that it is placed in memory that does not overlap your ROM-monitor. This means that you must also adapt the linker command file that you use for your application program accordingly. In other words, make sure to exclude the memory reserved for the ROM-monitor from the linker command file used when building your application program. See *Resources used by the ROM-monitor*, page 308.

Some error messages might be generated; in that case each error is located close to a function that needs to be corrected. You can easily move from error to error with the F4 key.

When you have successfully built your ROM-monitor project, you can download the generated ROM-monitor using an appropriate download tool. Create a simple test project. In the **Options** dialog box, choose **ROM-monitor** as the driver and set the baud rate.

The ROM-monitor in detail

This illustration shows the program flow (thick arrow), as well as actions and events (thin arrow) that can occur:

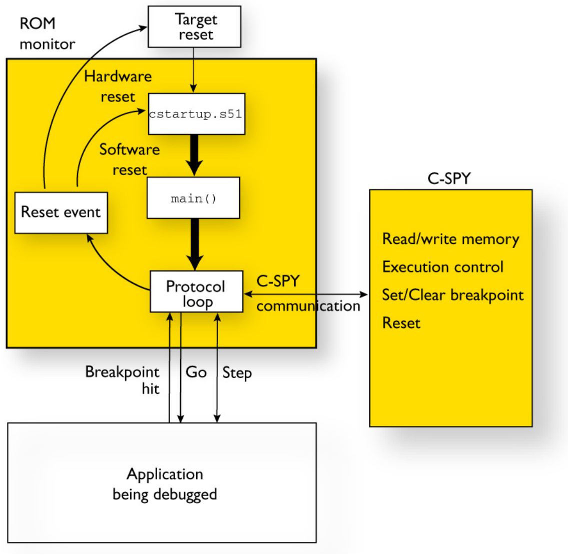


Figure 111: ROM-monitor program execution overview

The ROM-monitor will be described by investigating source code implementation details for:

- Early initializations
- The protocol loop
- Leaving the ROM-monitor
- Entering the ROM-monitor

- Resources used by the ROM-monitor.

Note: Functions whose names end in `_R10` are ROM-monitor library functions, delivered with IAR Embedded Workbench.

EARLY INITIALIZATIONS

Before the main function

After a reset, neither variables nor the stack have any known values, which means they first have to be initialized. The ROM-monitor will execute, beginning with the `cstartup.s51` file which initializes the stack pointer, going on to call `__low_level_init` to perform any low-level initializations, and—depending on its return value—continuing with the initialization of variables.

In the main function

When the `main` function has been entered, the required subsystems can be initialized. This is illustrated by the following process:

1 The `main` function is structured in the following way:

```
void main( void )
{
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
* Enter a known runtime model.
*/
high_level_init();
```

In `high_level_init` you should implement all target-specific hardware initializations that are not time critical.

2 If you have any LEDs on your target system that you want to use for debugging the ROM-monitor, there is support for this prepared. In this case, the LED needs to be initialized:

```
#ifdef DEBUG_METHOD_BLINK
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
* Initialize the LED used for debugging.
*/
led_init();
```

To use this debug method, you must define the `DEBUG_METHOD_BLINK` symbol, for example in `debug_method.h`. In the same way, other visual and audible devices available can be used.

3 For communication between the ROM-monitor and the C-SPY debugger, you must initialize the UART for serial communication:

```
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
* Initialize the low level communication.
*/
uart_init();
```

Modify the `uart_init` function according to the needs of your target device.

The `uart_init` function will call the function `set_baudrate` to set the communication speed, which initially is set to 9600 baud. If required, you can add support for additional rates, for example 38400 and 57600. In this case, you must also modify the function `set_baudrate`.

Make sure the communication speeds supported by `set_baudrate` are also defined in the file `chip_layout.h`.

4 The following source code lines initialize the flags and variables used by the ROM-monitor protocol loop:

```
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
* Initialize the ROM-monitor protocol.
*/
communication_init();
```

You should not need to modify this function.

5 To notify C-SPY that a hardware reset has occurred and that the ROM-monitor has been reinitialized, the reset command `0xD8` is sent to C-SPY. This must be performed during the early initialization:

```
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
* Let the host debugger know that there is a reset.
* You will see the character 'Ø' (the hexadecimal value
* 0xD8) in a terminal that is connected to the UART.
*/
send_hw_reset();
```

You should not need to modify this function.

6 The ROM-monitor requires some space in the IData memory area `0x00-0x7F`. Any application data in this area will be temporarily stored in an array named `remapped_idata` while the ROM-monitor is executing. A small part of the IData memory, the special register area (see Figure 113, *Data memory used by the ROM-monitor*), is reserved for the ROM-monitor.

During the early initialization, registers are set to their initial values by the following part of the source code:

```
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
* Application reset values for the critical registers
*/

spec_reg_A      = 0xCE;
spec_reg_PSW    = 0xCE;
spec_reg_R0     = 0xCE;
spec_reg_R1     = 0xCE;

#ifdef CHIP_PBANK_SFR
  spec_reg_PBANK  = 0xCE;

  CHIP_PBANK_SFR = (MON_REMAPPED_IDATA_ADDR >> 8);
#endif /* CHIP_PBANK */

#ifdef CHIP_PBANK_EXT
  spec_reg_PDATA_EXT = 0xCE;

  CHIP_PBANK_EXT = (MON_REMAPPED_IDATA_ADDR >> 16);
#endif /* CHIP_PBANK_EXT */

#ifdef CHIP_XDATA_EN_SFR
  spec_reg_XDATA_EN = 0xCE;
#endif /* CHIP_XDATA_EN_REG */
```

There are additional fixed return codes that are used:

- `0xCC` is returned when reading a protected SFR
- `0xCD` is returned when the memory zone cannot be verified
- `0xCE` is the uninitialized value for internal ROM-monitor registers.

7 To minimize stack consumption, entering the protocol loop is not performed using a standard function call. Instead a jump instruction is used. In addition, the return address is popped from the stack as the ROM-monitor never will return.

```
* Go to communication() step
* -------------------------
* Free the stack and LJMP to communication()
* because we will never exit
*/
  asm("POP A");
  asm("POP A");
#if (__CORE__ == __CORE_EXTENDED1__)
  asm("POP A");
#endif /* __CORE_EXTENDED1__ */
```

```
  asm("LJMP communication");

#endif /* STEP2_BAUDRATE */
```

THE PROTOCOL LOOP

After all initializations have been performed, the ROM-monitor is ready to be used for debugging. The function `communication` in the file `communication.c` holds the main protocol loop. In this loop, the `Rx` flag is continuously polled to monitor whether a byte has been received on the UART. The `Tx` flag is used for two different purposes. In the main loop, the `Tx` flag is continuously polled to check whether data should be transmitted. While the application program is executing, the `Tx` flag is used for catching any requests for execution stops. Figure 112, *The protocol loop* illustrates the protocol loop.

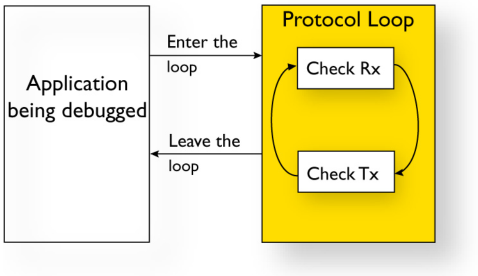


Figure 112: The protocol loop

Enter the loop

1 Before the execution enters the actual loop, the `Tx` flag is checked:

```
  if(CHIP_Tx_bit)
  {
    CHIP_Tx_bit   = 0;

    modeRegister |= Mode_HALTED;

    /* Tell host driver that we have stopped */
    sendByte_R10( Ind_BREAK_REACHED );
  }
```

This means that the application execution has stopped, either by:

- A stop execution command

- Single step
- Breakpoint hit.

In the loop

2 The protocol loop will loop while the halted flag `MODE_HALTED` is set, which means until the Go or any step commands are ordered by C-SPY:

```
while( modeRegister & Mode_HALTED) /* Main loop */
{...
```

Check Rx

3 The `Rx` flag is continuously checked and if it is set, a character has been received from C-SPY. The data is moved to the `receiveBuffer` and the `Rx` flag is cleared.

```
  /* A byte was received */
  if( CHIP_Rx_bit )
  {
    receiveBuffer = CHIP_SBUF;

    CHIP_Rx_bit = 0;

    /* decode the received byte */
    received_byte = byteReceived_R10();

    /* if receivedByte != 0x0 then send the byte back */
    if( received_byte )
      sendByte_R10( received_byte );
  }
```

The `byteRecieved_R10()` function call will decode the received byte and update the ROM-monitor state accordingly. A packet of data is constructed by a start byte, payload, checksum, and a stop byte. If the received byte is part of a larger packet of data, the data is stored and the loop is re-entered.

The `byteRecieved_R10` function might sometimes also return a value to be sent back to C-SPY. In most cases, this return value is an acknowledgment for the received data.

Check Tx

4 After the Rx flag has been checked, the Tx flag is checked for transmitted data. When sending data packets to C-SPY, each byte is sent as soon as the UART is free. A protocol library function holds the next byte to be sent, and the getNextByte_R10 function is used for getting it.

```
/* A byte has been transmitted */
if( CHIP_Tx_bit )
{
  CHIP_Tx_bit = 0;

  if( modeRegister & Mode_MEMORY_READ )
  {
    /* load TXData with next character */
    getNextByte_R10();

    /* check if this is the last byte to send */
    if( TXData == Resp_END )
      modeRegister &= ~Mode_MEMORY_READ;

    /* send byte   */
    sendByte_R10(TXData);
  }
}
```

Leave the loop

5 When the Mode_HALTED flag is cleared, the ROM-monitor execution leaves the main loop to initiate a step or Go of the application program being debugged. Before application data and registers are restored, and before the execution is handed over to the application, the Rx and Tx flags are set/cleared.

```
  if( flagRegister & Flag_LETS_STEP )
  {
    CHIP_Rx_bit = 0;
    CHIP_Tx_bit = 1;
  }
  else /* GO is implicit */
  {
    CHIP_Rx_bit = 0;
    CHIP_Tx_bit = 0;
  }

  c_runtime_leave();

}
```

The ROM-monitor uses the `Tx` interrupt mechanism by setting the `Tx` flag to execute one single instruction of the application (single step). Otherwise (Go), both `Rx` and `Tx` are cleared before continuing with the leave sequence.

LEAVING THE ROM-MONITOR

When the execution leaves the ROM-monitor, shared resources such as registers and data memory must be restored. This is done by the functions `c_runtime_leave` (written in C) and `monitor_leave` (written in assembler):

```
  /* copy user R0 and R1 to special register area */
  /* from user register bank                      */
  spec_reg_R0 = remapped_idata[PSW & 0x18];
  spec_reg_R1 = remapped_idata[(PSW & 0x18) + 1];

  /* restore DPTR and SP */
#ifdef APP_EXTENDED_DPTR
  CHIP_DPX0 = app_reg[SPEC_SFR_DPX];
#endif /* APP_EXTENDED_DPTR */

  DPH       = app_reg[SPEC_SFR_DPH];
  DPL       = app_reg[SPEC_SFR_DPL];
  SP        = app_reg[SPEC_SFR_SP];

  /* Leave C-runtime level */
  asm("LJMP   monitor_leave");
```

Before jumping to the `monitor_leave` function, the registers `R0`, `R1`, and `DPTR`, as well as the stack pointer of the application are restored.

The main reason for splitting the leave sequence into one C and one assembler routine is to keep as much source code as possible in C, which benefits portability and makes the code less device-specific.

The `monitor_leave` routine differs depending on where the IData memory of the debugged application is stored during ROM-monitor execution. This is controlled by the symbol `MON_REMAP_IDATA_TO_MEM` which is defined in `chip_layout.h`. In the following example, the source code for PData storage is used:

```
monitor_leave:

   MOV      R0, #MON_IDATA_END_ADDR      ; destination source end
address -8 since we dont copy 0x78-0x7F
   MOV      R1, #BYTE1(remapped_idata) + MON_IDATA_END_ADDR ;
source end address

#ifdef CHIP_PBANK_SFR
   MOV      CHIP_PBANK_SFR,#BYTE2(remapped_idata)    ; PData bank
```

```
#else
  #pragma message="CHIP_PBANK_SFR not set (defined in
chip_config.h)"
#endif

#ifdef CHIP_PBANK_EXT_SFR
   MOV      CHIP_PBANK_EXT_SFR, #BYTE3(remapped_idata)    ; high
bank
#endif

loop:
   MOVX     A, @R1
   XCH      A, @R0
   MOVX     @R1, A
   DEC      R1
   DEC      R0
   CJNE     R0, #0x01, loop

   ;; restore register values
#ifdef CHIP_XDATA_EN_SFR
   MOV      CHIP_XDATA_EN_SFR,DATA_XDATA_EN
#endif

#ifdef CHIP_PBANK_SFR
   MOV      CHIP_PBANK_SFR,DATA_PBANK
#endif ;  CHIP_PBANK_SFR

#ifdef CHIP_PBANK_EXT_SFR
   MOV      CHIP_PBANK_EXT_SFR, DATA_PBANK_EXT
#endif ;  CHIP_PBANK_EXT_SFR

#ifdef CHIP_PBANK_SFR
   MOV      CHIP_PBANK_SFR, DATA_PBANK
#endif

   MOV      R1,DATA_R1
   MOV      R0,DATA_R0
   MOV      PSW,DATA_PSW
   MOV      A,DATA_A

   SETB     CHIP_EA_reab
   RETI
```

This assembler routine can be divided into two parts—copying data and restoring registers.

- Copying

  `R0` is loaded with the destination address in IData (`0x7F`). Then, `R1` is set to the used offset within PData. The PData bank used is also initiated. The loop will then exchange data between the two memory areas and thus restore application IData memory.

- Restoring

  Because the debugged application might also use the PData bank, its control register also needs to be restored, as well as the XData enable register if used. `R0`, `R1`, `PSW`, and `A` are restored. Interrupts are enabled. Finally, executing `RETI` will make the application `PC` be loaded from the stack, and thus the application program starts executing.

ENTERING THE ROM-MONITOR

While the application is executing, the ROM-monitor can be entered by one of the following possible reasons:

- Stop execution
- Single step
- Breakpoint hit.

Stop execution	When you click the **Stop** icon in the C-SPY toolbar or choose **Debug>Stop Debugging**, a stop command is sent to the ROM-monitor. The UART will then initiate an interrupt and thus break the application execution, and then jump back to the ROM-monitor. Then, the ROM-monitor will be entered again via the UART interrupt vector call to `__monitor_enter`.
Single step	Because single step execution is initialized by setting the `Tx` interrupt flag before leaving the ROM-monitor, this will trigger the interrupt after executing one instruction. Then, the ROM-monitor will be entered again via the UART interrupt vector call to `__monitor_enter`.

Breakpoint hit	Depending on which breakpoints that are supported, hitting a breakpoint might need some extra handling. Assuming that software breakpoints are supported, the original instruction is replaced by `CALL __monitor_enter`. In this case, a breakpoint hit is more like a change of program flow rather than an interrupt of program execution. Nevertheless, `__monitor_enter` will be executed and from there the ROM-monitor main protocol loop is entered.

In all these three cases, the status of the application execution is stored and the `PC` will be located on the application stack.

For any custom breakpoint implementations, you must make sure that the `PC` is stored on the stack in the same way as for interrupts and function calls, and to call the assembler routine `__monitor_enter` with all registers unchanged.

Note: Because execution of both single step and stop rely on triggering the UART interrupt, these stop mechanisms will be disabled if the debugged application program disables global interrupt handling.

RESOURCES USED BY THE ROM-MONITOR

The ROM-monitor executes in the same environment as your application program, which means it is vital that the two do not use the same resources concurrently. The ROM-monitor uses the following resources:

- The UART interrupt
- Approximately 128 bytes of IData/XData/PData memory
- Approximately 3.5 Kbytes of code memory
- Approximately 7 bytes of Data memory for the special register area.

When the ROM-monitor is executing, it uses IData memory for variables and the stack, and it stores application data.

The following illustration shows the ROM-monitor working memory:

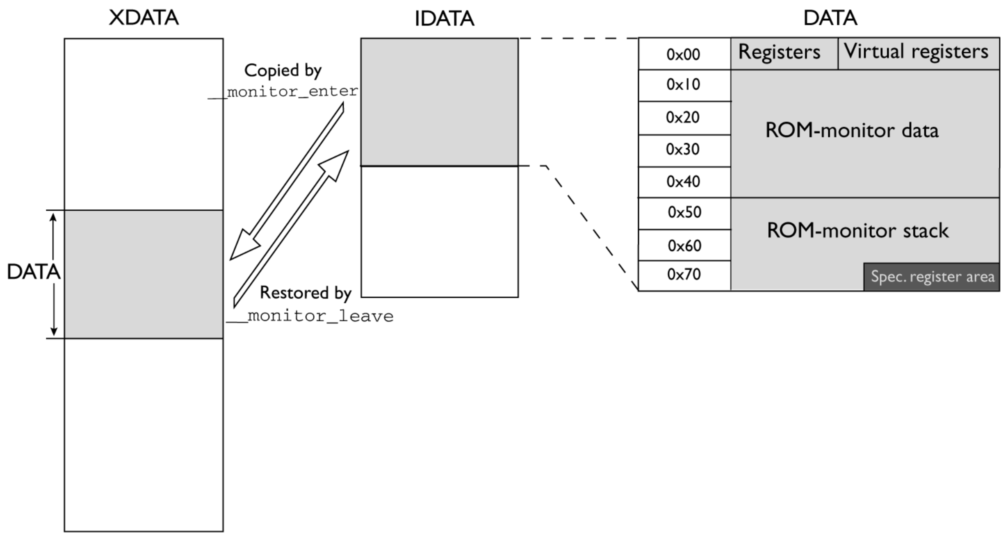


Figure 113: Data memory used by the ROM-monitor

All memory except for the range `0x78–0x7F` is restored when switching between the ROM-monitor and the application program.