IAR C-SPY® Nexus Debugger Systems

User Guide

for National Semiconductor's
CR16C Microcontroller Family

CSCR16CHW-2

IAR SYSTEMS

EDITION NOTICE

Second edition: February 2011

Part number: CSCR16CHW-2

This guide applies to version 3.x of IAR Embedded Workbench® for National Semiconductor’s CR16C microprocessor family.

Internal reference: Too6.0, IMAE.

Contents

Preface

Welcome to the *IAR C-SPY® Nexus Debugger Systems for CR16C User Guide*. The purpose of this guide is to provide you with detailed reference information that can help you use the features in the IAR C-SPY® Hardware Debugger Systems for CR16C.

Who should read this guide

You should read this guide if you want to get the most out of the features in the C-SPY hardware debugger systems. 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 CR16C microprocessor (refer to the chip manufacturer's documentation)
- The operating system of your host machine.

This guide also assumes that you already have working knowledge of the target system you are using, as well as some working knowledge of the IAR C-SPY Debugger. For a quick introduction to the IAR C-SPY Debugger, see the tutorials available in the *IAR Embedded Workbench® IDE User Guide*.

How to use this guide

This guide describes the C-SPY interface to the target system you are using; this guide does not describe the general features available in the IAR C-SPY debugger or the hardware target system. To take full advantage of the whole debugger system, you must read this guide in combination with:

- The *IAR Embedded Workbench® IDE User Guide* which describes the general features available in the C-SPY debugger
- The documentation supplied with the target system you are using.

Note that additional features might have been added to the software after the *IAR C-SPY® Nexus Debugger Systems for CR16C User Guide* was printed. The release notes `cscr16cx.htm`, and `cssc14x.htm` contain the latest information.

What this guide contains

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

- *Introduction to C-SPY® Nexus debugger systems* introduces you to the C-SPY Nexus debugger. The chapter briefly shows the difference in functionality provided by the different debugger systems.
- *C-SPY® Nexus debugger-specific debugging* describes the additional options, menus, and features specific to the C-SPY Nexus debugger.
- *Using flash loaders* describes the flash loader, what it is and how to use it.

Other documentation

The complete set of IAR development tools for the CR16C microprocessor are described in a series of guides. These guides can be found in the `cr16c\doc` directory or reached from the **Help** menu.

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

Recommended web sites:

- The National Semiconductor web site, **www.national.com**, contains information and news about the CR16C microprocessors.
- The SiTel Semiconductor web site, **www.sitelsemi.com**, contains information about the SC14 co-processor.
- The IAR Systems web site, **www.iar.com**, holds application notes and other product information.

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 `cr16c\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench 6.`*`n`*`\cr16c\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`	Alternatives in a command.
`{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 CR16C	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for CR16C	the IDE
IAR C-SPY® Debugger for CR16C	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for CR16C	the compiler

Table 2: Naming conventions used in this guide

Brand name	Generic term
IAR Assembler™ for CR16C	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)

Introduction to C-SPY® Nexus debugger systems

This chapter introduces you to the C-SPY Nexus debugger systems and to how they differ from the C-SPY Simulator.

The C-SPY hardware debugger systems

The CR16C microcontrollers have on-chip debug support for the Nexus 5001 standard for debugger interfaces. Because the hardware debugger kernel is built into the microprocessor, no ordinary ROM-monitor program or extra specific hardware is needed to make the debugging work. It is also possible to use the debugger on your own hardware design.

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

There are two C-SPY Nexus drivers to choose between, one for the CR16C microprocessor and, depending on which IAR product package you are using, one for the SC14 co-processor.

The C-SPY Nexus drivers use the USB port to communicate with the interface module, and the interface module communicates with the JTAG interface on the chip. The connections and cables that are used on different evaluation boards might differ.

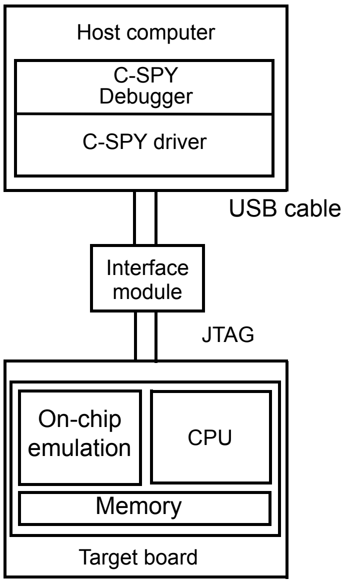


Figure 1: C-SPY CR16C Nexus communication overview

DIFFERENCES BETWEEN THE NEXUS DEBUGGER AND THE SIMULATOR

This table summarizes the key differences between the Nexus and simulator drivers:

Feature	Simulator	CR16C Nexus	SC14 Nexus
OP-fetch	x	x	x
Data breakpoints	x	x	
Execution in real time		x	x
Zero memory footprint	x		
Simulated interrupts	x		
Real interrupts		x	x
Cycle counter	x		x[1]
Code coverage	x		

Table 3: Driver differences

Feature	Simulator	CR16C Nexus	SC14 Nexus
Profiling	x	x[2]	
Real-time code profiling	x	x[3]	
Trace	x	x[4]	

Table 3: Driver differences (Continued)

1 In the SC14 Nexus debugger, the cycle counter is supported during single stepping. You can then view the value of the cycle counter in the Register window.
2 Requires software breakpoints.
3 Requires a device with CR16CPlus support.
4 Trace is available only if your hardware supports it.

Getting started

This section demonstrates a demo application that flashes the LED on the SC14480 board. The application is built and downloaded to the target system, and then executed.

There is a demo workspace file supplied with the C-SPY Nexus debugger, `LED.eww`. This workspace contains only one project, and the files are provided in the directory `cr16c\src\examples\LED`.

RUNNING THE DEMO PROGRAM

1 In the IAR Embedded Workbench IDE, choose **File>Open Workspace** to open the workspace file `LED.eww`.

2 Select the **Debug** build configuration from the drop-down list at the top of the workspace window.

3 Choose **Project>Options**. In addition to the factory settings, verify that these settings are used:

Category	Page	Option/Setting
General Options	Target	Co-processor variant: SC14480 - DIP
C/C++ Compiler	Output	Generate debug information
C/C++ Compiler	Preprocessor	Defined symbols: SC14480
Linker	Config	Linker command file: Override default: `lnk14480_ram.xcl`.
Debugger	Setup	Driver: `CR16C Nexus driver` Device description: Override default: `iosc14480.ddf`

Table 4: Project options for Nexus example

For more information about the C-SPY Nexus options, see *Setting options for debugging using the C-SPY Nexus debugger*, page 7.

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

4 Choose **Project>Make** to compile and link the source code.

5 Start C-SPY by clicking the **Download and Debug** button or by choosing **Project>Download and Debug**. C-SPY will download the application to the target system.

6 In C-SPY, choose **Debug>Go** or click the **Go** button to start the program. The LED should flash.

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

Mixing CR16C and SC14 source code

This section demonstrates how code for the SC14 Dedicated Instruction Processor (DIP) co-processor and CR16C can be integrated using the IAR Embedded Workbench IDE and how to run the code on the target hardware.

INTEGRATING SC14 CO-PROCESSOR CODE

This project contains both CR16C and DIP program source code. When running this code, it will produce a sweeping square-waveform on one of the output pins. The output pin is controlled by the DIP, while the CR16C will control the width of the cycle, by patching in the DIP code.

The CR16C program will start by copying the DIP program code from CR16C memory into the sequencer RAM. The program could also be linked directly to the sequencer area, but because it cannot be programmed into this location, there is little use for this, other than when debugging the DIP code.

When the application is loaded, the DIP execution is started and the CR16C goes into an idle, infinite loop.

The CR16C code also contains an interrupt service routine for DIP interrupts. This interrupt will be triggered from the DIP code, which is also in an infinite loop, toggling PD1 pin on the PD port.

When both targets are loaded and running, there will be two loops: one running on CR16C and one on the DIP.

THE DIP INTEGRATION PROJECT

1 In the IAR Embedded Workbench IDE, choose **File>Open Workspace** to open the workspace file `cr16c\src\examples\DipInteg.eww`.

2 Select the **Debug** build configuration from the drop-down list at the top of the workspace window.

The `main` function in `dip_int.c` will first initialize and start the DIP and then loop forever. When the interrupt instruction is executed by the DIP, the CR16C `Dip_Isr` interrupt service routine will be executed. This will patch the `WNT` instruction and change the length of the wait-cycle to produce the sweeping square-waveform on the PD1 pin.

The defined start address `SEQUENCER_RAM_START` and the labels `WAIT_H` and `WAIT_L` are used when patching the DIP program.

Because the DIP loop generates an interrupt, the CR16C DIP interrupt service routine will eventually be executed. When executed, this service routine will redefine instructions in the DIP, thus generating a sweeping square-waveform on the PD1 pin.

3 Before you build the project, make sure to use the following settings:

Category	**Page**	**Option**
General Options	Target	Data model: Large
General Options	Target	Co-processor variant: SC14480 - DIP
C/C++ Compiler	Preprocessor	Defined symbols: SC14480
SC14 Assembler	Preprocessor	Defined symbols: SC14480
Linker	Config	Linker command file: Override default: `lnk14480_ram.xcl`.
Debugger	Setup	Driver: CR16C Nexus Device description: Override default: `iosc14480.ddf`

Table 5: Options for project 6

4 Check that the hardware is connected and powered.

5 Choose **Project>Download and Debug** to start C-SPY. Alternatively, click the **Download and Debug** button.

The project should now compile and link automatically before downloading. Before starting the execution, connect a logging device such as an oscilloscope or a logic analyzer to the output pin PD1 to watch the waveform. You can also open the Live Watch window and drag the labels `WAIT_H` and `WAIT_L` to it so that you can watch the patched instruction.

Now start the execution by selecting **Go**. You can now see from the sweeping square-waveform on the logging device, or in the Live Watch window, how the memory contents in the DIP RAM area changes over time.

C-SPY® Nexus debugger-specific debugging

This chapter describes the additional options, menus, and features specific to the C-SPY Nexus debugger. The chapter contains the following sections:

- Setting options for debugging using the C-SPY Nexus debugger
- The Nexus menu
- Using the trace system
- Using breakpoints
- Real-time code profiling
- Resolving problems.

Setting options for debugging using the C-SPY Nexus debugger

Before you start the C-SPY Nexus debugger you must set some options for the debugger system—both C-SPY generic options and options required for the hardware system (C-SPY driver-specific options). Follow this procedure:

1. To open the **Options** dialog box, choose **Project>Options**.
2. To set C-SPY generic options and select a C-SPY driver:
   - Select **Debugger** from the **Category** list
   - On the **Setup** page, select the appropriate C-SPY driver from the **Driver** list.

   For information about the settings **Setup macros**, **Run to**, and **Device descriptions**, as well as for information about the pages **Extra Options** and **Plugins**, see the *IAR Embedded Workbench® IDE User Guide*.
3. To set the driver-specific options, select the appropriate driver from the **Category** list. Choose either **CR16C Nexus** or **SC14 Nexus**. Depending on which C-SPY driver you are using, different sets of available option pages appear.

For details about each page, see:

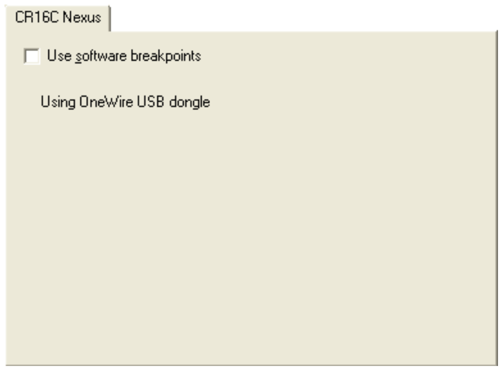
- *CR16C Nexus*, page 8
- *SC14 Nexus*, page 9
- *Download*, page 9
- *Extra Options*, page 10.

4 When you have set all the required options, click **OK** in the **Options** dialog box.

CR16C NEXUS

The **CR16C Nexus** driver options control the CR16C Nexus driver interface.

The only hardware connection supported is the OneWire USB dongle.


CR16C Nexus

☐ Use software breakpoints

Using OneWire USB dongle

Figure 2: CR16C Nexus driver options

Use software breakpoints

By default, all used breakpoint types use hardware breakpoints that are supported by the CR16C on-chip debug module. If the number of available hardware breakpoints is not sufficient for your needs, select the **Use software breakpoints** option. In this case, software breakpoints will be used for all code breakpoints, and all hardware breakpoints will be available for use by data and range breakpoints. For additional information, see *Hardware and software breakpoints*, page 19.

SC14 NEXUS

The **SC14 Nexus** driver options control the SC14 Nexus driver interface.

The only hardware connection type supported is the OneWire USB dongle.

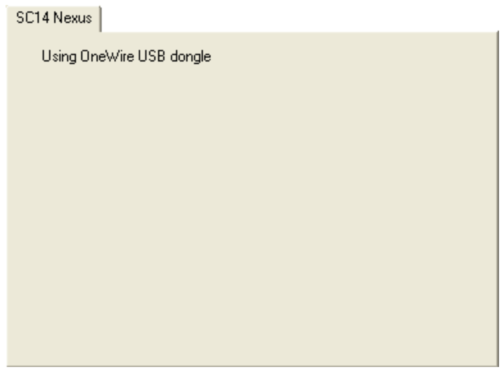


Figure 3: SC14 Nexus driver options

DOWNLOAD

By default, C-SPY downloads the application into RAM or flash when a debug session starts. The **Download** options lets you control the download.

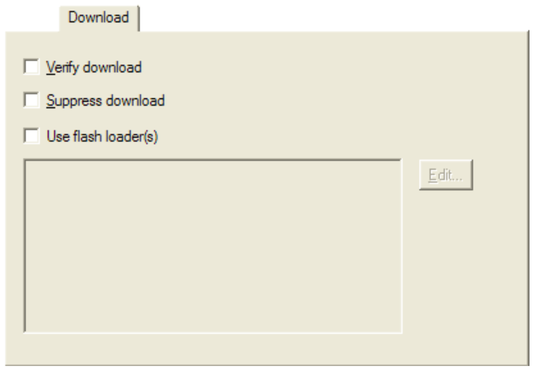


Figure 4: C-SPY Download options

Verify download

Use this option to verify that the downloaded code image can be read back from target memory with the correct contents.

Suppress download

Use this option to debug an application that already resides in target memory. When this option is selected, the code download is disabled, while preserving the present content of the flash.

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 program.

EXTRA OPTIONS

The **Extra Options** page provides you with a command line interface to C-SPY.

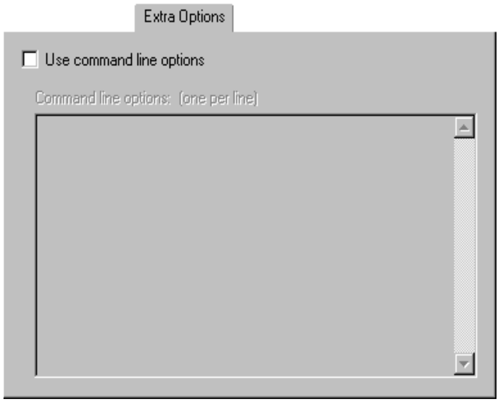


Figure 5: Extra Options page for C-SPY command line options

Use command line options

Additional command line arguments (not supported by the GUI) for C-SPY can be specified here. Currently there are no such command line arguments.

The Nexus menu

When the C-SPY Nexus Debugger is used, a new menu becomes available—the **Nexus** menu.

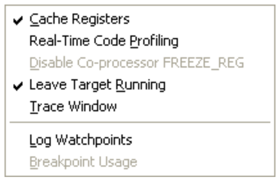


Figure 6: The Nexus menu

The following commands are available on the menu:

Menu command	Description
Cache Registers	Enables the register cache in the debugger, see *Cache registers*, page 11.
Real-Time Code Profiling	Displays the Real-Time Code Profiling window, see *Real-time code profiling*, page 26.
Disable Co-processor FREEZE_REG	Disables the `FREEZE_REG` register on the co-processor, see *Disable co-processor FREEZE_REG*, page 12.
Leave Target Running	When selected, the debugger will not stop the target application when the debug session closes.
Trace Window	Displays the Trace window, see *The Trace window*, page 13.
Log Watchpoints	Reports any pending watchpoints to the Debug Log window, but the timing and number of reported watchpoints may not be accurate. For information about watchpoints, see *Action*, page 22. Because the CR16C microprocessor only supports Nexus Class I, watchpoint support is not required. This menu command is only available in the C-SPY CR16C Nexus Debugger.
Breakpoint Usage	Opens the Breakpoint Usage dialog box, see *Breakpoint Usage dialog box*, page 26.

Table 6: The Nexus menu

CACHE REGISTERS

During a debug session, when for instance the Register window is open, all processor registers are read every time the execution is halted.

When communicating with real hardware it is sometimes more efficient to read all processor registers at the same time instead of issuing a read command for every single register.

In those cases it is possible to use a register cache in the debugger, which is enabled by choosing **Nexus>Cache Registers**. This means that the first read of any CPU register will read all registers and put the values in a cache within C-SPY. Subsequent readings of other CPU registers or modification of register values will then use the cache. When a **Step/Go** command is given, all values in the cache will be written to the hardware.

We recommend that the cache system is used when the CPU Register window is open during a debug session or when watching many register variables. In other cases the cache should be disabled to limit the register read/write overhead.

The execution speed is depending on the application as well as the usage of a register cache.

Note: This function is only available in the C-SPY CR16C Nexus Debugger.

DISABLE CO-PROCESSOR FREEZE_REG

The option **Disable co-processor FREEZE_REG** disables the `FREEZE_REG` register on the co-processor. When using this command, the co-processor will be left running even when a breakpoint is reached.

For instance, when a breakpoint is reached and triggered, not only the CR16C microprocessor but also the running co-processor will be halted. This is sometimes unwanted behavior; for instance when debugging a SC14*xxx* device with an attached radio frequency unit, a stop of the co-processor could damage this unit.

This command can only be used when running a SC14*xxx* device, and using the CR16C Nexus driver, and thus is not available for other devices.

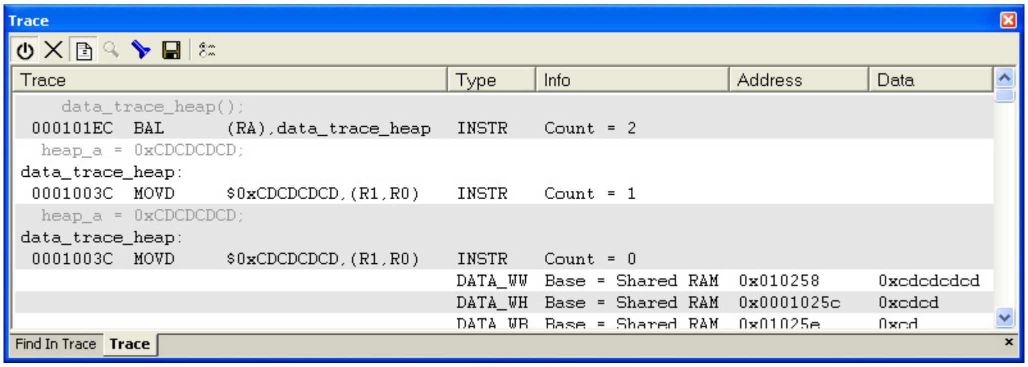
Using the trace system

In C-SPY, a *data trace* is generated from memory reads and writes. An *event trace* is generated from occurred events, such as bus events, DIP, DSP, or external events. *Instruction trace* is generated when single-stepping is performed or at a diverted program flow, for instance at the instructions `BRANCH`, `JUMP`, or `RETURN`.

For more detailed information about using the common features in the trace system, see the *IAR Embedded Workbench® IDE User Guide.*

THE TRACE WINDOW

The Trace window—available from the **Nexus** menu—displays a recorded sequence of executed machine instructions, events, and data reads and writes.



```
Trace                                     Type     Info                  Address       Data
    data_trace_heap();
 000101EC  BAL      (RA),data_trace_heap  INSTR    Count = 2
  heap_a = 0xCDCDCDCD;
data_trace_heap:
 0001003C  MOVD     $0xCDCDCDCD,(R1,R0)   INSTR    Count = 1
  heap_a = 0xCDCDCDCD;
data_trace_heap:
 0001003C  MOVD     $0xCDCDCDCD,(R1,R0)   INSTR    Count = 0
                                          DATA_WW  Base = Shared RAM  0x010258      0xcdcdcdcd
                                          DATA_WH  Base = Shared RAM  0x0001025c    0xcdcd
                                          DATA_WB  Base = Shared RAM  0x01025e      0xcd
```

Figure 7: The Trace window

C-SPY generates trace information based on the location of the program counter.

The Trace toolbar

The Trace toolbar at the top of the Trace window and in the Function trace window provides these toolbar buttons:

Toolbar button	Description
Enable/Disable	Enables and disables tracing. This button is not available in the Function trace window.
Clear trace data	Clears the trace buffer. Both the Trace window and the Function trace window are cleared.
Toggle Source	Toggles the Trace column between showing only disassembly or disassembly together with corresponding source code.
Browse	Toggles browse mode on and off for a selected item in the Trace window. For more information about browse mode, see the *IAR Embedded Workbench® IDE User Guide*.
Find	Opens the **Find In Trace** dialog box where you can perform a search; see *The Find in Trace dialog box*, page 17.
Save	Opens a standard Save dialog box where you can save the recorded trace information to a text file, with tab-separated col-umns.

Table 7: The Trace toolbar commands

Toolbar button	Description
Edit Settings	Opens the Trace Window Settings dialog box, see *Trace Window Settings*, page 15. This button is enabled only when you are using the Nexus debugger.

Table 7: The Trace toolbar commands (Continued)

The display area

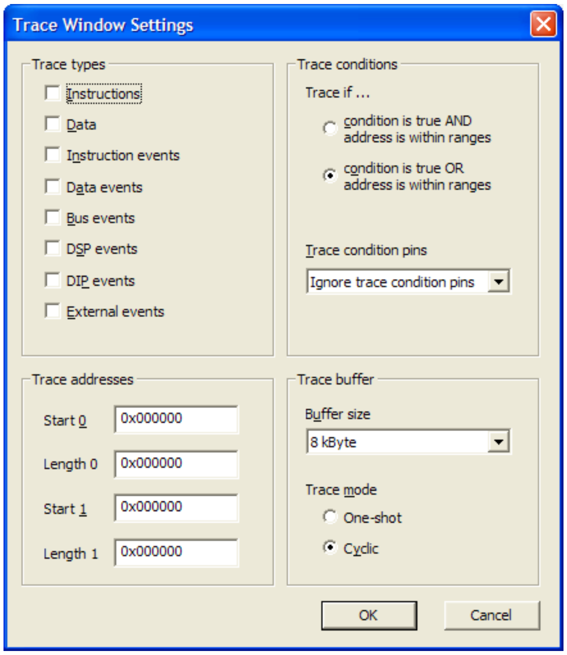
The display area displays trace information in these columns:

Trace window column	Description
Trace	The recorded sequence of executed machine instructions. Optionally, the corresponding source code can also be displayed. If instruction trace is not available, this column is blank.
Type	The type of trace record received from the trace hardware. For more information, see the documentation from your hardware manufacturer.
Info	Generic information pertaining to the trace, such as the memory zone for data trace, the instruction count since the latest instruction trace record for instruction trace and triggers and timer for event trace.
Address	The address of the data trace.
Data	The data value of the data trace.

Table 8: Trace window columns

TRACE WINDOW SETTINGS

Click the **Edit settings** button in the Trace window to open the **Trace Window Settings** dialog box.


Trace Window Settings

Trace types
- [ ] Instructions
- [ ] Data
- [ ] Instruction events
- [ ] Data events
- [ ] Bus events
- [ ] DSP events
- [ ] DIP events
- [ ] External events

Trace conditions

Trace if ...
- ( ) condition is true AND address is within ranges
- [✓ condition is true OR address is within ranges]

Trace condition pins: Ignore trace condition pins

Trace addresses

Start 0	0x000000
Length 0	0x000000
Start 1	0x000000
Length 1	0x000000

Trace buffer

Buffer size: 8 kByte

Trace mode
- ( ) One-shot
- [✓ Cyclic]

Trace types

Use the **Trace types** options to choose the trace type you want to use.

Option	Description
Instructions	Enables instruction trace
Data	Enables data trace
Instruction Events	Enables instruction timing events trace
Data Events	Enables data timing events trace
Bus Events	Enables bus events trace
DSP Events	Enables DSP events trace
DIP Events	Enables DIP events trace
External Events	Enables external events trace

Table 9: Trace window settings: Trace types

Trace addresses

Use the **Trace addresses** text fields to set the addresses and sizes of the trace memory windows.

Option	Description
Start 0	The start address of the trace memory window 0
Length 0	The size of the trace memory window 0
Start 1	The start address of the trace memory window 1
Length 1	The size of the trace memory window 1

Table 10: Trace window settings: Trace addresses

Trace conditions

Use the **Trace conditions** options to set the conditions for the trace based on the trace condition pins and the trace addresses.

Trace if	Description
condition is true AND address is within ranges	Traces if the condition for the trace condition pins is true *and* the trace address is within ranges. Both of the conditions must be fulfilled.
condition is true OR address is within ranges	Traces if the condition for the trace condition pins is true *or* the trace address is within ranges. Only one of the conditions must be fulfilled.

Table 11: Trace window settings: Trace conditions

Use the drop-down menu **Trace condition pins** to choose the logical expression configuration for the trace condition pins.

Option	Description
Ignore trace condition pins	The settings of the trace condition pins are ignored.
P2[2] or P2[5]	Pin P2[2] or pin P2[5] must be true.
P2[2] nor P2[5]	Pin P2[2] and pin P2[5] must be false.
P2[2] and P2[5]	Pin P2[2] and pin P2[5] must be true.
P2[2] nand P2[5]	Pin P2[2] and pin P2[5] cannot both be true.
P2[2] xor P2[5]	Either pin P2[2] or pin P2[5] must be true, but not both.
P2[2] xnor P2[5]	Both pin P2[2] and pin P2[5] must be either true or false.

Table 12: Trace window settings: Trace Condition Pins

Trace buffer

Use the Trace buffer options to set options related to the trace buffer. **Buffer size** is the size of the trace buffer. A trace buffer is the memory buffer containing the trace data.

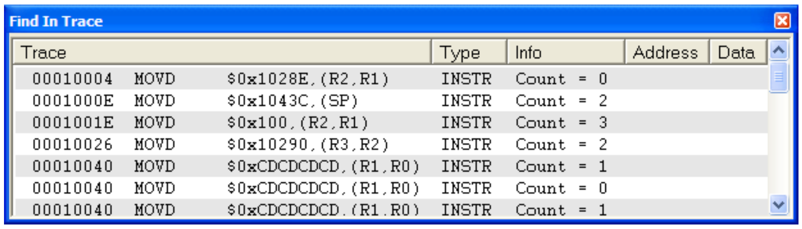
Trace mode controls the behavior of the trace once the trace buffer is full.

Option	Description
One shot	The trace stops once the trace buffer is full.
Cyclic	The trace continues, and when the buffer is full it wraps around.

Table 13: Trace window settings: Trace Buffer options

THE FIND IN TRACE WINDOW

The Find In Trace window—available from the **View>Messages** menu—displays the result of searches in the trace data.



```
Trace                                   Type   Info          Address  Data
 00010004  MOVD   $0x1028E,(R2,R1)      INSTR  Count = 0
 0001000E  MOVD   $0x1043C,(SP)         INSTR  Count = 2
 0001001E  MOVD   $0x100,(R2,R1)        INSTR  Count = 3
 00010026  MOVD   $0x10290,(R3,R2)      INSTR  Count = 2
 00010040  MOVD   $0xCDCDCDCD,(R1,R0)   INSTR  Count = 1
 00010040  MOVD   $0xCDCDCDCD,(R1,R0)   INSTR  Count = 0
 00010040  MOVD   $0xCDCDCDCD,(R1,R0)   INSTR  Count = 1
```

Figure 8: The Find In Trace window

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. Double-click an item in the Find in Trace window to bring up the same item in the Trace window.

You specify the search criteria in the **Find In Trace** dialog box. For information about how to open this dialog box, see *The Find in Trace dialog box*, page 17.

THE FIND IN TRACE DIALOG BOX

Use the **Find in Trace** dialog box—available by choosing **Edit>Find and Replace>Find** or from the Trace window toolbar—to specify the search criteria for advanced searches in the trace data. Note that the **Edit>Find and Replace>Find** command is context-dependent. It displays the **Find in Trace** dialog box if the Trace window is the current window or the **Find** dialog box if the editor window is the current window.

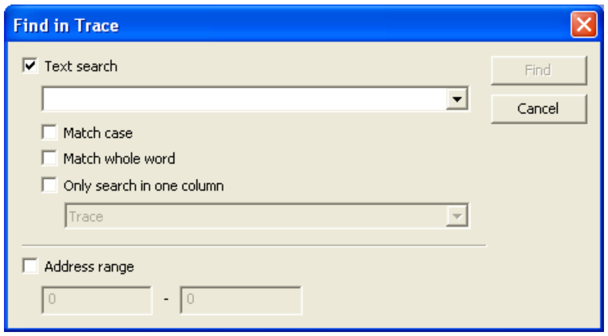


Figure 9: The Find in Trace dialog box

The search results are displayed in the Find In Trace window—available by choosing the **View>Messages** command, see *The Find In Trace window*, page 17.

In the **Find in Trace** dialog box, you specify the search criteria with the following settings:

Text search

A text field where you type the string you want to search for. Use these options to fine-tune the search:

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

Address Range

Use the text fields to specify an address range. The trace data within the address range is displayed. If you also have specified a text string in the **Text search** field, the text string is searched for within the address range.

Using breakpoints

The C-SPY CR16C Nexus Debugger provides a wide and flexible choice of breakpoints; you can set:

- *Code*, *data*, and *range* breakpoints

  For more information, see *Range breakpoints*, page 20, and the *IAR Embedded Workbench® IDE User Guide*.
- *Start points* and *stop points*, which are triggers for real-time code profiling.

  For more information, see *Start Point breakpoints*, page 24, and *Stop Point breakpoints*, page 25.
- *Watchpoints*, which when triggered display a message in the Debug Log window without stopping the application execution.

  When a breakpoint has been reached, execution will normally be halted. If you do not want to stop the execution when a breakpoint is triggered, define it as a *watchpoint* by selecting the action **Watch** on the **Range** page in the Breakpoints dialog box, see *Action*, page 22. Then choose **Nexus>Log Watchpoints** to log your watchpoints, see *The Nexus menu*, page 11.

For information about the different methods for setting breakpoints, the generic facilities for monitoring breakpoints, and the different breakpoint consumers, see the *IAR Embedded Workbench® IDE User Guide*.

See also the *Breakpoint Usage dialog box*, page 26.

HARDWARE AND SOFTWARE BREAKPOINTS

When you set a breakpoint, trigger, or watchpoint, the physical breakpoint used on the target can be one of two types, *hardware* and *software*. Which type is used depends on several factors.

With the CR16C Nexus driver, and for data and range breakpoints, hardware breakpoints will be used. For code breakpoints, software breakpoints will be used if they are enabled, otherwise hardware breakpoints will be used.

To use software breakpoints for code breakpoints, choose **Project>Options>CR16C** and select the option **Use software breakpoints**. By doing so, all hardware breakpoints will be available for use by data and range breakpoints.

With the SC14 Nexus driver, there are no hardware breakpoints available. However, there is an unlimited number of software breakpoints. This means that you can only set code breakpoints.

The table below summarizes the characteristics of hardware and software breakpoints:

Type	Category	Number	Execution overhead
Hardware[1]	Code[2], Data, Range, Watchpoints, Start/Stop points	8[3]	No
Software	Code	Unlimited	Yes

Table 14: Hardware and software breakpoints in Nexus debugger

1 CR16C only.
2 When software breakpoints are disabled.
3 With CR16CPlus, the number of hardware breakpoints is 16.

Hardware breakpoints

Hardware breakpoints are supported by the CR16C Nexus debug module located on the chip. The number of available breakpoints is 8 (16 on devices with CR16CPlus support). Data and range breakpoints require one or more hardware breakpoints.

Software breakpoints

Software breakpoints can only be used for code breakpoints and only when the program is located in read/write memory. When enabled, this option will cause the breakpoints to be implemented by a temporary substitution of the actual instruction with the `EXCP BP` instruction. Before resuming execution, the original instruction will be restored. This will generate execution time overhead when running an application.

RANGE BREAKPOINTS

A *range breakpoint* can be set either as a range or as a single location break. The breakpoint will be detected when the specified memory access within the range is made. Range breakpoints are only available in the CR16C Nexus driver.

Range breakpoints can be set by using the New Breakpoint dialog box, which is available from the Breakpoints window context menu.

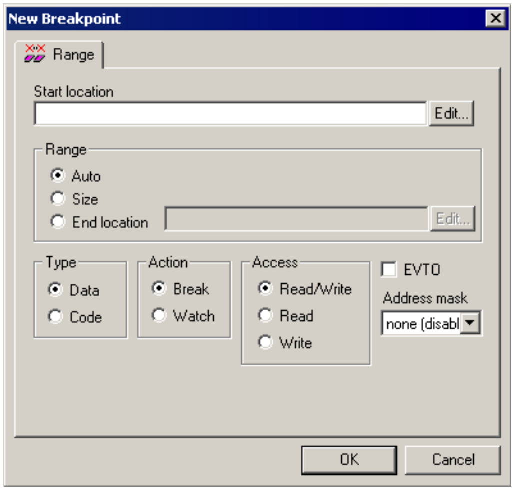


Figure 10: The Range breakpoints dialog box

Start location

Specify the location of the breakpoint in the **Start location** text box. Alternatively, click the **Edit** browse button to open the **Enter Location** dialog box. You can choose between these locations and their possible settings:

Location type	Description/Examples
Expression	Any expression that evaluates to a valid address, such as a function or variable name. For example, `my_var` refers to the location of the variable `my_var`, and `arr[3]` refers to the third element of the array `arr`.
Absolute Address	An absolute location on the form *`zone`*`:`*`hexaddress`* or simply *`hexaddress`*. Zone specifies in which memory the address belongs. For example `Memory:0x42`. If you enter a combination of a zone and address that is not valid, C-SPY will indicate the mismatch.

Table 15: Location types

Location type	Description/Examples
Source Location	A location in the C source code using the syntax: `{file path}.row.column.` *File* specifies the filename and full path. *Row* specifies the row in which you want the breakpoint. *Column* specifies the column in which you want the breakpoint. Note that the Source Location type is usually meaningful only for code breakpoints. For example, `{C:\IAR Systems\xxx\Utilities.c}.22.3` sets a breakpoint on the third character position on line 22 in the source file `Utilities.c`.

Table 15: Location types (Continued)

Type

The following table shows the available types of range breakpoints:

Type	Break when
Code	Breakpoint is set as code access
Data	Breakpoint is set as data access

Table 16: Range breakpoint categories

Action

The following types of actions for range breakpoints are available:

Action	Description
Break	When the breakpoint is detected, execution will be halted.
Watch	When the breakpoint is detected, execution will continue. This is referred to as a watchpoint. Watchpoints are logged, if the command **Log Watchpoints**—available on the Nexus menu—is selected. The **Watch** option can also be used in conjunction with the **EVTO** option if an external signal/trigger is needed, without stopping the execution.

Table 17: Range breakpoint types

Access

Specifies the type of memory access guarded by the breakpoint:

Type	Description
Read/Write	Read or write from location
Read	Read from location

Table 18: Range breakpoint memory access types

Type	Description
Write	Write to location

Table 18: Range breakpoint memory access types

EVTO

Selecting **EVTO** will set the Event Out (EVTO) pin on the CR16C when the breakpoint is detected. This can be used to trigger external hardware such as measuring equipment.

Note: There is only one EVTO pin, and enabling the EVTO toggle on more than one breakpoint will set the EVTO pin on the first breakpoint detected.

Address mask

Using the mask will set the 0–3 least significant bits in the address as indifferent. This can be used to minimize the number of used breakpoints because setting a range breakpoint on one address (only consuming one hardware breakpoint) and masking all three least significant bits will cover a larger range.

Example

Setting a single-range break on address `0x1000` and selecting an address mask of two bits will guard `0x1000–0x1003`. Setting a single-range break on address `0x1004` with an address mask of two bits will cover the range `0x1000–0x1004`. The actual range depends on the address that the mask is used on.

Using Address Mask on a range breakpoint will make both the start and end address masked.

START POINT BREAKPOINTS

Start points for real-time code profiling (see *Real-time code profiling*, page 26) can be set by using the **Start Point** dialog box, which is available from the Breakpoints window context menu.

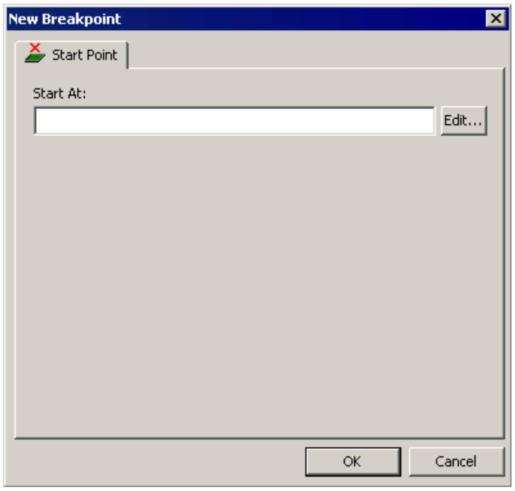


Figure 11: Start Point breakpoints dialog box

Start At

Specify the location of the breakpoint in the **Start At** text box. Alternatively, click the **Edit** browse button to open the **Enter Location** dialog box. For information about the locations and their possible settings, see *Start location*, page 21.

STOP POINT BREAKPOINTS

Stop points for real-time code profiling (see *Real-time code profiling*, page 26) can be set by using the **Stop Point** dialog box, which is available from the Breakpoints window context menu.

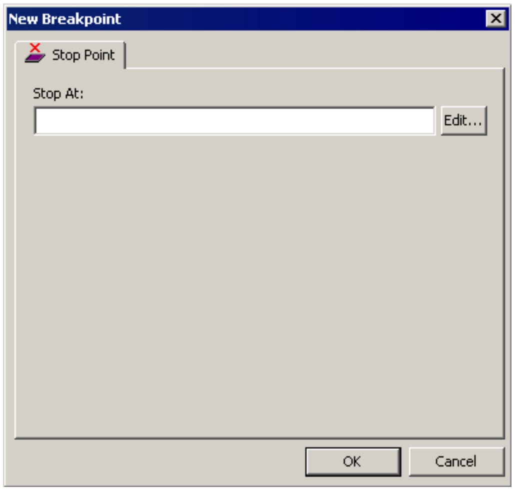


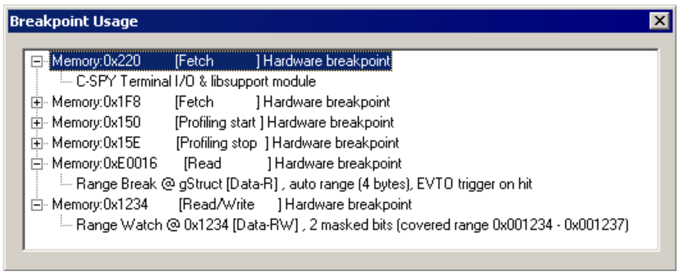
Figure 12: The Stop Point dialog box

Stop At

Specify the location of the stop point in the **Stop At** text box. Alternatively, click the **Edit** browse button to open the **Enter Location** dialog box. For information about the locations and their possible settings, see *Start location*, page 21.

BREAKPOINT USAGE DIALOG BOX

The **Breakpoint Usage** dialog box—available from the driver-specific menu—lists all active breakpoints.



```
Memory:0x220     [Fetch          ] Hardware breakpoint
    C-SPY Terminal I/O & libsupport module
Memory:0x1F8     [Fetch          ] Hardware breakpoint
Memory:0x150     [Profiling start ] Hardware breakpoint
Memory:0x15E     [Profiling stop  ] Hardware breakpoint
Memory:0xE0016   [Read           ] Hardware breakpoint
    Range Break @ gStruct [Data-R] , auto range (4 bytes), EVT0 trigger on hit
Memory:0x1234    [Read/Write     ] Hardware breakpoint
    Range Watch @ 0x1234 [Data-RW] , 2 masked bits (covered range 0x001234 - 0x001237)
```

Figure 13: The Breakpoint Usage dialog box

In addition to listing all breakpoints that you have defined, this dialog box also lists the internal breakpoints that the debugger is using.

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

For more information, see the *IAR Embedded Workbench® IDE User Guide*.

Real-time code profiling

Real-time code profiling is only available on devices with CR16CPlus support. It provides you with information about the number of clock cycles executed within a code range.

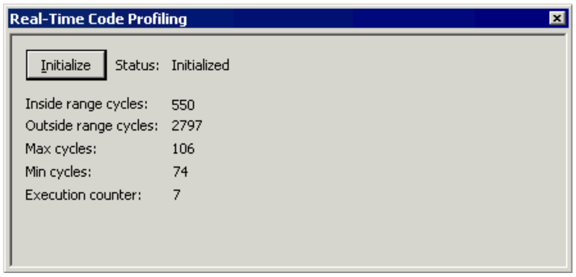
To set up the profiling, follow these steps:

1. In the editor window or the Disassembly window, click to place the insertion point where you want the profiled code range to start in the source code. Right-click to bring up the context menu and choose **Toggle Breakpoint (StartPoint)**.

2. In the editor or the Disassembly window, navigate to the location where you want the profiled code range to end and click to place the insertion point. Then right-click and choose **Toggle Breakpoint (StopPoint)**. Now you have specified the code range to be monitored.

   Note: To set the start point and the stop point for the profiling, you can also choose **Edit>Breakpoints** to open the **Breakpoints** dialog box. See *Start Point breakpoints*, page 24 and *Stop Point breakpoints*, page 25.

3 Choose **Nexus>Real-time Code Profiling** to open the Real-Time Code Profiling window. Click **Initialize** to enable the profiling.



Status: Initialized

Inside range cycles: 550
Outside range cycles: 2797
Max cycles: 106
Min cycles: 74
Execution counter: 7

Figure 14: Real-Time Code Profiling window

The Real-Time Code Profiling window displays the following information:

Type	Description
Status	Indicates whether the profiling is enabled and running or not.
Inside range cycles	The number of executed clock cycles within the specified range.
Outside range cycles	The number of executed clock cycles outside of the specified range.
Max cycles	The maximum number of executed cycles between the starting and stopping points. This value is used for worst case evaluation.
Min cycles	The minimum number of executed cycles between the starting and stopping points. This value is used for best case evaluation.
Execution counter	The number of times the profiling range has been executed.

Table 19: Real-time code profiling information

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 the contents of your linker command file and make sure that your application has not been linked to the wrong address.
- Check that you are using correct linker command file.

To override the default linker command file:

- Choose **Project>Options**
- Select the **Linker** category
- Click the **Config** tab
- Choose the appropriate linker command file in the **Linker command 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.

- Check the communication devices on your host computer
- Verify that the cable is properly plugged in and not damaged or of the wrong type
- Verify that the target chip is properly mounted on the evaluation board
- 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; see *Communication options*, page 10
- Examine the linker command file to make sure that your application has not been linked to the wrong address.

Using flash loaders

This chapter describes the flash loader, what it is and how to use it. More specifically, this means:

- Introduction to the flash loader
- Reference information on the flash loader.

Introduction to the flash loader

This section introduces the flash loader.

These topics are covered:

- Briefly about the flash loader
- Setting up the flash loader(s)
- The flash loading mechanism
- Build considerations.

BRIEFLY ABOUT THE FLASH LOADER

A flash loader is an agent that is downloaded to the target. It fetches your application from the debugger and programs it into flash memory. The flash loader uses the file I/O mechanism to read the application program from the host. You can select one or several flash loaders, where each flash loader loads a selected part of your application. This means that you can use different flash loaders for loading different parts of your application.

A set of flash loaders for various microcontrollers is provided with IAR Embedded Workbench for CR16C. The flash loader API, documentation, and several implementation examples are available to make it possible for you to implement your own flash loader.

SETTING UP THE FLASH LOADER(S)

To use a flash loader for downloading your application:

1. Choose **Project>Options**.
2. Choose the **Debugger** category and click the **Download** tab.

3. Select the **Use Flash loader(s)** option. A default flash loader configured for the device you have specified will be used. The configuration is specified in a preconfigured `board` file.

4. To override the default flash loader or to modify the behavior of the default flash loader to suit your board, select the **Override default .board file** option, and **Edit** to open the **Flash Loader Configuration** dialog box. A copy of the `*.board` file will be created in your project directory and the path to the `*.board` file will be updated accordingly.

5. The **Flash Loader Overview** dialog box lists all currently configured flash loaders; see *Flash Loader Overview dialog box*, page 31. You can either select a flash loader or open the **Flash Loader Configuration** dialog box.

   In the **Flash Loader Configuration** dialog box, you can configure the download. For reference information about the various flash loader options, see *Flash Loader Configuration dialog box*, page 33.

THE FLASH LOADING MECHANISM

When the **Use flash loader(s)** option is selected and one or several flash loaders have been configured, these steps are performed when the debug session starts:

1. C-SPY downloads the flash loader into target RAM.
2. C-SPY starts execution of the flash loader.
3. The flash loader programs the application code into flash memory.
4. The flash loader terminates.
5. C-SPY switches context to the user application.

   Steps 2 to 4 are performed for each memory range of the application.

   The steps 1 to 4 are performed for each selected flash loader.

BUILD CONSIDERATIONS

When you build an application that will be downloaded to flash, special consideration is needed. Two output files must be generated. The first is the usual UBROF file (`d45`) that provides the debugger with debug and symbol information. The second file is a simple-code file (filename extension `sim`) that will be opened and read by the flash loader when it downloads the application to flash memory.

The simple-code file must have the same path and name as the UBROF file except for the filename extension.

To create the extra output file, choose **Project>Options** and select the **Linker** category. Select the **Allow C-SPY-specific extra output file** option. On the **Extra Output** page, select the **Generate extra output file** option. Choose the **simple-code** output format and the format variant **None**. Do not override the default output file. For reference information about these options, see the *IAR Embedded Workbench® IDE User Guide*.

Reference information on the flash loader

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

- *Flash Loader Overview dialog box*, page 31
- *Flash Loader Configuration dialog box*, page 33.

Flash Loader Overview dialog box

The **Flash Loader Overview** dialog box is available from the **Debugger>Download** page.

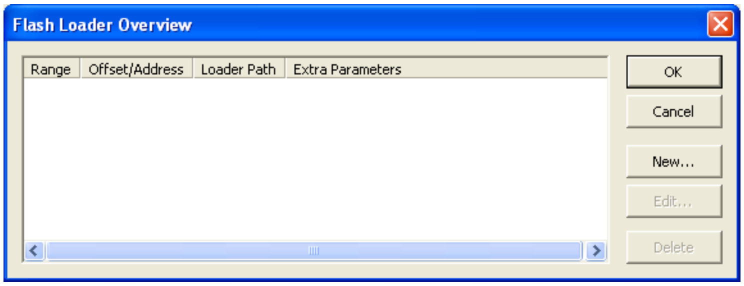


Figure 15: Flash Loader Overview dialog box

This dialog box lists all defined flash loaders. If you have selected a device on the **General Options>Target** page for which there is a flash loader, this flash loader is by default listed in the **Flash Loader Overview** dialog box.

The display area

Each row in the display area shows how you have set up one flash loader for flashing a specific part of memory:

Range	The part of your application to be programmed by the selected flash loader.
Offset/Address	The start of the memory where your application will be flashed. If the address is preceded with a, the address is absolute. Otherwise, it is a relative offset to the start of the memory.

Loader Path	The path to the flash loader `*.flash` file to be used (`*.out` for old-style flash loaders).
Extra Parameters	List of extra parameters that will be passed to the flash loader.

Click on the column headers to sort the list by range, offset/addrses, etc.

Function buttons

These function buttons are available:

OK	The selected flash loader(s) will be used for downloading your application to memory.
Cancel	Standard cancel.
New	Displays a dialog box where you can specify what flash loader to use; see *Flash Loader Configuration dialog box*, page 33.
Edit	Displays a dialog box where you can modify the settings for the selected flash loader; see *Flash Loader Configuration dialog box*, page 33.
Delete	Deletes the selected flash loader configuration.

Flash Loader Configuration dialog box

The **Flash Loader Configuration** dialog box is available from the **Flash Loader Overview** dialog box.

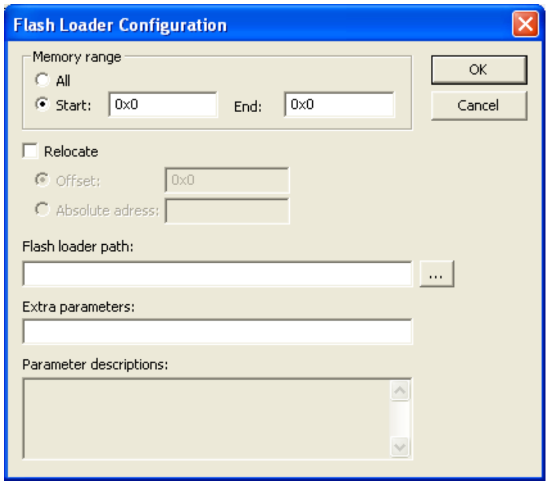


Figure 16: Flash Loader Configuration dialog box

Use the **Flash Loader Configuration** dialog box to configure the download to suit your board. A copy of the default `board` file will be created in your project directory.

Memory range

Specify the part of your application to be downloaded to flash memory. Choose between:

All	The whole application is downloaded using this flash loader.
Start/End	Specify the start and the end of the memory area for which part of the application will be downloaded.

Relocate

Overrides the default flash base address, that is relocate the location of the application in memory. This means that you can flash your application to a different location from where it was linked. Choose between:

Offset	A numeric value for a relative offset. This offset will be added to the addresses in the application file.

Absolute address	A numeric value for an absolute base address where the application will be flashed. The lowest address in the application will be placed on this address. Note that you can only use one flash loader for your application when you specify an absolute address.

You can use these numeric formats:

`123456`	Decimal numbers.
`0x123456`	Hexadecimal numbers
`0123456`	Octal numbers

The default base address used for writing the first byte—the lowest address—to flash is specified in the linker configuration file used for your application. However, it can sometimes be necessary to override the flash base address and start at a different location in the address space. This can, for example, be necessary for devices that remap the location of the flash memory.

Flash loader path

Use the text box to specify the path to the flash loader file (`*.flash`) to be used by your board configuration.

Extra parameters

Some flash loaders define their own set of specific options. Use this text box to specify options to control the flash loader. For information about available flash loader options, see the **Parameter descriptions** field.

Parameter descriptions

The **Parameter descriptions** field displays a description of the extra parameters specified in the **Extra parameters** text box.