

# Getting Started

with IAR Embedded Workbench<sup>®</sup>



GSEW-3

The logo for IAR Systems, featuring a stylized 'I' icon to the left of the text 'IAR' and 'SYSTEMS' stacked vertically.

## **COPYRIGHT NOTICE**

Copyright © 2009–2012 IAR Systems AB.

No part of this document may be reproduced without the prior written consent of IAR Systems AB. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

## **DISCLAIMER**

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

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

## **TRADEMARKS**

IAR Systems, IAR Embedded Workbench, C-SPY, visualSTATE, The Code to Success, IAR KickStart Kit, I-jet, IAR, and the logotype of IAR Systems are trademarks or registered trademarks owned by IAR Systems AB.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

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

## **EDITION NOTICE**

Third edition: April 2012

Part number: GSEW-3

Internal reference: Too6.4, Mys12, ISUD.

# Contents

Preface .....	5
<b>About this guide</b> .....	5
<b>Document conventions</b> .....	5
Introduction .....	7
<b>Product portfolio overview</b> .....	7
<b>Device support</b> .....	9
<b>Tutorials</b> .....	9
<b>User documentation</b> .....	9
<b>More resources</b> .....	10
IAR Embedded Workbench tools overview .....	11
<b>The IDE</b> .....	11
<b>The IAR C/C++ Compiler</b> .....	14
<b>The IAR Assembler</b> .....	15
<b>The IAR Linker and related tools</b> .....	15
<b>The IAR C-SPY Debugger</b> .....	17
Developing embedded applications .....	21
<b>The development cycle</b> .....	21
<b>Commonly used software models</b> .....	22
<b>The build process</b> .....	24
<b>Programming for performance</b> .....	27
<b>Considering hardware and software factors</b> .....	29
<b>Application execution</b> .....	32
Creating an application project .....	37
<b>Creating a workspace</b> .....	37
<b>Creating a new project</b> .....	38
<b>Setting project options</b> .....	39
<b>Adding source files to the project</b> .....	39

<b>Setting tool-specific options</b> .....	40
<b>Compiling</b> .....	42
<b>Linking</b> .....	43
<b>Debugging</b> .....	45
<b>Setting up for debugging</b> .....	45
<b>Starting the debugger</b> .....	46
<b>Executing your application</b> .....	48
<b>Inspecting variables</b> .....	49
<b>Monitoring memory and registers</b> .....	51
<b>Using breakpoints</b> .....	52
<b>Viewing terminal I/O</b> .....	54
<b>Analyzing your application's runtime behavior</b> .....	55

# Preface

Welcome to *Getting Started with IAR Embedded Workbench*®.

## ABOUT THIS GUIDE

The purpose of this guide is to provide an introduction to IAR Embedded Workbench, how to work in the IDE, and how to use the tools for developing embedded systems software. By highlighting selected features, the guide explores the purpose and capabilities of the tools.

Note that you should have working knowledge of the C or C++ programming language, application development for embedded systems, the architecture and instruction set of the microcontroller you are using (refer to the chip manufacturer's documentation), and finally the operating system of your host computer.

**Note:** Some descriptions in this guide only apply to certain product packages of IAR Embedded Workbench, depending on for example, the microcontroller or your specific variant of the product package. For example, not all packages support C++.

## DOCUMENT CONVENTIONS

When this text refers 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 `target\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench N.n\target\doc`.



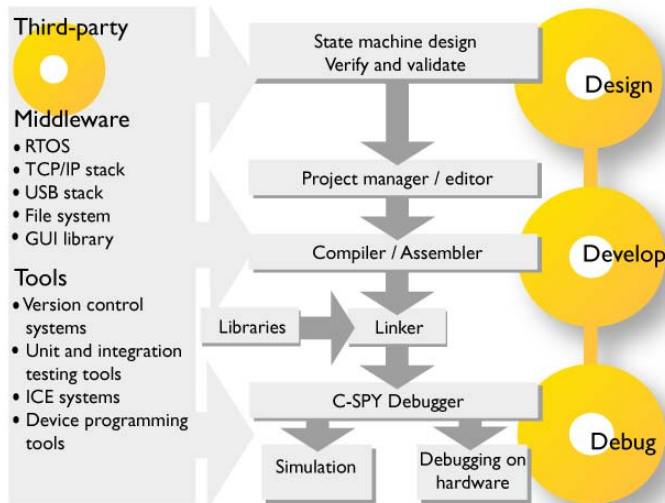
# Introduction

- Product portfolio overview
- Device support
- Tutorials
- User documentation
- More resources

For information about installation and licensing, see the *Quick Reference* booklet that is provided in the product box.

## PRODUCT PORTFOLIO OVERVIEW

This figure shows the various tools from IAR Systems and third-party vendors and how they interact with each other:

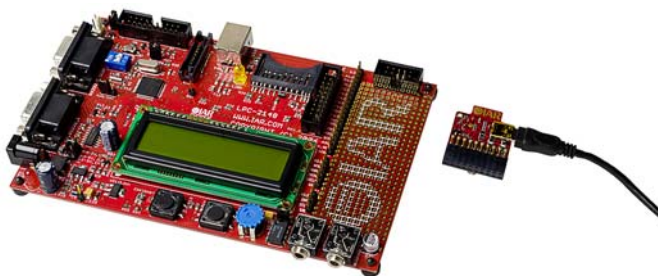


**IAR Embedded Workbench** provides an integrated development environment that allows you to develop and manage complete application projects for embedded systems. The environment comprises tools for compiling, linking, and

debugging with comprehensive and specific target support. You will have the same user interface regardless of which microcontroller you use.

**IAR visualSTATE** integrates a set of development tools for designing, testing, and implementing embedded applications based on state machines. It provides formal verification and validation tools and generates C source code from your design. To get started, the product installation provides tutorials, a Getting Started guide and several examples, where some are generic and some are adapted for certain evaluation boards.

**IAR KickStart Kits** are integrated kits for developing embedded applications for certain microcontrollers. Each kit contains an evaluation board, software development tools with sample projects, and a hardware debug probe or emulator.



**I-jet** is an in-circuit debugging probe, which connects the chip's JTAG/SWD ports via USB to the host PC. I-jet integrates into IAR Embedded Workbench and is plug-and-play compatible.

### **Third-party tools and utilities**

There is a wide range of third-party tools and utilities that can be integrated with IAR Embedded Workbench. Examples of such products are version control systems, editors, C-SPY plugin modules for RTOS-aware debugging and different debug probes, protocol stacks, etc.

### **Interoperability with other build tools**

Depending on which object format and which application binary interfaces your products use, the IAR toolchain can be interoperable with toolchains from other vendors. The level of interoperability ranges from debugging an IAR tools executable file in a third-party debugger to full link-level compatibility.



## DEVICE SUPPORT

To get a smooth start with your product development, the IAR product installation includes preconfigured files for different devices:

**Header files for peripheral I/O** are device-specific I/O header files that define peripheral units.

**Linker configuration files** contain the information required by the linker to place code and data in memory. Depending on your product package, either templates for linker configuration files, or ready-made linker configuration files for supported devices are provided.

**Device description files** handle several of the device-specific details required by the debugger, such as definitions of peripheral registers and groups of these, which means that you can view SFR addresses and bit names while debugging.

**A flash loader** is an agent that is downloaded to the target, that fetches your application from the debugger and programs it into flash memory. Depending on your product package, flash loaders are available for a selection of devices. If your device is not among them, you can build your own flash loader. Note that some debug probes provide the corresponding functionality, which means that a dedicated flash loader is not needed.

**Examples for getting started** with your software development are available to give you a smooth start. Depending on your product package, there are either a few or several hundreds of working source code examples where the complexity ranges from simple LED blink to USB mass storage controllers. You can access the examples via the Information Center, available from the **Help** menu.

## TUTORIALS

The tutorials give you hands-on training to help you get started using the IAR Embedded Workbench IDE and its tools. The tutorials are divided into different parts and you can work through all tutorials as a suite or you can choose to go through the tutorials individually. The tutorials are set up for the C-SPY simulator so that you can get started using the debugger without any hardware available.

You can access the tutorials from the Information Center available from the **Help** menu in the IDE. You can find all the files needed for the tutorials in the `target\tutor` directory.

## USER 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 in the IDE.

For last minute changes, we recommend that you read the release notes—also available from the **Help** menu—for recent information that might not be included in the user documentation.

All documentation is located in the directories `target\doc` and `common\doc`.

## **MORE RESOURCES**

On the IAR Systems web site [www.iar.com/support](http://www.iar.com/support) you can find technical notes and application notes.

If you have a software update agreement (SUA) you can also access the latest product information, and download product updates and support files for new devices from MyPages on the IAR Systems web site.

### **Requesting technical assistance**

If you discover a problem with the IAR Systems tools, work through this list of troubleshooting tips:

- 1** Learn more about the topic in the user documentation. For guidelines, see the Information Center.
- 2** Read the section Known Problems in the release notes to see if you can find something related that has already been reported.
- 3** If the problem remains, try to isolate it as much as possible. A small code example, project settings, and a description about how to reproduce the problem will significantly help in providing timely support.
- 4** Send in a report either via the web site or by contacting your local IAR Systems representative. Make sure also to include information about product name, product version, and the license number.

# IAR Embedded Workbench tools overview

This chapter gives an overview of the different tools in IAR Embedded Workbench®.

## THE IDE

The IDE is the framework where all tools needed to build your application are integrated: a C/C++ compiler, an assembler, a linker, library tools, an editor, a project manager, and the IAR C-SPY® Debugger.

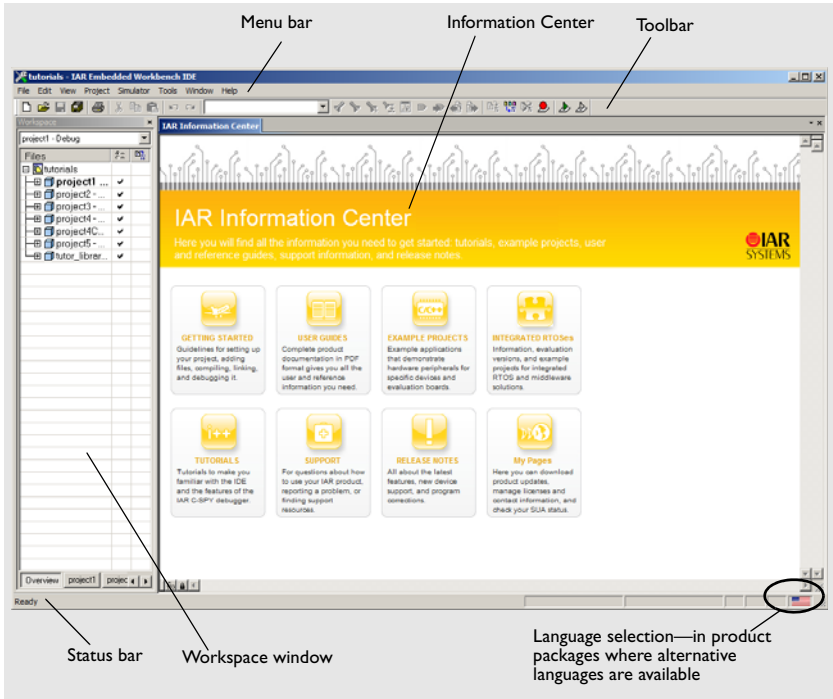
The toolchain that comes with your product package supports a specific microcontroller. However, the IDE can simultaneously contain multiple toolchains for various microcontrollers. This means that if you have IAR Embedded Workbench installed for several microcontrollers, you can choose which microcontroller to develop for.

**Note:** The compiler, assembler, linker, and the C-SPY debugger can also be run from a command line environment, if you want to use them as external tools in an already established project environment.

### To start the IDE:

Click the start button on the Windows **Start** menu, or double-click a workspace filename (filename extension *eww*), or use the file `IarIdePm.exe`, located in the `common\bin` directory of your IAR Embedded Workbench installation.

The IDE main window is opened:



When you first open IAR Embedded Workbench, the IDE main window displays the IAR Information Center. Here you can find all the information you need to get started: tutorials, example projects, user guides, support information, and release notes.

## Configuring the IDE

There are many ways of making the IDE suit your preferences and requirements:

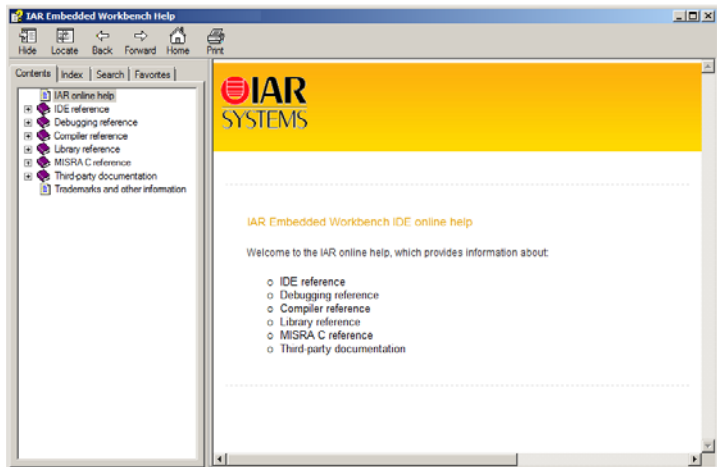
- Organizing the windows—you can *dock* windows at specific places, and organize them in tab groups. You can also make a window *floating*, which means it is always on top of other windows. If you change the size or position of a floating window, other currently open windows are not affected. The status bar, located at the bottom of the main window, contains useful help about how to arrange windows.

- Extending the toolchain with an external tool, for example a revision control system or an editor of your choice. You can also add IAR visualSTATE to the toolchain, which means that you can add state machine diagrams directly to your project in the IDE.
- Invoking external tools from the **Tools** menu
- Customizing the IDE, with commands for example for:
  - Configuring the editor
  - Changing common fonts
  - Changing key bindings
  - Using an external editor of your choice
  - Configuring the project build command
  - Configuring the amount of output to the Messages window.

### To view the online help system:

Choose **Help>Content** or click in a window or a dialog box in the IDE and press F1.

The online help system is displayed:



Here you can find context-sensitive help about for example:

- The IDE and C-SPY
- The compiler
- The library
- MISRA-C.

# THE IAR C/C++ COMPILER

## Programming languages

For most product packages, there are two high-level programming languages you can use with the IAR C/C++ Compiler:

**C**, the most widely used high-level programming language in the embedded systems industry. You can build freestanding applications that follow these standards:

- Standard C—also known as C99. Hereafter, this standard is referred to as *Standard C* in this guide.
- C89—also known as C94, C90, C89, and ANSI C. This standard is required when MISRA C is enabled.

**C++** (depends on your product package). Any of these standards can be used:

- Standard C++—can be used with different levels of support for exceptions and runtime type information (depends on your product package).
- Embedded C++ (EC++), a subset of the C++ programming standard. It is defined by an industry consortium, the Embedded C++ Technical committee.
- IAR Extended Embedded C++, with additional features such as full template support, multiple inheritance (depends on your product package), namespace support, the new cast operators, as well as the Standard Template Library (STL).

## MISRA C

MISRA C is a set of rules, suited for use when developing safety-critical systems. The rules that make up MISRA C are meant to enforce measures for stricter safety in the ISO standard for the C programming language.

Depending on your product package, there is support for both MISRA C:2004 and MISRA C:1998.

## Compiler extensions

The compiler provides the standard features of the C and C++ languages, as well as a wide range of extensions:

**C language extensions** can be divided in three groups:

- *Extensions for embedded systems programming*—extensions specifically tailored for efficient embedded programming for the specific

microcontroller you are using, typically to meet memory restrictions or to declare special function types such as interrupts.

- *Relaxations to Standard C*—that is, the relaxation of some minor Standard C issues and also some useful but minor syntax extensions.

**Pragma directives** is a mechanism defined by the C standard to be used for vendor-specific extensions to make sure that the source code is still portable. The predefined directives control the behavior of the compiler, for example how it allocates memory, whether it allows extended keywords, and whether it outputs warning messages.

The pragma directives are always enabled in the compiler.

**Preprocessor features**, for example:

- Predefined preprocessor symbols which let you inspect your compile-time environment, for example time of compilation, and different compiler settings.
- User-defined preprocessor symbols defined either by a compiler option or in the IDE, in addition to the `#define` directive.

**Accessing low-level features of the microcontroller** is essential. The compiler supports several ways of doing this: intrinsic functions—which provide direct access to low-level processor operations—mixing C and assembler modules, and inline assembler. You should carefully choose which method to use.

## THE IAR ASSEMBLER

The IAR Assembler is a relocating macro assembler with a versatile set of directives and expression operators for the microcontroller you are using. The assembler features a built-in C language preprocessor and supports conditional assembly.

The assembler translates symbolic assembler language mnemonics into executable machine code. To write efficient assembler applications, you should be familiar with the architecture and instruction set of the microcontroller you are using.

Even if you do not intend to write a complete application in assembler language, there might be situations where you find it necessary to write parts of the code in assembler, for example, when using mechanisms in the microcontroller that require precise timing and special instruction sequences.

## THE IAR LINKER AND RELATED TOOLS

Depending on your product package, IAR Embedded Workbench comes with either the XLINK linker or the ILINK linker.

They are both equally well suited for linking small, single-file, absolute assembler applications as for linking large, relocatable, multi-module, C/C++, or mixed C/C++ and assembler applications.

Both linkers use a *configuration file* where you can specify separate locations for code and data areas of your target system memory map, to give you full control of code and data placement.

Before linking, the linker performs a full dependency resolution of all symbols in all input files, independent of input order (except for libraries). It also checks for consistent compiler settings for all modules and makes sure that the correct version and variant of the C or C++ runtime library is used.

The linker will automatically load only those library modules—user libraries and standard C or C++ library variants—that are actually needed by the application you are linking. More precisely, only the functions of the library module that are actually used will be loaded.

**The IAR ILINK Linker** combines one or more relocatable object files with selected parts of one or more object libraries to produce an executable image.

The final output produced by ILINK is an absolute object file containing the executable image in the ELF (including DWARF for debug information) format. The file can be downloaded to C-SPY or any other debugger that supports ELF/DWARF, or it can be programmed into EPROM after it has been converted to any suitable format.

To handle ELF files various utilities are included, such as an archiver, an ELF dumper, and a format converter.

Depending on your product package, ILINK can also calculate the maximum stack usage for many applications.

**The IAR XLINK Linker** combines one or more relocatable object files produced by the IAR Systems compiler or assembler to produce machine code for the microcontroller you are using. XLINK can generate more than 30 industry-standard loader formats, in addition to UBROF which is used by the C-SPY debugger.

XLINK also performs full C/C++ type checking across all modules.

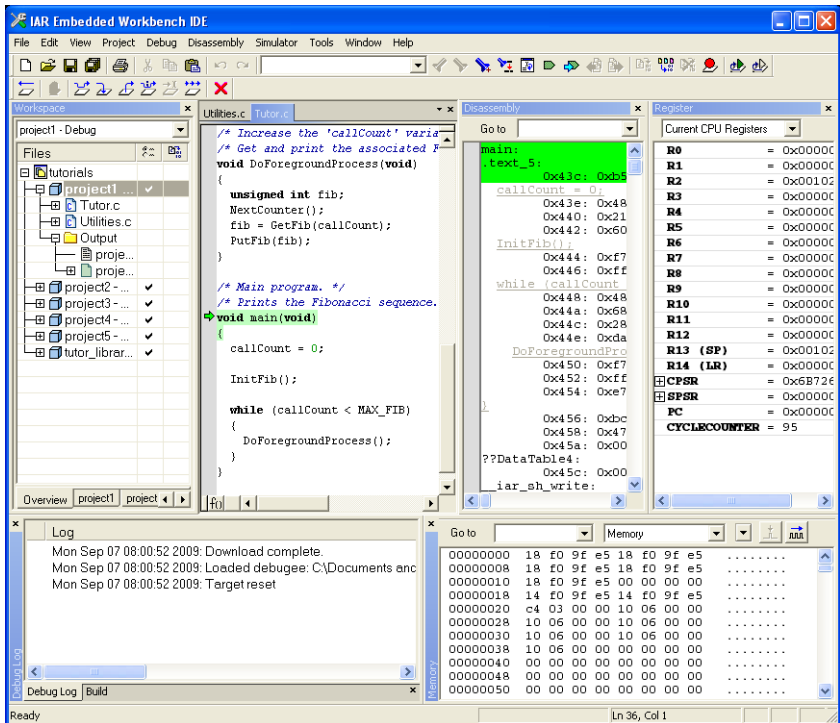
The final output from XLINK is an absolute, target-executable object file that can be downloaded to the microcontroller or to a hardware emulator. Optionally, the output file might contain debug information depending on the output format you choose.

To handle libraries, the library tools XAR and XLIB are included.



## THE IAR C-SPY DEBUGGER

The IAR C-SPY Debugger is a high-level-language debugger for embedded applications development.



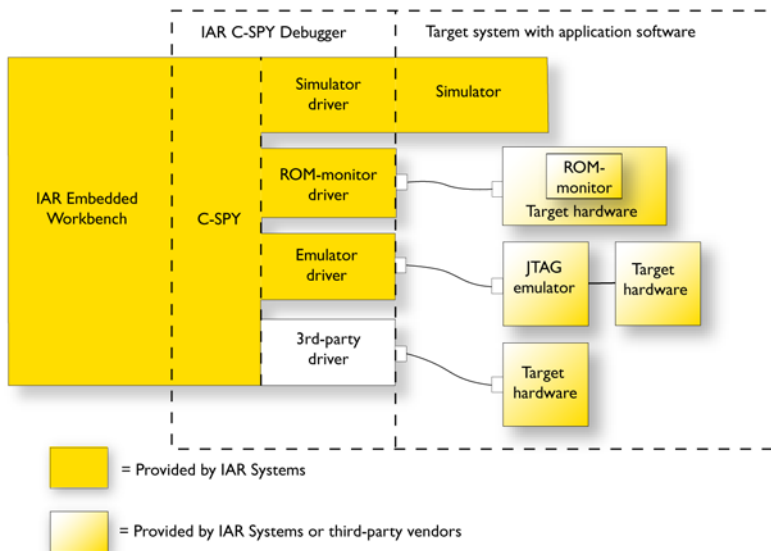
It is designed for use with the IAR Systems compilers and assemblers, and it is completely integrated in the IDE, providing seamless switching between development and debugging. This will give you possibilities such as:

- Editing while debugging. During a debug session, you can make corrections directly in the same source code window that is used to control the debugging. Changes will be included in the next project rebuild.
- Setting source code breakpoints before starting the debugger. Breakpoints in source code will be associated with the same piece of source code even if additional code is inserted.

C-SPY consists both of a general part which provides a basic set of debugger features, and of a driver. The C-SPY driver is the part that provides communication with and control of the target system. The driver also provides a user

interface—special menus, windows, and dialog boxes—to the features that the target system provides, for instance, special breakpoints.

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



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

C-SPY is explored in more detail in this guide, see *Debugging*, page 45.

## C-SPY plugin modules

C-SPY is designed as a modular architecture. An SDK (Software Development Kit) is available for implementing additional functionality to the debugger in the form of plugin modules. These modules can be integrated in the IDE.

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

- Code Coverage, Symbols and the Stack plugin, all well integrated in the IDE.
- The various C-SPY drivers for debugging using certain debug systems.
- RTOS plugin modules for real-time OS-awareness 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 about the C-SPY SDK, contact IAR Systems.



# Developing embedded applications

Before you start developing your embedded application software, you should read about these concepts:

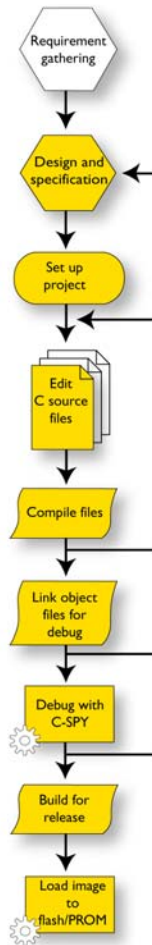
- The development cycle
- Commonly used software models
- The build process
- Programming for performance
- Considering hardware and software factors
- Application execution.

## THE DEVELOPMENT CYCLE

Before the actual development starts you must gather requirements and design and specify your application architecture (manually or using automated code generation tools, such as visualSTATE). Then, you are ready to start the IAR Embedded Workbench IDE.

This is a typical development cycle:

- Set up a project, which includes general and tool-specific options
- Create your source code files in C, C++, or assembler
- Build—compile and link—your project for debugging
- Correct any errors in your source code
- Test and debug your application
- Build for release
- Load the image to flash or PROM memory.



## COMMONLY USED SOFTWARE MODELS

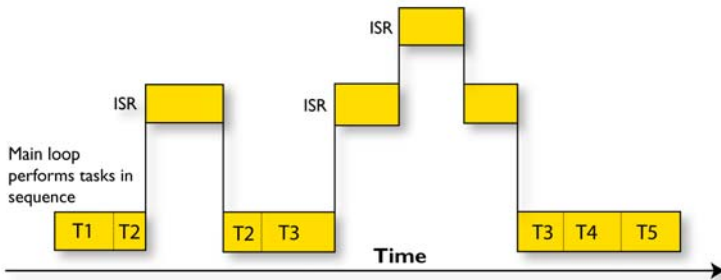
These are some commonly used software models:

- Superloop systems (tasks are performed in sequence)
- Multitask systems (tasks are scheduled by an RTOS)
- State machine models.

Typically, you have either a superloop system or a multitask system, and a popular way of organizing the logic of your application is to design it using state machines.

### Superloop systems

Without a multitasking kernel, only one task can be executed by the CPU at a time. This is called a single-task system or a superloop; basically a program that runs in an endless loop and executes the appropriate operations in sequence. No real-time kernel is used, so interrupt service routines (ISRs) must be used for real-time parts of the software or critical operations (interrupt level).



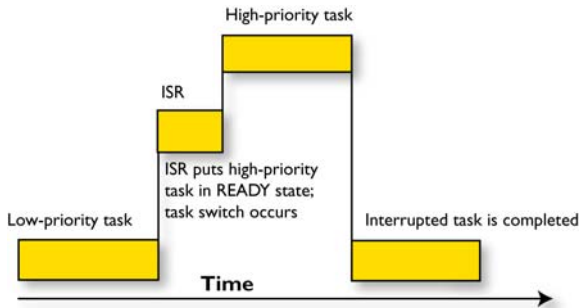
Superloops can become difficult to maintain if the program becomes too large. Because one software component cannot be interrupted by another component (only by ISRs), the reaction time of one component depends on the execution time of all other components in the system. Real-time behavior is therefore poor.

This type of system is typically used if real-time behavior is not critical.

## Preemptive multitasking systems

When a real-time operating system is used, multiple tasks can be executed simultaneously on a single CPU. All tasks execute as if they completely owned the entire CPU. The tasks are scheduled, meaning that the RTOS can activate and deactivate every task. In a multitasking system, there are different scheduling algorithms in which the calculation power of the CPU can be distributed among tasks.

Real-time systems operate with preemptive multitasking. A real-time operating system needs a regular timer interrupt to interrupt tasks at defined times and to perform task switches if necessary. The highest-priority task in the READY state is therefore always executed, whether it is an interrupted task or not. If an interrupt service routine (ISR) makes a higher priority task ready, a task switch will occur and the task will be executed before the interrupted task is returned to.



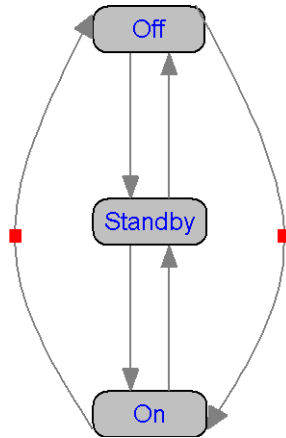
## State machine models

A state machine model simply transforms incoming events to deduced outgoing actions, that is a purely reactive engine or core, not to be confused with an operating system. At any given point in time, the system is in one of several possible states. The system can change states depending on input from the environment. As a state change occurs, actions can be performed on the environment.

For example, an electronic device subsystem can be On or Off, a door can be Open, or it can be ClosedAndUnlocked, etc. A state machine does not have to map all the possible physical states of the problem, only the states that are important to the solution.

One very important feature of a state machine model is its ability to handle concurrency. In this context, the term concurrency refers to the handling of multiple parallel state systems simultaneously.

For example, assume a vending machine and all the cases that must be considered:



- What happens if a cup is removed before it is full?
- What happens to a credit card account if the customer cancels the order while a payment is being processed?
- What happens if a new order is started before the previous order has been completed?
- Will the money be correctly returned to the customer if one of the electromechanical parts causes the machine to stop in the middle of processing an order?

A statechart diagram provides a high-level view of the design that makes it possible to maintain the overview needed to handle the complexity. Once the statechart model has been created, it can be verified to make sure that it behaves as intended. IAR visualSTATE also generates C/C++ source code that is 100% consistent with your design.

The use of state machines is exceptionally beneficial for controlling logic-oriented applications, such as monitoring, metering, and control applications where reliability, size, and deterministic execution are the main criteria.

## THE BUILD PROCESS

This section gives an overview of the build process; how the various build tools—compiler, assembler, and linker—fit together, going from source code to an executable image. The build process can be further divided into:

- The translation process
- The linking process



- After linking.

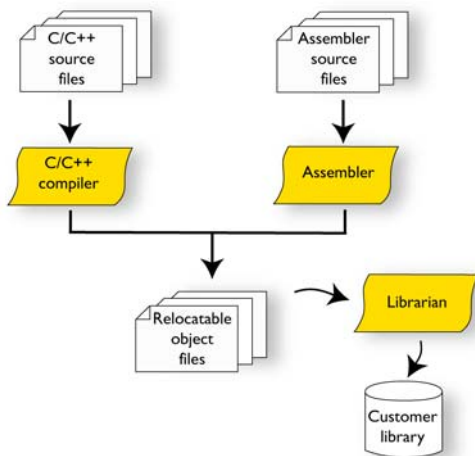
## The translation process

There are two tools in the IDE that translate application source files to intermediary object files: the IAR C/C++ Compiler and the IAR Assembler. Both produce relocatable object files, in ELF/DWARF format for products using ILINK, and in UBROF for products using XLINK.

**Note:** The compiler can also be used for translating C/C++ source code into assembler source code. If required, you can then modify the assembler source code and assemble it into object code.

This figure illustrates the translation process:

After the translation, you can choose to organize your files by packing any number of modules into an archive, or in other words, a library.



## The linking process

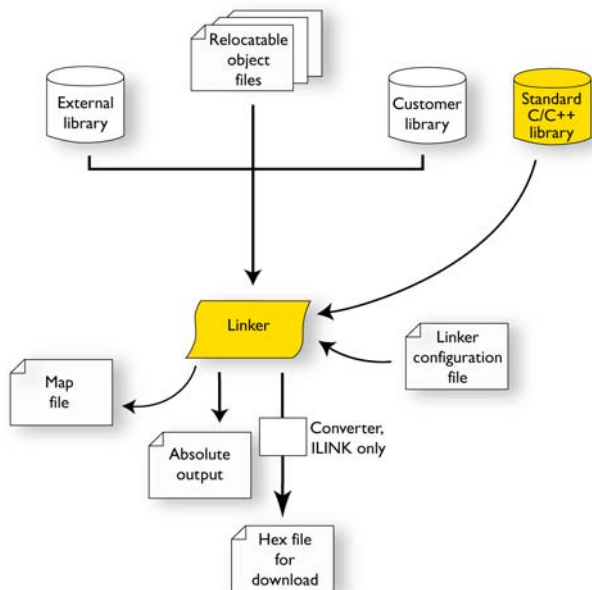
The relocatable modules, in object files and libraries, produced by the IAR compiler and assembler cannot be executed as is. To become an executable application, they must be *linked*.

**Note:** Depending on your product package, modules produced by a toolset from another vendor can be included in the build as well. Be aware that this might also require a compiler utility library from the same vendor.

The linker is used for building the final application. Normally, the linker requires the following information as input:

- Several object files and possibly libraries
- A program start label (set by default in the ILINK linker and user-configurable in the XLINK linker)
- The linker configuration file that describes placement of code and data in the memory of the target system.

This figure illustrates the linking process:



**Note:** The standard C/C++ library contains support routines for the compiler, and the implementation of the C/C++ standard library functions.

The ILINK linker produces an absolute object file in ELF format that contains the executable image. XLINK can generate more than 30 industry-standard loader formats, in addition to the IAR Systems proprietary debug format used by the C-SPY debugger—UBROF.

During the linking, the linker might produce error messages on `stdout` and `stderr`. The ILINK linker also produces log messages, which are useful for understanding why an application was linked the way it was, for example, why a module was included or a section removed.

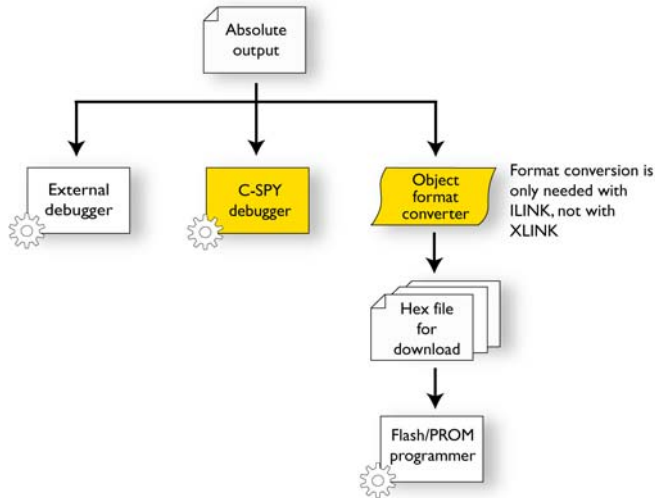
## After linking

After linking, the produced absolute executable image can be used for:

- Loading into the IAR C-SPY Debugger or any other external debugger that reads the produced format.
- Programming a flash/PROM memory using a flash/PROM programmer. When linking using ILINK, before this is possible, the actual bytes in the

image must be converted into the standard Motorola S-record format or the Intel-hex format. XLINK can generate any of these formats directly; thus, no extra conversion is needed.

This figure illustrates the possible uses of the output:



## PROGRAMMING FOR PERFORMANCE

This section provides some hints for:

- Using data types appropriately
- Facilitating register allocation
- Facilitating compiler transformations.

### Using data types appropriately

**Data sizes** should be used appropriately. 8-bit operations are often less efficient on 32-bit CPUs. Conversely, 32-bit operations are inefficient for 8-bit CPUs. If you use integer constants, make sure to add appropriate suffixes, for example `36L`.

**Signed values** means that negative values can be used. However, when the compiler modifies such values, arithmetic operations are used. For *unsigned values*, the compiler instead uses bit and shift operations which are usually cheaper than arithmetic operations. If you do not need negative values, make sure to use unsigned types.

**Floating-point operations** are usually very expensive as they might require large library functions. Consider replacing such operations with integer operations (which are more efficient).

**Memory placement and pointer types** on 8- and 16-bit architectures become more efficient and generate less code if you strive for: small memory areas, small addresses, and small pointers. Avoid using the largest memory types or pointers.

**Casting** to and from pointers should be avoided, as well as mixed types in expressions. This generates inefficient code and there is a risk for information loss.

**Padding in structures** occurs when the CPU requires alignment. To avoid this memory waste, order fields by size to ensure that the amount of memory used for padding is reduced to a minimum.

## **Facilitating register allocation**

**Function parameters and local variables** (as opposed to global variables) reduces memory consumption because they can be placed in registers and only need to exist while they are in scope. The use of global variables introduces overhead because they must be updated whenever a function that accesses them is called.

**Variable arguments** (printf-style) should be avoided, because they force arguments to the stack. These arguments would otherwise be passed in registers.

## **Facilitating compiler transformations**

**Function prototypes** should be used because that makes it easier to find problems in the source code as type promotion (implicit casting) is not needed. Prototyping also makes it easier for the compiler to generate more efficient code.

**Static-declared variables and functions** should only be used in the file or module where they are declared, to achieve the best optimizations.

**Inline assembler** might be a major obstacle for the compiler when it optimizes the code. Make sure to check how the `asm` keyword works in the compiler you are using.

**“Clever” source code** should be replaced with clear code, as clear code is easier to maintain, less likely to contain programming errors, and usually much easier for the compiler to optimize.

**The volatile keyword** should be used for protecting simultaneously accessed variables, that is, variables accessed asynchronously by, for example, interrupt routines or code executing in separate threads. The compiler will then always read from and write to memory when such variables are accessed.

**Empty loops**, that is, code that has no effect other than to achieve delays, might be removed by the compiler. Instead, use OS services, intrinsic functions, CPU timers, or access `volatile` declared variables.

**Long basic blocks** should be created if possible. A basic block is an uninterrupted sequence of source code with no function calls. This facilitates more efficient register allocation and better optimization results.

## CONSIDERING HARDWARE AND SOFTWARE FACTORS

Typically, embedded software written for a dedicated microcontroller can be designed as an endless loop waiting for some external events to happen. The software is located in ROM and executes on reset. You must consider several hardware and software factors when you write this kind of software.

### CPU features and constraints

The features available in the microcontroller you are using, for example instruction set interworking, different processor modes, and alignment constraints need to be fully understood. To configure them correctly, it is important to read and understand the hardware documentation.

The compiler supports such features by means of, for example, extended keywords, pragma directives, and compiler options.

When you set up your project in the IDE, you must select a device option that suits the device you are using. This selection will automatically:

- Set the CPU-specific options to match the device you are using
- Determine the default linker configuration file (depending on your product package)

Depending on your product package, the `target\config` directory contains either templates for linker configuration files, or ready-made linker configuration files for some or all supported devices. The files have the filename extension `xcl` or `icf`, for XLINK and ILINK respectively.

- Determine the default device description file

These files are located in the `target\config` directory and have the filename extension `ddf` or `svd` (depending on your product package).

## Mapping internal and external memory

Embedded systems typically contain various types of memory, such as on-chip RAM, external DRAM or SRAM, ROM, EEPROM, or flash memory.

As an embedded software developer, you must understand the features of the different memory types. For example, on-chip RAM is often faster than other types of memories, and variables that are accessed often would in time-critical applications benefit from being placed here. Conversely, some configuration data might be accessed seldom but must maintain its value after power off, so it should be saved in EEPROM or flash memory.

For efficient memory usage, the compiler provides several mechanisms for controlling placement of functions and data objects in memory. The linker places code and data in memory according to the directives you specify in the linker configuration file.

## Communication with peripheral units

If external devices are connected to the microcontroller, you might need to initialize and control the signalling interface, for example by using chip select pins, and detecting and handling external interrupt signals. Typically, this must be initialized and controlled at runtime. The normal way to do this is to use special function registers (SFRs). These are typically available at dedicated addresses, containing bits that control the chip configuration.

Standard peripheral units are defined in device-specific I/O header files with the filename extension `h`, located in the `target\inc` directory. Make sure to include the appropriate include file in your application source files. If you need additional I/O header files, they can be created using one of the provided files as a template.

## Interrupt handling

In embedded systems, using *interrupts* is a method for handling external events immediately; for example, detecting that a button was pressed. In general, when an interrupt occurs in the code, the microcontroller stops executing the code that currently is running and starts executing an interrupt routine instead.

The compiler supports processor exception types with dedicated keywords, which means that you can write your interrupt routines in C.

## System startup

In all embedded systems, system startup code is executed to initialize the system—both the hardware and the software system—before the `main` function of the application is called.

As an embedded software developer, you must ensure that the startup code is located at the dedicated memory addresses, or can be accessed using a pointer from the vector table. This means that startup code and the initial vector table must be placed in non-volatile memory, such as ROM, EPROM, or flash memory.

A C/C++ application must initialize all global variables. This initialization is handled by the linker and the system startup code in conjunction. For more information, see *Application execution*, page 32.

## The runtime libraries

Depending on your product package, either one of the following two libraries is provided or both. You must choose which library to use:

- The IAR DLIB Library, which supports Standard C and C++. This library also supports floating-point numbers in IEEE 754 format and it can be configured to include different levels of support for locale, file descriptors, multibyte characters, etc.
- The IAR CLIB Library is a light-weight library, which is not fully compliant with Standard C. Neither does it fully support floating-point numbers in IEEE 754 format or does it support Embedded C++. If the legacy CLIB library is provided, it is for backward compatibility. It should not be used for new application projects.

**Note:** Note that if your project only contains assembler source code, you do not need to choose a runtime library.

The runtime library is delivered as prebuilt libraries which are built for different project configurations. The IDE automatically uses the library that matches your project configuration. Depending on your product package, there might not be a prebuilt library for the configuration that you are using. In that case you must build a library yourself.

Depending on your product package, the library might also be delivered as source files, and you can find them in the directory `target\src\lib`. This means that you can customize the library and build it yourself. The IDE provides a library project template that you can use for building your own library version.

## The runtime environment

The runtime environment is the environment in which your application executes. This environment depends on the selected runtime library, target hardware, the software environment, and the application source code.

To configure the most code-efficient runtime environment, you must determine your application and hardware requirements. The more functionality you need, the larger your code will become.

To get the required runtime environment, you might want to customize it by:

- Setting library options, for example, for choosing `scanf` input and `printf` output formatters.
- Specifying the size of the stack (or stacks if there are several, which depends on your microcontroller).

Depending on the microcontroller, you must also specify whether non-static auto variables should be placed on the stack or in a static overlay area. The stack is dynamically allocated at runtime, whereas the static overlay area is statically allocated at link time.

- Specifying the size of the heap and where in memory it should be placed. Depending on your product package, you can also use more than one heap, and place the heaps in different memory areas.
- Overriding certain library functions, for example `cstartup`, with your own customized versions.
- Choosing the level of support for certain standard library functionality, for example, locale, file descriptors, and multibyte characters, by choosing a *library configuration*: Normal or Full (only possible for the DLIB library). You can also make your own library configuration, but that requires that you *rebuild* the library. This gives you full control of the runtime environment.

To run the application on hardware, you must implement low-level routines for character-based input and output (typically, `putchar` and `getchar` for CLIB, and `__read` and `__write` for DLIB).

## APPLICATION EXECUTION

This section gives an overview of how the execution of an embedded application is divided into three phases:

- Initialization
- Execution
- Termination.



## The initialization phase

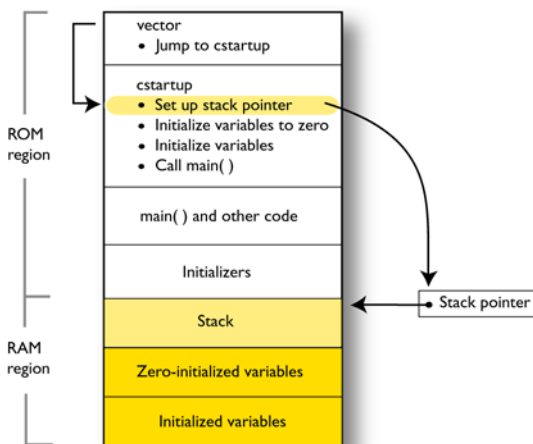
Initialization is executed when an application is started (the CPU is reset) but before the `main` function is entered. The initialization phase can, somewhat simplified, be divided into:

- Hardware initialization, for example initializing the stack pointer  
The hardware initialization is typically performed by the system startup code `cstartup` and if required, by an extra low-level routine that you provide. It might include resetting/starting the rest of the hardware, setting up the CPU, etc, in preparation for the software C/C++ system initialization.
- Software C/C++ system initialization  
Typically, this includes making sure that every global (statically linked) C/C++ object receives its proper initialization value before the `main` function is called.
- Application initialization  
This depends entirely on your application. Typically, it can include setting up an RTOS kernel and starting initial tasks for an RTOS-driven application. For a bare-bone application, it can include setting up various interrupts, initializing communication, initializing devices, etc.

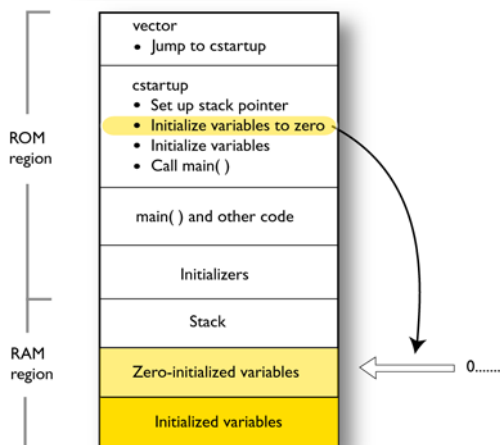
For a ROM/flash-based system, constants and functions are already placed in ROM. All symbols placed in RAM must be initialized before the `main` function is called. The linker has already divided the available RAM into different areas for variables, stack, heap, etc.

The following sequence of figures gives a simplified overview of the different stages of the initialization. Note that the memory layout is generalized in these figures.

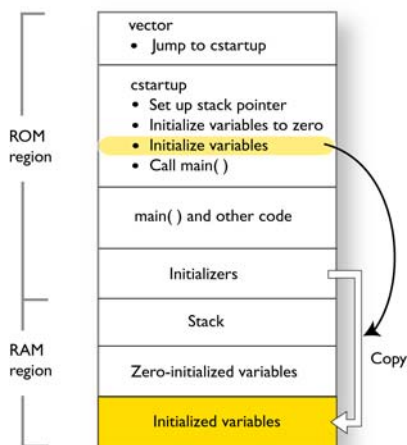
- 1 When an application is started, the system startup code first performs hardware initialization, such as initialization of the stack pointer to point at either the start or the end—depending on your microcontroller—of the predefined stack area.



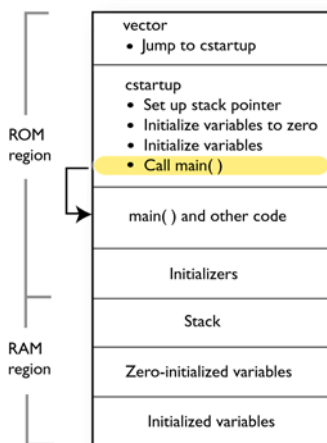
- 2 Then, memories that should be zero-initialized are cleared, in other words, filled with zeros. Typically, this is data referred to as *zero-initialized data*; variables declared as, for example, `int i = 0;`



- 3 For *initialized data*, data declared with a non-zero value, like `int i = 6;`, the initializers are copied from ROM to RAM.



- 4 Finally, the `main` function is called.



## The execution phase

The software of an embedded application is typically implemented as a loop which is either interrupt-driven or uses polling for controlling external interaction or internal events. For an interrupt-driven system, the interrupts are typically initialized at the beginning of the `main` function.

In a system with real-time behavior and where responsiveness is critical, a multi-task system might be required. This means that your application software should be supplemented with a real-time operating system. In this case, the RTOS and the different tasks must also be initialized at the beginning of the `main` function.

### **The termination phase**

Typically, an embedded application should never stop executing. If it does, you must define a proper end behavior.

To terminate an application in a controlled way, either call one of the standard C library functions `exit`, `_Exit`, or `abort`, or return from `main`. If you return from `main`, the `exit` function is executed, which means that C++ destructors for static and global variables are called (C++ only) and all open files are closed.

# Creating an application project

This chapter demonstrates a development cycle for setting up your application project in the IDE. Typically, the cycle consists of these steps:

- Creating a workspace
- Creating a new project
- Setting project options
- Adding source files to the project
- Setting tool-specific options
- Compiling
- Linking.

If you instead want to work through one of the step-by-step tutorials, you can access them from the Information Center available from the Help menu.

## CREATING A WORKSPACE

Choose **File>New>Workspace** to create a *workspace* to which you can add one or several *projects*. An empty workspace window appears.

**Note:** When you start the IDE for the first time, there is a ready-made workspace, which you can use for your project instead.

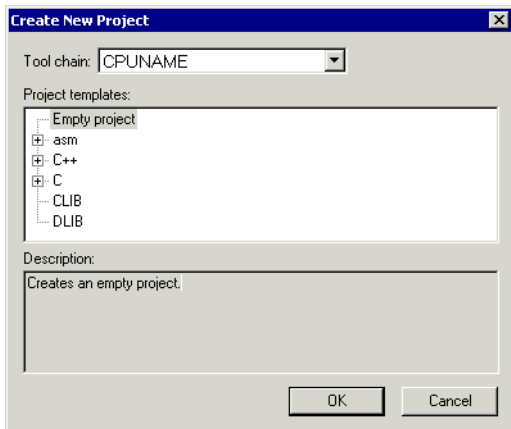
Now you are ready to create a project and add it to the workspace.

Examples for getting started are available to give you a smooth start. Depending on your product package, there are from a few to several hundreds of working source code examples where the complexity ranges from simple LED blink to USB mass storage controllers. Depending on your product package, there are examples for most of the supported devices. You can access the examples via the Information Center, available from the **Help** menu.

## CREATING A NEW PROJECT

- 1 Choose **Project>Create New Project**.

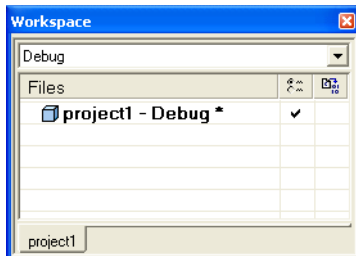
From the **Tool chain** drop-down list, choose the toolchain you are using. If you have the IDE installed for several microcontrollers, they will all appear in the drop-down list.



In the list of project templates, select a template to base your new project on. For example, select **Empty project**, which simply creates an empty project that uses default project settings.

- 2 Save your project.
- 3 The project will appear in the Workspace window.

By default, two *build configurations* are created—Debug and Release—which let you define variants of your project (project settings and files part of the build). You can also define your own build configurations. You choose configuration from the drop-down menu at the top of the window.



- 4 Before you add any files to your project, you should save the workspace.

Project-related files have now been created:

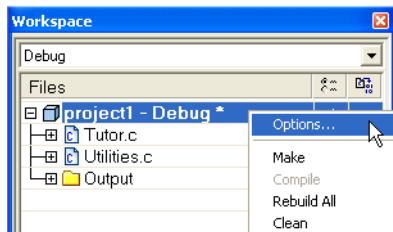
- A workspace file with the filename extension *eww*. This file lists all projects that you have added to the workspace.
- Project files with the filename extensions *ewp* and *ewd*. These files contain information about your project-specific settings, such as build options.

- Information related to the current session, such as the placement of windows and breakpoints, is located in the files created in the `projects\settings` directory.

## SETTING PROJECT OPTIONS

To set options that must be the same for the whole build configuration:

- 1 Select the project folder icon in the Workspace window, right-click, and choose **Options**.
- 2 The **General Options** category provide options for target, output, library, and runtime environment. The settings you make here must be the same for the whole build configuration.

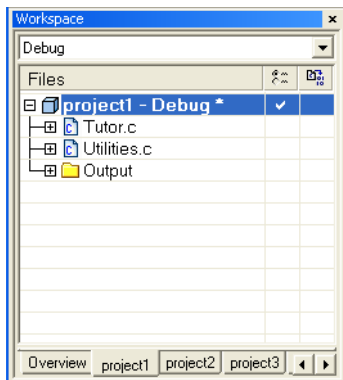


Note specifically that your choice of device on the **Target** page will automatically determine the default debugger device description file, and depending on your product package, also the default linker configuration file. In addition, other options will be set automatically to suit the selected device.

## ADDING SOURCE FILES TO THE PROJECT

- 1 In the Workspace window, select the destination to which you want to add a source file—a group or, as in this case, directly to the project.
- 2 Choose **Project>Add Files** to open a standard browse dialog box. Locate the files and click **Open** to add them to your project.

You can create several *groups* of files to organize your source files logically according to your project needs.



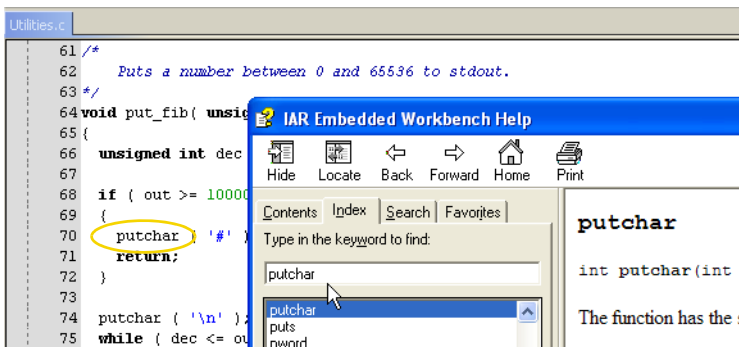
**To create a new document:**



Click **New Document** on the toolbar. The file is displayed in the editor window. You can create or open one or several text files, and if you open several files, they are organized in a *tab group*. Several editor windows can be open at the same time.

## To look up a function reference:

In the editor window, select the item for which you want help and press F1. The online help system is displayed.



In the editor window, you can get help for any C or Embedded C++ library function, and for any compiler language extension, such as keywords, intrinsic functions etc.

## To configure the editor:

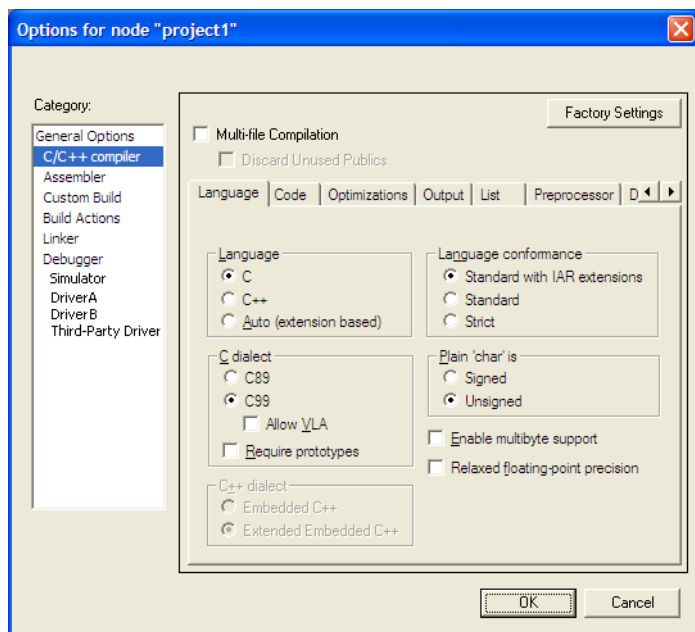
Choose **Tools>Options** and select the appropriate category of options in the **IDE Options** dialog box.

## SETTING TOOL-SPECIFIC OPTIONS

- In the Workspace window, select the project, a group of files, or an individual file. Choose **Project>Options** to open the **Options** dialog box.



- 2 Select a tool in the **Category** list, and make your settings on the appropriate pages. Note that the tools available in the list depend on your product package.

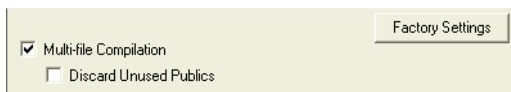


In addition to the standard tools part of the toolchain, you can set options for prebuild and postbuild actions and invoke external tools.

Before you set specific compiler options, you can decide if you want to use multi-file compilation. If

the compiler compiles multiple source files in one invocation, it can in many cases optimize more efficiently. However, this might affect the build time. Thus, it can be advisable to disable this option during the development phase of your work.

**Note:** If your product package does not support multi-file compilation, the **Multi-file Compilation** option is not available.



## COMPILING

### To compile one or several files:

- 1 Select the file in the Workspace window or click the editor window that displays the file you want to compile.
- 2 Click the **Compile** button on the toolbar.



Alternatively, use any of these commands available from the Project menu:

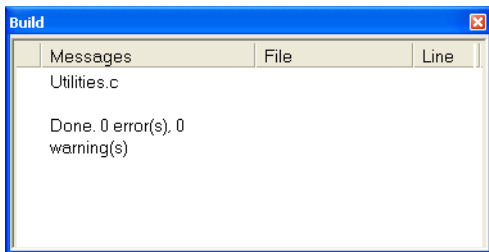


**Make**—brings the current build configuration up to date by compiling, assembling, and linking only the files that have changed since the last build.

**Rebuild All**—rebuilds and relinks all files in the active project configuration.

**Batch Build**—displays a dialog box where you can configure named batch build configurations, and build a named batch.

- 3 If any source code errors are generated, switch to the correct position in the appropriate source file by double-clicking the error message in the Build window.



- 4 After you have compiled or assembled one or more files, the IDE has created new directories and files in your project directory. If the name of your build configuration is **Debug**, a directory with the same name has been created containing these directories:

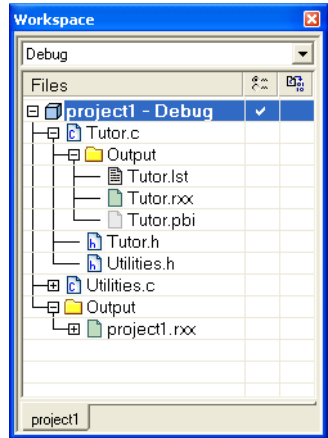
- **List**—the destination directory for the list files, which have the extensions `lst`, `map`, and `log`.
- **Obj**—the destination directory for the object files from the compiler and the assembler. These files are used as input to the linker and their extension is `rnn` (where `nn` depends on your product package) for products with **XLINK**, and `o` for products with **ILINK**.
- **Exe**—the destination directory for the executable file. It is used as input to **C-SPY** and its extension is `dnn` (where `nn` depends on your product package) for products with **XLINK**, and `out` for products with **ILINK**. Note that this directory is empty until you have linked the object files.

## To view the result in the Workspace window:

After compiling, click the plus icons in the Workspace window to expand the view.

As you can see, the IDE has also created an output folder icon in the Workspace window containing any generated output files. All included header files are displayed as well, showing the dependencies between the files.

Note that the filename extensions on the generated files depend on your product package.



## LINKING

- 1 Select the project in the Workspace window, right-click and choose **Options** from the context menu. Then select **Linker** in the **Category** list to display the linker option pages.
- 2 After you made your settings, choose **Project>Make**. The progress will be displayed in the Build messages window. The result of the linking is an output file that contains debug information (if you built with debug information).

When setting linker options, pay attention to the choice of output format, linker configuration file, and the map and log files.

## Output format

The **XLINK linker** can produce a number of formats. It is important to choose the output format that suits your purpose. You might want to load your output to a debugger—which means that you need output with debug information.

Alternatively, in your final application project, you might want to load the output to a PROM programmer—in which case you need an output format supported by the programmer, such as Intel-hex or Motorola S-records.

The **ILINK linker** produces an output file in the ELF format, including DWARF for debug information. If you need to use the Motorola or Intel-standard format instead, for example to load the file to a PROM memory, you must convert the file. Choose the **Converter** category in the **Options** dialog box and set the appropriate options.

## Linker configuration file

Program code and data are placed in memory according to the configuration specified in the linker configuration file (filename extension `icf` for ILINK and `xc1` for XLINK). It is important to be familiar with its syntax for how sections are placed in memory.

Depending on your product package, the `target\config` directory contains either templates for linker configuration files, or ready-made linker configuration files for some or all supported devices. You can use the files or templates supplied with the product as they are with the C-SPY simulator, but when you use them for your target system, you must adapt them to your actual hardware memory layout.

To examine the linker configuration file, use a text editor, such as the IAR Embedded Workbench editor, or print a copy of the file, and verify that the definitions match the requirements of your hardware memory layout.

## Linker map and log files file

XLINK and ILINK can both generate extensive listings:

- XLINK can generate a map file which optionally contains a segment map, symbol listing, module summary, etc
- ILINK can generate a map file, which typically contains a placement summary. ILINK can also generate a log file, which logs decisions made by the linker regarding initializations, module selections, section selections etc.

Typically, this information can be useful if you want to examine:

- How the segment/sections and code were placed in memory
- Which source files that actually contributed to the final image
- Which symbols that were actually included and their values
- Where individual functions were placed in memory.

# Debugging

By exploring some of the C-SPY debugger features, this chapter shows their capabilities and how to use them:

- Setting up for debugging
- Starting the debugger
- Executing your application
- Inspecting variables
- Monitoring memory and registers
- Using breakpoints
- Viewing terminal I/O
- Analyzing your application's runtime behavior.

Note that, depending on the product package you have installed, C-SPY might or might not be included.

Depending on your hardware, additional features not explored here might be available in the C-SPY driver you are using. Typically, this applies to setting different types of watchpoints, additional breakpoint types, various triggering systems, more complex trace systems etc.

## SETTING UP FOR DEBUGGING

- 1 Before starting 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 review your settings.
- 3 When you have made your C-SPY settings, click **OK**.
- 4 Choose **Tools>Options>Debugger** to configure:
  - The debugger behavior
  - The debugger's tracking of stack usage.

## Setting up the hardware before C-SPY starts

You can use C-SPY macros to initialize target hardware before C-SPY starts. For example, 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 the application you want to debug 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. Select the option **Use Setup file** and choose the macro file you just created. Your startup macro will now be loaded during the C-SPY startup sequence.

## STARTING THE DEBUGGER

To start the debugger, you can either:



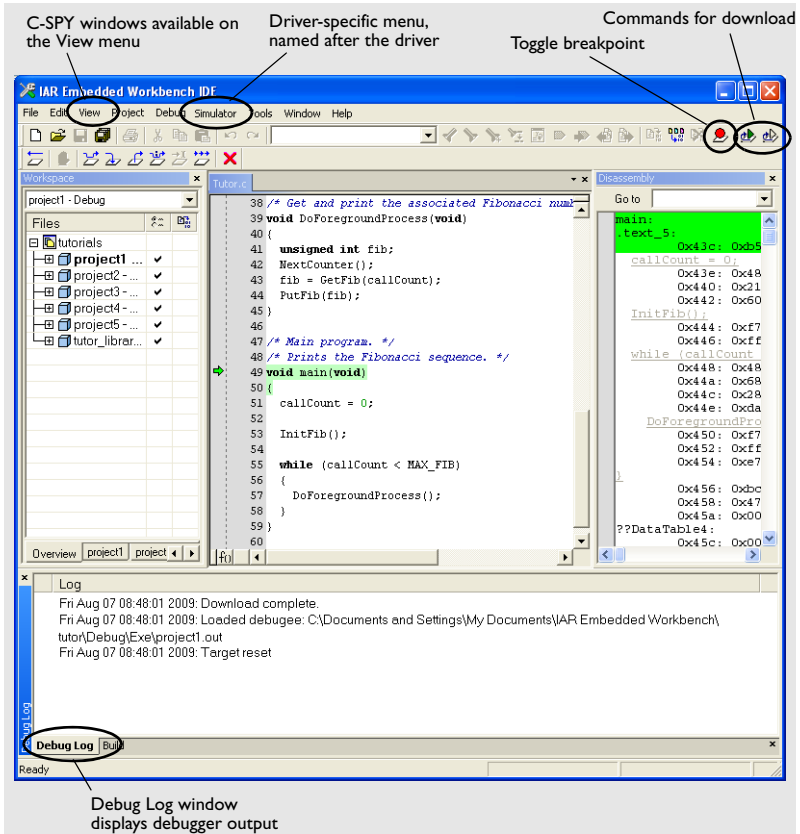
**Download and Debug** starts C-SPY and loads the current project to the target system.



**Debug without Downloading** starts C-SPY without reloading the current project to the target system. It is assumed that the code image is already on the target and therefore this command is not applicable to the simulator.

You can load multiple debug files (images) to the target system. To load an additional debug file in the IDE, choose **Project>Options>Debugger>Images**. This means that the complete program consists of several images. For example, your application (one image) is started by a bootloader (another image). The application image and the bootloader are built using separate projects and generate separate output files.

C-SPY starts with the application loaded.



C-SPY must read from the target system to update the contents of the windows (for windows that need to be updated, for example the Memory and Trace windows). This affects the response time while debugging. If you have several windows open at the same time and the response time is too long (especially if your application executes on hardware), just close one or two windows to reduce the response time.

### To exit from C-SPY:



Click the **Stop Debugging** button on the **Debug** toolbar.

## EXECUTING YOUR APPLICATION

You can find commands for executing on the **Debug** menu and on the **Debug** toolbar, such as:



**Step Over** executes the next statement, function call, or instruction, without entering C/C++ functions or assembler subroutines.



**Step Into** executes the next statement or instruction, entering C/C++ functions or assembler subroutines.



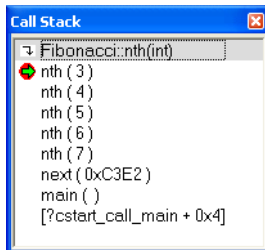
**Next Statement** executes directly to the next C/C++ statement without stopping at individual function calls.

You can also find commands like **Go**, **Break**, **Reset**, **Run to Cursor**, **Autostep**, etc on the menu and the toolbar.

C-SPY allows more stepping precision than most other debuggers because it is not line-oriented but statement-oriented, due to *step points*. The possibility of stepping into an individual function call that is part of a more complex statement is particularly useful when you use C source code that contains 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.

### To inspect function calls:

- 1 Choose **View>Call Stack** to open the Call Stack window. It displays the C/C++ function call stack with the current function at the top. Double-click on any function, and the contents of all affected windows in the IDE are updated to display the state of that particular call frame.



Typically, this is useful for two purposes:

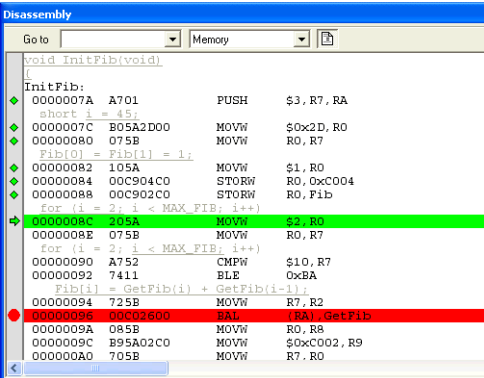
- Determining in what context the current function has been called
- Tracing the origin of incorrect values in variables or parameters, thus locating the function in the call chain where the problem occurred.



## To debug in disassembly mode:

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

Disassembly mode lets you execute the application exactly one assembler instruction at a time. C/C++ mode, on the other hand, executes your application one statement or function at a time. Regardless of which mode you are debugging in, you can display registers and memory, and change their contents.



```
Disassembly
Go to: [ ] Memory: [ ]
void InitFib(void)
{
InitFib:
◆ 0000007A A701 PUSH $3, R7, RA
  short i = 1;
◆ 0000007C B05A2D00 MOVW $0x2D, R0
◆ 00000080 075B MOVW R0, R7
  Fib[0] = Fib[1] = 1;
◆ 00000082 105A MOVW $1, R0
◆ 00000084 00C904C0 STORW R0, 0xC004
◆ 00000088 00C902C0 STORW R0, Fib
  for (i = 2; i < MAX_FIB; i++)
◆ 0000008C 305A MOVW $1, R0
◆ 0000008E 075B MOVW R0, R7
  for (i = 2; i < MAX_FIB; i++)
00000090 A752 CMPW $10, R7
00000092 7411 BLE 0xB
  Fib[i] = GetFib(i) + GetFib(i-1);
00000094 725B MOVW R7, R2
◆ 00000096 00C92600 DAL, #A GetFib
0000009A 085B MOVW R0, R8
0000009C B95A02C0 MOVW $0xC002, R9
000000A0 705B MOVW R7, R0
}
```

## To switch modes:

Use the mouse pointer to make either the editor window or the Disassembly window active, depending on which mode you want to use.

## To view code coverage information:

Right-click in the Disassembly window and choose **Code Coverage>Enable** and then **Code Coverage>Show** from the context menu. Code that has been executed is marked with green diamonds. See also *Code coverage*, page 56.

## INSPECTING VARIABLES

C-SPY allows you to watch variables or expressions in the source code, so that you can keep track of their values as you execute your application. You can look at a variable in several ways:

**Tooltip watch** provides the simplest way of viewing the value of a variable or more complex expressions in the editor window. Just point at the variable with the mouse pointer. The value is displayed next to the variable.

**The Locals window**, available from the **View** menu, automatically displays the local variables, that is, auto variables and function parameters for the active function.

**The Watch window**, available from the **View** menu, allows you to monitor the values of C-SPY expressions and variables of your choice.

**The Live Watch window**, available from the **View** menu, 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. Note that this window requires that the target system supports reading memory during program execution.

**The Statics window**, available from the **View** menu, automatically displays the values of variables with static storage duration. In addition, you can make your own selection of such variables to be displayed.

**The Auto window**, available from the **View** menu, displays an automatic selection of variables and expressions in, or near, the current statement.

**The Quick Watch window**, provides a fast method where you have precise control of when to evaluate or watch the value of a variable or an expression.

**Collecting trace data**, available from the driver-specific menu, can collect a sequence of events in the target system, typically executed machine instructions. Depending on your target system, additional types of trace data can be collected. For example, read and write accesses to memory, and the values of C-SPY expressions. See also *Trace*, page 59.

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

You can add, modify, and remove expressions, and change the display format. A context menu is available with commands for operations in all windows. Drag-and-drop between windows is supported where applicable.

### **To inspect the value of a variable:**

- 1 For example, choose **View>Watch** to open the Watch window.
- 2 To select a variable, follow this procedure:
  - Click the dotted rectangle in the Watch window.
  - In the entry field that appears, type the name of the variable and press the Enter key.
  - You can also drag a variable from the editor window to the Watch window.

In this example, the Watch window shows the current value of the variable `i` and the array `Fib`. You can expand the `Fib` array to watch it in more detail.

Expression	Value	Location	Type
i	5	0x7	short
Fib	<array>	Memory:0xC002	unsigned int[10]
[0]	1	Memory:0xC002	unsigned int
[1]	1	Memory:0xC004	unsigned int
[2]	2	Memory:0xC006	unsigned int
[3]	3	Memory:0xC008	unsigned int
[4]	5	Memory:0xC00A	unsigned int
[5]	0	Memory:0xC00C	unsigned int
[6]	0	Memory:0xC00E	unsigned int
[7]	0	Memory:0xC010	unsigned int
[8]	0	Memory:0xC012	unsigned int
[9]	0	Memory:0xC014	unsigned int

- 3 To remove a variable from the Watch window, select it and press the Delete key.

## 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—in C-SPY referred to as a *memory zone*—and allows you to edit it. Colors are used for indicating data coverage (depends on your product package) and how your application executes. 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 Symbolic memory window** displays how variables with static storage duration are laid out in memory. This can be useful for a better understanding of memory usage or for investigating problems caused by variables being overwritten, for example by buffers that exceed their limits.

**The Stack window** displays the contents of the stack, including how stack variables are laid out in memory. For more details, see *Stack usage*, page 58.

**The Register window** gives an up-to-date display of the contents of the processor registers and SFRs, and allows you to edit them.

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.

## USING BREAKPOINTS

Depending on the C-SPY driver you are using, you can set various kinds of breakpoints:

**Code breakpoints** are used for code locations to investigate whether your program logic is correct or to get trace printouts.

**Log breakpoints** provide a convenient way to add trace printouts without having to add any code to your application source code.

**Trace Start and Stop breakpoints** start and stop trace data collection—a convenient way to analyze instructions between two execution points. See also *Trace*, page 59.

**Data breakpoints** are triggered for read or write memory accesses. Typically, data breakpoints are used for investigating how and when the data changes.

In addition to these breakpoints, the C-SPY driver might support more complex or other breakpoints or triggers of different kinds, depending on the debugging system you are using.

### To set a breakpoint:

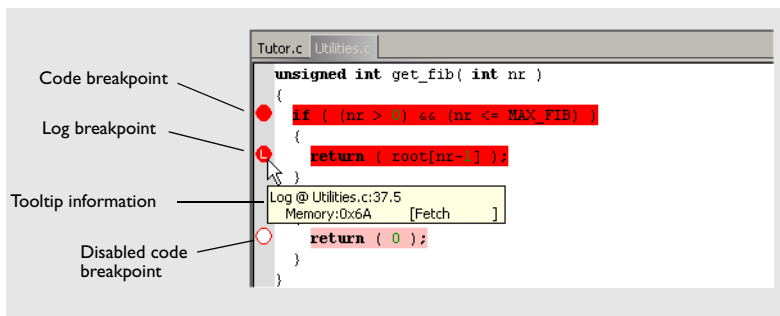


Position the insertion point in the left-side margin, or in or near a statement and double click to toggle a code breakpoint.

Alternatively, use the **Breakpoints** dialog box available from the context menus in the editor window, Breakpoints window, and the Disassembly window. The dialog box gives you a more fine-grained way to set different types of breakpoints and edit them.

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

A breakpoint is marked with an icon in the left margin of the editor window:



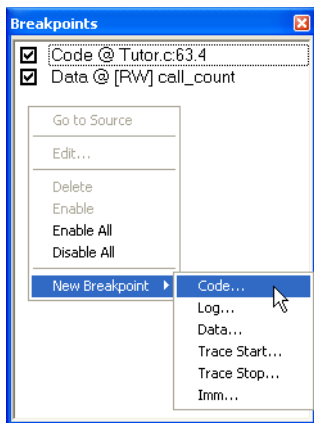
If the breakpoint icon does not appear, make sure the option **Show bookmarks** is selected in the **IDE Options>Editor** dialog box.



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.

### To view all defined breakpoints:

Choose **View>Breakpoints** to open the Breakpoints window, which lists all breakpoints. Here you can conveniently monitor, enable, and disable breakpoints; you can also define new breakpoints, and modify and delete existing breakpoints.



## To investigate breakpoint consumers:

Open the Breakpoints Usage window—available from the C-SPY driver-specific menu—to get a low-level view of all breakpoints, both the ones you have defined and the ones used internally by C-SPY.

Usually, target hardware has a limited amount of hardware breakpoints (used by C-SPY to set breakpoints), sometimes as few as one or two. Exceeding the number of available hardware breakpoints will force the debugger to single step while executing. This will significantly reduce the execution speed.



In a hardware debugger system with limited number of hardware breakpoints, use the Breakpoint Usage window to:

- Identify all consumers of breakpoints
- Check that the number of active breakpoints is supported by the target system
- Configure the debugger to use the available breakpoints in a better way, if possible.

## To execute up to a breakpoint:



- 1 Click the **Go** button on the toolbar.

The application will execute up to the next set breakpoint. The Debug Log window will contain information about the breakpoint triggering.

- 2 Select the breakpoint, right-click and choose **Toggle Breakpoint (xxx)** from the context menu to remove a breakpoint.

## VIEWING TERMINAL I/O

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

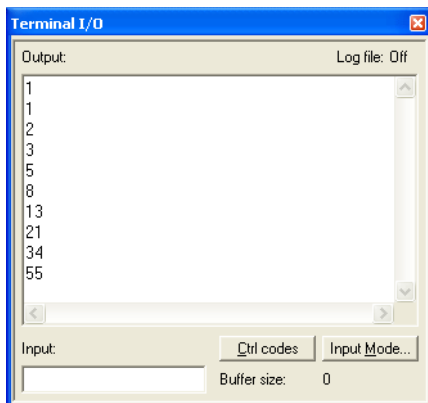
### To use the Terminal I/O window:

- 1 Build your application using these options:

Category	Setting
Linker>Config (for XLINK)	With I/O emulation modules
General Options>Library Configuration (for ILINK)	Library low-level interface implementation

This means that some low-level routines are linked that direct `stdin` and `stdout` to the Terminal I/O window.

- 2 Build your application and start C-SPY.
- 3 Choose **View>Terminal I/O** to open the Terminal I/O window, which displays the output from the I/O operations.



## ANALYZING YOUR APPLICATION'S RUNTIME BEHAVIOR

C-SPY provides various features that you can use to analyze your application's runtime behavior, to locate any bottlenecks and verify that all parts of your application have been tested:

- Profiling
- Code coverage
- Stack usage
- Trace.

### Profiling

You can choose between two profiling variants:

**Function profiling** can help you find the functions where most of the execution time is spent. Those functions are the parts you should focus on when you optimize your code. A simple method of optimizing a function is to compile it using speed optimization. Alternatively, you can move the function into the memory that uses the most efficient addressing mode.

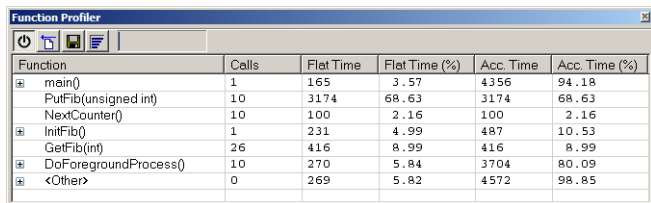
**Instruction profiling** information—that is, the number of times each instruction has been executed—is displayed in the Disassembly window and can help you fine-tune your code on a very detailed level, especially assembler source code.

## To use profiling:

- 1 Build your application using these options:

Category	Setting
C/C++ Compiler	Output>Generate debug information
Linker (for XLINK)	Format>Debug information for C-SPY
Linker (for ILINK)	Output>Include debug information in output

- 2 Build your application and start C-SPY.
- 3 Before you can use profiling, you must set it up. The setup varies with the C-SPY drivers and the target system.
- 4 To open the Function Profiler window, choose **Profiling** from the driver-specific menu.
- 5 Click the **Enable** button to turn on the profiler.
- 6 Start executing your application to collect the profiling information.
- 7 Profiling information is displayed in the Function Profiler window.



Function	Calls	Flat Time	Flat Time (%)	Acc. Time	Acc. Time (%)
main()	1	165	3.57	4356	94.18
PutFib(unsigned int)	10	3174	68.63	3174	68.63
NextCounter()	10	100	2.16	100	2.16
InitFib()	1	231	4.99	487	10.53
GetFib(int)	26	416	8.99	416	8.99
DoForegroundProcess()	10	270	5.84	3704	80.09
<Other>	0	269	5.82	4572	98.85

To sort the information, click on the relevant column header.

- 8 Before you start a new sampling, click the **Clear** button.
- 9 Click the **Graph** button to toggle the percentage columns to be displayed either as numbers or as bar charts.

## Code coverage

Code coverage is useful when you design your test procedure to make sure that all parts of your code have been executed. It also helps you identify parts of your code that are not reachable.

**Note:** When you debug on hardware, code coverage might have limitations; in particular, cycle counter statistics might not be available.



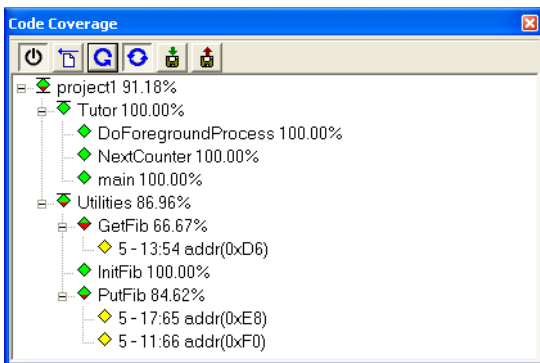
## To use code coverage:

- 1 Build your application using these options:

Category	Setting
C/C++ Compiler	Output>Generate debug information
Linker (for XLINK)	Format>Debug information for C-SPY
Linker (for ILINK)	Output>Include debug information in output
Debugger	Plugins>Code Coverage

- 2 Build your application and start C-SPY.

- 3 Choose **View>Code Coverage** to open the Code Coverage window.



- 4 Click the **Activate** button to turn on the code coverage analyzer.

- 5 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.

The Code Coverage window now reports the status of the current code coverage analysis, that is, which parts of the code that were executed at least once since the start of the analysis. The compiler generates detailed stepping information in the form of *step points* at each statement, and at each function call. The report includes information about all modules and functions. It reports the amount of all step points, in percentage, that were executed and lists all step points that were not executed up to the point where the application was stopped.

- 6 The coverage will continue until turned off.

**Note:** Code coverage can also be displayed in the Disassembly window. Executed code is indicated with a green diamond.

## Stack usage

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.



The Stack window shows the contents of the stack. This can be useful when:

- 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.

### To track stack usage:

- 1 Choose **Project>Options>Debugger>Plugins** and select **Stack** from the list of plugins.
- 2 Choose **Tools>Options>Stack** to configure the stack tracking. Note specifically that you might need to specify when the stack pointer(s) are valid.
- 3 Build your application and start C-SPY.
- 4 Choose **View>Stack**.

Stack view

Current stack pointer

Used stack memory, in dark gray

Unused stack memory, in light gray

Stack

Location	Data	Variable	Value	Frame
0x6FF8	0x0B			
+1	0x0B			
+2	0x0000	p.mStatus	0	[1] __exit
+4	0x4A			
+5	0x67			
+6	0xE0			
+7	0x04			

Current stack pointer

The graphical stack bar with tooltip information

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



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

### To detect stack overflows:

Choose **Tools>Options>Stack** and select the option **Enable stack checks**.

This means that C-SPY can issue warnings for stack overflow when the application stops executing. Warnings are issued either when the stack usage exceeds a threshold that you can specify, or when the stack pointer is outside the stack memory range.

### Trace

By collecting trace data, you can analyze 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.

A *trace* is a collected sequence of executed machine instructions. Available trace data depends very much on the C-SPY driver you are using:

- The C-SPY simulator collects the values of C-SPY expressions that you select in the Trace Expressions window. The Function Trace window only shows trace data corresponding to calls to and returns from functions, whereas the Trace window displays all instructions.
- C-SPY drivers for hardware debugger systems can collect trace data if the hardware you are using supports this, for example if there are dedicated communication channels or dedicated trace buffers for trace collection. In this case, the Trace window will reflect the collected data.

### To collect trace data:

- 1 No specific build settings are required for collecting trace data in the simulator. If you are using a hardware debugger system, its trace data generation must be configured first. Refer to the driver documentation for information.
- 2 Build your application and start C-SPY.
- 3 Choose **Trace** from the driver-specific menu to open the Trace window, and click the **Activate** button to turn on trace data collection.
- 4 Start the execution. When the execution stops, for instance because a breakpoint is triggered, trace data is displayed in the Trace window.



### **To start trace data collection using breakpoints:**

A convenient way to collect trace data between two execution points is to start and stop data collection using dedicated breakpoints. In the editor or Disassembly window, right-click, and toggle a **Trace Start** or **Trace Stop** breakpoint from the context menu. In the C-SPY simulator, the C-SPY system macros `__setTraceStartBreak` and `__setTraceStopBreak` can also be used.