PICMICRO™ C-SPY

User Guide

First edition: October 1998

Part no: CWPIC-1

WELCOME

Welcome to the PICmicro™ C-SPY User Guide.

This guide describes how to configure and run the IAR C-SPY Debugger for the PICmicro™ family of microcontrollers, and provides reference information about the features of C-SPY.

Before reading this guide refer to the *QuickStart Card*, or the chapter *Installation and documentation*, for information about installing C-SPY and an overview of the documentation.

ABOUT THIS GUIDE

This guide consists of the following chapters:

Installation and documentation explains how to install and run the Embedded Workbench or C-SPY, and gives an overview of the documentation supplied with the IAR Systems tools.

The *Introduction* provides a brief summary of C-SPY's features, and describes the simulator, emulator, and ROM-monitor versions of C-SPY.

The *Overview* explains how C-SPY works, and gives an overview of the features it provides.

The *Tutorial* is designed to introduce you to the most important features of C-SPY, and to serve as a useful starting point for debugging your own software.

Reference provides complete reference information about the C-SPY windows, menu commands, and their associated dialog boxes.

C-SPY expressions and macros defines the syntax of the expressions and variables used in C-SPY macros, and gives examples to show how to use macros in debugging.

System macros lists the built-in system macros supplied with C-SPY.

C-SPY configuration gives information about customizing C-SPY using command line options or setup macros.

ASSUMPTIONS AND CONVENTIONS

ASSUMPTIONS

This guide assumes that you already have a working knowledge of the following:

- The C programming language.
- The PICmicro™ family of microcontrollers.
- The PICmicro™ assembler language.
- The procedures for using menus, windows, and dialog boxes in Windows.

Note that the illustrations in this guide show the Embedded Workbench running in a Windows 95 style environment, and their appearance will be slightly different if you are using another platform.

CONVENTIONS

This guide uses the following typographical conventions:

Style	*Used for*
`computer`	Text that you type in, or that appears on the screen.
`parameter`	A label representing the actual value you should type as part of a command.
`[option]`	An optional part of a command.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
reference	A cross-reference to another part of this guide, or to another guide.

CONTENTS

INSTALLATION AND DOCUMENTATION

This chapter explains how to install and run the Embedded Workbench and command line versions of the IAR products, and gives an overview of the available documentation. It also describes the `iar` subdirectories and file types.

INCLUDED IN THIS PACKAGE

The PICmicro™ package contains the following items:

- CD-ROM or floppy disks.
- Product documentation:

  PICmicro™ Embedded Workbench Interface Guide

  PICmicro™ Command Line Interface Guide

  PICmicro™ Assembler, Linker, and Librarian Programming Guide

  PICmicro™ C Compiler Programming Guide

  PICmicro™ C-SPY User Guide, if you have purchased the IAR C-SPY debugger.
- Licence agreement including the *Product Registration Form*, which we urge you to fill out and send us to ensure that you receive the latest release of the IAR development tools for the PICmicro™ family of microcontrollers.

INSTALLING THE EMBEDDED WORKBENCH WITH C-SPY

This section explains how to install and run the Embedded Workbench with C-SPY.

WHAT YOU NEED

- Windows 95/98, or Windows NT 3.51 or later.
- At least 40 Mbytes of free disk space for the Embedded Workbench.
- 32 Mbytes of RAM recommended for the Embedded Workbench and the IAR C-SPY Debugger.

If you are using C-SPY you should install the Workbench before C-SPY.

INSTALLING FROM WINDOWS 95/98 OR NT 4.0

1 Insert the installation CD-ROM or the first installation disk.

If you install from a CD-ROM, follow the instructions on the screen. If you install from a floppy disk, follow the instructions below:

2 Click the **Start** button in the taskbar, then click **Settings** and **Control Panel**.

3 Double-click the **Add/Remove Programs** icon in the **Control Panel** folder.

4 Click **Install**, then follow the instructions on the screen.

RUNNING FROM WINDOWS 95/98 OR NT 4.0

1 Click the **Start** button in the taskbar, then click **Programs** and **IAR Embedded Workbench**.

2 Click the **IAR Embedded Workbench** program icon.

INSTALLING FROM WINDOWS NT 3.51

1 Insert the first installation disk or the installation CD-ROM.

If you install from a CD-ROM, follow the instructions on the screen. If you install from a floppy disk, follow the instructions below:

2 Double-click the **File Manager** icon in the **Main** program group.

3 Click the floppy disk icon in the **File Manager** toolbar.

4 Double-click the **setup.exe** icon, then follow the instructions on the screen.

RUNNING FROM WINDOWS NT 3.51

Go to the Program Manager and double-click the **IAR Embedded Workbench** icon.

RUNNING C-SPY

Either:

Start C-SPY in the same way as you start the Embedded Workbench (see above).

Or:

Choose **Debugger** from the Embedded Workbench **Project** menu.

INSTALLING THE COMMAND LINE TOOLS

This section describes how to install and run the command line versions of the IAR Systems tools. You should be familiar with your operating system.

WHAT YOU NEED

- Windows 95/98, or Windows NT 3.51 or later.
- At least 35 Mbytes of free disk space.
- 32 Mbytes of RAM recommended for the IAR applications.

INSTALLATION

1. Insert the first installation disk.
2. At the command line prompt type `a:\install` and press Enter.
3. Follow the instructions on the screen.

When the installation is complete:

4. Add the path to the IAR Systems command line executable files to the PATH variable. For a default installation you would add `c:\iar\exe`.

   Define the environment variables `APIC_INC`, `C_INCLUDE`, `QPICINFO` and `XLINK_DFLTDIR` specifying the paths to the `inc` and `lib` directories; for example:

   ```
   set APIC_INC=c:\iar\inc\
   set C_INCLUDE=c:\iar\inc\
   set QPICINFO=c:\iar\setup\
   set XLINK_DFLTDIR=c:\iar\lib\
   ```

RUNNING THE TOOLS

Type the appropriate command at the command line prompt.

For more information refer to the *PICmicro™ C Compiler Programming Guide*, and the *PICmicro™ Assembler, Linker, and Librarian Programming Guide*.

INSTALLED FILES

The installation procedure creates several directories to contain the different types of files used with the IAR Systems development tools. The following sections give a description of the files contained by default in each directory. During installation you have the option to specify other directories than the ones created by default.

DOCUMENTATION FILES

Your installation may include a number of ASCII-format text files (`*.txt`) containing recent additional information. It is recommended that you read all of these files before proceeding.

ASSEMBLER FILES

The `apic` subdirectory holds the document files and assembler include files for the PICmicro™ Assembler.

The `iar\ew`*`nn`*`\picmicro\apic\tutor` directory contains the files used for the PICmicro™ Assembler tutorials.

MISCELLANEOUS FILES

The `etc` subdirectory holds the XLINK-related files.

EXECUTABLE FILES

The `iar` directory contains the `ew`*`nn`*`.exe` and `cw`*`nn`*`.exe` files. Other executable files are located in the `iar\ew`*`nn`*`\picmicro\bin` directory.

The `bin` subdirectory holds the executable program files.

The installation procedure also includes an addition to the `autoexec.bat` `PATH` statement, directing having the `bin` subdirectory searched for command files. This allows you to issue a command from any directory.

C COMPILER FILES

The `iccpic` subdirectory holds various source files for basic I/O library routines.

The `iar\ewnn\picmicro\iccpic\tutor` directory contains the files used for the PICmicro™ C Compiler tutorials.

INCLUDE FILES

The `inc` subdirectory holds include files, such as the header files for the standard C library, as well as a specific header file defining SFRs (special function registers) for each supported PICmicro™ derivative. These files are used by the C compiler.

The C compiler searches for include files in the directory specified by the `C_INCLUDE` environment variable. If you set this environment variable to the path of the include subdirectory, as suggested in the installation procedure, you can refer to `inc` header files simply by their base names.

The assembler has an equivalent environment variable, `APIC_INC`.

LIBRARY FILES

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

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

No library modules are installed; instead the modules should be built for the appropriate microcontroller configuration using the included Buildlib utility, see the chapter *General C library definitions* in the *C Compiler Programming Guide*.

Pre-built library modules with standard configuration are supplied in the `\lib` directory on the CD. A library is supplied for each supported microcontroller and is named `cl`*`xxxxx`*`.r39`, where *`xxxxx`* corresponds to the microcontroller name.

LINKER COMMAND FILES

The `iccpic` subdirectory holds an example linker command file for each supported microcontroller.

FILE TYPES

The PICmicro™ versions of the IAR Systems development tools use the following default file extensions to identify the IAR-specific types of file:

Ext.	*Type of file*	*Output from*	*Input to*
`.a39`	Target program	XLINK	EPROM, C-SPY, etc
`.c`	C program source	Text editor	C compiler
`.d39`	Target program with debug information	XLINK	C-SPY, etc
`.h`	C header source	Text editor	C compiler `#include`
`.i39`	Compiler/debugger	Processor description file	–
`.inc`	Assembler header	Text editor	Assembler `#include` file
`.lst`	List	C compiler and assembler	–
`.mac`	C-SPY macro definition	Text editor	C-SPY
`.map`	XLINK map	XLINK	–
`.mem`	Target memory layout	Text editor	C-SPY
`.prj`	Embedded Workbench project	Embedded Workbench	Embedded Workbench
`.r39`	Object module	C compiler and assembler	XLINK and XLIB
`.s39`	Assembler program source	Text editor	Assembler
`.xcl`	Extended command line	Text editor	XLINK and C compiler

The default extension may be overridden by simply including an explicit extension when specifying a filename.

Note that, by default, XLINK listings (maps) will have the `.lst` extension, and this may overwrite the listing file generated by the compiler. It is recommended that you explicitly name XLINK map files, for example `demo1.map`.

Files with the extensions `.ini` and `.cfg` are created dynamically when you install and run the tools. These files contain information about your configuration and other settings.

DOCUMENTATION

THE OTHER GUIDES

The other guides provided with the Embedded Workbench are as follows:

PICmicro™ Command Line Interface Guide
This guide explains how to configure and run the IAR Systems development tools from the command line. It also includes reference information about the command line environment variables.

PICmicro™ Embedded Workbench Interface Guide
This guide explains how to configure and run the IAR Systems development tools from the Embedded Workbench interface. It also includes complete reference information about the Embedded Workbench commands and dialog boxes, and the Workbench editor.

PICmicro™ C Compiler Programming Guide
This guide provides programming information about the PICmicro™ C Compiler. It includes reference information about the C library functions and language extensions, and provides information about support for the target-specific options such as memory models.

You should refer to this guide when you are setting up the C compiler configuration options in the Embedded Workbench, and for information about the C language when writing and debugging C source programs.

This guide also describes the diagnostic functions and lists the PICmicro™- specific warning and error messages.

PICmicro™ Assembler, Linker, and Librarian Programming Guide
This guide provides reference information about the PICmicro™ Assembler, XLINK Linker, and XLIB Librarian for use with the Embedded Workbench.

The assembler programming sections include details of the assembler source format, and reference information about the assembler operators, directives, and mnemonics.

The XLINK Linker programming reference sections provide information about the XLINK Linker commands and output formats.

The XLIB Librarian programming sections provide information about the XLIB Librarian commands.

Finally, the guide includes a list of diagnostic messages for each of these tools.

In addition to the information contained in the PICmicro™ guides, also online information is available.

ONLINE HELP

From the **Help** menu in the Embedded Workbench and the IAR C-SPY Debugger, you can access the PICmicro™ online information. It contains complete reference information for the PICmicro™ Embedded Workbench, C-SPY, C compiler, assembler, XLINK Linker, and XLIB Librarian.

READ-ME FILES

We recommend that you read the following `Read-Me` files for recent information that is not included in the guides:

```
apic.txt
cwpic.txt
ewpic.txt
iccpic.txt
xlink.txt
```

IAR ON THE WEB

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

- Product announcements.
- Special offerings.
- Evaluation copies of the IAR products.
- Technical Support including FAQs (frequently asked questions).
- Links to chip manufacturers and other interesting sites.
- Distributor information.

INTRODUCTION

C-SPY for Windows is a high-level language debugger for embedded applications. It runs under Windows, and provides a range of menu commands and toolbar buttons for the most frequently-needed debugging operations.

C-SPY is designed for use with the C compilers, assemblers, XLINK Linker, and XLIB Librarian produced by IAR Systems. It provides the best of both worlds by allowing you to switch between source mode and disassembly mode debugging as required.

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

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

KEY FEATURES

C-SPY offers the following unique combination of features:

FEATURES

- Intuitive Windows interface.
- Source and disassembly mode debugging.
- Fast simulator.
- Log file option.
- Powerful macro language.
- Complex breakpoints.
- Memory validation.
- Interrupt simulation.
- Code coverage.

- UBROF, INTEL-HEX, and Motorola input formats supported.
- C expression analyzer.
- Extensive type recognition of variables.
- Function trace.
- C call stack with parameters.
- Watchpoints on expressions.
- Function level profiling.
- Optional `getchar`/`putchar` emulation.
- Watch, Locals, and QuickWatch windows allow you to expand `arrays` and `structs`.
- Full support for auto and register variables.
- Built-in assembler/disassembler.

VERSIONS

C-SPY for PICmicro™ is currently available in a simulator version. An emulator version and a ROM-monitor version may also become available; please contact your IAR representative for further information.

SIMULATOR VERSION

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

EMULATOR VERSION

The emulator version of C-SPY provides control over one of a number of popular in-circuit emulators, which is connected to the host computer via a serial or parallel port.

C-SPY uses the hardware features of the emulator, such as breakpoint logic and memory inspection, to allow an application to be executed in real time and in the real target environment.

If you are using the emulator version of C-SPY, refer to the emulator supplement for additional information.

ROM-MONITOR VERSION

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

If you are using the ROM-monitor version of C-SPY, refer to the ROM-monitor supplement for additional information.

OVERVIEW

The IAR C-SPY Debugger is a powerful interactive debugger for embedded applications.

This chapter explains how C-SPY works, and gives an overview of the features it provides.

DISASSEMBLY AND SOURCE MODE DEBUGGING

C-SPY provides the best of both worlds by allowing you to switch between source and disassembly mode debugging as required.

Wherever source code is available the source mode debugging displays the source program, and you can execute the program one statement at a time while monitoring the values of variables and data structures. Source mode debugging provides the quickest and easiest way of developing your application, without having to worry about how the compiler or assembler has implemented the code.

Disassembly mode debugging displays a mnemonic assembler listing of your program based on actual memory contents rather than source code, and lets you execute the program exactly one assembler instruction at a time. Disassembly mode debugging lets you focus on the critical sections of your application, and provides you with precise control over the simulated hardware.

During both source and disassembly mode debugging you can display the registers and memory, and change their contents.

SOURCE WINDOW

As you debug an application the source or disassembly source is displayed in a Source window, with the next source or disassembly statement to be executed highlighted.

You can navigate quickly to a particular file or function in the source code by selecting its name from the file or function box at the top of the Source window.

For convenience the Source window uses colors and text styles to identify key elements of the syntax. For example, by default C keywords are displayed in bold and constants in red. However, the colors and font styles are fully configurable, so that you can change them to whatever you find most convenient.

PROGRAM EXECUTION

C-SPY provides a flexible range of options for executing the target program.

The **Go** command continues execution from the current position until a breakpoint or program exit is reached. You can also execute up to a selected point in the program, without having to set a breakpoint, with the **Go to Cursor** command. Alternatively, you can execute out of a C function with the **Go Out** command.

SINGLE STEPPING

The **Step** and **Step Into** commands allow you to execute the program a statement or instruction at a time. **Step Into** continues stepping inside function or subroutine calls whereas **Step** executes each function call in a single step.

The **Autostep** command steps repeatedly, and the **Multi Step** command lets you execute a specified number of steps before stopping.

BREAKPOINTS

You can set breakpoints in the program being debugged using the **Toggle Breakpoint** command. Statements or instructions at which breakpoints are set are shown highlighted in the Source window listing.

Alternatively the **Edit Breakpoints** command allows you to define and alter complex breakpoints, including break conditions. You can optionally specify a macro which will perform certain actions when the breakpoint is encountered.

INTERRUPT SIMULATION

C-SPY includes an interrupt system allowing you to optionally simulate the execution of interrupts when debugging with C-SPY. The interrupt system can be turned on or off as required either with a system macro, or using the **Interrupt** dialog box. The interrupt system is activated by default, but if it is not required it can be turned off to speed up instruction set simulation.

The interrupt system has the following features:

- Interrupts, single or periodical, can be set up so that they are generated based on the cycle counter value.
- C-SPY provides interrupt servicing support suitable for the PICmicro™ derivatives.

- By combining an interrupt with a data breakpoint you can simulate peripheral devices, such as a serial port.

C FUNCTION INFORMATION

C-SPY keeps track of the currently-active functions and their local variables, and a list of the function calls can be displayed in the Calls window. You can also trace functions during program execution using the **Trace** command and tracing information is displayed in the Report window.

You can use the **Quick Watch** command to examine the value of any local, global, or static variable that is in scope. You can monitor the value of a macro, variable, or expression in the Watch window as you step through the program.

VIEWING AND EDITING MEMORY AND REGISTERS

You can display the contents of the processor registers in the Register window, and specified areas of memory in the Memory window.

The Register window allows you to edit the content of any register, and the register is automatically updated to reflect the change.

The Memory window can display the contents of memory in groups of 8, 16, or 32 bits, and you can double-click any memory address to edit the contents of memory at that address.

TERMINAL I/O

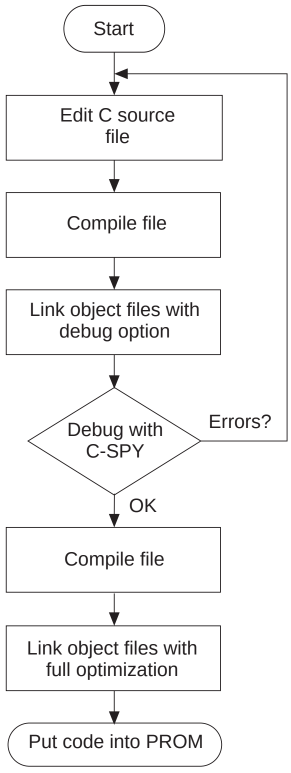
C-SPY can simulate terminal input and output using the Terminal I/O window.

MACRO LANGUAGE

C-SPY includes a powerful internal macro language, to allow you to define complex sets of actions to be performed; for example, when breakpoints are encountered. The macro language includes conditional and loop constructs, and you can use variables and expressions.

DEVELOPMENT CYCLE

The following diagram shows a typical development cycle using the IAR Systems C Compiler and XLINK Linker in conjunction with C-SPY:



If the application requires some assembler language programming, this can be developed using the IAR Systems Assembler, and linked in a similar way.

Because C-SPY makes use of information compiled into the C modules using a special debug option, C source-level debugging is only available when using C-SPY with the appropriate C compiler available from IAR Systems.

TUTORIAL

This tutorial illustrates how you might use C-SPY to debug a simple program, and it illustrates some of C-SPY's most important features.

Before reading this chapter you should:

- Have installed the C-SPY software, as described in the *QuickStart Card* and in the chapter *Installation and documentation*.
- Be familiar with the architecture and instruction set of the PICmicro™ family of microcontrollers.
- Be familiar with the Windows environment.

GETTING STARTED

The tutorial uses the source files `tutor.c` and `common.c`, and the include files `tutor.h` and `common.h`, which are all supplied by default in the `iccpic\tutor` and `inc` directories in your installation directory.

The program initializes an array with the 10 first Fibonacci numbers and prints the result in the Terminal I/O window.

COMPILING THE PROGRAM

If you have installed the PICmicro™ C compiler, you can compile the demonstration program to produce a file for use with C-SPY as follows:

We recommend that you create a directory where you have your project files, eg `c:\projects`. In this tutorial we assume that there is a directory called `projects` on the `c:` drive and that the source and project files are located in this directory.

Create a project called `Demo` containing the source files `tutor.c` and `common.c`. Then compile and link the project with the following options:

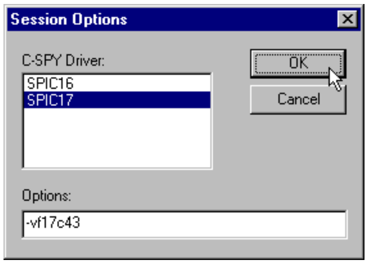
Category	*Page*	*Options*
General	Target	Processor: 17C43
ICCPIC	Debug	Generate debug information
XLINK	Output	Format: Debug info with terminal I/O

The linker generates a file `demo.d39`.

RUNNING C-SPY

To run C-SPY from the Embedded Workbench choose **Debugger** from the Embedded Workbench **Project** menu.

Alternatively, you can start C-SPY outside the Embedded Workbench. Then choose **Open...** from the **File** menu and open the demo file `demo.d39`. This dialog box allows you to select the appropriate C-SPY driver, and specify other options:

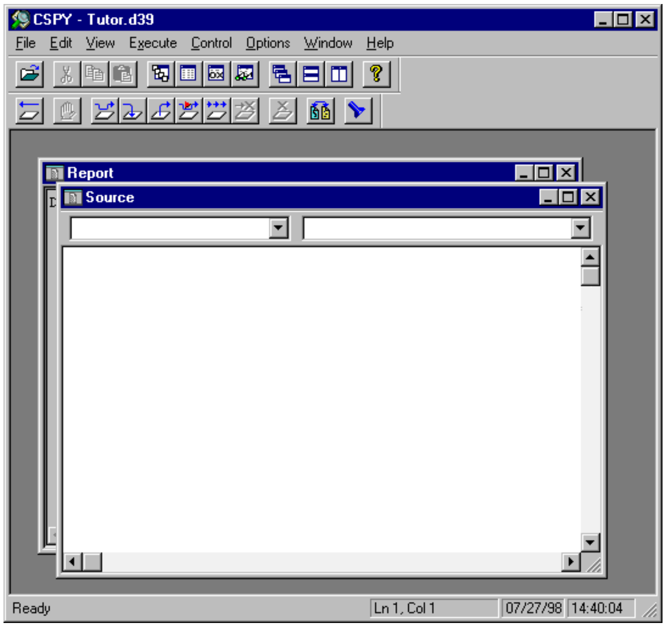

Session Options

C-SPY Driver: SPIC16, SPIC17 (selected)

Options: -vf17c43

Select **SPIC**17 and type `-vf17c43` in the option box. Click **OK**.

An empty Source window will be opened for this file:



DEBUGGING IN SOURCE MODE

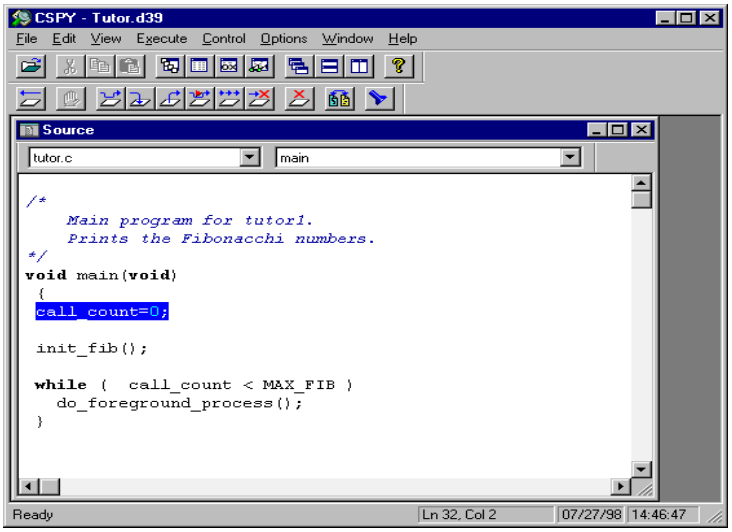
C-SPY starts in Source mode, but the Source window is initially blank because a program actually starts in a low level assembler module, assembled without debug information, so there is no source code corresponding to this.

To inspect the assembler code, select **Toggle Source/Disassembly** from the **View** menu. Alternatively, click the **Toggle Source/Disassembly** button in the debug bar.

Now choose Source level and execute one step by choosing **Step** from the **Execute** menu. Alternatively, click the **Step** button in the debug bar.

At Source level **Step** executes a Source statement at a time.

Step takes you to the first executable statement of `main`, where C programs usually start:



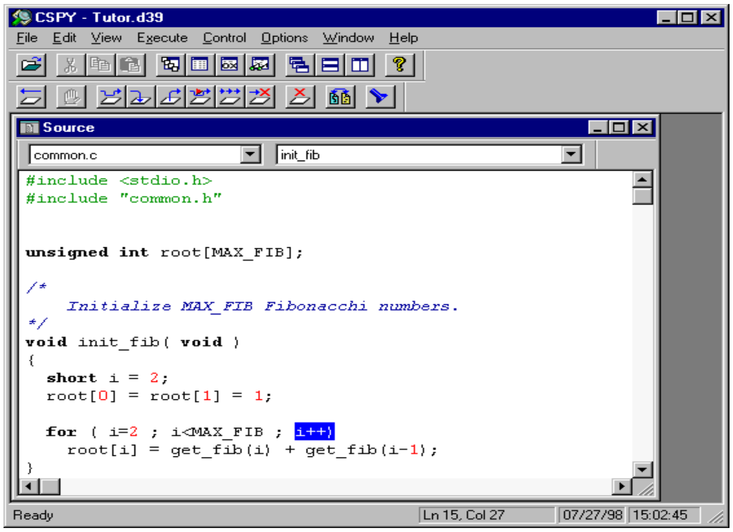
```
/*
    Main program for tutor1.
    Prints the Fibonacchi numbers.
*/
void main(void)
 {
 call_count=0;

 init_fib();

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

The current position in the program, ie the next C statement to be executed, is shown highlighted in the Source window.

Now give one more step. The current position should be the call to the `init_fib` function. We choose **Step Into** from the **Execute** menu to execute `init_fib` one step at the time. Alternatively, click the **Step Into** button in the debug bar.

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

Step four more times. You will notice that the three individual parts of a `for` statement are separated, as C-SPY debugs on statement level, not on line level. The current position should now be `i++`:



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


unsigned int root[MAX_FIB];

/*
    Initialize MAX_FIB Fibonacchi numbers.
*/
void init_fib( void )
{
  short i = 2;
  root[0] = root[1] = 1;

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

WATCHING VARIABLES

C-SPY allows you to set watchpoints on C variables or expressions, to allow you to keep track of their values as you execute the program.

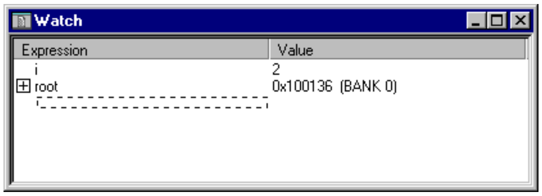
Set a watchpoint on the variable `i` as follows:

Choose **Watch** from the **Window** menu to open the Watch window, or click the **Watch Window** button in the debug bar.

Select the dotted rectangle, then click and briefly hold the left mouse button, and type: `i` ↵.

You can also drag and drop a variable in the Watch window. Double-click on the `root` array in the `init_fib` function. When `root` is marked, drag and drop it in the Watch window.

The Watch window will show the current value of `i` and `root`:



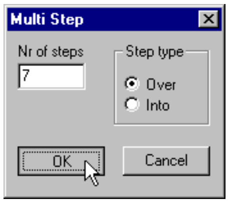
Expression	Value
i	2
⊞ root	0x100136 (BANK 0)

The `root` is an array and can be watched in more detail. This is indicated in the Watch window by the ⊞ symbol to the left of the variable. Click the symbol to display the current contents of `root`.

If you have several variables with the same name in different modules or functions, you can include the module or function name with the variable name in order to separate them.

If necessary, resize and rearrange the windows so that the Watch window is visible. Alternatively, you can watch the variable by pointing at it in the Source window with the mouse pointer, or by opening the Locals window.

Choose **Multi Step…** from the **Execute** menu, and enter `7`, to step seven more times to see how the value of `i` changes.



Nr of steps: 7

Step type: [✓ Over] Into

SETTING BREAKPOINTS

You can set breakpoints at C function names or line numbers, or at assembler symbols or addresses.

You can also set breakpoints interactively, simply by positioning the cursor in a statement and then choosing the **Toggle Breakpoint** command.

First, ensure that the Report window is open by choosing **Report** from the **Window** menu. The Report window shows information about breakpoint execution.

You should now have the Source, Report, and Watch windows on the screen; position them neatly before proceeding.

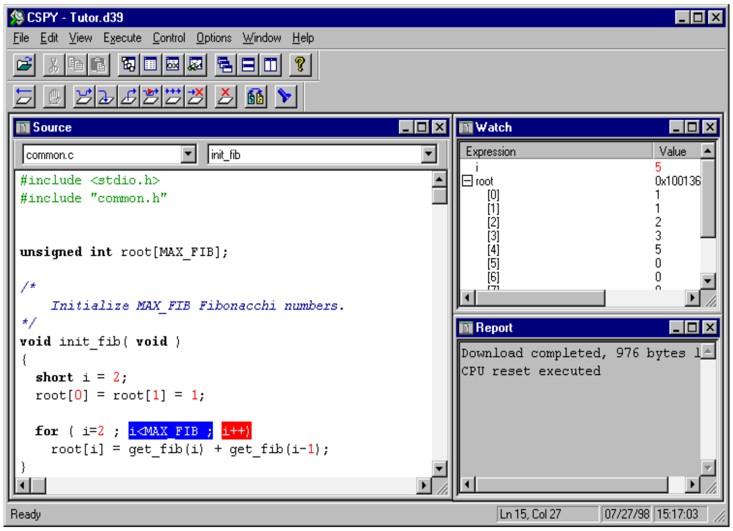
Now use the following procedure to set a breakpoint at the statement:

```
i++;
```

First click in this statement in the Source window, to position the cursor.

Then choose **Toggle Breakpoint** from the **Control** menu, or click the **Toggle Breakpoint** button in the toolbar.

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



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


unsigned int root[MAX_FIB];

/*
    Initialize MAX_FIB Fibonacchi numbers.
*/
void init_fib( void )
{
  short i = 2;
  root[0] = root[1] = 1;

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

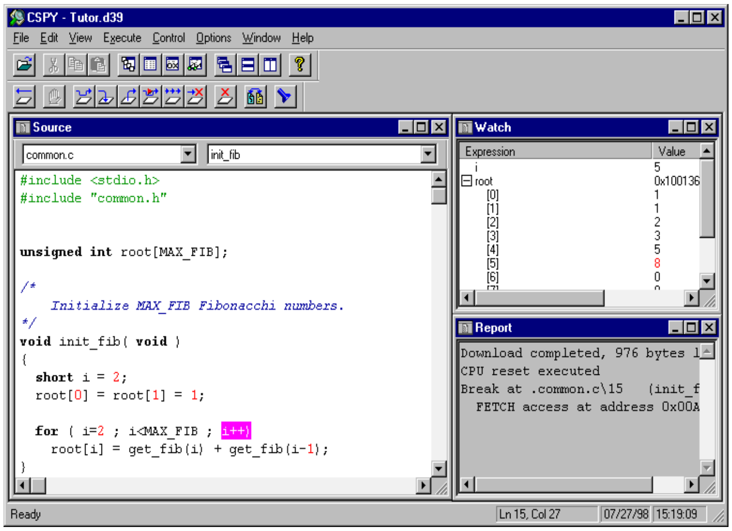
Expression	Value
i	5
⊟ root	0x100136
[0]	1
[1]	1
[2]	2
[3]	3
[4]	5
[5]	0
[6]	0

```
Download completed, 976 bytes l
CPU reset executed
```

EXECUTING UP TO A BREAKPOINT

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

The program will execute up to the breakpoint you set, and information about the breakpoint will be displayed in the Report window:



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


unsigned int root[MAX_FIB];

/*
    Initialize MAX_FIB Fibonacci numbers.
*/
void init_fib( void )
{
  short i = 2;
  root[0] = root[1] = 1;

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

Expression	Value
i	5
⊟ root	0x100136
[0]	1
[1]	1
[2]	2
[3]	3
[4]	5
[5]	8
[6]	0

```
Download completed, 976 bytes l
CPU reset executed
Break at .common.c\15   (init_f
  FETCH access at address 0x00A
```

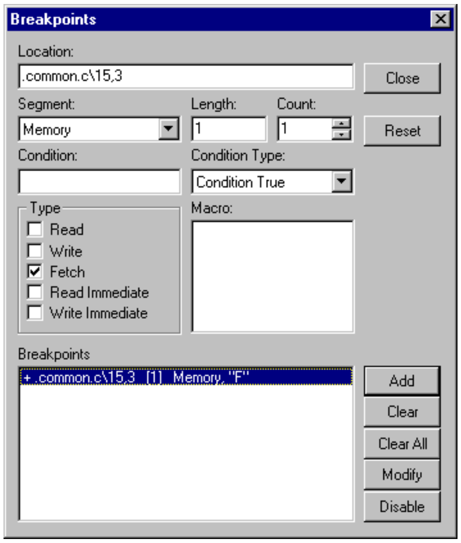
DEFINING COMPLEX BREAKPOINTS

C-SPY allows you to define complex breakpoint conditions, to allow you to detect when your program has reached a particular state of interest.

For example, modify the breakpoint you have set so that it detects when the value of `i` exceeds 8.

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

Then select the breakpoint in the **Breakpoints** list to display information about the breakpoint you defined:



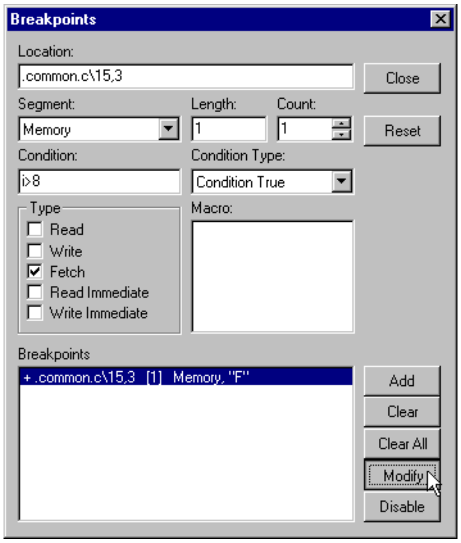
Location: .common.c\15,3
Segment: Memory
Length: 1
Count: 1
Condition:
Condition Type: Condition True
Type: [ ] Read [ ] Write [✓ Fetch] [ ] Read Immediate [ ] Write Immediate
Macro:
Breakpoints: + .common.c\15,3 [1] Memory, "F"

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

Add a condition to the breakpoint using the following procedure:

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

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



Location: .common.c\15,3
Segment: Memory
Length: 1
Count: 1
Condition: i>8
Condition Type: Condition True
Type
- [ ] Read
- [ ] Write
- [x] Fetch
- [ ] Read Immediate
- [ ] Write Immediate

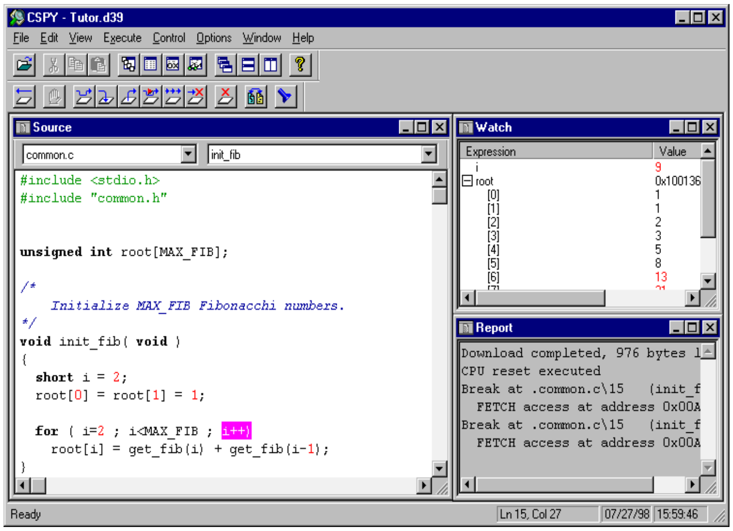
Macro:

Breakpoints
+ .common.c\15,3 [1] Memory, "F"

Finally, choose **Close** to close the **Breakpoints** dialog box.

EXECUTING UNTIL A CONDITION IS TRUE

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



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


unsigned int root[MAX_FIB];

/*
   Initialize MAX_FIB Fibonacchi numbers.
*/
void init_fib( void )
{
  short i = 2;
  root[0] = root[1] = 1;

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

Expression	Value
i	9
⊟ root	0x100136
[0]	1
[1]	1
[2]	2
[3]	3
[4]	5
[5]	8
[6]	13

```
Download completed, 976 bytes l
CPU reset executed
Break at .common.c\15   (init_f
  FETCH access at address 0x00A
Break at .common.c\15   (init_f
  FETCH access at address 0x00A
```

Ln 15, Col 27 | 07/27/98 | 15:59:46

EXECUTING UP TO THE CURSOR

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

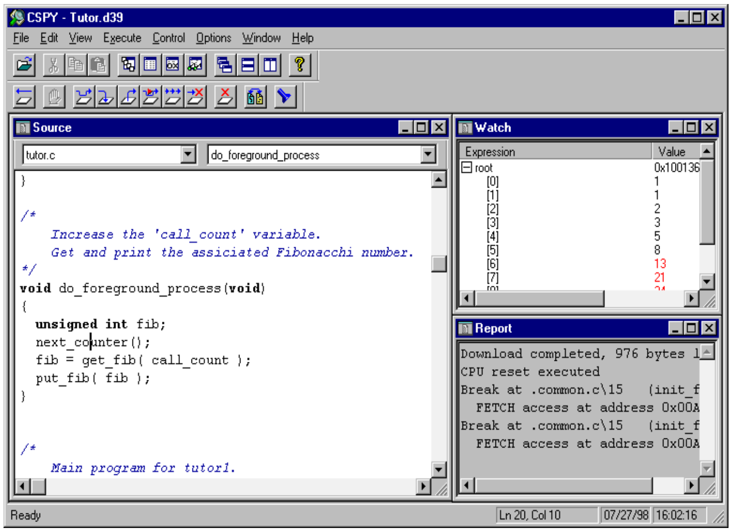
In the `do_foreground_process` function we print the calculated numbers. We shall place a breakpoint at the statement where the call to `put_fib` is executed in the `do_foreground_process` function.

First remove the variable `i` from the Watch window. Select the variable in the Watch window and press Delete.

Then select the file `tutor.c` in the **Source file** box. Now you can choose the `do_foreground_process` function in the **Function** box.

Position the cursor in the Source window in the statement:

```
next_counter();
```



```
}

/*
    Increase the 'call_count' variable.
    Get and print the assiciated Fibonacchi number.
*/
void do_foreground_process(void)
{
  unsigned int fib;
  next_counter();
  fib = get_fib( call_count );
  put_fib( fib );
}


/*
    Main program for tutor1.
```

Expression	Value
root	0x100136
[0]	1
[1]	1
[2]	2
[3]	3
[4]	5
[5]	8
[6]	13
[7]	21

```
Download completed, 976 bytes l
CPU reset executed
Break at .common.c\15   (init_f
  FETCH access at address 0x00A
Break at .common.c\15   (init_f
  FETCH access at address 0x00A
```

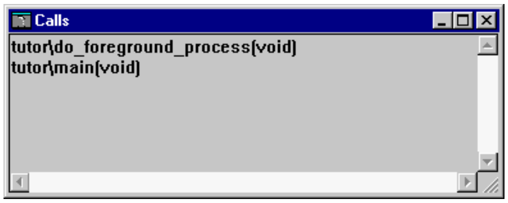
Then choose **Go to Cursor** from the **Execute** menu, or click the **Go to Cursor** button in the toolbar.

The program will then execute up to the statement at the cursor position.

DISPLAYING FUNCTIONS CALLS

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

Choose **Calls** from the **Window** menu to open the Calls window and display the function calls:



```
tutor\do_foreground_process(void)
tutor\main(void)
```

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

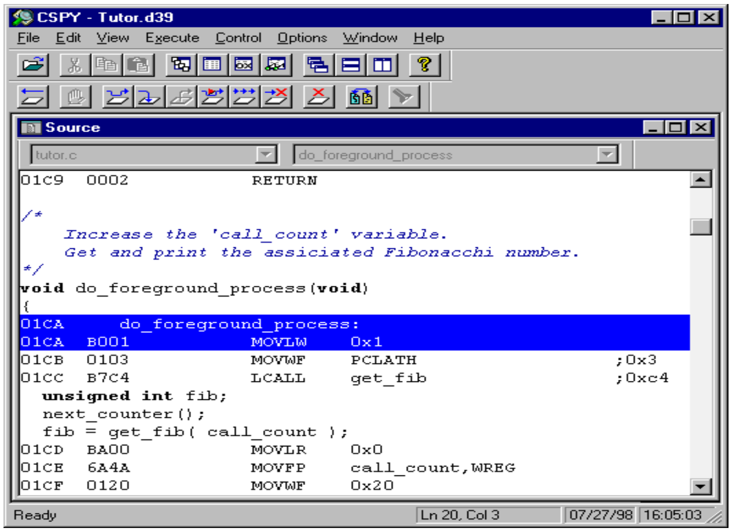
Close the Calls window by clicking its close box.

DEBUGGING IN DISASSEMBLY MODE

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

Change the mode by choosing **Toggle Source/Disassembly** from the **View** menu. Alternatively, click the **Toggle Source/Disassembly** button in the debug bar.

You will see the assembler code corresponding to the current C statement:



```
01C9   0002              RETURN

/*
    Increase the 'call_count' variable.
    Get and print the assiciated Fibonacchi number.
*/
void do_foreground_process(void)
{
01CA       do_foreground_process:
01CA   B001              MOVLW     0x1
01CB   0103              MOVWF     PCLATH                        ;0x3
01CC   B7C4              LCALL     get_fib                       ;0xc4
  unsigned int fib;
  next_counter();
  fib = get_fib( call_count );
01CD   BA00              MOVLR     0x0
01CE   6A4A              MOVFP     call_count,WREG
01CF   0120              MOVWF     0x20
```

Stepping is now one assembler instruction at a time.

When you are debugging in disassembly mode every assembler instruction that has been executed is marked with a * (asterisk). Note that there may be a delay before this information is displayed, due to the way the Source window is updated.

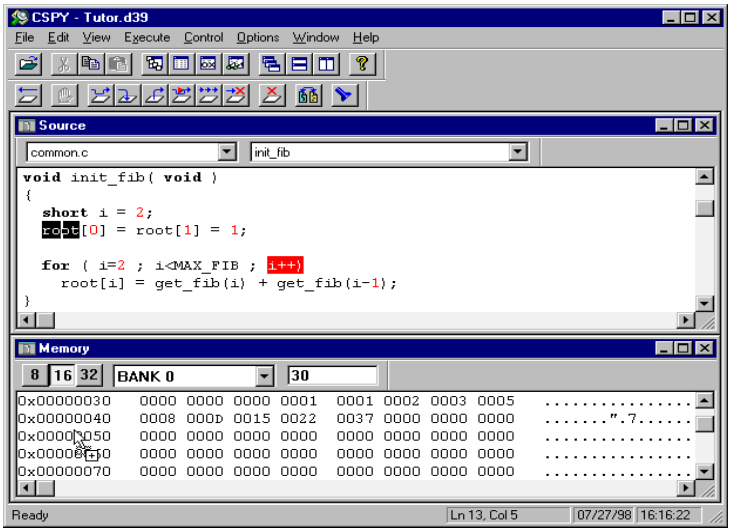
MONITORING MEMORY

You can monitor selected areas of memory by opening the Memory window. For example, open a window to monitor the memory corresponding to the variable `root` as follows.

Choose **Memory…** from the **Window** menu to open the Memory window. Position the Source and Memory windows conveniently on the screen.

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

Then select `root` in the file `common.c`, and drag it from the Source window, and drop it into the Memory window:

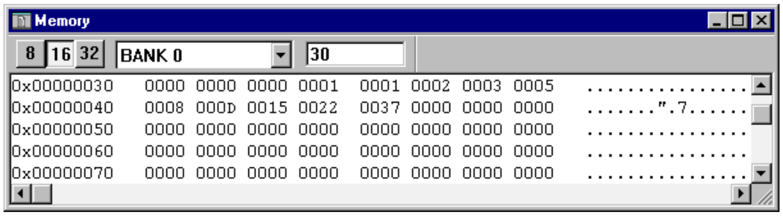


```
void init_fib( void )
{
  short i = 2;
  root[0] = root[1] = 1;

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

```
0x00000030   0000 0000 0000 0001   0001 0002 0003 0005   ................
0x00000040   0008 000D 0015 0022   0037 0000 0000 0000   .......".7......
0x00000050   0000 0000 0000 0000   0000 0000 0000 0000   ................
0x00000060   0000 0000 0000 0000   0000 0000 0000 0000   ................
0x00000070   0000 0000 0000 0000   0000 0000 0000 0000   ................
```

The Memory window will show the contents of memory corresponding to `root`.

In this case it is convenient to display the memory contents as words, so click the **16** button in the Memory window toolbar:



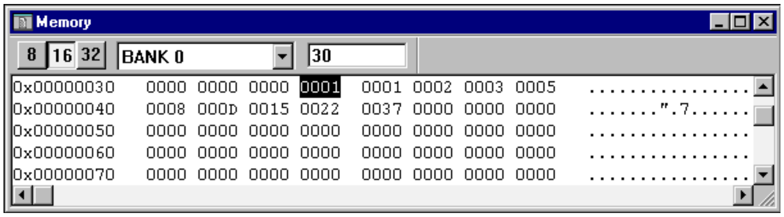
```
0x00000030   0000 0000 0000 0001   0001 0002 0003 0005   ................
0x00000040   0008 000D 0015 0022   0037 0000 0000 0000   .......".7......
0x00000050   0000 0000 0000 0000   0000 0000 0000 0000   ................
0x00000060   0000 0000 0000 0000   0000 0000 0000 0000   ................
0x00000070   0000 0000 0000 0000   0000 0000 0000 0000   ................
```

Note that the 10 words have been initialized by the `init_fib` function of the C program.

CHANGING MEMORY

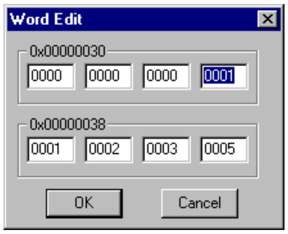
You can change the contents of memory by editing the values in the Memory window.

Double-click the row of bytes you want to edit:



```
0x00000030   0000 0000 0000 0001  0001 0002 0003 0005   ................
0x00000040   0008 000D 0015 0022  0037 0000 0000 0000   .......".7......
0x00000050   0000 0000 0000 0000  0000 0000 0000 0000   ................
0x00000060   0000 0000 0000 0000  0000 0000 0000 0000   ................
0x00000070   0000 0000 0000 0000  0000 0000 0000 0000   ................
```

A dialog box is displayed:



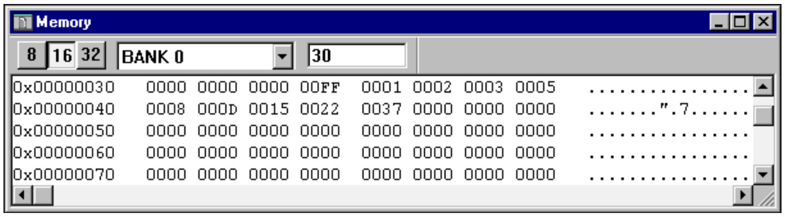
0x00000030: 0000 0000 0000 0001

0x00000038: 0001 0002 0003 0005

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

For example, suppose you want the number 255 to be written as the first number in the `root` array instead of the number 1 (`0x0001`). Select the `0001` word at address `0x36` in the Memory window and change it to `00FF` in the **Word Edit** dialog box.

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



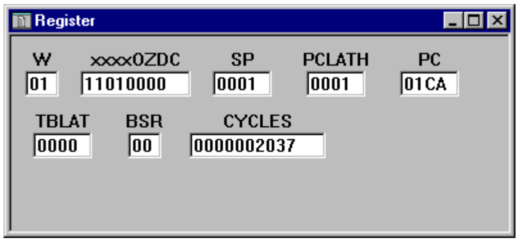
```
0x00000030   0000 0000 0000 00FF  0001 0002 0003 0005   ................
0x00000040   0008 000D 0015 0022  0037 0000 0000 0000   .......".7......
0x00000050   0000 0000 0000 0000  0000 0000 0000 0000   ................
0x00000060   0000 0000 0000 0000  0000 0000 0000 0000   ................
0x00000070   0000 0000 0000 0000  0000 0000 0000 0000   ................
```

Before proceeding switch back to disassembly mode by choosing **Toggle Source/Disassembly** or clicking the **Toggle Source/Disassembly** button in the debug bar, and close the Memory window.

MONITORING REGISTERS

You can monitor the contents of the processor registers, and modify their contents, using the Register window.

Open the Register window by choosing **Register** from the **Window** menu.



W: 01
xxxx0ZDC: 11010000
SP: 0001
PCLATH: 0001
PC: 01CA
TBLAT: 0000
BSR: 00
CYCLES: 0000002037

Now choose **Step** from the **Execute** menu, or click the **Step** button in the debug bar, to execute the next instructions, and watch the effect in the Register window.

When you have finished close the Register window.

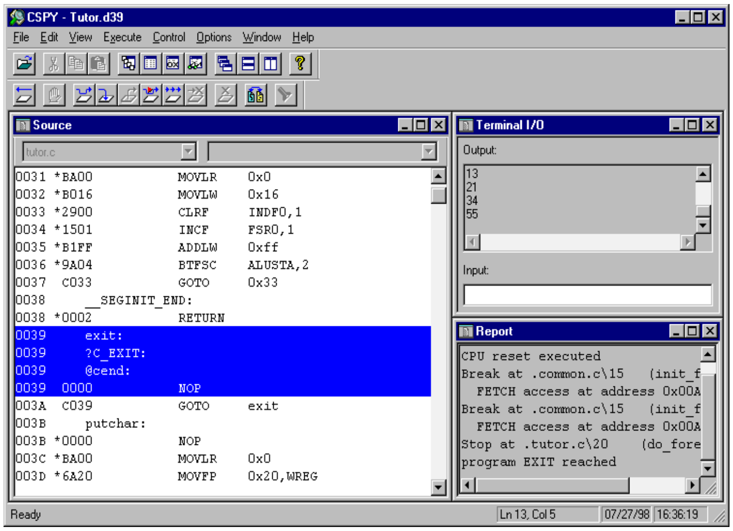
CONTINUING EXECUTION

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

To complete execution of the program choose **Go** from the **Execute** menu, or click the **Go** button in the debug bar.

Choose **Report** from the **Window** menu to bring the Report window to the front.

Since no more breakpoints are encountered C-SPY reaches the end of the program, and prints out a `program EXIT reached` message:



```
0031 *BA00          MOVLR    0x0
0032 *B016          MOVLW    0x16
0033 *2900          CLRF     INDF0,1
0034 *1501          INCF     FSR0,1
0035 *B1FF          ADDLW    0xff
0036 *9A04          BTFSC    ALUSTA,2
0037  C033          GOTO     0x33
0038      __SEGINIT_END:
0038 *0002          RETURN
0039      exit:
0039      ?C_EXIT:
0039      @cend:
0039  0000          NOP
003A  C039          GOTO     exit
003B      putchar:
003B *0000          NOP
003C *BA00          MOVLR    0x0
003D *6A20          MOVFP    0x20,WREG
```

```
13
21
34
55
```

```
CPU reset executed
Break at .common.c\15   (init_f
  FETCH access at address 0x00A
Break at .common.c\15   (init_f
  FETCH access at address 0x00A
Stop at .tutor.c\20    (do_fore
program EXIT reached
```

To start again with the existing program perform a processor reset by choosing **Reset** from the **Execute** menu, or click the **Reset** button in the toolbar.

EXITING FROM C-SPY

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

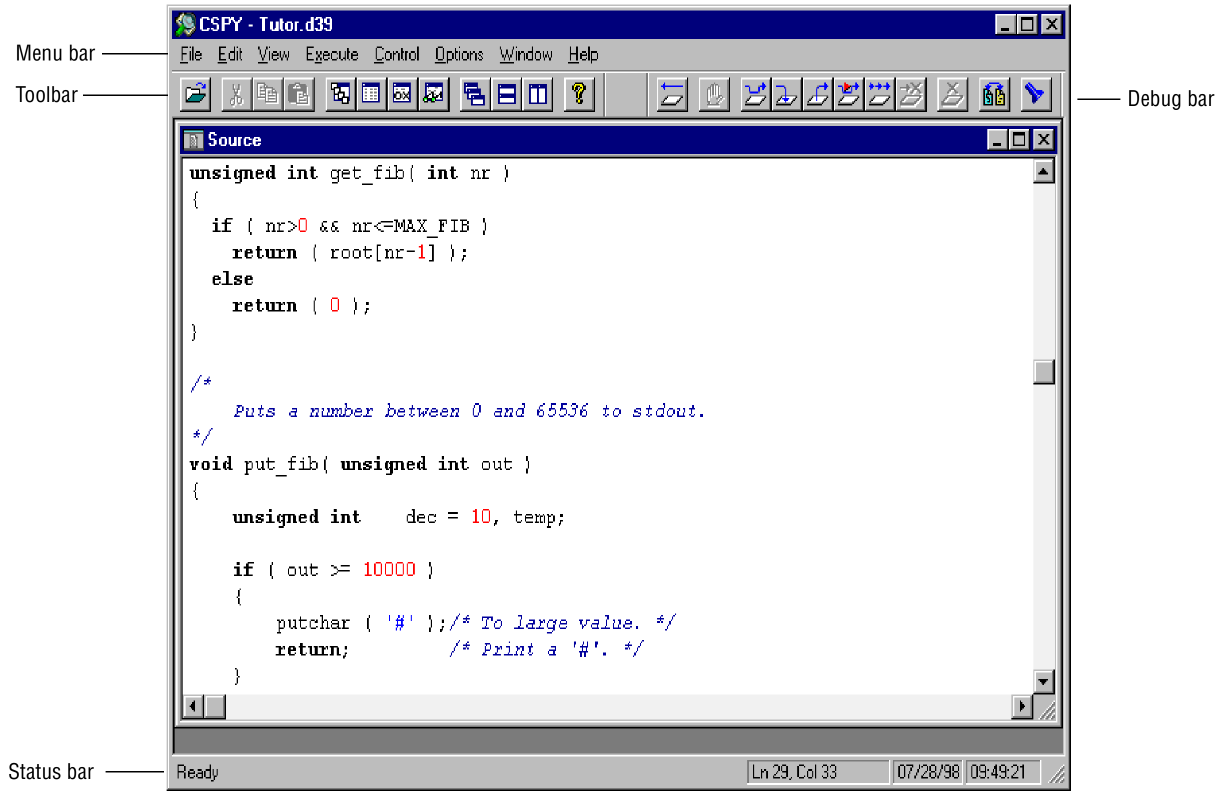
REFERENCE

This chapter provides complete reference information about C-SPY.

It first gives information about the components of the C-SPY window, and each of the different types of window it encloses. It then gives details of the menus, and the commands on each menu.

THE C-SPY WINDOW

The following illustration shows the main C-SPY window:



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

/*
    Puts a number between 0 and 65536 to stdout.
*/
void put_fib( unsigned int out )
{
    unsigned int    dec = 10, temp;

    if ( out >= 10000 )
    {
        putchar ( '#' );/* To large value. */
        return;         /* Print a '#'. */
    }
```

TYPES OF C-SPY WINDOWS

There are a number of C-SPY windows:

- Source window
- Watch window
- Report window
- Register window
- Profiling window
- Terminal I/O window
- Locals window
- Memory window
- Calls window

The windows are described in greater detail on the following pages.

MENU BAR

Gives access to the C-SPY menus:

Menu	*Description*
File	The **File** menu provides commands for opening and closing files, and exiting from C-SPY.
Edit	The **Edit** menu provides commands for use with the Source window.
View	The **View** menu provides commands to allow you to select which windows are displayed in the C-SPY window.
Execute	The **Execute** menu provides commands for executing and debugging the source program. Most of the commands are also available as icon buttons in the debug bar.
Control	The **Control** menu provides commands allowing you to control the execution of the program.

Menu	*Description*
Options	The commands on the **Options** menu allow you to change the configuration of your C-SPY environment, register and display macros.
Window	The **Window** menu lets you select or open C-SPY windows and control the order and arrangement of the windows.
Help	The **Help** menu provides help about C-SPY.

The menus are described in greater detail on the following pages.

TOOLBAR AND DEBUG BAR

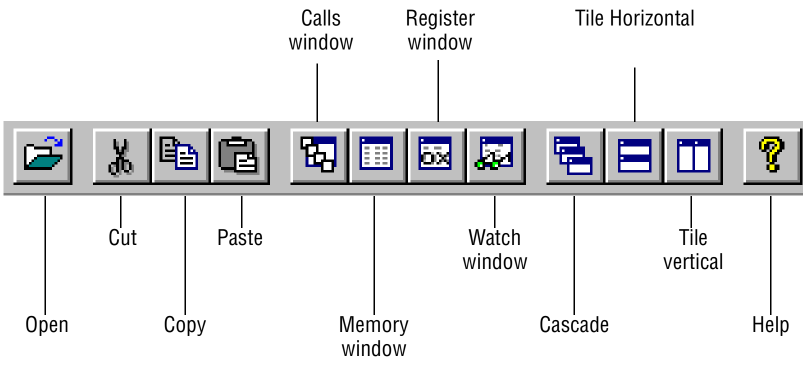
The toolbar and debug bar provide buttons for the most frequently-used commands on the menus.

You can move each bar to a different position in the C-SPY window, or convert it to a floating palette, by dragging it with the mouse.

You can display a description of any button by pointing to it with the mouse pointer. When a command is not available the corresponding button will be grayed out and you will not be able to select it.

Toolbar

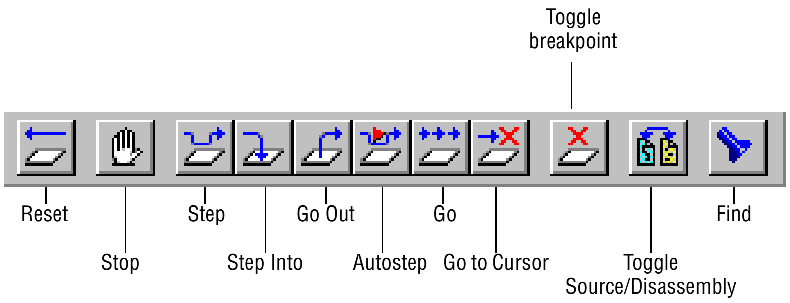
The following diagram shows the command corresponding to each of the toolbar buttons:



You can choose whether the toolbar is displayed using the **Toolbar** command on the **View** menu.

Debug bar

The following diagram shows the command corresponding to each button:



You can choose whether the debug bar is displayed using the **Debug Bar** command on the **View** menu.

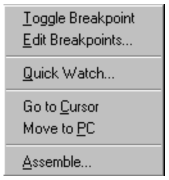
STATUS BAR

Shows help text, and the position of the cursor in the Source window.

You can choose whether the status bar is displayed using the **Status Bar** command on the **View** menu.

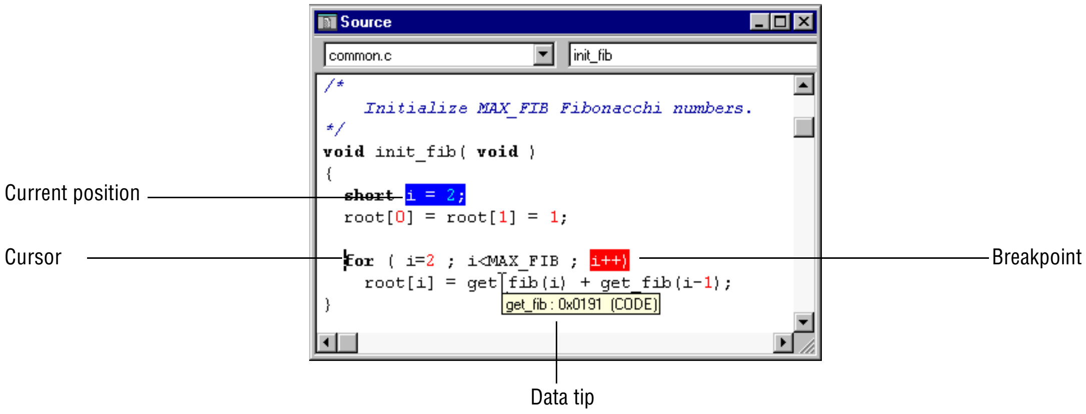
SOURCE WINDOW

The C-SPY Source window shows the source program being debugged, as either C or assembler source code or disassembled program code. You can switch between source and disassembly mode by choosing **Toggle Source/Disassembly** from the **View** menu, or by clicking the **Toggle Source/Disassembly** button in the debug bar.



Clicking the right mouse button in the Source window displays a pop-up menu which gives you access to several useful commands.

The following types of highlighting are used in the Source window:



```
/*
    Initialize MAX_FIB Fibonacchi numbers.
*/
void init_fib( void )
{
  short i = 2;
  root[0] = root[1] = 1;

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

Current position

The current position indicates the next C statement or assembler instruction to be executed, and is highlighted in blue.

Cursor

Any statement in the Source window can be selected by clicking on it with the mouse pointer. The currently-selected statement is indicated by the cursor.

Alternatively, you can move the cursor using the navigation keys.

The **Go to Curso**r command on the **Execute** menu will execute the program from the current position up to the statement containing the cursor.

Breakpoint

C statements or assembler instructions at which breakpoints have been set are highlighted in red in the Source window.

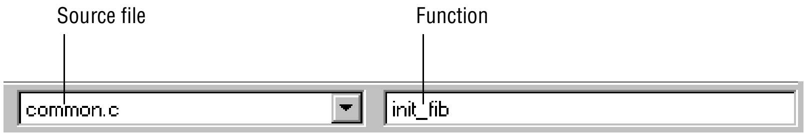
To set a breakpoint choose **Toggle Breakpoint...** from the **Control** menu or click the **Toggle Breakpoint** button in the toolbar.

Data tip

If you position the mouse pointer over a function, variable, or constant name in the C source shown in the Source window, the function start address or the current value of the variable or constant is shown below the mouse pointer.

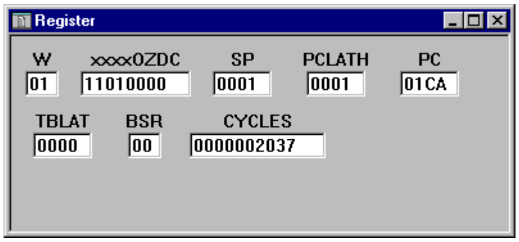
Source bar

The **Source file** and **Function** boxes show the name of the current source file and function displayed in the Source window, and allow you to move to a different file or function by selecting a name from the corresponding drop-down list.



REGISTER WINDOW

The Register window gives a continuously updated display of the contents of the processor registers, and allows you to edit them. When a value changes it is displayed in red.



W	xxxx0ZDC	SP	PCLATH	PC
01	11010000	0001	0001	01CA

TBLAT	BSR	CYCLES
0000	00	0000002037

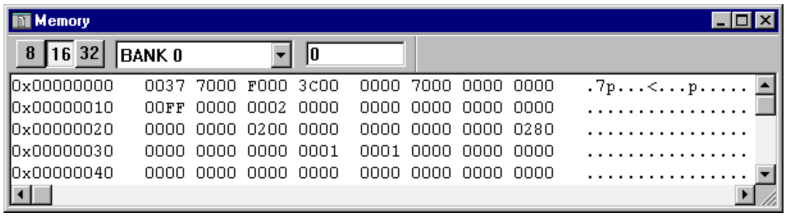
To change the contents of a register, edit the corresponding text box. The register will be updated when you tab to the next register or press ↵.

You can configure the registers displayed in the Register window using the **Register Setup** page in the **Settings** dialog box.

Note that all registers in the Register window are displayed in hexadecimal format, except for the `CYCLES` register which is displayed in decimal format and the `STATUS` which is displayed in binary format.

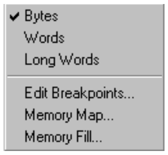
MEMORY WINDOW

The Memory window gives a continuously updated display of a specified block of memory and allows you to edit it.



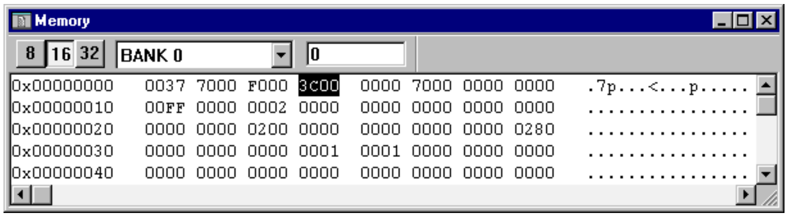
```
0x00000000   0037 7000 F000 3C00   0000 7000 0000 0000   .7p...<...p.....
0x00000010   00FF 0000 0002 0000   0000 0000 0000 0000   ................
0x00000020   0000 0000 0200 0000   0000 0000 0000 0280   ................
0x00000030   0000 0000 0000 0001   0001 0000 0000 0000   ................
0x00000040   0000 0000 0000 0000   0000 0000 0000 0000   ................
```

Choose **8**, **16**, or **32** to display the memory contents in blocks of bytes, words, or long words. Note that the 8-bit and 32-bit buttons are disabled for `CODE` memory as it is word-based.



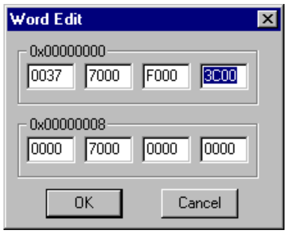
Clicking the right mouse button in the Memory window displays a pop-up menu which gives you access to several useful commands.

To edit the contents of memory double-click the address or value you want to edit:



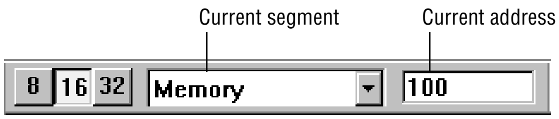
```
0x00000000   0037 7000 F000 3C00   0000 7000 0000 0000   .7p...<...p.....
0x00000010   00FF 0000 0002 0000   0000 0000 0000 0000   ................
0x00000020   0000 0000 0200 0000   0000 0000 0000 0280   ................
0x00000030   0000 0000 0000 0001   0001 0000 0000 0000   ................
0x00000040   0000 0000 0000 0000   0000 0000 0000 0000   ................
```

The following dialog box then allows you to edit the memory:



Memory bar

The following diagram shows the commands corresponding to the memory bar buttons:



The **8**, **16**, or **32** displays the memory contents in blocks of bytes, words, or long words.

The **Current segment** drop-down list indicates what memory segment is currently displayed. To display the contents of another memory segment, choose it from the drop-down list.

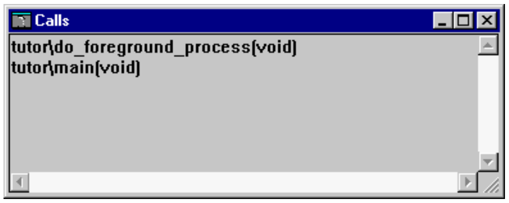
The **Current address** edit-box indicates the current address, i.e. the address being displayed at the top of the memory window. To view the memory of a particular address in the current segment, enter the desired address in hexadecimal format and press ↵.

CALLS WINDOW

Displays the C call stack. Each entry has the format:

module\function(*values*)

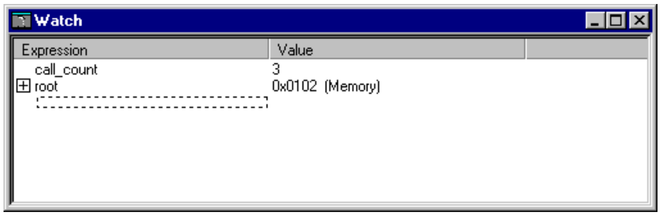
where *values* is a list of the parameter values, or *void* if the function does not take any parameters.



```
tutor\do_foreground_process(void)
tutor\main(void)
```

WATCH WINDOW

Allows you to monitor the values of C expressions or variables:



Expression	Value
call_count	3
⊞ root	0x0102 (Memory)

Viewing the contents of an expression

To view the contents of an expression such as an array, a structure or a union, click ✚ to expand the tree structure.

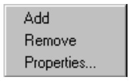
Adding an expression to the Watch window

To add an expression to the Watch window, click in the dotted rectangle, then hold down and release the mouse button.

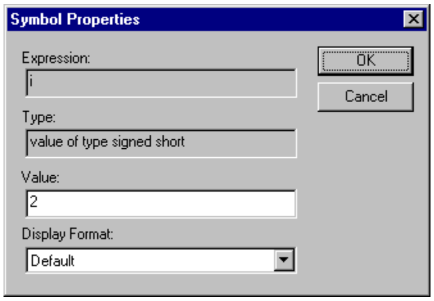
Alternatively, click the right mouse button in the Watch window and choose **Add** from the pop-up menu. Then type the expression followed by ↵.

You can also drag and drop an expression from the Source window.

Inspecting expression properties



Select the expression in the Watch window and choose **Properties...** from the pop-up menu. You can then edit the value of an expression and change the display format in the **Symbol Properties** dialog box:



Expression: i

Type: value of type signed short

Value: 2

Display Format: Default

Removing an expression

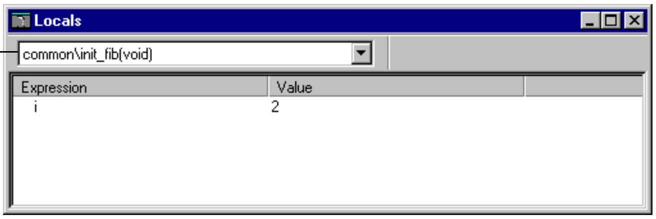
Select the expression and press Delete, or choose **Remove** from the pop-up menu.

When a value changes it is displayed in red.

LOCALS WINDOW

Automatically displays the local variables and their values:

Call stack



```
Expression                    Value
  i                           2
```

Editing the value of a local variable

To change the value of a local variable, click the right mouse button, and choose **Properties...** from the pop-up menu.

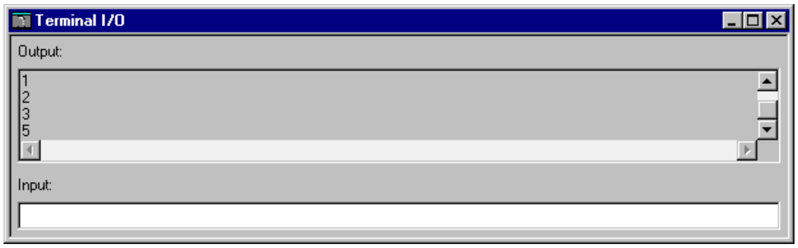
You can then change the value or display format in the **Symbol Properties** dialog box.

Locals bar

The drop-down list displays the **Call stack** with the currently active function on top by default. To see the variables of another function on the call stack, select the function from the drop-down list.

TERMINAL I/O WINDOW

Allows you to enter input to your program, and display output from it.



```
1
2
3
5
```

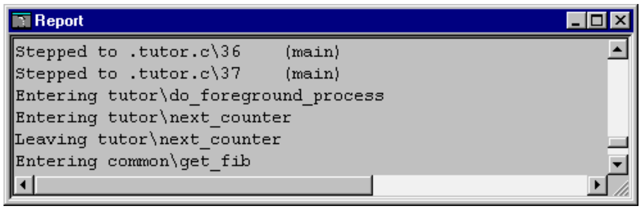
To use the Terminal I/O window you need to link the program with the XLINK option **Debug info with terminal I/O** (`-rt`), to send printing output to the screen instead of to a terminal.

If the Terminal I/O window is open C-SPY will write output to it, and read input from it.

If the Terminal I/O window is closed C-SPY will read a random value.

REPORT WINDOW

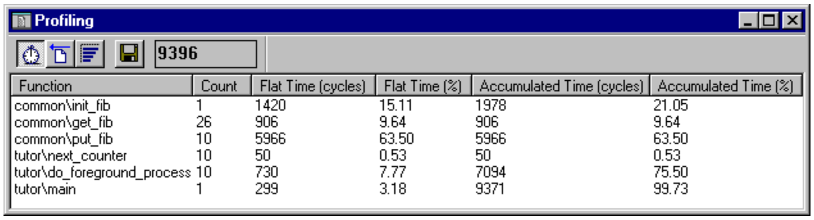
Displays debugger output, such as diagnostic messages and trace information.



```
Stepped to .tutor.c\36    (main)
Stepped to .tutor.c\37    (main)
Entering tutor\do_foreground_process
Entering tutor\next_counter
Leaving tutor\next_counter
Entering common\get_fib
```

PROFILING WINDOW

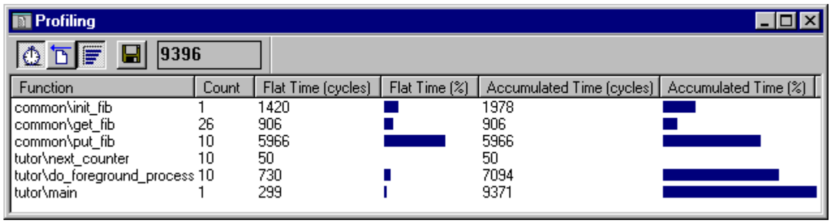
Displays the results of a function profiling session. Function level profiling presents statistics regarding the time spent executing in different functions and the number of times they are called. This information can help you enhance your application's performance by identifying bottlenecks or see where optimizations would be most beneficial.



Function	Count	Flat Time (cycles)	Flat Time (%)	Accumulated Time (cycles)	Accumulated Time (%)
common\init_fib	1	1420	15.11	1978	21.05
common\get_fib	26	906	9.64	906	9.64
common\put_fib	10	5966	63.50	5966	63.50
tutor\next_counter	10	50	0.53	50	0.53
tutor\do_foreground_process	10	730	7.77	7094	75.50
tutor\main	1	299	3.18	9371	99.73

Clicking on the column header sorts the complete list depending on column. Double-clicking on an item in the **Function** column automatically displays the function in the Source window.

The information in the columns **Flat Time** and **Accumulated Time** can be displayed either as digits or as bar charts.


Profiling

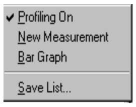
9396

Function	Count	Flat Time (cycles)	Flat Time (%)	Accumulated Time (cycles)	Accumulated Time (%)
common\init_fib	1	1420	▬	1978	▬
common\get_fib	26	906	▬	906	▬
common\put_fib	10	5966	▬	5966	▬
tutor\next_counter	10	50		50	
tutor\do_foreground_process	10	730	▬	7094	▬
tutor\main	1	299	▬	9371	▬

Flat time is the time spent executing in a function, not including further function calls from within that function.

Accumulated time is the time spent executing in a function, including function called from within that function.

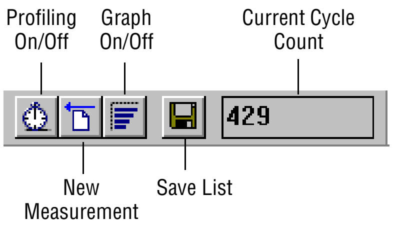
Note: Because of startup code, function entry and exit code combined with compiler optimizations, the percentage does not necessarily add up to 100 %.



Clicking the right mouse button in the profiling window displays a pop-up menu which gives you access to the commands on the profiling bar.

PROFILING BAR

The following diagram shows the command corresponding to each of the profiling bar buttons:



Profiling On/Off

Switches profiling on/off when executing. Profiling can be switched on/off either from the **Control menu** or by clicking on the icon in the **Profiling bar.**

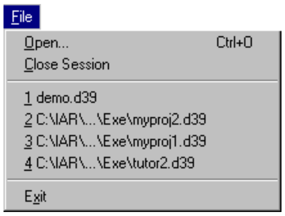
New Measurement
Starts a new measurement. By clicking on the icon the values displayed are reset to zero.

Graph On/Off
Displays the relative numbers as graphs or numbers.

Save List
Saves a list to a file.

Current Cycle Count
Displays the current value of the cycle counter.

FILE MENU


File
Open... Ctrl+O
Close Session
1 demo.d39
2 C:\IAR\...\Exe\myproj2.d39
3 C:\IAR\...\Exe\myproj1.d39
4 C:\IAR\...\Exe\tutor2.d39
Exit

The **File** menu provides commands for opening and closing files, and exiting from C-SPY.

OPEN…

Displays a standard **Open** dialog box to allow you to select a program file to debug.

If another file is already open it will be closed first.

CLOSE SESSION

Closes the current C-SPY session.

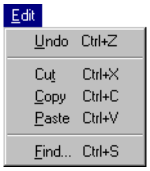
RECENT FILES

Displays a list of the files most recently opened, and allows you to select one to open it.

EXIT

Exits from C-SPY.

EDIT MENU



The **Edit** menu provides commands for use with the Source window.

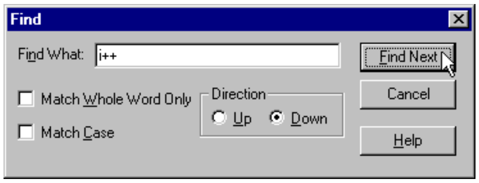
UNDO, CUT, COPY, PASTE

Provides the usual Windows editing features for editing text in some of the windows and dialog boxes.

FIND…

Allows you to search for text in the Source window.

This dialog box allows you to specify the text to search for:



Find What: i++

Direction: [✓ Down]

Enter the text you want to search for in the **Find What** text box.

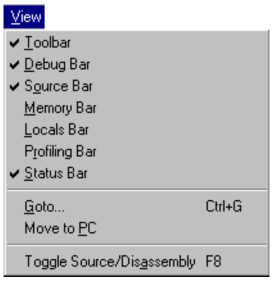
Select **Match Whole Word Only** to find the specified text only if it occurs as a separate word. Otherwise `int` will also find `print`, `sprintf` etc.

Select **Match Case** to find only occurrences that exactly match the case of the specified text. Otherwise specifying `int` will also find `INT` and `Int`.

Select **Up** or **Down** to specify the direction to search.

Choose **Find Next** to start searching. The source pointer will be moved to the next occurrence of the specified text.

VIEW MENU



The **View** menu provides commands to allow you to select which toolbars are displayed in the C-SPY window.

TOOLBAR

Toggles on or off the display of the toolbar, ie the row of buttons below the menu bar.

DEBUG BAR

Toggles on or off the display of the debug bar, ie the second row of buttons below the menu bar.

SOURCE BAR

Toggles on or off the Source window toolbar.

MEMORY BAR

Toggles on or off the Memory window toolbar.

LOCALS BAR

Toggles on or off the Locals window toolbar.

PROFILING BAR

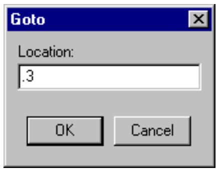
Toggles on or off the Profiling window toolbar.

STATUS BAR

Toggles on or off the display of the status bar, along the bottom of the C-SPY window.

GOTO…

Displays the following dialog box to allow you to move the source pointer to a specified location in the Source window.


Goto

Location: .3

To go to a specified source line, prefix the line number with a period(`.`). For example:

Location	*Description*
`.12`	Moves to line 12 in current file.
`.tutor.c\12`	Moves to line 12 in file `tutor.c`.
`main`	Moves to function `main` in current scope.

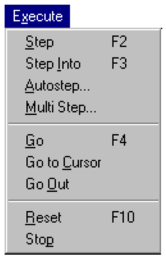
MOVE TO PC

Moves the source pointer to the current PC position in the Source window.

TOGGLE SOURCE/DISASSEMBLY

Switches between source and disassembly mode debugging.

EXECUTE MENU



The **Execute** menu provides commands for executing and debugging the source program. Most of the commands are also available as icon buttons in the debug bar.

STEP

Executes the next statement or instruction, without entering C functions or assembler subroutines.

STEP INTO

Executes the next statement or instruction, entering C functions or assembler subroutines.

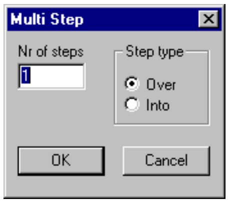
AUTOSTEP...

Steps continuously, with a selectable time delay, until a breakpoint or program exit is detected.

MULTI STEP...

Allows you to execute a specified number of **Step** or **Step Into** commands.

Displays the following dialog box to allow you to specify the number of steps:



1

[✓ Over] Into

Select **Over** to step over C functions or assembler subroutines, or **Into** to step into each C function or assembler subroutine.

Then choose **OK** to execute the steps.

GO

Executes from the current statement or instruction until a breakpoint or program exit is reached.

GO TO CURSOR

Executes from the current statement or instruction up to a selected statement or instruction.

GO OUT

Executes from the current statement up to the statement after the call to the current function.

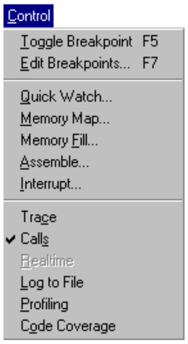
RESET

Resets the target processor.

STOP

Stops program execution or automatic stepping.

CONTROL MENU



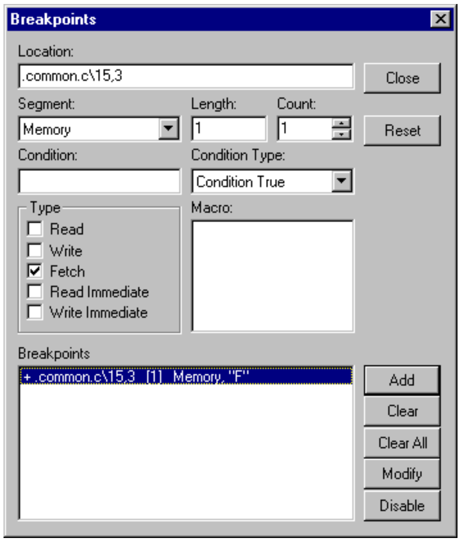
The **Control** menu provides commands to allow you to define breakpoints and change the memory mapping.

TOGGLE BREAKPOINT

Toggles on or off a breakpoint at the statement or instruction containing the cursor in the Source window.

EDIT BREAKPOINTS…

Displays the following dialog box which shows the currently defined breakpoints, and allows you to edit them or define new breakpoints:



Location: .common.c\15,3
Segment: Memory
Length: 1
Count: 1
Condition:
Condition Type: Condition True
Type: Read [ ] Write [ ] Fetch [✓] Read Immediate [ ] Write Immediate [ ]
Macro:
Breakpoints: + .common.c\15,3 [1] Memory, "F"

This dialog box lists the breakpoints you have set with the **Toggle Breakpoint** command, and allows you to define, modify, or remove breakpoints with break conditions.

To define a new breakpoint enter the characteristics of the breakpoint you want to define and choose **Add**.

To modify an existing breakpoint select it in the **Breakpoints** list and choose one of the following buttons:

Choose this	*To do this*
Clear	Clear the selected breakpoint.
Clear All	Clears all the breakpoints in the list.
Modify	Modifies the breakpoint to the settings you select.
Disable/Enable	Toggles the breakpoint on or off. Enabled breakpoints are prefixed with a + in the **Breakpoints** list.

For each breakpoint you can define the following characteristics:

Location
The address in memory or any expression that evaluates to a valid address; eg a function or variable name.

When setting a code breakpoint, you can specify a location in the C source program with the formats:

`.`*`source`*`\`*`line`* `or .`*`line`*

For example, `.common.c\12` sets a breakpoint at the first statement on line 12 in the source file `common.c`.

When setting a data breakpoint, enter the name of a variable or any expression that evaluates to a valid memory location. For example, `my_var` refers to the location of the variable `my_var`, and `arr[3]` refers to the third element of the array `arr`.

Note that you cannot set a breakpoint on a variable that does not have a constant address in memory.

Segment
The memory segment in which the location or address belongs.

Length
The number of bytes to be guarded by the breakpoint.

Count
The number of times that the breakpoint condition must be fulfilled before a break takes place.

Condition
A valid expression conforming to C-SPY expression syntax; see the chapter *C-SPY expressions and macros*. If left blank no condition is evaluated.

Condition type	*Description*
Condition True	The breakpoint is triggered if the value of the expression is true.
Condition Changed	The breakpoint is triggered if the value of the condition expression has changed.

Note that the condition is evaluated only when the breakpoint is encountered.

Type
Specifies the type of memory access guarded by the breakpoint:

Type	*Description*
Read	Read from location.
Write	Write to location.
Fetch	Fetch opcode from location.
Read Immediate	Read from location, immediate break.
Write Immediate	Write to location, immediate break.

The **Read**, **Write**, and **Fetch** breakpoint types never break execution within a single assembler instruction. **Read** and **Write** breakpoints are recorded and reported after the instruction is completed. If a **Fetch** breakpoint is detected on the first byte of an instruction, it will be reported before the instruction is executed; otherwise the breakpoint is reported after the instruction is completed.

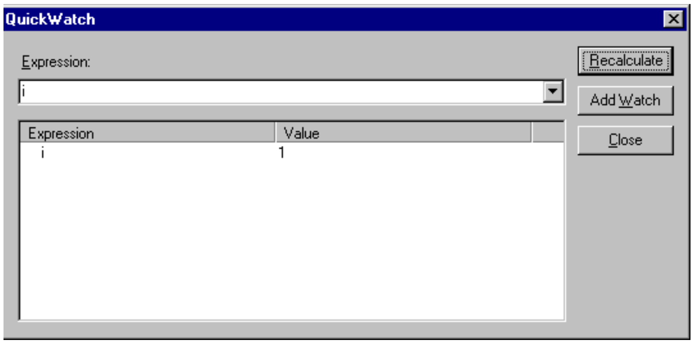
The **Read Immediate** and **Write Immediate** breakpoint types are only applicable to simulators and will cause a break as soon as encountered, even in the middle of executing an instruction. Execution will automatically continue, and the only action is to execute the associated macro. They are provided to allow you to simulate the behavior of a port. For example, you can set a **Read Immediate** breakpoint at a port address, and assign a macro to the breakpoint that reads a value from a file and writes it to the port location.

Macro
An expression to be executed once the breakpoint is activated.

QUICK WATCH…

Allows you to watch the value of a variable or expression.

Displays the following dialog box to allow you to specify the expression to watch:



Enter the C-SPY variable or expression you want to evaluate in the **Expression** box. Alternatively, you can select an expression you have previously watched from the drop-down list. For detailed information about C-SPY expressions, see *C-SPY expressions*, page 67.

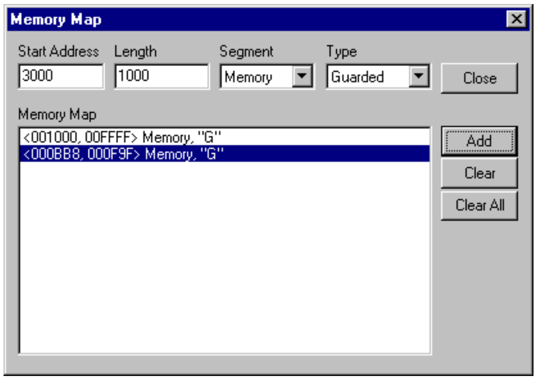
Then choose **Recalculate** to evaluate the expression, or **Add Watch** to evaluate the expression and add it to the Watch window.

Choose **Close** to close the **Quick Watch** dialog box.

MEMORY MAP…

Allows you to define a map of the memory space in the system. C-SPY will then check accesses to memory and execution stops if a violation occurs.

The following dialog box is displayed to show the current memory map and allow you to modify it:



Start Address: 3000
Length: 1000
Segment: Memory
Type: Guarded

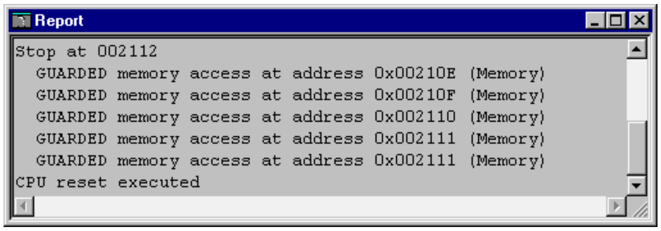
```
<001000, 00FFFF> Memory, "G"
<000BB8, 000F9F> Memory, "G"
```

To define a new segment enter the **Start Address**, **Length**, and **Segment** and choose the **Type** according to the following table:

Type	*Description*
Guarded	Simulates addresses with no memory by flagging all accesses as illegal.
Protected	Simulates ROM memory by flagging all write accesses to this address as illegal.

To delete an existing segment select it in the **Memory Map** list and choose **Clear**.

Illegal accesses are reported in the Report window:

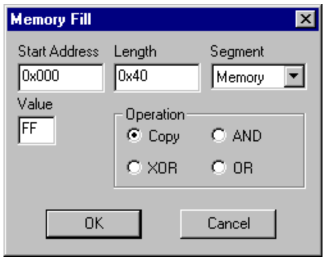


```
Stop at 002112
  GUARDED memory access at address 0x00210E (Memory)
  GUARDED memory access at address 0x00210F (Memory)
  GUARDED memory access at address 0x002110 (Memory)
  GUARDED memory access at address 0x002111 (Memory)
  GUARDED memory access at address 0x002111 (Memory)
CPU reset executed
```

MEMORY FILL…

Allows you to fill a specified area of memory with a value.

The following dialog box is displayed to allow you to specify the area to fill:



Start Address: 0x000
Length: 0x40
Segment: Memory
Value: FF
Operation: [✓ Copy] AND XOR OR

Enter the **Start Address** and **Length**, and select the segment type from the drop-down **Segment** menu.

Enter the **Value** to be used to fill each memory location, and select the logical operation. The default is **Copy**, but you can choose one of the following operations:

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

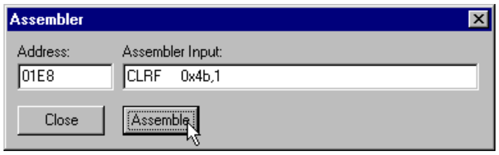
Finally choose **OK** to proceed with the memory fill.

ASSEMBLE…

Displays the assembler mnemonic for a machine-code instruction, and allows you to modify it and assemble it into memory.

Assemble… is only available in disassembly mode debugging. If necessary, choose **Toggle Source/Disassembly** from the **View** menu to change mode.

Then double-click a line in the Source window, or position the cursor in the line and choose **Assemble…**. This dialog box shows the address and assembler instruction at that address:



Address: 01E8
Assembler Input: CLRF 0x4b,1

To modify the instruction, edit the text in the **Assembler Input** box and click **Assemble**.

You can also enter an address in the **Address** box, and press Tab, to display the assembler instruction at a different address.

INTERRUPT…

The interrupt simulation can be used in conjunction with macros and complex breakpoints to simulate interrupt-driven ports. For example, to simulate port input, specify an interrupt that will cause the appropriate interrupt handler to be called, set a breakpoint at the entry of the interrupt handler routine, and associate it with a macro that sets up the input data by reading it from a file or by generating it using an appropriate algorithm.

Note that C-SPY only polls for interrupts between instructions, regardless of how many cycles an instruction takes.

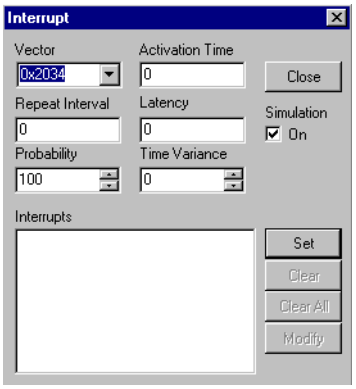
The C-SPY interrupt system uses the cycle counter as a clock to determine when an interrupt should be raised in the simulator. Changing the cycle counter will affect any interrupts you have set up in the **Interrupt** dialog box.

Performing a C-SPY reset will reset the cycle counter, and any interrupt orders with a fixed activation time will be cleared.

For example, consider the case where the cycle counter is 123456, a repeatable order raises an interrupt every 4000 cycles, and a single order is about to raise an interrupt at 123500 cycles.

After a system reset the repeatable interrupt order remains and will raise an interrupt every 4000 cycles, with the first interrupt at 4000 cycles. The single order is removed.

The **Interrupt...** command displays the following dialog box to allow you to configure C-SPY's interrupt simulation:



Vector: 0x2034
Activation Time: 0
Repeat Interval: 0
Latency: 0
Probability: 100
Time Variance: 0
Simulation: [✓ On]

To define a new interrupt enter the characteristics of the interrupt you want to simulate and choose **Set**.

To edit an existing interrupt select it in the **Interrupts** list and choose **Modify** to display or edit its characteristics, or **Clear** to delete it. Note that deleting an interrupt does not remove any pending interrupt from the system.

For each interrupt you can define the following characteristics:

Vector
Defines the interrupt vector. A list of possible predefined interrupt vectors are available under the pull-down menu. Possible values depend on the version of the PICmicro™ microcontroller you are simulating; refer to the `cwpic.txt` file for full details.

Activation Time
The time, in cycles, after which the specified type of interrupt can be generated.

Repeat Interval
The periodicity in cycles of the interrupt.

Latency
Describes how long, in cycles, the interrupt remains pending until removed if it has not been processed. Latency is not relevant for the PICmicro™ microcontroller.

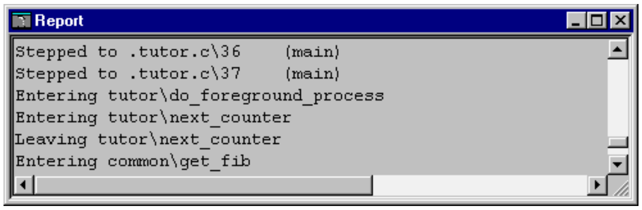
Probability
The probability, in percent, that the interrupt will actually appear in a period.

Time Variance
A timing variation range, as a percentage of the repeat interval, in which the interrupt may occur for a period. For example, if the repeat interval is 100 and the jitter 5 %, the interrupt may occur anywhere between T = 95 and T = 105, to simulate a variation in the timing.

Simulation On/Off
Enables or disables interrupt simulation. If the interrupt is disabled the definition remains but no interrupts will be generated.

TRACE

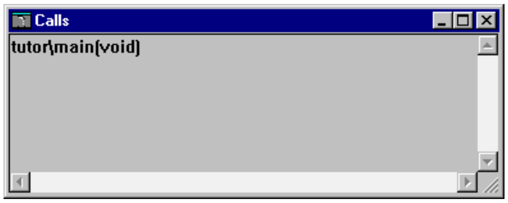
Toggles trace mode on or off. When trace mode is on each step and function call is listed in the Report window:



```
Stepped to .tutor.c\36     (main)
Stepped to .tutor.c\37     (main)
Entering tutor\do_foreground_process
Entering tutor\next_counter
Leaving tutor\next_counter
Entering common\get_fib
```

CALLS

Toggles calls mode on or off. When calls mode is on function calls are listed in the Calls window:



```
tutor\main(void)
```

REALTIME

Reserved for Emulator and ROM-monitor versions of C-SPY. For information about the C-SPY versions available, please contact your IAR representative.

LOG TO FILE

Toggles writing to the log file on or off. When log file mode is on, the contents of the Report window are logged to a file. Choose **Select Log File…** from the **Options** menu to enable the log file function.

PROFILING

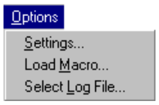
Toggles profiling on or off. For further information regarding profiling, see *Profiling window*, page 43.

CODE COVERAGE

Reports the current code coverage status to the Report window. The report contains the functions that have not been called and the statements that have not been executed so far in program execution.

Code coverage is reset when the processor is reset.

OPTIONS MENU

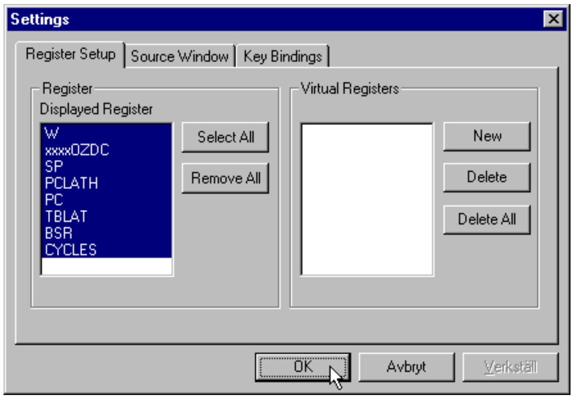


The commands on the **Options** menu allow you to change the configuration of your C-SPY environment, and register macros.

SETTINGS...

Displays the **Settings** dialog box to allow you to define the colors and fonts used in the windows, and set up registers.

Register Setup
Allows you to specify which registers are displayed in the Register window and to define virtual registers.

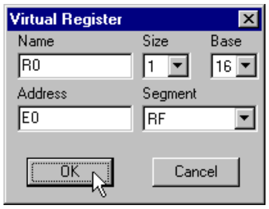


To specify which registers are displayed in the Register window select them in the **Displayed Register** list and click **OK**.

Click **Select All** or **Remove All** to select or deselect all the registers.

Virtual registers allow you to specify any memory location and have them displayed in the Register window, in addition to the standard registers.

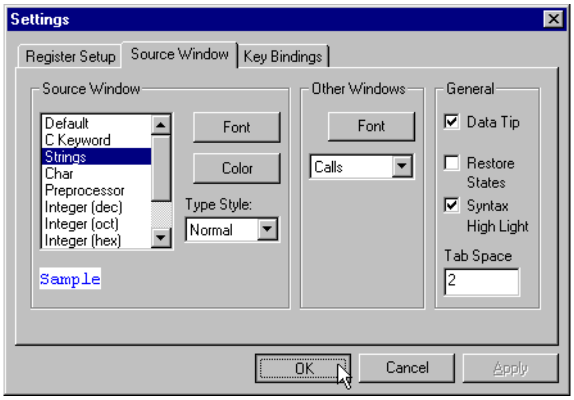
To define virtual registers click **New** to display the **Virtual Register** dialog box:


Virtual Register

Name: R0
Size: 1
Base: 16
Address: E0
Segment: RF

Enter the **Name** and **Address** for the virtual register, and select the **Size** in bytes, **Base,** and **Segment** from the drop-down menus. Then choose **OK** to define the register. It will be displayed in the Register window after any standard registers you have selected.

Source Window
Allows you to specify the colors and fonts used for text in the Source window, the font used for text in the other windows, and several general settings:



To specify the style used for each element of C syntax in the Source window select the item you want to define from the **Source Window** list. The current setting is shown by the **Sample** below the list box.

You can choose a text color by clicking **Color**, and a font by clicking **Font**. You can also choose the type style from the **Type Style** drop-down menu.

To specify the font used in the other windows choose the window from the drop-down list box and click **Font**.

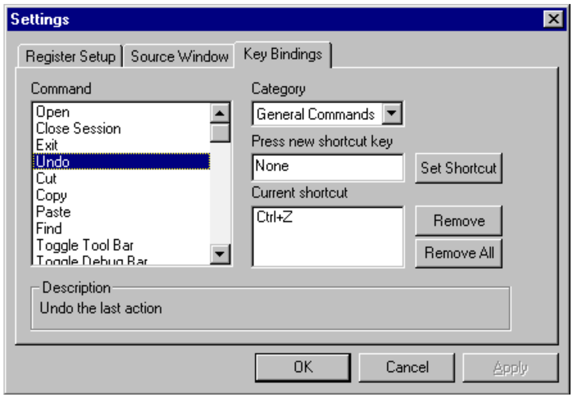
You can also specify the following general settings:

Settings	*Description*
Data Tip	Shows the current value or function start address when the mouse pointer is moved over a variable constant, or function name in the Source window.
Restores States	Restores breakpoints, memory maps, and interrupts between sessions.

Settings	*Description*
Syntax Highlight	Highlights the syntax of C programs in the Source window.
Tab Space	Specifies the number of spaces used to expand tabs.

Then choose **OK** to use the new styles you have defined, or **Cancel** to revert to the previous styles.

Key Bindings
Displays the shortcut keys used for each of the menu options, and allows you to change them.



Command: Open, Close Session, Exit, Undo, Cut, Copy, Paste, Find, Toggle Tool Bar, Toggle Debug Bar
Category: General Commands
Press new shortcut key: None
Current shortcut: Ctrl+Z
Description: Undo the last action

To define a shortcut key select the command category from the **Category** list, and then select the command you want to edit in the **Command** list. Any currently defined shortcut keys are shown in the **Current shortcut** list.

To add a shortcut key to the command click in the **Press new shortcut key** box and type the key combination you want to use. Then click **Set Shortcut** to add it to the **Current shortcut** list. You will not be allowed to add it if it is already used by another command.

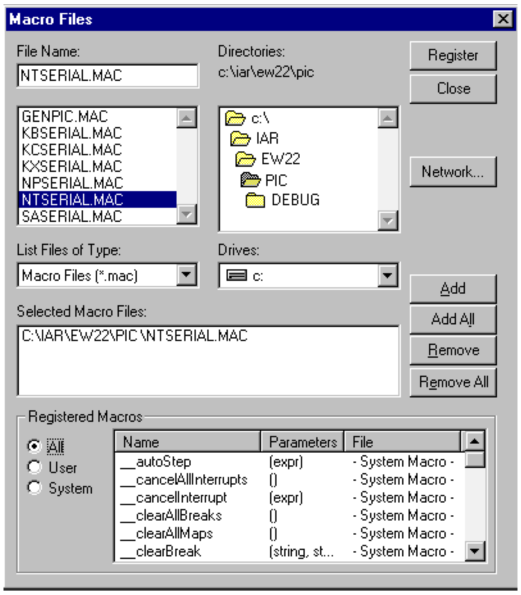
To remove a shortcut key select it in the **Current shortcut** list and click **Remove**, or click **Remove All** to remove all the command's shortcut keys.

Then choose **OK** to use the key bindings you have defined, and the menus will be updated to show the new shortcuts.

You can set up more than one shortcut for a command, but only one will be displayed in the menu.

LOAD MACRO...

Displays the following dialog box, to allow you to specify a list of files from which to read macro definitions into C-SPY:



File Name: NTSERIAL.MAC

Directories: c:\iar\ew22\pic

GENPIC.MAC
KBSERIAL.MAC
KCSERIAL.MAC
KXSERIAL.MAC
NPSERIAL.MAC
NTSERIAL.MAC
SASERIAL.MAC

c:\
IAR
EW22
PIC
DEBUG

List Files of Type: Macro Files (*.mac)

Drives: c:

Selected Macro Files: C:\IAR\EW22\PIC \NTSERIAL.MAC

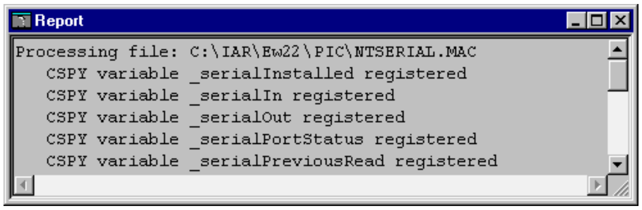
Registered Macros: [✓ All] User System

Name	Parameters	File
__autoStep	(expr)	- System Macro -
__cancelAllInterrupts	()	- System Macro -
__cancelInterrupt	(expr)	- System Macro -
__clearAllBreaks	()	- System Macro -
__clearAllMaps	()	- System Macro -
__clearBreak	(string, st...	- System Macro -

Select the macro definition files you want to use in the file selection list, and click **Add** to add them to the **Selected Macro Files** list or **Add All** to add all the listed files.

You can remove files from the **Selected Macro Files** list using **Remove** or **Remove All**.

Once you have selected the macro definition files you want to use click **Register** to register them, replacing any previously-defined macros or variables. The macros are listed in the Report window as they are registered:



```
Processing file: C:\IAR\Ew22\PIC\NTSERIAL.MAC
   CSPY variable _serialInstalled registered
   CSPY variable _serialIn registered
   CSPY variable _serialOut registered
   CSPY variable _serialPortStatus registered
   CSPY variable _serialPreviousRead registered
```

Registered macros are also displayed in the scroll window under **Registered Macros**.

Clicking on either **Name** or **File** under **Registered Macros** displays the column contents sorted by macro names or by file. Clicking once again sorts the contents in the reverse order. Selecting **All** displays all macros, selecting **User** displays all user macros, and selecting **System** displays all system macros.

Double-clicking on a user-defined macro in the **Name** column automatically opens the file in Notepad, where it is available for editing.

Click **Close** to exit the **Macro Files** window.

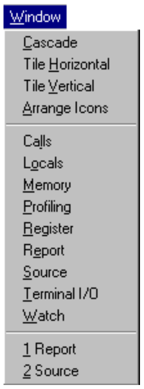
SELECT LOG FILE...

Allows you to log input and output from C-SPY to a file. It displays a dialog box to allow you to select the name and the location of the log file.

Browse to a suitable folder and type in a filename; the default extension is `.log`. Then click **Save** to select the specified file.

Choose **Log to File** in the **Control** menu to turn on or off the logging to the file.

WINDOW MENU



The first section of the **Window** menu contains commands to let you control the order and arrangement of the C-SPY windows.

The central section of the menu lists each of the C-SPY windows. Select a menu command to open the corresponding window.

The last section of the menu lists the currently-opened windows. Selecting a window makes it the currently-active window.

CASCADE

Rearranges the windows in a cascade on the screen.

TILE HORIZONTAL

Tiles the windows horizontally in the main C-SPY window.

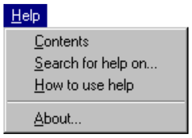
TILE VERTICAL

Tiles the windows vertically in the main C-SPY window.

ARRANGE ICONS

Tidies minimized window icons in the main C-SPY window.

HELP MENU



Provides help about C-SPY.

CONTENTS

Displays the Contents page for help about C-SPY.

SEARCH FOR HELP ON…

Allows you to search for help on a keyword.

HOW TO USE HELP

Displays help about using help.

ABOUT…

Displays the version number of C-SPY for Windows.

C-SPY EXPRESSIONS AND MACROS

In addition to C symbols defined in your program, C-SPY allows you to define C-SPY variables and macros, and to use them when evaluating expressions. Expressions that are built with these components are called C-SPY expressions and can be used in the Watch and QuickWatch windows and in C-SPY macros.

The comprehensive macro capabilities provided in C-SPY allow you to automate the debugging process and simulate peripheral devices. Macros can be used in conjunction with breakpoints and interrupt simulation to perform a wide variety of tasks.

This chapter describes the C-SPY expression syntax extensions and defines the syntax of C-SPY variables and macros.

C-SPY EXPRESSIONS

C-SPY expressions can include any type of C expression, apart from function calls. Symbols used in expressions can be:

- C symbols.
- C-SPY variables.
- C-SPY macros.
- Assembler symbols; ie CPU register names and assembler labels.

C symbols can be referenced by their names or using an extended C-SPY format which allows you to reference symbols outside the current scope.

Expression	*What it means*
`i`	C variable `i` in the current scope or C-SPY variable `i`.
`\i`	C variable `i` in the current function.
`\func\i`	C variable `i` in the function `func`.
`mod\func\i`	C variable `i` in the function `func` in the module `mod`.

Note that when using the module name to reference a C symbol, the module name must be a valid C identifier or it must be encapsulated in backquotes ' (ASCII character 0x60), for example:

```
nice_module_name\func\i
'very strange () module + - name'\func\i
```

In case of a name conflict, C-SPY variables have a higher precedence than C variables. Extended C-SPY format can be used to solve such ambiguities.

Examples of valid C-SPY expressions are:

```
i = my_var * my_mac() + #asm_label
another_mac(2, my_var)
mac_var = another_module\another_func\my_var
```

ASSEMBLER SYMBOLS

Assembler symbols can be used in C expressions if they are preceded by #. These symbols can be assembler labels or CPU register names.

Example	*What it does*
`#pc++`	Increments the value of the program counter.
`myptr = #main`	Sets `myptr` to point to label `main`.

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

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

C-SPY VARIABLES

A C-SPY variable can be defined in a macro definition file with the syntax:

```
VAR name1, name2 …
```

and it is initialized to signed integer zero.

When the value is assigned to a C-SPY variable it is given both a value and a type. For example:

Expression	*What it means*
`myvar = 3.5`	`myvar` is now type float, value `3.5`.
`myvar = (int*)i`	`myvar` is now type pointer to `int`, and the value is the same as `i`.

FORMAT SPECIFIERS

The following format specifiers can be used in the **Display Format** drop-down menu in the **Symbol Properties** dialog box and in a macro `message` statement:

Specifier	*What it means*
`%c`	Char format.
`%o`	Unsigned octal format.
`%u`	Unsigned decimal format.
`%d`	Signed decimal format.
`%X`	Unsigned hexadecimal format.
`%f`	Float format `[-]ddd.ddd`.
`%s`	String format.
`%p`	Pointer format.

The precision for formats specified with `f` is seven or 15 decimals for four and eight byte floats.

Strings with format `s` are printed in quotation marks. If no `NULL` character (`'\0'`) is found within 1000 characters, the printout will stop without a final quotation mark.

C-SPY MACROS

The C-SPY macros work in a similar way to C functions, and for convenience they follow the C language naming conventions and statement syntax as closely as possible.

USING MACROS

C-SPY allows you to define both macro variables (global or local) and macro functions. In addition, several pre-defined system macro variables and macro functions are provided which return information about the system status, and perform complex tasks such as opening or closing files, file I/O operations, etc. For full details of the system macros, see the chapter *System macros*.

By clicking on **Load Macro** the macros available will be displayed under **Registered Macros**, see *Load Macro…*, page 63.

Defining macros

To define a macro variable or macro function first create a text file containing its definition, using any suitable text editor such as the Embedded Workbench editor. Then register the file by choosing **Load Macro…** from the **Options** menu. For more information see *Load Macro…*, page 63. Macros can also be registered using a system macro.

Executing C-SPY macros

You can assign values to a macro variable, or execute a macro function, using the **Quick Watch…** command on the **Control** menu, or from within another C-SPY macro including setup macros. For details of the setup macros see *C-SPY setup macros*, page 74. A macro can also be executed if it is associated with a breakpoint that is activated.

MACRO VARIABLES

A macro variable is a variable defined and allocated outside the user program space. It can then be used in a C-SPY expression.

The command to define a macro variable(s) has the following form:

`var` *`nameList`*`;`

where *`nameList`* is a list of C-SPY variable names separated by commas.

A macro variable defined outside a macro body has global scope, and retains its value and type through the whole debugging session; a macro variable defined within a macro body is created when its definition is executed and deallocated on return from the macro.

By default a macro variable is initialized to signed integer 0. When a C-SPY variable is assigned a value in an expression its type is also converted to the type of the operand.

A complex type (`struct` or `union`) cannot be assigned to a macro variable but a macro variable can contain an address to such an object.

MACRO FUNCTIONS

C-SPY macro functions consist of a series of C-SPY variable definitions and macro statements which are executed when the macro is called. An unlimited number of parameters can be passed to a macro, and macros can return a value on exit.

A C-SPY macro has the following form:

```
macroName (parameterList)
{
  macroBody
}
```

where *`parameterList`* is a list of macro formal names separated by commas, and *`macroBody`* is any series of C-SPY variable definitions and C-SPY statements.

No type checking is performed on the values passed to the macro parameters. When an `array`, `struct`, or `union` is passed only the address of it is passed.

MACRO STATEMENTS

The following C-SPY macro statements are accepted:

Expressions

```
expression;
```

Conditional statements

```
if (expression)
  statement
```

```
if (expression)
  statement
else
  statement
```

Loop statements

```
for (init_expression; cond_expression; after_expression)
  statement
```

```
while (expression)
  statement
```

```
do
  statement
while (expression);
```

Return statements

```
return;
```

```
return (expression);
```

If the return value is not explicitly set by default, `signed int 0` is returned.

Blocks

```
{
  statement1
  statement2
  .
  .
  .
  statementN
}
```

In the above *expression* means a C-SPY expression; statements are expected to behave in the same way as corresponding C statements would do.

Printing messages

The `message` statement allows you to print messages while executing a macro. Its definition is as follows:

```
message argList;
```

where *`argList`* is a list of C-SPY expressions or strings separated by commas. The value of expression arguments or strings are printed to the Report window.

It is possible to override the default display format of an element in *`argList`* by suffixing it with a `:` followed by a format specifier, for example:

```
message int1:%X, int2;
```

will print `int1` in hexadecimal format and `int2` in default format (decimal for an integer type).

Resume statement

The `resume` statement allows you to resume execution of a program after a breakpoint is encountered. For example, specifying:

```
resume;
```

in a breakpoint macro will resume execution after the breakpoint.

Error handling in macros

Two types of error can occur while a macro is being executed:

- Stop errors, which stop execution.
- Minor errors, which cause the macro to return an error number.

Stop errors are caused by mismatched macro parameter types, missing parameters, illegal addresses when setting a breakpoint or map, or illegal interrupt vectors when setting up an interrupt. They are handled by the C-SPY error handler, and execution stops with an appropriate error message.

Minor errors are caused by actions such as failing to open a file, or cancelling a non-existent interrupt. You can test for minor errors by checking the value returned by the system macro; zero indicates successful execution, and any other value is a C-SPY error number.

C-SPY SETUP MACROS

The setup macros are reserved macro names which will be called by C-SPY at specific stages during execution. To use them you should create and register a macro with the name specified in the following table:

Macro	*Description*
`execUserInit()`	Called before communication with the target system is established. Implement this macro to choose the processor option, if it has not already been selected using the Embedded Workbench or C-SPY command line option, and perform other initialization; for example, port initialization for the emulator and ROM-monitor variants. Note that since there is still no code loaded, you cannot, for example, set a breakpoint from this macro.
`execUserPreload()`	Called after communication with the target system is established but before downloading the target program. Implement this macro to initialize memory locations and/or registers which are vital for loading data properly.
`execUserSetup()`	Called once after the target program is downloaded. Implement this macro to set up the memory map, breakpoints, interrupts, register macro files, etc.
`execUserReset()`	Called each time the reset command is issued. Implement this macro to set up and restore data.
`execUserExit()`	Called each time the program is about to exit. Implement this macro to save status data, etc.
`execUserTrace()`	Called each time C-SPY issues a trace printout (when the `Trace` command is active).

SYSTEM MACROS

This chapter lists the built-in system macros provided with C-SPY.

System macro names start with double underscore and are reserved names.

The following system macros are currently defined.

For information on how to load system macros, see *Load Macro…*, page 63.

__cancelAllInterrupts

Cancels all ordered interrupts.

SYNTAX

`__cancelAllInterrupts ()`

RETURN VALUE

`int 0.`

EXAMPLE

`__cancelAllInterrupts ();`

__cancelInterrupt

Cancels an interrupt.

SYNTAX

`__cancelInterrupt` *`(interrupt_id)`*

PARAMETERS

`interrupt_id`	The value returned by the corresponding `__orderInterrupt` macro call (integer).

RETURN VALUE

Result	*Value*
Successful	`int 0.`
Unsuccessful	Non-zero error number.

__clearAllBreaks

Clears all user-defined breakpoints.

SYNTAX

```
__clearAllBreaks ()
```

RETURN VALUE

`int 0.`

EXAMPLE

```
__clearAllBreaks ();
```

__clearAllMaps

Clears all user-defined memory mapping.

SYNTAX

```
__clearAllMaps ()
```

RETURN VALUE

`int 0.`

EXAMPLE

```
__clearAllMaps ();
```

__clearBreak

Clears a given breakpoint.

SYNTAX

```
__clearBreak (address, segment, access)
```

PARAMETERS

address	The breakpoint location (string).
segment	The memory segment name (string).
access	The memory access type (string); concatenation of any of `"R"`, `"W"`, `"F"`, `"I"`, or `"O"`.

RETURN VALUE

Result	*Value*
Successful	`int 0.`
Unsuccessful	Non-zero error number.

EXAMPLE

```
__clearBreak (".demo\\12", "CODE","RWF");
```

__clearMap

Clears a given memory mapping.

SYNTAX

`__clearMap (`*`address, segment`*`)`

PARAMETERS

address	The address (integer).
segment	The memory segment name (string).

RETURN VALUE

Result	*Value*
Successful	`int 0.`
Unsuccessful	Nonzero error number.

EXAMPLE

```
__clearMap (0x1234, "CODE");
```

__closeFile

Closes a file previously opened by `__openFile`.

SYNTAX

```
__closeFile (filehandle)
```

PARAMETERS

`filehandle`	The macro variable used as filehandle by the `__openFile` macro.

RETURN VALUE

`int 0.`

EXAMPLE

```
__closeFile (filehandle);
```

__disableInterrupts

Disables the generation of interrupts.

SYNTAX

```
__disableInterrupts ()
```

RETURN VALUE

Result	*Value*
Successful	`int 0.`
Unsuccessful	Non-zero error number.

EXAMPLE

```
__disableInterrupts ();
```

__enableInterrupts

Enables the generation of interrupts.

SYNTAX

```
__enableInterrupts ()
```

RETURN VALUE

Result	*Value*
Successful	`int 0.`
Unsuccessful	Non-zero error number.

EXAMPLE

```
__enableInterrupts ();
```

__getLastMacroError

Returns the last macro error code (excluding stop errors).

SYNTAX

```
__getLastMacroError ()
```

RETURN VALUE

Value of the last system macro error code.

EXAMPLE

```
__getLastMacroError ();
```

__openFile

Opens a file for I/O operations.

SYNTAX

`__openFile` *`(filehandle, filename, access)`*

PARAMETERS

`filehandle`	The macro variable to contain the file handle.
`filename`	The filename as a string.
`access`	The access type (string); one of the following

`"r"`	ASCII read.
`"rb"`	Binary read.
`"w"`	ASCII write.
`"wb"`	Binary write.

RETURN VALUE

Result	*Value*
Successful	`int 0.`
Unsuccessful	Non-zero error number.

EXAMPLE

```
var filehandle;
__openFile (filehandle, "C:\\TESTDIR\\TEST.TST", "r");
```

__orderInterrupt

Orders an interrupt.

SYNTAX

```
__orderInterrupt (address, activation_time,
repeat_interval, jitter, latency, probability)
```

PARAMETERS

address	The interrupt vector (string).
activation_time	The activation time in cycles (integer).
repeat_interval	The periodicity in cycles (integer).
jitter	The timing variation range (integer between 0 and100).
latency	The latency (integer).
probability	The probability in percent (integer between 0 and 100).

RETURN VALUE

The macro returns an interrupt identifier (`unsigned long`).

EXAMPLE

```
__orderInterrupt ("0x2000", 5000, 1500, 50, 0, 75);
```

__printLastMacroError

Prints the last system macro error message (excluding stop errors) to the Report window.

SYNTAX

```
__printLastMacroError ()
```

RETURN VALUE

`int 0.`

EXAMPLE

```
__printLastMacroError ();
```

__processorOption

Sets a given processor option.

SYNTAX

`__processorOption (`*`procOption`*`)`

PARAMETERS

procOption	The processor option given in the same way it would have been given on the command line (string).

RETURN VALUE

`int 0.`

DESCRIPTION

This macro can only be called from the `execUserInit()` macro.

EXAMPLE

```
__processorOption ("16c16");
```

__readFile

Reads from a file.

SYNTAX

`__readFile (filehandle)`

PARAMETERS

filehandle	The macro variable used as the file handle by the `__openFile` macro.

RETURN VALUE

The return value depends on the access type of the file.

In ASCII mode a series of hex digits, delimited by space, are read and converted to an `unsigned long`, which is returned by the macro.

In binary mode one byte is read and returned.

DESCRIPTION

When the end of the file is reached, the file is rewound and a message is printed in the Report window.

EXAMPLE

Assuming a file was opened with `r` access type containing the following data:

```
1234 56 78
```

Calls to `__readFile()` would return the numeric values `0x1234`, `0x56`, and `0x78`.

__readFileGuarded

Reads from a file.

SYNTAX

`__readFileGuarded (filehandle, errorstatus)`

PARAMETERS

filehandle	The macro variable used as the file handle by the `__openFile` macro.
errorstatus	A C-SPY variable to contain the error status.

RETURN VALUE

Result	*Value*
Successful	The value read.
Unsuccessful	`-1L`

DESCRIPTION

This macro works in exactly the same way as `__readFile`, except that when the end of the file is encountered `-1L` is returned, and the value of *`errorstatus`* is set to the corresponding error number.

__readMemoryByte

Reads one byte from a given memory location.

SYNTAX

```
__readMemoryByte (address, segment)
```

PARAMETERS

address	The memory address (integer).
segment	The memory segment name (string).

RETURN VALUE

The macro returns the value from memory.

EXAMPLE

```
__readMemoryByte (0x200, "CODE");
```

__registerMacroFile

Registers macros from a specified macro file.

SYNTAX

`__registerMacroFile (`*`filename`*`)`

PARAMETERS

filename	A file containing the macros to be registered (string).

RETURN VALUE

`int 0.`

EXAMPLE

```
__registerMacroFile ("c:\\testdir\\macro.mac");
```

__resetFile

Resets the file previously opened by __openFile.

SYNTAX

`__resetFile (`*`filehandle`*`)`

PARAMETERS

filehandle	The macro variable used as file handle by the __openFile macro.

RETURN VALUE

`int 0.`

EXAMPLE

```
__resetFile (filehandle);
```

__setBreak

Sets a given breakpoint.

SYNTAX

```
__setBreak (address, segment, length, count, condition,
cond_type, access, macro)
```

PARAMETERS

address	The breakpoint location (string).
segment	The memory segment name (string).
length	The number of bytes to be covered by the breakpoint (integer).
count	The counter value (integer).
condition	The breakpoint condition (string).
cond_type	The condition type; either `"CHANGED"` or `"TRUE"` (string).
access	The memory access type (string); concatenation of any of `"R"`, `"W"`, `"F"`, `"I"`, or `"O"`.
macro	The expression to be executed after the breakpoint is accepted (string).

RETURN VALUE

`int 0.`

EXAMPLES

```
__setBreak (".demo.c\\12", "CODE", 2, 3, "d>16", "TRUE",
"RWF", "afterMacro ()");
```

__setMap

Sets a given memory mapping.

SYNTAX

`__setMap (`*`address, segment, length, type`*`)`

PARAMETERS

`address`	The start location (integer).
`segment`	The memory segment name (string).
`length`	The number of bytes to be covered by mapping (integer).
`type`	The memory mapping type; `"G"` (guarded) or `"P"` (protected) (string).

RETURN VALUE

`int 0.`

EXAMPLE

```
__setMap (0x1234, "CODE", 1000, "G");
```

__writeFile

Writes to a file.

SYNTAX

`__writeFile (`*`filehandle, value`*`)`

PARAMETERS

`filehandle`	The macro variable used as the file handle set by the `__openFile` macro.
`value`	The value to be written to the file. *`value`* is written using a format depending on what access type the file was opened with. In ASCII mode the value is written to the file as a string of hex digits corresponding to *`value`*. In binary mode the lowest byte of *`value`* is written as a binary byte.

RETURN VALUE

`int 0.`

EXAMPLE

```
__writeFile (filehandle, 123);
```

__writeMemoryByte

Writes one byte to a given memory location.

SYNTAX

```
__writeMemoryByte (value, address, segment)
```

PARAMETERS

value	The value to be written (integer).
address	The memory address (integer).
segment	The memory segment name (string).

RETURN VALUE

`int 0`.

EXAMPLE

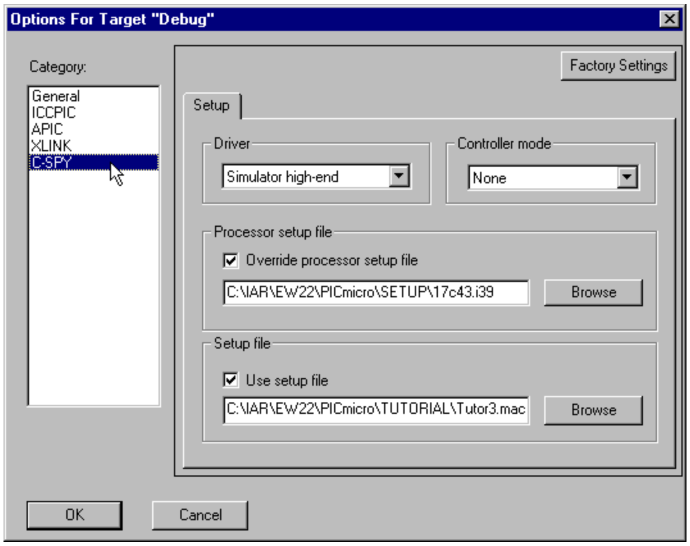
```
__writeMemoryByte (0xFF, 0x2000, "CODE");
```

C-SPY CONFIGURATION

This chapter gives information about customizing C-SPY.

SETTING C-SPY OPTIONS

To set C-SPY options in the Embedded Workbench choose **Options…** from the **Project** menu, and select **C-SPY** in the **Category** list to display the C-SPY options pages:



Driver: Simulator high-end
Controller mode: None
Processor setup file
☑ Override processor setup file
C:\IAR\EW22\PICmicro\SETUP\17c43.i39
Setup file
☑ Use setup file
C:\IAR\EW22\PICmicro\TUTORIAL\Tutor3.mac

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

Setting C-SPY options from the command line

You can specify C-SPY options when you start the IAR C-SPY Debugger from the command line with the Windows **Run…** command, or from an MS-DOS window. The following options are available from the command line:

Option	*Description*	*Section*
`-d` *`driver`*	Selects C-SPY driver.	Setup
`-m[MC|MP|EMC|EMP]`	Selects controller mode.	Setup

Option	*Description*	*Section*
`-vf`*`setupfile`*	Selects processor setup file	Setup
`-f` *`file`*	Loads macro file.	Setup

SETUP

The **Setup** options specify the C-SPY driver, setup file, and memory layout file.

DRIVER

Syntax: `-d` *`driver`*

Selects the appropriate driver for use with C-SPY, for example a simulator or an emulator. The following drivers are available:

C-SPY version	*Driver*
Simulator mid-range	`spic16.cdr`
Simulator high-end	`spic17.cdr`

Example
Debug `demo.d39` with the high-end simulator driver:

```
cwnn -d spic17 demo.d39
```

CONTROLLER MODE

Syntax: `-m[MC|MP|EMC|EMP]`

The controller mode option is only available for the high-end controllers.

To specify a mode:

Mode	*Command line*
Microcontroller mode.	`-mMC`
Microprocessor mode.	`-mMP`
Extended microcontroller mode.	`-mEMC`
Extended microprocessor mode.	`-mEMP`

Example
Debug `demo.d39` with the high-end simulator driver and extended micro controller mode:

```
cwnn -d spic17 -mEMC demo.d39
```

PROCESSOR SETUP FILE

Syntax: `-vfsetupfile`

Specifies the microcontroller as follows:

Mid-range (sPIC16 driver)

`16C16.i39` (default)	`16C62A.i39`	`16C621.i39`
`16C622.i39`	`16C63.i39`	`16C64A.i39`
`16C641.i39`	`16C642.i39`	`16C65A.i39`
`16C66.i39`	`16C661.i39`	`16C662.i39`
`16C67.i39`	`16C71.i39`	`16C710.i39`
`16C711.i39`	`16C715.i39`	`16C72.i39`
`16C73A.i39`	`16C74A.i39`	`16C76.i39`
`16C77.i39`	`16F84.i39`	`16C923.i39`
`16C924.i39`		

High-end (sPIC17 driver)

`17C42A` (default)`.i39`	`17C43.i39`	`17C44.i39`
`17C752.i39`	`17C756.i39`	

The default processor setup file is selected automatically for driver `SPIC16` or `SPIC17` when starting C-SPY. You can override this by selecting **Override default**, and then specifying an alternative file.

You must specify the processor setup file on the command line. For example to debug `demo.d39` with the high-end simulator driver and the `17c43.i39` microcontroller setup file:

```
cwnn -d spic17 -vf17c43.i39 demo.d39
```

Note, if the environment variable `QPICINFO` is not set, the full path to the processor set-up file must be given:

```
-vfc:\iar\ewnn\picmicro\setup\17c43.i39
```

For more information about the `QPICINFO` environment variable, see *Environment variable QPICINFO* below.

SETUP FILE

Syntax: `-f file`

Registers the contents of the specified macro file in the C-SPY startup sequence. If no extension is specified, the extension `.mac` is assumed.

By default no setup file is selected. You can override this by selecting **Use setupfile**, and then specifying a file.

Example
Register `watchdog.mac` at startup when debugging `watchdog.d39`:

```
cwnn -f watchdog.mac watchdog.d39
```

ENVIRONMENT VARIABLE QPICINFO

C-SPY uses an environment variable, `QPICINFO`, to locate the set-up files for the microcontrollers. It can be set using the MS-DOS `set` command. You can make this setting automatic by placing the `set` command in your `autoexec.bat` file. In Windows NT you can add the environment variable in the **Environment** page of the **System** option in the Control Panel.

Note: The path must end with a backslash.

Example:

```
set QPICINFO = c:\iar\setup\
```