MSP430 IAR Embedded Workbench™

Tutorials

for Texas Instruments'

MSP430 Family

First edition: June 2000

Part number: T430-1

WELCOME

Welcome to the MSP430 IAR Embedded Workbench™ Tutorials.

This guide exemplifies how you use the IAR Embedded Workbench™ with its integrated Windows development tools for the MSP430 Family.

Refer to the complete set of manuals for detailed information about the development tools incorporated in the IAR Embedded Workbench.

If you want to know more about IAR Systems, visit the website **www.iar.com** where your will find company information, product news, technical support, and much more.

ABOUT THIS GUIDE

This guide consists of the following parts:

- *IAR Embedded Workbench tutorial* describes a typical development cycle using the IAR Embedded Workbench, the MSP430 IAR Compiler, and the IAR XLINK Linker™. It also introduces you to the IAR C-SPY Debugger.
- *Compiler tutorials* illustrates how you might use the IAR Embedded Workbench and the IAR C-SPY Debugger to develop a series of typical programs for the MSP430 IAR Compiler, using some of the compiler's most important features.
- *Assembler tutorials* illustrates how you might use the IAR Embedded Workbench and the IAR C-SPY Debugger to develop machine-code programs, using some of the most important features of the MSP430 IAR Assembler. It also introduces you to the IAR XLIB Librarian™.
- *Advanced tutorials* illustrates how you might use both code written for the MSP430 IAR Compiler and code written for the MSP430 IAR Assembler in the same project. It also explores the functionality of the IAR C-SPY Debugger.

ASSUMPTIONS AND CONVENTIONS

ASSUMPTIONS

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

- The C programming language and the IAR MSP430 assembly language.
- The architecture and instruction set of the MSP430 Family.
- The procedures for using menus, windows, and dialog boxes in a Windows environment.

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

CONVENTIONS

This user 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.
`{a | b | c}`	Alternatives in a command.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
reference	A cross-reference to another part of this guide, or to another guide.
🛠	Identifies instructions specific to the IAR Embedded Workbench versions of the IAR development tools.
🖥	Identifies instructions specific to the command line versions of the IAR development tools.

Contents

TUTORIALS

The MSP430 IAR Embedded Workbench™ Tutorials contains the following chapters:

- *IAR Embedded Workbench tutorial*
- *Compiler tutorials*
- *Assembler tutorials*
- *Advanced tutorials*.

You should install the IAR development tools before running these tutorials.

IAR EMBEDDED WORKBENCH TUTORIAL

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

Tutorial 1 describes a typical development cycle:

- We first create a project, add source files to it, and specify target options.
- We then compile the program, examine the list file, and link the program.
- Finally we run the program in the IAR C-SPY Debugger.

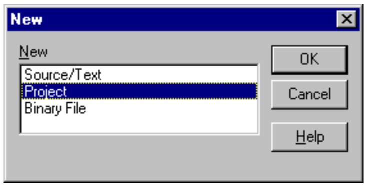
  Alternatively, you may follow this tutorial by examining the list files created. They show which areas of memory to monitor.

TUTORIAL 1

We recommend that you create a specific directory where you can store all your project files, for example the `430\projects` directory.

CREATING A NEW PROJECT

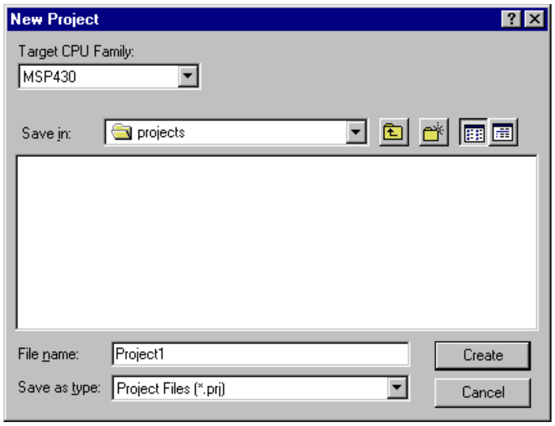
The first step is to create a new project for the tutorial programs. Start the IAR Embedded Workbench, and select **New...** from the **File** menu to display the following dialog box:



The **Help** button provides access to information about the IAR Embedded Workbench. You can at any time press the F1 key to access the online help.

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

Enter `Project1` in the **File name** box, and set the **Target CPU Family** to **MSP430**. Specify where you want to place your project files, for example in a `projects` directory:



Target CPU Family: MSP430

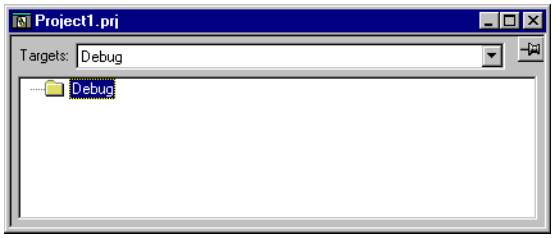
Save in: projects

File name: Project1

Save as type: Project Files (*.prj)

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

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

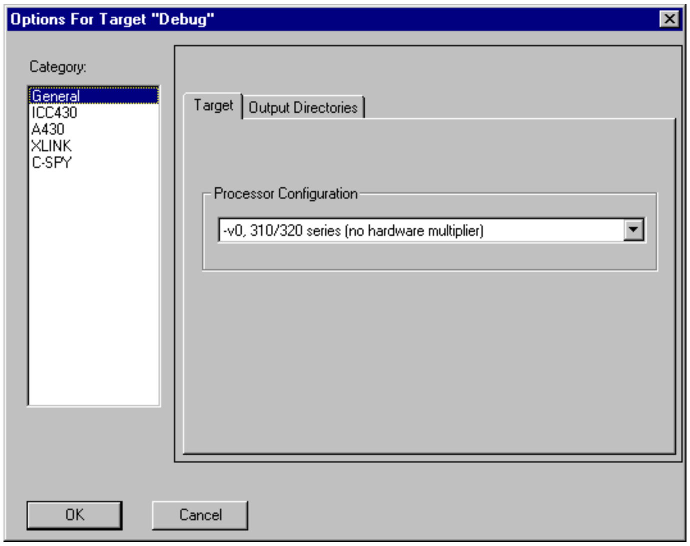


Targets: Debug

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

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

Make sure that the **Processor Configuration** option is set to `-v0`:


Options For Target "Debug"

Category: General

Processor Configuration: -v0, 310/320 series (no hardware multiplier)

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

THE SOURCE FILES

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

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

The tutor.c program

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

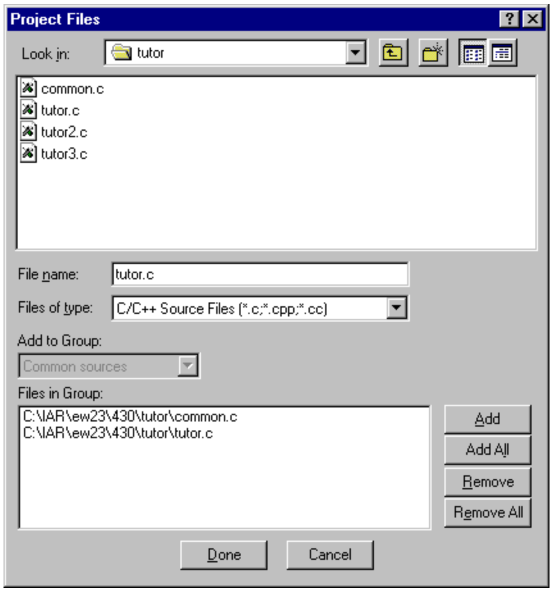
```
#include "tutor.h"


   /* Global call counter */
int call_count;

   /* Get and print next Fibonacci number. */
void do_foreground_process(void)
{
  unsigned int fib;
  next_counter();
  fib = get_fib( call_count );
  put_fib( fib );
}

   /* Main program. Prints the Fibonacci numbers. */
void main(void)
 {
 call_count = 0;
 init_fib();
 while (  call_count < MAX_FIB )
   do_foreground_process();
 }
```

ADDING FILES TO THE PROJECT

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

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

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



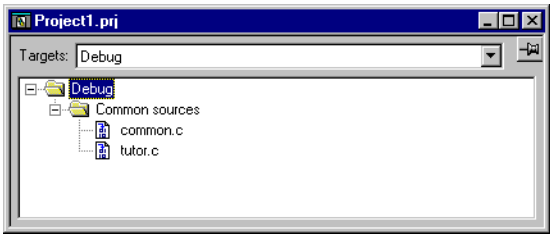
Look in: tutor

common.c
tutor.c
tutor2.c
tutor3.c

File name: tutor.c
Files of type: C/C++ Source Files (*.c;*.cpp;*.cc)
Add to Group: Common sources
Files in Group:
C:\IAR\ew23\430\tutor\common.c
C:\IAR\ew23\430\tutor\tutor.c

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

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

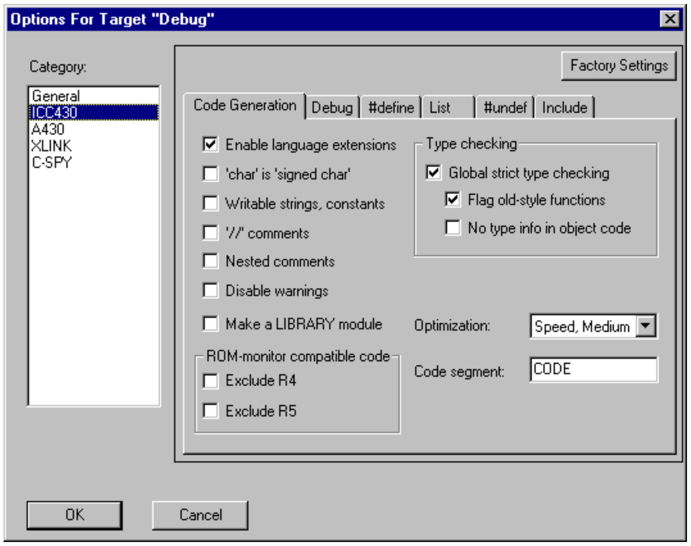


The **Common Sources** group was created by the IAR Embedded Workbench when you created the project.

SETTING COMPILER OPTIONS

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

Select the **Debug** folder icon in the Project window, choose **Options...** from the **Project** menu, and select **ICC430** in the **Category** list to display the IAR Compiler options pages:



- [✓] Enable language extensions
- [ ] 'char' is 'signed char'
- [ ] Writable strings, constants
- [ ] '//' comments
- [ ] Nested comments
- [ ] Disable warnings
- [ ] Make a LIBRARY module

ROM-monitor compatible code
- [ ] Exclude R4
- [ ] Exclude R5

Type checking
- [✓] Global strict type checking
- [✓] Flag old-style functions
- [ ] No type info in object code

Optimization: Speed, Medium

Code segment: CODE

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

Page	*Options*
Code Generation	Enable language extensions Type checking: Global strict type checking Flag old-style functions Optimization: Speed, Medium
Debug	Generate debug information
List	List file Insert mnemonics

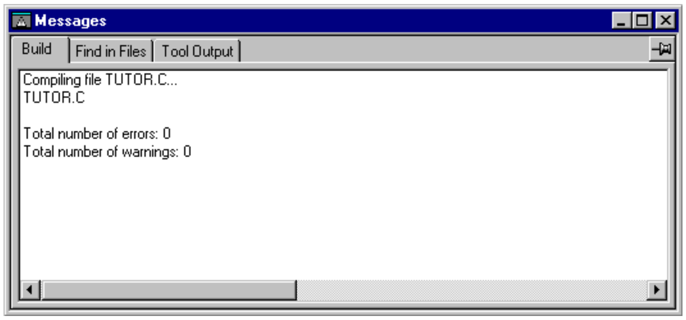
When you have made these changes, choose **OK** to set the options you have specified. The remaining options should remain at their default settings.

COMPILING THE TUTOR.C AND COMMON.C FILES

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

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

The progress will be displayed in the Messages window.



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

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

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

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

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

- In the `list` directory, your list files from the `Debug` target will be placed. The list files have the extension `lst` and will be located here.
- In the `obj` directory, the object files from the compiler and the assembler will be placed. These files have the extension `r43` and will be used as input to the IAR XLINK Linker.
- In the `exe` directory, you will find the executable files. These files have the extension `d43` and will be used as input to the IAR C-SPY Debugger.

VIEWING THE LIST FILE

Open the list file `tutor.lst` by selecting **Open…** from the **File** menu, and selecting `tutor.lst` from the `debug\list` directory. Examine the list file, which contains the following information:

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

```
##########################################################################
#                                                                        #
# IAR MSP430 C-Compiler VX.xxx/WIN                                       #
#                                                                        #
#    Compile time  =  dd/Mmm/yyyy  hh:mm:ss                              #
#    Target option =  SP430x31x                                          #
#    Memory model  =  small                                              #
#    Source file   =  c:\iar\ew23\430\tutor\tutor.c                      #
#    List file     =  c:\iar\ew23\430\projects\debug\list\tutor.lst      #
#    Object file   =  c:\iar\ew23\430\projects\debug\obj\tutor.r43       #
#    Command line  =  -OC:\IAR\ew23\430\projects\Debug\Obj\ -e -gA -s6   #
#                     -RCODE -r0 -LC:\IAR\ew23\430\projects\Debug\List\  #
#                     -q -t8 -IC:\IAR\ew23\430\inc\                      #
#                     C:\IAR\ew23\430\tutor\tutor.c                      #
#                                                                        #
#                      Copyright yyyy IAR Systems. All rights reserved.  #
##########################################################################
```

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

```
     25           void main(void)
     26            {
     27            call_count = 0;
  \   0012  82430000          MOV     #0,&call_count
     28            init_fib();
  \   0016  B0120000          CALL    #init_fib
  \   001A            ?0001:
     29            while (  call_count < MAX_FIB )
  \   001A  B2900A00          CMP     #10,&call_count
  \   001E  0000
  \   0020  0334              JGE     (?0000)
     30              do_foreground_process();
  \   0022  B0120000          CALL    #do_foreground_process
     31            }
  \   0026  F93F              JMP     (?0001)
  \   0028            ?0000:
  \   0028  3041              RET
     32
  \   0000                    RSEG    UDATA0
```

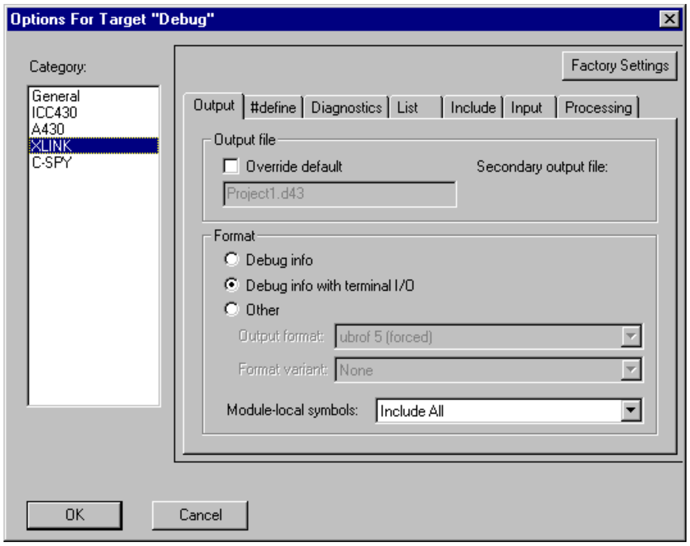
The *end* of the list file shows the amount of code memory required, and contains information about error and warning messages that may have been generated:

```
Errors: none
Warnings: none
Code size: 42
Constant size: 0
Static variable size: 2
```

LINKING THE TUTOR.C PROGRAM

First set up the options for the IAR XLINK Linker™:

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



Options For Target "Debug"

Category: General, ICC430, A430, XLINK, C-SPY

Output file: Override default [ ] Project1.d43 Secondary output file:

Format: ( ) Debug info (●) Debug info with terminal I/O ( ) Other

Output format: ubrof 5 (forced)

Format variant: None

Module-local symbols: Include All

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

Page	*Options*
Output	Debug info with terminal I/O
List	Generate linker listing Segment map Module map
Include	Override default: `lnk430.xcl` (the linker command file)

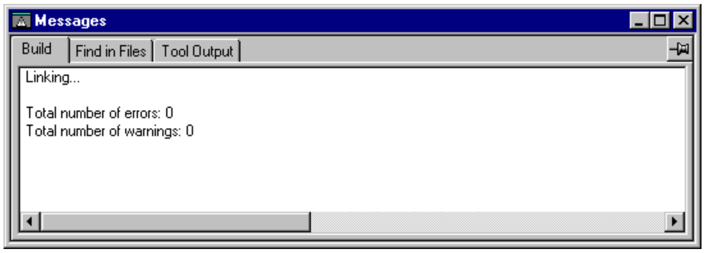
If you want to examine the linker command file, use a suitable text editor, such as the IAR Embedded Workbench editor, or print a copy of the file.

The definitions in the linker command file are not permanent; they can be altered later on to suit your project if the original choice proves to be incorrect, or less than optimal. For more information about linker command files, see the *Configuration* chapter in the *MSP430 C Compiler Programming Guide.*

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

The chapter *XLINK options reference* in the *MSP430 Assembler, Linker, and Librarian Programming Guide* contains information about the XLINK options available in the IAR Embedded Workbench, to be used in the linker command file and on the command line.

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



```
Linking...

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

The result of the linking is a code file `project1.d43` with debug information and a map file `project1.map`.

Viewing the map file

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

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

- The `SEGMENTS IN ADDRESS ORDER` section lists all the segments that constitute the program.

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

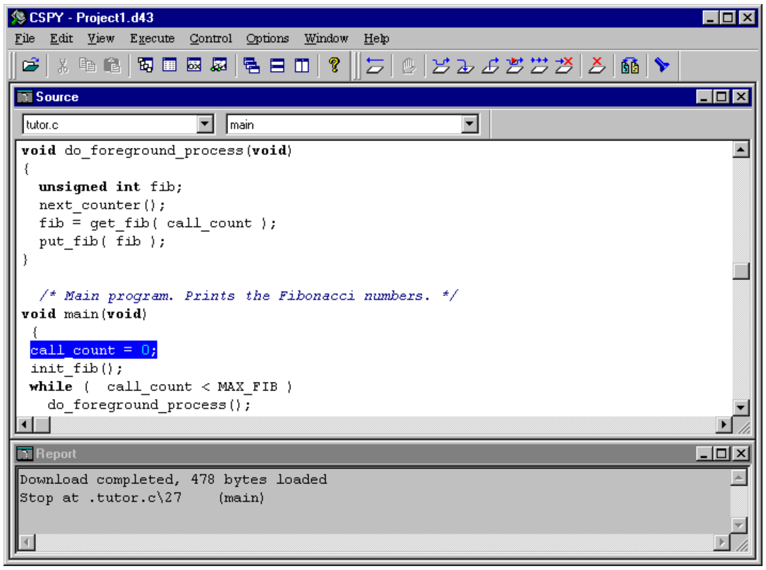
Notice that the text file will contain the command line equivalents to the options that you have specified in the IAR Embedded Workbench. The command line options are described in the *MSP430 C Compiler Programming Guide* and *MSP430 Assembler, Linker, and Librarian Programming Guide*, respectively.

RUNNING THE PROGRAM

Now we will run the `project1.d43` program using the IAR C-SPY Debugger to watch variables, set a breakpoint, and print the program output in the Terminal I/O window.

Choose **Debugger** from the **Project** menu in the IAR Embedded Workbench. Alternatively, click the C-SPY button in the toolbar.

A C-SPY window will be opened for this file:



```
void do_foreground_process(void)
{
  unsigned int fib;
  next_counter();
  fib = get_fib( call_count );
  put_fib( fib );
}

  /* Main program. Prints the Fibonacci numbers. */
void main(void)
 {
 call_count = 0;
 init_fib();
 while (  call_count < MAX_FIB )
   do_foreground_process();
```

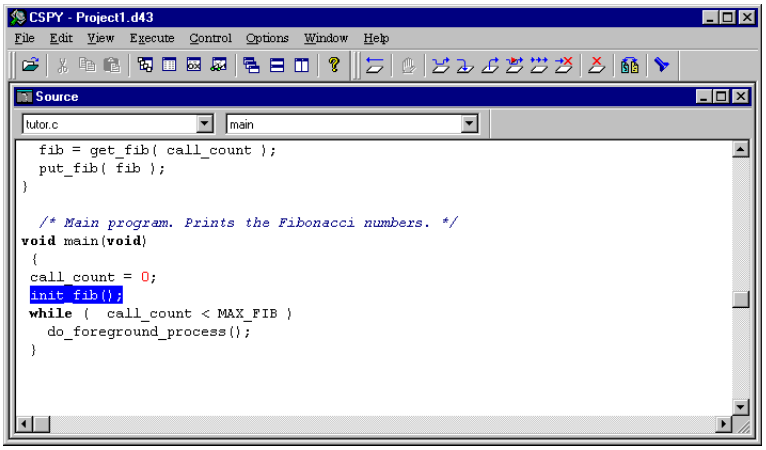
```
Download completed, 478 bytes loaded
Stop at .tutor.c\27     (main)
```

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

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

Execute one step by choosing **Step** from the **Execute** menu. Alternatively, click the **Step** button in the toolbar. At source level **Step** executes one source statement at a time.

The current position should be the call to the `init_fib` function:

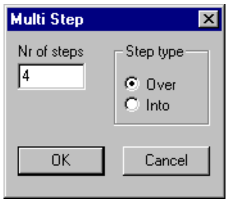


```
  fib = get_fib( call_count );
  put_fib( fib );
}

  /* Main program. Prints the Fibonacci numbers. */
void main(void)
 {
 call_count = 0;
 init_fib();
 while (  call_count < MAX_FIB )
   do_foreground_process();
 }
```

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

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

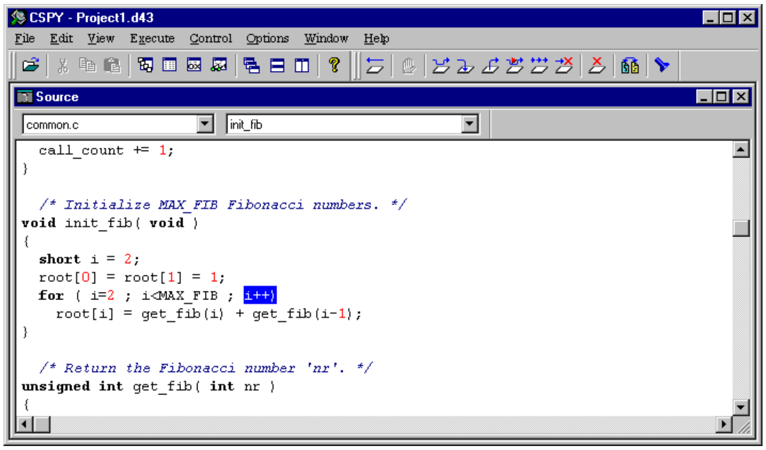
Step four more times. Choose **Multi Step…** from the **Execute** menu, and enter `4`.



Nr of steps: 4

Step type: [✓ Over] Into

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++`:



```
  call_count += 1;
}

  /* 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);
}

  /* Return the Fibonacci number 'nr'. */
unsigned int get_fib( int nr )
{
```

WATCHING VARIABLES

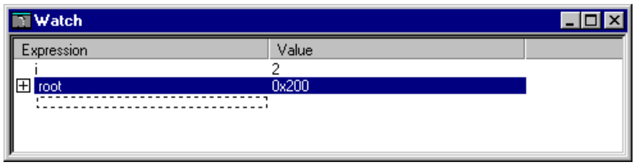
C-SPY allows you to set watchpoints on C variables or expressions, to allow you to keep track of their values as you execute the program. You can watch variables in a number of ways; for example, you can watch a variable by pointing at it in the Source window with the mouse pointer, or by opening the Locals window. Alternatively, you can open the QuichWatch window from the pop-up menu that appears when you press the right mouse button in the Source window.

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

Set a watchpoint on the variable `i` using the following procedure: Select the dotted rectangle, then click and briefly hold the left mouse button. In the entry field that appears when you release the button, type: `i` and press the Enter key.

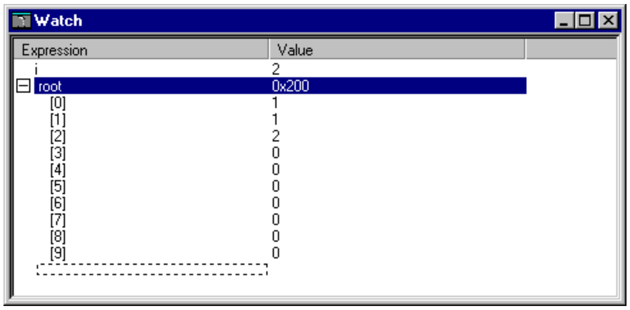
You can also drag and drop a variable into the Watch window. Select the `root` array in the `init_fib` function in the Source window. When `root` is marked, drag and drop it in the Watch window.

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



Expression	Value
i	2
⊞ root	0x200

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



Expression	Value
i	2
⊟ root	0x200
[0]	1
[1]	1
[2]	2
[3]	0
[4]	0
[5]	0
[6]	0
[7]	0
[8]	0
[9]	0

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

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

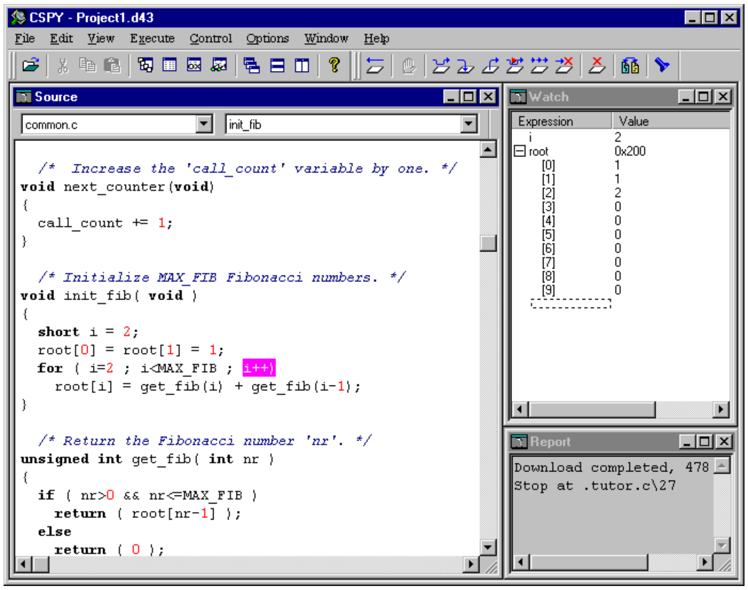
SETTING BREAKPOINTS

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

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

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

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



```
  /*  Increase the 'call_count' variable by one. */
void next_counter(void)
{
  call_count += 1;
}

  /* 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);
}

  /* Return the Fibonacci number 'nr'. */
unsigned int get_fib( int nr )
{
  if ( nr>0 && nr<=MAX_FIB )
    return ( root[nr-1] );
  else
    return ( 0 );
```

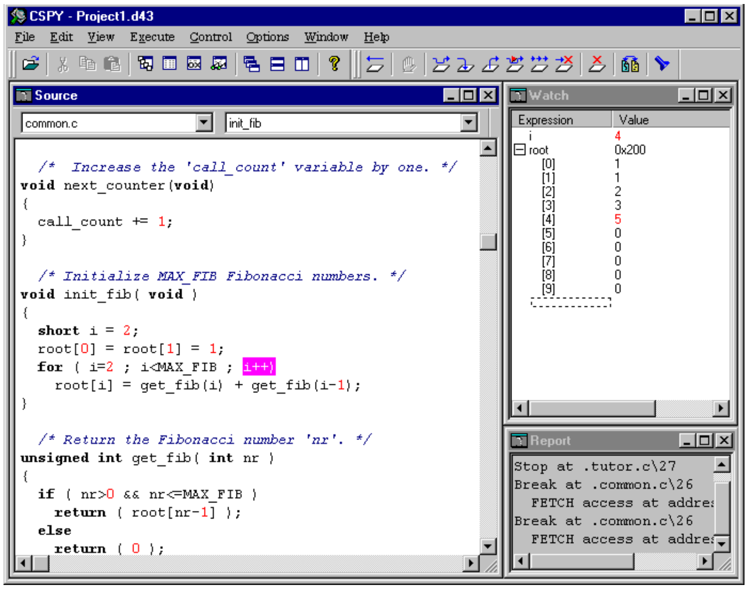
Expression	Value
i	2
⊟ root	0x200
[0]	1
[1]	1
[2]	2
[3]	0
[4]	0
[5]	0
[6]	0
[7]	0
[8]	0
[9]	0

```
Download completed, 478
Stop at .tutor.c\27
```

EXECUTING UP TO A BREAKPOINT

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

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



```
  /*  Increase the 'call_count' variable by one. */
void next_counter(void)
{
  call_count += 1;
}

  /* 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);
}

  /* Return the Fibonacci number 'nr'. */
unsigned int get_fib( int nr )
{
  if ( nr>0 && nr<=MAX_FIB )
    return ( root[nr-1] );
  else
    return ( 0 );
```

Expression	Value
i	4
⊟ root	0x200
[0]	1
[1]	1
[2]	2
[3]	3
[4]	5
[5]	0
[6]	0
[7]	0
[8]	0
[9]	0

```
Stop at .tutor.c\27
Break at .common.c\26
  FETCH access at addre
Break at .common.c\26
  FETCH access at addre
```

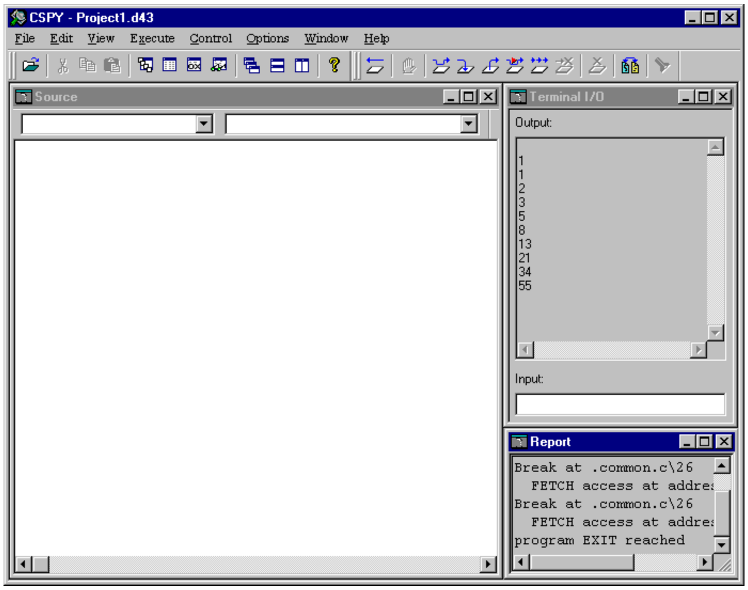
Remove the breakpoint by selecting **Edit breakpoint** from the **Control** menu. Alternatively, click the right mouse button to display a pop-up menu. Select the breakpoint in the **Breakpoints** list and press **Clear**. Then close the **Breakpoints** dialog box.

CONTINUING EXECUTION

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

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

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



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

```
Break at .common.c\26
  FETCH access at addre
Break at .common.c\26
  FETCH access at addre
program EXIT reached
```

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

EXITING FROM C-SPY

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

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

For detailed information about the features of C-SPY, see the *Command reference* chapter in *MSP430 C-SPY User Guide.*

COMPILER TUTORIALS

This chapter introduces you to some of the IAR Compiler's MSP430-specific features:

- Tutorial 2 demonstrates how to utilize MSP430 peripherals with the IAR Compiler features. The `#pragma` directive allows us to use the MSP430-specific language extensions. Our program will be extended to handle polled I/O. Finally, we run the program in C-SPY and create virtual registers.
- In Tutorial 3 we modify the tutorial project by adding an interrupt handler. The system is extended to handle the real-time interrupt using the MSP430 IAR Compiler intrinsics and keywords. Finally, we run the program using the C-SPY interrupt system in conjunction with complex breakpoints and macros.

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

TUTORIAL 2

This IAR Compiler tutorial will demonstrate how to simulate the MSP430 Universal Synchronous/Asynchronous Receive/Transmit (USART) Communication Interface using the IAR Compiler features. The `#pragma` directive allows us to use the MSP430-specific language extensions. Our program will be extended to handle polled I/O. We will run the program in C-SPY and create virtual registers.

THE TUTOR2.C SERIAL PROGRAM

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

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

   /* Global call counter */
int call_count;

   /* Get and print next Fibonacci number. */
void do_foreground_process(void)
```

```
{
  unsigned int fib;
  if ( receive_ok() )/* wait for receive data */
  {
    next_counter();
    fib = get_fib( call_count );
    put_fib( fib );
  }
  else
    putchar( '.' );
}

   /* Main program. Prints the Fibonacci numbers. */
void main(void)
{
  init_cntr();
  init_fib();
  while ( call_count < MAX_FIB)
    do_foreground_process();
}
```

COMPILING AND LINKING THE TUTOR2.C SERIAL PROGRAM

Modify the `Project1` project by replacing `tutor.c` with `tutor2.c`:

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

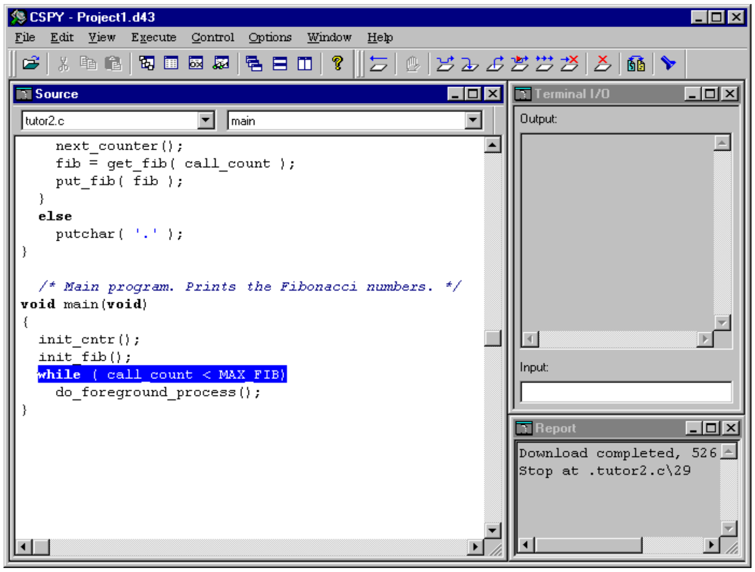
Click on the **Done** button to close the **Project Files** dialog box.

Then select **Options...** from the **Project** menu. In the **General** category, select target processor configuration `-v1`. In the **ICC430** category, make sure that language extensions are enabled and that debug information will be generated. In the **XLINK** category, click the **Include** tab and select the linker command file `lnk430m.xcl`.

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

RUNNING THE TUTOR2.C SERIAL PROGRAM

Start the IAR C-SPY Debugger and run the modified `project1` project. Step until you reach the `while` loop, where the program waits for input. Open the Terminal I/O window, where the `tutor2` result will be printed.



```
    next_counter();
    fib = get_fib( call_count );
    put_fib( fib );
  }
  else
    putchar( '.' );
}

  /* Main program. Prints the Fibonacci numbers. */
void main(void)
{
  init_cntr();
  init_fib();
  while ( call_count < MAX_FIB)
    do_foreground_process();
}
```

```
Download completed, 526
Stop at .tutor2.c\29
```

DEFINING VIRTUAL REGISTERS

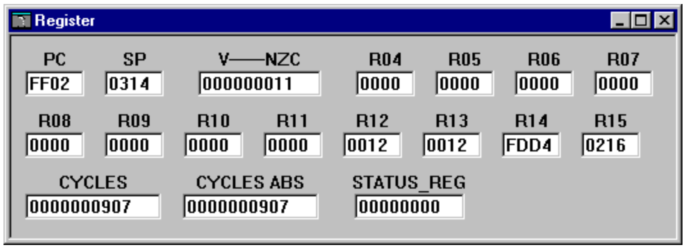
To simulate different values for the serial interface, we will make a new virtual register called `STATUS_REG`**.**

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

Enter the following information in the dialog box:

Input field	*Input*	*Description*
Name	`STATUS_REG`	Virtual register name
Size	`1`	One byte
Base	`2`	Binary values
Address	`03`	Memory location (in hex)
Segment	`Memory`	Segment name

Then choose **OK** in both dialog boxes. Open the Register window from the **Window** menu, or select the **Register Window** button in the toolbar. `STATUS_REG` should now be included in the displayed list of registers. The current value of each bit in the serial interface register is shown:



Register

PC	SP	V——NZC	R04	R05	R06	R07
FF02	0314	000000011	0000	0000	0000	0000

R08	R09	R10	R11	R12	R13	R14	R15
0000	0000	0000	0000	0012	0012	FDD4	0216

CYCLES	CYCLES ABS	STATUS_REG
0000000907	0000000907	00000000

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

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

TUTORIAL 3

In this tutorial we change from polled I/O to interrupt-driven I/O.

We will define an interrupt function that handles the received interrupt, and we will use the C-SPY macro system to simulate the interrupt that feeds the Fibonacci numbers to the buffer.

Note: In the C library, all I/O functions are buffered; this may cause input or output to be lost when an interrupt occurs. To prevent this problem, you can guard your call to the I/O functions by using the extended keyword `monitor`, which disables all interrupts during execution of guarded functions. To watch the consequences of unguarded functions, you may remove all occurrences of the `monitor` keyword in the `common.c` and `tutor3.c` files and run the program with different intervals between the interrupts.

THE TUTOR3.C INTERRUPT PROGRAM

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

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

   /* Global call counter */
int call_count;

   /* Get and print next Fibonacci number. */
/* The receive buffer has vector address 0xFFEE, ie
offset 0xE in INTVECT */
interrupt [0x0E] void RXinterrupt(void)
{
  unsigned int fib;

  next_counter();
  fib = RXBUF;
  put_fib( fib );
}

monitor void do_foreground_process(void)
{
  putchar ( ’.’ );
}
```

```
   /* Main program. Prints the Fibonacci numbers. */
void main(void)
{
  init_cntr();
  init_fib();

  /* Enable interrupts */
  _EINT();

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

The addresses for the SFR registers are defined in the header file `io330.h`.

The start address of the interrupt handler must be located at the correct offset in the interrupt table. The receive buffer interrupt has the vector address `0xFFEE`, i.e. offset `0x0E`.

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

```
interrupt [0x0E] void RXinterrupt(void)
```

Extended keywords are described in the *MSP430 C Compiler Programming Guide*.

The interrupt handler will read the latest Fibonacci value from the receive buffer. It will then print the value using the `put_fib` function.

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

THE C-SPY TUTOR3.MAC MACRO FILE

In the C-SPY macro file called `tutor3.mac`, we use system and user-defined macros. Notice that this example is not intended as an exact simulation; the purpose is to illustrate a situation where C-SPY macros can be useful. For detailed information about macros, see the chapter *System macros* in *MSP430 C-SPY User Guide*.

Initializing the system

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

```
execUserSetup()
{
  message "execUserSetup() called\n";
  Tutor3Setup ();
}
```

First we print a message in the C-SPY Message window so that we know that this macro has been executed.

Then we call the `Tutor3Setup()` user-defined macro:

```
Tutor3Setup()
{
   message "Tutor3Setup() called\n";
  // Open the tutor3.txt as input file
  // Note: The path should be modified
   if(__openFile(_FileHandle,
"C:\\iar\\ew23\\430\\tutor\\tutor3.txt", "r"))
     __printLastMacroError();

  // Set up an interrupt
   _InterruptID =
   __orderInterrupt("0xFFEE", 3000, 2000, 0, 0, 100);

  // Set breakpoint that triggers the Access()
  // Recieve data register (RXBUF)
 __setBreak("0x76", "PGM", 1, 1, "", "TRUE",
                 "I", "Access()");
 }
```

The Fibonacci numbers are read from a supplied file, `tutor3.txt`. We open the file for reading in ASCII mode, which means that the data in the file is in hexadecimal notation without the prefix `0x`. Notice that you may need to modify the path of the `tutor3.txt` file.

Generating interrupts

The `__orderInterrupt` system macro orders C-SPY to generate interrupts. The following parameters are used:

`0xFFEE`	Specifies which interrupt vector to use (a string).
`3000`	Specifies the activation moment for the interrupt. The interrupt is activated when the cycle counter has passed this value.
`2000`	Specifies the repeat interval for the interrupt, measured in clock cycles.
`0`	Time variance, not used here.
`0`	Latency, not used here.
`100`	Specifies probability. Here it denotes 100 %. We want the interrupt to occur at the given frequency. Another percentage could be used to simulate a more randomized interrupt behavior.

During execution, C-SPY will wait until the cycle counter has passed the activation time, in this case 3000 cycles. Then it will, with 100 % certainty, generate an interrupt approximately every 2000 cycles.

Using breakpoints to simulate incoming values

We must also simulate the incoming values to the SCI. This is done by setting a breakpoint at the receive buffer address and connecting a user-defined macro to it. Here we use the `__setBreak` system macro.

The following parameters are used:

`0x76`	Receive buffer address.
`“PGM”`	The memory segment where this address is found. `PGM` is the valid MSP430 segment.
`1`	Length.
`1`	Count.
`““`	Denotes unconditional breakpoint.
`“TRUE”`	Condition type.

`“I”`	The memory access type. Here we use “Read Immediate” which means that C-SPY breaks *before* reading the value at the specified address. This gives us the opportunity to put the correct Fibonacci value in the receive buffer before C-SPY reads the value.
`“Access ()”`	The macro connected to the breakpoint.

During execution, when C-SPY detects a read from the receive buffer address, it will temporarily halt the simulation and run the `Access` macro. Since this macro ends with a `resume` statement, C-SPY will then resume the simulation and start by reading the receive buffer value.

The `Access` macro is executed whenever C-SPY tries to read the receive buffer value, as defined in the `__setBreak` macro:

```
Access()
{
 message "Access() called\n";
 // Read a value from a file and write to sfr
 //  Receive data register (RXBUF)
 __writeMemoryByte(__readFile(_FileHandle),0x76,"PGM");
resume;
}
```

First we print a message which is useful for debugging purposes.

Next we will read a value from the file and write it to the receive buffer register in the `PGM` area.

Finally, the `resume` statement causes C-SPY to continue the simulation process.

Resetting the system

The macro `execUserReset()` is automatically executed during C-SPY reset. At reset, we want to rewind the input file:

```
execUserReset()
{
  message "execUserReset() called\n";
  __rewindFile(_FileHandle);
}
```

Exiting the system

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

```
execUserExit()
{
  message "execUserExit() called\n";
  Tutor3Shutdown();
}
```

The`Tutor3Shutdown()` user-defined macro is called to cancel the used interrupt and to clear the receive-buffer breakpoint:

```
Tutor3Shutdown()
{
 message "Tutor3Shutdown() called\n";
 __cancelInterrupt(_InterruptID);
 __clearBreak("0x76","PGM","I");
 __closeFile(_FileHandle);
}
```

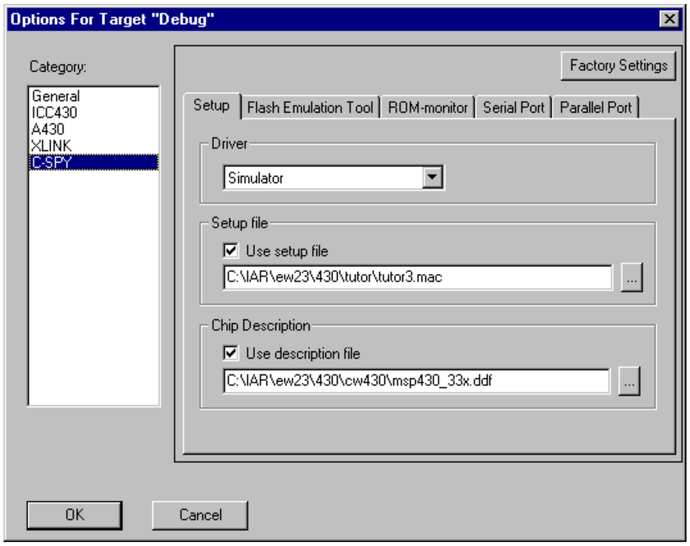
Finally we close the input file.

COMPILING AND LINKING THE TUTOR3.C PROGRAM

Modify `Project1` by removing `tutor2.c` from the project and adding `tutor3.c` to it. Compile and link the program by choosing **Make** from the **Project** menu. Alternatively, select the **Make** button from the toolbar. The **Make** command compiles and links those files that have been modified.

RUNNING THE TUTOR3.C INTERRUPT PROGRAM

To run the `tutor3.c` program, we first specify the macro to be used. The macro file, `tutor3.mac`, is specified in the **C-SPY** options page in the IAR Embedded Workbench:



Driver: Simulator

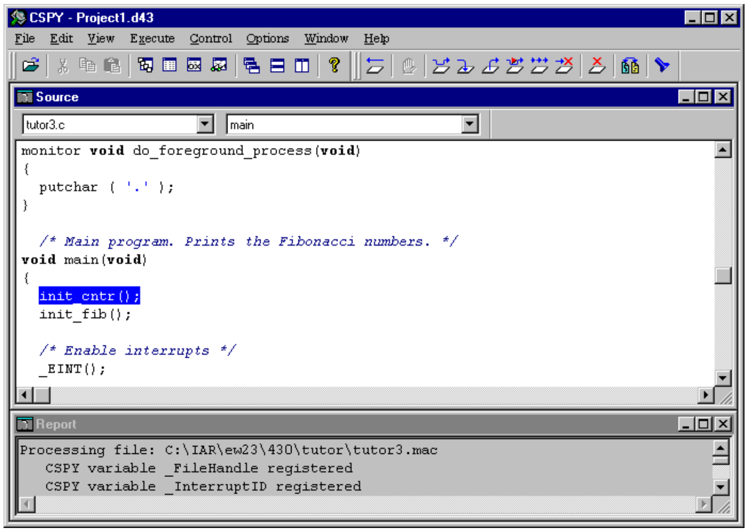
Setup file: ☑ Use setup file: C:\IAR\ew23\430\tutor\tutor3.mac

Chip Description: ☑ Use description file: C:\IAR\ew23\430\cw430\msp430_33x.ddf

If you use the IAR C-SPY Debugger without using the IAR Embedded Workbench, the macro file can be specified via the command line option `-f`.

Note: Macro files can also be loaded via the **Options** menu in the IAR C-SPY Debugger. Due to its contents, the `tutor3.mac` file cannot, however, be used in this manner because the `execUserSetup` macro will not be activated until you load the `project1.d43` file.

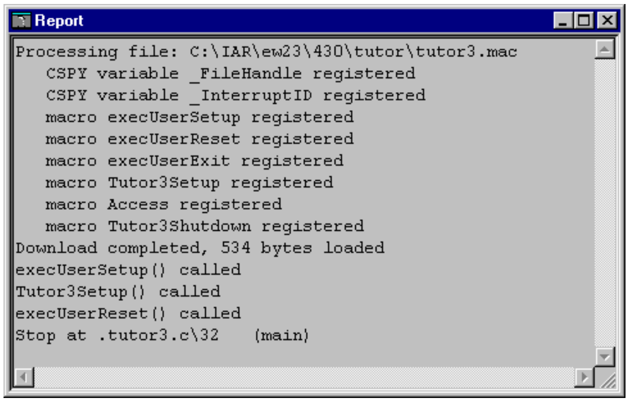
Start the IAR C-SPY Debugger by selecting **Debugger** from the **Project** menu or click the **Debugger** icon in the toolbar. The C-SPY Source window will be displayed:



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

  /* Main program. Prints the Fibonacci numbers. */
void main(void)
{
  init_cntr();
  init_fib();

  /* Enable interrupts */
  _EINT();
```

```
Processing file: C:\IAR\ew23\430\tutor\tutor3.mac
   CSPY variable _FileHandle registered
   CSPY variable _InterruptID registered
```

The Report window will display the registered macros:



```
Processing file: C:\IAR\ew23\430\tutor\tutor3.mac
   CSPY variable _FileHandle registered
   CSPY variable _InterruptID registered
   macro execUserSetup registered
   macro execUserReset registered
   macro execUserExit registered
   macro Tutor3Setup registered
   macro Access registered
   macro Tutor3Shutdown registered
Download completed, 534 bytes loaded
execUserSetup() called
Tutor3Setup() called
execUserReset() called
Stop at .tutor3.c\32    (main)
```

If warning or error messages should also appear in the Report window, make sure that the breakpoint has been set and that the interrupt has been registered. If not, the reason is probably that an incorrect path is specified in the `tutor3.mac` macro file.

Now you have a breakpoint in the interrupt function and an interrupt that will be activated every 2000 cycles. To inspect the details of the breakpoint, open the **Breakpoints** dialog box by selecting **Edit Breakpoints...** from the **Control** menu. To inspect the details of the interrupt, open the **Interrupt** dialog box by selecting **Interrupt...** from the **Control** menu.

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

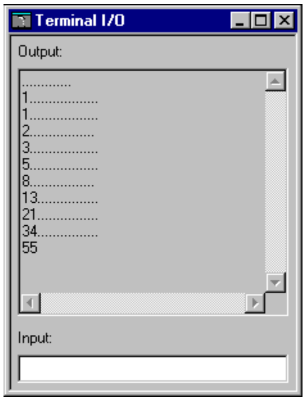
Place the cursor on the `put_fib()` statement in the `RXinterrupt` function. Set a breakpoint by selecting **Toggle Breakpoint** from the **Control** menu, or click the **Toggle Breakpoint** button in the toolbar. Alternatively, use the pop-up menu.

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

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

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

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



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

ASSEMBLER TUTORIALS

These tutorials illustrate how you might use the IAR Embedded Workbench™ to develop a series of simple machine-code programs for the MSP430 Family, and demonstrate some of the IAR Assembler's most important features:

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

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

TUTORIAL 4

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

CREATING A NEW PROJECT

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

Set up the target options in the **General** category. Make sure that the **Processor Configuration** is set to `-v0`.

The procedure is described in *Creating a new project*, page 3.

THE FIRST.S43 PROGRAM

The first assembler tutorial program is a simple count loop which counts up the registers R4 and R5 in binary-coded decimal. A copy of the program `first.s43` is provided with the product.

```
        NAME         first
        ORG          0FFFEh
        DW           main

; counts up R4,R5 in binary coded decimal

        ORG          0200h
```

```
main    CLR         R4
        CLR         R5
loop    CLRC
        DADD.B      #1,R4
        CMP         #10h,R4
        JNE         loop
        CLR         R4
        CLRC
        DADD.B      #1,R5
        JNE         loop
done    JMP         done

        END         main
```

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

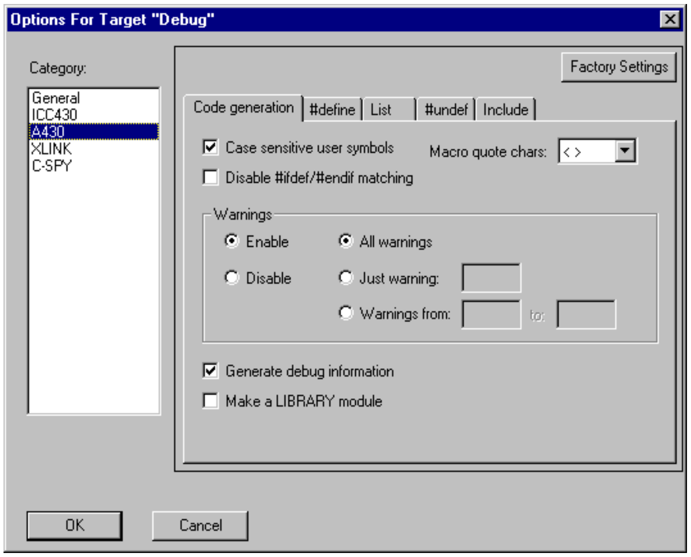
Add the program to the `Project2` project. Choose **Files…** from the **Project** menu to display the **Project Files** dialog box. Locate the file `first.s43` in the `A430` subdirectory and choose **Add** to add it to the **Common Sources** group.

You now have a source file which is ready to assemble.

ASSEMBLING THE PROGRAM

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

Select the **Debug** folder icon in the Project window, choose **Options…** from the **Project** menu, and select **A430** in the **Category** list to display the assembler options pages.



[✓] Case sensitive user symbols
Macro quote chars: < >
[ ] Disable #ifdef/#endif matching

Warnings
(●) Enable (●) All warnings
( ) Disable ( ) Just warning:
( ) Warnings from: to:

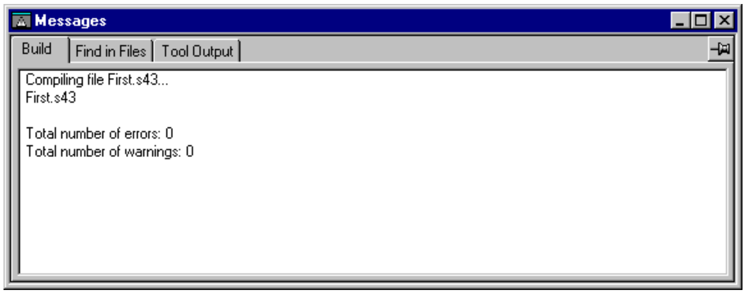
[✓] Generate debug information
[ ] Make a LIBRARY module

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

Page	*Option*
Code generation	Case sensitive user symbols Enable all warnings Generate debug information
List	List file

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

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



```
Compiling file First.s43...
First.s43

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

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

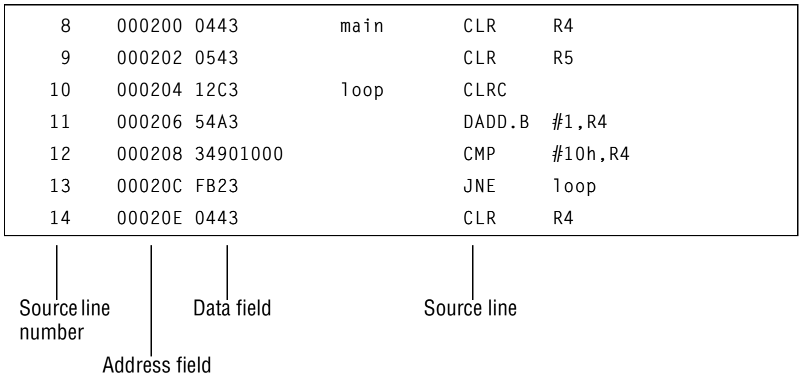
VIEWING THE FIRST.LST LIST FILE

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

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

  Note: The CRC number depends on the date of assembly, and may vary.

The format of the listing is as follows:



```
  8    000200 0443        main      CLR     R4
  9    000202 0543                  CLR     R5
 10    000204 12C3        loop      CLRC
 11    000206 54A3                  DADD.B  #1,R4
 12    000208 34901000              CMP     #10h,R4
 13    00020C FB23                  JNE     loop
 14    00020E 0443                  CLR     R4
```

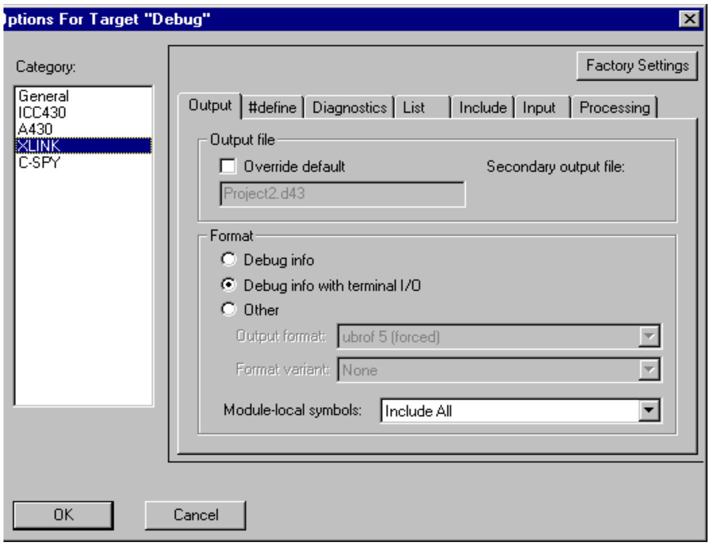
If you make any errors when writing a program, these will be displayed on the screen during the assembly and will be listed in the list file. If this happens, return to the editor by double-clicking on the error message. Check carefully through the source code to locate and correct all the mistakes,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

Assuming that the source assembled successfully, the file `first.r43`, will also be created, containing the linkable object code.

LINKING THE PROGRAM

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

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



Output file
☐ Override default
Project2.d43
Secondary output file:

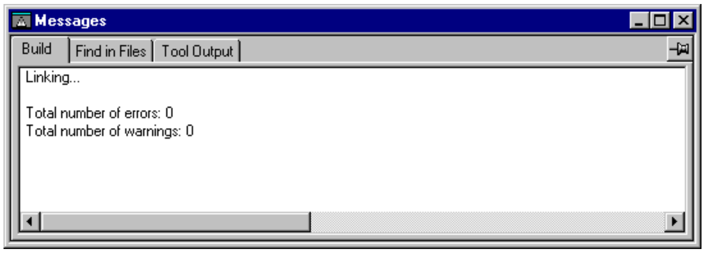
Format
○ Debug info
● Debug info with terminal I/O
○ Other
Output format: ubrof 5 (forced)
Format variant: None
Module-local symbols: Include All

Specify the following XLINK options:

Page	*Option*
Output	Debug info with terminal I/O Module-local symbols: Include all
Include	In the **XCL file name** area, select **Override default** and specify the `first.xcl` linker command file. This is a basic linker command file, designed for assembler-only projects.

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

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



```
Linking...

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

The code will be placed in a file `project2.d43.`

RUNNING THE PROGRAM

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

The following warning message will be displayed in the Report window:

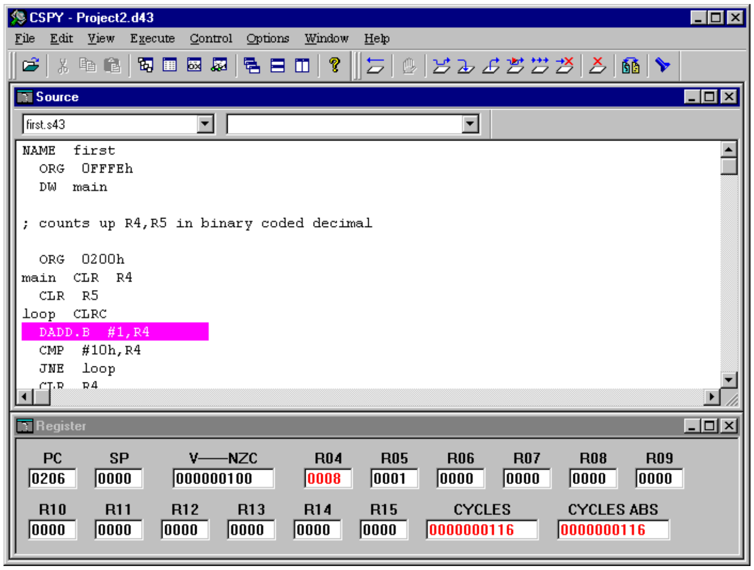
```
Warning [12]: Exit label missing
```

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

In this example, set a breakpoint on the `DADD.B #1,R4` instruction within the loop.

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

Then choose **Go** from the **Execute** menu, or click the **Go** button in the debug bar. When you repeatedly select **Go**, you can watch the R4 and R5 registers counting in binary coded decimal.



```
NAME   first
  ORG   0FFFEh
  DW   main

; counts up R4,R5 in binary coded decimal

  ORG   0200h
main  CLR   R4
  CLR   R5
loop  CLRC
  DADD.B  #1,R4
  CMP   #10h,R4
  JNE   loop
  CLR   R4
```

PC	SP	V——NZC	R04	R05	R06	R07	R08	R09
0206	0000	000000100	0008	0001	0000	0000	0000	0000

R10	R11	R12	R13	R14	R15	CYCLES	CYCLES ABS
0000	0000	0000	0000	0000	0000	0000000116	0000000116

TUTORIAL 5

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

USING LIBRARIES

If you are working on a large project you will soon accumulate a collection of useful routines that are used by several of your programs. To avoid having to assemble a routine each time the routine is needed, you can store such routines as object files, i.e., assembled but not linked.

A collection of routines in a single object file is referred to as a *library*. It is recommended that you use library files to create collections of related routines, such as a device driver.

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

- Change modules from PROGRAM to LIBRARY type, and vice versa.
- Add or remove modules from a library file.

- Change the names of entries.
- List module names, entry names, etc.

THE MAIN.S43 PROGRAM

The following listing shows the `main.s43` program. A copy of the program is provided with the product.

```
          NAME    main

          PUBLIC  main
          EXTERN  r_shift

          ORG     0FFFEh
          DW      main

          RSEG    PROM
main      MOV     #H'ABCD,R4
          MOV     #4,R5
          CALL    #r_shift
done_it   JMP     done_it

          END     main
```

This simply uses a routine called `r_shift` to shift the contents of register R4 to the right the number of times specified by the contents of register R5. The `EXTERN` directive declares `r_shift` as an external symbol, to be resolved at link time.

THE LIBRARY ROUTINES

The following two library routines will form a separately assembled library. It consists of the `r_shift` routine called by `main`, and a corresponding `l_shift` routine, both of which operate on the contents of the register variables R4 and R5. The file containing these library routines is called `shifts.s43`, and a copy is provided with the product.

```
          MODULE  r_shift
          PUBLIC  r_shift
          RSEG    PROM

r_shift   TST     R5
          JEQ     r_shift2
          RRA     R4
```

```
           DEC     R5
           JNE     r_shift
r_shift2   RET
           ENDMOD

           MODULE  l_shift
           PUBLIC  l_shift
           RSEG    prom
l_shift    TST     R5
           JEQ     l_shift2
           RLA     R4
           DEC     R5
           JNE     l_shift
l_shift2   RET

           END
```

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

The `r_shift` and `l_shift` entry addresses are made public to other modules with a `PUBLIC` directive.

For detailed information about the `MODULE` and `PUBLIC` directives, see the *MSP430 Assembler, Linker, and Librarian Programming Guide*.

CREATING A NEW PROJECT

Create a new project called `Project3`. Add the files `main.s43` and `shifts.s43` to the new project.

Then set up the target options to suit the project. Make sure that the **Processor Configuration** is set to `-v0`.

The procedure is described in *Creating a new project*, page 3.

ASSEMBLING AND LINKING THE SOURCE FILES

To assemble and link the `main.s43` and `shifts.s43` source files, you must first specify the linker command file to be used.

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

Page	*Option*
Include	`main.xcl` (a basic linker command file, designed for assembler-only projects)

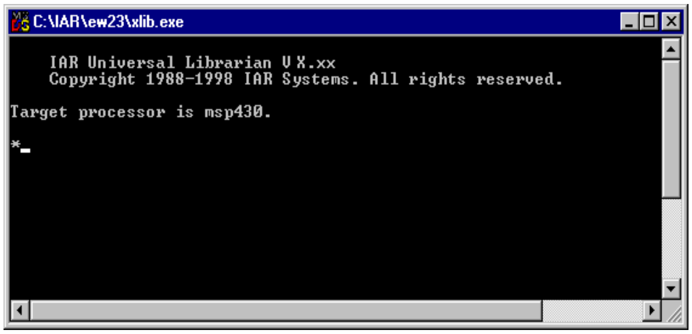
To assemble and link the `main.s43` and the `shifts.s43` files , select **Make** from the **Project** menu. Alternatively, select the **Make** button in the toolbar.

For more information about the XLINK options see the *MSP430 Assembler, Linker, and Librarian Programming Guide*.

USING THE IAR XLIB LIBRARIAN

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

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



```
    IAR Universal Librarian V X.xx
    Copyright 1988-1998 IAR Systems. All rights reserved.

Target processor is msp430.

*_
```

You can now enter XLIB commands at the * prompt.

Giving XLIB commands

Extract the modules you want from `shifts.r43` into a library called `math.r43`. To do this enter the command:

```
FETCH-MODULES
```

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

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

This creates the file `math.r43` which contains the code for the `r_shift` and `l_shift` routines.

You can confirm this by typing:

```
LIST-MODULES
```

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

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

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

Finally, leave the librarian by typing:

```
EXIT
```

or

```
QUIT
```

Then press Enter.

ADVANCED TUTORIALS

This chapter describes some of the more advanded features of the IAR development tools, which are very useful when you work on larger projects.

- The tutorials that follow both explore the features of C-SPY. In Tutorial 6 we define complex breakpoints, profile the application, and display code coverage. Tutorial 7 describes how to debug in disassembly mode.
- Tutorial 8 describes how to create a project containing both C and assembly language source files.

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

TUTORIAL 6

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

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

CREATING PROJECT4

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

Category	*Page*	*Option*
General	Target	Make sure that the **Processor Configuration** is set to `-v0` (default).
ICC430	Code Generation	Set the **Optimization** to **None**.
	Debug	Select **Generate debug information** (default).
	List	Select **List file**.
XLINK	Output	Select **Debug info with terminal I/O** (default).

Click **OK** to set the options.

Select **Make** from the **Project** menu, or select the **Make** button in the toolbar to compile and link the files. This creates the `project4.d43` file.

Start C-SPY to run the `project4.d43` program.

DEFINING COMPLEX BREAKPOINTS

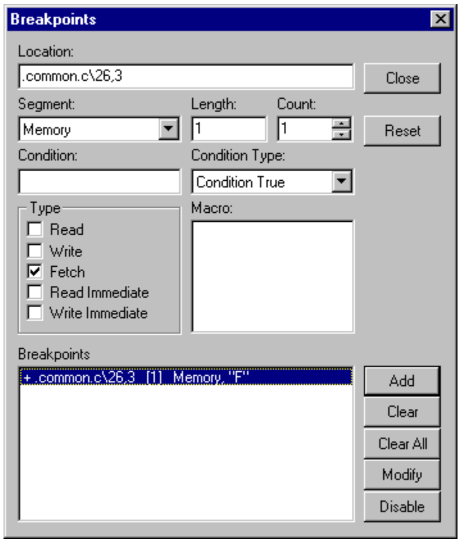
You can define complex breakpoint conditions in C-SPY, allowing you to detect when your program has reached a particular state of interest.

The file `project4.d43` should now be open in C-SPY. Execute one step. The current position should be the call to the `init_fib` function.

Then select **Step into** to move to the `init_fib` function. Set a breakpoint at the statement `i++`.

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

Choose **Edit Breakpoints…** from the **Control** menu to display the **Breakpoints** dialog box. Then select the breakpoint in the **Breakpoints** list to display information about the breakpoint you have defined:


Breakpoints

Location: .common.c\26,3

Segment: Memory

Length: 1

Count: 1

Condition:

Condition Type: Condition True

Type: Read [ ], Write [ ], Fetch [✓], Read Immediate [ ], Write Immediate [ ]

Macro:

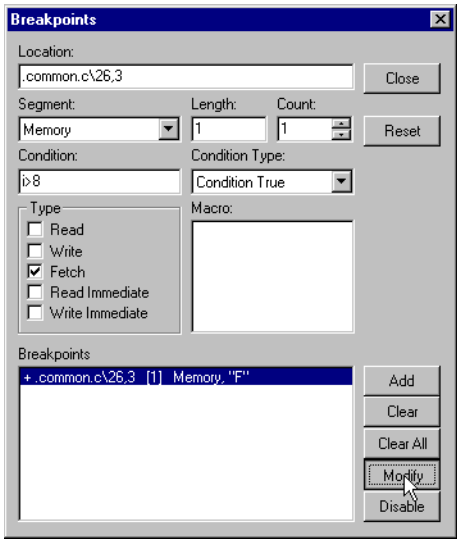
Breakpoints: + .common.c\26,3 [1] Memory, "F"

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.

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



Location: .common.c\26,3
Segment: Memory
Length: 1
Count: 1
Condition: i>8
Condition Type: Condition True
Type: Read [ ], Write [ ], Fetch [✓], Read Immediate [ ], Write Immediate [ ]
Macro:
Breakpoints: + .common.c\26,3 [1] Memory, "F"

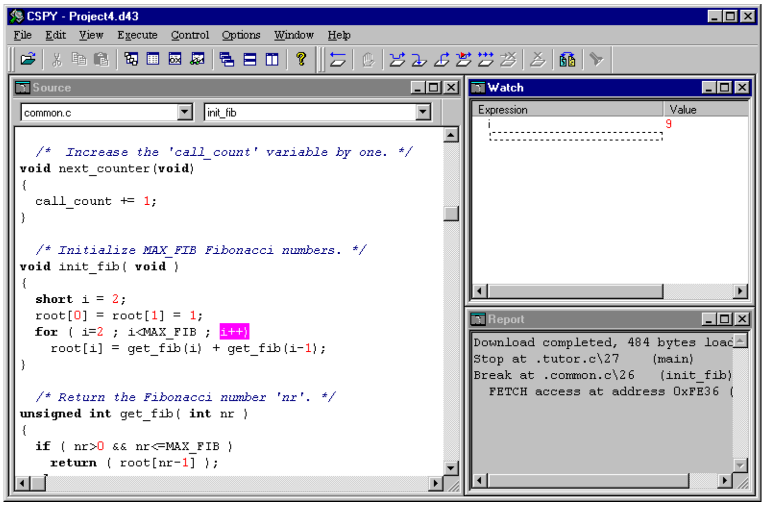
Finally, choose **Close** to close the **Breakpoints** dialog box.

Open the Watch window and add the variable `i`. The procedure is described in *Watching variables*, page 17.

Position the Source, Watch, and Report windows conveniently.

EXECUTING UNTIL A CONDITION IS TRUE

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



```
  /*  Increase the 'call_count' variable by one. */
void next_counter(void)
{
  call_count += 1;
}

  /* 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);
}

  /* Return the Fibonacci number 'nr'. */
unsigned int get_fib( int nr )
{
  if ( nr>0 && nr<=MAX_FIB )
    return ( root[nr-1] );
```

Expression	Value
i	9

```
Download completed, 484 bytes loac
Stop at .tutor.c\27     (main)
Break at .common.c\26    (init_fib)
  FETCH access at address 0xFE36 (
```

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.

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

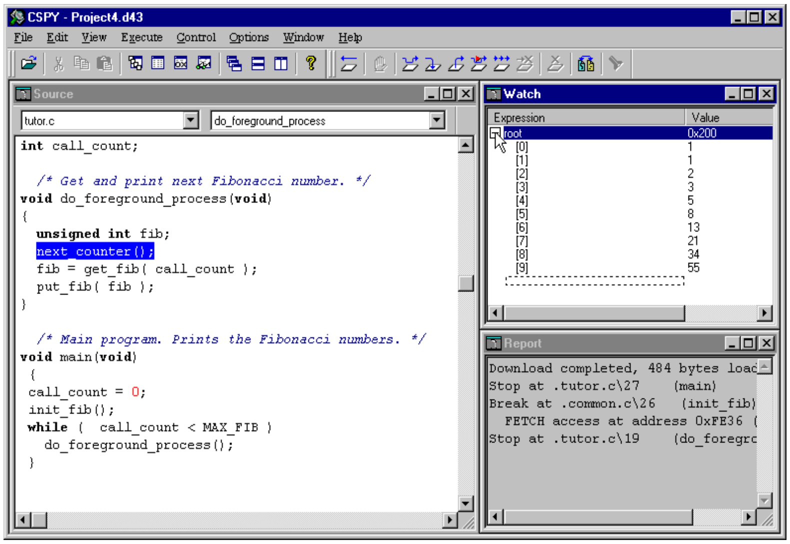
Then remove the variable `i` from the Watch window. Select the variable in the Watch window and press the Delete key. Instead add `root` to watch the array during execution.

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();
```

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



```
int call_count;

  /* Get and print next Fibonacci number. */
void do_foreground_process(void)
{
  unsigned int fib;
  next_counter();
  fib = get_fib( call_count );
  put_fib( fib );
}

  /* Main program. Prints the Fibonacci numbers. */
void main(void)
 {
 call_count = 0;
 init_fib();
 while (  call_count < MAX_FIB )
   do_foreground_process();
 }
```

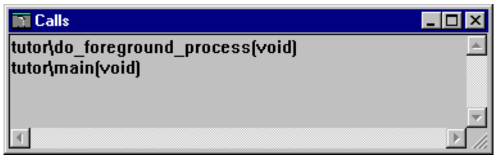
```
Expression      Value
root            0x200
  [0]           1
  [1]           1
  [2]           2
  [3]           3
  [4]           5
  [5]           8
  [6]           13
  [7]           21
  [8]           34
  [9]           55
```

```
Download completed, 484 bytes load
Stop at .tutor.c\27     (main)
Break at .common.c\26   (init_fib)
  FETCH access at address 0xFE36 (
Stop at .tutor.c\19     (do_foregro
```

DISPLAYING FUNCTION CALLS

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

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



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

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

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

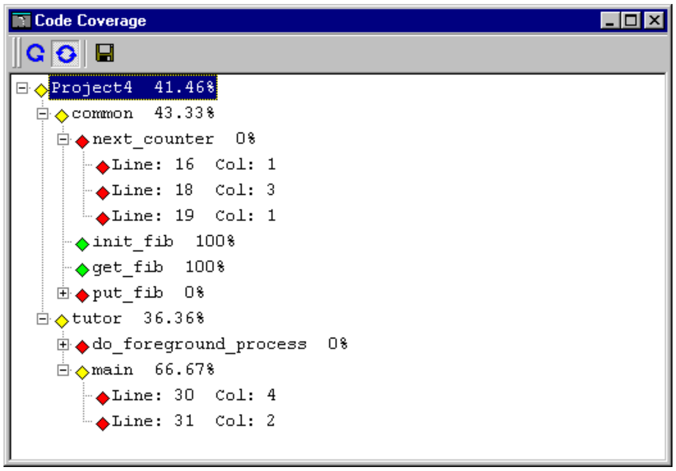
DISPLAYING CODE COVERAGE INFORMATION

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

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

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

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



```
Project4   41.46%
  common   43.33%
    next_counter   0%
      Line: 16   Col: 1
      Line: 18   Col: 3
      Line: 19   Col: 1
    init_fib   100%
    get_fib   100%
    put_fib   0%
  tutor   36.36%
    do_foreground_process   0%
    main   66.67%
      Line: 30   Col: 4
      Line: 31   Col: 2
```

PROFILING THE APPLICATION

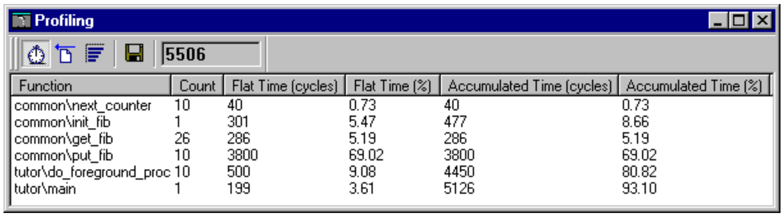
The profiling tool provides you with timing information on your application.

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

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

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

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



Function	Count	Flat Time (cycles)	Flat Time (%)	Accumulated Time (cycles)	Accumulated Time (%)
common\next_counter	10	40	0.73	40	0.73
common\init_fib	1	301	5.47	477	8.66
common\get_fib	26	286	5.19	286	5.19
common\put_fib	10	3800	69.02	3800	69.02
tutor\do_foreground_proc	10	500	9.08	4450	80.82
tutor\main	1	199	3.61	5126	93.10

The Profiling window contains the following information:

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

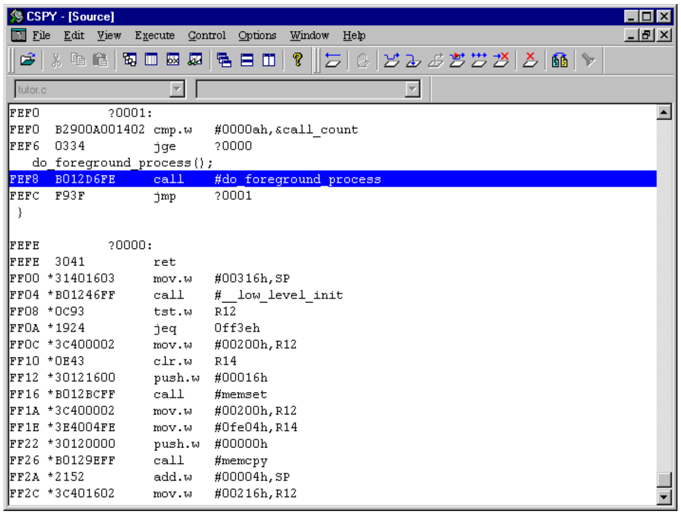
From the **Profiling** toolbar it is possible to display the profiling information graphically, to save the information to a file, or to start a new measurement.

TUTORIAL 7

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

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

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



```
FEF0          ?0001:
FEF0  B2900A001402 cmp.w   #0000ah,&call_count
FEF6  0334         jge     ?0000
   do_foreground_process();
FEF8  B012D6FE     call    #do_foreground_process
FEFC  F93F         jmp     ?0001
 }

FEFE          ?0000:
FEFE  3041         ret
FF00 *31401603     mov.w   #00316h,SP
FF04 *B01246FF     call    #__low_level_init
FF08 *0C93         tst.w   R12
FF0A *1924         jeq     0ff3eh
FF0C *3C400002     mov.w   #00200h,R12
FF10 *0E43         clr.w   R14
FF12 *30121600     push.w  #00016h
FF16 *B012BCFF     call    #memset
FF1A *3C400002     mov.w   #00200h,R12
FF1E *3E4004FE     mov.w   #0fe04h,R14
FF22 *30120000     push.w  #00000h
FF26 *B0129EFF     call    #memcpy
FF2A *2152         add.w   #00004h,SP
FF2C *3C401602     mov.w   #00216h,R12
```

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

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

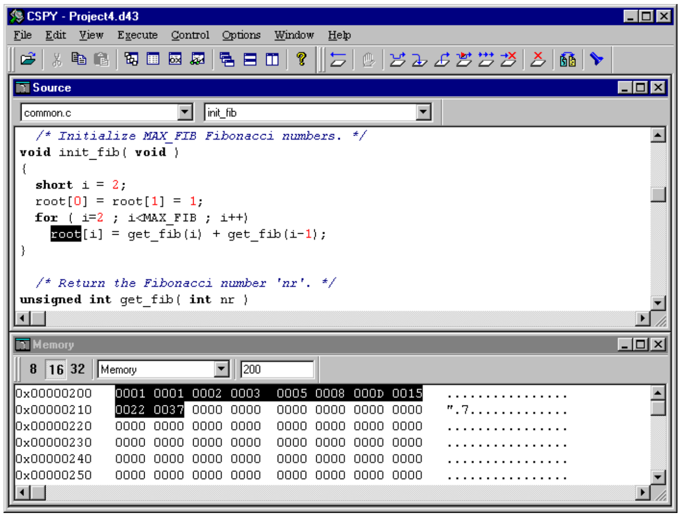
MONITORING MEMORY

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

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

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

Select `root` in the file `common.c`. Then drag it from the Source window and drop it into the Memory window. The Memory window will show the contents of memory corresponding to `root`:



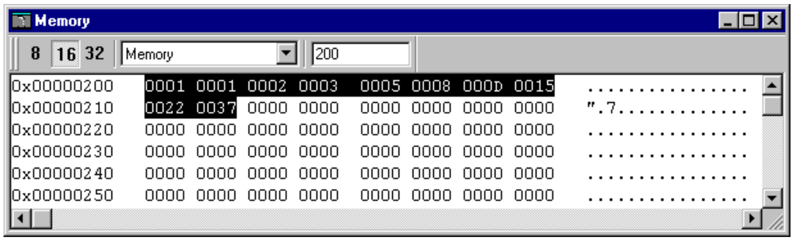
common.c — init_fib

```
  /* 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);
}

  /* Return the Fibonacci number 'nr'. */
unsigned int get_fib( int nr )
```

Memory — 200

```
0x00000200   0001 0001 0002 0003  0005 0008 000D 0015   ................
0x00000210   0022 0037 0000 0000  0000 0000 0000 0000   ".7.............
0x00000220   0000 0000 0000 0000  0000 0000 0000 0000   ................
0x00000230   0000 0000 0000 0000  0000 0000 0000 0000   ................
0x00000240   0000 0000 0000 0000  0000 0000 0000 0000   ................
0x00000250   0000 0000 0000 0000  0000 0000 0000 0000   ................
```

Since we are displaying `word` data, it is convenient to display the memory contents as words. Make sure the **16** button is selected in the Memory window toolbar:



Memory — 200

```
0x00000200   0001 0001 0002 0003  0005 0008 000D 0015   ................
0x00000210   0022 0037 0000 0000  0000 0000 0000 0000   ".7.............
0x00000220   0000 0000 0000 0000  0000 0000 0000 0000   ................
0x00000230   0000 0000 0000 0000  0000 0000 0000 0000   ................
0x00000240   0000 0000 0000 0000  0000 0000 0000 0000   ................
0x00000250   0000 0000 0000 0000  0000 0000 0000 0000   ................
```

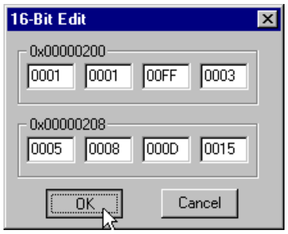
Notice that the 10 words have been initialized by the `init_fib` function of the C program.

CHANGING MEMORY

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

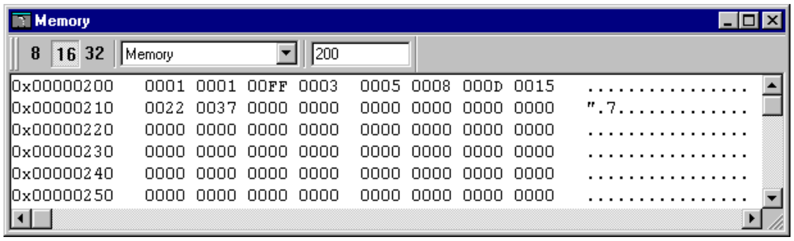
You can now edit the corresponding values directly in the memory.

For example, if you want to write the number 255 in the third position in number in the `root` array, select the value `0002` at address `0x204` in the Memory window and change it to `00FF` in the **16-Bit Edit** dialog box:



```
0x00000200
0001  0001  00FF  0003
0x00000208
0005  0008  000D  0015
```

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



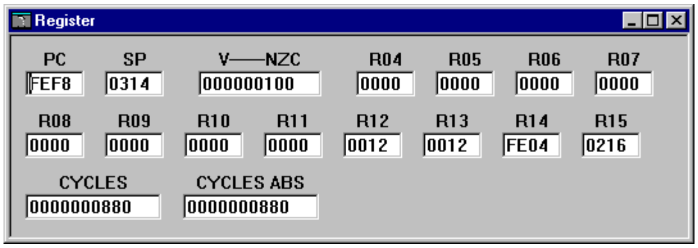
```
0x00000200    0001 0001 00FF 0003   0005 0008 000D 0015    ................
0x00000210    0022 0037 0000 0000   0000 0000 0000 0000    ".7.............
0x00000220    0000 0000 0000 0000   0000 0000 0000 0000    ................
0x00000230    0000 0000 0000 0000   0000 0000 0000 0000    ................
0x00000240    0000 0000 0000 0000   0000 0000 0000 0000    ................
0x00000250    0000 0000 0000 0000   0000 0000 0000 0000    ................
```

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

MONITORING REGISTERS

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

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


Register

PC	SP	V——NZC	R04	R05	R06	R07
FEF8	0314	000000100	0000	0000	0000	0000

R08	R09	R10	R11	R12	R13	R14	R15
0000	0000	0000	0000	0012	0012	FE04	0216

CYCLES	CYCLES ABS
0000000880	0000000880

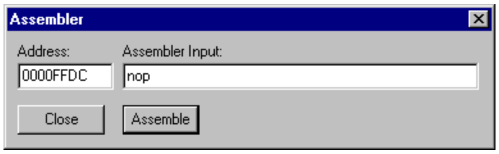
Select **Step** from the **Execute** menu, or click the **Step** button in the toolbar, to execute the next instructions, and watch how the values change in the Register window.

Then close the Register window.

CHANGING ASSEMBLER VALUES

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

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



Address: 0000FFDC
Assembler Input: nop

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

TUTORIAL 8

CREATING A COMBINED C COMPILER AND ASSEMBLER PROJECT

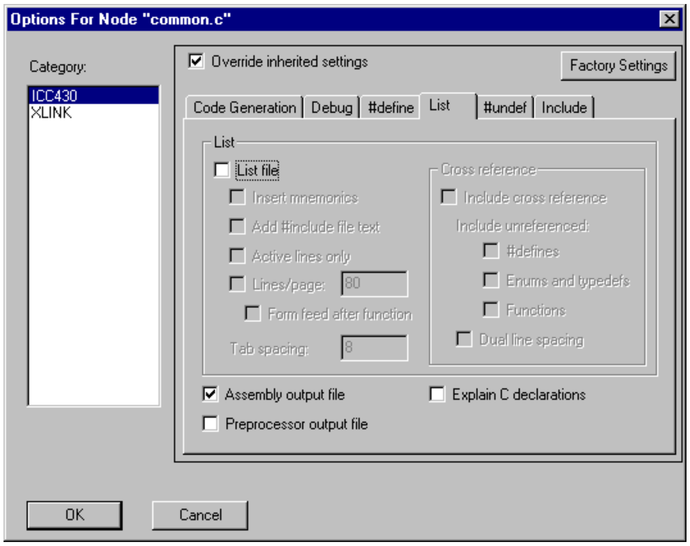
In large projects it may be convenient to use both C-written and assembly-written source files. In this tutorial we will demonstrate how they can be combined by substituting the file `common.c` with the assembler file `common.s43` and compiling the project.

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

Now you should create the assembler file `common.s43`. In the Project window, select the file `common.c`. Then select **Options...** from the **Project** menu. You will notice that only the **ICC430** and **XLINK** categories are available.

In the **ICC430** category, select **Override inherited settings** and set the following options:

Page	*Option*
List	Deselect List file.
	Select Assembly output file.


Options For Node "common.c"

Category: ICC430 (selected), XLINK

[✓] Override inherited settings

Page: List

- [ ] List file
- [✓] Assembly output file
- [ ] Explain C declarations
- [ ] Preprocessor output file

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

Compile each of the files. To see how the C code is represented in assembly language, open the file `common.s43` that was created from the file `common.c`.

Now modify `Project4` by removing the file `common.c` and adding the file `common.s43` instead. Then select **Make** from the **Project** menu to relink `Project4`.

Start C-SPY to run the `project4.d43` program and see that it behaves like in the previous tutorials.