

1. CPE 325: Laboratory Assignment #1

Introduction to MSP430 IAR Embedded Workbench – Integrated Development Environment (IDE)

Objectives: This tutorial will help you get started with the MSP30 IAR Embedded Workbench and includes the following topics:

- Creating an application project
- Debugging using the IAR C-SPY® Debugger

Notes:

The latest version of IAR Kickstart can be downloaded for free from the TI's MSP430 web site:

[Download IAR Embedded Workbench Kickstart](http://processors.wiki.ti.com/index.php/IAR_Embedded_Workbench_for_TI_MSP430)

(http://processors.wiki.ti.com/index.php/IAR_Embedded_Workbench_for_TI_MSP430).

1.1. Creating an Application Project

This section introduces you to the IAR Embedded Workbench integrated development environment (IDE) that will be used for software development in the Embedded Systems Laboratory. In a form of a step-by-step tutorial it demonstrates a typical development cycle and shows you how to use the IAR compiler and the IAR linker to create a small application for the MSP430 microcontroller. It includes topics such as, creating a workspace, setting up a project with C source files, compiling and linking your application, and debugging.

1.1.1. Creating a New Workspace

Using the IAR Embedded Workbench IDE, you can design advanced project models. For more information read the MSP430 IAR Embedded Workbench® IDE User Guide. Here, we will walk through a relatively simple project with several source files.

Step 1. Create a working directory (e.g., projects, and then lab1).

Step 2. Copy the following source files (Lab1_D1.c, twofact.c) to your working directory projects/lab1 (the files are available at http://www.ece.uah.edu/~milenska/msp430/cpe323/lab1/Demo_codes/).

Step 3. Create a new workspace window.

Before you can create your project you must first create a workspace. When you start the IAR you will see the screen as shown in Figure 1. It contains the information you need to get started with the IAR for MSP430 including tutorials, example projects, user guides, and other relevant information. At any point of time you can get to this window by pressing Help>Information Center for MSP430.

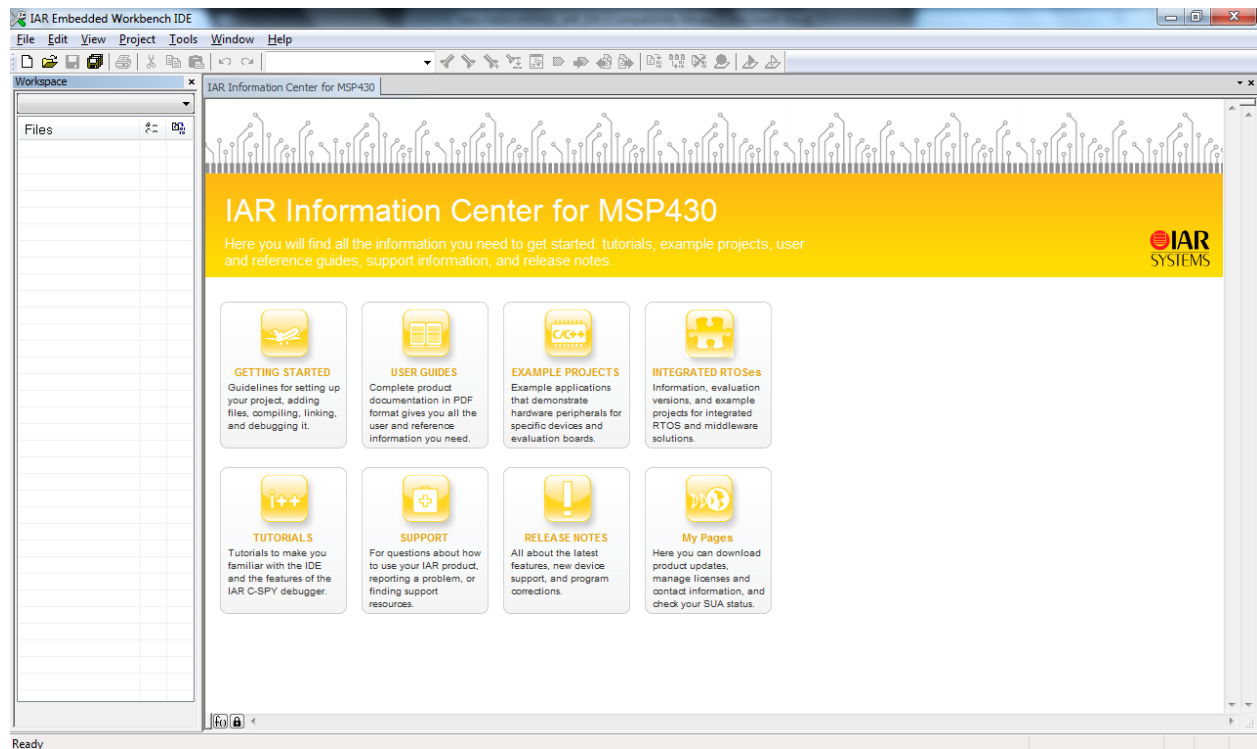


Figure 1. IAR Embedded Workbench v5.40.7 Startup (Information Center).

1.1.2. Setting up a New Project

Step 1. Create a new project.

1.1. To create a new project, choose *Project>Create New Project*. The *Create New Project* dialog box appears (Figure 2), which lets you base your new project on a project template. Make sure the Tool chain is set to MSP430, and click *OK*. For this tutorial, select the project template *Empty project*, which simply creates an empty project that uses default project settings.

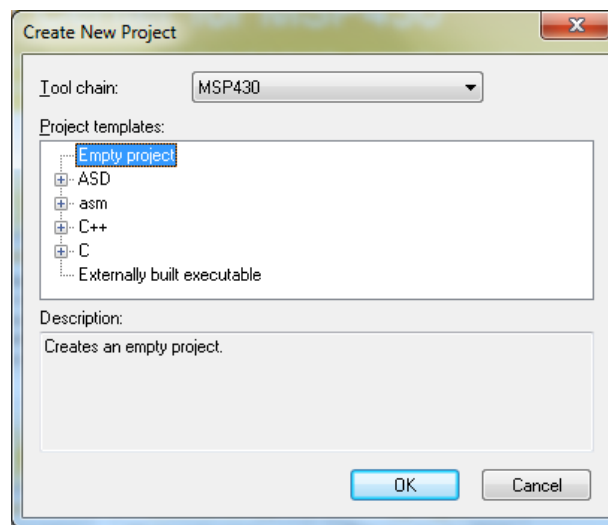


Figure 2. Create New Project dialog box.

In the standard *Save As* dialog box that appears, specify where you want to place your project file, that is, in your newly created *projects/lab1* directory. Type the project name (e.g., *Lab1_D1*) in the *File* name box, and click *Save* to create the project. The project will appear in the Workspace window.

By default two build configurations are created: Debug and Release. In this tutorial only the Debug configuration is used. You may choose the build configuration from the drop-down menu at the top of the window. The asterisk in the project name indicates that there are changes that have not been saved.

A project file—with the filename extension **ewp**—will be created in the *projects/lab1* directory, not immediately, but later on when you save the workspace. This file contains information about your project-specific settings, such as build options.

1.2. Choose *File>Save Workspace* and specify where you want to place your workspace file. In this tutorial, you should place it in your *projects* directory. Type *tutorials* in the File name box, and click *Save* to create the new workspace.

A workspace file—with the filename extension **eww**—has now been created in the *projects* directory. This file lists all projects that you will add to the workspace (you may have one workspace for all your laboratory assignments). Information related to the current session, such as the placement of windows and breakpoints is located in the files created in the *projects/lab1/settings* directory.

Step 2. Adding Files to the Project.

This tutorial uses the source files *Lab1_D1.c* and *twofact.c*.

- The *Lab1_D1.c* is a simple program that calls two functions calculating factorial of an integer input. The first one *ifact* returns the integer result, and the second one *lifact* returns the long integer type. Only standard features of the C language are used and the results are printed on the stdout (Terminal I/O in this example).
- The *twofact.c* contains factorial functions *ifact* and *lifact*.

In the Workspace window, select the destination to which you want to add a source file; a group or, as in this case, directly to the project.

Choose *Project>Add Files* to open a standard browse dialog box. Locate the files *Lab1_D1.c* and *twofact.c*, select them in the file selection list, and click *Open* to add them to the *Lab1_D1* project (Figure 3).

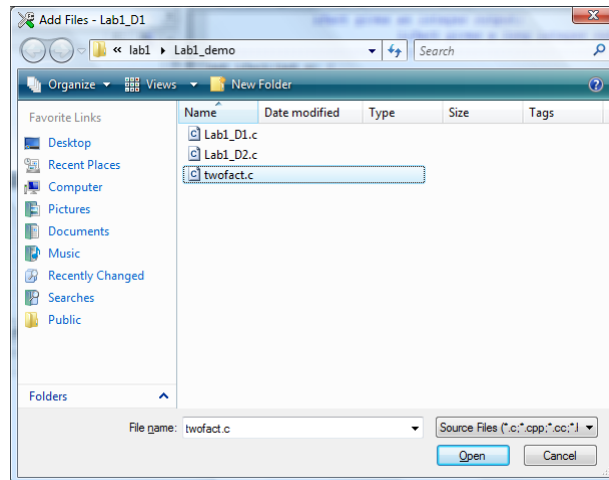


Figure 3. Adding files to Lab1_D1.

Step 3. Setting Project Options.

Now you will set the project options. For application projects, options can be set on all levels of nodes. First you will set the general options to suit the processor configuration in this tutorial. Because these options must be the same for the whole build configuration, they must be set on the project node.

3.1. Select the project folder icon for Lab1_D1 with the Debug configuration in the Workspace window and choose *Project>Options*.

The Target options page in the General Options category is displayed (Figure 4).

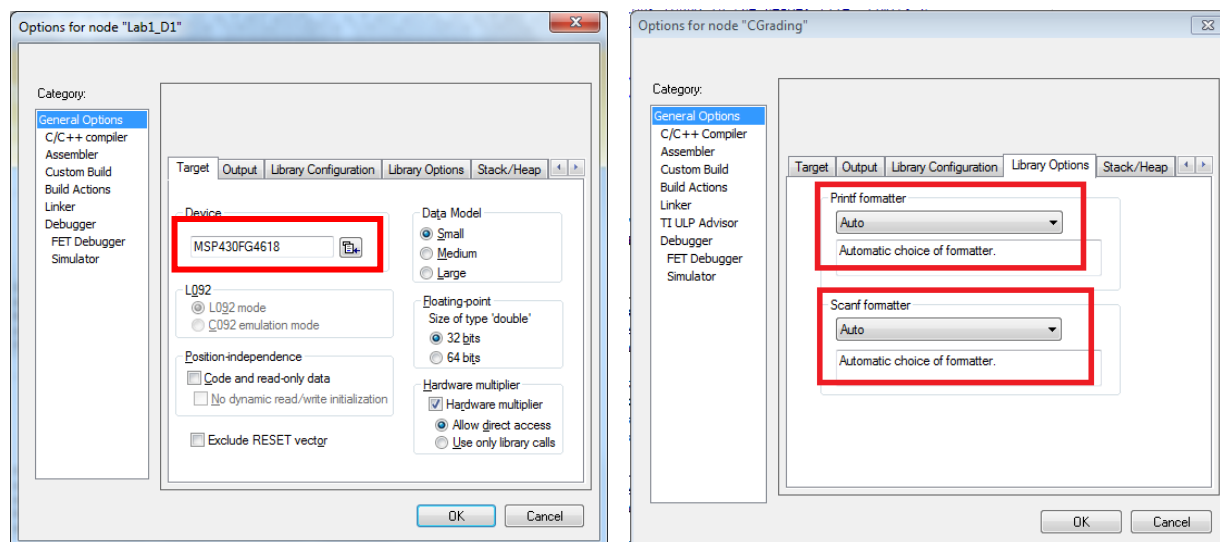


Figure 4. Setting general options.

Select the following settings:

<i>Page</i>	<i>Setting</i>
<i>Target Device:</i>	<i>MSP430FG4618</i>
<i>Output Output file:</i>	<i>Executable</i>
<i>Library Configuration Library:</i>	<i>Normal DLIB</i>
<i>Library options: Printf formatter</i>	<i>Auto</i>
<i>Library options: Scanf formatter</i>	<i>Auto</i>

3.2. Select C/C++ Compiler in the Category list to display the compiler option pages (Figure 5).

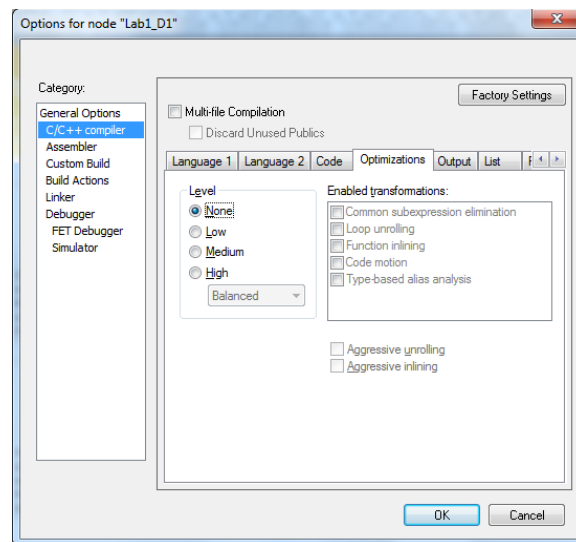


Figure 5. Setting C/C++ compiler options.

Select the following settings for the C/C++ compiler:

<i>Page</i>	<i>Setting</i>
<i>Optimizations</i>	<i>Optimizations, Size: None (Best debug support)</i>
<i>Output</i>	<i>Generate debug information</i>
<i>List</i>	<i>Output list file/Assembler mnemonics</i>

3.3. Click *OK* to set the options you have specified.

Note: It is possible to customize the amount of information to be displayed in the Build messages window. In this tutorial, the default setting is not used. Thus, the contents of the Build messages window on your screen might differ from the screen shots.

The project is now ready to be built.

1.1.3. Compiling and Linking the Application

You can now compile and link the application. You should also create a compiler list file and a linker map file and view both of them.

Step 1. Compiling the source files.

To compile the file `twofact.c`, select it in the Workspace window.

1.1. Choose *Project>Compile*. Alternatively, click the *Compile* button in the toolbar or choose the Compile command from the context menu that appears when you right-click on the selected file in the Workspace window. The progress will be displayed in the Build messages window.

1.2. Compile the file `Lab1_D1.c` in the same manner.

The IAR Embedded Workbench IDE has now created new directories in your project directory. Because you are using the build configuration Debug, a Debug directory has been created containing the directories List, Obj, and Exe:

- The List directory is the destination directory for the list files. The list files have the extension `lst`. For example, `twofact.lst` is generated containing a C source code with the corresponding assembly language mnemonics for `twofact.c` source file.
- The Obj directory is the destination directory for the object files from the compiler and the assembler. These files have the extension `r43` and will be used as inputs to the IAR XLINK Linker.
- The Exe directory is the destination directory for the executable file. It has the extension `d43` and will be used as an input to the IAR C-SPY® Debugger. Note that this directory will be empty until you have linked the object files.

1.3. Click on the plus signs in the Workspace window to expand the view (Figure 6). As you can see, IAR Embedded Workbench has also created an output folder icon in the Workspace window containing any generated output files. All included header files are displayed as well, showing the dependencies between the files.

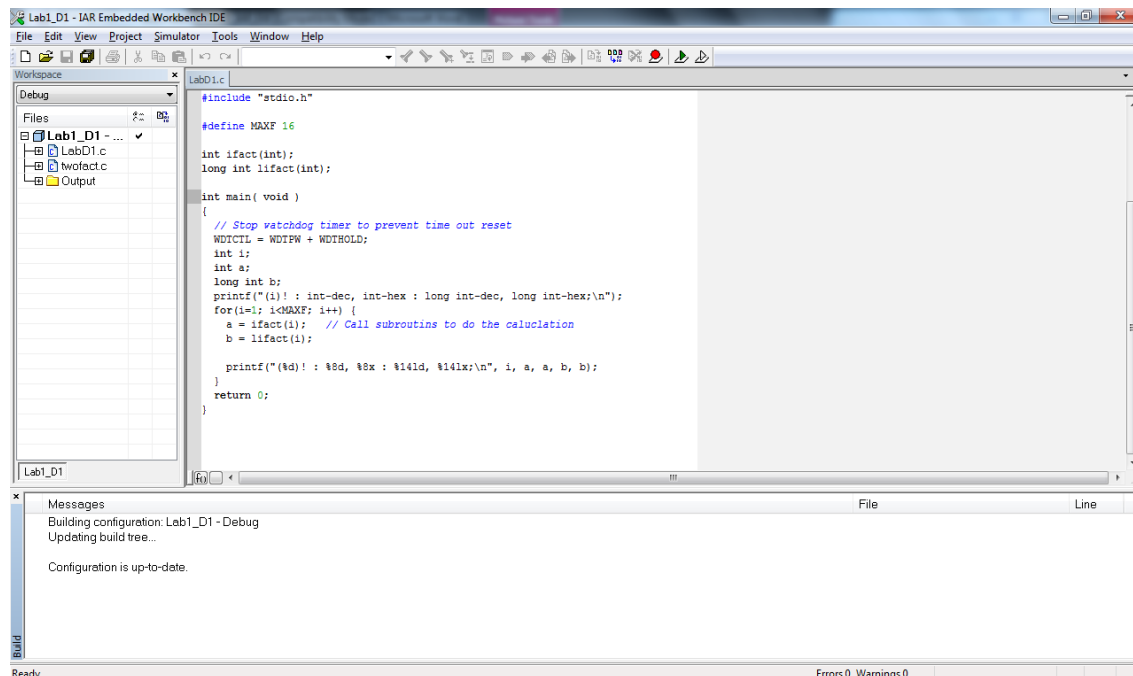


Figure 6. Workspace window after compilation.

Step 2. Viewing the List file.

Now examine the compiler list file and notice how it is automatically updated when you, as in this case, will investigate how different optimization levels affect the generated code size.

2.1. Open the list file *twofact.lst* by double-clicking it in the Workspace window. Examine the list file, which contains the following information:

- The header shows the product version, information about when the file was created, and the command line version of the compiler options that were used
- The body of the list file shows the assembler code and binary code generated for each statement. It also shows how the variables are assigned to different segments
- The end of the list file shows the amount of stack, code, and data memory required, and contains information about error and warning messages that might have been generated.

Notice the amount of generated code at the end of the file and keep the file open. It is (assuming you use no optimizations):

```
CSTACK Function
-----
6  ifact
8  lifact
```

Segment part sizes:

Bytes	Function/Label
52	ifact
64	lifact

116 bytes in segment CODE

116 bytes of CODE memory

2.2. Choose *Tools>Options* to open the IDE Options dialog box and click the Editor tab. Select the option *Scan for Changed Files*. This option turns on the automatic update of any file open in an editor window, such as a list file. Click the *OK* button.

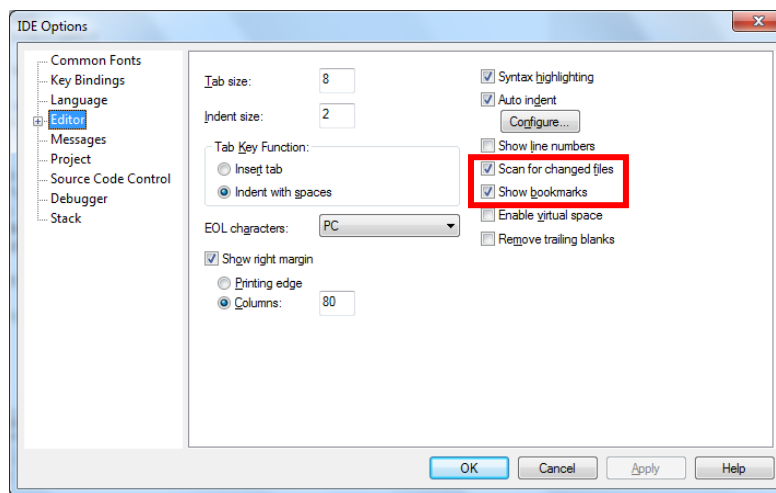


Figure 7. Setting the option *Scan for Changed Files*.

2.3. Select the file *twofact.c* in the Workspace window. Open the *C/C++ Compiler* options dialog box by right-clicking on the selected file in the Workspace window.

Click the *Optimizations* tab and select the *Override* inherited settings option. Choose High from the Optimizations radio button list. Click OK.

Notice that the options override on the file node is indicated in the Workspace window.

2.4. Recompile the source files. Now you will notice two things. First, note the automatic updating of the open list file due to the selected option *Scan for Changed Files*. Second, look at the end of the list file and notice the effect on the code size due to the increased optimization.

CSTACK	Function
8	ifact
8	lifact

Segment part sizes:

Bytes	Function/Label
36	ifact
46	lifact

82 bytes in segment CODE
82 bytes of CODE memory

2.5. For this tutorial, the optimization level “None” should be used, so before linking the application, restore the default optimization level. Open the *C/C++ Compiler* options dialog box by right-clicking on the selected file in the Workspace window.

Step 3. Linking the application.

Now you should set up the options for the IAR XLINK Linker.

3.1. Select the project folder icon Lab1_D1 - Debug in the Workspace window and choose *Project>Options* (Figure 8). Then select Linker in the Category list to display the XLINK option pages. For this tutorial, default factory settings are used. However, pay attention to the choice of output format and linker command file.

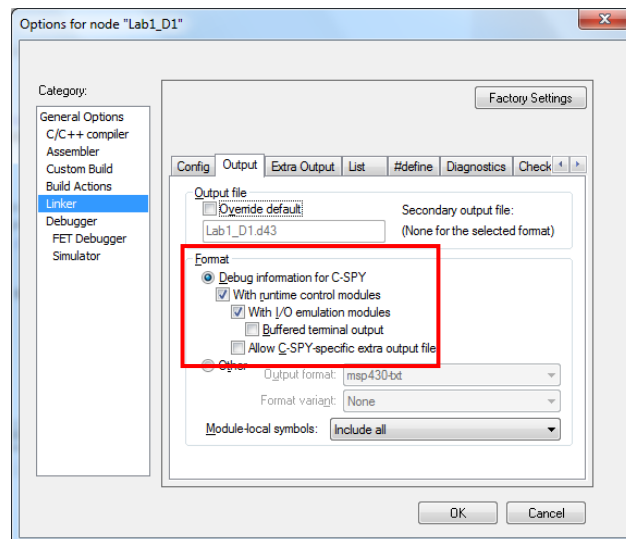


Figure 8. XLINK options dialog box.

Output format

It is important to choose the output format that suits your purpose. You might want to load it to a debugger—which means that you need output with debug information. In this tutorial you will use the default output options suitable for the C-SPY debugger—*Debug information for C-SPY*, *With runtime control modules*, and *With I/O emulation modules*—which means that some low-level routines will be linked that direct stdin and stdout to the Terminal I/O window in the C-SPY Debugger. You find these options on the *Output* page.

Alternatively, in your real application project, you might want to load the output to a PROM programmer—in which case you need an output format without debug information, such as Intel-hex or Motorola S-records.

Linker command file

In the linker command file, the XLINK command line options for segment control are used for placing segments. It is important to be familiar with the linker command file and placement of segments. You can read more about this in the MSP430 IAR C/C++ Compiler Reference Guide.

The linker command file templates supplied with the product can be used as is in the simulator, but when using them for your target system you might have to adapt them to your actual hardware memory layout. You can find supplied linker command files in the config directory. In this tutorial you will use the default linker command file, which you can see on the Config page.

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, and verify that the definitions match your requirements.

Linker map file

By default no linker map file is generated. To generate a linker map file, click the List tab and select the options Generate linker listing, Segment map, and Module map.

3.2. Click OK to save the XLINK options. Now you should link the object file, to generate code that can be debugged.

3.3. Choose Project>Make. The progress will as usual be displayed in the Build messages window. The result of the linking is a code file Lab1_D1.d43 with debug information and a map file Lab1_D1.map.

Step 4. Viewing the map file.

Examine the file Lab1_D1.map to see how the segment definitions and code were placed in memory. These are the main points of interest in a map file:

- The header includes the options used for linking.
- The CROSS REFERENCE section shows the address of the program entry.
- The RUNTIME MODEL section shows the runtime model attributes that are used.
- The MODULE MAP shows the files that are linked. For each file, information about the modules that were loaded as part of your application, including segments and global symbols declared within each segment, is displayed.
- The SEGMENTS IN ADDRESS ORDER section lists all the segments that constitute your application.

The Lab1_D1.d43 application is now ready to be run in the IAR C-SPY Debugger.

1.2. Debugging Using the IAR C-Spy Debugger

This section continues the development cycle started in the previous section and explores the basic features of the IAR C-SPY Debugger.

1.2.1. Starting the C-SPY Debugger

Before starting the IAR C-SPY Debugger you must set a few C-SPY options.

Step 1. Choose *Project>Options* and then the *Debugger* category (Figure 9). On the *Setup* page, make sure that you have chosen *Simulator* from the *Driver* drop-down list and that *Run to main* is selected. Click OK.

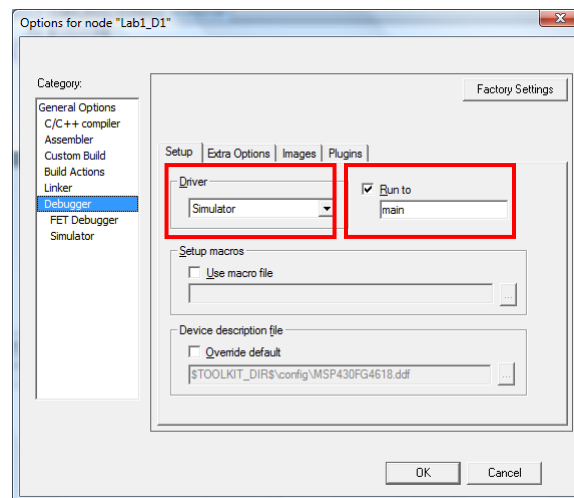


Figure 9. Setting the debugger options.

Step 2. Choose *Project>Debug*. Alternatively, click the *Debugger* button in the toolbar. The IAR C-SPY Debugger starts with the lab1_D1.d43 application loaded. In addition to the windows already opened in the Embedded Workbench, a set of C-SPY-specific windows are now available (Figure 10).

Make sure the following windows and window contents are open and visible on the screen: the Workspace window with the active build configuration tutorials – Lab1_D1, the editor window with the source files *Lab1_D1.c* and *twofact.c*, and the *Debug Log* window.

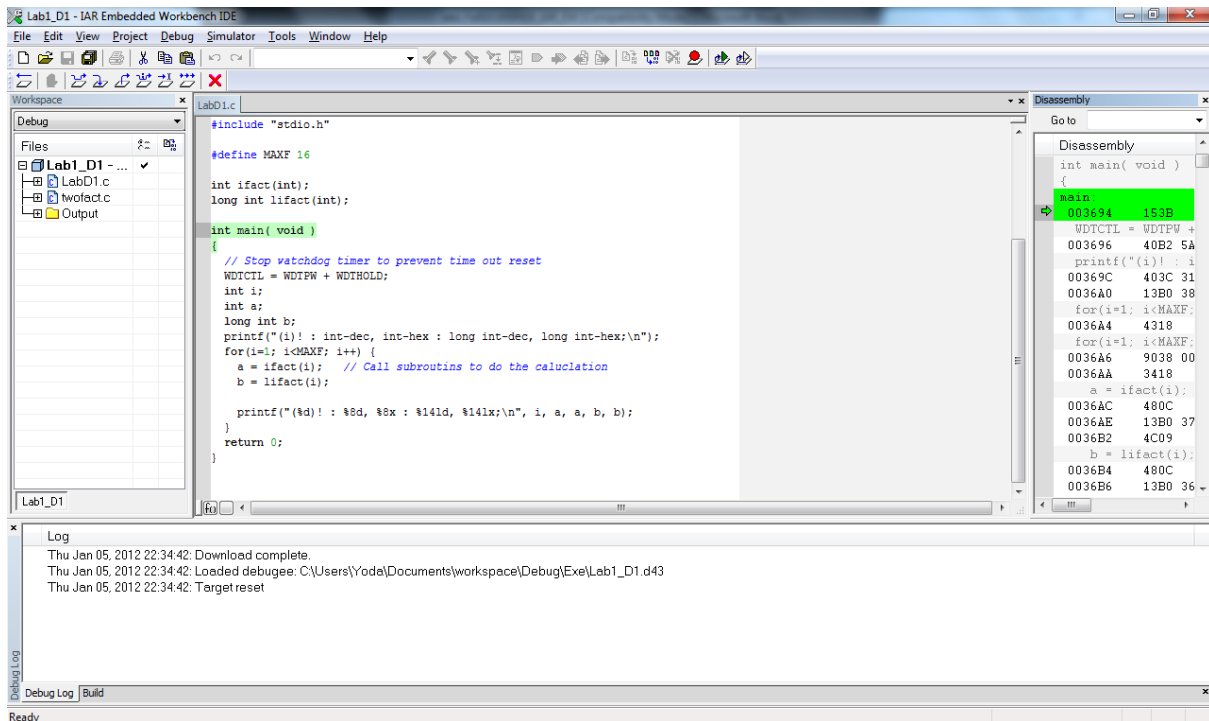



Figure 10. The C-SPY debugger main window.

1.2.2. Inspecting Source Statements

To inspect the source statements, double-click the file Lab1_D1.c in the Workspace window.

Step 1. With the file Lab1_D1.c displayed in the editor window, first step over with the Debug>Step Over command. 


Alternatively, click the Step Over button on the toolbar.

Step 2. Choose Debug>Step Into to step into the functions ifact and lifact.

Alternatively, click the Step Into button on the toolbar. 

At source level, the Step Over and Step Into commands allow you to execute your application a statement or instruction at a time. The Step Into continues stepping inside function or subroutine calls, whereas Step Over executes each function call in a single step. When the Step Into is executed you will notice that the active window changes to twofact.c as the functions are located in this file.

Step 3. Use the Step Into command until you reach the end of the ifact function.

Step 4. You can also step on a statement level. Choose the Debug>Next statement to execute one statement at a time. Alternatively, click the Next statement button on the toolbar. Notice how this command differs from the Step Over and the Step Into commands. 

Explore other options (Step out, Run to, etc).

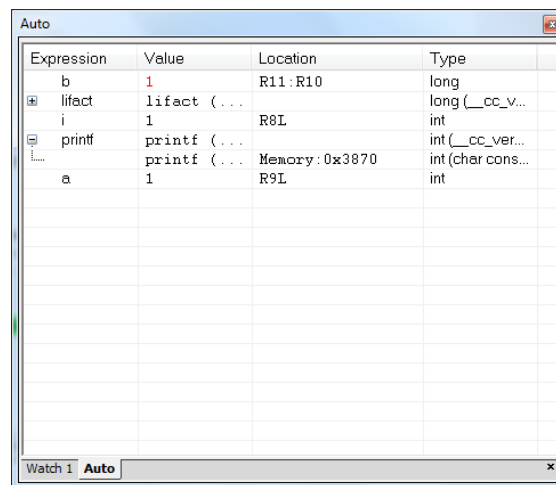
1.2.3. Inspecting Variables

C-SPY allows you to watch variables or expressions in the source code, so that you can keep track of their values as you execute your application. You can look at a variable in a number of ways; for example by pointing at it in the source window with the mouse pointer, or by opening one of the Locals, Watch, Live Watch, or Auto windows.

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

Using the Auto Window

Choose View>Auto to open the Auto window. Keep stepping to see how the values change.



The screenshot shows the 'Auto' window in a debugger. It contains a table with four columns: Expression, Value, Location, and Type. The table lists several variables and expressions, including 'b', 'lifact', 'i', 'printf', and 'a'. The values are displayed in the 'Value' column, and the locations are shown in the 'Location' column. The 'Type' column shows the data type for each variable.

Expression	Value	Location	Type
b	1	R11:R10	long
lifact	lifact (...)		long (__cc_v...
i	1	R8L	int
printf	printf (...)		int (__cc_ver...
printf	printf (...)	Memory: 0x3870	int (char cons...
a	1	R9L	int

Figure 11. Inspecting variables in the Auto window.

1.2.4. Setting a Watchpoint

Next you will use the Watch window to inspect variables.

Choose View>Watch to open the Watch window. Notice that it is by default grouped together with the currently open Auto window; the windows are located as a tab group.

Set a watchpoint on the variable *i* using the following procedure: Click the dotted rectangle in the Watch window. In the entry field that appears, type *i* and press the Enter key. You can also drag a variable from the editor window to the Watch window.

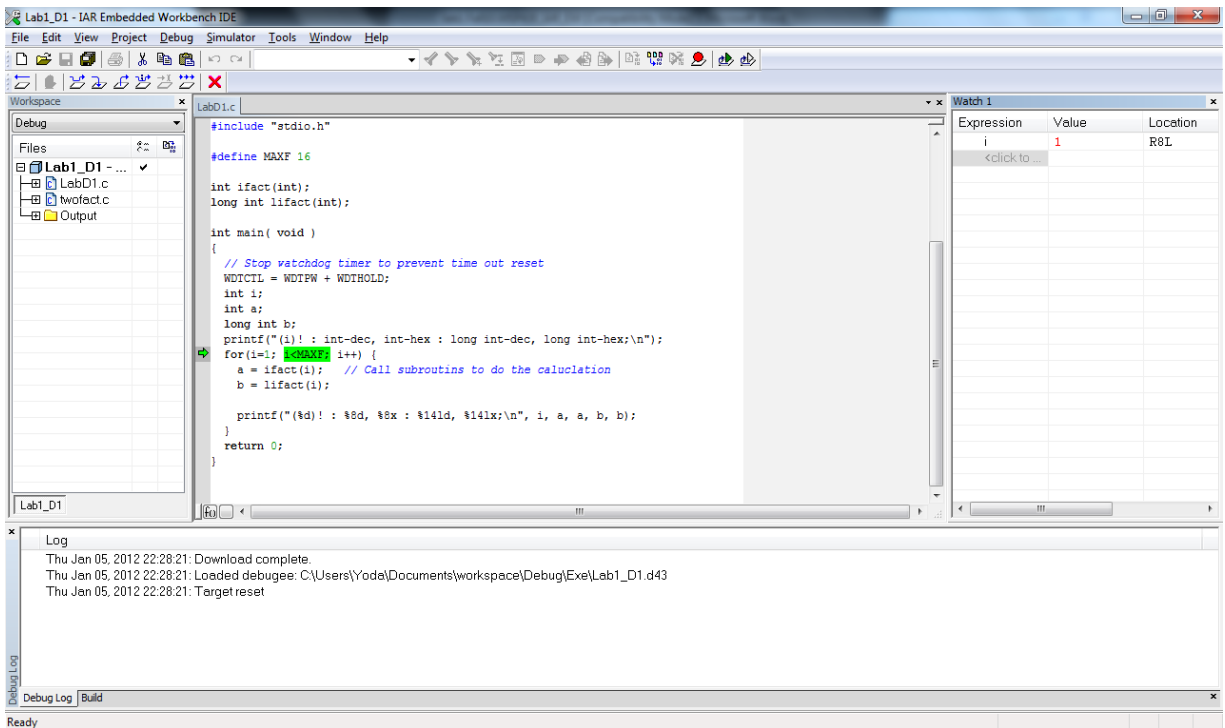


Figure 12. Watching variables in the Watch window.

Execute some more steps to see how the values of the variable `i` change.

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

1.2.5. Setting and Monitoring Breakpoints

The IAR C-SPY Debugger contains a powerful breakpoint system with many features. For more information read the Help> MSP430 IAR Embedded Workbench® IDE User Guide.

The most convenient way is usually to set breakpoints interactively, simply by positioning the insertion point in or near a statement and then choosing the Toggle Breakpoint command.

Set a breakpoint on a statement using the following procedure: First, click the `twofact.c` tab in the editor window and click in the statement to position the insertion point. Then choose *Edit>Toggle Breakpoint*.

Alternatively, click the *Toggle Breakpoint* button on the toolbar.

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

To view all defined breakpoints, choose *View>Breakpoints* to open the Breakpoints window. You can find information about the breakpoint execution in the Debug Log window.

1.2.6. Executing up to a Breakpoint

To execute your application until it reaches the breakpoint, choose *Debug>Go*. Alternatively, click the Go button on the toolbar.

The application will execute up to the breakpoint you set.

Select the breakpoint and choose *Edit>Toggle Breakpoint* to remove the breakpoint.

1.2.7. Debugging in Disassembly Mode

Debugging with C-SPY is usually quicker and more straightforward in C/C++ source mode. However, if you want to have full control over low-level routines, you can debug in disassembly mode where each step corresponds to one assembler instruction. C-SPY lets you switch freely between the two modes.

First reset your application by clicking the Reset button on the toolbar.

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

Try the different step commands also in the Disassembly window.

1.2.8. Monitoring Registers

The Register window lets you monitor and modify the contents of the processor registers. Notice registers PC (Program Counter), SP (Stack Pointer), SR (Status Register), R4-R15 (general-purpose registers), CYCLECOUNTER (this is actually not a real register, rather it serves as a clock cycle counter so you can determine the number of clock cycles each instruction takes to execute; it can also be used to determine execution time of functions or whole programs).

Register

CPU Registers

PC	= 0x03992	R11	= 0x02868
SP	= 0x030F4	R12	= 0x00001
<input checked="" type="checkbox"/> SR	= 0x0001	R13	= 0x030F8
R4	= 0xADB00	R14	= 0x00000
R5	= 0xC5300	R15	= 0x0000A
R6	= 0x05F99	CYCLECOUNTER	= 473234
R7	= 0x0718C	CCTIMER1	= 473234
R8	= 0x01FB0	CCTIMER2	= 473234
R9	= 0x0747B	CCSTEP	= 473234
R10	= 0x00000		

Figure 13. Register window.

Choose *View>Register* to open the Register window.

Step Over to execute the next instructions, and watch how the values change in the Register window.

Close the Register window.

1.2.9. Monitoring Memory

The Memory window (Figure 14) lets you monitor selected areas of memory. You can select RAM, flash, or SFR portion of the memory. Similarly you can select *View->Stack* option to inspect the current state of the program stack.

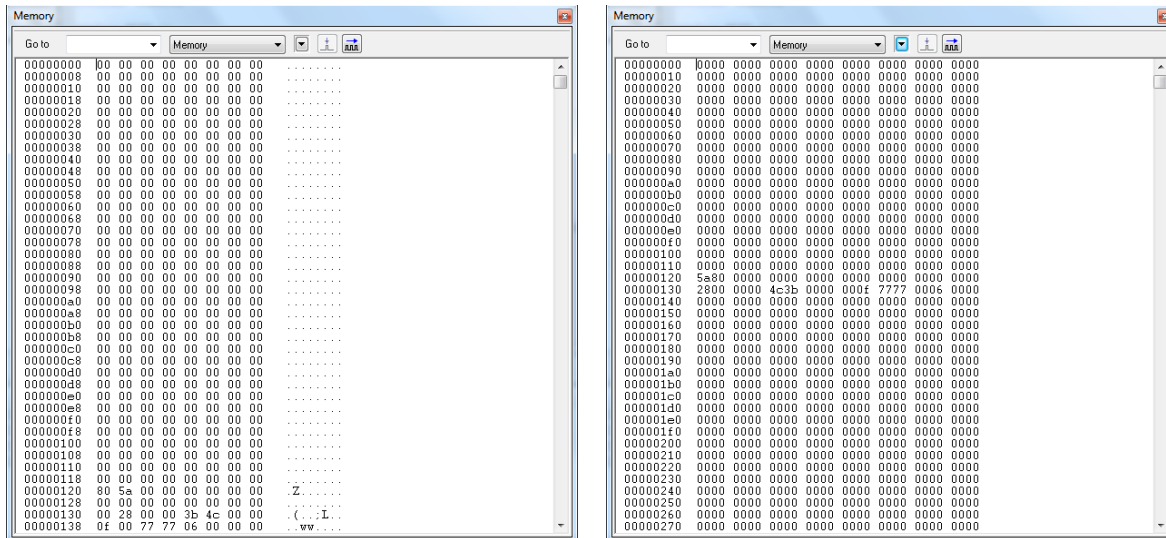


Figure 14. Memory window.

To display the memory contents as 16-bit data units, choose the x2 Units command from the drop-down arrow menu on the Memory window toolbar.

If not all of the memory units have been initialized yet, continue to step over and you will notice how the memory contents will be updated.

You can change the memory contents by editing the values in the Memory window. Just place the insertion point at the memory content that you want to edit and type the desired value.

Close the Memory window.

1.2.10. Viewing Terminal I/O

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

Note: The Terminal I/O window is only available in C-SPY if you have linked your project using the output option With I/O emulation modules. This means that some low-level routines will be linked that direct stdin and stdout to the Terminal I/O window.

Choose View>Terminal I/O to display the output from the I/O operations.

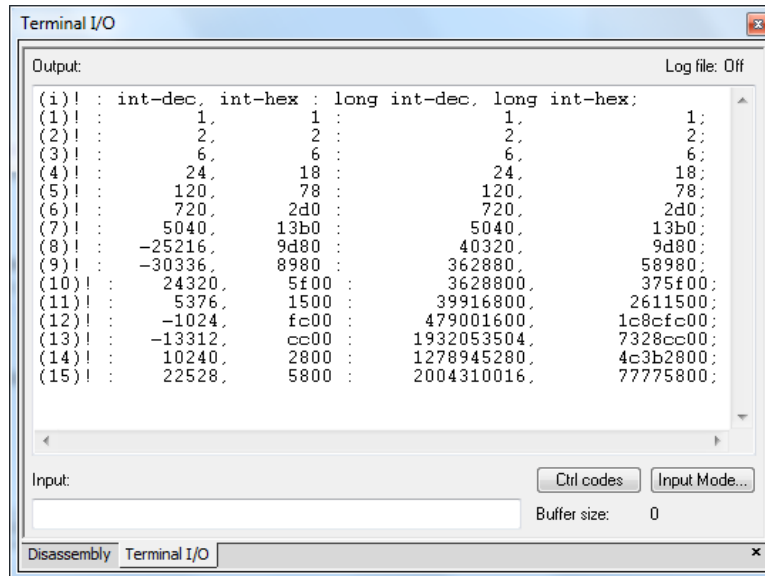


Figure 15. Terminal I/O window.

The contents of the window depend on how far you have executed the application.

1.2.11. Reaching Program Exit

To complete the execution of your application, choose *Debug>Go*.

Alternatively, click the Go button on the toolbar.

As no more breakpoints are encountered, C-SPY reaches the end of the application and a program exit reached message is printed in the *Debug Log* window. Notice a warning about the Stack. Its size can be increased (e.g., to 256 bytes) from the default (80 bytes).

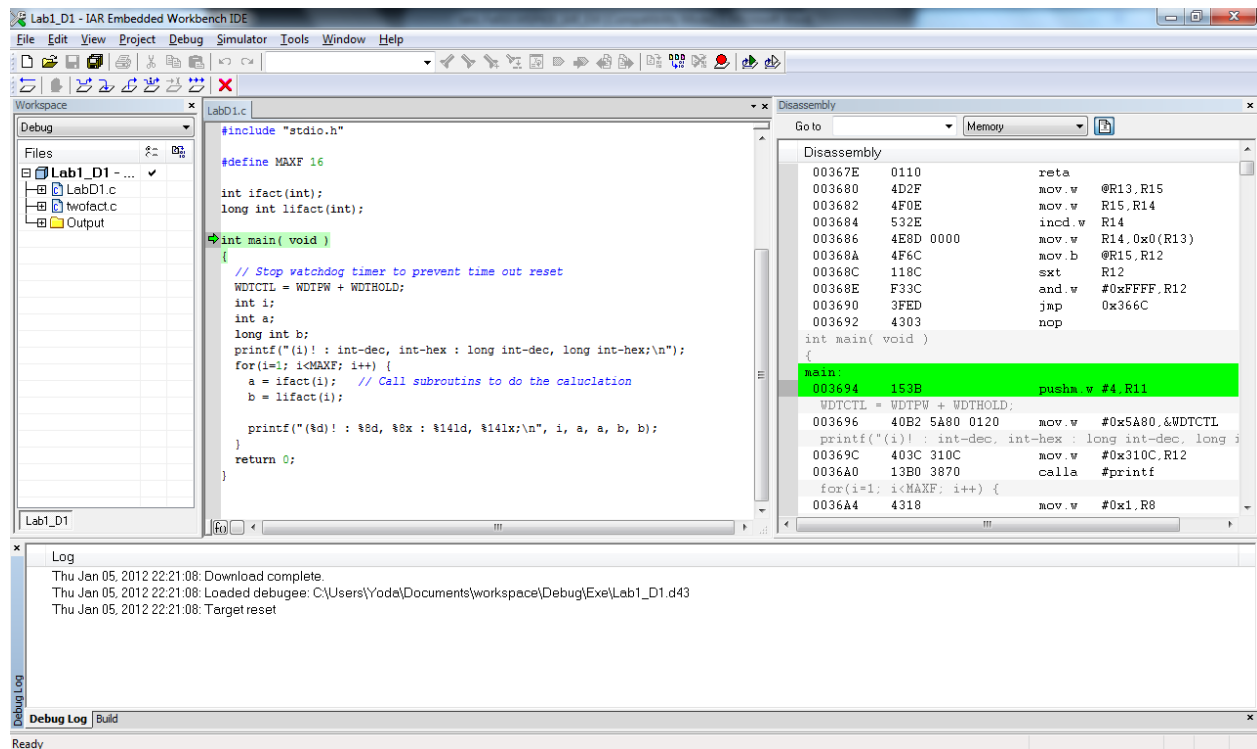




Figure 16. Debug Log window.

If you want to start again with the existing application, choose *Debug>Reset*, or click the *Reset* button on the toolbar. 

To exit from C-SPY, choose *Debug>Stop Debugging*. Alternatively, click the *Stop Debugging* button on the toolbar. The Embedded Workbench workspace is displayed. 

1.3. Software Documentation

Maintaining good software documentation is of key importance, especially wherever your code may be used by others. In this lab, a portion of your grade will be decided based on the documentation you provide with your code, particularly your header and comments, code formatting, and software diagrams or flowcharts.

1.3.1. Code formatting and organization

Part of good coding is ensuring that the code is neatly formatted and organized. This not only helps others who may need to access your code, but it also helps in debugging and maintaining your own code. Here are some general guidelines to assist in organizing your code in the IAR Workbench IDE:

- Ensure all your function prototypes are declared after your `#include` statements at the top of the program.
- Declare all of your global variables directly after your function prototypes in one area
- Consistently organize your functions. A good way to organize functions is to have your main function, followed by the other functions in order of call, followed by your

interrupt functions. You can choose a different way, but make sure to organize the types of functions and maintain consistency.

- Keep track of your indention. In the IAR workbench IDE, a tab is two spaces. Each function, loop, or other “nest” should be indented appropriately and consistently
- It is especially important to keep track of where your code is located on your workstation. IAR uses workspaces to help you keep track of your code. It is recommended that you create a directory with your name, and then create subdirectories for each lab assignment. In each of those directories, you should create a new workspace and a new project for every assignment.

1.3.2. Code headers and comments

By now, you’ve become familiarized with commenting in your programs. Comments help you and others keep up with the flow of the code, and it is important to maintain good comments. In this lab, you will be programming in C and assembly code. In assembly code, you generally should comment every line of code to explain its purpose. The reason for this is that the code is much less self-explanatory than common coding languages. In C, there are generally a few guidelines to remember when commenting:

- You should always include a header at the top of your code that gives basic information about you, when the code was written, and what it does.
- Each variable declaration should be commented.
- Each function declaration should be properly noted.
- Any segment of code, whether it is to initialize hardware, perform a calculation, or do another task, should have concise comments that explain it.

1.3.3. Software flowcharts

In this lab, you must provide a flowchart for each programming assignment you submit. A flowchart is a helpful way for you to decide on an approach to your program *before* you begin. It is also an extremely effective way of concisely relaying how your code works to others.

A flowchart does not contain information about every line of code, but it is a slightly higher-level picture that shows logically how problems are addressed. Hardware initializations and variable declarations should be documented. Also, any logical steps, function calls, or loops should be noted as well as their respective conditions. Below is the code for a program that uses a recursive function call to calculate a power.

```
/*-----  
* Name: Micah Harvey  
* Lab assignment: 1  
* Lab Section: 1 & 2  
* Date: May 20, 2013  
* Instructor: Micah Harvey
```

```

* Program: This program outputs a message to the terminal 10
* times, and counts up or down depending on a test variable.
*-----*/

#include "msp430.h"
#include "stdio.h"

//function prototypes
long int exponential(int, int);

int main(void)
{
    WDTCTL = WDTPW + WDTHOLD;      //stop the watchdog timer

    int value = 12;                //the base
    int power = 4;                 //the exponent, must be positive and 1 or greater
    long int answer = exponential(value, power); //find the power

    printf("%d raised to the %d power is %ld", value, power, answer);

    return 0;
}

//This function recursively calculates a power given a base and a power
long int exponential(int value, int power)
{
    int answer;                    //the solution
    if (power > 1)                  //if the current call has power > 1 then call again
    {
        power--;
        answer = value*exponential(value, power);
    }

    else                           //else the power = 1 and the answer is the base
    {
        answer = value;
    }

    return answer;                 //return the answer
}

```

A corresponding flowchart for the program can be seen below. Note that it doesn't include every line of code, but it does note the recursive portion of the program and the conditional statements associated with it.

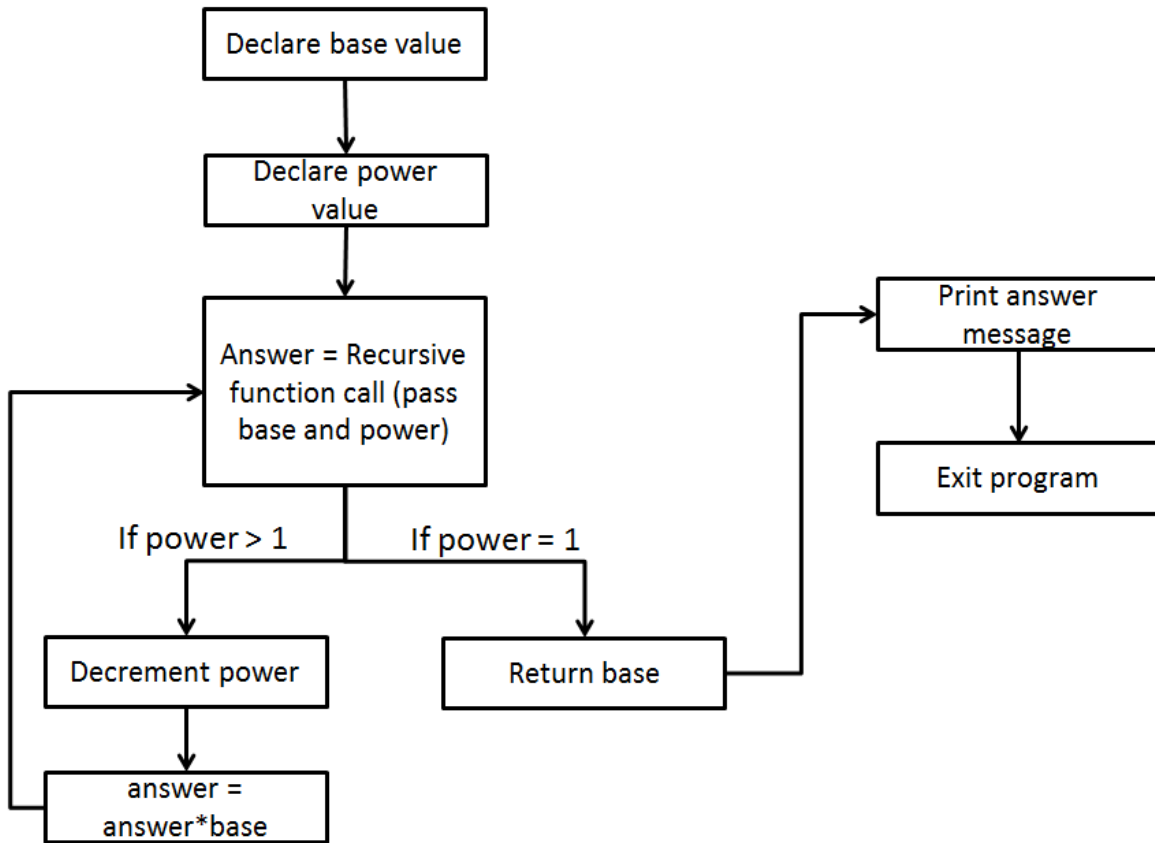


Figure 17. Flow chart for the recursive power calculating program.

1.4. Assignments

1. On ANGEL, you will find a piece of code that prints a message in the terminal window. You must first open the code in notepad and create a flowchart for the code. When you have completed that step, you must create an organized directory, subdirectory, workspace, and project for the code (your name/lab1). Once organized, you must properly compile and view the output. Make sure to get an instructor signoff when you've completed it.