ESE 381 Embedded Microprocessor Systems Design II

Spring 2016, K. Short

revised January 22, 2016 5:33 pm

MODULE 0: ESE 381 Design Environment

To be performed the week starting January 31st.

Prerequisite Reading

(All items are posted on Blackboard.)

1. ATMega128A Data Sheet, pages 1 through 18
2. ET-AVR Stamp ATMega128 Schematic
3. Resistor Network CTS SIP
4. BarGraph10_Element
5. Getting Started with IAR Embedded Workbench
6. inout.c code
7. sws_lvl.c code

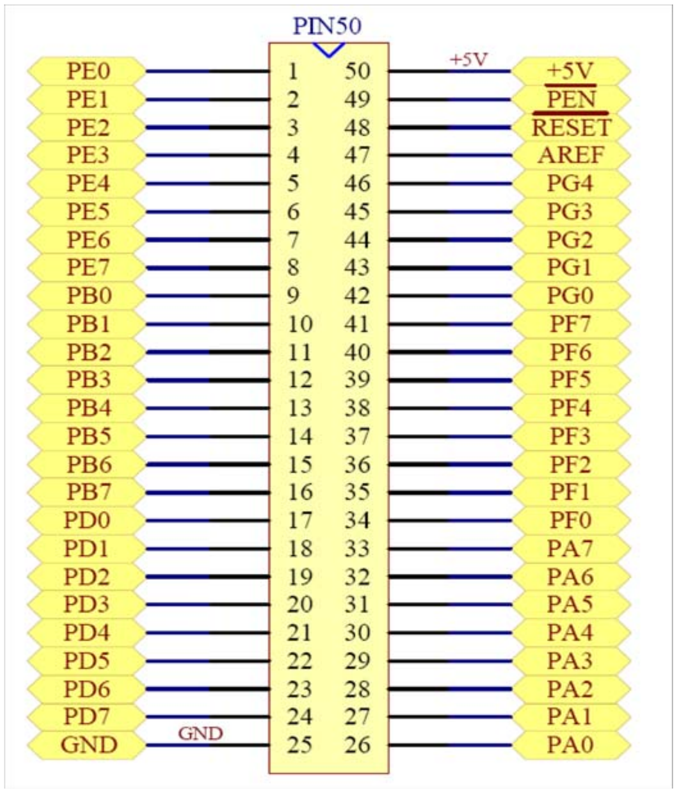
Overview

The purpose of this module is to acquaint you with the design environment used in ESE 381. That is, the basic prototyping hardware and software development tools.

Microcontroller and JTAG Interface

We use the Atmel ATmega128 microcontroller. It has the same basic AVR architecture as the ATmega16 used in ESE 380. The ATmega128's instruction set is the same as for the ATmega16. However, the ATmega128 has more memory and additional peripherals and other features. Most importantly to us, initially, is that it has additional ports: Port E, Port F, and Port G.

The ATmega128 the we use is packaged in a 64 pin thin quad flat package (TQFP). This package is designed for surface mount, so it is not directly compatible with the prototype boards in the laboratory. To solve this problem, we use the ET-AVR Stamp ATmega128 board from Futurlec. We will refer to this simply as the ATMega128 board. Except for PortC, the ATmega128 IC's port pins are brought out to pins of the ATMega128 board. The schematic shows to what board pins the ATmega128 IC's port pins are connected. The pins of PortC are available on 10-pin header on the board.



AVR IAR Embedded Workbench IDE Software Tools
Assembly language software development for the ATmega128 could be accomplished using Atmel's Studio 6 IDE, used in ESE 380 for the ATmega16. However, in this course programming is done in both embedded C and assembler. We could use the Studio 6 IDE that supports both assembler and C. However, instead we will use the AVR IAR Embedded Workbench IDE. This is a C/C++ and assembler development environment from IAR Systems. There are versions of this IDE that target code to a wide range of different manufacturers' microcontrollers. The version we will use targets code to the AVR microcontrollers.

We use the full version of this IDE in the Embedded Systems Design Laboratory (ESDL). There is also a free version called the KickStart Version. You should download this version to your own PC from:

http://supp.iar.com/Download/SW/?item=EWAVR-KS4

The KickStart version has a code size limitation of 4K, but is useful for software development outside of the ECE department laboratories.

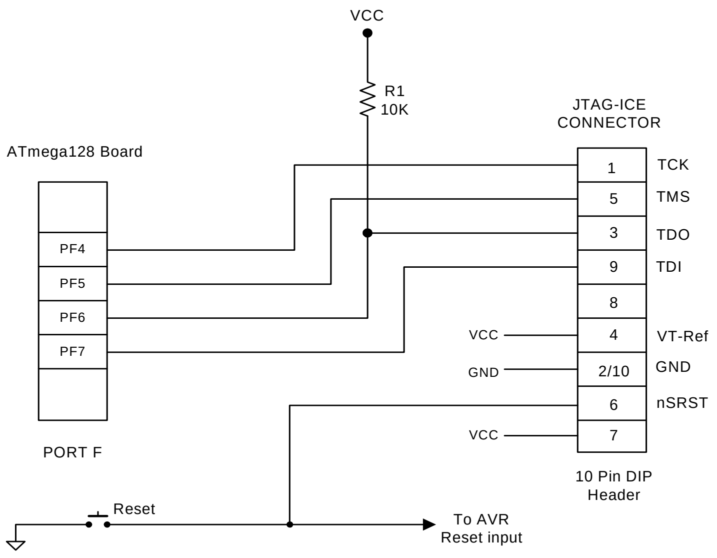
Design Task
Since the primary purpose of this module is to introduce you to the design environment, the applications chosen are simple switch input and LED output designs similar to those used in Laboratories 1 and 2 of ESE 380. The differences are that you use the ATmega128 board and you develop and debug the software in C, using the IAR Embedded Workbench IDE.

You must write the programs for Tasks 4 and 5. See the next section for a description of these programs. The other programs were discussed in the class lectures and are available on Blackboard.

Laboratory Activity

Task 1: Circuit Wiring
Using the ET-AVR Stamp ATmega128 schematic, wire the ground and power connections to the ATMega128 board.

Following the schematic below, wire the connections for the JTAG header to the ATMega128 board. Do not apply power.



Next wire an 8 position DIP switch to provide inputs to Port D. Use the ATmega128's internal pull-up resistors with these switches so that external pull-up resistors are not needed. Wire 8 of the LEDs in the bargraph LED to Port B. Use a SIP with 330 ohm resistors (Bussed CTS Schematic 1) for current limiting. Wire the LEDs so that a logic 0 at a PortB pin turns ON the associated LED. Connect the JTAGICE to the JTAG header on the breadboard. Make absolutely sure you have the connectors properly aligned. Have a TA verify that your wiring is correct before applying power.

This hardware is used in all of the remaining tasks in this module.

Task 2: inout.c
Follow the instructions, in the tutorial at the end of this document, to use IAR Embedded Workbench to compile and debug the program the `inout.c` and execute it using JTAGICE. This program is provided on Blackboard. Single step through the program to verify that it operates properly. Then, run the program full speed. When you have your system working properly, have a TA verify its operation.

Task 3: sws_lvl.c
Following the procedures outlined in the tutorial, create a new project named `sws_lvl` in the same workspace (`mod0`). The program `sws_lvl.c` is also provided on Blackboard. Build this program and single step through it to verify proper operation. Then, run the program at full speed. When you have your system working properly, have a TA verify its operation.

Task 4: sws_and.c
Write a program named `sws_and.c` that performs the logical AND of bits PD7-PD4 with bits PD3-PD0, respectively, and displays the result on the LEDs connected to bits PB3-PB0. A logic 1 in the AND result must be indicated by the associated LED being ON. A logic 0 in the result must be indicated by the associated LED being OFF. The LEDs connected to bits PB7-PB4 must all remain OFF.

Create a new project named `mod0_AND`. Type in your program `sws_and.c`. Build this program and single step through it to verify that it operates properly. If there are errors, debug and correct them. Next, run the program at full speed. When you have your system working properly, have a TA verify its operation.

You must submit your `sws_and.c` program listing as part of your prelab. Prelabs must be submitted electronically as was done in ESE380. See the *ESE-38X Prel-lab Assignment Submission Procedure* document on Blackboard.

Task 5: sws_alu.c
Write a program named `sws_alu.c` that performs the logical operation specified by bits PD7-PD6 on two 3-bit operands. The 3-bit operands are input on bits PD5-PD3 and PD2-PD0. The result is displayed on the LEDs connected to bits PB2-PB0. A logic 1 in the result must be indicated by the associated LED being ON. A logic 0 in the result must be indicated by the associated LED being OFF. The LEDs connected to bits PB7-PB3 must all remain OFF.

The four logical operations specified by bits PD7-PD6 are:

PD7-PD6	Operation
00	AND
01	OR
10	EXOR
11	Complement of PD2-PD0

You may wish to use a C switch statement in your program to implement this functionality.

Create a new project named `mod0_ALU`. Type in your program `sws_alu.c`. Build this program and single step through it to verify that it operates properly. If there are errors, debug and correct them. Then, run the program at full speed. When you have your system working properly, have a TA verify its operation.

You must include your `sws_alu.c` program listing as part of your prelab.

Debug Capabilities of IAR Embedded Workbench's C-SPY

The ability to debug a program is critical to locating and correcting program errors. Use of debugging techniques is also an excellent way to develop a better understanding of the semantics of the constructs in a programming language. To encourage you to improve your debugging skills we will require that each student be able to individually explain the purpose of and demonstrate the execution of particular C-SPY simulator and debugger commands at the end of each laboratory module. You should be able to demonstrate these commands with C-SPY in the simulator mode or JTAGICE mode.

The commands below are used to execute a program. They can be accessed from the Debug toolbar or from the Debug Menu. You can use the help menu to find an explanation of each command.

The commands to start and end a debugging session are:

Make
Down Load and Debug
Stop Debugging

The primary commands used to execute statements in a program are:

reset
step into
next statement
run to cursor
go
break

Two commands that are on the Debug toolbar that we will not consider until later are step over and step out. These commands are useful later when dealing with functions.

You must also be able to explain the purpose of and demonstrate the execution of the breakpoint commands:

set a breakpoint
toggle a breakpoint

When you single step a simple program, you want to be able to observe the values of variables used by the program. With the simple programs of this first module the variables are primarily port registers and general purpose registers. From the View menu you can select Registers to open a register window. You then have a drop down box that allows you display one of many single registers (including ports) or groups of registers in the ATmega128's architecture.

You will be asked to step through one of the programs and demonstrate that the registers have the expected values after the execution of each instruction. We will encounter many more debugging commands and techniques as the semester progresses.

Leave the hardware that you have wired on the breadboard intact. This hardware may be used again in later laboratories.

Getting Started with the IAR Embedded Workbench IDE for Atmel AVR

Introduction

The IAR Embedded Workbench IDE is a powerful integrated development environment for developing and debugging C/C++ and assembly software for Atmel AVR microcontrollers. The purpose of this tutorial is to guide new users through the compilation and debugging of a simple C-program using the IAR Embedded Workbench IDE. This tutorial shows you how to setup a new project and generate a debug target that can be loaded into C-SPY. C-SPY is the high-level language debugger used for debugging and to download code into the microcontroller's flash program memory via JTAGICE. A simple in/out program written for the Atmega128 is used as an example.

Workspaces and Projects in the IAR Embedded Workbench IDE

In the IAR Embedded Workbench IDE, source files for an application are contained in a project. A project must, in turn, be contained in a workspace. Each workspace can contain one or more projects. Normally, projects in a workspace are related in that they share some files and/or have related target hardware.

How to Open a New Workspace and Project in IAR Embedded Workbench

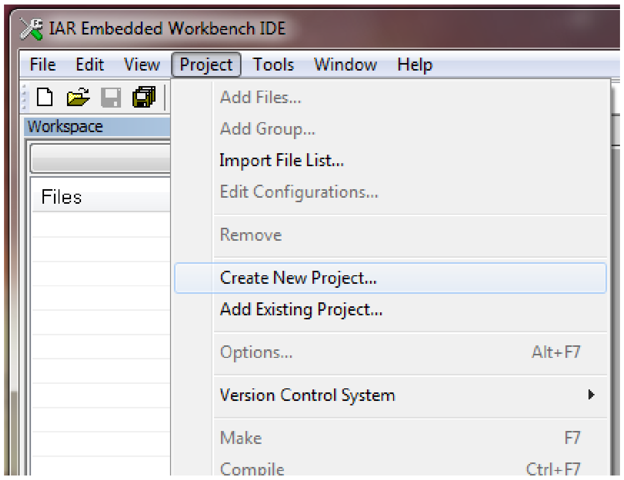
The sample application code `in_out.c` is available on Blackboard. Using Windows Explorer create a folder named "H:\381\mod0\IAR\" and save the sample code in that folder.

Open IAR Embedded Workbench by double clicking its icon on your desktop.

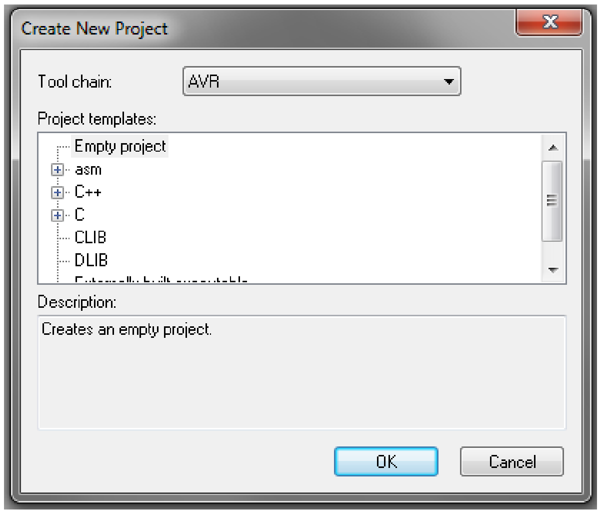
IAR Embedded Workbench

Use the steps that follow to make your first project.

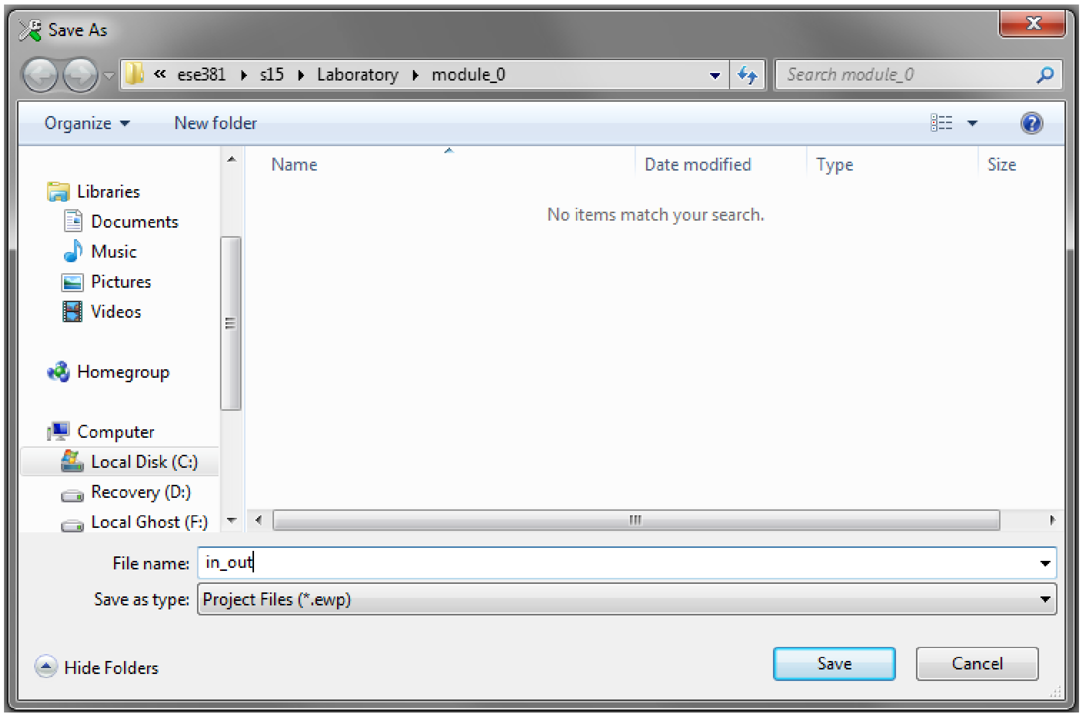
1. From the Project tab, click on **Create New Project**.



2. Verify that **Tool chain AVR** is selected, then click **OK**.



3. Save the project in the folder “H:\381\mod0\IAR\” as `in_out.ewp`

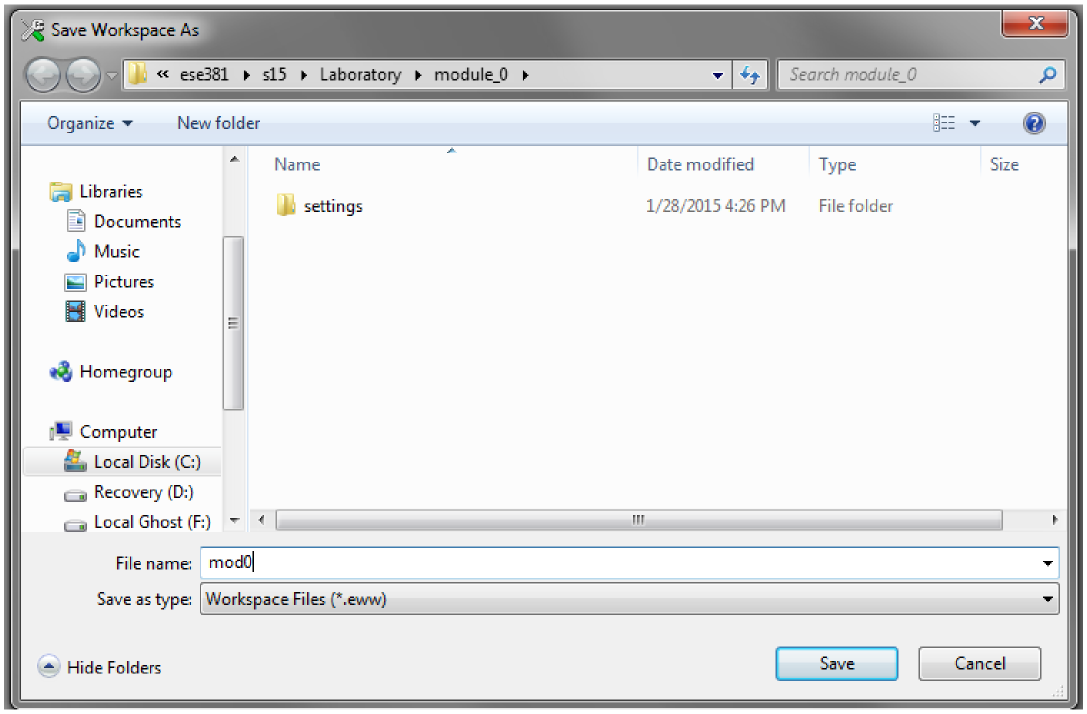


No items match your search.

File name: in_out

Save as type: Project Files (*.ewp)

4. Next you must save the current workspace. Choose **File>Save Workspace**. Save the workspace in folder “H:\381\mod0\IAR\” as `mod0.eww`.

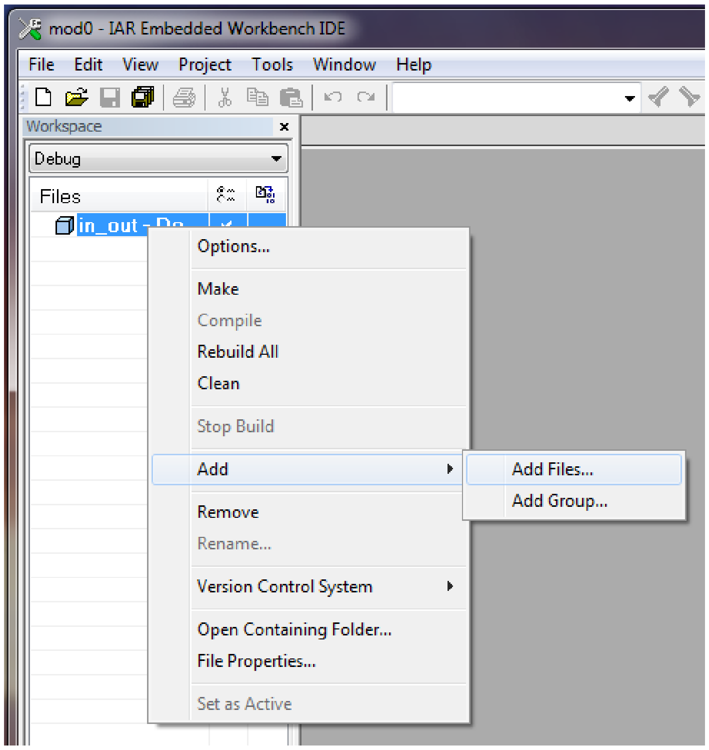


settings 1/28/2015 4:26 PM File folder

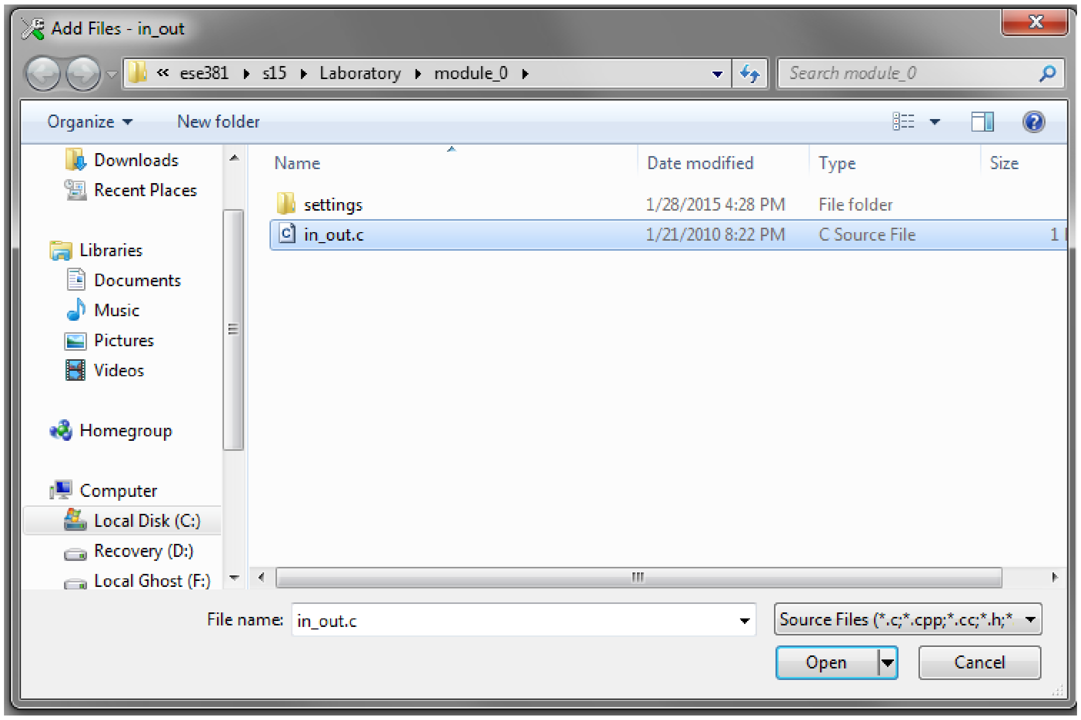
File name: mod0

Save as type: Workspace Files (*.eww)

5. Right click on “in_out-debug, choose **Add>Add Files**….



Select `in_out.c` and then click **Open**.

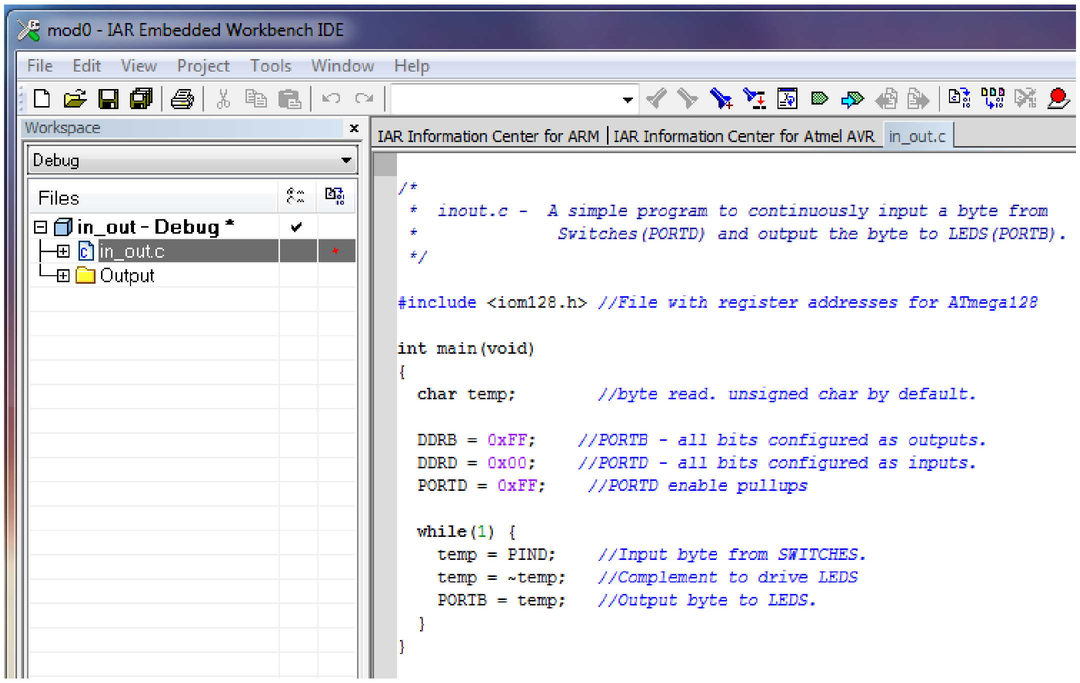


Name	Date modified	Type	Size
settings	1/28/2015 4:28 PM	File folder	
in_out.c	1/21/2010 8:22 PM	C Source File	1

File name: in_out.c

Source Files (*.c;*.cpp;*.cc;*.h;*

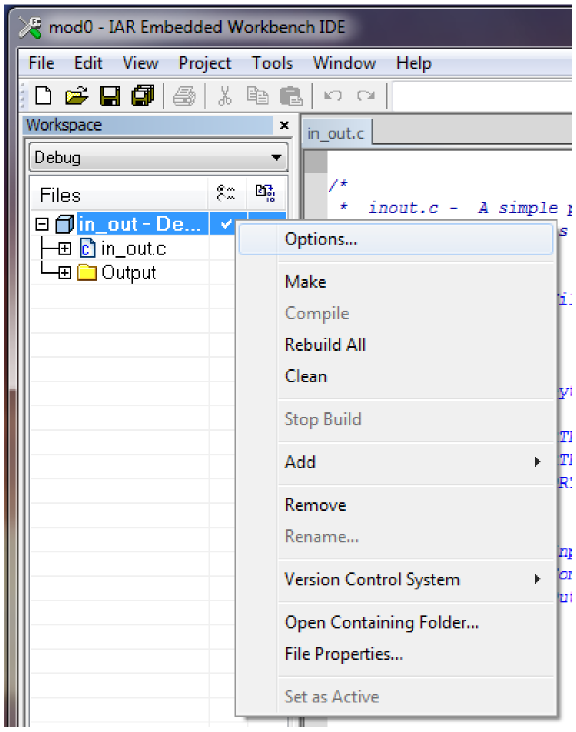
Double-click `in_out.c` in the workspace window to display the added file.


mod0 - IAR Embedded Workbench IDE
File Edit View Project Tools Window Help
Workspace
Debug
Files
in_out - Debug *
in_out.c
Output
IAR Information Center for ARM
IAR Information Center for Atmel AVR
in_out.c

```
/*
 *  inout.c -  A simple program to continuously input a byte from
 *             Switches(PORTD) and output the byte to LEDS(PORTB).
 */

#include <iom128.h> //File with register addresses for ATmega128

int main(void)
{
  char temp;        //byte read. unsigned char by default.

  DDRB = 0xFF;    //PORTB - all bits configured as outputs.
  DDRD = 0x00;    //PORTD - all bits configured as inputs.
  PORTD = 0xFF;    //PORTD enable pullups

  while(1) {
    temp = PIND;    //Input byte from SWITCHES.
    temp = ~temp;   //Complement to drive LEDS
    PORTB = temp;   //Output byte to LEDS.
  }
}
```

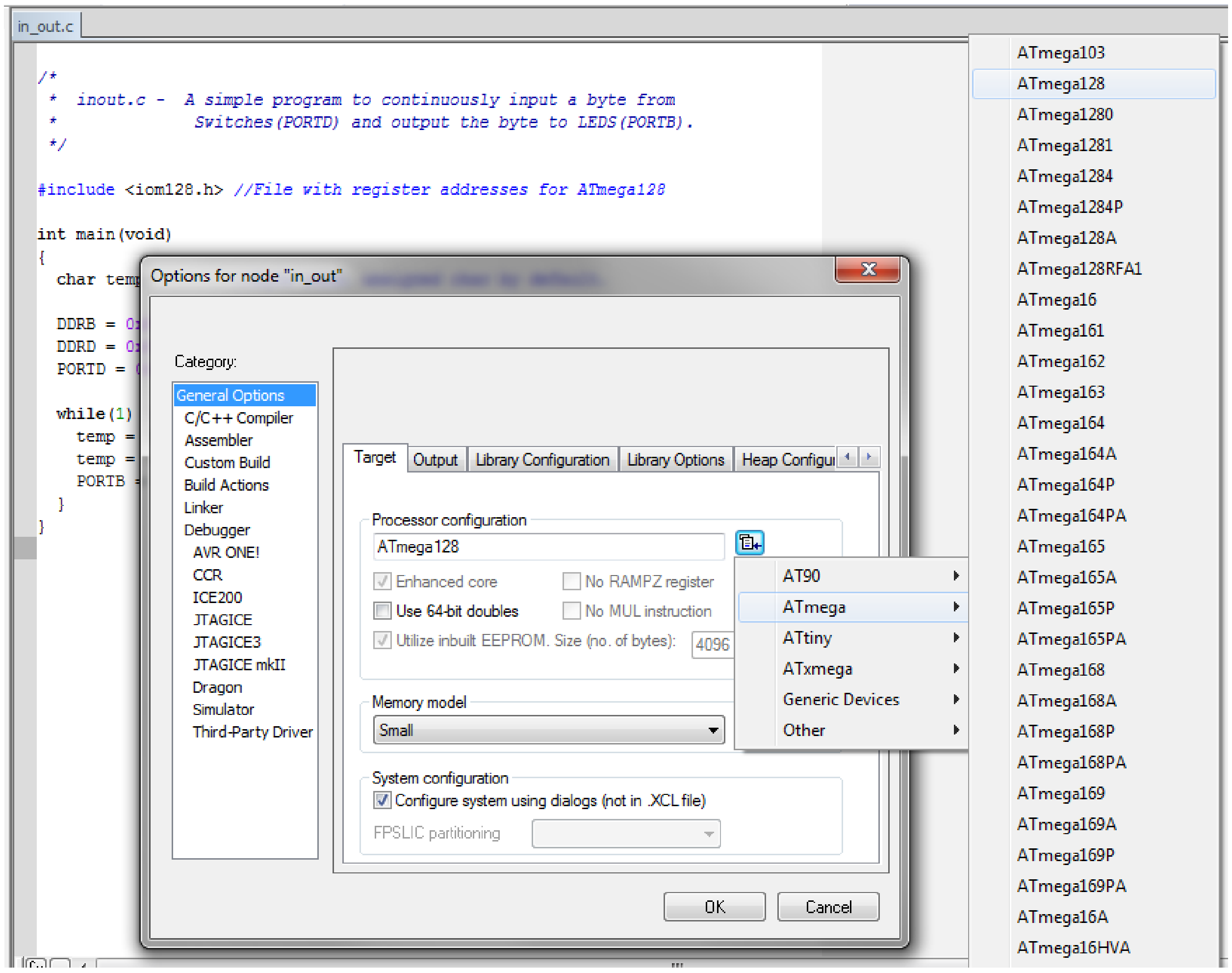
How to Set Up the Project Options

There are a number of options that must be selected for each project. Right click on the project "in_out - Debug" and select Options:



Make sure the correct target microcontroller is selected in the drop-down menu of the project box in your IDE.

1. In the **General Options** category, under the **Target** pane, select the processor configuration for ATmega128. This is done by clicking on the icon to the right of the processor configuration text box. Then select ATmega and the list of ATmega devices will be displayed where you can select ATmega128.
2. Choose memory model **Small**.


in_out.c

```
/*
 *  inout.c -  A simple program to continuously input a byte from
 *             Switches(PORTD) and output the byte to LEDS(PORTB).
 */

#include <iom128.h> //File with register addresses for ATmega128

int main(void)
{
  char temp

  DDRB = 0
  DDRD = 0
  PORTD = (

  while(1)
    temp =
    temp =
    PORTB =
  }
}
```

Options for node "in_out"

Category: General Options

Target

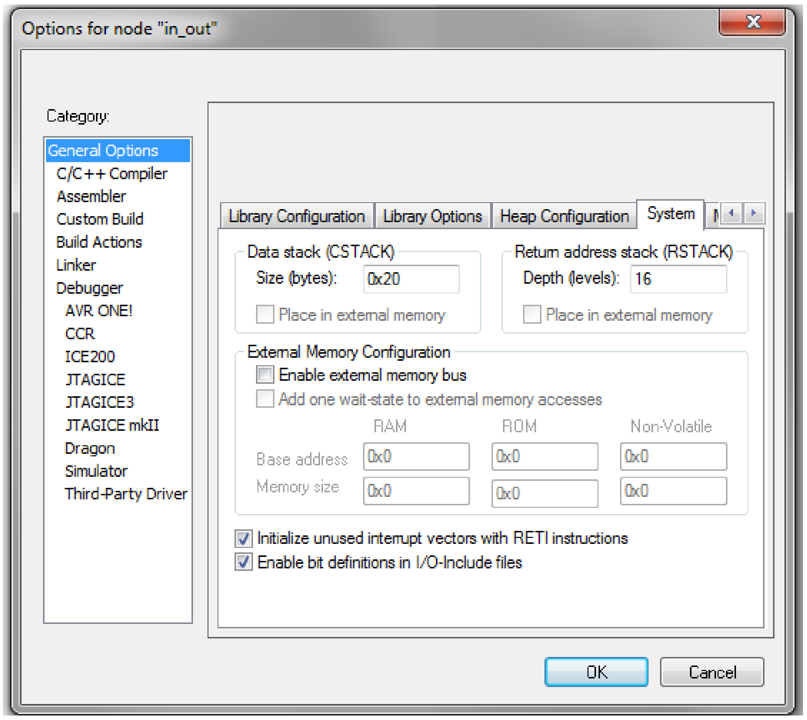
Processor configuration: ATmega128

[✓] Enhanced core [ ] No RAMPZ register
[ ] Use 64-bit doubles [ ] No MUL instruction
[✓] Utilize inbuilt EEPROM. Size (no. of bytes): 4096

Memory model: Small

System configuration
[✓] Configure system using dialogs (not in .XCL file)
FPSLIC partitioning

3. In the **General Options** category, under the **Options** pane, select **Enable bit definitions in I/O include files**.

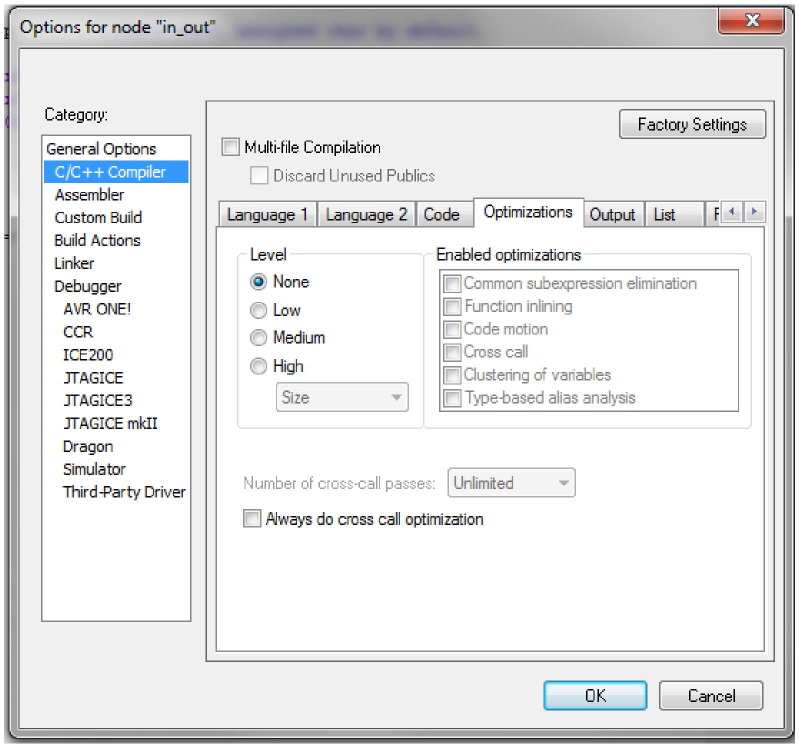


Data stack (CSTACK)
Size (bytes): 0x20
Place in external memory
Return address stack (RSTACK)
Depth (levels): 16
Place in external memory
External Memory Configuration
Enable external memory bus
Add one wait-state to external memory accesses

	RAM	ROM	Non-Volatile
Base address	0x0	0x0	0x0
Memory size	0x0	0x0	0x0

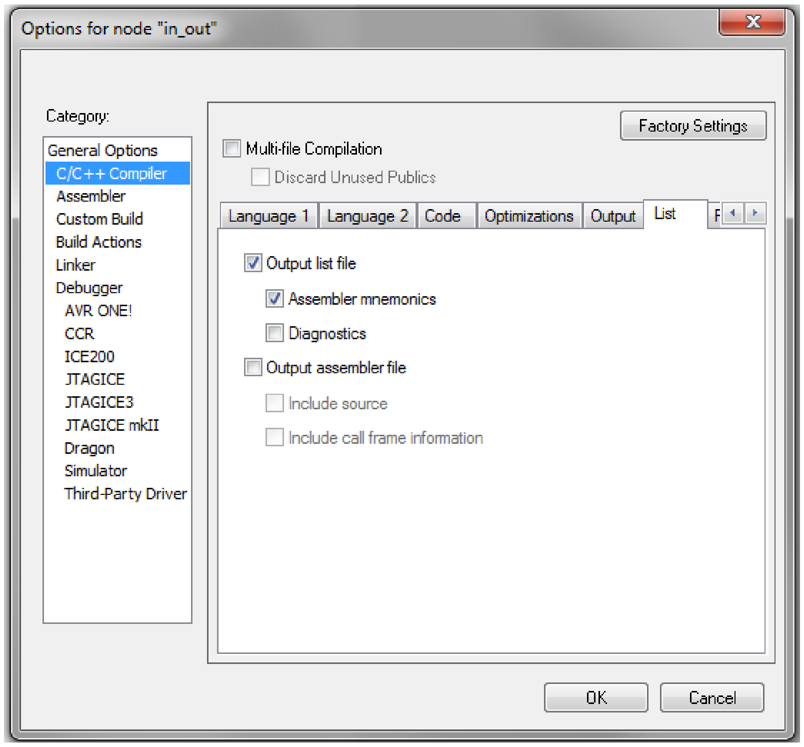
[✓] Initialize unused interrupt vectors with RETI instructions
[✓] Enable bit definitions in I/O-Include files

4. In the **C/C++ compiler** category, under the **Optimizations** pane select **None (Best debug support).**



Multi-file Compilation
Discard Unused Publics
Level
[✓ None] Low Medium High
Size
Enabled optimizations
Common subexpression elimination
Function inlining
Code motion
Cross call
Clustering of variables
Type-based alias analysis
Number of cross-call passes: Unlimited
Always do cross call optimization

5. In the **C/C++ compiler** category, under the **List** pane check to enable **Output list files** and **Assembler mnemonics**.


Options for node "in_out"

Category:
General Options
C/C++ Compiler
Assembler
Custom Build
Build Actions
Linker
Debugger
AVR ONE!
CCR
ICE200
JTAGICE
JTAGICE3
JTAGICE mkII
Dragon
Simulator
Third-Party Driver

Factory Settings

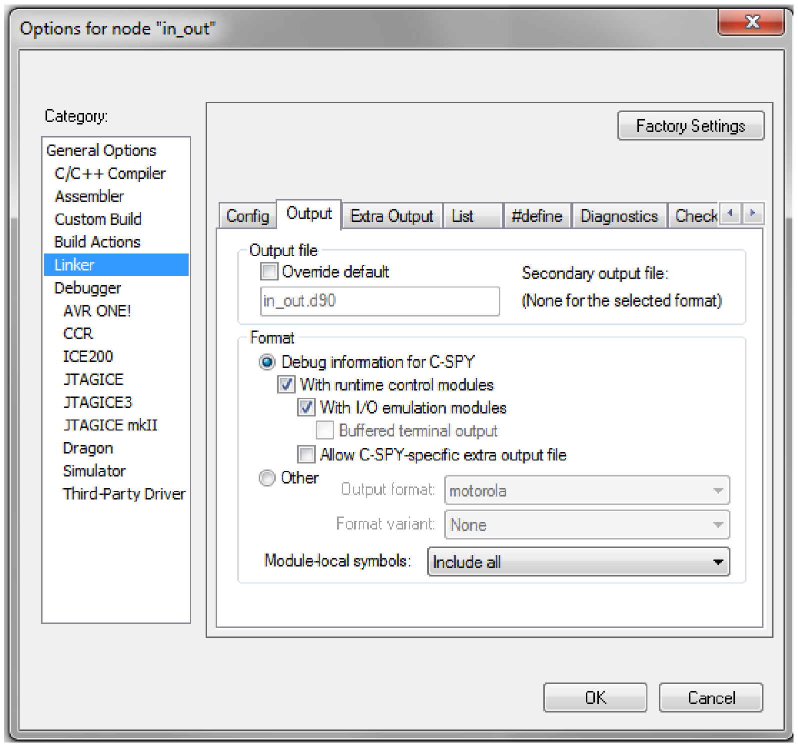
[ ] Multi-file Compilation
[ ] Discard Unused Publics

Language 1 | Language 2 | Code | Optimizations | Output | List | F

[✓] Output list file
[✓] Assembler mnemonics
[ ] Diagnostics
[ ] Output assembler file
[ ] Include source
[ ] Include call frame information

OK | Cancel

In the **Linker** Category, under the **Output** pane verify that **Debug information for C-SPY** is selected and check **With runtime control modules** and **With I/O emulation modules**.


Options for node "in_out"

Category:
General Options
C/C++ Compiler
Assembler
Custom Build
Build Actions
Linker
Debugger
AVR ONE!
CCR
ICE200
JTAGICE
JTAGICE3
JTAGICE mkII
Dragon
Simulator
Third-Party Driver

Factory Settings

Config | Output | Extra Output | List | #define | Diagnostics | Check

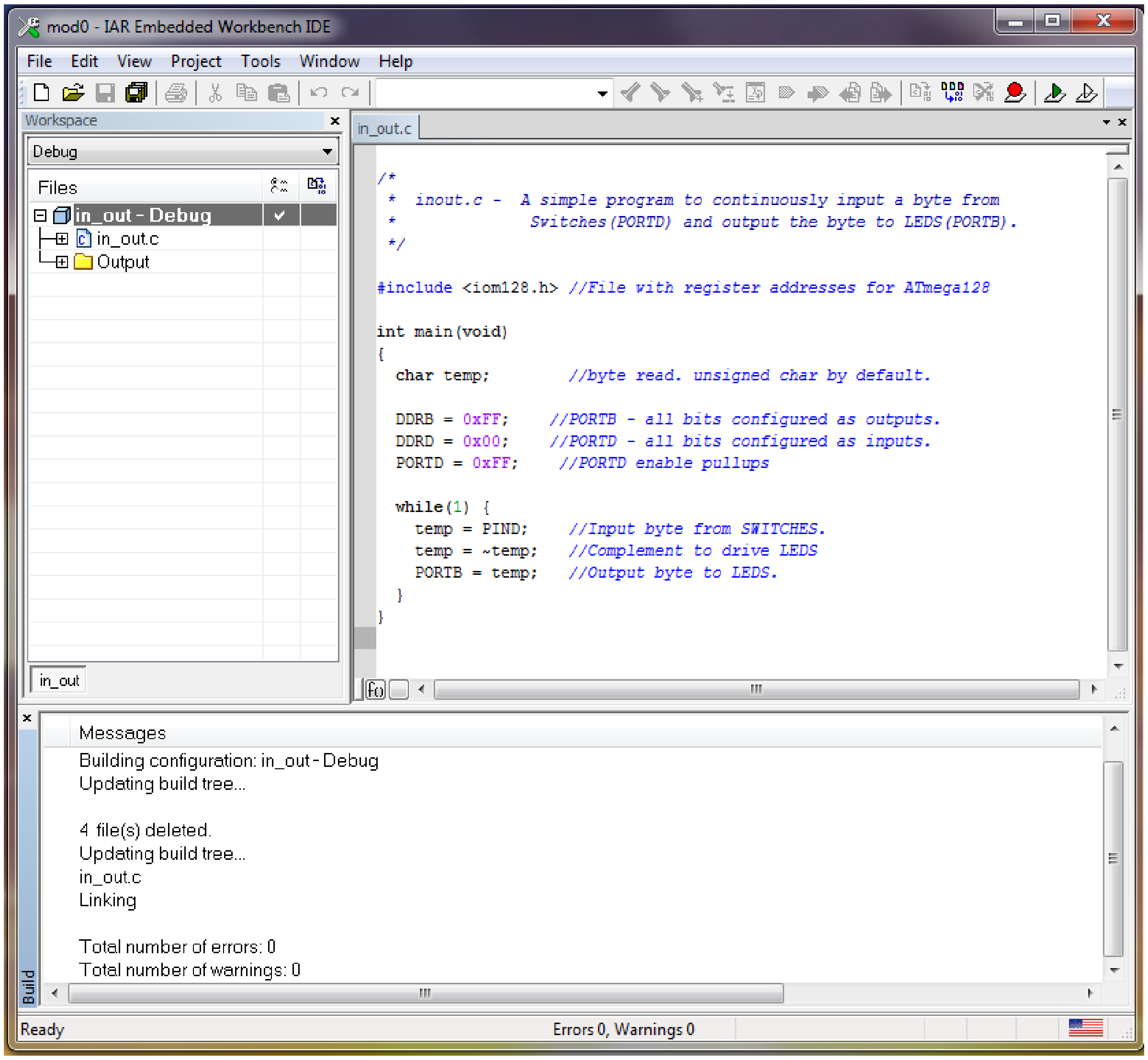
Output file
[ ] Override default
in_out.d90
Secondary output file:
(None for the selected format)

Format
(●) Debug information for C-SPY
[✓] With runtime control modules
[✓] With I/O emulation modules
[ ] Buffered terminal output
[ ] Allow C-SPY-specific extra output file
( ) Other
Output format: motorola
Format variant: None
Module-local symbols: Include all

OK | Cancel

Now click **OK**. All the options are set and you are ready to compile and link.

To compile and link, choose **Project>Make** or hit F7. The program is compiled and linked.

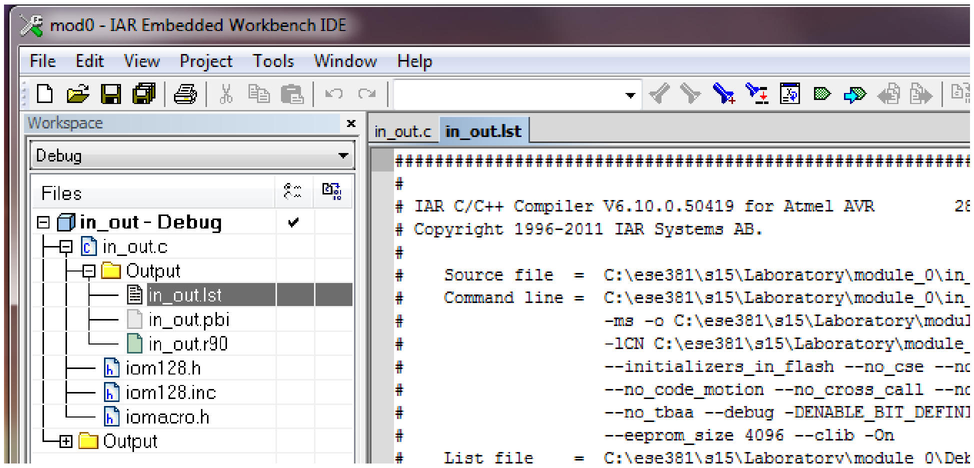

mod0 - IAR Embedded Workbench IDE
File Edit View Project Tools Window Help
Workspace
Debug
Files
in_out - Debug
in_out.c
Output
in_out
in_out.c

```
/*
 *  inout.c -  A simple program to continuously input a byte from
 *             Switches(PORTD) and output the byte to LEDS(PORTB).
 */

#include <iom128.h> //File with register addresses for ATmega128

int main(void)
{
  char temp;         //byte read. unsigned char by default.

  DDRB = 0xFF;     //PORTB - all bits configured as outputs.
  DDRD = 0x00;     //PORTD - all bits configured as inputs.
  PORTD = 0xFF;     //PORTD enable pullups

  while(1) {
    temp = PIND;     //Input byte from SWITCHES.
    temp = ~temp;    //Complement to drive LEDS
    PORTB = temp;    //Output byte to LEDS.
  }
}
```

```
Messages
Building configuration: in_out - Debug
Updating build tree...

4 file(s) deleted.
Updating build tree...
in_out.c
Linking

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

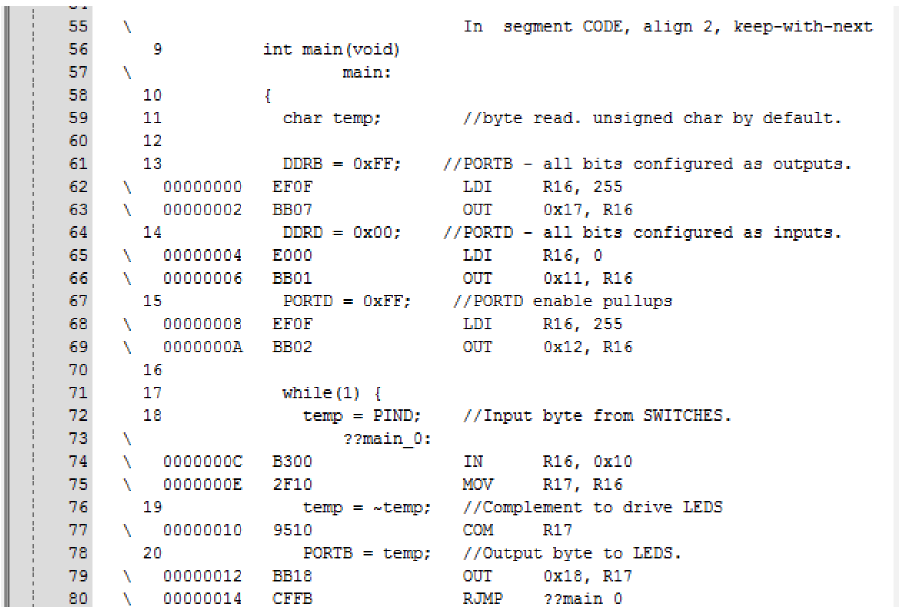
Now to take a look at the listing file:

1. Expand `in_out.c` in the Workspace pane by clicking the + sign on its left. Next expand the output folder by clicking the + sign on its left.



```
##############################################################
#
# IAR C/C++ Compiler V6.10.0.50419 for Atmel AVR       28
# Copyright 1996-2011 IAR Systems AB.
#
#    Source file  =  C:\ese381\s15\Laboratory\module_0\in_
#    Command line =  C:\ese381\s15\Laboratory\module_0\in_
#                    -ms -o C:\ese381\s15\Laboratory\modul
#                    -lCN C:\ese381\s15\Laboratory\module_
#                    --initializers_in_flash --no_cse --nc
#                    --no_code_motion --no_cross_call --nc
#                    --no_tbaa --debug -DENABLE_BIT_DEFINI
#                    --eeprom_size 4096 --clib -On
#    List file    =  C:\ese381\s15\Laboratory\module_0\Deb
```

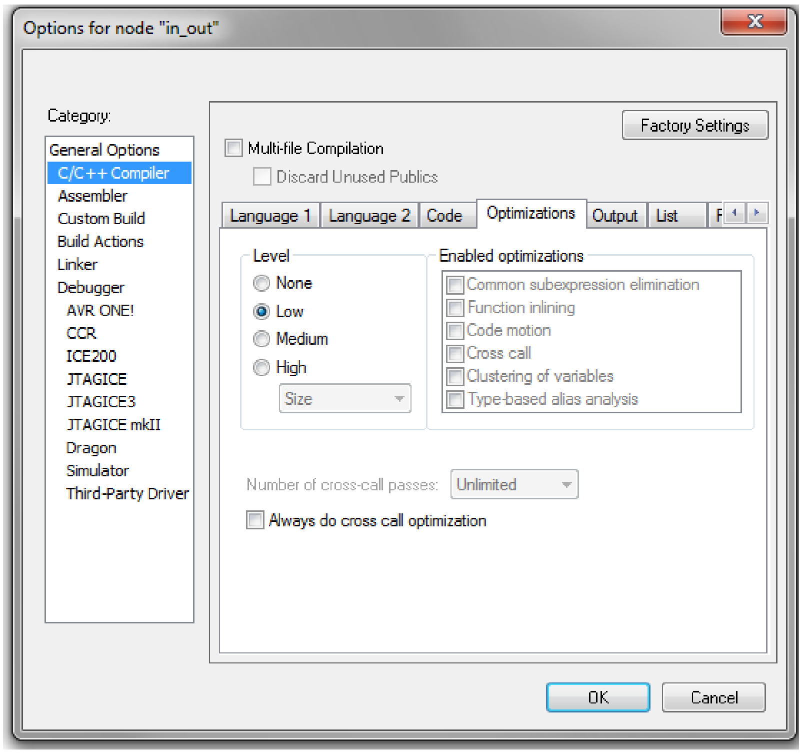
2. Double-click `in_out.lst`.
3. Carefully review the listing file paying particular attention to the part shown below.



```
  \                                 In  segment CODE, align 2, keep-with-next
     9          int main(void)
  \                     main:
    10          {
    11            char temp;        //byte read. unsigned char by default.
    12
    13            DDRB = 0xFF;    //PORTB - all bits configured as outputs.
  \   00000000   EF0F               LDI     R16, 255
  \   00000002   BB07               OUT     0x17, R16
    14            DDRD = 0x00;    //PORTD - all bits configured as inputs.
  \   00000004   E000               LDI     R16, 0
  \   00000006   BB01               OUT     0x11, R16
    15            PORTD = 0xFF;    //PORTD enable pullups
  \   00000008   EF0F               LDI     R16, 255
  \   0000000A   BB02               OUT     0x12, R16
    16
    17            while(1) {
    18              temp = PIND;    //Input byte from SWITCHES.
  \                     ??main_0:
  \   0000000C   B300               IN      R16, 0x10
  \   0000000E   2F10               MOV     R17, R16
    19              temp = ~temp;   //Complement to drive LEDS
  \   00000010   9510               COM     R17
    20              PORTB = temp;   //Output byte to LEDS.
  \   00000012   BB18               OUT     0x18, R17
  \   00000014   CFFB               RJMP    ??main_0
```

Note that in the while loop, data is input from PIND to r16, moved to r17, complemented in r17 and then output to PORTB. You are probably thinking that you wrote tighter code for this in ESE 380, and so much for using C - right!

Go to **Project>Options>C/C++Compiler>Optimizations**, select **Low** and press **OK**.


Options for node "in_out"

Category: C/C++ Compiler

Multi-file Compilation [ ]

Discard Unused Publics [ ]

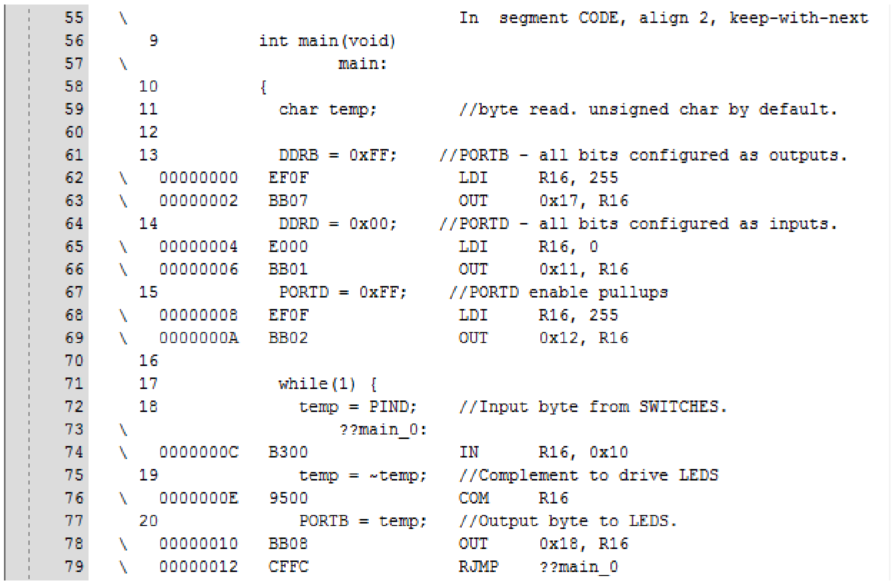
Level: None ( ) [✓ Low] Medium ( ) High ( )

Enabled optimizations:
- [ ] Common subexpression elimination
- [ ] Function inlining
- [ ] Code motion
- [ ] Cross call
- [ ] Clustering of variables
- [ ] Type-based alias analysis

Number of cross-call passes: Unlimited

Always do cross call optimization [ ]

Remake the project (**Project>Make**) and examine the new list file.



```
\                                   In  segment CODE, align 2, keep-with-next
   9          int main(void)
\                     main:
  10          {
  11            char temp;        //byte read. unsigned char by default.
  12
  13            DDRB = 0xFF;    //PORTB - all bits configured as outputs.
\   00000000  EF0F                  LDI     R16, 255
\   00000002  BB07                  OUT     0x17, R16
  14            DDRD = 0x00;    //PORTD - all bits configured as inputs.
\   00000004  E000                  LDI     R16, 0
\   00000006  BB01                  OUT     0x11, R16
  15            PORTD = 0xFF;    //PORTD enable pullups
\   00000008  EF0F                  LDI     R16, 255
\   0000000A  BB02                  OUT     0x12, R16
  16
  17            while(1) {
  18              temp = PIND;      //Input byte from SWITCHES.
\                     ??main_0:
\   0000000C  B300                  IN      R16, 0x10
  19              temp = ~temp;     //Complement to drive LEDS
\   0000000E  9500                  COM     R16
  20              PORTB = temp;     //Output byte to LEDS.
\   00000010  BB08                  OUT     0x18, R16
\   00000012  CFFC                  RJMP    ??main_0
```

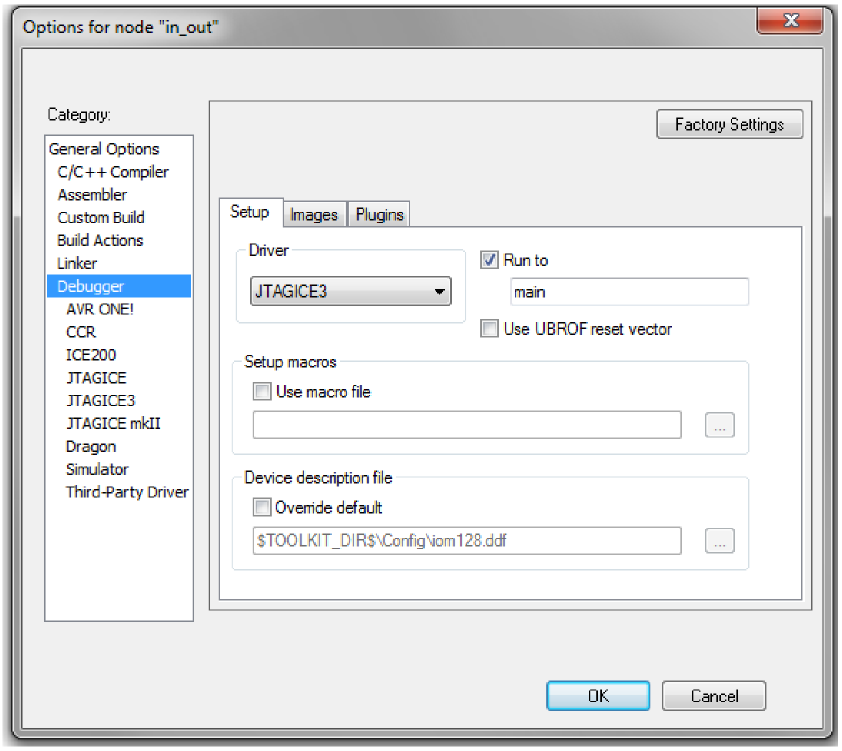
Well, the C compiler is not so dumb after all! It has optimized away the use of r17 in the while loop and is now as smart as you. During initial debugging, the optimization level should be set to

None. This is the best choice for debugging. Later when you have debugged your program and are creating a release version of your code you can increase the optimization to a higher level.

How to use C-SPY and JTAGICE3 to debug a Program

1. Connect the JTAGICE cable to your JTAG header. Apply power to the JTAGICE using the “strobe test” and to the CADET breadboard. Verify ICC is at a proper level.

2. Choose **Project>Options** and then the **Debugger** category. On the Setup page, make sure that you have chosen **JTAGICE3** from the **Driver** drop-down list and that **Run to main** is selected. Click **OK**. When working on your own PC using KickStart, the driver choice would be **Simulator**.


Options for node "in_out"

Category:
- General Options
  - C/C++ Compiler
  - Assembler
  - Custom Build
  - Build Actions
  - Linker
  - Debugger
    - AVR ONE!
    - CCR
    - ICE200
    - JTAGICE
    - JTAGICE3
    - JTAGICE mkII
    - Dragon
    - Simulator
    - Third-Party Driver

Factory Settings

Setup | Images | Plugins

Driver: JTAGICE3

[✓] Run to: main

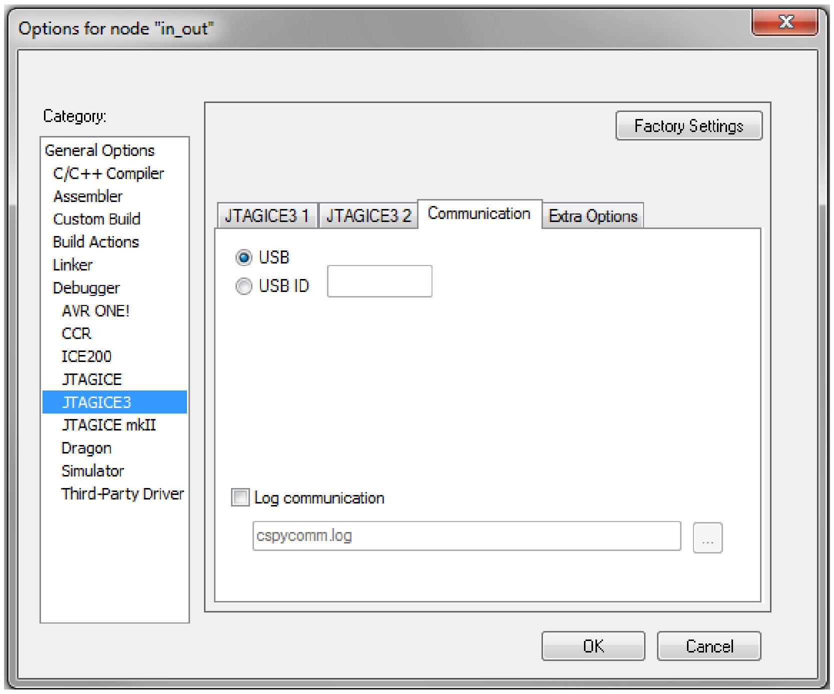
[ ] Use UBROF reset vector

Setup macros
[ ] Use macro file

Device description file
[ ] Override default
$TOOLKIT_DIR$\Config\iom128.ddf

OK Cancel

3. Choose **Project>Options** and then in the Category box choose **Debugger>JTAGICE3**. On the **JTAGICE3** Communication page, make sure that you have chosen the correct communications port (USB). Click **OK**.


Options for node "in_out"

Category:

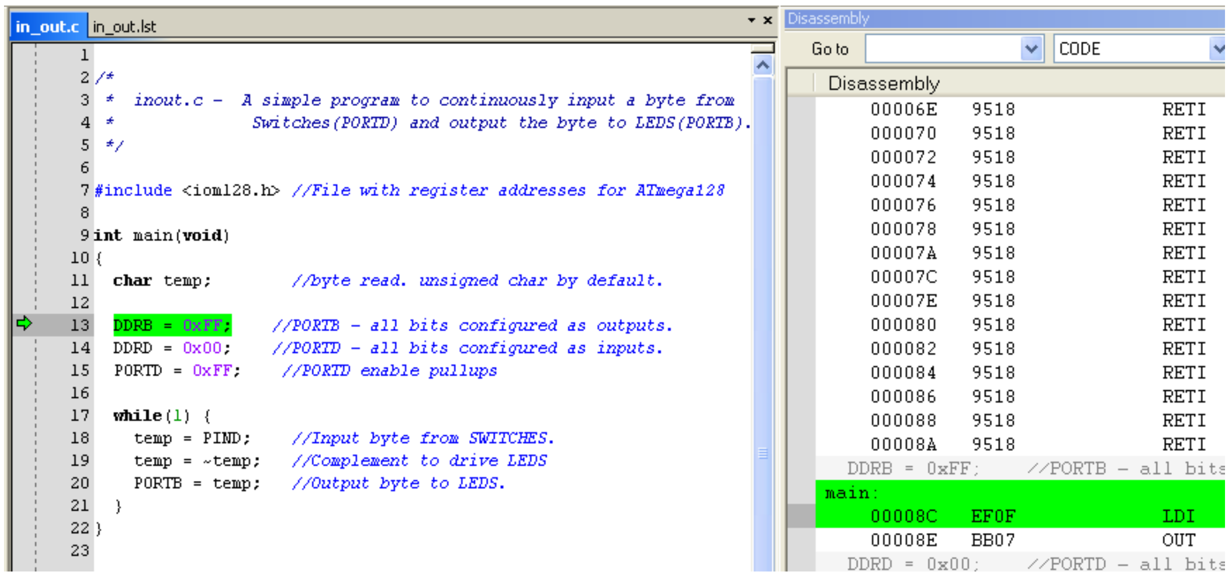
- General Options
  - C/C++ Compiler
  - Assembler
  - Custom Build
  - Build Actions
  - Linker
  - Debugger
    - AVR ONE!
    - CCR
    - ICE200
    - JTAGICE
    - JTAGICE3
    - JTAGICE mkII
    - Dragon
    - Simulator
    - Third-Party Driver

[✓ USB]
USB ID

Log communication [ ]
cspycomm.log

4. Next from the main menu choose **Project>Debug**. Alternatively, click the Download and Debug button in the toolbar (icon on right).

The IAR C-SPY Debugger starts with the application loaded. In addition to the windows already opened in the Embedded Workbench, a set of C-SPY-specific windows are now available.



```
/*
 *  inout.c -  A simple program to continuously input a byte from
 *             Switches(PORTD) and output the byte to LEDS(PORTB).
 */

#include <ioml28.h> //File with register addresses for ATmega128

int main(void)
{
  char temp;        //byte read. unsigned char by default.

  DDRB = 0xFF;     //PORTB - all bits configured as outputs.
  DDRD = 0x00;     //PORTD - all bits configured as inputs.
  PORTD = 0xFF;     //PORTD enable pullups

  while(1) {
    temp = PIND;    //Input byte from SWITCHES.
    temp = ~temp;   //Complement to drive LEDS
    PORTB = temp;   //Output byte to LEDS.
  }
}
```

```
Disassembly
     00006E    9518              RETI
     000070    9518              RETI
     000072    9518              RETI
     000074    9518              RETI
     000076    9518              RETI
     000078    9518              RETI
     00007A    9518              RETI
     00007C    9518              RETI
     00007E    9518              RETI
     000080    9518              RETI
     000082    9518              RETI
     000084    9518              RETI
     000086    9518              RETI
     000088    9518              RETI
     00008A    9518              RETI
  DDRB = 0xFF;     //PORTB - all bits
main:
     00008C    EF0F              LDI
     00008E    BB07              OUT
  DDRD = 0x00;     //PORTD - all bits
```

The source file window shows the first instruction in main() highlighted in green. The assembly language window shows the first of the corresponding assembly language instructions also highlighted in green.

5. With the source file window active, single step the C source statements using the **Debug>Step Into** command (F11). Single step through the entire program.

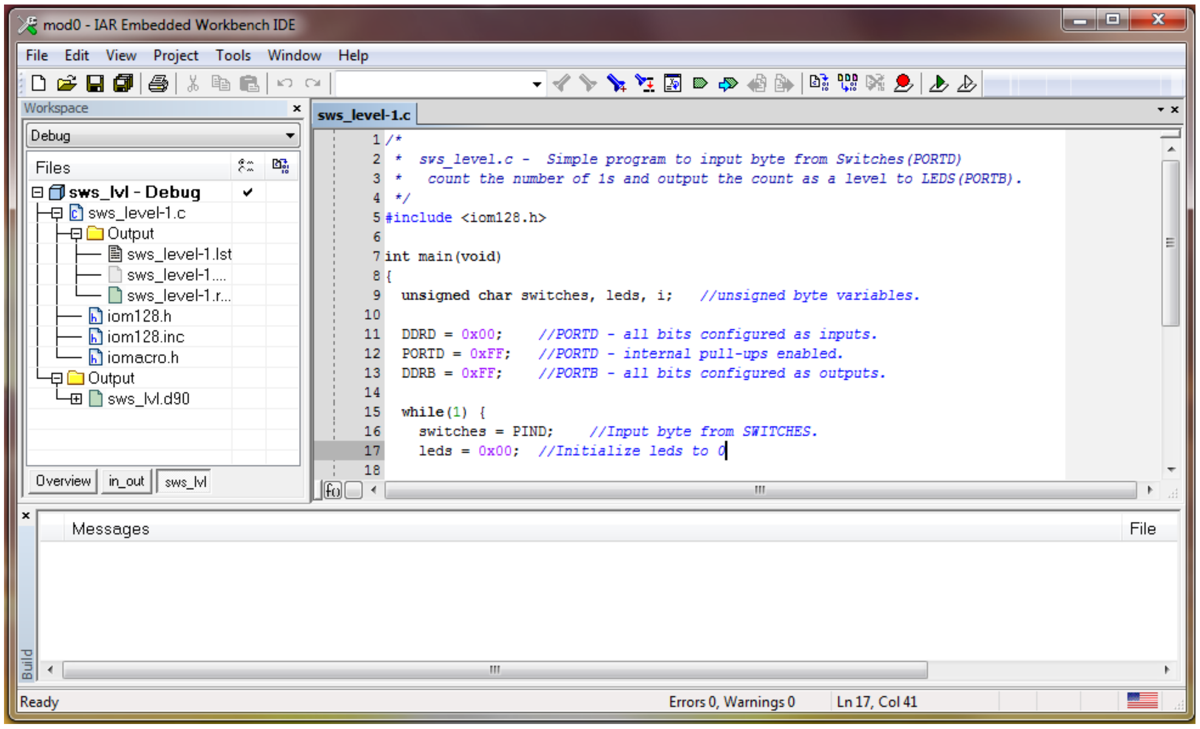
Running the Program Full Speed

To run the program full speed use the **Debug>Go** command. To break the execution of the program use the **Debug>Break** command.

Adding New Projects to the Workspace

To add a new project to the same workspace (as is required in Tasks 3, 4, and 5, of Mod0) choose **Project>Create New Project** and then follow the procedures previously discussed in this tutorial. ***You will have to set the options for each new project, even though it is in the same workspace.*** A new project does not inherit the options set for the previous project.

The following screen shot shows the workspace window after adding the project `sws_lvl`. Note the tabs that appear at the bottom of the workspace to select the different projects or the overview of the entire workspace.



```
/*
 *  sws_level.c -  Simple program to input byte from Switches(PORTD)
 *   count the number of 1s and output the count as a level to LEDS(PORTB).
 */
#include <iom128.h>

int main(void)
{
  unsigned char switches, leds, i;   //unsigned byte variables.

  DDRD = 0x00;    //PORTD - all bits configured as inputs.
  PORTD = 0xFF;   //PORTD - internal pull-ups enabled.
  DDRB = 0xFF;    //PORTB - all bits configured as outputs.

  while(1) {
    switches = PIND;    //Input byte from SWITCHES.
    leds = 0x00;  //Initialize leds to 0

```

Ready   Errors 0, Warnings 0   Ln 17, Col 41

Learning to use C-SPY

The C-SPY Debugger contains many powerful debugging features. The greater your knowledge of and ability to use these features, the easier it will be for you to get your programs working properly in this course.

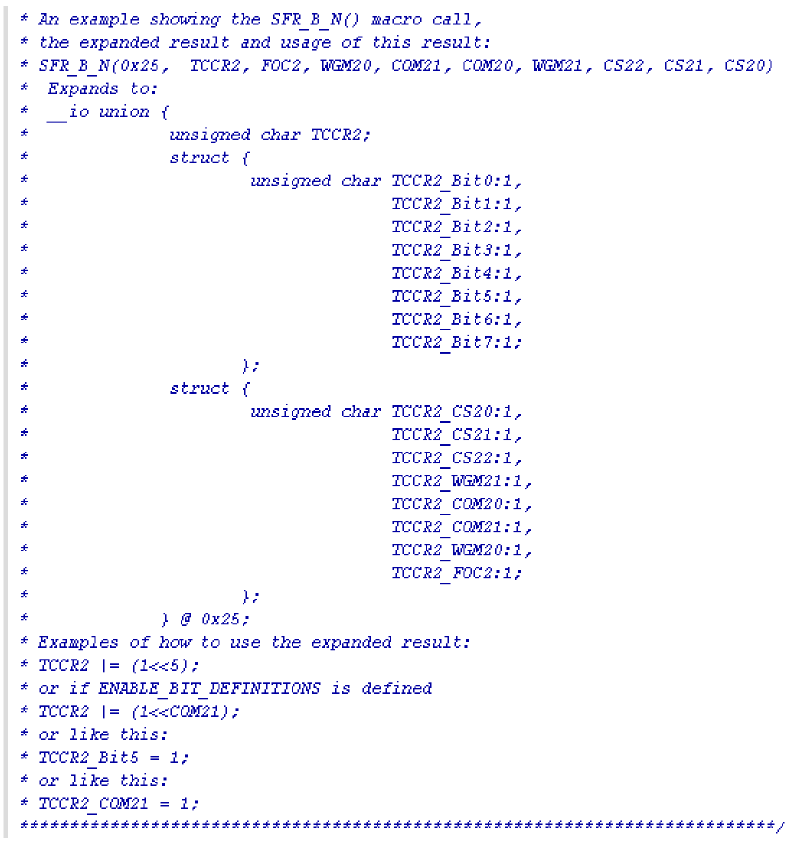
The AVR Embedded Workbench Users Guide provides information on using the features of C-SPY in:
Part 4. Debugging
Part 5. IAR C-SPY Simulator
Part 6 C-SPY Hardware Debugger Systems

This manual is available through the help menu of the IAR Embedded Workshop IDE and as a PDF file on the KickStart distribution.

You should read this documentation and learn to use the C-SPY debugging features as soon as you can. As you read about the various features of C-SPY, apply them to the programs you are developing to become familiar with exactly how they work. This material will likely be the subject of an upcoming quiz or exam problem.

Embedded C Programming Note: Simple I/O Using C

When writing code in Embedded C for the IAR compiler there are many ways to access I/O registers and peripheral registers located in extended I/O space. Included with the IAR compiler is a set of include files, one of these is specific to the microcomputer you are using. For example, for the ATmega128 the include file named “iom128.h” contains all port definitions needed for the microcontroller. To show how to use these port definitions in C, we will examine a section of “iom128.h”.



```
* An example showing the SFR_B_N() macro call,
* the expanded result and usage of this result:
* SFR_B_N(0x25,  TCCR2, FOC2, WGM20, COM21, COM20, WGM21, CS22, CS21, CS20)
*  Expands to:
*  __io union {
*             unsigned char TCCR2;
*             struct {
*                     unsigned char TCCR2_Bit0:1,
*                                   TCCR2_Bit1:1,
*                                   TCCR2_Bit2:1,
*                                   TCCR2_Bit3:1,
*                                   TCCR2_Bit4:1,
*                                   TCCR2_Bit5:1,
*                                   TCCR2_Bit6:1,
*                                   TCCR2_Bit7:1;
*                    };
*             struct {
*                     unsigned char TCCR2_CS20:1,
*                                   TCCR2_CS21:1,
*                                   TCCR2_CS22:1,
*                                   TCCR2_WGM21:1,
*                                   TCCR2_COM20:1,
*                                   TCCR2_COM21:1,
*                                   TCCR2_WGM20:1,
*                                   TCCR2_FOC2:1;
*                    };
*            } @ 0x25;
* Examples of how to use the expanded result:
* TCCR2 |= (1<<5);
* or if ENABLE_BIT_DEFINITIONS is defined
* TCCR2 |= (1<<COM21);
* or like this:
* TCCR2_Bit5 = 1;
* or like this:
* TCCR2_COM21 = 1;
***************************************************************************/
```

This section of “iom128.h” defines the I/O register for TCCR2. Though this section looks complicated we do not need to know everything that is going on here at this time. What we need to know is how to use these definitions. IAR makes this very easy. To access an entire register we can just call it by name. These registers look like unsigned chars to us. For example to set TCCR2 equal to 0xF0 we would simple write:

TCCR2 = 0xF0;

To access individual bits of I/O registers we can also call them by name. To set WGM21 (bit 3) of TCCR2 we can simply write.

TCCR2_WGM21 = 1;

There are many other ways to access bits in I/O registers. These will be discussed in lecture. To get a better idea of how each of these works, you can browse through “iom128.h”.