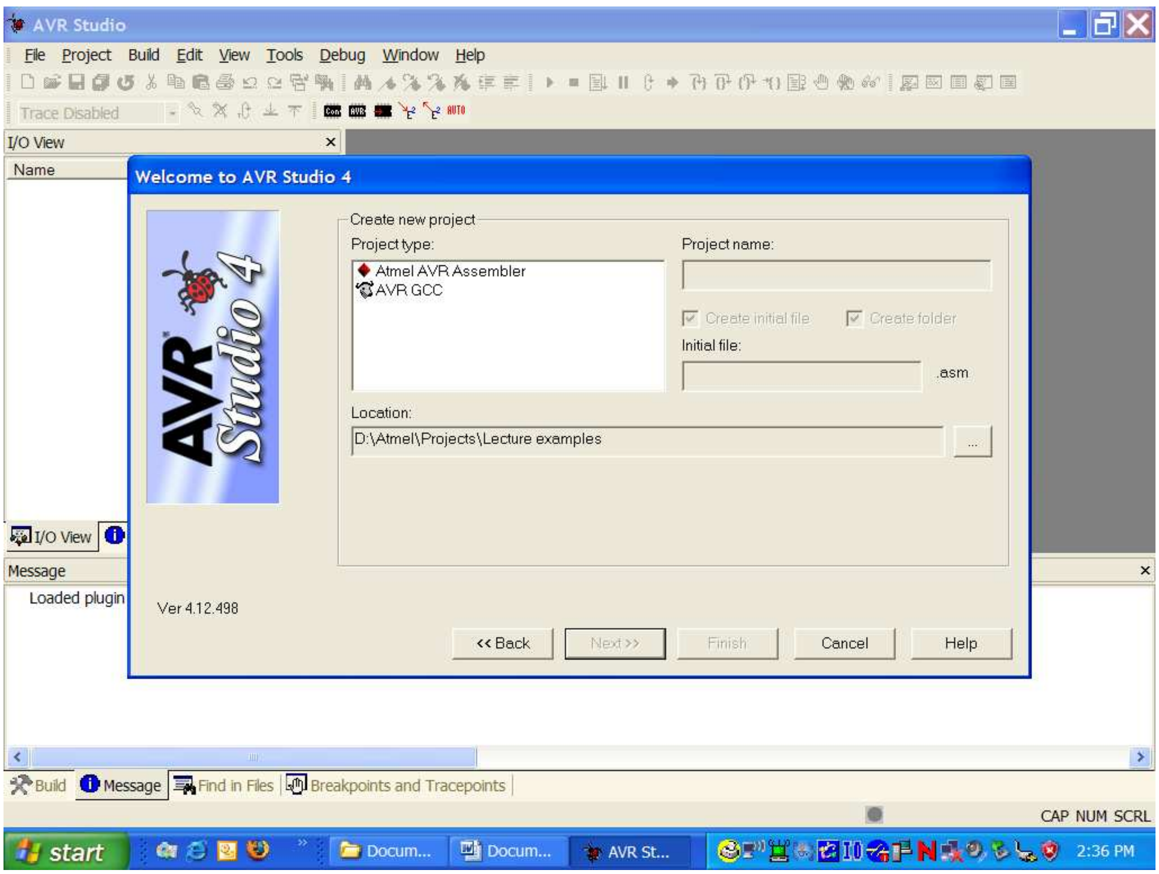
# Mixing C and assembly language programs



It is sometimes advantageous to call subroutines written in assembly language from programs written in C. The reverse is also true. This paper outlines the procedure for doing this with AVR Studio. AVR Studio has two assemblers, the built-in assembler that comes with AVR Studio and the assembler that comes with the GCC plug-in. When a new project is created with AVR Studio, you are given a choice as to the type of project to create. The choices are an Atmel AVR assembly project or an AVR GCC project.




In order to mix C and assembly language, you must create an AVR GCC project. The program you create may be a C program (.c extension), a C++ program (.cpp extension) or an assembly language program (.S extension). When creating an assembly language program, you must be aware of the differences between a GCC assembly program and an Atmel AVR assembly language program.

## Comparison of GCC assembler vs Atmel AVR assembler

This section illustrates the differences between the GCC assembler and the Atmel AVR assembler. The GCC assembler uses the same preprocessor as the GCC C/C++ compiler. Therefore note the use of #include instead of .include. Another difference is in the data segment definition. The GCC assembler allows the initialization of data in its data segment. The data is actually stored in the program memory. The assembler generates start up code that copies initialized data into the SRAM. The Atmel AVR assembler does not allow initialized data in the data segment. Instead, initialized data must be placed in the code segment (usually at the end of the program). The programmer must then supply the code to copy initialized data into SRAM. Note the use of the LPM instruction and the Z-pointer in the Atmel AVR program. Other differences:

| GCC | Atmel AVR |
|---|---|
| hi8 | high |
| lo8 | low |
| .asciz “hello” | .db “hello”, 0 |
| .section .data | .dseg |
| .section .text | .cseg |
| <avr/io.h> | “m32def.inc” |

The next page shows an example of code written for the GCC assembler and repeated for the Atmel AVR assembler.

**GCC assembly language**

```
#include <avr/io.h>

/* The following is needed to
subtract 0x20 from I/O addresses
*/
#define __SFR_OFFSET 0
.section .data
.org 0x0
message:
      .asciz "hello"

.section .text
.global main


main:
      ldi r16, 0xff
      out DDRB, r16
      ldi r16, hi8(RAMEND-0x20)
      out SPH, r16
      ldi r16, lo8(RAMEND-0x20)
      out SPL, r16
      sei
      rcall lcd_init
      ldi XH, hi8(message)
      ldi XL, lo8(message)
      rcall prtmsg
quit:
      rjmp quit
prtmsg:
      ld r24, X+
      cpi r24, 0
      breq done
      rcall lcd_print_char
      rjmp prtmsg
done:
      ret
.end
```

**Atmel AVR assembly language**

```
.include “m32def.inc”

.cseg
.org 0
      rjmp main
.org 0x2A
main:
      ldi r16, 0xff
      out DDRB, r16
      ldi r16, high(RAMEND-0x20)
      out SPH, r16
      ldi r16, low(RAMEND-0x20)
      out SPL, r16
      sei
      rcall lcd_init
      ldi ZH, high(message)
      ldi ZL, low(message)
      rcall prtmsg
quit:
      rjmp quit
prtmsg:
      lpm r24, Z+
      cpi r24, 0
      breq done
      rcall lcd_print_char
      rjmp prtmsg
done:
      ret
message:
      .db “hello”, 0
```

## Mixing C and Assembly

To allow a program written in C to call a subroutine written in assembly language, you must be familiar with the register usage convention of the C compiler. The following summarizes the register usage convention of the AVR GCC compiler.

### Register Usage

**r0** This can be used as a temporary register. If you assigned a value to this register and are calling code generated by the compiler, you'll need to save r0, since the compiler may use it. Interrupt routines generated with the compiler save and restore this register.

**r1** The compiler assumes that this register contains zero. If you use this register in your assembly code, be sure to clear it before returning to compiler generated code (use "clr r1"). Interrupt routines generated with the compiler save and restore this register, too.

**r2–r17, r28, r29** These registers are used by the compiler for storage. If your assembly code is called by compiler generated code, you need to save and restore any of these registers that you use. (r29:r28 is the Y index register and is used for pointing to the function's stack frame, if necessary.)

**r18–r27, r30, r31** These registers are up for grabs. If you use any of these registers you need to save its contents if you call any compiler generated code.

### Function call conventions

### Fixed Argument Lists

Function arguments are allocated left to right. They are assigned from r25 to r8, respectively. All arguments take up an even number of registers (so that the compiler can take advantage of the **movw** instruction on enhanced cores.) If more parameters are passed than will fit in the registers, the rest are passed on the stack. This should be avoided since the code takes a performance hit when using variables residing on the stack.

### Variable Argument Lists

Parameters passed to functions that have a variable argument list (printf, scanf, etc.) are all passed on the stack. **char** parameters are extended to **ints**. The parameters are pushed to the stack in right to left order. The variable, x, is a uint8_t and notice that it is extended to a 16-bit value with the upper 8-bits set to zero (eor r25, r25).

```
lcd_printf(++x, x);
fa8: 89 81          ldd r24, Y+1   ; This is x
faa: 99 27          eor r25, r25   ; 0-extended to 16-bits
```

```
fac: 9f 93          push r25        ; and pushed to the stack
fae: 8f 93          push r24
fb0: 89 81          ldd r24, Y+1 ;
fb2: 8f 5f          subi r24, 0xFF ; This forms ++x
fb4: 89 83          std Y+1, r24
fb6: 99 27          eor r25, r25    ; 0-extended to 16-bits
fb8: 9f 93          push r25        : and pushed to the stack
fba: 8f 93          push r24
fbc: 0e 94 03 06    call 0xc06
fc0: 0f 90          pop r0
fc2: 0f 90          pop r0
fc4: 0f 90          pop r0
fc6: 0f 90          pop r0
```

In this example, the function has two arguments that are passed in left to right order. Here is the function prototype:

```
void lcd_goto_xy(uint8_t x,uint8_t y);
```

The parameter, x, is passed via r24 and the parameter, y, is passed in r22. Each parameter is passed as 2-bytes. Therefore, x is actually passed in r25:r24. Since r25 is not explicitly cleared it is ambiguous as to the value actually passed. The function apparently ignores the value in r25.

```
lcd_goto_xy(0, 1);
fc8: 61 e0          ldi r22, 0x01   ; 1
fca: 80 e0          ldi r24, 0x00   ; 0
fcc: 0e 94 ab 01    call 0x356
```

**Return Values**

8-bit values are returned in r24. 16-bit values are returned in r25:r24.
32-bit values are returned in r25:r24:r23:r22. 64-bit values are returned in r25:-r24:r23:r22:r21:r20:r19:r18.

**Examples**

The following examples illustrate the calling convention and register usage of the GCC compiler. In this example, an assembly language program calls functions written in C. Below the function prototypes are listed.

```
; initilaize LCD
void lcd_init(void);
;set cursor position
void lcd_goto_xy(uint8_t x,uint8_t y);
; print character
void lcd_print_char(uint8_t symbol);
;print string at current position
```

```
void lcd_print_string(char *string);
;print hex number on LCD
void lcd_print_hex(uint8_t hex);
;print int8 on LCD
void lcd_print_int8(int8_t no);

#include <avr/io.h>
.section .data
message:
        .asciz "aloha"
.section .text
.global main
main:
        ldi r16, lo8(RAMEND)        ;Initialize Stack Pointer
        out SPL, r16                ;RAMEND is defined in iom32.h
        ldi r16, hi8(RAMEND)        ;RAMEND = 0x083f for Atmon compatibility
        out SPH, r16
        sei                         ;Needed for Atmon compatibility
        rcall lcd_init
        clr r25
        ldi r24, 255                ;8-bit param passed via r24
        rcall lcd_print_int8
        ldi r24, ' '
        rcall lcd_print_char
        ldi r24, 255
        rcall lcd_print_hex
        ldi r24, ' '
        rcall lcd_print_char
        ldi r24, 255
        rcall lcd_print_uint8
        ldi r24, 0                  ;First 8-bit param passed via r24
        ldi r22, 1                  ;Second 8-bit param passed via r22
        rcall lcd_goto_xy           ;Cursor at position 0 of line 1
        ldi r25, hi8(message)
        ldi r24, lo8(message)       ;16-bit pointer passed via r25:r24
        rcall lcd_print_string
done:
        rjmp done
.end
```

If calling a function written in assembly language from a program written in C, the calling convention must be followed as described above. Here are some guidelines to follow when writing assembly functions that can be called from C.

·If you use registers r2-r17, r28, r29 you must preserve them by pushing them to the stack and pop them before you return. The C compiler expects these registers to be preserved across function calls.

· Parameters are passed to your function via registers r25-r8 as discussed earlier.

· Results are returned via r25-r18 as discussed earlier.

· The C compiler expects register r1 to contain the value 0. If you use it in your function, be sure to clear it before you return.

· If you are going to call a C function from within your assembly function and if you are using r18-r27, r30, r31 in your function, you should push these before you call the C function. The C compiler treats these as registers that it may clobber. Therefore their contents are not guaranteed to be the same as before the call.

The following is an example of a program written in C that calls a function written in assembly language.

**The C program**

```
////////////////////////////////////////////////////////////////////////////
//
//	Program to demonstrate how an assembly language function can be called from C
//	To make it compatible with ATMON, the following has to be done:
//
//		Project/Configuration Options
//			Custom Options
//				On command line type -minit-stack=0x83f
//				Click Add
//
//	This causes the compiler to initialize the Stack Pointer to RAMEND-0x20.
//	This will prevent the application from corrupting the stack used by ATMON.
//	The start up code inserted by the compiler still initializes the Stack Pointer
//	to 0x85f however. The following code causes the linker to add code to
//	re-initialize the Stack Pointer to 0x83f:
//
//	void my_init_stack (void) __attribute__ ((naked)) __attribute__ ((section (".init2")));
//
//		void my_init_stack (void) {
//			SPH = 0x08;
//			SPL = 0x3F;
//		}
//
```

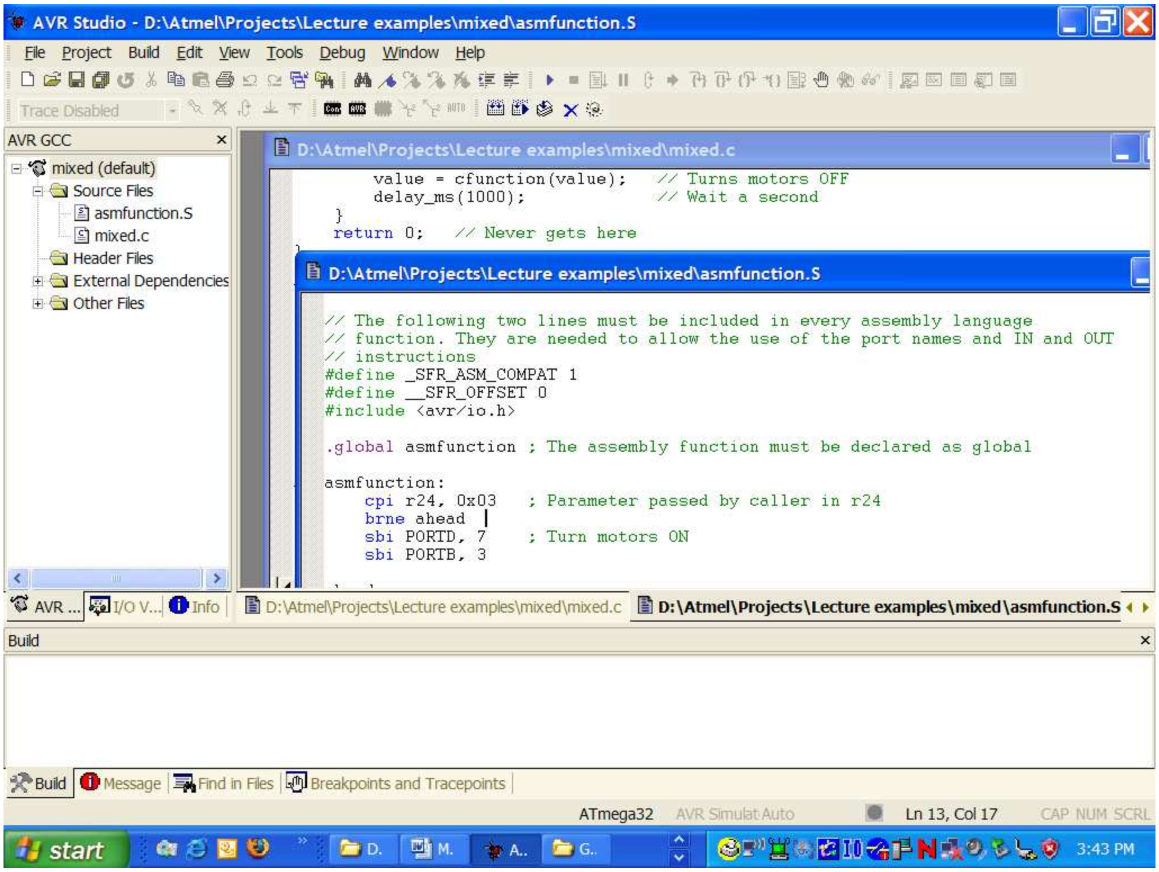
```
//	See the avrlibc documentation for details.
////////////////////////////////////////////////////////////////////////////////

#include <avr/io.h>			// needed for IO port declarations
#include <inttypes.h>			// needed for type declarations
#include <stdlib.h>
#include <MSOE/delay.h>
#include <avr/interrupt.h>

extern uint8_t asmfunction(uint8_t); // Assembler function is external
uint8_t cfunction(uint8_t);		// C function prototype

// Global variable accessible by assembler code and C code
uint8_t value;

void my_init_stack (void) __attribute__ ((naked)) __attribute__ ((section (".init2")));
void my_init_stack (void) {
	SPH = 0x08;
	SPL = 0x3F;
}

int main(void)
{
	sei();
	DDRB = 0xff;			// PB3 is output
	DDRD = 0xff;			// PD7 is output
	DDRA = 0xc0;			// Motor direction bits are outputs
	PORTA = 0xc0;
	PORTB = 0;
	PORTD = 0;
	value= 0x03;
	while(1)

	{

		value = asmfunction(value);	// Turns motors ON
		delay_ms(1000);			// Wait a second
		value = cfunction(value);	// Turns motors OFF
		delay_ms(1000);			// Wait a second
	}
	return 0;				// Never gets here
}
```

```
uint8_t cfunction(uint8_t a)
{

	if (a == 4)
	{
		PORTB = PORTB & ~(1<<3);  // Turn motors OFF
		PORTD = PORTD & ~(1<<7);
		a = 3;
	}
	return a;
}
#include <MSOE/delay.c>
```

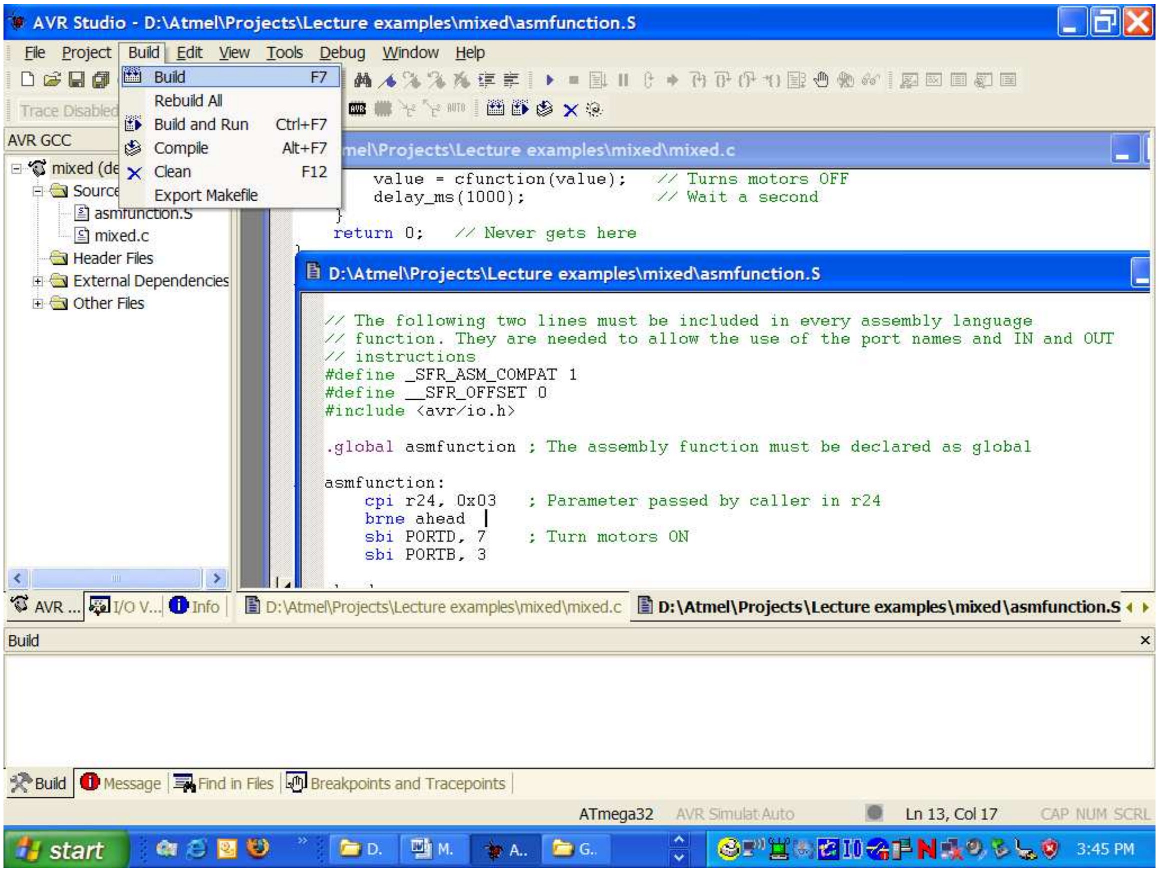
**The assembly language function**

```
// The following two lines must be included in every assembly language
// function. They are needed to allow the use of the port names and IN and OUT
// instructions
#define _SFR_ASM_COMPAT 1
#define __SFR_OFFSET 0
#include <avr/io.h>

.global asmfunction    ; The assembly function must be declared as global

asmfunction:
	cpi r24, 0x03   ; Parameter passed by caller in r24
	brne ahead
	sbi PORTD, 7 ; Turn motors ON
	sbi PORTB, 3

ahead:
	ldi r24, 0x04;  ; Return value to caller in r24
	ret
```

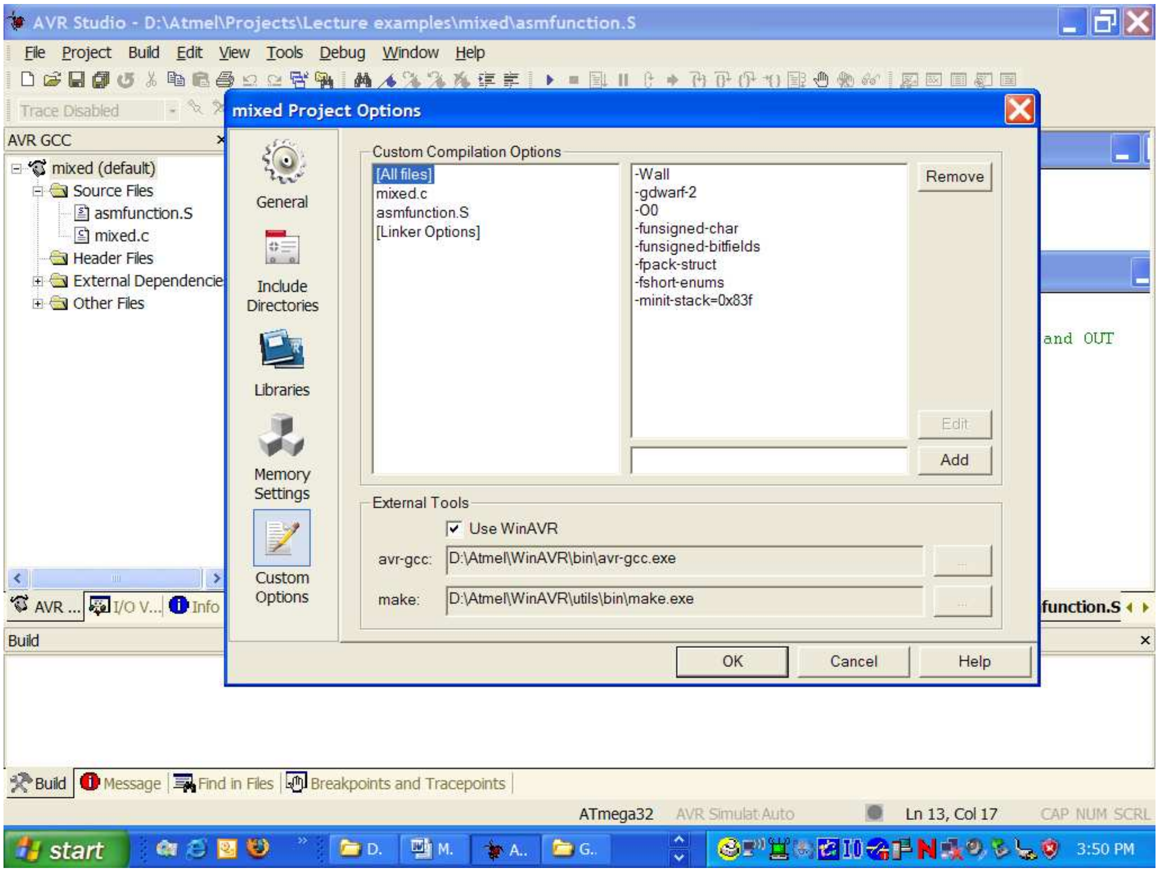
The assembly function is in a separate file that must be added to the main project which is the C program. Do this by right clicking on Source Files and adding it to the project. You will then see both files listed as shown below.




To build the project just click the Build menu and choose Build.




As mentioned earlier, another thing that must be done for Atmon compatibility is to choose the correct Project Configuration Options as illustrated below:




The important option is –minit-stack=0x83f. This is needed to insure that 0x20 locations are reserved for the bootloader (Atmon). Enter this line on the command line and click the Add button. Do this BEFORE building the project.