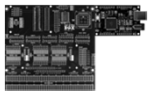


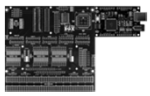
EEL 3744

Menu

• Introduction to C for Atmel XMega

See docs/examples on web-site:
Usage of simple …, EBI.c, usart_Serial.c
See Software/Docs:
Getting Started Writing C …,
Info on C for the Atmel XMEGA,
Tips to Optimize C Code

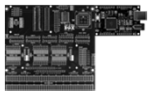
For mixed C/Assembly:
- **"Mixed C and Assembly (for Atmel XMEGA)",**
- **VectorAdd_Mixed_ASM.s, VectorAdd_Mixed.c,**
- **VectorADD_Casm.c, Input_Port_C.c**



EEL 3744

Menu

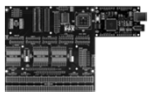
- Motivation
- Overview of C Basics
- Variables, Types, and Definitions
- Conditionals
- Ports and Registers
- Interrupts
- Pointers
- C Example
- **NOT** covered, but possibly useful info after 3744
  >Using C with Assembly: slides 42-…



EEL 3744

Introduction to C & other High-level Languages

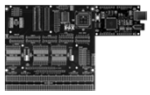
- Source Languages (e.g., C, C++, Java)
  - > Most modern programs are written in high-level languages (such as C), because it is generally easier than Assembly
  - > A compiler is used to convert a source language into a target language (e.g., Assembly), resulting in object code (just as an assembler converts Assembly)
  - > A compiler is given limited time to "optimize" the object code in terms of speed, memory usage, etc.
  - > The resulting object code is not guaranteed to be as fast or efficient as can be done with Assembly code



EEL 3744

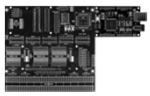
Mixed C/Assembly

- Why Mixed Coding?
  - > Occasionally a programmer may want to take advantage of the increased specificity in Assembly to improve the resulting object code (usually for increased speed)
  - > When programming in high-level language, there may be limitations due to processor specific features
    - – Memory Mapping
    - – External Bus Control


EEL 3744

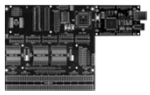
Costs/Benefits of Mixed C/Assembly versus C or Assembly

- Benefits
  - > Power of a high-level source language such as C
    - – Libraries (Graphical, Math, etc.), String-processing functions, etc.
  - > Use of C structures and layout
  - > Speed and control of assembler (optimization)
  - > Direct control of code placement
  - > Access to processor specific functions
- Drawbacks
  - > Assembly coding in mixed coding is slightly different from standard Assembly
    - – Naming conventions, function usages
  - > Code is less portable (i.e., is often specific to the processor)


EEL 3744

What is Mixed C/Assembly Coding?

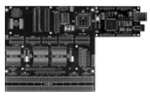
- Use Assembly code to improve C code or take advantage of a specific processor's capabilities
- For our board, mixed coding is handled by the AVR-GCC toolchain for compiling with the GNU Assembler (GAS)
- There are two ways to mix C and Assembly code
  - > Use separate files for C code and Assembly code, the **.c** extension and **.s** extension respectively
  - > Inline Assembly code directly inserted into the C code
  - > You will **NOT** be expected to write mixed code
    - – See the end of this lecture for more mixed C/Assembly info



EEL 3744

Basic C Structures

- Overview
  - (The primary C structures are also used in Mixed C/Assembly)
  - Preprocessor Directives
  - Functions (prototypes)
  - Main Function
  - Function Calls



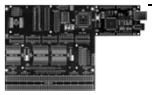
EEL 3744

Basic C Structures

- C (or Mixed C/Assembly) start with a standard structure
  > Example:

```
#define F_CPU 2000000
#include <avr/io.h>

// function prototype below
int add(int x, int y);
```

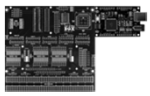
```
// main routine below
int main(void)
{
   int x=3, y=7, z;
   while(1)
   {
      z=add(x,y);
   }
}
// function is below
int add(int x, int y)
{
   return (x+y);
}
```



EEL 3744 Basic C Structures Preprocessor Directives

- Preprocessor Directives **(#** sign)
  - > The **#define** is an "object-like" macro definition (similar to **.equ** or **.def**)
    - – Allows us to define a value for a symbolic name that may be used in our code or the systems code
  - > The **#include** is a method used to include other files that include code (similar to **.include**)
    - – If using brackets (< >), the file is expected in standard compiler include paths
    - – If using parenthesis (**" "**), the path for the file will include the current source directory
  - > Example:

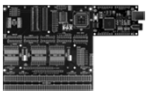
```
#define F_CPU 2000000
#include <avr/io.h>
```

EEL 3744 Basic C Structures Preprocessor Directives

- There are various other types of preprocessor directives that may be used
  - >The given example shows
    - – A defined rate to be used for the clock frequency of the XMEGA
    - – A definitions file to be used for an AVR processor
  - > Example:

```
#define F_CPU 2000000
#include <avr/io.h>
```

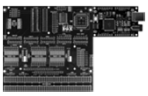


EEL 3744

Basic C Structures
Function Prototypes

- Function Prototypes
  - > Functions may not be called unless they have been defined with a **prototype**
    - – Some compiles do not require this, **but 3744 does!**
  - > Function **prototypes** allow a function to be partially defined
    - – Prototypes are typically found near the top of the file, below preprocessor directives
    - – Defines the function, but does not supply a body of code
    - – Functions are defined later in the program with its body of code
    - – Allows the function to be called before its complete definition
    - – Example:

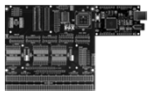
```
int add(int x, int y);
```

EEL 3744

Overview of C Basics

- Main Functions
  - > One in every program, starting point for all code
- Functions
  - > Organized scheme for holding code
  - > Allows passing of parameters and returning results
  - > Use of prototypes for organizing code
    - – Prototypes should **ALWAYS** be used; they are **NOT** optional, even if Atmel Studio does not require them in the present version
- Preprocessor Directives
  - > Defining variables or names as values
  - > Including extra files detailing code
  - > Creating Macros to detail functions or values

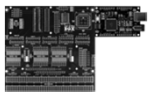


EEL 3744

Basic C Structures
Main Function

- Main Function
  - > A few properties of the **main** function
    - – Resembles a standard function
    - – A single **main** function is required for **each** project
    - – Starting point for the project
  - > Example:

```
int main(void)
{
   while(1)
   {
      add(x,y);
   }
}
```

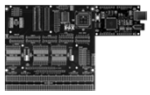



EEL 3744

Basic C Structures
Main Function

- > When the **main** function ends, the program ends
  - – A **while** loop may be used to run a block of code "forever"
    - ☞Like the "dog chasing his tail" loop used at the end of Assembly programs
- > The example also shows how a function may be called
  - – The name of the function to be called is used
  - – If the function requires arguments, they may be passed within parenthesis (x and y in the below example)
- > Example:

```
int main(void)
{  …..
   while(1)
   {
      add(x,y);
   }
}
```

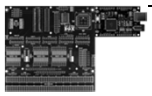


EEL 3744

Basic C Structures
Function Prototypes

- Function Prototype
  - > Example:

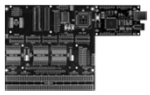
```
int add(int x, int y);    // this is the prototype

int main(void)
{
   while(1)
   {
      add(x,y);           // this is the function call
   }
}
int add(int x, int y)     // this is the function
{
   return (x+y);
}
```

EEL 3744

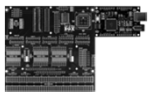
Overview of C Basics

- We can (and will) write programs entirely in C
- Values are defined using variables (not registers)
  - > No registers are directly referenced (although they will be used “underneath the hood,” i.e., after compilation)
- High-level conditional structures are available for flow control
  - > Easier use of comparisons
  - > No branch functions (used in Assembly) available (or necessary)
- Cleaner way of looking at port usage and interrupts


EEL 3744

Variables

- Variables
  - >Type
    - – Standard: int, char, float, double, etc.
    - – Special: uint8_t, uint16_t, int8_t, int16_t, etc.
  - >Scope of Variables
    - – Local: Declare at the beginning of a function in which it is to be used
    - – Global: Declare outside of any function, typically at the top of the c file
  - >Modifiers: causes variable to use more or less memory The following are typical examples
    - – short: 2 bytes
    - – int: 4 bytes
    - – long: 4 to 8 bytes


EEL 3744

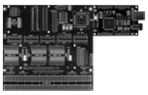
How to use Variables

- When defining variables, there are many types available

Type	Expression
char	
standard	'j'
ascii	106, 0x6A
string	"microp"
int	
decimal	37
hex	0x37
binary	0b110111
float	0.00037
double	37.000001

Example:

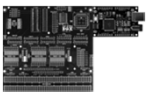
```
char char1 = 'j';
char char2 = 0x6A
char str[7] = "microp"
char str[] = "3744 #1"
char *str = "Hi!"
int x = 37;
int y = 0x37;
float = 0.00037;
double = 37.000001;
```


EEL 3744

Arrays

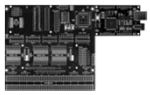
- An Array is a method of grouping a series of same type elements in a single variable located in contiguous memory locations
  - > Syntax: *type name [elements] = {initialized value list};*
    - – Type may be any variable type
    - – Elements states the size or number of variables in the array
    - – The initialized value list represents the initial values populating the array
      - ☞If defining an initial list, the value of elements may be omitted
  - > Examples:

```
uint8_t buffer[20]; // unsigned character (8 bits)
char message[] = {'m', 'i', 'c', 'r', 'o', 'p'}
                       // no 0x0 appended
char string[] = "microp"; // an 0x0 is appended
```



EEL 3744

Conditionals

- Controls flow of code given programmer defined conditions
- Handles the concept of comparisons and branches (that were used in Assembly)

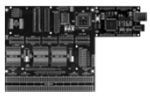


EEL 3744

If, Else If, Else

- Check conditional statements for truth values
  - > **if** conditional
    - – If expression is true, execute expressions within conditional block
    - – If expression is false, check any following conditionals tied to if conditional
  - > **Else if** conditional (may be omitted)
    - – Follows same concept as if conditional, giving more conditional checks
  - > **Else** conditional (may be omitted)
    - – If all other conditionals fail, this block is executed

`simple_if_statements.c`



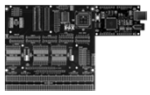
EEL 3744

If, Else If, Else

> Syntax:

```
if (expression) {
    <statements>
} else if (expression) {
    <statements>
} else {
    <statements>
}
```

`simple_if_statements.c`

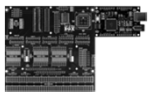


EEL 3744

Relational Operators

- To create a conditional expression, utilize one of relational operators

Relational Operator	Definition	Example (True results)
>	Greater than	47 > 37
>=	Greater than or equal to	47 >= 47
<	Less than	37 < 47
<=	Less than or equal to	37 <= 47
==	Equal to	47 == 47
!=	Not equal to	37 != 47

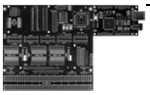


EEL 3744

Boolean Operators

- To create more complex conditional expressions, Boolean operators may be used

Boolean Operator	Definition	Example (True results)
&&	AND two expressions	((47 >= 47) && (47 > 37))
||	OR two expressions	((37 != 47) || (37 > 47))
!	Complement expression	!(37 > 47)



EEL 3744

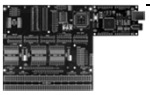
While and Do While

- **While** loops allow for repetition
  - >If expression is true, execute expressions within conditional block and continue to execute until false
  - >If expression is false, exit conditional block and continue with code following the while block
- **Do While** loops allow for repetition
  - >Follows same concept as While loop, except condition expression happens at the end of the code
  - >Will execute code block at least once
- Syntax:

```
while (expression) {
   <statements>
}
```

```
do {
   <statements>
} while (expression)
```

simple_whiles_loops.c
multiple_whiles_loops.c



EEL 3744

For Loop

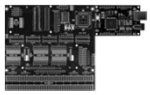
- **For** loops allow for repetition while also iterating
  - >Has a start value, e.g., **`int i = 0`**
  - >Loops until an end condition has been met, e.g., **`i < 10`**
  - >Every loop, the start value will either be increase or decrease, e.g., **`i++`** or **`i--`**
  - >Syntax:

```
for (start value; end condition; inc/dec value) {
  <statements>
}
```

  - >Example:

```
for (int i = 0; i < 10; i++) {
   <statements>
}
```

if_and_for_loops.c

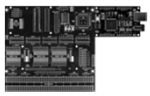

EEL 3744

Switch Statements

- **Switch** statements acts as selection control, changing the code flow through a multi-way branch
  - > Multi conditional system such as a large **if** conditional structure
  - > May also be used to create a state machine
  - > Has a single variable that multiple cases may be used to compare to that value and then be executed
  - > Syntax:

```
switch ( variable )
{
case value1:
   <statements>
   break;
case value2:
   <statements>
   break;
default:
   <statements>
   break;
}
```


switch_statements.c

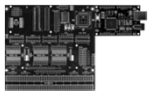

EEL 3744

Break

- While running loops, it is possible to break out of the code at anytime using the break expression
  - > If using nested loops, the break expression will break out of all loops
  - > Can use labels to jump to the outer loop
- Example:

```
outer_loop:
for (int i = 0; i < 10; i++) {
   for (int j = 0; j < 10; j++) {
      if ( (i * j) == 37 ) //won't happen
           break outer_loop;
   }
}
```


if_and_for_loops.c



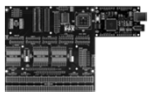
EEL 3744

Volatile (test this)

- XXX

```
asm volatile ("nop");
--------------
void RoughDelay1sec(void)
{
    volatile uint32_t ticks;
//Volatile prevents compiler optimization
    for(ticks=0;ticks<=F_CPU;ticks++);
//increment 2e6 times -> ~ 1 sec
}
```

if_and_for_loops.c



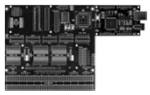
EEL 3744

PORT / Modules

- May access Ports similarly to Assembly, use the **header** file
  `#include <avr/io.h>`
  - >HINT: Right mouse click on variable name and choose goto implementation
- When programming, make use of intelli-sense
- Naming follows the AVR Manuals
- Access PORTS (or modules) by name directly
  - >Examples:

```
PORTA_DIRCLR = 0xFF
USARTC0_CTRLA = 0xFF
```

simple_whiles_loops.c
multiple_whiles_loops.c

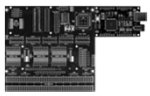

EEL 3744

Registers

- It is possible to use C syntax and structures to access registers of various modules in a cleaner manner
- Instead of typing the entire name, you can
  - >Enter the module name
  - >Enter a period
  - >Enter the register name (with autocomplete)
  - >Example:

```
PORTA.DIRCLR = 0xFF
USARTC0.CTRLA = 0xFF
```


`simple_whiles_loops.c`
`multiple_whiles_loops.c`


EEL 3744

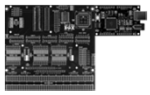
Bitwise Operators

- When modifying ports and registers, bitwise operators are often used


`usart_serial.c`

Symbol	Bitwise Operation
&	AND
|	OR
^	Exclusive OR
<<	Left Shift
>>	Right Shift
~	One's Complement

- `&`, `|`, and `^` may be tied to equal sign to simply AND or OR the variable with another variable
- Example: `PORTB_DIRCLR |= 0xF0;`



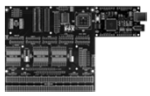
EEL 3744

Bitmasks

- Programming in C allows a user to use various defined enumerations or structures when working with PORTs and control registers
- Standard bitmasks allow a user to change only specific bits in a register when desired (noted by bm)
  - >Example:

```
PORTB_DIRCLR = PIN2_bm | PIN4_bm;
```

usart_serial.c



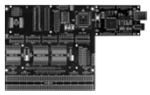
EEL 3744

Bitmasks

- Group **Configuration Bitmasks** allow a user to change only multiple bits representing aspects of a control register
  - >Example

```
USARTC0.CTRLC = USART_CMODE_ASYNCHRONOUS_gc
              | USART_PMODE_DISABLED_gc
              | USART_CHSIZE_8BIT_gc;
```

usart_serial.c



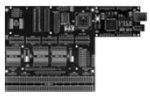
EEL 3744

Interrupts

- First must include the interrupt header file
  ```
  >#include <avr/interrupt.h>
  ```
- After defining interrupts use module registers, enable or clear the global interrupts as needed
  ```
  >sei();
  >cli();
  ```
- Finally, write the interrupt service routine for the required interrupt vector

usart_serial.c

```
ISR(USARTC0_RXC_vect)
{
  USARTC0.DATA = USARTC0.DATA;
}
```

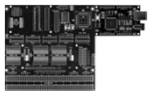



EEL 3744

Interrupts

- Occasionally clearing the interrupts is often desired, but preserving the status register is **ALMOST ALWAYS** required
- I think the below is automatically done, but if it isn't, this would not be a bad idea
  - Example:

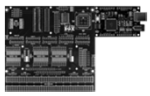
```
uint8_t sreg = SREG; // save status reg
cli();
<statements>
SREG = sreg;         // restore status reg
```



EEL 3744

Using C with Assembly

- It is possible to use C and Assembly more seamlessly by creating variables and functions in C and using them in various ways with Assembly
  - > This puts less emphasis on the Assembly code, using it only as needed to improve code
  - > When using more of C's capabilities, some extra considerations must be placed on the choice of Registers in Assembly (as described in the lecture *Intro to Mixed C and Assembly*)

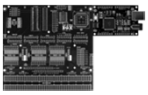


EEL 3744

Using C with Assembly Passing Arguments

- Arguments are passed to Assembly functions in register pairs or via the stack if more than 9 arguments
  - > **Word Data** takes both registers
  - > **Byte Data** takes the lower register

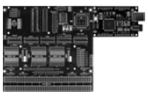
Argument	Registers
1	r25:r24
2	r23:r22
3	r21:r20
….	
9	r9:r8



EEL 3744 Using C with Assembly Returning Values

- Return values always use the following convention

Type	Registers
8 bit data (sign or zero extended)	r25:r24
32 bit data	r25:r22
64 bit data	r25:r18



EEL 3744 Pointers

- Pointers may be used to contain the address of a variable. You create pointers using the * symbol

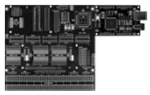
```
>int value = 5;      // A variable holding a value of type int
>int *valuePtr;      // A pointer to a value of type int
```

- To reference the address of the pointer you use the & symbol

```
>valuePtr = &value;  // Place address of value in pointer
```

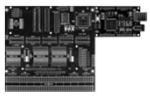
- To get the data that the pointer points to, you can "dereference it" by using the * symbol on the pointer

```
>int data = *valuePtr;   // Get data pointed to by pointer
```


EEL 3744

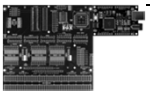
Examples

- External Bus Interface (EBI) example: `ebi.c`
- Asynchronous Serial example: `usart_serial.c`


EEL 3744

C Projects and I**nline** Assembly

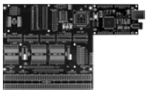
- If it is only desired to add a few lines of assembly code to a C Project, it is possible to add assembly "inline"
- Inline assembly uses the asm function with the following template
  asm volatile(*asm-template* : *output-operand-list* :
  list-input-operand : *clobber list*)
- When using the **asm** function, the compiler will have a harder time optimizing code
- The **volatile** keyword may be used to prevent the compiler from attempting to optimize the line
  > The keyword volatile may be omitted, but then the compiler might optimize away your intended structure



EEL 3744 **Inline** Assembly

asm volatile (asm-template : output-operand-list :
list-input-operand : clobber list)

- The ***asm-template*** component of the asm function follows standard Assembly with small changes
  - > The *Mixed C and Assembly (for Atmel XMEGA)* document detail any required changes
    - Example:
      asm volatile ("STS %0, r18" : "=m" (EBI_CTRL));
      - ☞ STS command above is used to define EBI_CTRL
      - ☞ The %0 is a place holder showing that the defined operand will come later in the template
      - ☞ The output operand section, EBI_CTRL, is defined as an output only memory ("=m") location address

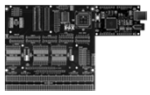


EEL 3744 **Inline** Assembly

asm volatile (asm-template : output-operand-list :
list-input-operand : clobber list)

- The *asm-template* may use "%" expressions to define placeholders replaced by operands in the *output-operand-list* and *list-input-operand*

Placeholder	Replaced by
% n	By argument in operands where n = 0 to 9 for argument
A% n	The first register of the argument n (bits 0 to 7)
% B n	The second register of the argument n (bits 8 to 15)
% C n	The third register of the argument n (bits 16 to 23)
% D n	The fourth register of the argument n (bits 24 to 31)
% A n	The Address register X, Y, or Z
% %	The % symbol when needed
\ \	The \ symbol when needed
\ N	A newline to separate multiple asm commands
\ T	A tab used in generated asm



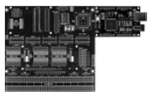
EEL 3744

Inline Assembly

asm volatile (asm-template : output-operand-list :
list-input-operand : clobber list)

- The *output-operand-list* and *list-input-operand uses various modifiers as needed for the operands given*

Modifier	Meaning
=	Output operand
&	Not used as input but only an output
+	Input and Output Operand



EEL 3744

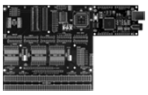
Examples

- Vector Add Mixed
  - > **.s** File Compilation Example
  - > Requires both .c and .s file
- Later
  - > Vector Add
    - – Inline Assembly version
  - > Input Port
    - – Inline Assembly version

`VectorAdd_Mixed.c`
`VectorAdd_Mixed.s`

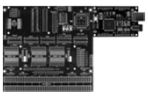
`VectorAdd_Casm.c`

`Input_Port_C.c`



EEL 3744

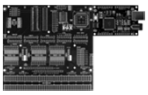
Intro to C Programming
by Daniel Gonzalez



EEL 3744

Common Types in C

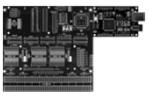
–uint8_t , uint16_t, uint32_t	• Unsigned 8, 16, 32 bit umbers, respectively
int8_t, int16_t, int32_t	• Signed 8, 16, 32 bit numbers, respectively
Char	• Single character - 8 bits
Char[100]	• Array of characters, or a string - 800 bits
Int	• Integer - 32 bits
Float	• Decimal number with precision of 7 decimal places - 32 bits
Double	• Twice the precision of floats - 64 bits
Bool	• True or False



EEL 3744

Good Habits for Initializations and Declarations of Variables

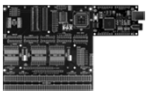
- Declaring your variables at the very beginning of the program lets you (and any other programmer) know what variables are being used throughout the application
- Giving variables appropriate and detailed names lets you (and anyone else) know their general purpose
- Always initialize number types to 0, or char types to ‘ ‘
  - This avoids the variables being initialized to some random address, so you know exactly what that variable is initially assigned.



EEL 3744

Additional IF-ELSE-THEN Example

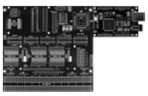
```
PORTE.DIRSET = 0xFF;

uint8_t key = 0; // usually this is at the top of a program, so I initialize it to zero
key = scan_keypad(); // get key pressed from keypad

if(key == 0)
    PORTE.OUTCLR = 0xFF;    // notice if there is only one line after the if statement, we don't need brackets
else if(key == 1)
    PORTE.OUTSET = 1;
else
{
    PORTE.OUTSET = 0xFF;
    //if we had more lines of code, brackets are necessary
}
```



EEL 3744

Additional Switch Example

```
PORTE.DIRSET = 0xFF;

uint8_t key = 0; // usually this is at the top of a program, so I initialize it to zero
key = scan_keypad(); // get key pressed from keypad

switch(key) // testing key
{
    case 0:
        PORTE.OUTCLR = 0xFF;
        break;  // need this break, or the code below will run
    case 1:
        PORTE.OUTSET = 1;
        break;
    default:    // equivalent to an else statement
        PORTE.OUTSET = 0xFF;
}
```

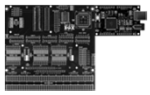



EEL 3744

Loops - Overview

- For Loops:
  - Use when you (or the program) know exactly how many times to iterate some section of code.
- While Loops:
  - Use when the number of iterations is dependent on some state(s) of a variable(s)
- Do-While Loops:
  - Similar to the While, except it does a post test instead of a pre test.
  - Always iterate at least once

Pro tip: For IF or LOOP statements, if there's only one line of code after the loop, you don't need brackets.

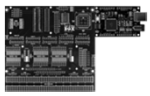


EEL 3744

Additional Loop Examples

Using a Uint8_t would be faster!

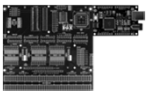
```
for(int i = 0; i < 10; i++)
{
    //code loops exactly 10 times
}

while(test != 0 || test != 0xFF)
{
    //code will loop as long as test does not equal 0 OR 0xFF
    //if the variable test is never changed in this loop, we will be in an infinite loop
}

do
{
    // code will loop at least once, and until test1 does not equal 0 AND
    // test 2 does not equal 0xFF
} while (test1 != 0 && test2 != 0xFF);
```

EEL 3744

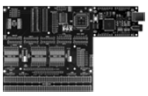
Functions

- Functions can be thought of as subroutines in Assembly
- They usually perform one specific task, and can be used as modular code
- Function types include
  - > Uint8_t, uint16_t, etc.
  - > Int
  - > Void (does not return anything)
  - > Char
  - > Float
  - > Double
- The parameters passed into a function are called the function's argument.
- Global variables can be used in functions, and can also be changed!



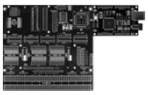
EEL 3744 Good Habits for Using Functions

- **USE PROTOTYPES!!!**
  - > A prototype is the function's definition. This lets you know what type the function is, and what parameters are passed into its argument.
  - > Also, it allows you to place your functions AFTER the main routine, making the "meat" of your program the first thing you see, as opposed to the last.
- Try to make your main routine consist of primarily function calls (along with loops and IF-ELSE statements). This makes your program very clean and readable.



EEL 3744 Headers

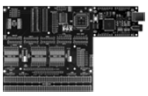
- Headers (or .h files) are files that can be "included" in your main source file. This enables whatever code in the header files to be accessible by the main program.
- Typically, header files contain multiple function prototypes, and has a related .c file for the function's actual implementation.
- For our purposes, we will put both the prototypes AND the function's implementation in the same header file. It's not convention, but it is easier for the TAs to grade, as you will only need to submit a header file.
- By using headers, we have effectively made sections of code completely modular, which is one of our main goals in this class.



EEL 3744

Headers

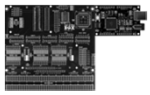
- Some example of helpful header files to create for this class:
  - > INITs
    - – Any initialization of a particular system can be put into a header
    - – i.e. EBI_INIT for LCD, EBI_DRIVER, DAC_INIT, ADC_INIT, etc.
    - – Remember to confirm all your configurations are correct! This is where bit masks come in handy in C, as you can clearly see what your code is exactly doing.
  - > LCD Instructions
  - > Keypad Scan
- By throwing these chunks of code like these into header files, our main source file will effectively minimize substantially.



EEL 3744

How to Add & Include a Header File

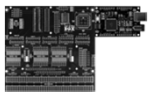
- Adding a header file:
  - > File > new > file > include file
  - > Save this file in a separate folder (maybe a folder called 3744_Headers)
- Including a header file:
  - > Right click on your project's name in the solution explorer
  - > Add > Existing Item > browse to your .h file you saved > add as link*
  - > Type `#include "your_file.h"` at the top of your main source file
- For me, I have to "add" the file to my project. You "should" be able to add as link, but it doesn't work on my Surface Pro 3. If anyone figures out a way around this bug, please let me know! But if you run into the same thing, just add the file.
- Note: if you DON'T add as link, any later changes to the header file will not affect the original, as adding directly just makes a duplicate file in your new project.



EEL 3744

Mixed C/Assembly

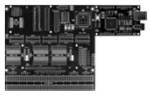
- The slides that follow are **NOT** covered this semester, i.e., you will not write mixed C/Assembly code, nor will you be responsible to know this for labs or exams



EEL 3744

C Projects and **.s** Assembly Files

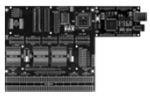
- When creating a C project in Atmel Studio, a simple **.c** file is created with a template structure
  > C code should be restricted to **.c** files
- When adding Assembly to a C project, a **.s** file is used to hold all Assembly code (**not** a **.asm** file)
- A **.s** file will resemble a standard assembly file, however, there are some considerations that must be made when in C projects
  > Registers are used differently since C also uses them
  > Assembly preprocessor directives have a different format



EEL 3744 ".s" File Compilation Registers

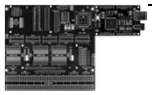
- When writing assembly in a C project, registers have different rules

Register	Description	Assembly code called from C	Assembly code that calls C code
r0	Temporary	Save and restore	Save and restore
r1	Always Zero	Must clear before returning	Must clear before returning
r2-r17 r28 r29	"call-saved"	Save and restore	Can freely use
r18-r27 r30 r31	"call-used"	Can freely use	Save and restore



EEL 3744 ".s" File Compilation Registers

- ***r0***: defined as a temporary register which may be used by compiler generated code
- ***r1***: assumed to always be zero by the compiler, so any assembly code that uses this should clear the register before calling compiler generated code
- ***r2 – r31***: defined as "call-saved" or "call-used"
  - > ***call-saved***: registers that a called C function may leave unaltered, however, assembly functions called from C should save and restore the contents of the register (using stack)
  - > ***call-used***: registers available for any code to use, but if calling a C function, these registers should be saved since compiler generated code will not attempt to save them

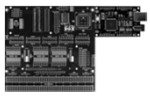


EEL 3744 ".s" File Compilation Syntax

- When writing assembly in a C project, some syntax is different

Path for the equivalent file to the ATxmega128A1Udef.inc is: C:\Program Files (x86)\Atmel\Atmel Toolchain\AVR8 GCC\Native\3.4.2.1002\avr8-gnu-toolchain\avr\include\avr\iox128a1u.h

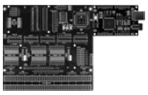
Atmel AVR	AVR-GCC
.include "ATxmega128A1Udef.inc"	#include <avr/io.h>
.dseg	.section .data
.cseg	.section .text
.db 1,2,3,4	.byte 1,2,3,4
.db "message"	.ascii "message"
.db "message", 0x00	.asciz "message"
.byte 37 ;save space for bytes	.ds.b 37
.dw	.word
HIGH(), LOW()	hi8(), lo8()



EEL 3744 ".s" File Compilation .dseg and Data Memory

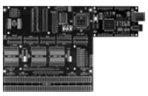
- Data memory defaults to start at 0x2000
- **.section .data** replaces the use of **.dseg** to access the Data Memory space
  - >Example:

```
.section  .data          // old way .dseg
Var1:     .ds.b   7      // save 7 bytes
Var2:     .ds.w   3      // save 3 words
Var3:     .byte   0x37   // Var3 = 0x37
// Previously, .byte saved space; now value
Text:     .asciz  "hello world"

.global __do_copy_data  // needed for Var3
                        // and Text
```



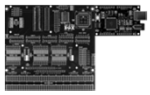
EEL 3744 ".s" File Compilation .dseg and Data Memory

- **.section .data** is necessary to begin the Data Memory (i.e., volatile memory = RAM) segment
- The **.asciz** command is used to define a **specific** null terminated string, a constant
  - > **.ascii** is like .asciz, but with no null termination
- The **ds.b** and **ds.w** commands are used to define **storage** of varying sizes (like .byte in .asm files)
- The **.byte** command is used to define a **specific** byte, i.e., a constant (like .db in .asm files)



EEL 3744 ".s" File Compilation .dseg and Data Memory

- Data memory is typically used to create storage of variables like **Var1** and **Var2**
- It is occasionally desired to create memory **and store an initial value** in that memory space, as we did for **Var3** and **Text**
  - > The initial value is stored in program memory
  - > The **.global __do_copy_data** special command handles copying the data from program memory to data memory

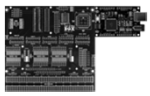


EEL 3744 “.s” File Compilation
.cseg and Program Memory

- Like Assembly, program memory is used to hold program constants and assembly code
- **.section .text** replaces the use of **.cseg** to access the Program Memory space
  - > Example:

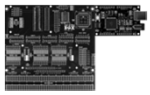
```
.section      .text
VA:     .byte  1, 2, 3, 4, 5, 6
VB:     .byte  0xA0,0xB0,0xC0,0xD0,0xE0,0xF0
.global MAIN_ASM  //Required for mixed

MAIN_ASM:
        ldi    R18, 6
        ...
```

VectorAdd_Mixed.s



EEL 3744 “.s” File Compilation
.cseg and Program Memory

- **.section .text** is shown to begin the Program Memory segment
- The **.byte** command is used just as it was under the Data Memory section
  - > May be used to defined multiple bytes in a section
  - > Saved in Program Memory, not desired to transfer to Data Memory (no need of **.global __do_copy_data)**
- The rest of the example follows standard Assembly



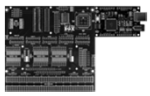
EEL 3744 .s File Program Memory & Watch Window

- In a .s file, we do **NOT** have to do the shifting that we did in .asm files for reading Program Memory Section (.dseg in .asm and .section .text in .s)
  - >Example for a .s file

`VectorAdd_Mixed.s`

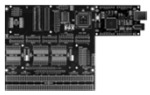
```
ldi ZL, lo8(VA)    // Load the address of program
ldi ZH, hi8(VA)    //    memory for VA
```

- The watch window can **NOT** display XL, XH, YL, YH, ZL, or ZH in .s files, nor most other things (other than registers, Rx)



EEL 3744 ".s" File Compilation Functions Example

- One of the main aspects of using Assembly in a C project is to benefit from using Assembly functions
- Functions must be declared in both C files and Assembly files
  - > Function prototypes should be defined in C code for any function called from Assembly
    - extern int funct();
  - > Functions defined in Assembly code that will be called from C code should be declared global
    - .global funct



EEL 3744 “.s” File Compilation

Functions in C/Assembly

> .c file syntax:

```
extern int funct();

int main(void)
{
   funct();
}
```

> .s file syntax:

```
.global funct

funct:
   ldi   R18, 0x47
   ...
   ret
```

> .c file example:

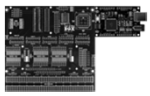
```
extern void MAIN_ASM();
int main(void)
{
   MAIN_ASM();
}
```

VectorAdd_Mixed.c

> .s file example:

```
.global MAIN_ASM
MAIN_ASM:
   ldi   R18, N
   ...
   ret
```

VectorAdd_Mixed.s



EEL 3744

The End!