Lab 3: Mixing Assembly and C codes

Revised by : Muhammad Imran (Developed by: Muhammad Amir Yousaf)

Date : 2014-11-18

Course : ET032G, Elektroteknik GR (A), Mikrodatorteknik at Mittuniversitetet, Sweden.

Often, C is a language of choice and assembly is included based on the requirements of the application. AVR GCC compiler for Atmel XMEGA controllers offers a feature of 'Inline Assembler' that allows a low level code in assembly language integration with high level C code in the same project.

This feature is useful in achieving:

- Optimizing time critical and the most performance-sensitive parts of an algorithm.
- Access to processor specific instructions.
- System calls.

With inline assembler high level language code is replaced by human written assembly code, allowing the programmer to use the full extent of his ingenuity, without being limited by a compiler's higher-level constructs.

Task 1:

In this task, you will see how c code is translated into assembly code which consists of mnemonics. These mnemonics are abbreviation for instruction names, symbols for representing variable, registers and constants.

The Disassembly window is only available when debugging. Enable it by selecting **Debug→Windows→Disassembly** or **Ctrl+Alt+D** during a debugging session.

```
#include <avr/io.h>
#include <asf.h>
int main (void)
{
            board_init();
            PORTB.DIR=0xFF;
            while(1)
            {
              PORTB.OUT = 0xA0;

            }
}
```

In the above code **PORTB.DIR=0xFF;** is translated into assembly code (**with no compiler optimization**);

```
LDI R24,0x20                Load immediate
LDI R25,0x06                Load immediate
SER R18                     Set Register
MOVW R30,R24                Copy register pair
STD Z+0,R18                 Store indirect with displacement
```

Each instruction requires certain amount of memory, cycles and use an addressing mode. See the XMEGA Instruction Set Nomenclature file and identify.

a. Number of bytes required for the above C code?
b. Number of clock cycles required for the code?

c. Types of addressing modes are used and
d. The OPCODE and OPERANTS?

Task 2:

In this task, you will mix C code with assembly code in the same AVRGCC project using the Studio IDE. A detailed document can be accessed online. The assembly code is written with .S extension in order allow the compiler to call assembler and linker as required. Commonly, the "main" is written in C code so that linker is aware of where to start but there is no special requirements for doing this. You can write subroutine named "main" and declared to be global (using the ".global" directive) will also produce a module named "main".

1. Keeping in view the above description, write a C code that intends to call the assembly language routine. The C code is required to have function prototype declared as external in order to call assembly language routine. For example

   extern unsigned char my_assembly_fct (unsigned char, unsigned int);

   The C code passes two unsinged character variables (Var1, Var2), one (Var1) with constant value and the other (Var2) with incremental values to the assembly subroutine (addition_routine).

2. After writing the C code, write sub-routine in an assembly file with extension .S and it should be called from a C code written in step1. The assembly language routine needs to be declared as global in the assembly code in order to make it visible to the C compiler. This is done using the using the ".global" directive:

   .global my_assembly_fct

   The language subroutine performs
   - addition,
   - clear the upper odd number register since addition results fits in 8 bits in this case
   - return the value and control from subroutine

   Note! Arguments in a fixed argument list are assigned, from left to right, to registers r25 through r8. All arguments use an even number of registers. This results in char arguments consuming two registers.
   Return values use the registers r25 through r18, depending on the size of the return value.

3. The return value in a third variable (Var3) is displayed on LCD. You can add delay function to see counter values on the LCD.

Write in the report, when it is good to mix the assembly code with C. List some applications with such requirements.

Task 3:

(a) Write an inline assembly code which uses virtual port O to turn on LEDs in the same fashion as Task1. In functionality, this code is exactly same as the code in Task1.
Calculate now the number of bytes and cycles required for this code and compare it with Task1 values.
(b) Increment the counter and use it to blink LEDs with some delay

TIPS for task3

The format for using inline assembly is given here with two commands. You need to replace the OPCODE, register and operands with suitable commands, registers and values. Note that register is also an operand but for sake of simplicity I wrote register.

```
asm volatile("OPCODE register, operands" "\n\t"
             "OPCODE operands, register" "\n\t"
 );
```

Page 140/410 and page 394/410 of the XMEGA B MANUAL provides the details on how to map a port associated with LED to virtual port.

Reference documents

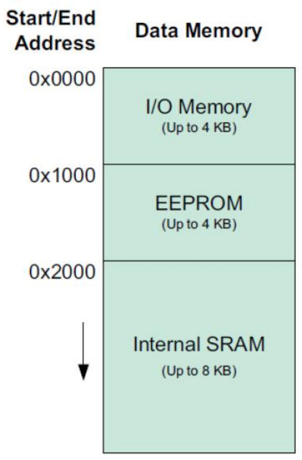
Using XMEGA IOs
Atmel AT1886: Mixing Assembly and C with AVRGCC
XMEGA B Manual.

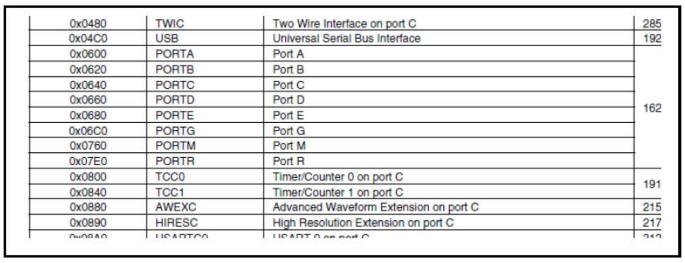
Appendix: Virtual Ports

Virtual Ports allow mapping of PORTS into IO memory space. Using instructions to access IO memory space instead of data space is both faster and consumes less program memory.

In AVR XMEGA128B1 we have 16KB data memory with 4KB IO memory which carries the status and configuration registers for peripherals and module including the CPU.

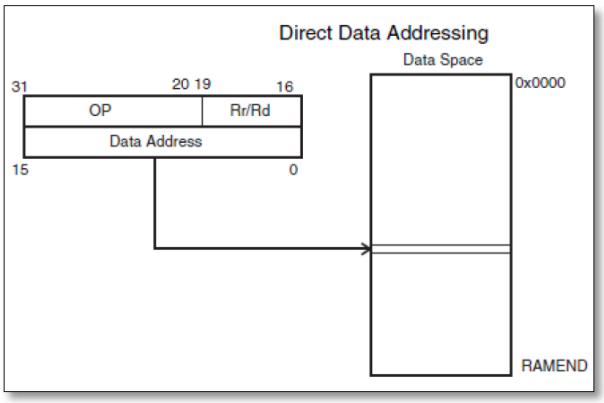
Data memory map.

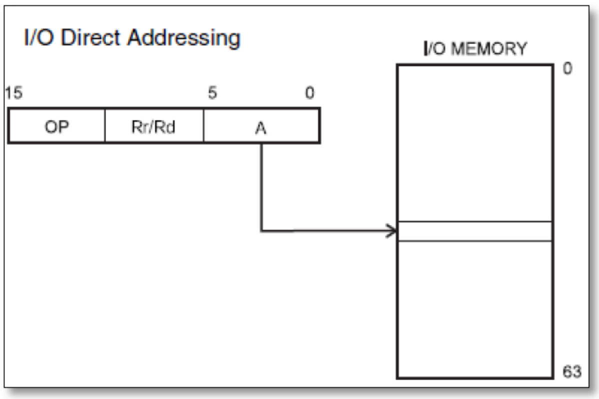




0x0480	TWIC	Two Wire Interface on port C	285
0x04C0	USB	Universal Serial Bus Interface	192
0x0600	PORTA	Port A	162
0x0620	PORTB	Port B	
0x0640	PORTC	Port C	
0x0660	PORTD	Port D	
0x0680	PORTE	Port E	
0x06C0	PORTG	Port G	
0x0760	PORTM	Port M	
0x07E0	PORTR	Port R	
0x0800	TCC0	Timer/Counter 0 on port C	191
0x0840	TCC1	Timer/Counter 1 on port C	
0x0880	AWEXC	Advanced Waveform Extension on port C	215
0x0890	HIRESC	High Resolution Extension on port C	217

All data space can be accessed by the Load (LD/LDS/LDD) and Store (ST/STS/STD) instructions which are used to transfer data between the 32 registers in register file and the data memory. Such instructions are categorized in 'Direct Data Addressing' instructions.





XMEGA controllers offer instructions with another addressing mode called ‘IO Direct Addressing’ that allow faster access the IO Memory with less program memory consumption. The address space in such instructions is only 6 bit i.e. they can only access up to first 64 bytes IO memory. Examples of such instructions are IN, OUT instructions.
Virtual Port registers allow IO port registers to be mapped virtually in first 64 bytes and hence make it possible to access these registers by fast and efficient ‘IO Direct Addressing’ instructions. There are four virtual ports, and so four ports can be mapped at the same time.