APPLICATION NOTE

Atmel AT1886: Mixing Assembly and C with AVRGCC

8-bit Atmel Microcontrollers

Features

- Easily mix C and assembly source in Atmel® Studio 6
  - C can call assembly routines
  - Assembly can call C functions
- Passes variables and pointers between C and assembly
- Allows C and assembly to share global variables

Introduction

This application note describes how to mix both C and assembly code in an AVRGCC project using Studio 6 IDE. This application note is written from the perspective that C is the language of choice and assembly language is included in situations where it is either necessary or convenient.

Studio 6 (and previous versions) will allow you to work with both assembly language and C files in the same project. The question is how does the software engineer organize the assembly code and C code? How are parameters, variables, and similar passed between the assembly code and the C code?

Table of Contents

1. Pre-requisites

The solution discussed in this document requires basic familiarity with the following skills and technologies. Please refer to Chapter 11 References to learn more.

- Atmel Studio 6
- Atmel debugger JTAGICE mkII or JTAGICE 3
- Atmel STK®600 Starter Kit

2. Project organization

There are no special requirements for organizing the project. Files identified as header files, whether C code or assembly, will be placed in the Header files folder of the project. Files identified as source files will be placed in the Source files folder of the project.

Although not strictly required, all assembly language source files should have an".S" extension. This will allow the C compiler frontend to automatically call the assembler and linker as needed. In addition, the C preprocessor will be invoked automatically allowing the user of symbolic constants.

One of the source files will need to produce a "main" module for the linker so that the linker knows where to start the application. The most common is a C code file with a function called "main". However, an assembly file with a subroutine named "main" and declared to be global (using the ".global" directive) will also produce a module named "main".

Combining C and assembly in a single project raises several questions depending on the needs of the application. These questions might include:

- How can an assembly routine be made visible to the C compiler such that a C function can call the assembly routine?
- Similarly, can a C function be made visible to an assembly routine such that the assembly routine can call the C function?
- How are variables passed to the assembly code?
- How are variables passed to the C function?
- Can the assembly code and C use the same global variables?

3. Function visibility

A C language function needs to be declared as external in the assembly code in order to be "seen" by the assembler:

```
.extern my_C_function
```

An assembly language routine needs to be declared as global in the assembly code in order to be visible to the C compiler. This is done using the using the ".global" directive:

```
.global my_assembly_fct
```

In addition, a C file that intends to call the assembly language routine will need to have a function prototype declaring the assembly language routine to be external:

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

4. Variables

Both the C code and the assembly code can access variables independently. As a practical matter it is advisable to let the C code manage the variables and pass parameters to the assembly code either by value or by reference. The Chapters 5 Register usage and 6 Parameter passing describes how the registers sets are used by the C compiler and how parameters are passed.

It is possible for both assembly and C to access the same global variable. Such a variable would need to be a global variable in the C code and declared as external in the assembly code. Consider a variable “my_value” intended to be global. In the C code it would be declared, outside of any function, like any other variable:

```
unsigned char my_value;
```

In the assembly it would be coded:

```
.extern my_value
```

5. Register usage

Writing assembly language routines to mix with C code requires knowledge of how the compiler uses the registers:

- **r0** is a temporary register and can be used by compiler generated code. If you write assembly code that uses this register and calls a C function, you will need to save and restore this register since the compiler might use it
- **r1** is assumed by the compiler to always contain zero. Assembly code that uses this register should clear the register before returning to or calling any compiler generated code
- **r2-r17, r28, r29** are “call-saved” registers meaning that calling C functions should leave these registers unaltered. An assembly language routine called from C that uses these registers will need to save and restore the contents of any of these registers it uses
- **r18-r27, r30, r31** are “call-used” registers meaning that the registers are available for any code to use. Assembly code that calls a C function will need to save any of these registers used by the assembly code since compiler generated code will not save any of these registers that it uses

Table 5-1 summarizes the register interfaces between C and assembly.

Table 5-1. Summary of the register interfaces between C and assembly.

Register	Description	Assembly code called from C	Assembly code that calls C code
r0	Temporary	Save and restore if using	Save and restore if using
r1	Always zero	Must clear before returning	Must clear before calling
r2-r17	“call-saved”	Save and restore if using	Can freely use
r28			
r29			
r18-r27	“call-used”	Can freely use	Save and restore if using
r0			
r31			

6. Parameter passing

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. Additional arguments beyond that which will fit in the registers are passed on the stack.

Arguments in a variable argument list are pushed on the stack in right to left order. Char arguments consume two bytes.

Return values use the registers r25 through r18, depending on the size of the return value. The relationship between the register and the byte order is shown in Table 6-1:

Table 6-1. Relationship between the register and the byte order.

Register	r19	r18	r21	r20	r23	r22	r25	r24
Byte order	b7	b6	b5	b4	b3	b2b	b1	b0

7. How to build and run the demo

The demo code includes simple functions to illustrate calling assembly language routines from within C and to pass parameters between the two languages. In addition, an assembly language interrupt service routine is included.

The code begins by calling the “change_clock” assembly language routine which changes the clock prescaler passing to the routine a new prescaler value.

The code then calls an assembly language routine to add two passed parameters together returning to C the result of the addition.

The code then initializes Timer 0 to produce a periodic interrupt and then the code stays in a “while” loop. An assembly language interrupt service routine toggles Port D, bit zero when an interrupt occurs.

8. Required hardware

- An Atmel STK600 starter kit
- STK600-ATmega2560 card (included with the STK600)
- An Atmel debugger (JTAGICE mkII, JTAGICE 3, Atmel AVR Dragon™, etc)
- Atmel Studio 6
- 10-pin IDC ribbon cable (included with the STK600)

9. Hardware setup

Make the following connections on the STK600 board:

- Mount the STK600-ATmega2560 card on the STK600
- Connect the 10-pin ribbon cable between the PORTD and LED headers
- Connect the debugger’s cable to the device’s JTAG header on the STK600

10. Procedure

1. Download the AT1886.zip file from www.atmel.com.
2. Unpack the files into a working directory.
3. Open the AVR1886.atsln project file with Atmel Studio 6.
4. Choose your debugger tool in the Studio box.
5. Specify the ATmega2560 as the target device.
6. Build the project.
7. Enter debug mode.
8. Run the application.
9. Add “val_1”, “val_2”, and “val_3” to the watch window.
10. Set a breakpoint on the “val_1 =” assignment in main.
11. In the Processor tab on the right, expand the registers view.
12. Click the run button. The code should execute the “change_clock” function and then break.
13. Use the single step function to step through the next instructions until you reach the “init_timer_0” function call. You should be able to observe the parameter get passed via registers from C to the assembly routine “add_two”.
14. Once you reach the “init timer_0” function, click the run button. This will initialize the timer using C and enable a periodic interrupt from the timer. The interrupt service is in assembly. You should see the LED on PD.0 blink.

11. References

1. Atmel Studio 6 – www.atmel.com
2. Atmel AVR JTAGICE mkII – www.atmel.com
3. Atmel JTAGICE 3 – www.atmel.com
4. Atmel STK600 - www.atmel.com

12. Revision history

Doc. Rev.	Date	Comments
42055B	11/2012	The document is renamed from AVR1886 to AT1886
42055A	11/2012	Initial document release