# Programming the Atmel ATmega32 in C and assembly using gcc and AVRStudio

by

Dr.-Ing. Joerg Mossbrucker
EECS Department
Milwaukee School of Engineering

March 6, 2006

updated Dec. 08 2009

This manual covers the following:

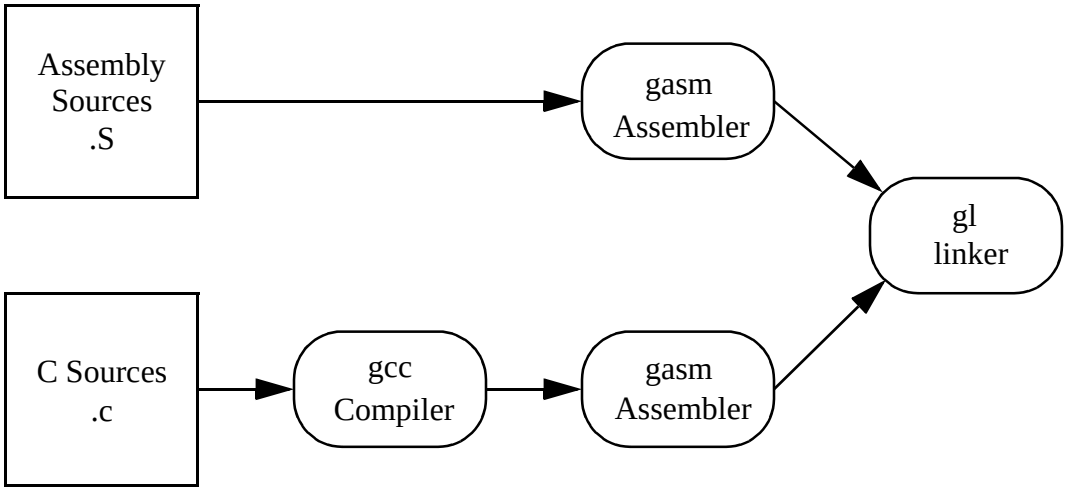
- Projects with multiple sources in C and assembly
- Register usage by gcc
- Interrupt handlers written in C and assembly

## 1 Project layout

This manual assumes familiarity with AVRStudio and the gcc plug-in compiler. The example project consists of two sources, csource.c and assembly.S. The file extension determines the type of source:

- .c C-source file
- .S Assembly source (must be uppercase)

Both sources can be directly inserted into the project structure in AVRStudio. The project MUST be defined as a gcc project. This disables the build-in Atmel assembler and ALL sources are handled by the plug-in (gcc).




All sources in C are compiled BEFORE assembly language sources are assembled. This has the following consequences:

- C functions called from assembly do not need to be declared inside the assembly source, since they are already in the symbol table
- Global variables should be declared in C source files (where they are automatically initialized to 0 or nil)
- Assembly functions called by C sources must be declared external in C sources and global in assembly sources
- Interrupt handlers in assembly must be declared global in assembly sources (interrupt handlers in C MUST NOT be declared); interrupt handlers are NOT spell-checked, misspelled interrupt handlers might not be caught by the linker -> causes RESET upon interrupt

Example sources:

```
1:
2:
3: // csource.c
4:
5:
6:
7: #include <avr/io.h>
8: #include <inttypes.h>
9: #include <avr/interrupt.h>
10: #include <stdlib.h>
11:
12:
13: extern uint8_t asmfunction(uint8_t);
14:
15: void cfunction(uint8_t);
16: uint8_t variable;
17:
18: int main(void)
19: {
20:     uint8_t value;
21:     uint16_t time;
22:
23:     variable=12;
24:     value=0;
25:     DDRB=0xFF;
26:
27:     sei();  // enables interrupts globally
28:
29:     for(;;)
30:     {
31:             PORTB=value;
32:             value++;
33:             for(time=0;time<4;time++)
34:             {
35:                     value=value;
36:             }
37:
38:             value=asmfunction(value);
39:             cfunction(value);
40:     }
41:
42:     cli();  // disables interrupts
43: }
44:
45: void cfunction(uint8_t passing)
46: {
47:     PORTB=passing;
48: }
49:
50:
51: ISR(SIG_OVERFLOW0) // C handler for timer0
52:                    //overflow interrupt
53: // remember that SIGNAL does NOT re-enable
54: // interrupts inside the handler
55: // If you need a re-entrant handler then
56: // use INTERRUPT instead
57: {
58:     PORTB = 10;
59:     TCNT0 = 50;
60: }
61:
62:
```

```
63:
64:
65: // assebmly.S
66:
67: // Remember that every assembly source file
68: // must have the suffix .S
69: // (upper case) since it is processed by GCC
70: // and NOT the Atmel assembler
71:
72: #define _SFR_ASM_COMPAT 1
73: // this is needed for every assembly source
74: // file so we can use
75: // the IO names as usual
76:
77: #define __SFR_OFFSET 0
78: // this is needed for every assembly source
79: // file so we can use IO with IN and OUT
80: // statements
81:
82: #include <avr/io.h>
83: // since this file is processed by the
84: // C-preprocessor
85: // comments after #include or #define
86: // must be preceded by //
87:
88: ; afterwards we can use the traditional
89: ; semicolon
90:
91: .global asmfunction       ; all assembly labels
92:                           ; which are called
93: ; from C need to be defined global
94:
95: asmfunction:      ; entry point for
96:                   ; assembly function
97:
98:                   out PORTB,r24
99:                   call   cfunction
100: ; cfunction does not need to be
101: ; declared here, the linker will find
102: ; it (no type checking however)
103:
104:                  out    PORTB,r23
105:
106:                  lds    r14,variable
107:
108:                  out    PORTB,r22
109:
110:                  mov r24,r14
111:
112:                  ret
113:
114: .global SIG_OVERFLOW1
115: SIG_OVERFLOW1:
116:                  out    PORTB,r20
117:                  out PORTB,r19
118:                  reti
119:
120:
121:
122:
123:
124:
```

Comments:

- C sources must have extension .c, assembly source must have extension .S (upper case)
- 9: must be included if interrupt handlers are needed
- 13: all assembly subroutines must be declared external to the C source
- 16: global variables; initialized to 0 automatically upon start-up (C99 standard)
- 27: enables globally all interrupts, uses inline assembly
- 38: assembly function call with parameter and return; see below for location of parameters
- 42: disables globally all interrupts, uses inline assembly
- 51: interrupt handler in C for timer0 overflow interrupt; please note that names for interrupt handlers are NOT spell-checked (the linker might not even catch it either)
- 65: assembly sources are pre-processed by the C-preprocessor, therefore comments must be preceded by a // or surrounded by /* and */ until line 82
- 72: is needed for the gcc assembler so we can use IO names as usual
- 77: is needed so the gcc assembler can IO with in and out statements
- 82: is needed for all IO names; after this include we can use the ; as the traditional comment separator
- 91: all assembly labels which are called from C must be declared global
- 95: label of assembly subroutine
- 99: assembly calls C-function; note that C-functions called from assembly do not have to be declared, since they are already compiled and hence the linker “knows” of them. However, absolutely no parameter and return checking is done for function calls between assembly and C in either way; the linker catches unresolved references though
- 106: global variables can also be accessed in assembly
- 112: assembly subroutine terminates with a ret
- 114: all assembly interrupt handlers also need to be declared global
- 115: interrupt handler for timer1 overflow interrupt; same types available as in C (see comment for 51)
- 118: interrupt handlers in assembly terminate with reti

## 2 Register usage by gcc

Data types: char is 8 bits, int is 16 bits, long is 32 bits, long long is 64 bits, float and double are 32 bits (this is the only supported floating point format), pointers are 16 bits (function pointers are word addresses, to allow addressing the whole 128K program memory space on the ATmega devices with > 64 KB of flash).

Call-used registers (r18-r27, r30-r31): May be allocated by gcc for local data. You may use them freely in assembler subroutines. Calling C subroutines can clobber any of them - the caller is responsible for saving and restoring.

Call-saved registers (r2-r17, r28-r29): May be allocated by gcc for local data. Calling C subroutines leaves them unchanged. Assembler subroutines are responsible for saving and restoring these registers, if changed. r29:r28 (Y pointer) is used as a frame pointer (points to local data on stack) if necessary.

Fixed registers (r0, r1): Never allocated by gcc for local data, but often used for fixed purposes:

r0 - temporary register, can be clobbered by any C code (except interrupt handlers which save it), may be used to remember something for a while within one piece of assembler code

r1 - assumed to be always zero in any C code, may be used to remember something for a while within one piece of assembler code, but must then be cleared after use (clr r1). This includes any use of the [f]mul[s[u]] instructions, which return their result in r1:r0. Interrupt handlers save and clear r1 on entry, and restore r1 on exit (in case it was non-zero).

Arguments - allocated left to right, r25 to r8. All arguments are aligned to start in even-numbered registers (odd-sized arguments, including char, have one free register above them). This allows making better use of the movw instruction on the enhanced core. If too many, those that don’t fit are passed on the stack.

Return values: 8-bit in r24 (not r25!), 16-bit in r25:r24, up to 32 bits in r22-r25, up to 64 bits in r18-r25. 8-bit return values are zero/sign-extended to 16 bits by the caller (unsigned char is more efficient than signed char - just clr r25). Arguments to functions with variable argument lists (printf etc.) are all passed on stack, and char is extended to int.