Microcontroller Programming

ASM & C/C++

(@AVR Studio)

Assembler in AVR Studio

- Built-in assembler (AVRASM2)
- AVR GCC plugin / Independent Assembler module (separate source file, .s)
- AVR GCC plugin / Inline in C code

- be careful …

C/C++ in AVR Studio

- AVR Studio 5+ native C/C++ compiler (GCC based)
  - GCC C ASF Board Project
  - GCC C Executable Project
  - GCC C Static Library Project
    - avr-gcc.exe
  - GCC C++ Executable Project
  - GCC C++ Static Library Project
    - avr-g++.exe
  - Create project from Arduino sketch
- AVR Studio 4+ AVR-GCC via WinAVR
- be careful …

Built-in Assembler

```
.include "m32def.inc"
.cseg
.org 0x2A
main:
...
```

Assembler module via AVR GCC plugin

```
#include <avr/io.h>
.section .text
.org 0x2A
.global main
main:
…



.extern something
```

Inline assembler via AVR GCC plugin

```
asm volatile("cli"::);

asm("in %0, %1" : "=r" (value) : "I"
(_SFR_IO_ADDR(PORTD)) );

asm("in %[retval], %[port]" : [retval]
"=r" (value) : [port] "I"
(_SFR_IO_ADDR(PORTD)) );
```

asm() statement

```
asm(code : output_operand_list :
      input_operand_list [: clobber_list]);
```

- very similar if writing “pure” assembler programs
- different if referring to / interacting with C

- **`code`** may contain many instructions, separated by **`\n`**

- **`asm volatile(`**…

asm() code part

```
asm("in %0, %1" : "=r" (value) : "I" (_SFR_IO_ADDR(PORTD)) );
asm("in %[retval], %[port]" : [retval] "=r" (value) : [port] "I"
(_SFR_IO_ADDR(PORTD)) );
```

- % followed by reference
  - single digit
  - [name]
- special registers
  - __SREG__ __SP_H_ __SP_L_ __tmp_reg__ __zero_reg__

asm() input operand list
asm() output operand list

```
asm("in %0, %1" : "=r" (value) : "I" (_SFR_IO_ADDR(PORTD)) );
asm("in %[retval], %[port]" : [retval] "=r" (value) : [port] "I"
(_SFR_IO_ADDR(PORTD)) );
```

- constraint string
  - modifier (= + &)
  - constraint
    - analogous to ASM instruction parameter type
- (C expression)
  - lvalue

asm() clobber list

```
asm("in %0, %1" : "=r" (value) : "I‟ (_SFR_IO_ADDR(PORTD)) );
asm("in %[retval], %[port]" : [retval] "=r" (value) : [port] "I"
(_SFR_IO_ADDR(PORTD)) );
```

- list of registers which are used in the code and are not input / output parameters

ASM / C coexistence

- ASM code may call functions written in C
- C functions may call functions written in ASM

- It is essential to understand compiler task & way of working
- Accessibility:

  functions: ASM: .extern my_C_function

  .global my_ASM_function

  C: extern whatever my_ASM_function(whatever)

  variables: ASM: .extern C_variable

  BE CAREFUL WITH ATMEL STUDIO 7 ☹

Function call conventions

Fixed Argument Lists:

- arguments allocated left to right into r25-r8
  - rest onto stack, but this should be avoided
- all arguments take even number of registers
  - even single-byte

Function call conventions

Variable Argument Lists:

- arguments allocated on stack
- pushed right to left
- char extended to int (hi=0)

Function call conventions

Return value

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

Variables

char	8 bits
int	16 bits
long	32 bits
long long	64 bits
float and double	32 bits

this is the only supported floating point format

pointers	16 bits

function pointers are word addresses, to allow addressing up to 128K program memory space

Register Usage

- r0
  - temporary
  - not preserved by generated code
  - preserved by generated IRQ routines
- r1
  - always 0 (zero)
  - preserved by generated IRQ routines
- r2-r17, r28-r29(Y)
  - storage
  - must be preserved (“Call saved”)
- r18-r27, r30-31(Z)
  - grabs, local data
  - may be changed (“Call used”)

Cookbook

for writing assembly code called from C

- you *must* preserve r2-r17,r28-r29 (if you use them)
  - push at start / pop before return
- parameters are passed using r25-r8
- results must be returned using r25-r18
  - single byte in r24, *not in r25*
- C compiler expects r1=0, be sure to clear it if you use it
- make your functions public

Cookbook

for calling C functions from assembly code

- pass parameters as described above
- if you use r18-r27, r30-r31 in your code, preserve it by yourself
- make your functions public

Fixed address relocation

Code:

- function must be put in named section
- use __*attribute*__ *section* in function declaration

  void boot(void) __attribute__ ((section (".bootloader")));

Linker:

- define section starting address

  -Wl,--section-start=.bootloader=0x1E000

Fixed register usage

- binding variable to a specific general register

  register unsigned char counter asm("r3");
- safe for r2-r7
- r8-r15 may not be safe as they might be used for argument passing
  - safe if not too many / long parameters are passed

Variable access optimization

```
uint8_t flag;
...
ISR(SOME_vect) {
   flag = 1;
}
...

while (flag == 0) {
     ...}                    // ☹  but why?
```

volatile variables

```
volatile uint8_t flag;
...
ISR(SOME_vect) {
   flag = 1;
}
...

while (flag == 0) {
     ...                    // ☺
}
```

global / static variables

- initialized to 0 based on C standard
  - appropriate code goes to section .init4
  - “somewhat simplified”
  - *uninitialized* variables go to .bss section

    and do not consume space in object code
  - *initialized variables* go to .data section

    and consume space both in object code *and in flash ROM*
    - *Variables should only be explicitly initialized if the initial value is non-zero*

Interrupt handlers

- void f () __attribute__ ((interrupt ("IRQ")));
- void __attribute__ ((interrupt)) f ();
  - compiler will generate function entry and exit sequences suitable for use in an interrupt handler
  - interrupts will be enabled inside the function

- ISR(*vectorNo*){…}

```
ISR(USART0_RX_vect){
      if(num<BUFSIZE) {
                buffer[tail] = UDR0;
                tail = (tail+1)%(BUFSIZE-1);
                num++;
      }
}
```

Good practices – variables

- Preferable 8 bit unsigned
- Local preferred to global
  - For global, use static wherever possible
  - For local, disregard static

Good practices – pointers

```
char * ptr1, *ptr2;
…
*++ptr1 = *ptr2--;    // ☹
  ADIW R30, 0x01
  LD R24, X
  ST Z, R24
  SBIW r26, 0x01
*ptr1++ = *--ptr2;    // ☺
  LD R24, -X
  ST Z+, R24
```

Good practices – Loops

- for(;;){}
- do{}while(expr) is more efficient than while(expr){} and for(expr1;expr2;expr3)
- Use pre-decrement for counters
- Use loop jamming
- Unroll loops

Good practices – functions vs macros

- Single-shot functions ⇨ inline/macro
- Small functions ⇨ macro
- Use static functions

Good practices – assembly

- Well coded assembly instructions are always the best optimized code.

Good practices – conditional statements

- For if-else put the most probable first
- For multiple decisions, use switch-case instead of if-else sequences
- If using if-else long sequences, minimize depth

18 Hints to Reduce Code Size

1. Compile with full size optimization.
2. Use local variables whenever possible.
3. Use the smallest applicable data type. Use unsigned if applicable.
4. If a non-local variable is only referenced within one function, it should be declared static.
5. Collect non-local data in structures whenever natural. This increases the possibility of indirect addressing without pointer reload.
6. Use pointers with offset or declare structures to access memory mapped I/O.
7. Use for(;;) { } for eternal loops.
8. Use do { } while(expression) if applicable.
9. Use descending loop counters and pre-decrement if applicable.
10. Access I/O memory directly (i.e., do not use pointers).
11. Declare main as C_task if not called from anywhere in the program.
12. Use macros instead of functions for tasks that generates less than 2-3 lines assembly code.
13. Reduce the size of the Interrupt Vector segment (INTVEC) to what is actually needed by the application. Alternatively, concatenate all the CODE segments into one declaration and it will be done automatically.
14. Code reuse is intra-modular. Collect several functions in one module (i.e., in one file) to increase code reuse factor.
15. In some cases, full speed optimization results in lower code size than full size optimization. Compile on a module by module basis to investigate what gives the best result.
16. Optimize C_startup to not initialize unused segments (i.e., IDATA0 or IDATA1 if all variables are *tiny* or *small).*
17. If possible, avoid calling functions from inside the interrupt routine.
18. Use the smallest possible memory model.

5 Hints to Reduce RAM Requirements

1. All constants and literals should be placed in Flash by using the Flash keyword.
2. Avoid using global variables if the variables are local in nature. This also saves code space. Local variables are allocated from the stack dynamically and are removed when the function goes out of scope.
3. If using large functions with variables with a limited lifetime within the function, the use of subscopes can be beneficial.
4. Get good estimates of the sizes of the software Stack and return Stack (Linker File).
5. Do not waste space for the IDATA0 and UDATA0 segments unless you are using tiny variables (Linker File).

Hints – Further reading

AVR035 Efficient C Coding for 8-bit AVR microcontrollers

AVR4027 Tips and Tricks to Optimize Your C Code for 8-bit AVR Microcontrollers

Atmel AT1886: Mixing Assembly and C with AVRGCC [APPLICATION NOTE]

http://www.nongnu.org/avr-libc/user-manual

- The AVR Libc package provides a subset of the standard C library for Atmel AVR 8-bit RISC microcontrollers. In addition, the library provides the basic startup code needed by most applications.

http://www.nongnu.org/avr-libc/user-manual/FAQ.html

- reentrant code
- compiler optimization flags
- string storage
- external RAM usage

http://gcc.gnu.org/onlinedocs/gcc/

Atmel Studio, gcc and AVR

- gcc originally for von Neumann architecture
- AVR is Harvard architecture with separated instruction and data memories

- “nifty tricks” used

  These peculiarities have been abstracted away by the GUI, but users will see the truth when building projects with relocated segments.
  - .text segment starts at 0x0.
  - .data segment starts at 0x800000.
  - .eeprom segment starts at 0x810000.

Atmel Studio
C/C++ GCC … Project

- For C/C++ project, Atmel prepares a project, which allows to use .s and .c (.cpp) modules

- Executable projects expect main() function to be the entry point
- Library Project doesn’t link to elf/hex

Atmel Studio
AVR Assembler Project

- Project allows only for one assembler source
- pre-prepared structure
- “modules” have to be included in the main one