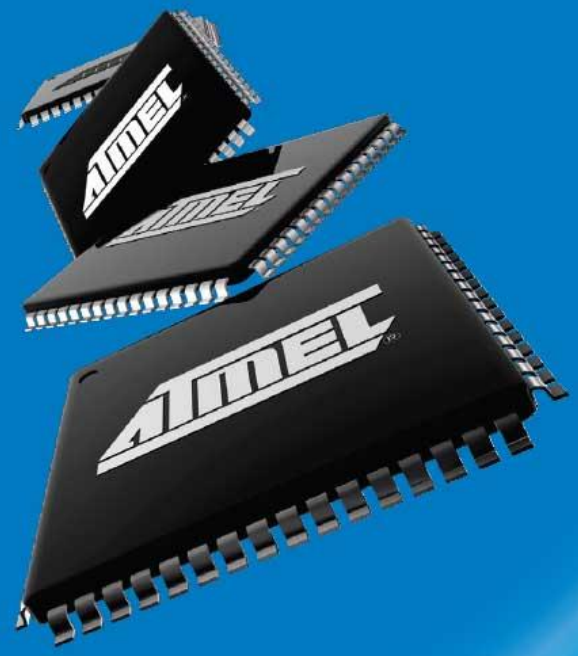
AVR®

8-bit Microcontrollers

AVR®32

32-bit Microcontrollers and Application Processors



AVR Subroutine Basics

February 2009

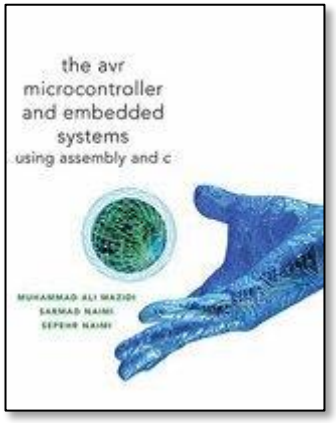
Everywhere You Are®

AVR Subroutine Basics

READING

The AVR Microcontroller and Embedded Systems using Assembly and C)
by Muhammad Ali Mazidi, Sarmad Naimi, and Sepehr Naimi

Chapter 3: Branch, Call, and Time Delay Loop, pages 118 to 125



CONTENTS

- How do I go to and return from a subroutine?

```
rcall   label
call    label
icall   label
ret
```

- AVR Call Addressing Modes

Relative	The relative address is encoded in the machine instruction using 12 bits. Assuming that the Program Counter (PC) is pointing at *the next instruction to be executed*, a relative call can jump within a range of -2^{n-1} to $2^{n-1} - 1$ program words, in other words $-2K \leq PC < 2K - 1$ n = 12 bits, $K = 2^{10} = 1024$, and a program word is 16-bits.
Long	*full* 16 K word (32K byte) address space
Indirect	*full* 16 K word (32K byte) address space

- Why Subroutines?
- My Little Subroutine Dictionary
- Assembly Subroutine Template
- How to Send Information to and/or from the Calling Program
- Rules for Working with Subroutines

WHY SUBROUTINES?

- Divide and Conquer – Allow you to focus on one small “chunk” of the problem at a time.
- Code Organization – Gives the code organization and structure. A small step into the world of object-oriented programming.
- Modular and Hierarchical Design – Moves information about the program at the appropriate level of detail.
- Code Readability – Allows others to read and understand the program in digestible “bites” instead of all at once. Higher level subroutines with many lower level subroutine calls take on the appearance of a high level language.
- Encapsulation – Insulates the rest of the program from changes made within a procedure.
- Team Development – Helps multiple programmers to work on the program in parallel; a first step to configuration control. Allows a programmer to continue writing his code, independent of other team members by introducing “stub” subroutines. A stub subroutine may be as simple as the subroutine label followed by a return instruction.

My Little Subroutine Dictionary

Subroutine versus Function[1]

- Functions and subroutines are the most basic building block you can use to organize your code.
- Functions are very similar to subroutines; their syntax is nearly identical, and they can both perform the same actions. However, Functions return a value to the code that called it.
- For this course the terms Subroutine, Procedure and Method may describe a Subroutine or Function based on context.

Parameter versus Argument[2]

- In everyday usage, "parameter" and "argument" are used interchangeably to refer to the things that you use to define and call methods or functions.
- Often this interchangeability doesn't cause ambiguity. It should be noted, though, that conventionally, they refer to different things.
- A "parameter" is the thing used to *define* a method or function while an "argument" is the thing you use to *call* a method or function.

  Parameter:

  ```
  void mySubroutine (uint8_t N){ … } ← N is a parameter
  ```

  Argument:

  ```
  uint8_t X
  X = 10
  mySubroutine(X)   ← X is an argument
  ```

- Ultimately, it doesn't really matter what you say. People will understand from the context.

[1] Source: http://www.codeproject.com/KB/aspnet/VBnet_Methods.aspx
[2] Source: http://project.ioni.st/post/790

ASSEMBLY SUBROUTINE TEMPLATE

```
; ---- My Subroutine -------
; Called from Somewhere
; Input: Registers, SRAM variables, or I/O registers
; Outputs: None for a subroutine or r25:r24 register pair for a C function
; No others registers or flags are modified by this subroutine
; --------------------------
MySubroutine:
   push    r15          // push any flags or registers modified by the procedure
   in      r15,SREG
   push    r16

   my assembly code

endMySubroutine:
   clr     r25          // zero-extended to 16-bits for C++ call (optional)
   pop     r16          // pop any flags or registers placed on the stack
   out     SREG,r15
   pop     r15
   ret
```

How to Send Information to and/or from the Calling Program

There are many way to send information to and from a subroutine or function. Here are a few…

- In Register(s) or Register Pair(s) agreed upon between the calling program and Procedure or Function.
- By setting or clearing one of the bits in SREG (I, **T**, H, S, V, N, Z, **C**).
- In an SRAM variable, *this method is not recommended*.
- As part of a Stack Frame, *this method is beyond the scope of a course on microcontrollers but is highly recommended*.

How to Send Information to and/or from Your C Program

When working in a Mixed C and Assembly programming environment, our subroutines and functions communicate using Register Pairs.

- Mixed C and Assembly parameter passing Register Pairs

  In your C Program…

```
// C Assembly External Declarations
extern void mySubr(uint8_t param1, uint16_t param2, uint16_t param3);

extern uint8_t myFunc(uint8_t param1, uint16_t param2, uint16_t param3);
```

  In your Assembly Program…

```
; Define Assembly Directives
.DEF   parm1H = r25
.DEF   parm1L = r24
.DEF   parm2H = r23
.DEF   parm2L = r22
.DEF   parm3H = r21
.DEF   parm3L = r20

mySubr:
     Assembly Code
       ret
```

- 8-bit return values (`uint8_t` data type) are zero/sign-extended to 16-bits in `r25:r24` by called function.

RULES FOR WORKING WITH SUBROUTINES

Here are a few rules to remember when writing your main program and subroutines.

- **Always disable interrupts and initialize the stack pointer at the beginning of your program.**

```
; Disable interrupts and configure stack pointer for 328P
   cli

   ldi     r16,low(RAMEND) // RAMEND address 0x08ff
   out     SPL,r16         // Stack Pointer Low SPL at i/o address 0x3d
   ldi     r16,high(RAMEND)
   out     SPH,r16         // Stack Pointer High SPH at i/o address 0x3e
```

- **Always initialize variables and registers at the beginning of your program. Do not re-initialize I/O registers used to configure the GPIO ports or other subsystems within a loop or a subroutine. For example, you only need to configure the port pins assigned to the switches as inputs with pull-up resistors once.**

- **Push (`push r7`) any registers modified by the subroutine** at the beginning of the subroutine and pop (`pop r7`) in reverse order the registers at the end of the subroutine. This rule does not apply if you are using one of the registers or SREG flags to return a value to the calling program. Comments should clearly identify which registers are modified by the subroutine.

- **You cannot save the Status Register SREG directly onto the stack. Instead, first push one of the 32 registers on the stack and then save SREG in this register. Reverse the sequence at the end of the subroutine.**

```
push    r15
in      r15, SREG
:
out     SREG, r15
pop     r15
```

Rules for Working with Subroutines – Continued –

- **Never jump into a subroutine**. Use a call instruction (`rcall, call`) to start executing code at the beginning of a subroutine.
- **Never jump out of a subroutine**. Your subroutine should contain a single return (`ret`) instruction as the last instruction (`ret` = last instruction).
- You do not need an `.ORG` assembly directive. As long as the previous code segment ends correctly (`rjmp, ret, reti`) your subroutine can start at the next address.
- You do not need to clear a register or any variable for that matter before you write to it.

```
~~clr   r16~~          ; this line is not required
lds   r16, A
```

- All blocks of code within the subroutine or Interrupt Service Routine (ISR) should exit the subroutine through the `pop` instructions and the return (`ret, reti`).
- It is a *good programming practice* to include only one return instruction (`ret, reti`) located at the end of the subroutine.
- Once again, never jump into or out of a subroutine from the main program, an interrupt service routine, or any other subroutine. However, subroutines or ISRs may call (`rcall`) other subroutines.

BASIC STRUCTURE OF A SUBROUTINE – *A REVIEW*

1. Load argument(s) into input registers (parameters) as specified in the header of the subroutine (typically r24, r22).
2. Call the Subroutine
3. Save an image of the calling programs CPU state by pushing all registers modified by the subroutine, including saving SREG to a register.
4. Do something with the return value(s) stored in the output register(s) specified in the header of the subroutine (typically r24, r22).
5. Restore image of the calling programs CPU state by popping all registers modified by the subroutine, including loading SREG from a register.
6. Return