Assembly Subroutine Guide.

What registers are used by the AVR C compiler?

•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 up to 128K program memory space).

•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. The requirement for the callee to save/preserve the contents of these registers even applies in situations where the compiler assigns them for argument passing.

•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).

•Function call conventions:

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 called function (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.

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.
- 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 (eg. push r7) any registers modified by the subroutine at the beginning of the subroutine and pop (eg. 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
```

- 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.