Macros for AVR assembler programming

Reference manual

Release: 4.1
March 21, 2019

For new releases and source files refer to:
https://www.realcomputerprojects.dev
https://www.controllersandpcs.de/projects

1. Purpose

- Toolbox to ease assembler programming of AVR microcontrollers.
- Compensate for peculiarities and eccentricities of the AVR architecture.
- Example to demonstrate the general approach of small virtual machines and purpose-written macros.

2. The virtual machine

It has three working registers A, B, C and three address registers X, Y, Z (Fig. 1, Tables 1 and 2). These registers consist of two bytes each, which can be accessed separately. Register A is the accumulator. Register B typically receives the second operand. Register C is mainly used for counting and auxiliary functions. The address registers X, Y, Z belong to the basic AVR architecture. Additionally, the macros may access a general working register TEMP, the registers R0 and R1, the stack pointer (SP) and the status register (SREG).

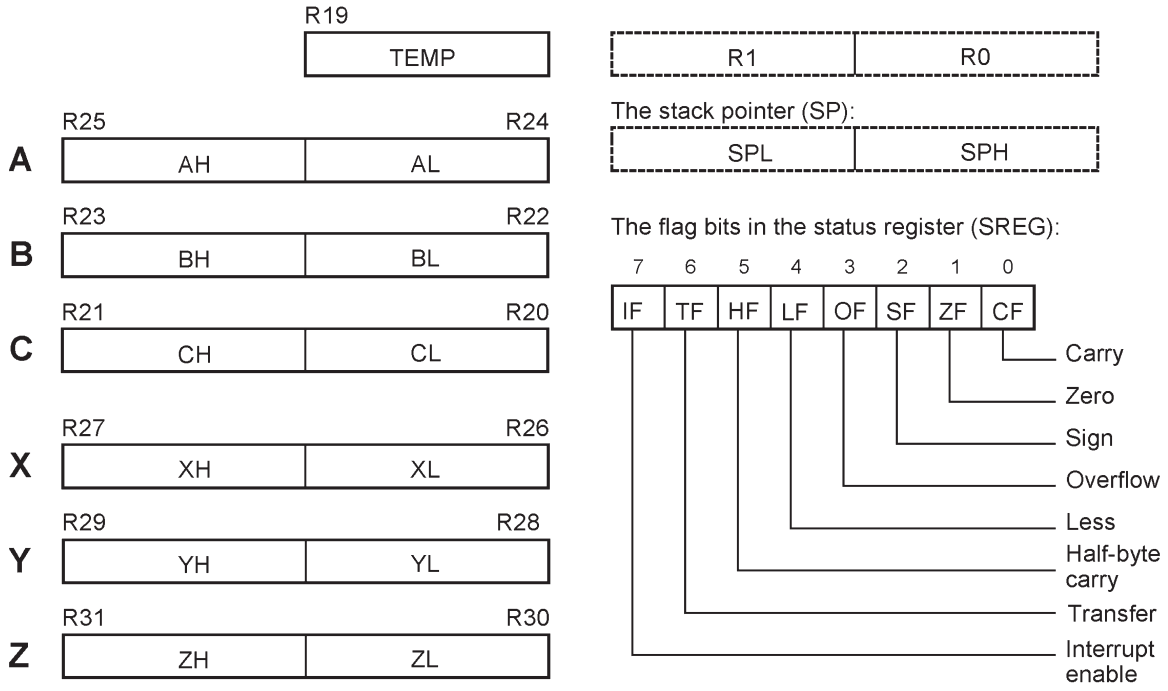


Fig. 1 Registers and flag bits of the virtual machine.

Bit[1]	AVR[2]	Meaning	Bit[1]	AVR[2]	Meaning
CF	C	Carry Flag. Carry-out (addition) or borrow-out (subtraction)	LF	S	Less Flag. LF = OF xor SF. Indicates that A < B, if A – B has been calculated
ZF	Z	Zero Flag. Result = 0	HF	H	Half-byte Flag. Carry out of low-order 4-bit nibble
SF	N	Sign Flag. Negative number	TF	T	Transfer Flag. Bit buffer for the bit-transfer instructions BLD, BST
OF	V	Overflow Flag. 2's complement overflow	IF	I	Interrupt Flag. Interrupts enabled

1. Flag mnemonics used in the macros (similar to some well-established architectures).
2. Bit designations in the manufacturer's literature.

Table 1 The flag bits. The manufacturer's bit designations (like C or Z) have been replaced by mnemonics (like ZF or CF) that are closer to the industry standards set by venerable, well-established architectures.

Register	Use	Register	Use
R0	Low-order byte of a multiplication result	R16	
R1	High-order byte of a multiplication result	R17	
R2	CX3. Extending register C to 24 bits (if required)	R18	
R3		R19	TEMP. General-purpose auxiliary and working register
R4		R20	C, CL. 3rd operand register; counting register
R5		R21	CH
R6		R22	B, BH. 2nd operand register
R7		R23	BH
R8		R24	A, AL. Accumulator
R9		R25	AH

Register	Use	Register	Use
R10		R26	X, XL. 1st address register<
R11		R27	XH
R12		R28	Y, YL. 2nd address register
R13		R29	YH
R14		R30	Z, ZL. 3rd address register
R15		R31	ZH

Table 2 The general-purpose register file of the AVR architecture and the register model of the virtual machine.

Principles of register allocation:

- The registers R3 to R18 are freely available.
- The address registers X, Y, Z are intrinsic to the AVR architecture.
- The accumulator register A and the address registers X, Y, Z can be operated upon with immediate word operands (instructions ADIW, SBIW). Therefore, the register A must be assigned to the registers R24, R25.
- The registers B, C, and TEMP could be allocated freely to the registers R16 to R23 (i.e., registers accessible by immediate instructions).
- AL, BL, CL, XL, YL, ZL are the low-oder bytes of the corresponding 16-bit registers. Their register addresses must be even.
- AH, BH, CH, XH, YH, ZH are the high-oder bytes of the corresponding 16-bit registers. Their register addresses must be odd.
- Some macros use the register R0 and R1 as auxiliary working registers.
- TEMP is the general working and auxiliary register. It may be used outside the macros. However, the programmer should be well aware that its content could be lost when invoking a macro.
- It goes without saying that TEMP or the register this name is assigned to (like r19) is excluded if registers r0...r31 are provided as macro parameters. By referring to the source code one can recognize where TEMP may be used nevertheless as a parameter.

3. Macro syntax

The application programmer sees a macro as a new instruction mnemonic. Some macros have no parameters at all. Some have one parameter, some two or more parameters. The syntax must consider some conventions and restrictions: the built-in keywords, the predefined mnemonics, and the capabilities and peculiarities of the assembler's preprocessor.

Host assembler: the AVR assembler 2 (AVRASM2) or higher.

Peculiarities and restrictions of the AVR assembler:

- The assembler is built around certain keywords, denoting instructions, registers, and built-in functions. Additional mnemonics are defined in device definition (*.INC*) files, belonging to the particular microcontroller devices.
- No overloading. Built-in keywords and pre-defined mnemonics cannot be used as user-defined symbols (in other words, as labels or names in *.def*, *.equ*, or *.set* directives).
- Consequently, all mnemonics that are already defined, must be circumvented, thus requiring to create own. So, for example, what in AVR terms is dubbed an immediate, is called a literal here.
- To the built-in keywords belong the instruction mnemonics and the register designations R0...R1 and X, Y, Z.
- XL, XH, YL, YH, ZL, ZH, and the designations of the I/O or peripheral registers belong to the predefined mnemonics, declared in the INC files of the devices.
- Uppercase and lowercase. The assembler is case-insensitive. Uppercase or lowercase does not matter.
- The registers R0...R31 and X, Y, Z can be renamed by .def statements:

```
temp = r19
```

- Addresses and other numeric values can be assigned mnemonics to by .equ statements:

```
.equ stacksize = 200
.equ ACK   = 0x06
```

The layout of a macro body

The *.macro* directive denotes the start of a macro. The *.endmacro* directive terminates the macro body. The parameters are denoted by @0, @1 and so on. The AVR assembler 2 does not limit the number of parameters. When invoking a macro, the parameters are passed as mnemonics, labels, numeric values, or ASCII character strings. A macro definition may invoke other macros. How deep macros may be nested, is not specified. However, the manufacturer advises not to overuse this feature. According to practical experience, a nesting depth of three should work.

Example of a simple macro body:

```
.macro bit0                 ; Clear bit in register. All registers allowed
        ldi temp, ~(1<<@1)  ; register adrs, bit adrs. Make the bitmask
        and @0,temp         ; clear bit by AND-ing the bitmask
.endmacro
```

Example of a macro body with a nested macro invocation:

```
.macro gclear                ; Clear byte at general adrs
        eor temp,temp        ; Make a 0x00 byte
        gst @0,temp          ; Store the byte at the general address
.       endmacro             ; by invoking the GST macro
```

Macro parameters
AVRASM2 accepts register names, single ASCII characters, ASCII character strings and binary numbers up to 64 bits long.

The AVRASM2 Preprocessor
Macro bodies may contain alternatives. They are selected by statements *.if, .else, .elif*, and *.endif*. Conditional statements can only query binary numbers and single ASCII characters, but no register designations:

```
.if @0 == r0
```

or

```
.if @0 == X
```

will lead to the error message

```
syntax error, unexpected REGISTER
```

It is possible, however, to define

```
.equ ser_buffer = udr        ; ATmega16 used for example
```

and to invoke this definition in conditional statements within the macro body:

```
.if @0 == ser_buffer
```

Where denote the particular function?
A toolbox should accommodate not only a few but many tools. As in most toolsets, there will be tools, which are different in size and functional details, but quite similar in their basic function, type, and overall appearance. The designers of instruction sets and assembler languages have to solve this problem when it comes to assembler syntax. There are basic operations with some variants and modifications. To provide one instruction mnemonic for each variant is obvious. When the number of variants is large, however, the mnemonics will become unwieldy and not that easy to learn. Therefore, some assembler languages provide instruction mnemonics only for the basic operations (like add, compare, or jump) and supplement it by modifiers related to the particular data types, addressing modes and so

on. Fig. 2 illustrates both principles. The AVR assembler, however, imposes some limitations. Not all kinds of characters, abbreviations and the like can be passed as parameters.

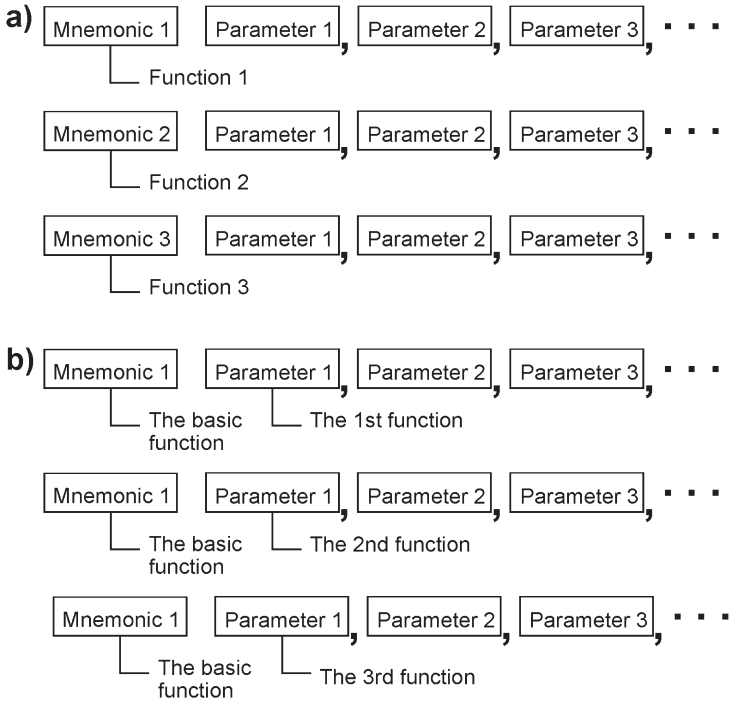


Fig. 2 Laying out different instructions or macros invoking the same principal (basic) kind of function, but with particular differences, like register size, word length, data type, addressing mode and so on. a) For each of the variants, a dedicated macro mnemonic will be provided. b) The macro mnemonic designates only the basic function (e. g., load, store, add, jump and so on). The particular function is selected by parameters, having own mnemonics (e.g., *w* to access to a word or @ to designate indirect addressing).

Macros with register parameters

Register designations, like r0 or X, can be passed to a macro. However, the preprocessor will not accept register names in conditional statements.

Coping with 16-bit registers

The assembler knows only the 16-bit registers X, Y, and Z. These register designations are built-in keywords. Beyond that, the assembler has no provisions to support register pairs containing 16-bit words. There are three alternatives to write macros coping with 16-bit words in two registers:

1) Two registers are passed as two parameters:

```
gwlit r4, r5, 0xF001
```

2) For each register pair to be coped with by a macro function, a separate macro body is provided (like in Fig. 2a):

```
lita 0xF001

litb 0xF002

litc 0xF003
```

3) Register designations are defined by .equ statements (as shown in Fig. 2b). Within the macro body, the particular register pair is selected by means of conditional statements:

   Defining the 16-bit registers:

```
.equ AW = 1

.equ BW = 2

.equ CW = 3

.equ XW = 4

.equ YW = 5

.equ ZW = 6
```

   It would be not expedient to define simply A, B, and C, because X, Y, and Z are, as built-in keywords, not supported this way. Thus all 16-bit registers have been christened with a two-letter mnemonic (wherein w = word).

   The macro body:

```
.macro LITW
        .if @0 == AW
                ldi AL, low(@1)
                ldi AH, high(@1)
        .elif @0 == BW
                ldi BL, low(@1)
                ldi BH, high(@1)
        .elif @0 == CW
                ldi BL, low(@1)
                ldi BH, high(@1)
        .elif @0 == XW
                ldi XL, low(@1)
                ldi XH, high(@1)
        . elif @0 == YW
```

```
            ldi YL, low(@1)
            ldi YH, high(@1)
        .else
            ldi ZL, low(@1)
            ldi ZH, high(@1)
        .endif
    .endmacro
```

An example invocation:

```
litw aw, 0x9911
```

The basic macros as described in paragraph 8 use the alternatives 1) and 2). When programming, the notation is shorter, hence it is less to type in, and the macro bodies are not that entangled with conditional statements. Macro bodies for registers A, B, C and so on are written easily by copying, pasting and substituting register designations. It is a quite natural method to build a macro toolbox step by step according to the requirements of the application programming. However, when the number of macros grows, the mnemonics would be more and more confusing. Then it may be more appropriate to change to alternative 3). This will be demonstrated in paragraph 9.

4. Memory and I/O addressing

Fig. 3 depicts the memory and I/O addressing of different AVR series.

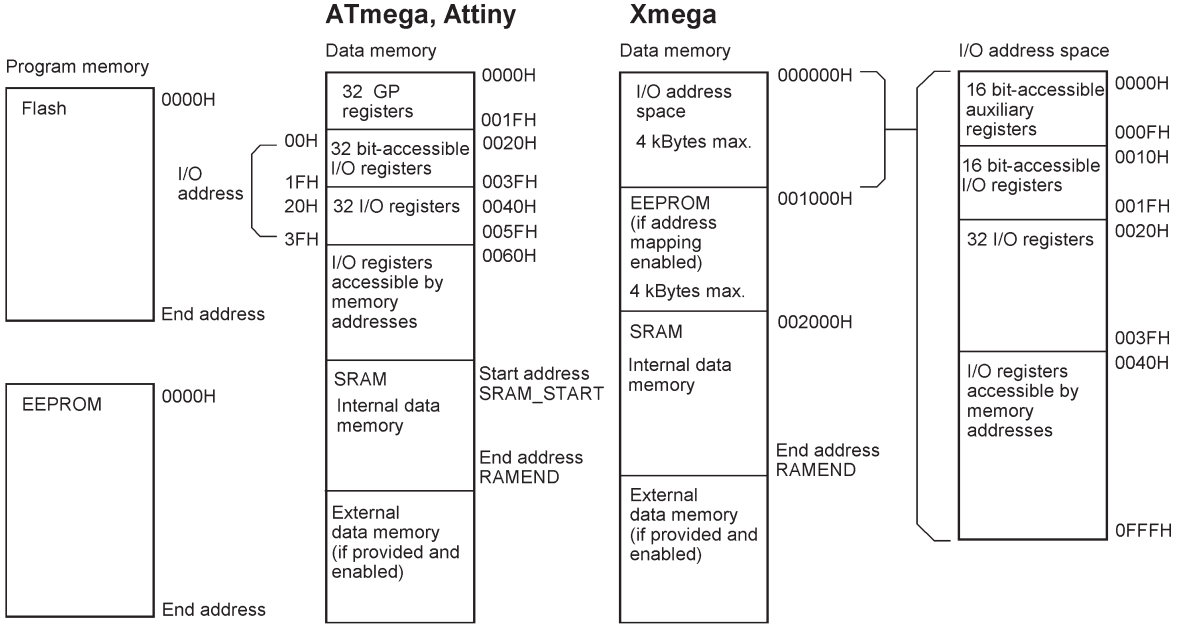


Fig. 3 AVR memory and I/O addressing.

ATmega / Attiny and Xmega

All I/O registers are accessible under data memory (SRAM) addresses (Table 3). A maximum of 64 I/O registers can be addressed in I/O instructions. Single-bit accesses are supported for the first 32 I/O addresses. In the Xmega, the I/O addresses are equal to the data memory addresses. In the ATmega or Attiny the I/O addresses are higher by 32 (20H). This must be declared (by *.equ* statements) or programmed in the application program.

I/O Address	Data memory (SRAM) address	
	ATmega etc.	Xmega
00H	0020H	0000H
1FH	003FH	001FH
20H	0040H	0020H
3FH	005FH	003FH

Table 3 I/O and memory addresses for accessing peripheral registers.

I/O and data memory (SRAM) addresses

AVR I/O instructions support only direct addressing. The address parameter is a literal. If the programmer wants to address an peripheral unit with an address parameter held in a register (indirect addressing), the peripheral unit must be addressed with its data memory (SRAM) addresses.

Example:

```
OUT udr, r16            ; Output to the UDR register (ATmega16) via its I/O address

LDS udr + 0x20, 0x55    ; Output of an immediate value (literal) to the register UDR via its
                        ; data memory (SRAM) address
```

Accessing 16-bit registers in peripheral units

When 16-bit registers in I/O units are accessed, the byte order matters. In ATmega or ATtiny devices, write into the high-order byte first, read from the low-order byte first. In Xmega devices, always access the low-order byte first. When writing into the data memory (SRAM), the byte order is not significant. Since AVR is a little-endian architecture, it is quite natural to access the low-order byte first.

Macro support of 16-bit I/O accesses

If the program is for a microcontroller that requires the low-order byte of a peripheral 16-bit register to be written first (for example, Xmega), set the following definition at the very beginning of the main program:

```
#define lowbytefirst (0)
```

If the program is for a microcontroller that requires the high-order byte of a peripheral 16-bit register to be written first (for example, ATmega or ATtiny), omit this definition.

How the macros address the periphery and the data memory (SRAM)
The macros know three types of address: the general address, the general SRAM address, and the SRAM address.

1) The general address (general_adrs)
The address parameters cover I/O as well as memory addresses. It depends on the address range, which type of access instructions (memory or I/O) will be used. Furthermore, 16-bit accesses will be executed in the proper sequence (low-order or high-order byte first), depending on the definition of *lobytefirst (0)* described above. If the address value is < 40H or (for single-bit access) < 20H, it is considered an I/O address and accessed by I/O instructions. Otherwise, it is a data memory (SRAM) address.

2) The general memory address (general_mem_adrs)
The address parameter is a data memory (SRAM) address. 16-bit accesses will be executed in the proper sequence (low-order or high-order byte first), depending on the definition of *lobytefirst (0)* described above. Those macros can be used to access peripheral registers via their corresponding memory addresses.

3) The data memory (SRAM) address (SRAM_adrs)
The address parameter is a data memory (SRAM) address. In 16-bit accesses, the low-order byte will be accessed first.

Address displacements
A displacement is an offset value to be added to a base address. There are two sizes of displacements as macro parameters:

1) 6-bit unsigned. These macros use the AVR instructions LDD and STD. The instruction will add a 6-bit displacement as a positive number (0...63) to the content of an address register X, Y, or Z.
2) 16-bit signed. These macros include an address calculation by software. The macro's instructions will add the displacement as a signed 16-bit number (–32 768...32 767) to the particular base address in a 16-bit register or in the SRAM.

5. Branching, subroutine call, and return

The AVR's conditional branch instructions have only a 7-bit address field and hence a short branch distance, covering the area from PC – 63 to PC + 64. If the branch target is further away, one has to program around, e.g., by using a branch instruction to skip over an unconditional jump instruction. Appropriate macros support branching within the complete address space. Conditional execution has been provided by jump, skip, call, and return macros.

Choose the right jump and subroutine call instructions
The AVR assembler cannot automatically choose between JMP and RJMP or CALL and RCALL. Some devices support only RJMP or RCALL. The macros need to know that. Therefore, if the device supports only RJMP / RCALL, set the following definition at the very beginning of the main program:

```
#define reljump (0)
```

If the device supports JMP / CALL, omit this definition.

6. Single-bit operations

The bit is the most basic data structure. There are control bits and output signals to be set, and status bits or input signals to be sensed. Macros are provided to set, clear or toggle a selected bit and to move a selected bit into one of the flags ZF or CF and vice versa. There are different ways to address the byte containing the bit (Table 4).

General mnemonic	Mnemonic in jump, skip, call, and return instructions	Bit selection
bit	0, 1. Examples: jp0, jp1	A bit in a register r0...r31
gbit	g0, g1. Examples: jpg0, jpg1	A bit in a byte at a general address (data memory (SRAM) or I/O register)
ubit	u0, u1. Examples: jpu0, jpu1	A bit in an unified bit-field (see below)
dbitx	x0, x1. Examples: jpx0, jpx1	Byte address (SRAM) = <X> + displacement
dbity	y0, y1. Examples: jpy0, jpy1	Byte address (SRAM) = <Y> + displacement
dbitz	z0, z1. Examples: jpz0, jpz1	Byte address (SRAM) = <Z> + displacement

Table 4 Different ways to address a bit in a byte.

Bit addressing in unified bit-fields (unified_bitfield_adrs)
When dealing with bits, one has to know the address of the byte the particular bit belongs to. For example, when writing routines to support the serial communication, one has to deal with a bit indicating that the USART has transmitted a byte. When programming an ATmega16, this bit is called *TXC* and resides in the register *UCSRA* at bit position 6. When programming an ATXMega64A4U and using the USART 0 on port E, the bit is called *TXCF* and resides in the register *USARTE0_STATUS* at bit position 6. As a remedy, the concept of the unified bit-field address has been introduced. It allows referring to individual bits, located anywhere in the general address space, by a single name, without having to worry about the byte address. The unified bit-field address is a general address extended by the bit address in the byte. In the bitfield definition, the general address is to be shifted one byte to the left (address << 8).

Definition examples

The bits to be defined are called *sercom_txc*, *strobe*, and *slave_buffer_empty*:

1) sercom_txc = bit 6 in the register UCSRA (e.g., ATmega16):

```
.equ sercom_txc = (ucsra << 8) + 6
```

2) sercom_txc = bit 6 in the register USARTE0_STATUS (e.g., ATXMega64A4U):

```
.equ sercom_txc = (usarte0_status << 8) + 6
```

3) strobe = bit 6 in port D:

```
.equ strobe = (portd << 8) + 6
```

4) slave_buffer_empty = bit 3 in the byte SERIAL_CHECKS (SRAM)

```
.equ slave_buffer_empty = (serial_checks << 8) + 3
```

For example, to set the bit slave_buffer_empty, simply write

```
UBIT1 slave_buffer_empty
```

This macro invocation substitutes something like

```
LDS r16, slave_buffer_empty
ORI r16, 8                    ;Bit 3 will be set in r16
STS slave_buffer_empty, r16
```

Macros of other functional groups support unified bit-fields, too. For example, to call a subroutine if the bit slave_buffer_empty is set, simply write:

```
CALLU1 slave_buffer_empty, slave_buffer_exception
```

7. Device-related definitions

Macro bodies must consider the byte order of 16-bit accesses to peripheral devices and the type of jump and call instructions the particular device supports. This is indicated by definitions. They have been described already on pages 9 and 11. Table 5 summarizes these definitions.

Definition	Insert the definition at the begin of the main program...
#define lowbytefirst (0)	if the device requires 16-bit peripheral registers to be loaded with the low-order byte first (e.g. Xmega). See also page 9
#define reljump (0)	if the device supports only relative jumps (RJMP, RCALL). See also page 11

Table 5 Device-related definitions to be inserted at the begin of the main program (if required).

8. Basic macros

Overview

The macros are divided into 28 functional groups (Table 6).

No.	Macro functions	No.	Macro functions
1	Basic transports	15	Load or store a 16-bit word (registers A, B, C, X, Y, Z)
2	Single-bit operations	16	Operations with 16-bit literals (registers A, B, C, X, Y, Z)
3	Jump on a flag	17	Operations with 8-bit literals and registers r0...r31
4	Jump on a single bit	18	Operations between 16-bit registers
5	Jump after subtracting A – B (compare arithmetically)	19	Operations between 16-bit and 8-bit registers
6	Skip a single instruction word	20	Operations register – memory (SRAM)
7	Skip two instruction words	21	Operations 16-bit register – memory (SRAM)
8	Skip the following instruction	22	24-bit and 32-bit operations
9	Call a subroutine	23	Load a register out of flash memory or SRAM
10	Return from a subroutine	24	Delays
11	Access memory and I/O with the Y-register as the base address register	25	Specialties
12	Access bytes in SRAM by indirect addressing (all registers r0...r31)	26	Save and restore register contents
13	Access words in SRAM by indirect addressing (registers A, B, C, X, Y, Z)	27	Supporting stack machine emulation
14	Clear a 16-bit register (A, B, C, X, Y, Z)	28	Subroutine call and parameter addressing

Table 6 The macros are divided into 28 functional groups No. 1 to No. 28.

8.1 Basic transports

Those macros are basic move operations dealing with general addresses. They keep care of the address space (data memory or I/O) and of the byte order (low-order or high-order byte first).

Mnemonic	Parameters	Function
lit	reg_adrs, literal	Load an immediate 8-bit value (literal) into a register (r0...31)[1]
gst	general_adrs, reg_adrs	Store a register content at a general address
gwst	general_adrs, reg_adrs, reg_adrs	Store the contents of two registers (r0...r31) as a 16-bit word at a general address[2]
gld	reg_adrs, general_adrs	Load a register (r0...r31) with the byte at a general address
gwld	reg_adrs, reg_adrs, general_adrs	Load two registers (r0...r31) with the 16-bit word at a general address[2]
glit	general_adrs, literal	Move an immediate 8-bit value (literal) to a general address
gwlit	general_adrs, 16-bit-literal	Move an immediate 16-bit value (literal) to a general address
gmov	general_adrs, general_adrs	Move the byte at the 2nd general address to the 1st
gwmov	general_adrs, general_adrs	Move the 16-bit word at the 2nd general address to the 1st
gclear	general_adrs	Clear the byte at a general address
gwclear	general_adrs	Clear the 16-bit word at a general address
gzero	general_adrs	Test, whether the byte at a general address is zero
gwzero	general_adrs	Test, whether the 16-bit word at a general address is zero
zero	reg_adrs	Test, whether the register content (r0...31) is zero

1. In contrast to the native LDI instruction, this macro can be applied to all registers of the AVR register file. However, when one of the registers r16...r31 is to be loaded, LDI will be faster.
2. 1st reg_adrs = high-order byte, 2nd reg_adrs = low-order byte.

Examples:

```
GLD al, udr              ; Load the register AL with the content of the peripheral register
                           ;UDR

GWLD  ch, cl, bytecount; Load the 16-bit register C with the content of the 16-bit word ;
                           ; bytecount in the SRAM

GWLIT bytecount, 1024; Load the 16-bit-word bytecount in SRAM with the value 1024
```

8.2 Single-bit operations

Clear a single bit

Mnemonic	Parameters	Function
gbit0	general_adrs, bit_adrs	Clear a bit at a general address
ubit0	unified_bitfield_adrs	Clear a bit in an unified bit-field
bit0	reg_adrs, bit_adrs	Clear a bit in a register r0...r31
bitf0	reg_adrs, bit_adrs	Clear a bit in a register r16...r31 (faster than bit0)
dbitx0	displacement,bit_adrs	Clear a bit in memory (SRAM). Addressed via the X-register + displacement
dbity0	displacement,bit_adrs	Clear a bit in memory (SRAM). Addressed via the Y-register + displacement
dbitz0	displacement,bit_adrs	Clear a bit in memory (SRAM). Addressed via the Z-register + displacement

Set a single bit

Mnemonic	Parameters	Function
gbit1	general_adrs, bit_adrs	Set a bit at a general address
ubit1	unified_bitfield_adrs	Set a bit in an unified bit-field
bit1	reg_adrs, bit_adrs	Set a bit in a register r0...r31
bitf1	reg_adrs, bit_adrs	Set a bit in a register r16...r31 (faster than bit1)
dbitx1	displacement,bit_adrs	Set a bit in memory (SRAM). Addressed via the X-register + displacement
dbity1	displacement,bit_adrs	Set a bit in memory (SRAM). Addressed via the Y-register + displacement
dbitz1	displacement,bit_adrs	Set a bit in memory (SRAM). Addressed via the Z-register + displacement

Toggle a single bit

Mnemonic	Parameters	Function
gbitt	general_adrs, bit_adrs	Toggle a bit at a general address
ubitt	unified_bitfield_adrs	Toggle a bit in an unified bit-field
bitt	reg_adrs, bit_adrs	Toggle a bit in a register r0...r31
dbitxt	displacement,bit_adrs	Toggle a bit in memory (SRAM). Addressed via the X-register + displacement
dbityt	displacement,bit_adrs	Toggle a bit in memory (SRAM). Addressed via the Y-register + displacement
dbitzt	displacement,bit_adrs	Toggle a bit in memory (SRAM). Addressed via the Z-register + displacement

Move a flag (ZF or CF) into a single bit

Mnemonic	Parameters	Function
gbitz	general_adrs, bit_adrs	Set a bit at a general address according to the zero flag (ZF). Bit = ZF inverted (if ZF = 1, then bit = 0)
ubitz	unified_bitfield_adrs	Set a bit in an unified-bit-field according to the zero flag (ZF). Bit = ZF inverted (if ZF = 1, then bit = 0)
bitz	reg_adrs, bit_adrs	Set a bit in a register (r0...r31) according to the zero flag (ZF). Bit = ZF inverted (if ZF = 1, then bit = 0)
dbitxz	displacement,bit_adrs	Set a bit in memory (SRAM) according to the zero flag (ZF). Bit = ZF inverted (if ZF = 1, then bit = 0). Addressed via the X-register + displacement
dbityz	displacement, bit_adrs	Set a bit in memory (SRAM) according to the zero flag (ZF). Bit = ZF inverted (if ZF = 1, then bit = 0). Addressed via the Y-register + displacement
dbitzz	displacement, bit_adrs	Set a bit in memory (SRAM) according to the zero flag (ZF). Bit = ZF inverted (if ZF = 1, then bit = 0). Addressed via the Z-register + displacement
gbitc	general_adrs, bit_adrs	Set a bit at a general address according to the carry flag (CF). Bit = CF
ubitc	unified_bitfield_adrs	Set a bit in an unified bit-field according to the carry flag (CF). Bit = CF

Mnemonic	Parameters	Function
ubitci	unified_bitfield_adrs	Set a bit in an unified bit-field according to the inverted carry flag (CF). Bit = CF inverted (if CF = 1, then bit = 0)
bitc	reg_adrs, bit_adrs	Set a bit in a register (r0...r31) according to the carry flag (CF). Bit = CF
dbitxc	displacement,bit_adrs	Set a bit in memory (SRAM) according to the carry flag (CF). Bit = CF. Addressed via the X-register + displacement
dbityc	displacement,bit_adrs	Set a bit in memory (SRAM) according to the carry flag (CF). Bit = CF. Addressed via the Y-register + displacement
dbitzc	displacement,bit_adrs	Set a bit in memory (SRAM) according to the carry flag (CF). Bit = CF. Addressed via the Z-register + displacement

Test (query, examine) a single bit

Mnemonic	Parameters	Function
gbit	general_adrs, bit_adrs	Test a bit at a general address for zero. ZF = bit inverted (if bit = 0, then ZF=1)
ubit	unified_bitfield_adrs	Test a bit in an unified bit-field for zero. ZF = bit inverted (if bit = 0, then ZF=1)
bit	reg_adrs, bit_adrs	Test a bit in a register (r0...r31) for zero. ZF = bit inverted (if bit = 0, then ZF=1)
dbitx	displacement,bit_adrs	Test a bit in memory (SRAM) for zero. ZF = bit inverted (if bit = 0, then ZF=1). Addressed via the X-register + displacement
dbity	displacement,bit_adrs	Test a bit in memory (SRAM) for zero. ZF = bit inverted (if bit = 0, then ZF=1). Addressed via the Y-register + displacement
dbitz	displacement,bit_adrs	Test a bit in memory (SRAM) for zero. ZF = bit inverted (if bit = 0, then ZF=1). Addressed via the Z-register + displacement
gbittoc	general_adrs, bit_adrs	Move a bit at a general address into the carry flag (CF). CF = bit
ubittoc	unified_bitfield_adrs	Move a bit in an unified bit-field into the carry flag (CF). CF = bit

Mnemonic	Parameters	Function
bittoc	reg_adrs, bit_adrs	Move a bit in a register (r0...r31) into the carry flag (CF). CF = bit
dbitxtoc	displacement,bit_adrs	Move a bit in memory (SRAM) into the carry flag (CF). Addressed via the X-register + displacement. CF = bit
dbitytoc	displacement,bit_adrs	Move a bit in memory (SRAM) into the carry flag (CF). Addressed via the Y-register + displacement. CF = bit
dbitztoc	displacement,bit_adrs	Move a bit in memory (SRAM) into the carry flag (CF). Addressed via the Z-register + displacement. CF = bit

Examples:

```
GBIT1 led_control, 1    ; Set bit 1 in the SRAM byte led_control

UBIT1 blink_green       ; The same effect as the macro above, when blink_geen  is
                        ; defined by:

                        .equ blink_green = (led_control << 8) + 1

UBIT sercom_udre        ; Query the bit udre1 in the USART register UCSR1A
                        ; (ATmega1284):

                        .equ sercom_udre = (ucsr1a  <<  8) + udre1
```

8.3 Jump on a flag

Relative or absolute jump instructions (RJMP, JMP) are chosen according to the definition *reljump(0)* (see Table 5). For the flag bits, refer to Fig. 1 and Table 1.

Mnemonic	Parameters	Function
brz	branch_adrs	Short branch on zero (branch if ZF = 1). Synonymous with the BREQ instruction (alias instruction)
brnz	branch_adrs	Short branch on not zero (branch if ZF = 0). Synonymous with the BRNE instruction (alias instruction)
jp; goto	jump_adrs	Jump always (unconditional)
jpz; jpeq	jump_adrs	Jump if zero, jump if equal (jump if ZF = 1)
jpnz; jpne	jump_adrs	Jump if not zero, jump if not equal (jump if ZF = 0)

Mnemonic	Parameters	Function
jpc	jump_adrs	Jump if carry (jump if CF = 1)
jpnc	jump_adrs	Jump if no carry (jump if CF = 0)
jpn	jump_adrs	Jump if negative (jump if SF = 1)
jpp	jump_adrs	Jump if positive (jump if SF = 0)
jpov	jump_adrs	Jump if overflow (jump if OF = 1)
jpnov	jump_adrs	Jump if no overflow (jump if OF = 0)
jplf	jump_adrs	Jump if less. Signed subtraction A – B yields A < B
jpnlf	jump_adrs	Jump if not less = greater or equal (A ≥ B)
jph	jump_adrs	Jump if carry-out of low-order 4-bit nibble (jump if HF = 1)
jpnh	jump_adrs	Jump if no carry-out of low-order 4-bit nibble (jump if HF = 0)
jpt	jump_adrs	Jump if Transfer-bit set (jump if TF = 1)
jpnt	jump_adrs	Jump if Transfer-bit cleared (jump if TF = 0)
jpen	jump_adrs	Jump if interrups enabled (jump if IF = 1)
jpdi	jump_adrs	Jump if interrupts disabled (jump if IF = 0)

8.4 Jump on a single bit

Relative or absolute jump instructions (RJMP, JMP) are chosen according to the definition *reljump(0)* (see Table 5).

Mnemonic	Parameters	Function
jpg0	general_adrs, bit_adrs, jump_adrs	Jump if a bit at a general address is = 0
jpg1	general_adrs, bit_adrs, jump_adrs	Jump if a bit at a general address is = 1
jp0	reg_adrs, bit_adrs, jump_adrs	Jump if a bit in a register (r0...r31) is = 0
jp1	reg_adrs, bit_adrs, jump_adrs	Jump if a bit in a register (r0...r31) is = 1
jpu0	unified_bitfield_adrs, jump_adrs	Jump if a bit in a unified bit-field is = 0

Mnemonic	Parameters	Function
jpu1	unified_bitfield_adrs, jump_adrs	Jump if a bit in a unified bit-field is = 1
jpx0	displacement, bit_adrs,jump_adrs	Jump if a bit in memory (SRAM) is = 0. Addressed via the X-register + displacement
jpx1	displacement, bit_adrs,jump_adrs	Jump if a bit in memory (SRAM) is = 1. Addressed via the X-register + displacement
jpy0	displacement, bit_adrs,jump_adrs	Jump if a bit in memory (SRAM) is = 0. Addressed via the Y-register + displacement
jpy1	displacement, bit_adrs,jump_adrs	Jump if a bit in memory (SRAM) is = 1. Addressed via the Y-register + displacement
jpz0	displacement, bit_adrs,jump_adrs	Jump if a bit in memory (SRAM) is = 0. Addressed via the Z-register + displacement
jpz1	displacement, bit_adrs,jump_adrs	Jump if a bit in memory (SRAM) is = 1. Addressed via the Z-register + displacement

8.5 Jump after subtracting A – B (compare arithmetically)

Relative or absolute jump instructions (RJMP, JMP) are chosen according to the definition *reljump(0)* (see Table 5). Here, A means the first operand, B the second. A and B are generic operand names. Do not confuse with the virtual machine's 16-bit registers A, B.

Mnemonic	Parameter	Function
jpb	jump_adrs	Jump if below. Unsigned subtraction A – B yields A < B
jpbe	jump_adrs	Jump if below or equal. Unsigned subtraction A – B yields A ≤ B
jpa	jump_adrs	Jump if above. Unsigned subtraction A – B yields A > B
jpae	jump_adrs	Jump if above or equal. Unsigned subtraction A – B yields A ≥ B
jpl	jump_adrs	Jump if less. Signed subtraction A – B yields A < B
jple	jump_adrs	Jump if less or equal. Signed subtraction A – B yields A ≤ B
jpg	jump_adrs	Jump if greater. Signed subtraction A – B yields A > B
jpge	jump_adrs	Jump if greater or equal. Signed subtraction A – B yields A ≥ B

8.6 Skip a single instruction word

Here, A means the first operand, B the second. A and B are generic operand names. Do not confuse with the virtual machine's 16-bit registers A, B. The macros will skip a single instruction word if the particular condition is satisfied.

Caution: Programmers should be aware that the instruction to be skipped is really one word long.

Mnemonic	Parameters	Function
skipz; skipeq		Skip if zero, skip if equal (skip if ZF = 1)
skipnz; skipne		Skip if not zero, skip if not equal (skip if ZF = 0)
skipb		Skip if below. Unsigned subtraction A – B yields A < B
skipbe		Skip if below or equal. Unsigned subtraction A – B yields A ≤ B
skipa		Skip if above. Unsigned subtraction A – B yields A > B
skipae		Skip if above or equal. Unsigned subtraction A – B yields A ≥ B
skipl		Skip if less. Signed subtraction A – B yields A < B
skiple		Skip if less or equal. Signed subtraction A – B yields A ≤ B
skipg		Skip if greater. Signed subtraction A – B yields A > B
skipge		Skip if greater or equal. Signed subtraction A – B yields A ≥ B
skipg0	general_adrs, bit_adrs	Skip if a bit at a general address is = 0
skipg1	general_adrs, bit_adrs	Skip if a bit at a general address is = 1
skipu0	unified_bitfield_adrs	Skip if a bit in an unified bit-field is = 0
skipu1	unified_bitfield_adrs	Skip if a bit in an unified bit-field is = 1

8.7 Skip two instruction words

Here, A means the first operand, B the second. A and B are generic operand names. Do not confuse with the virtual machine's 16-bit registers A, B. The macros will skip two consecutive instruction words if the particular condition is satisfied.

Caution: Programmers should be aware that the instruction to be skipped is really two words long or that the intention is to skip two consecutive single-word instructions.

Mnemonic	Parameters	Function
skipz2; skipeq2		Skip if zero, skip if equal (skip if ZF = 1)
skipnz2; skipne2		Skip if not zero, skip if not equal (skip if ZF = 0)
skipb2		Skip if below. Unsigned subtraction A – B yields A < B
skipbe2		Skip if below or equal. Unsigned subtraction A – B yields A ≤ B
skipa2		Skip if above. Unsigned subtraction A – B yields A > B
skipae2		Skip if above or equal. Unsigned subtraction A – B yields A ≥ B
skipl2		Skip if less. Signed subtraction A – B yields A < B
skiple2		Skip if less or equal. Signed subtraction A – B yields A ≤ B
skipg2		Skip if greater. Signed subtraction A – B yields A > B
skipge2		Skip if greater or equal. Signed subtraction A – B yields A ≥ B
skipg02	general_adrs, bit_adrs	Skip if a bit at a general address is = 0
skipg12	general_adrs, bit_adrs	Skip if a bit at a general address is = 1
skipu02	unified_bitfield_adrs	Skip if a bit in an unified bit-field is = 0
skipu12	unified_bitfield_adrs	Skip if a bit in an unified bit-field is = 1

8.8 Skip the following instruction

Here, A means the first operand, B the second. A and B are generic operand names. Do not confuse with the virtual machine's 16-bit registers A, B. The number of instruction words to be skipped will be considered automatically. Those macros employ the TEMP register. Some SKIPF-macros are slower than SKIP or SKIP2, some not (refer to the source listing, when this detail matters).

Mnemonic	Parameters	Function
skipfz; skipfeq		Skip if zero, skip if equal (skip if ZF = 1)
skipfnz, skipfne		Skip if not zero, skip if not equal (skip if ZF = 0)
skipf		Skip always (unconditional)
skipfc		Skip if carry (CF = 1)
skipfnc		Skip if no carry (CF = 0)

Mnemonic	Parameters	Function
skipfn		Skip if negative (SF = 1)
skipfp		Skip if positive (SF = 0)
skipfov		Skip if overflow (OF = 1)
skipfnov		Skip if no overflow (OF = 0)
skipflf		Skip if less (LF = 1). Signed subtraction A – B yields A < B
skipfnlf		Skip if not less (LF = 0) = greater or equal (A ≥ B)
skipfh		Skip if carry-out of low-order 4-bit nibble (half-byte carry; HF = 1)
skipfnh		Skip if no carry-out of low-order 4-bit nibble (HF = 0)
skipft		Skip if Transfer-bit set (TF = 1)
skipfnt		Skip if Transfer-bit cleared (TF = 0)
skipfen		Skip if interrups enabled (IF = 1)
skipfdi		Skip if interrupts disabled (IF = 0)
skipfb		Skip if below. Unsigned subtraction A – B yields A < B
skipfbe		Skip if below or equal. Unsigned subtraction A – B yields A ≤ B
skipfa		Skip if above. Unsigned subtraction A – B yields A > B
skipfae		Skip if above or equal. Unsigned subtraction A – B yields A ≥ B
skipfl		Skip if less. Signed subtraction A – B yields A < B
skipfle		Skip if less or equal. Signed subtraction A – B yields A ≤ B
skipfg		Skip if greater. Signed subtraction A – B yields A > B
skipfge		Skip if greater or equal. Signed subtraction A – B yields A ≥ B
skipfg0	general_adrs, bit_adrs	Skip if a bit at a general address is = 0
skipfg1	general_adrs, bit_adrs	Skip if a bit at a general address is = 1
skipfu0	unified_bitfield_adrs	Skip if a bit in an unified bit-field is = 0
skipfu1	unified_bitfield_adrs	Skip if a bit in an unified bit-field is = 1

8.9 Call a subroutine

Relative or absolute call instructions (RCALL, CALL) are chosen according to the definition *rel-jump(0)* (see Table 5). Here, A means the first operand, B the second. A and B are generic operand names. Do not confuse with the virtual machine's 16-bit registers A, B.

Mnemonic	Parameters	Function
cal; gosub	subroutine_adrs	Call always (unconditional)
callz; calleq	subroutine_adrs	Call if zero, jump if equal (jump if ZF = 1)
callnz; callne	subroutine_adrs	Call if not zero, jump if not equal (jump if ZF = 0)
callb	subroutine_adrs	Call if below. Unsigned subtraction A – B yields A < B
callbe	subroutine_adrs	Call if below or equal. Unsigned subtraction A – B yields A ≤ B
calla	subroutine_adrs	Call if above. Unsigned subtraction A – B yields A > B
callae	subroutine_adrs	Call if above or equal. Unsigned subtraction A – B yields A ≥ B
calll	subroutine_adrs	Call if less. Signed subtraction A – B yields A < B
callle	subroutine_adrs	Call if less or equal. Signed subtraction A – B yields A ≤ B
callg	subroutine_adrs	Call if greater. Signed subtraction A – B yields A > B
callge	subroutine_adrs	Call if greater or equal. Signed subtraction A – B yields A ≥ B
callg0	general_adrs, bit_adrs, subroutine_adrs	Call if a bit at a general address is = 0
callg1	general_adrs, bit_adrs, subroutine_adrs	Call if a bit at a general address is = 1
call0	reg_adrs, bit_adrs, subroutine_adrs	Call if a bit in a register (r0...r31) is = 0
call1	reg_adrs, bit_adrs, subroutine_adrs	Call if a bit in a register (r0...r31) is = 1
callu0	unified_bitfield_adrs, subroutine_adrs	Call if a bit in a unified bit-field is = 0
callu1	unified_bitfield_adrs, subroutine_adrs	Call if a bit in a unified bit-field is = 1
callx0	displacement, bit_adrs, subroutine_adrs	Call if a bit in memory (SRAM) is = 0. Addressed via the X-register + displacement
callx1	displacement,b it_adrs, subroutine_adrs	Call if a bit in memory (SRAM) is = 1. Addressed via the X-register + displacement

Mnemonic	Parameters	Function
cally0	displacement, bit_adrs, subroutine_adrs	Call if a bit in memory (SRAM) is = 0. Addressed via the Y-register + displacement
cally1	displacement, bit_adrs, subroutine_adrs	Call if a bit in memory (SRAM) is = 1. Addressed via the Y-register + displacement
callz0	displacement, bit_adrs, subroutine_adrs	Call if a bit in memory (SRAM) is = 0. Addressed via the Z-register + displacement
callz1	displacement, bit_adrs, subroutine_adrs	Call if a bit in memory (SRAM) is = 1. Addressed via the Z-register + displacement

8.10 Return from a subroutine

Here, A means the first operand, B the second. A and B are generic operand names. Do not confuse with the virtual machine's 16-bit registers A, B.

Mnemonic	Parameters	Function
retz; reteq		Return if zero, jump if equal (jump if ZF = 1)
retnz; retne		Return if not zero, jump if not equal (jump if ZF = 0)
retb		Return if below. Unsigned subtraction A – B yields A < B
retbe		Return if below or equal. Unsigned subtraction A – B yields A ≤ B
reta		Return if above. Unsigned subtraction A – B yields A > B
retae		Return if above or equal. Unsigned subtraction A – B yields A ≥ B
retl		Return if less. Signed subtraction A – B yields A < B
retle		Return if less or equal. Signed subtraction A – B yields A ≤ B
retg		Return if greater. Signed subtraction A – B yields A > B
retge		Return if greater or equal. Signed subtraction A – B yields A ≥ B
retg0	general_adrs, bit_adrs	Return if a bit at a general address is = 0
retg1	general_adrs, bit_adrs	Return if a bit at a general address is = 1
ret0	reg_adrs, bit_adrs	Return if a bit in a register (r0...r31) is = 0
ret1	reg_adrs, bit_adrs	Return if a bit in a register (r0...r31) is = 1
retu0	reg_adrs, bit_adrs	Return if a bit in a unified bit-field is = 0
retu1	reg_adrs, bit_adrs	Return if a bit in a unified bit-field is = 1

Mnemonic	Parameters	Function
retx0	displacement, bit_adrs,jump_adrs	Return if a bit in memory (SRAM) is = 0. Addressed via the X-register + displacement
retx1	displacement, bit_adrs	Return if a bit in memory (SRAM) is = 1. Addressed via the X-register + displacement
rety0	displacement, bit_adrs	Return if a bit in memory (SRAM) is = 0. Addressed via the Y-register + displacement
rety1	displacement, bit_adrs	Return if a bit in memory (SRAM) is = 1. Addressed via the Y-register + displacement
retz0	displacement, bit_adrs	Return if a bit in memory (SRAM) is = 0. Addressed via the Z-register + displacement
retz1	displacement, bit_adrs	Return if a bit in memory (SRAM) is = 1. Addressed via the Z-register + displacement

8.11 Access memory and I/O with the Y-register as the base address register

The addresses are general SRAM addresses. The access functions consider the byte order (low- or high-order byte first). The byte order is chosen according to the definition *lowbytefirst (0)* (see Table 5). These macros have been provided above all to encode I/O control programs ("device drivers"). Such programs should be usable for several similar devices. For that, the Y register is used as the base address register.

Caution: Peripheral (I/O) registers must be are accessed by their data memory (SRAM) addresses.

Mnemonic	Parameters	Function
litdy	displacement, literal	Store or output an immediate 8-bit value (literal)
litwdy	displacement, literal	Store or output an immediate 16-bit value (literal)
lddy	reg_adrs, displacement	Load or input into a register (r0...r31)
stdy	displacement, reg_adrs	Store or output a register content (r0...r31)
ldady	displacement	Load or input into the A-register (16 bits)
ldbdy	displacement	Load or input into the B-register (16 bits)
ldcdy	displacement	Load or input into the C-register (16 bits)
ldxdy	displacement	Load or input into the X-register (16 bits)
ldzdy	displacement	Load or input into the Z-register (16 bits)
stady	displacement	Store or output the content of the A-register (16 bits)

Mnemonic	Parameters	Function
stbdy	displacement	Store or output the content of the B-register (16 bits)
stcdy	displacement	Store or output the content of the C-register (16 bits)
cleardy	displacement	Clear the addressed byte
clearwdy	displacement	Clear the addressed 16-bit word
zerody	displacement	Test, whether the addressed byte is zero
zerowdy	displacement	Test, whether the addressed 16-bit word is zero
bitdy 0	displacement, bit_adrs	Clear the addressed bit
bitdy 1	displacement, bit_adrs	Set the addressed bit
bitdy	displacement, bit_adrs	Test, whether the addressed bit is zero
skipfdy0	displacement, bit_adrs	Skip the following instruction if the adressed bit = 0
skipfdy1	displacement, bit_adrs	Skip the following instruction if the adressed bit = 1

8.12 Access bytes in SRAM by indirect addressing (all registers r0...r31)

Those macros extend indirect addressing beyond the corresponding provisions of the AVR architecture (i.e., the addressing capabilities of the registers X, Y, Z). The *base_adrs* parameter points to a 16-bit word in the SRAM containing the operand address or the base address a displacement will be added to. The displacement parameter is either a literal or an address pointing to a 16-bit word in SRAM containing the displacement value. Refer to Fig. 4 (page 30). The displacement is a signed 16-bit number.

.

Mnemonic	Parameters	Function
ldir	reg_adrs, base_adrs	Load indirect. <reg_adrs> := <<base_adrs>>. Cf. Fig. 4a
ldirinc	reg_adrs, base_adrs	Load indirect. Post-increment the address pointer. <reg_adrs> := <<base_adrs>>; <base_adrs> := <base_adrs> +1. Cf. Fig. 4b
ldirdec	reg_adrs, base_adrs	Pre-decrement the address pointer. Then load indirect. <base_adrs> := <base_adrs> –1); <reg_adrs> := <<base_adrs>>. Cf. Fig. 4b
ldirb	reg_adrs, base_adrs, displacement_literal	Load indirect. <reg_adrs> := <<base_adrs> + displacement_literal>. Cf. Fig. 4c
ldirbd	reg_adrs, base_adrs, displacement_adrs	Load indirect. <reg_adrs> := <<base_adrs> + <displacement_adrs>>. Cf. Fig. 4d
stir	reg_adrs, base_adrs	Store indirect. <<base_adrs>> := <reg_adrs>. Cf. Fig. 4a
stirinc	reg_adrs, base_adrs	Store indirect. Post-increment the address pointer. <<base_adrs>> := <reg_adrs> ; <base_adrs> := <base_adrs> +1. . Cf. Fig. 4b

Mnemonic	Parameters	Function
stirdec	reg_adrs, base_adrs	Pre-decrement the address pointer. Then store indirect. <base_adrs> := <base_adrs> –1); <<base_adrs>> := <reg_adrs>. Cf. Fig. 4b
stirb	reg_adrs, base_adrs, displacement_literal	Store indirect. <<base_adrs> + displacement_literal> := <reg_adrs>. Cf. Fig. 4c
stirbd	reg_adrs, base_adrs, displacement_adrs	Store indirect. <<base_adrs> + <displacement_adrs>> := <reg_adrs>. Cf. Fig. 4d

8.13 Access 16-bit words in SRAM by indirect addressing (registers A, B, C, X, Y, Z)

Those macros extend indirect addressing beyond the corresponding provisions of the AVR architecture (i.e., the addressing capabilities of the registers X, Y, Z). The *base_adrs* parameter points to a 16-bit word in the SRAM containing the operand address or the base address a displacement will be added to. The displacement parameter is either a literal or an address pointing to a 16-bit word in SRAM containing the displacement value. The displacement in those macros is always a signed 16-bit number. Refer to Fig. 4 (page 30). The macro functions are the same as those provided for the 8-bit registers (cf.the previous paragraph 8.12), except the increments and decrements are by 2 (because words are addressed). Because the AVR assembler does not support 16-bit registers, the register cannot be passed as a parameter. Therefore, each 16-bit register needs a separate macro mnemonic. The first table describes the macro operations by generic mnemonics, ending with an „m“ as the generic register name. When programming, simply replace the „m“ by the desired register designator (A, B, X, Y, Z). The corresponding mnemonics are summarized in the second table. For an alternative syntax, refer to paragraph 9.

.

Mnemonic	Parameters	Function
ldiwm	base_adrs	Load indirect. <16-bit reg m> := <<base_adrs>> Cf. Fig. 4a
ldiwincm	base_adrs	Load indirect. Post-increment the address pointer. <16-bit reg m> := <<base_adrs>>; <base_adrs> := <base_adrs> + 2. Cf. Fig. 4b
ldiwdecm	base_adrs	Pre-decrement the address pointer. Then load indirect. <base_adrs> = <base_adrs> – 2; <16-bit reg m> := <<base_adrs>>. Cf. Fig. 4b
ldiwbm	base_adrs, displacement_literal	Load indirect. <16-bit reg m> := <<base_adrs> + displacement_literal. Cf. Fig. 4c
ldiwbdm	base_adrs, displacement_adrs	Load indirect. <16-bit reg m> := <<base_adrs> + <displacement_adrs>>. Cf. Fig. 4d
stiwm	base_adrs	Store indirect. <<base_adrs>> := <16-bit reg m>. Cf. Fig. 4a
stiwincm	base_adrs	Store indirect. Post-increment the address pointer. <<base_adrs>> := <16-bit reg m> ; <base_adrs> := <base_adrs> + 2 . Cf. Fig. 4b

Mnemonic	Parameters	Function
stiwdecm	base_adrs	Pre-decrement the address pointer. Then store indirect. <base_adrs> := <base_adrs> – 2; <<base_adrs>> := <16-bit reg m>. Cf. Fig. b
stiwbm	base_adrs, displacement_literal	Store indirect. <<base_adrs> + displacement_literal> := <16-bit reg m>. Cf. Fig. 4c
stiwbdm	base_adrs, displacement_adrs	Store indirect. <<base_adrs> + <displacement_adrs>> := <16-bit reg m>. Cf. Fig. 4d

Register A	Register B	Register C	Register X	Register Y	Register Z
ldiwa	ldiwb	ldiwc	ldiwx	ldiwy	ldiwz
ldiwinca	ldiwincb	ldiwincc	ldiwincx	ldiwincy	ldiwincz
ldiwdeca	ldiwdecb	ldiwdecc	ldiwdecx	ldiwdecy	ldiwdecz
ldiwba	ldiwbb	ldiwbc	ldiwbx	ldiwby	ldiwbz
ldiwbda	ldiwbdb	ldiwbdc	ldiwbdx	ldiwbdy	ldiwbdz
stiwa	stiwb	stiwc	stiwx	stiwy	stiwz
stiwinca	stiwincb	stiwincc	stiwincx	stiwincy	stiwincz
stiwdeca	stiwdecb	stiwdecc	stiwdecx	stiwdecy	stiwdecz
stiwba	stiwbb	stiwbc	stiwbx	stiwby	stiwbz
stiwbda	stiwbdb	stiwbdc	stiwbdx	stiwbdy	stiwbdz

8.14 Clear a 16-bit register (registers A, B, C, X, Y, Z)

Mnemonic	Parameter	Function
cleara		Clear the 16-bit register A
clearb		Clear the 16-bit register B
clearc		Clear the 16-bit register C
clearx		Clear the 16-bit register X
cleary		Clear the 16-bit register Y
clearz		Clear the 16-bit register Z

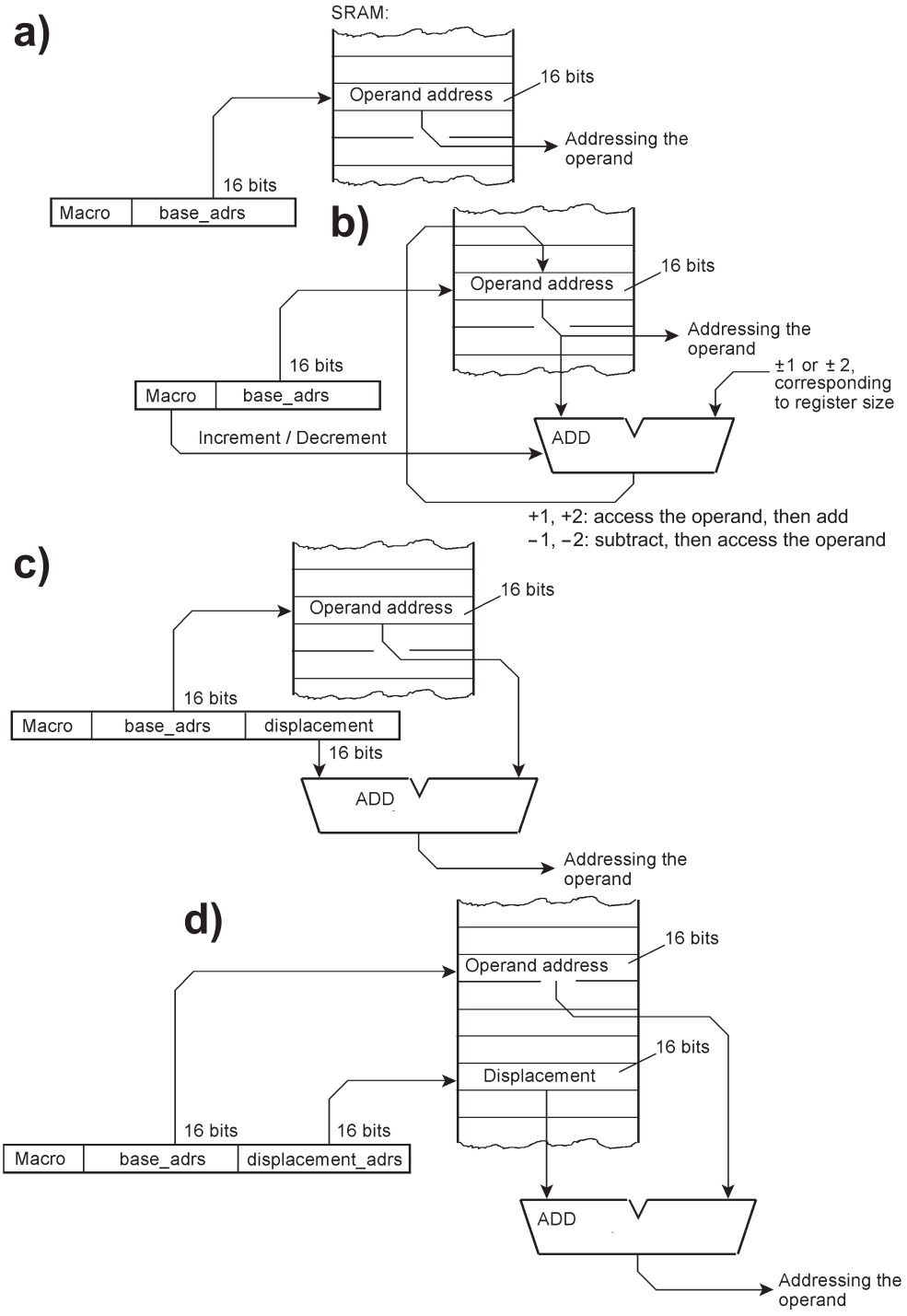


Fig. 4 Indirect addressing via address pointers in the SRAM. Refer to paragraphs 8.12 and 8.13

8.15 Load or store a 16-bit word (registers A, B, C, X, Y, Z)

Because the AVR assembler does not support 16-bit registers, the register cannot be passed as a parameter. Therefore, each 16-bit register needs a separate macro mnemonic. The first table describes the macro operations by generic mnemonics, ending with an „m“ as the generic register name. When programming, simply replace the „m“ by the desired register designator (A, B, X, Y, Z). The corresponding mnemonics are summarized in the second table. For an alternative syntax, refer to paragraph 9.

Caution: Not all macros are provided for all 16-bit registers. Refer to the second table.

Mnemonic	Parameter	Function
ldm	general_mem_adrs	Load the 16-bit register m
ldbym	SRAM_adrs	Load the 16-bit register m with a zero-extended byte out of the SRAM
ldxm		Load the 16-bit register m. General memory address in X
ldym		Load the 16-bit register m. General memory address in Y
ldzm		Load the 16-bit register m. General memory address in Z
stm	general_mem_adrs	Store the content of the 16-bit register m.
stxm		Store the content of the 16-bit register m. General memory address in X
stym		Store the content of the 16-bit register m. General memory address in Y
stzm		Store the content of the 16-bit register m. General memory address in Z
ldxincm		Load the 16-bit register m. General memory address in X. Post-increment the address pointer in X by 2
ldyincm		Load the 16-bit register m. General memory address in Y. Post-increment the address pointer in Y by 2
stxincm		Store the content of the 16-bit register m. General memory address in X. Post-increment the address pointer in X by 2
styincm		Store the content of the 16-bit register m. General memory address in Y. Post-increment the address pointer in Y by 2

Register A	Register B	Register C	Register X	Register Y	Register Z
lda	ldb	ldc	ldx	ldy	ldz
ldbya	ldbyb	ldbyc	–	–	–
ldxa	ldxb	ldxc	–	ldxy	ldxz
ldya	ldyb	ldyc	ldyx	–	ldyz
ldza	ldzb	ldzc	ldzx	ldzy	–
sta	stb	stc	stx	sty	stz
stxa	stxb	stxc	–	stxy	stxz
stya	styb	styc	styx	–	styz
stza	stzb	stzc	stzx	stzy	–
ldxinca	ldxincb	ldxincc	–	–	–
ldyinca	ldyincb	ldyincc	–	–	–
stxinca	stxincb	stxincc	–	–	–
styinca	styincb	styincc	–	–	–

8.16 Operations with 16-bit literals (registers A, B, C, X, Y, Z)

Because the AVR assembler does not support 16-bit registers, the register cannot be passed as a parameter. Therefore, each 16-bit register needs a separate macro mnemonic. The first table describes the macro operations by generic mnemonics, ending with an „m“ as the generic register name. When programming, simply replace the „m“ by the desired register designator (A, B, X, Y, Z). The corresponding mnemonics are summarized in the second table. For an alternative syntax, refer to paragraph 9.

Mnemonic	Parameter	Function
litm	16_bit_literal	Load the literal into the 16-bit register m (<16-bit reg m> := literal)
addlitm	16_bit_literal	Add the literal to the content of the 16-bit register m (<16-bit reg m> + literal)
sublitm	16_bit_literal	Subtract the literal from the content of the 16-bit register m (<16-bit reg m> – iteral)
cmplitm	16_bit_literal	Compare literal with the content of the 16-bit register m (by subtracting <16-bit reg m> – literal)
andlitm	16_bit_literal	Bitwise AND between the content of the 16-bit register m and the literal

Mnemonic	Parameter	Function
orlitm	16_bit_literal	Bitwise OR between the content of the 16-bit register m and the literal
xorlitm	16_bit_literal	Bitwise XOR between the content of the 16-bit register m and the literal
zerom		Test whether the content of the 16-bit register m is zero

Register A	Register B	Register C	Register X	Register Y	Register Z
lita	litb	litc	litx	lity	litz
addlita	addlitb	addlitc	addlitx	addlity	addlitz
sublita	sublitb	sublitc	sublitx	sublity	sublitz
cmplita	cmplitb	cmplitc	cmplitx	cmplity	cmplitz
andlita	andlitb	andlitc	–	–	–
orlita	orlitb	orlitc	–	–	–
xorlita	xorlitb	xorlitc	–	–	–
zeroa	zerob	zeroc	zerox	zeroy	zeroz

8.17 Operations with 8-bit literals and registers r0...r31

Mnemonic	Parameters	Function
addlit	reg_adrs, literal	<reg:_adrs> := <reg_adrs> + literal
adclit	reg_adrs, literal	<reg_adrs> := <reg_adrs> + literal + CF (add with carry)
sublit	reg_adrs, literal	<reg_adrs> := <reg_adrs> – literal
sbclit	reg_adrs, literal	<reg_adrs> := <reg_adrs> – literal – CF (subtract with borrow)
cmplit	reg_adrs, literal	Compare by subtracting <reg_adrs> – literal
cmpclit	reg_adrs, literal	Compare by subtracting <reg_adrs> – literal – CF (subtract with borrow)
andlit	reg_adrs, literal	<reg_adrs> := <reg_adrs> AND literal
orlit	reg_adrs, literal	<reg_adrs> := <reg_adrs> OR literal
xorlit	reg_adrs, literal	<reg_adrs> := <reg_adrs> XOR literal

8.18 Operations between 16-bit registers

Mnemonic	Parameter	Function
addab		<A> := <A> + <B>
subab		<A> := <A> – <B>
cmpab		Compare B with A by subtracting <A> – <B>
andab		<A> := <A> AND <B>
orab		<A> := <A> OR <B>
xorab		<A> := <A> XOR <B>
addac		<A> := <A> + <C>
subac		<A> := <A> – <C>
cmpac		Compare C with A by subtracting <A> – <C>
andac		<A> := <A> AND <C>
orac		<A> := <A> OR <C>
xorac		<A> := <A> XOR <C>
addax		<A> := <A> + <X>
subax		<A> := <A> – <X>
cmpax		Compare X with A by subtracting <A> – <X>
adday		<A> := <A> + <Y>
subay		<A> := <A> – <Y>
cmpay		Compare Y to A by subtracting <A> – <Y>
addaz		<A> := <A> + <Z>
subaz		<A> := <A> – <Z>
cmpaz		Compare Z with A by subtracting <A> – <Z>
addxa		<X> := <X> + <A>
subxa		<X> := <X> – <A>
cmpxa		Compare A with X by subtracting <X> – <A>

Mnemonic	Parameter	Function
addya		<Y> := <Y> + <A>
subya		<Y> := <Y> – <A>
cmpya		Compare A with Y by subtracting <Y> – <A>
addza		<Z> := <Z> + <A>
subza		<Z> := <Z> – <A>
cmpza		Compare A with Z by subtracting <Z> – <A>
lsla		Left-shift A by one bit. Fill with 0
lsla1		Left-shift A by one bit and fill with 1
lsra		Right-shift A by one bit. Fill with 0
lsra1		Right-shift A by one bit and fill with 1
asra		Right-shift A arithmetically
rola		Left-rotate A by one bit. The carry flag is not included
rora		Right-rotate A by one bit. The carry flag is not included
nega		Change the sign of the content of register A
absa		<A> := ABS <A> (absolute amount)

8.19 Operations between 16-bit and 8-bit registers

The content of the 8-bit register will be zero-extended to 16 bits.

Mnemonic	Parameter	Function
addaby	reg_adrs	<A> := <A> + <reg_adrs>
subaby	reg_adrs	<A> := <A> – <reg_adrs>
cmpaby	reg_adrs	Compare <reg_adrs> with A by subtracting <A> – <reg_adrs>
addbby	reg_adrs	<B> := <B> + <reg_adrs>
subbby	reg_adrs	<B> := <B> – <reg_adrs>
cmpbby	reg_adrs	Compare <reg_adrs> with B by subtracting <B> – <reg_adrs>
addcby	reg_adrs	<C> := <CA> + <reg_adrs>

Mnemonic	Parameter	Function
subcby	reg_adrs	<C> := <C> – <reg_adrs>
cmpcby	reg_adrs	Compare <reg_adrs> with C by subtracting <C> – <reg_adrs>
addxby	reg_adrs	<X> := <X> + <reg_adrs>
subxby	reg_adrs	<X> := <X> – <reg_adrs>
cmpxby	reg_adrs	Compare <reg_adrs> with X by subtracting <X> – <reg_adrs>
addyby	reg_adrs	<Y> := <Y> + <reg_adrs>
subyby	reg_adrs	<Y> := <Y> – <reg_adrs>
cmpyby	reg_adrs	Compare <reg_adrs> withY by subtracting <Y> – <reg_adrs>
addzby	reg_adrs	<Z> := <Z> + <reg_adrs>
subzby	reg_adrs	<Z> := <Z> – <reg_adrs>
cmpzby	reg_adrs	Compare <reg_adrs> with Z by subtracting <Z> – <reg_adrs>

8.20 Operations 8-bit register – memory (SRAM)

Mnemonic	Parameters	Function
adds	reg_adrs, SRAM_adrs	<reg_adrs>:= <reg_adrs> + <SRAM_adrs>
subs	reg_adrs, SRAM_adrs	<reg_adrs>:= <reg_adrs> – <SRAM_adrs>
cmps	reg_adrs, SRAM_adrs	Compare SRAM with register content by subtracting <reg_adrs> – <SRAM_adrs>
addsx	reg_adrs, displacement_literal	<reg_adrs>:=<reg_adrs> + <<X> + displacement_literal>. Base address in X
subsx	reg_adrs, displacement_literal	<reg_adrs>:=<reg_adrs> – <<X> + displacement_literal>. Base address in X
cmpsx	reg_adrs, displacement_literal	Compare SRAM with register content by subtracting <reg_adrs> – <<X> + displacement_literal>. Base address in X
addsy	reg_adrs, displacement_literal	<reg_adrs>:=<reg_adrs> + <<Y> + displacement_literal>. Base address in Y
subsy	reg_adrs, displacement_literal	<reg_adrs>:=<reg_adrs> – <<Y> + displacement_literal>. Base address in Y
cmpsy	reg_adrs, displacement_literal	Compare SRAM with register content by subtracting <reg_adrs> – <<Y> + displacement_literal>. Base address in Y

8.21 Operations 16-bit register – memory (SRAM)

Mnemonic	Parameter	Function
addbyas	SRAM_adrs	<A> := <A> + <SRAM_adrs>. Add a zero-extended byte out of SRAM to A
subbyas	SRAM_adrs	<A> := <A> – <SRAM_adrs>. Subtract a zero-extended byte out of SRAM from A
cmpbyas	SRAM_adrs	Compare a zero-extended byte out of SRAM with A by subtracting <A> – <SRAM_adrs>
addbybs	SRAM_adrs	<B> := <B> + <SRAM_adrs>. Add a zero-extended byte out of SRAM to B
subbybs	SRAM_adrs	<B> := <B> – <SRAM_adrs>. Subtract a zero-extended byte out of SRAM from B
cmpbybs	SRAM_adrs	Compare a zero-extended byte out of SRAM with B by subtracting <B> – <SRAM_adrs>
addwas	SRAM_adrs	<A> := <A> + <SRAM_adrs>. Add a 16-bit word out of SRAM to A
subwas	SRAM_adrs	<A> := <A> – <SRAM_adrs>. Subtract 16-bit word out of SRAM from A
cmpwas	SRAM_adrs	Compare a 16-bit word out of SRAM with A by subtracting <A> – <SRAM_adrs>
addwbs	SRAM_adrs	<B> := <B> + <SRAM_adrs>. Add a 16-bit word out of SRAM to B
subwbs	SRAM_adrs	<B> := <B> – <SRAM_adrs>. Subtract 16-bit word out of SRAM from B
cmpwbs	SRAM_adrs	Compare a 16-bit word out of SRAM with B by subtracting <B> – <SRAM_adrs>

8.22 24-bit and 32-bit operations

The 16-bit register C is extended by a third 8-bit register, dubbed CX3. An arbitrary of the freely available registers may be assigned. Example of an assignment: R2 (.def cx3 = r2).

Mnemonic	Parameters	Function
lit24	SRAM_adrs, 24_bit_literal	<SRAM_Adrs> := 24_bit_literal. Load a 24-bit literal into the SRAM
lit32	SRAM_adrs, 32_bit_literal	<SRAM_Adrs> := 32_bit_literal. Load a 32-bit literal into the SRAM
litab32	32_bit_literal	<A>, <B> := 32_bit_literal. Load a 32-bit literal, the low-order 16-bit word into A, the high-order 16-bit word into B
dec24	SRAM_adrs	<SRAM_adrs> := <SRAM_adrs> – 1. Decrement a 24-bit word in the SRAM by 1
litc3	24_bit_literal	<C>, <CX3> := 24_bit_literal. Load a 24-bit literal into the extended register C
addlitc3	24_bit_literal	<C>, <CX3> := <C>, <CX3> + 24_bit_literal. Add a 24-bit literal to the content of the extended register C
sublitc3	24_bit_literal	<C>, <CX3> := <C>, <CX3> – 24_bit_literal. Subtract a 24-bit literal from the content of the extended register C
cmplitc3	24_bit_literal	Compare a 24-bit literal with the content of the extended register C by subtracting <C>, <CX3> := <C>, <CX3> – 24_bit_literal
inccx		<C>, <CX3> := <C>, <CX3> + 1 . Increment the content of the extended register C by 1
deccx		<C>, <CX3> := <C>, <CX3> – 1. Decrement the content of the extended register C by 1

8.23 Load a register out of flash memory or SRAM

Memory base address in the 16-bit register Z. The memory is selected via bit 15 of the address. Address bit 15 = 0: flash; address bit 15 = 1: SRAM.

Mnemonic	Parameters	Function
lduz	reg_adrs	<reg_adrs> := <<Z>>. Load a byte into the addressed register. Thereafter, the address in Z will be augmented by 1 (post-increment)
lduzd	reg_adrs, displacement_literal	<reg_adrs> := <<Z> + displacement_literal>. The displacement_literal is a 16-bit signed number. The content of Z will not be changed

8.24 Delays

In those macros, microseconds and milliseconds will be implemented by device-specific subroutines.

Mnemonic	Parameter	Function
microsecs	16_bit_literal	Delay the program by n microseconds. n = 16-bit literal
millisecs	16_bit_literal	Delay the program by n milliseconds. n = 16-bit literal
microseca		Delay the program by n microseconds. n = <A> (16 bits)
milliseca		Delay the program by n milliseconds. n = <A> (16 bits)

8.25 Specialties

It is a collection of macro functions found useful in various projects.

Mnemonic	Parameters	Function
absreg	reg_adrs	<reg_adrs> := ABS <reg_adrs> (absolute amount of a number in two's complement notation)
memlitz	literal	<<Z>> = literal. Store the 8-bit literal to the address in the 16-bit register Z. Thereafter, the address in Z will be augmented by 1 (post-increment)
memwlitz	16_bit_literal	<<Z>> = literal. Store the 16-bit literal to the address in the 16-bit register Z. Thereafter, the address in Z will be augmented by 2 (post-increment)
incmem	SRAM_adrs	<SRAM_adrs> := <SRAM_adrs> + 1. Increment the addressed byte by 1
incwmem	SRAM_adrs	<SRAM_adrs> := <SRAM_adrs> + 1. Increment the addressed 16-bit word by 1
decmem	SRAM_adrs	<SRAM_adrs> := <SRAM_adrs> – 1. Decrement the addressed byte by 1
decwmem	SRAM_adrs	<SRAM_adrs> := <SRAM_adrs> – 1. Decrement the addressed 16-bit word by 1
addmemlit	SRAM_adrs, literal	<SRAM_adrs> := <SRAM_adrs> + literal. Add the 8-bit literal to the addressed byte in the SRAM
submemlit	SRAM_adrs, literal	<SRAM_adrs> := <SRAM_adrs> – literal. Subtract the 8-bit literal from the addressed byte in the SRAM
cmpmemlit	SRAM_adrs, literal	Compare a 8-bit literal with the addressed byte in the SRAM by subtracting <SRAM_adrs> – literal

Mnemonic	Parameters	Function
addwmemlit	SRAM_adrs, 16_bit_literal	<SRAM_adrs> := <SRAM_adrs> + literal. Add the 16-bit literal to the addressed 16-bit word in the SRAM
subwmemlit	SRAM_adrs, 16_bit_literal	<SRAM_adrs> := <SRAM_adrs> – literal. Subtract the 16-bit literal from the addressed 16-bit word in the SRAM
cmpwmemlit	SRAM_adrs, literal	Compare a 16-bit literal with the addressed 16-bit word in the SRAM by subtracting <SRAM_adrs> – literal
cmpa	SRAM_adrs	Compare the content of the 8-bit register AL with the addressed byte in the SRAM by subtracting <SRAM_adrs> – <AL>
cmpwa	SRAM_adrs	Compare the content of the 16-bit register A with the addressed word in the SRAM by subtracting <SRAM_adrs> – <A>
pointx	base_adrs, displacement_adrs	Load an address pointer into the X register. <X> := base_adrs + <displacement_adrs>. The base address is a literal. The displacement address points to the displacement value in the SRAM
pointy	base_adrs, displacement_adrs	Load an address pointer into the Y register. <Y> := base_adrs + <displacement_adrs>. The base address is a literal. The displacement address points to the displacement value in the SRAM
pointz	base_adrs, displacement_adrs	Load an address pointer into the Z register. <Z> := base_adrs + <displacement_adrs>. The base address is a literal. The displacement address points to the displacement value in the SRAM
textadrs	16_bit_literal	Load an immediate value into the 16-bit register Z so it can be used as a byte address in LPM instructions

8.26 Save and restore register contents

Those macros come handy to save and restore register contents. Fig. 5 shows, how the register contents are arranged within the stack.

Mnemonic	Parameter	Function
pushf		Push the status register (flagbits) onto the stack
popf		Retrieve the status register (flagbits) from the stack
pushft		Push the status register (flagbits) and the TEMP register onto the stack
popft		Retrieve the status register (flagbits) and the TEMP register from the stack

Mnemonic	Parameter	Function
pushft10		Push the status register (flagbits), the TEMP register, and the registers r0, r1 onto the stack
popft10		Retrieve the status register (flagbits) and the TEMP register, and the registers r0, r1 from the stack
pushaf		Push the status register (flagbits), the TEMP register, and the 16-bit register A onto the stack
popaf		Retrieve the status register (flagbits), the TEMP register, and the 16-bit register A from the stack
pushaf10		Push the status register (flagbits), the TEMP register, the registers r0, r1, and the 16-bit register A onto the stack
popaf10		Retrieve the status register (flagbits), the TEMP register, the registers r0, r1, and the 16-bit register A from the stack
pushregs		Push all registers belonging to the virtual machine onto the stack
popregs		Retrieve all registers belonging to the virtual machine from the stack

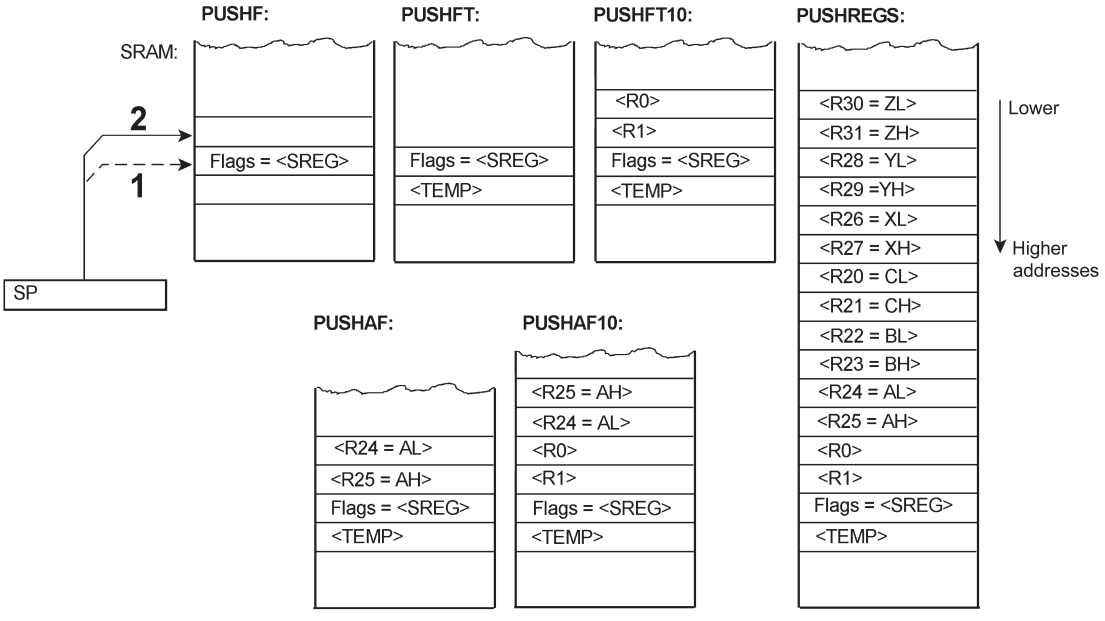


Fig. 5 How the register contents are saved in the stack. The AVR stack pointer SP points to the first free byte above the top of stack (TOS). 1 - before, 2 - after execution of the macro. For more details of stack operation, see paragraph 8.29, Basic stack operations.

8.27 Supporting stack machine emulation

Some macros are auxiliary functions to make good use of the stack; some have been inspired by the Forth programming language.

Mnemonic	Parameters	Function
pusha		Push the 16-bit register A onto the stack
pushb		Push the 16-bit register B onto the stack
pushc		Push the 16-bit register C onto the stack
pushx		Push the 16-bit register X onto the stack
pushy		Push the 16-bit register Y onto the stack
pushz		Push the 16-bit register Z onto the stack
popa		Retrieve the 16-bit register A from the stack
popb		Retrieve the 16-bit register B from the stack
popc		Retrieve the 16-bit register C from the stack
popx		Retrieve the 16-bit register X from the stack
popy		Retrieve the 16-bit register Y from the stack
popz		Retrieve the 16-bit register Z from the stack
pushlit	literal	Push an 8-bit literal onto the stack
pushwlit	16_bit_literal	Push a 16-bit literal onto the stack
fetch		Retrieve a 16-bit address from the stack. Read a byte and push it onto the stack. The 16-bit register X contains the address, incremented by 1 (next sequential address)
fetchw		Retrieve a 16-bit address from the stack. Read a word and push it onto the stack. The 16-bit register X contains the address, incremented by 1 (next sequential address)
store		Store the byte on TOS at the address on TOS+1. Remove both from stack. The 16-bit register X contains the address, incremented by 1 (next sequential address)
storew		Store the 16-bit word on TOS at the address on TOS+2. Remove both from stack. The 16-bit register X contains the address, incremented by 2 (next sequential address)
drop		Remove a byte from the stack

Mnemonic	Parameters	Function
dropw		Remove a 16-bit word from the stack
swaps		Swap both topmost bytes on the stack and load the new byte on TOS into the register AL
swapsw		Swap both topmost 16-bit words on the stack and load the new word on TOS into the register A
dup		Push the topmost byte on the stack onto the stack again and load it into the 8-bit register AL
dupw		Push the topmost 16-bit word on the stack onto the stack again and load it into the 16-bit register A
over		Push the byte on TOS+1 onto the stack again and load ite into the register AL
overw		Push the 16-bit word on TOS+2 onto the stack again and load this word into the register A
fetchr	reg_adrs	Retrieve a 16-bit address from the stack. Read a byte and load into the register selected by reg_adrs. The 16-bit register X contains the address, incremented by 1 (next sequential address)
fetchwa		Retrieve a 16-bit address from the stack. Read a 16-bit word and load into the register A. The 16-bit register X contains the address, incremented by 2 (next sequential address)
fetchwb		Retrieve a 16-bit address from the stack. Read a 16-bit word and load into the register B. The 16-bit register X contains the address, incremented by 2 (next sequential address)
fetchwc		Retrieve a 16-bit address from the stack. Read a 16-bit word and load into the register C. The 16-bit register X contains the address, incremented by 2 (next sequential address)
fetchwy		Retrieve a 16-bit address from the stack. Read a 16-bit word and load into the register Y. The 16-bit register X contains the address, incremented by 2 (next sequential address)
fetchwz		Retrieve a 16-bit address from the stack. Read a 16-bit word and load into the register Z. The 16-bit register X contains the address, incremented by 2 (next sequential address)
storr	reg_adrs	Retrieve a 16-bit address from the stack. Store the content of the register selected by reg_adrs. The 16-bit register X contains the address, incremented by 1 (next sequential address)

Mnemonic	Parameters	Function
storwa		Retrieve a 16-bit address from the stack. Store the content of the 16-bit register A. The 16-bit register X contains the address, incremented by 2 (next sequential address)
storwb		Retrieve a 16-bit address from the stack. Store the content of the 16-bit register B. The 16-bit register X contains the address, incremented by 2 (next sequential address)
storwc		Retrieve a 16-bit address from the stack. Store the content of the 16-bit register C. The 16-bit register X contains the address, incremented by 2 (next sequential address)
pusheab	SRAM_adrs, 16_bit_literal	Push an effective address onto the stack, calcultaed by adding the 16-bit literal to the base address. Effective adrs := <SRAM_Adrs> + 16-bit literal. The SRAM_adrs parameter points to a word in the SRAM containing the base address
pusheabd	SRAM_adrs, displacement_adrs	Push an effective address onto the stack, calcultaed by adding a 16-bit displacement out of the SRAM to the base address. Effective adrs :=<SRAM_Adrs> + <displacement_adrs>. The SRAM_adrs parameter points to a word in the SRAM containing the base address; the displacement_adrs points to a word in the SRAM containing the displacement
pusheax	16_bit_literal	Push an effective address onto the stack, calcultaed by adding the 16-bit literal to the content of the X register. Effective adrs := <X> + 16-bit literal.
pusheay	16_bit_literal	Push an effective address onto the stack, calcultaed by adding the 16-bit literal to the content of the Y register. Effective adrs := <Y> + 16-bit literal.
pusheaz	16_bit_literal	Push an effective address onto the stack, calcultaed by adding the 16-bit literal to the content of the Z register. Effective adrs := <Z> + 16-bit literal.

8.28 Subroutine call and parameter addressing

Those macros support passing parameters, calling subroutines, and creating a runtime environment for subroutine execution. For details of stack operation, see paragraph 8.29, Basic stack operations.

Notes:

- There are pairs of macros:
  - STACKINDEXX and CLRSTACKRETX
  - STACKINDEXY and CLRSTACKRETY
  - ENTER and LEAVE
  - AVR_ENTER and AVR_LEAVE

- All macros are concerned with parameter passing in the stack. The basic program sequence: PUSH 1st parameter, PUSH 2nd parameter and so on, then call the subroutine.
- The STACKINDEX / CLRSTACKRETX macros are to be used when the subroutine should be simply called, without creating a stack frame (assembler-style).
- STACKINDEXX or STACKINDEXY must be invoked at the begin of the subroutine.
- CLRSTACKRETX or CLRSTACKRETY are to be used instead of a RET instruction to return to the calling program.
- CLRSTACKRETX or CLRSTACKRETY expects that the register X or Y contains what the preceding STACKINDEXX or STACKINDEXY has been entered.
- Only the address registers Y and Z support the addressing mode base + displacement (to be invoked by LDD instructions). This is a property of the AVR architecture.
- When invoking STACKINDEXY / CLRSTACKRETY, the register Y is to be used to access stack content via LDD and STD instructions.
- When invoking STACKINDEXX / CLRSTACKRETX, it may be required to move the content of the address register X to the address register Y or Z, so that the base address pointing to the stack content can be used with LDD or STD instructions.
- The ENTER / LEAVE macros are to be used when a stack frame is to be generated (C-style; according to the industry standard set by the conventions of the C programming language and the Unix operating system).
- Invoke ENTER / LEAVE, if the stack frame should meet this widespread industry standard. Accessing the stack frame content, however, requires signed displacements.
- Invoke AVR_ENTER / AVR_LEAVE, if the stack frame should be accessible by LDD and STD instructions (which support only positive (unsigned) displacements).

Mnemonic	Parameters	Function
stackindexx		The X register will be loaded with an address pointing to the first byte in the stack preceding the saved return address, thus allowing to address parameters which have been passed in the stack
clrstackretx	No_of_bytes (to be removed from stack)	Remove bytes from the stack and return from the subroutine. According to the parameter, a certain number of bytes below the saved return address will be removed from the stack. X will contain the content of the stack pointer; Y will contain the return address
stackindexy		The Y register will be loaded with an address pointing to the first byte in the stack preceding the saved return address, thus allowing to address parameters which have been passed in the stack
clrstackrety	No_of_bytes (to be removed from stack)	Remove bytes from the stack and return from the subroutine. According to the parameter, a certain number of bytes below the saved return address will be removed from the stack. Y will contain the content of the stack pointer; X will contain the return address.

Mnemonic	Parameters	Function
enter	No_of_bytes (needed for local variables)	Entry into a function according to the C/Unix conventions. The register Y serves as the frame pointer (FP). Its content is pushed onto the stack. The stack pointer SP becomes the new frame pointer. Thereafter, SP will be decremented by the parameter to make room for the local variables
leave		Leave a function according to the C/Unix conventions. The register Y serves as the frame pointer (FP). The frame pointer becomes the new stack pointer SP. Thereafter, the FP will be restored out of the stack and a return from subroutine will be executed
AVR_enter	No_of_bytes (needed for local variables)	Like ENTER, but with the address in the frame pointer (Y) corrected, so that the öocal variables and the parameters within the stack frame can be accessed by positive displacements
AVR_leave	No_of_bytes (needed for local variables)	Like LEAVE, but the counterpart to AVR_ENTER, compensating for the address correction

8.29 Basic stack operations

The macros rely on the built-in stack mechanism. In contrast to other architectures, however, the AVR stack pointer (SP) addresses not the uppermost byte in the stack, in other words, the Top of Stack (TOS), but the first free byte above TOS (Fig. 6).

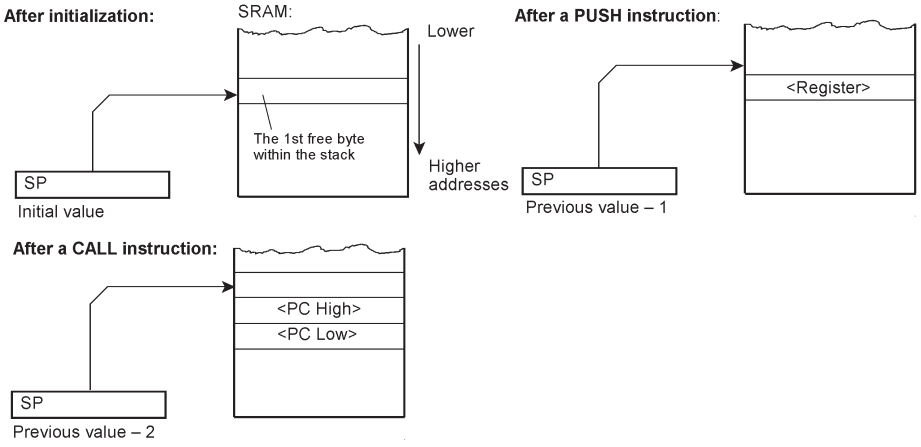


Fig. 6 The AVR stack. It is shown, how stack-related instructions push register contents and return addresses onto the stack. The stack grows in the direction to lower SRAM addresses.

Caution: Naive students of the AVR architecture would arguably consider the AVR a little-endian machine. The high-order byte of the saved PC content, however, is to be found within the stack at the lower SRAM address. In contrast, the macros push 16-bit word onto the stack according to the byte order of a little-endian machine (Fig. 7).

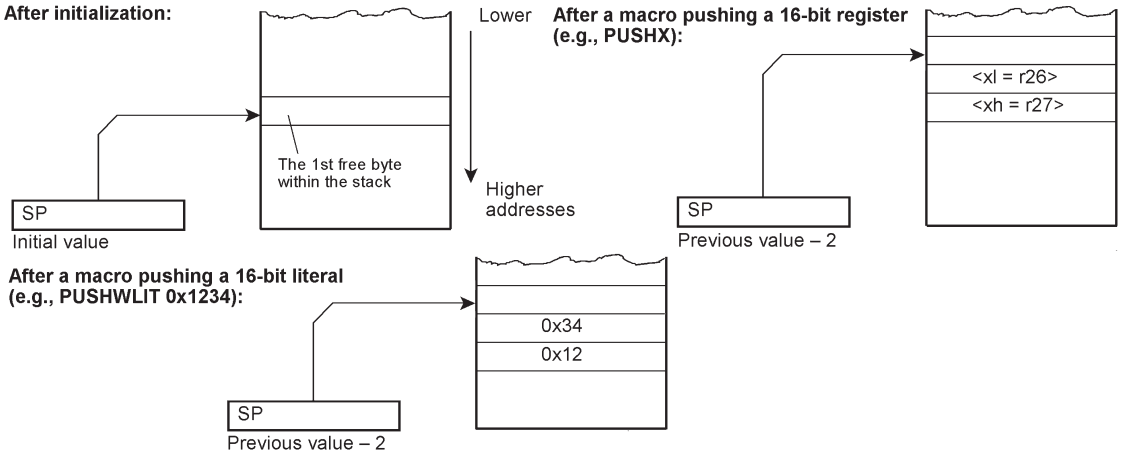


Fig. 7 The macros ensure the byte order of a little-endian machine (the high-order byte at the higher SRAM address).

Fig. 8 illustrates a venerable principle of passing the parameters and calling a subroutine. The parameters are pushed onto the stack. Thereafter, the subroutine is called. The problem is to address the parameters in the stack below the saved return address. Here the macros STACKINDEXX and STACKINDEXY come handy (Fig. 9).

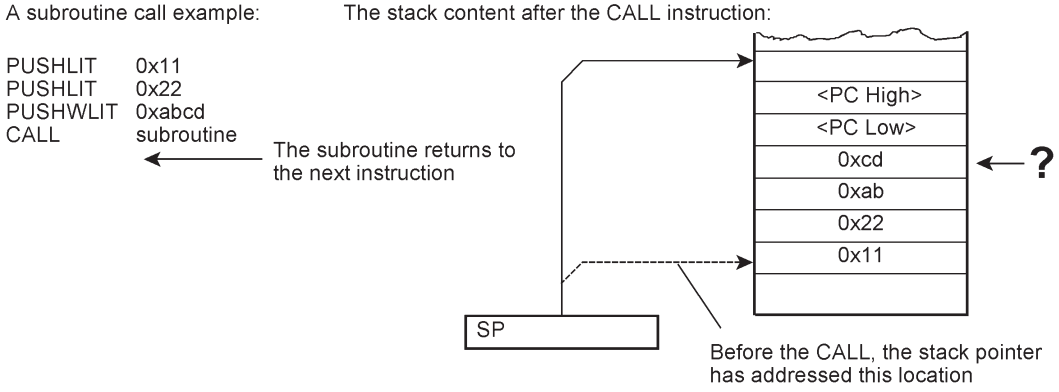


Fig. 8 How to access the parameters? This address is to be loaded into the address register Y or Z. Thereafter, parameters can be addressed via displacements (in other words, with appropriate LDD instructions).

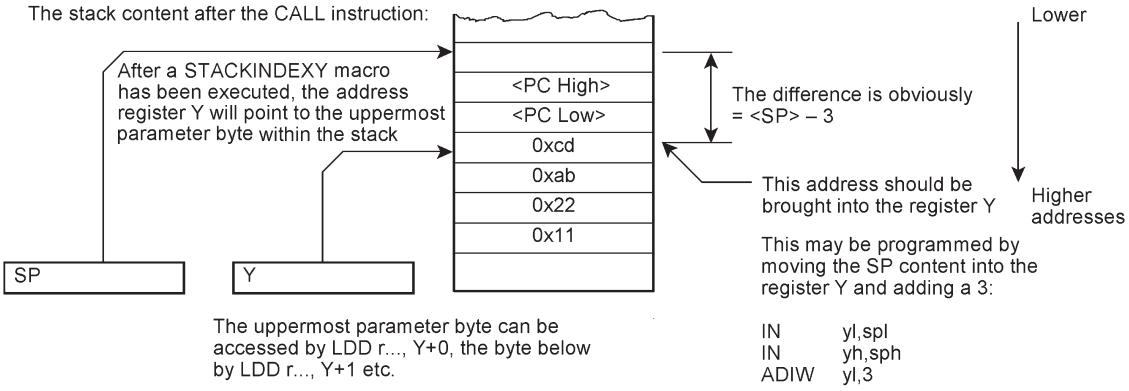


Fig. 9 How the macros STACKINDEXX and STACKINDEXY work. Here STACKINDEXY is used as an example.

When this parameter passing technique has been applied, it is not possible to end subroutine execution by a simpe RET instruction, because the parameters would remain on the stack. Instead of a RET, one of the macros CLRSTACKRETX or CLRSTACKRETY may be used (Fig. 10).

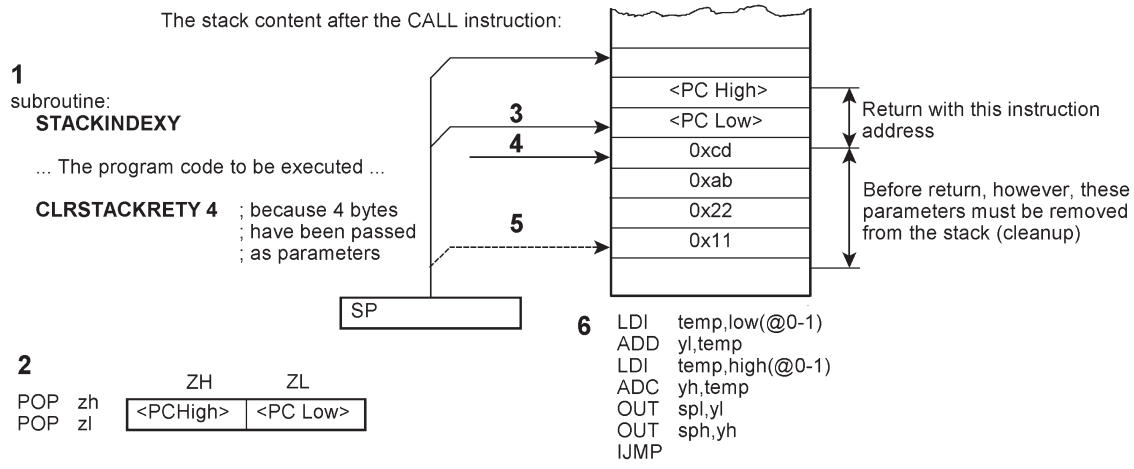
Caution: Macros accessing and manipulating the stack pointer SP may not be interrupted. The macros comply with this requirement by disabling the interrupts on entry, and by restoring the previous status (enabled or disabled, respectively) on exit.

Build and relinquish a stack frame

Stack frames are typical of the runtime environment of high-level languages. A stack frame accommodates the parameters, the return address, the local variables, and a stack area available to the particular subroutine. To address the parameters and local variables, the stack pointer SP is supplemented by an additional address register, the so-called base or frame pointer FP. Appropriate subroutines begin with an ENTER macro. Before returning to the calling program, they invoke an LEAVE macro. ENTER will build the stack frame, LEAVE will relinquish it. However, LEAVE dos not remove the parameters. Who the final cleanup is obliged to do, depends on the type of runtime environment. According to the C/Unix convention, the calling program must purge the stack (e.g., by some POP instructions). According to the Pascal convention, the called subroutine must do the work. To this end, instead of a RET instruction, a macro CLEARSTACKRETX or CLEARSTACKRETY could be invoked.

A widespread industry standard

The frame pointer points to a stack location above the saved return address. In this location, the frame pointer of the preceding stack frame (old frame pointer) will be saved. Above the saved old frame pointer, the local variables are stored. Still further above begins the freely available stack area, addressed by the stack pointer SP. With the frame pointer as the base address register, the parameters are accessible by positive, the local variables by negative displacements.



1 The principal layout of the subroutine

2 Two POP instructions load the return address into the address register Z

3 As a consequence of the two POPs, the SP will point to this location

4 The address register Y points to the first parameter byte

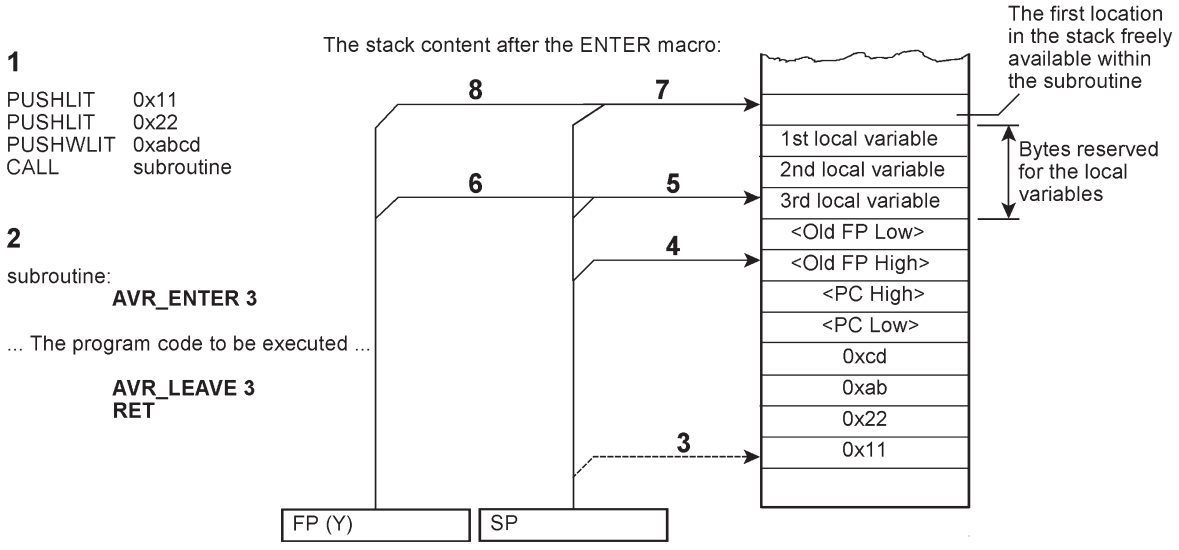
5 Before the call, the SP had been pointing to this address. After the return, the SP must point to the same address

6 To calculate the stack address before the subroutine call, the number of parameter bytes – 1 is to be added to the parameter address. Then the address is loaded into the stack pointer, and the return will be executed by an indirect jump instruction with the return address in the address register Z

Fig. 10 Returning form the subroutine by invoking a macro CLEARSTACKRETX or CLEARSTACKRETY. Here CLEARSTACKRETY is used as an example. When invoking the macro, the stack pointer SP must point to the return address.

An AVR solution

The AVR's load and store instructions (LDD, STD) feature only positive displacements. Hence not only the parameters must be accessible by positive displacements, but the local variables, too. To this end, the AVR_ENTER macro decrements the frame pointer by the number of bytes reserved for the local variables (Fig. 11). Consequently, the AVR_LEAVE macro must be invoked with the same parameter to increment the frame pointer accordingly.



1 Passing the parameters and calling the subroutine

2 The principal layout of the subroutine. In the example, it requires 3 bytes to accomodate the local variables

3 Before the call, the SP had been pointing to this address. After the return, the SP must point to the same address

4 When entering the subroutine, the SP will point to this location, with the saved return address straight beneath

5 The old frame pointer (FP) has been pushed onto the stack

6 The new frame pointer will address the same stack location like the stack pointer (SP => FP)

7 The stack pointer has been decremented by 3 to make room for the local variables. When an industry-standard stack frame is to be created, the ENTER macro ends here.

8 To accommodate the stack frame to the peculiarities of the AVR architecture, the frame pointer has to be decremented by 3, too (SP => FP). So not only the parameters but also the local variables may be addressed by positive displacements (using instructions LDD and STD with the frame pointer FP (= register Y) as the base address register).

Fig. 11 Creating a stack frame in C/Unix style.

9. Alternative macros

16-bit registers passed as parameters

This is the principal alternative. To facilitate this kind of parameter passing, particular register mnemonics must be created, and unique numeric values must be assigned to. This prerequisite allows writing macro bodies selecting appropriate instructions via conditional statements. These principles have been depicted already in paragraph 3 (cf. pages 7 and 8). The 16-bit registers can be christened and numbered arbitrarily. The whole register file of the AVR architecture could be seen as 16 16-bit registers, named, for example, as R0W, R2W, R4W and so on, up to R30W. The 16-bit registers of the virtual machine could be named as AW, BW, CW, XW, YW, and ZW. In principle, all corresponding macros could be written for all of the sixteen 16-bit registers. It is only a question of industriousness (to write macro bodies with 16 conditional statements and 16 instruction sequences). Therefore, some of the example macros have been written only with regard to the registers AW to ZW. For details, refer to the source code.

Basic transports (registers R0W to R30W, AW, BW, CW, XW, YW, ZW)

Those macros are basic move operations dealing with general addresses. They keep care of the address space (data memory or I/O) and of the byte order (low-order or high-order byte first).

Mnemonic	Parameters	Function
gstw	general_adrs, 16_bit_reg	Store the contents of the 16-bit register as a 16-bit word at a general address
gldw	16_bit_reg, general_adrs	Load the 16-bit register with the 16-bit word at a general address

Access 16-bit words in SRAM by indirect addressing (registers AW, BW, CW, XW, YW, ZW)

Those macros extend indirect addressing beyond the corresponding provisions of the AVR architecture (i.e., the addressing capabilities of the registers X, Y, Z). The *base_adrs* parameter points to a 16-bit word in the SRAM containing the operand address or the base address a displacement will be added to. The displacement parameter is either a literal or an address pointing to a 16-bit word in SRAM containing the displacement value. The displacement in those macros is always a signed 16-bit number. Refer to Fig. 4 (page 30). The macro functions are the same as those provided for the 8-bit registers (cf.the previous paragraph 8.12), except the increments and decrements are by 2 (because words are addressed).

.

Mnemonic	Parameters	Function
ldiw	16_bit_reg, base_adrs	Load indirect. <16-bit reg> := <<base_adrs>> Cf. Fig. 4a
ldiwinc	16_bit_reg, base_adrs	Load indirect. Post-increment the address pointer. <16-bit reg> := <<base_adrs>>; <base_adrs> := <base_adrs> + 2. Cf. Fig. 4b
ldiwdec	16_bit_reg, base_adrs	Pre-decrement the address pointer. Then load indirect. <base_adrs> = <base_adrs> – 2; <16-bit reg> := <<base_adrs>>. Cf. Fig. 4b
ldiwblit	16_bit_reg, base_adrs, displacement_literal	Load indirect. <16-bit reg> := <<base_adrs> + displacement_literal. Cf. Fig. 4c
ldiwbd	16_bit_reg, base_adrs, displacement_adrs	Load indirect. <16-bit reg> := <<base_adrs> + <displacement_adrs>>. Cf. Fig. 4d
stiw	base_adrs, 16_bit_reg	Store indirect. <<base_adrs>> := <16-bit reg>. Cf. Fig. 4a
stiwinc	base_adrs, 16_bit_reg	Store indirect. Post-increment the address pointer. <<base_adrs>> := <16-bit reg> ; <base_adrs> := <base_adrs> + 2 . Cf. Fig. 4b
stiwdec	base_adrs, 16_bit_reg	Pre-decrement the address pointer. Then store indirect. <base_adrs> := <base_adrs> – 2; <<base_adrs>> := <16-bit reg>. Cf. Fig. b
stiwblit	base_adrs, displacement_literal, 16_bit_reg	Store indirect. <<base_adrs> + displacement_literal> := <16-bit reg>. Cf. Fig. 4c
stiwbd	base_adrs, displacement_adrs, 16_bit_reg	Store indirect. <<base_adrs> + <displacement_adrs>> := <16-bit reg>. Cf. Fig. 4d

Clear a 16-bit register (registers R0W to R30W, AW, BW, CW, XW, YW, ZW)

Mnemonic	Parameter	Function
clearw	16_bit_reg	Clear the 16-bit register

Load or store a 16-bit word (registers AW, BW, CW, XW, YW, ZW)

Mnemonic	Parameter	Function
ldw	16_bit_reg, general_mem_adrs	Load the 16-bit register
ldbyw	16_bit_reg, SRAM_adrs	Load the 16-bit register with a zero-extended byte from the SRAM
ldxw	16_bit_reg	Load the 16-bit register. General memory address in X
ldyw	16_bit_reg	Load the 16-bit register. General memory address in Y
ldzw	16_bit_reg	Load the 16-bit register. General memory address in Z
stw	16_bit_reg	Store the content of the 16-bit register
stxw	16_bit_reg	Store the content of the 16-bit register General memory address in X
styw	16_bit_reg	Store the content of the 16-bit register General memory address in Y
stzw	16_bit_reg	Store the content of the 16-bit register General memory address in Z
ldxincw	16_bit_reg	Load the 16-bit register. General memory address in X. Post-increment the address pointer in X by 2
ldyincw	16_bit_reg	Load the 16-bit register. General memory address in Y. Post-increment the address pointer in Y by 2
stxincw	16_bit_reg	Store the content of the 16-bit register. General memory address in X. Post-increment the address pointer in X by 2
styincw	16_bit_reg	Store the content of the 16-bit register. General memory address in Y. Post-increment the address pointer in Y by 2

Load a 16-bit literal (registers R0W to R30W, AW, BW, CW, XW, YW, ZW)

Mnemonic	Parameters	Function
litw	16_bit_reg, 16_bit_literal	Load the literal into the 16-bit register (<16-bit reg> := literal)

Operations with 16-bit literals (registers AW, BW, CW, XW, YW, ZW)

Mnemonic	Parameters	Function
addlitw	16_bit_reg, 16_bit_literal	Add the literal to the content of the 16-bit register (<16-bit reg> + literal)
sublitw	16_bit_reg, 16_bit_literal	Subtract the literal from the content of the 16-bit register (<16-bit reg> – iteral)
complitw	16_bit_reg, 16_bit_literal	Compare literal with the content of the 16-bit register (by subtracting <16-bit reg> – literal)
andlitw	16_bit_reg, 16_bit_literal	Bitwise AND between the content of the 16-bit register and the literal
orlitw	16_bit_reg, 16_bit_literal	Bitwise OR between the content of the 16-bit register and the literal
xorlitw	16_bit_reg, 16_bit_literal	Bitwise XOR between the content of the 16-bit register and the literal
zerow	16_bit_reg,	Test whether the content of the 16-bit register is zero