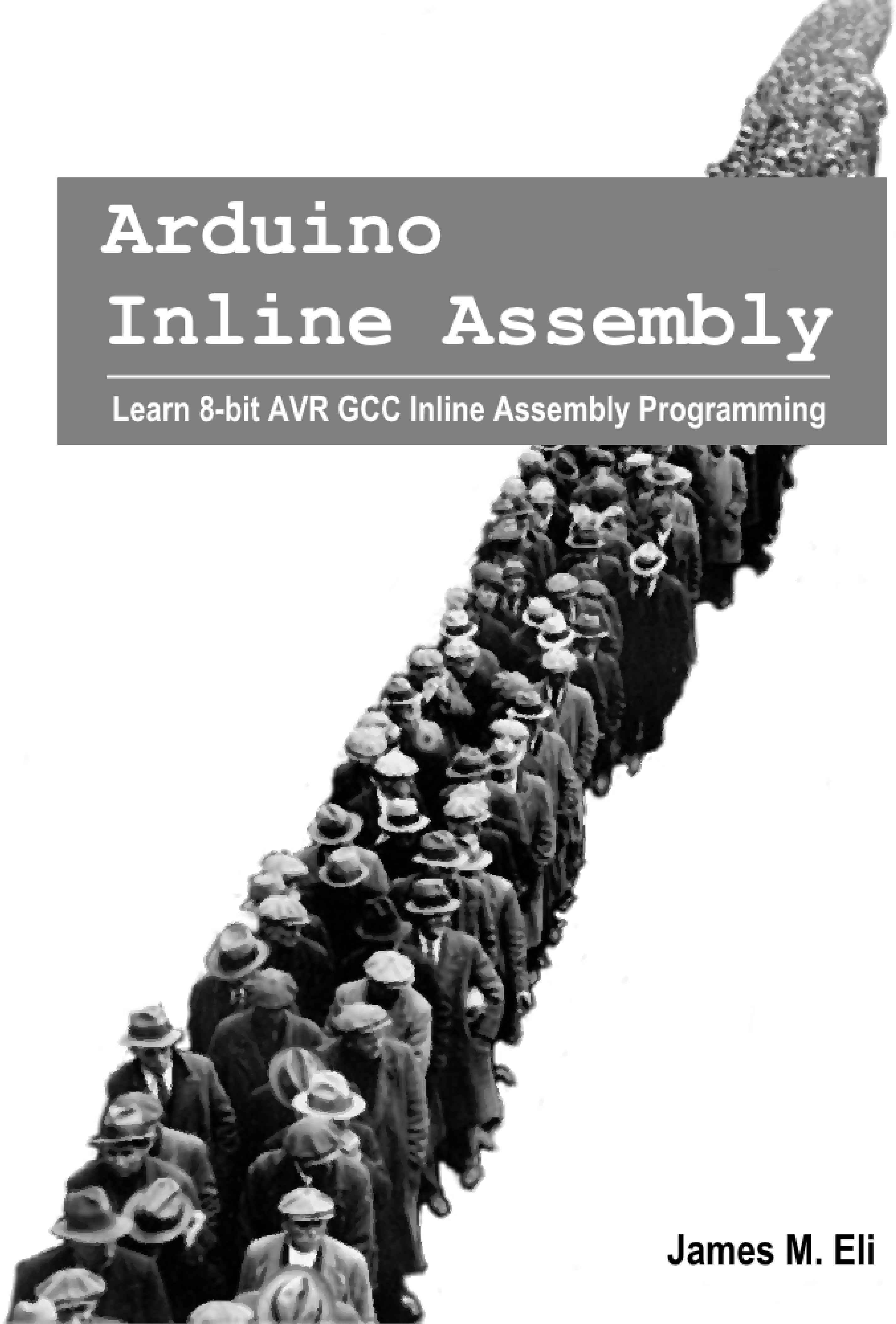
Arduino Inline Assembly

Learn 8-bit AVR GCC Inline Assembly Programming



James M. Eli

Table of Contents

Preface

Motivation

Learning inline assembly language on the Arduino AVR 8-bit platform is a daunting task for many (at least it was for me). Besides the cryptic syntax and the high level of understanding the semi-official documentation assumes, there exists very little information about GCC inline assembler coding. The main focus of existing documentation is the "Cookbook", which dates from 2002. Trust me when I state, any neophyte needs to spend hours searching and studying piece-meal examples while possessing overwhelming patience in order to crack the code.

At least I did.

Hopefully this series of tutorials will help alleviate many of the discouraging troubles I encountered while teaching myself inline assembly coding. An Arduino Inline Assembly Tutorial was long overdue!

Who This Book is for

This book has been written primarily for individual with one of the following qualifications:

- Knowledge of the Arduino and/or programmed Arduino inside the Arduino IDE
- Experienced C language embedded programmer

It is important to realize this book assumes at least a basic understanding of the underlying Arduino hardware (ports, timers, interrupts, USART, I2C, SPI, etc.).

Hopefully the reader has already written programs for the Arduino utilizing the functionality provided by the IDE, or has a general understanding of microcontroller programming. A wealth of information on the Arduino boards and software is available at the website:

`http://www.arduino.cc/`

No attempt is made in this book to teach how to interface to the underlying hardware of the Arduino AVR microcontroller. If a reader needs any specifics or details about the hardware, they should be comfortable with consulting the ATMEL datasheet. Furthermore, familiarity with the GCC compiler, assembler and/or general compiler functionality is helpful but not required.

Why Put Yourself Through this?

Why learn inline assembly language programming? Many experts loudly proclaim that current optimizing compilers are capable of producing code that is as good, if not better than hand-coded assembly language. However, keep in mind a large percentage of the AVR Libc library is hand-coded assembly. That should be something that at least makes you go, “hmmm?”

Any serious development on the Arduino platform, or an ATMEL based 8-bit AVR microcontroller requires an examination of the underlying assembly code produced by the compiler. This code should be checked for correctness, bugs, and inefficiencies. If nothing else, learning AVR assembly language will give you the knowledge to determine what is going on deep inside of your program.

There are times when using assembly is needed, and however, many cases when it is not. Here are some advantages to using inline assembly:

- Easy access to machine-dependent registers, I/O and features.
- Control exact code behavior in critical sections preventing conflicts between software or hardware devices.
- Override standard conventions of your compiler, which might allow optimization (in memory allocation, calling conventions, etc.).
- Build interfaces between code fragments using non-standard conventions (i.e. produce a separate low-level interface).
- Ease access to non-typical programming modes of your processor.
- Produce reasonably fast code for tight loops, etc.
- Produce hand-optimized code perfectly tuned for your particular hardware setup.
- Gain complete control of your code.

However, assembly is a very low-level language (the lowest above hand-coding the binary instructions). This means,

- It is long and tedious to write initially.
- It is bug-prone, and bugs can be very difficult to find.
- Assembly code can be fairly difficult to understand, modify, and maintain.
- Assembly code is not portable to other architectures.
- It causes the programmer to spend more time on a few details.
- Small changes in design may completely invalidate existing assembly code.
- It is very difficult to use complex data structures in assembly.
- Many times an optimizing compiler outperforms hand-coded assembly.

Here are some general guidelines for including inline assembly code:

- Minimize the use and amount of assembly code.
- Encapsulate the code in well-defined functions or blocks.
- Incorporate higher-level language MACROs to automatically generate assembly code.
- Thoroughly debug your code.

Even when assembly is absolutely required, you'll probably find that you can get away with far less than you initially think need.

Legal Information

Although the electronic design of the Arduino boards is open source (Creative Commons CC-SA-BY License) the Arduino name, logo and the graphics design of its boards is a protected trademark of Arduino LLC.

Atmel®, the Atmel logo and combinations thereof, and others are the registered trademarks or trademarks of Atmel Corporation of its subsidiaries.

Chapter 4

Extended asm

The Extended asm Statement

The general form of an extended inline assembler statement is:

```
asm(“code” : output operand list : input operand list : clobber list);
```

This statement is divided by colons into (up to) four parts. While the code part is required, the others are optional:

1. Code: the assembler instructions, defined as a single string constant.
2. A list of output operands, separated by commas.
3. A list of input operands, separated by commas.
4. A list of “clobbered” or “accessed” registers.

For now, we are going to ignore parts 2 through 4 and concentrate on the code part of the statement.

Our First Inline Assembly Program

This is a basic example of storing a value in memory.

```
volatile byte a=0;

void setup() {
  Serial.begin(9600);
  asm (
    "ldi r26, 42  \n" //load register r26 with 42
    "sts (a), r26 \n" //store r26 in a’s memory location
  );

  Serial.print("a = ");
  Serial.println(a);
}

void loop() { }
```

To fully understand what we are doing here first we need to cover some background.

Memory

The Arduino has 3 basic types of memory, flash, SRAM and EEPROM. Flash is the memory where our program is stored. Arduino flash is non-volatile storage[1] where data can be retrieved even after power has been cycled (turned off and then back on). When you upload an Arduino program, it gets loaded into flash memory.

EEPROM[2] is a form of non-volatile storage used for variable data that we want to maintain between operations of our program.

SRAM is the memory used to store variable information and data. SRAM is volatile storage[3], and anything placed here is immediately lost when power is removed. In our program above, variable a is stored in SRAM. For now, we are not concerned about EEPROM or Flash memory, and will concentrate on SRAM.

Registers Are Memory Too

Registers are special SRAM memory locations. The Arduino has 32 general purpose registers (labeled *r0* to *r31*). Each register is 8-bits, or 1-byte in size. *r0* occupies SRAM position 0, with each register following incrementally through SRAM position 31. These 32 general purpose registers are important because the Arduino cannot operate (change, do math, compare values, etc.) directly upon memory. Values stored in memory must first be loaded into a register(s).

[1] See https://en.wikipedia.org/wiki/Non-volatile_memory.
[2] See https://en.wikipedia.org/wiki/EEPROM.
[3] See https://en.wikipedia.org/wiki/Volatile_memory.

As you can imagine, the Arduino has more than 32 registers. Registers also have lots more specific details about them, but for now, these first 32 are more than enough.

Assembly Instructions

Our inline assembly consisted of just two instructions, LDI and STS:

```
     "ldi r26, 42  \n"
     "sts (a), r26 \n"
```

The LDI instruction is a mnemonic for "*L*oa*D I*mmediate". This instruction simply loads an 8 bit constant value directly into a register. The register must be between 16 and 31. For example, trying to use register #1, *r1* with the LDI instruction would cause an error.

In our program, we load the value "42" into register #26, *r26* (pedantically we take note, *r26* is actually the 27th register, since numbering starts at zero). We could have chosen *r18*, *r19* or even *r24* for that matter. Later, our register selection will become crucial, but for now #26 seems like a good choice.
The LDI instruction is followed by an STS instruction. STS is a mnemonic for "*ST*ore direct to data *S*pace". STS stores one byte from a Register into SRAM, the data space. In our inline code, we place the contents of register #26, *r26* into the memory location of variable a. Quietly, behind the scenes, the assembler replaces "a" with the memory location of a. Neat.

Our program finishes by printing the contents of variable location a through the Serial Terminal. Hopefully this produces the output:

```
a = 42
```

A Few Program Caveats

The variable must be global in scope. This causes the compiler to locate the variable inside SRAM. If we declared the variable a inside of the setup() function, it may have been stored temporarily on the stack or inside a register. If that was the case, the STS instruction would have caused an error because it only works when storing to SRAM. The stack[4] is a subject of a different chapter.

Furthermore, since the default Arduino compilation uses the *–Os* optimization level, we must declare it "volatile", so the optimizer doesn't eliminate it. The optimizer is very good at what it does, like when it sees code that's not necessary, it will artfully remove it. In our case here, we don't want this code to be removed.

[4] See https://ucexperiment.wordpress.com/2015/01/02/arduino-stack-painting/.

More to Come

With these two beginning chapters we have covered the fundamentals of the inline assembler, basic syntax and a few assembler instructions. We will continue looking at the extended inline assembler, introducing new instructions as we go. Next, we'll look at the "clobber list" then dive into the bizarre world of input and output operands.

Appendix D

Port and Pin Compendium

The following is a compendium of inline assembly functions dealing with ports and pins. Use these at your own risk. These functions have been trimmed of most bounds checking, so they can easily be abused.

analogWrite

This inline code writes an analog value (in the form of a PWM wave) to a particular pin. After executing, the pin will generate a steady square wave of the specified duty cycle until the next call (or call to digitalRead() or digitalWrite() on the same pin). The frequency of the PWM signal on most pins is approximately 490 Hz. On the Uno and similar boards, pins 5 and 6 have a frequency of approximately 980 Hz. On Arduino boards with the ATmega168/328, this function works on pins 3, 5, 6, 9, 10, and 11. The analogWrite function has nothing to do with the analog pins or the analogRead function.

A pinMode() call is included inside this function, so there is no need to set the pin as an output before executing this code.

This version of AnalogWrite, with no frills saves ~542 bytes over the built-in function:

```
//analogWrite requires a PWM pin
//PWM pin/timer table:
//3:  (TIMER2B) PD3/TCCR2A/COM2B1/OCR2B
//5:  (TIMER0B) PD5/TCCR0A/COM0B1/OCR0B
```

```
//6:  (TIMER0A) PD6/TCCR0A/COM0A1/OCR0A
//9:  (TIMER1A) PB1/TCCR1A/COM1A1/OCR1A
//10: (TIMER1B) PB2/TCCR1A/COM1B1/OCR1B
//11: (TIMER2A) PB3/TCCR2A/COM2A1/OCR2A
//set below 6 defines per above table
#define ANALOG_PORT         PORTB
#define ANALOG_PIN          PORTB3
#define ANALOG_DDR          DDRB
#define TIMER_REG           TCCR2A
#define COMPARE_OUTPUT_MODE COM2A1
#define COMPARE_OUTPUT_REG  OCR2A

volatile uint8_t val = 128; //0-255

  asm (
    "sbi  %0, %1   \n" //DDR set to output (pinMode)

    "cpi  %6, 0    \n" //if full low (0)
    "breq _SetLow  \n"
    "cpi  %6, 0xff \n" //if full high (0xff)
    "brne _SetPWM  \n"

    "sbi  %2, %1   \n" //set high
    "rjmp _SkipPWM \n"

  "_SetLow:        \n"
    "cbi  %2, %1   \n" //set low
    "rjmp _SkipPWM \n"

  "_SetPWM:        \n"
    "ld   r24, X   \n"
    "ori  r24, %3  \n"
    "st   X, r24   \n" //connect pwm pin timer# & channel
    "st   Z, %6    \n" //set pwm duty cycle (val)

  "_SkipPWM:       \n"

    : : "I" (_SFR_IO_ADDR(ANALOG_DDR)), "I" (ANALOG_PIN),
    "I" (_SFR_IO_ADDR(ANALOG_PORT)),
    "M" (_BV(COMPARE_OUTPUT_MODE)),
    "x" (_SFR_MEM_ADDR(TIMER_REG)),
    "z" (_SFR_MEM_ADDR(COMPARE_OUTPUT_REG)), "r" (val)
    : "r24"
  );
```

analogRead

The Arduino board contains a 6 channel, 10-bit analog to digital converter which is the brains beneath the analogRead function. It maps input voltages between 0 and 5 into integer values between 0 and 1023, thus yielding a resolution between readings of: 5/1024 units or, 0.0049 volts (4.9 mV) per unit. The input range and resolution can be changed through the *ANALOG_V_REF* define. This code reads the value from the specified analog channel (0-7), which correspond to the analog pins (note, do NOT use A0-A7 for the channel number in this code).

While this version of analogRead (*aRead*) saves a few bytes (~50), it also gives the option of changing the speed via the ADC prescaler. However, don't arbitrarily change the prescale without understanding the consequences. ATMEL advises the slowest prescale should be used (*PS128*). A higher speed (smaller prescale) reduces the accuracy of the AD conversion. The Arduino sets the prescale to 128 during initiation, just as the code below does.

```
//Define various ADC prescales
#define PS2   (1<<ADPS0)                             //8000kHz
ADC clock freq
#define PS4   (1<<ADPS1)                             //4000kHz
#define PS8   ((1<<ADPS0) | (1<<ADPS1))              //2000kHz
#define PS16  (1<<ADPS2)                             //1000kHz
#define PS32  ((1<<ADPS2) | (1<<ADPS0))              //500kHz
#define PS64  ((1<<ADPS2) | (1<<ADPS1))              //250kHz
#define PS128 ((1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0)) //125kHz
#define ANALOG_V_REF     DEFAULT //INTERNAL, EXTERNAL, or DEFAULT
#define ADC_PRESCALE     PS128   //PS16, PS32, PS64 or
P128(default)

uint16_t aRead(uint8_t channel) {
  uint16_t result;

  asm (
    "andi %1, 0x07    \n" //force pin==0 thru 7
    "ori  %1, (%6<<6) \n" //(pin | ADC Vref)
    "sts  %2, %1      \n" //set ADMUX

    "lds  r18, %3             \n" //get ADCSRA
    "andi r18, 0xf8           \n" //clear prescale bits
    "ori  r18, ((1<<%5) | %7) \n" //(new prescale | ADSC)
    "sts  %3, r18             \n" //set ADCSRA

    "_loop:       \n" //loop until ADSC cleared
    "lds  r18, %3 \n"
    "sbrc r18, %5 \n"
    "rjmp _loop   \n"

    "lds  %A0, %4   \n" //result = ADCL
    "lds  %B0, %4+1 \n" //ADCH

    : "=r" (result) : "r" (channel), "M" (_SFR_MEM_ADDR(ADMUX)),
    "M" (_SFR_MEM_ADDR(ADCSRA)), "M" (_SFR_MEM_ADDR(ADCL)),
    "I" (ADSC), "I" (ANALOG_V_REF), "M" (ADC_PRESCALE)
    : "r18"
  );

  return result;
}
```

pinMode(OUTPUT)

The Arduino pinMode function configures pin behavior. The code presented from here on, has been previously explained.

```
asm (
  "sbi %0, %1 \n" //1=OUTPUT

  : : "I" (_SFR_IO_ADDR(DDRB)), "I" (DDB5)
);
```

pinMode (INPUT PULLUP)

```
asm (
  "cbi %0, %2 \n"
  "sbi %1, %2 \n"

  : : "I" (_SFR_IO_ADDR(DDRB)), "I" (_SFR_IO_ADDR(PORTB)),
  "I" (DDB5)
);
```

pinMode (INPUT)

```
asm (
  "cbi %0, %2 \n"
  "cbi %1, %2 \n"

  : : "I" (_SFR_IO_ADDR(DDRB)), "I" (_SFR_IO_ADDR(PORTB)),
  "I" (DDB5)
);
```

pinMode with Multiple Pins

```
#define PIN_DIRECTION 0b00101000 //PIN 3 & 5 OUTPUT
//#define PIN_DIRECTION (1<<DDB3) | (1<<DDB5)
asm (
  "out %0, %1 \n"

  : : "I" (_SFR_IO_ADDR(DDRB)), "r" (PIN_DIRECTION)
);
```

digitalWrite HIGH

If a pin has been configured as an OUTPUT, its voltage will be set to the corresponding value: 5V (or 3.3V on 3.3V boards) for HIGH, 0V (ground) for LOW. However, if the pin is configured as an INPUT, digitalWrite enables (HIGH) or disables (LOW) the internal pullup on the input pin.

```
asm (
  "sbi %0, %1 \n"

  : : "I" (_SFR_IO_ADDR(PORTB)),"I" (PORTB5)
);
```