Programming Arduino in AVR assembly

Programming Exercise: Program your Arduino to be flashing continuously the Morse SOS code

. . . – – – . . .

Ideally, you will have pauses (LED off) of three different durations. The shorter pause will be between the transmission of two Morse symbols (dots, dashes), a longer one separating letters (S, O), and an even longer between words, that is, between repetitions of the SOS message.

If you are not sure about the task, you can watch the short Youtube video:

https://www.youtube.com/watch?v=iiluPgiSFbg

The flashing light can be one or more of the LED lights of the Durham Arduino Extension Board.

Below is a detailed explanation of a simpler example you might want to read before working on the programming exercise.

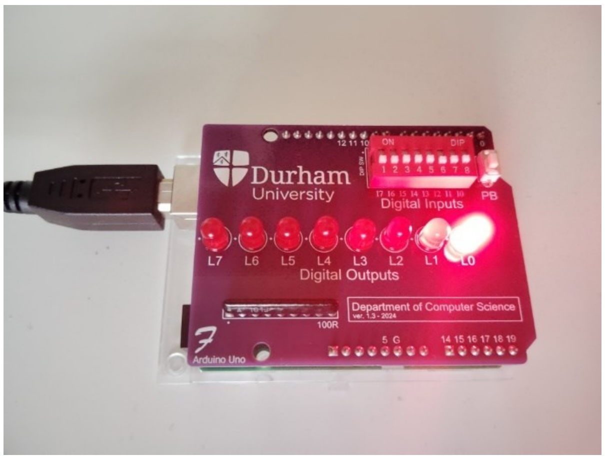
1. Compiling and running assembly code

If you haven't done already, install the Arduino IDE through AppsAnywhere, or download it externally on your laptop. See instructions at Appendix A at end of the document.

The easiest way to start programming in AVR assembly through the Arduino IDE, is to create a C++ shell file, and a separate linked file where the assembly code will be placed.

Create a folder called led and put inside the two files uploaded on Ultra: led.ino (the C++ shell) and led.S (the assembly code).

Put the Durham Arduino Extension Board on top of your Arduino. Compile and load the led.ino file with the Arduino IDE. The L0 LED of the extension board should start blinking.



2. Pin naming

On the extension board, you can read the labels of the eight LEDs, they are L0 - L7. The names were given by the designers of the board, as an easy way to refer to each individual LED. Each LED is controlled by an Arduino pin, which should be set as output pin.

There are also eight manual switches, named I0 - I7, which control input signals. They correspond to eight more Arduino pins, which this time should be set as input pins. In this practical we will not use input signals.

The table below shows the correspondence between the names of the LEDs on the extension board and AVR ports/pins.

AVR port/pin	Arduino pin	LED light name
PC0	D14	L0
PC1	D15	L1
PC2	D16	L2
PC3	D17	L3
PC4	D18	L4
PC5	D19	L5
PB2	D10	L6
PB3	D11	L7

In the first column, at the most basic level of the naming conventions, we have the AVR port/pin names, which have to be used in any programming at AVR assembly level.

In the second column, we have the Arduino's naming conventions (remember Arduino is just one of the many applications of the AVR microcontrollers). These should be used when we write code in the C++ derived high-level programming language developed specifically for Arduino. For example, if in that high-level language you refer to pin D17, the Arduino compiler that converts you program to AVR assembly, will translate D17 to PC3.

Finally, in the third column, we have the naming conventions used by the developers of the Arduino Extension Board, built on top of the Arduino pin naming, built on top of AVR pin naming.

For diagrams of the complete pin name mapping (not needed here) between Arduino Extension board, Arduino C++ language, and AVR assembly, see Appendix B at the end of the document.

2. The C++ code

The C++ code in led.ino defines a setup() function and a loop() function. The loop() repeats for ever, which usually is the desirable behaviour in an embedded system.

The setup() just calls the start() function, which is programmed in the led.S file.

Inside the loop(), the led() function, which is again programmed in led.S, is called twice, with two different arguments. The first time, called with argument 1, it will switch the LED light on, and the second time, called with argument 0, it will switch the LED light off.

3. The assembly code

The start function

```
SBI   DDRC, 0        ;set PC0 (D14) as o/p
RET                  ;return to setup() function
```

will set PC0 pin (digital pin 14) as the output pin and will return to led.ino.

You can check the SBI instruction in the AVR instruction set manual (p.151), but here will not need a more detailed documentation of the instruction.

3.1 Conditional branching

Next, the led function starts with the instructions

```
CPI     R24, 0x00        ;value in R24 passed by caller compared with zero
BREQ  ledOFF             ;jump (branch) if equal to subroutine ledOFF
```

By default, the argument of the function led was passed to register 24. This piece of code will compare the contents of the register 24 (the argument of the function) with zero, and if the argument was zero it will branch to ledOFF.

Note:

Notice that the instruction BREQ specifies where to branch to, but it does not encode the comparison determining whether we branch or not. This is different to the MIPS instruction beq, which encodes the two registers the values in which we compare and the address we branch to.

As AVR has smaller words, it works differently. We should rather see the two instructions above as a pair, doing what the single MIPS instruction beq does.

First, the CPI instruction (Compare With Immediate) compares the contents of a register Rd with an immediate K, that is, examines the difference Rd-K, without changing the contents of register Rd. Then, depending on the value of Rd-K, it sets accordingly some bits of the Status Register.

The Status Register is an 8-bit register, part of the CPU, but not one of the 32 general purpose registers we have discussed. Its 8-bits are used as flags.

I	T	H	S	V	N	Z	C
–	–	–	–	–	–	–	–

In particular, if Rd - K is zero, here, if Rd - 0 = 0, the Z flag (the second from the right) will be set.

Finally, the BREQ will just look at flag Z, and it will branch to the specified address if and only if that flag has been set.

3.2 The rest of the code

If the BREQ does not branch, the led function continues to the next line and switches the light on, then calls the myDelay function, and finally returns to led.ino.

```
    SBI       PORTC, 0           ;set D14 to high
    RCALL  myDelay
    RET                          ;return to loop() function
```

Otherwise, the ledOFF, switches the light off, delays, and returns to led.ino.

```
ledOFF:
    CBI        PORTC, 0           ;set D14 to low
    RCALL   myDelay
    RET                           ;return to loop() function
```

3.3 The myDelay function

The myDelay function, based on a previously declared global variable delayVal, makes a computation (counting down) just to delay the return to the led.ino, thus controlling the speed of the blinking.

It uses the LDI (Load Immediate) instruction to assign values to the registers, while decrementing and conditional branching uses pairs of instructions SBIW (Subtract Immediate from Word) or SUBI (Subtract Immediate), and BRNE (Branch if Not Equal). Notice that SBIW and SUBI do not only perform arithmetic operations, but also change the contents of the Status Register.

```
myDelay:
   LDI   R20, 100             ;initial count value for outer loop

outerLoop:
   LDI   R30, lo8(delayVal)   ;low byte of delayVal in R30
   LDI   R31, hi8(delayVal)   ;high byte of delayVal in R31

innerLoop:
   SBIW    R30, 1             ;subtract 1 from 16-bit value in R31, R30
   BRNE   innerLoop           ;jump if countVal not equal to 0
   SUBI    R20, 1             ;subtract 1 from R20
   BRNE  outerLoop            ;jump if R20 not equal to 0
   RET
```

4. AVR instructions used in led.S

If you want you can check in the AVR manual uploaded on Ultra (the following instructions:

LDI Load Immediate (p.115)

SBIW Subtract Immediate from Word (p.152)

It subtracts an immediate from the word (16 bits) formed by a pair of registers. Here the registers are the 30 and 31. Notice that only four pairs of registers can be used with this instruction, thus the operand specified the pair of registers is just 2 bits (dd), while the immediate is 6 bits (KKKKKK), and the opcode 8 bits.

SBIW sets the Z flag if the result of the operation is zero.

1001	0111	KKdd	KKKK

BRNE BRanch if Not Equal (p.54), branches if the Z flag is clear (its value is 0).

SUBI Subtract Immediate (p.183)

Subtracts an immediate from a register. Sets the Z flag if the result of the subtraction is zero.

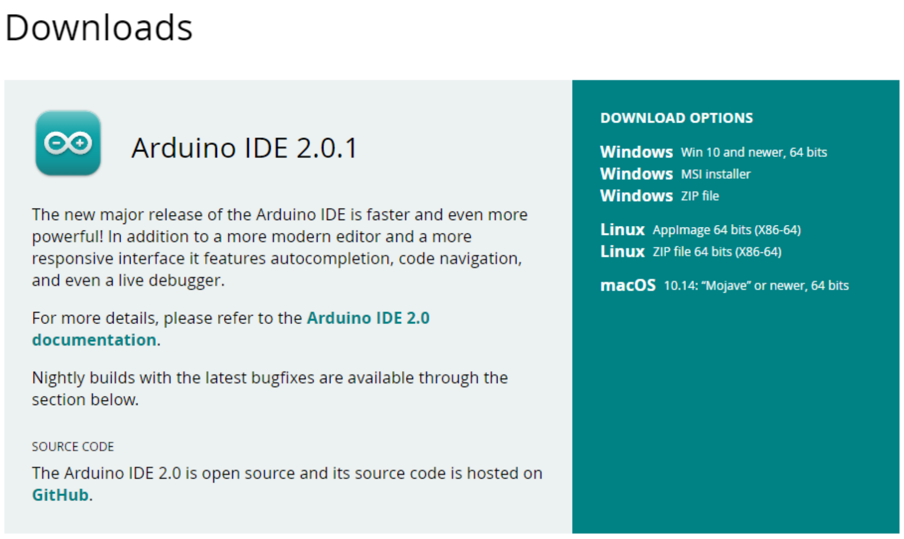
The example is a modified version of the publicly available code

https://akuzechie.blogspot.com/2021/09/assembly-programming-via-arduino-uno.html

where you can find many more examples.

Appendix A. Set up the Arduino IDE

Install the Arduino IDE through AppsAnywhere, or download it externally on your laptop:



Downloads

Arduino IDE 2.0.1

The new major release of the Arduino IDE is faster and even more powerful! In addition to a more modern editor and a more responsive interface it features autocompletion, code navigation, and even a live debugger.

For more details, please refer to the **Arduino IDE 2.0 documentation**.

Nightly builds with the latest bugfixes are available through the section below.

SOURCE CODE

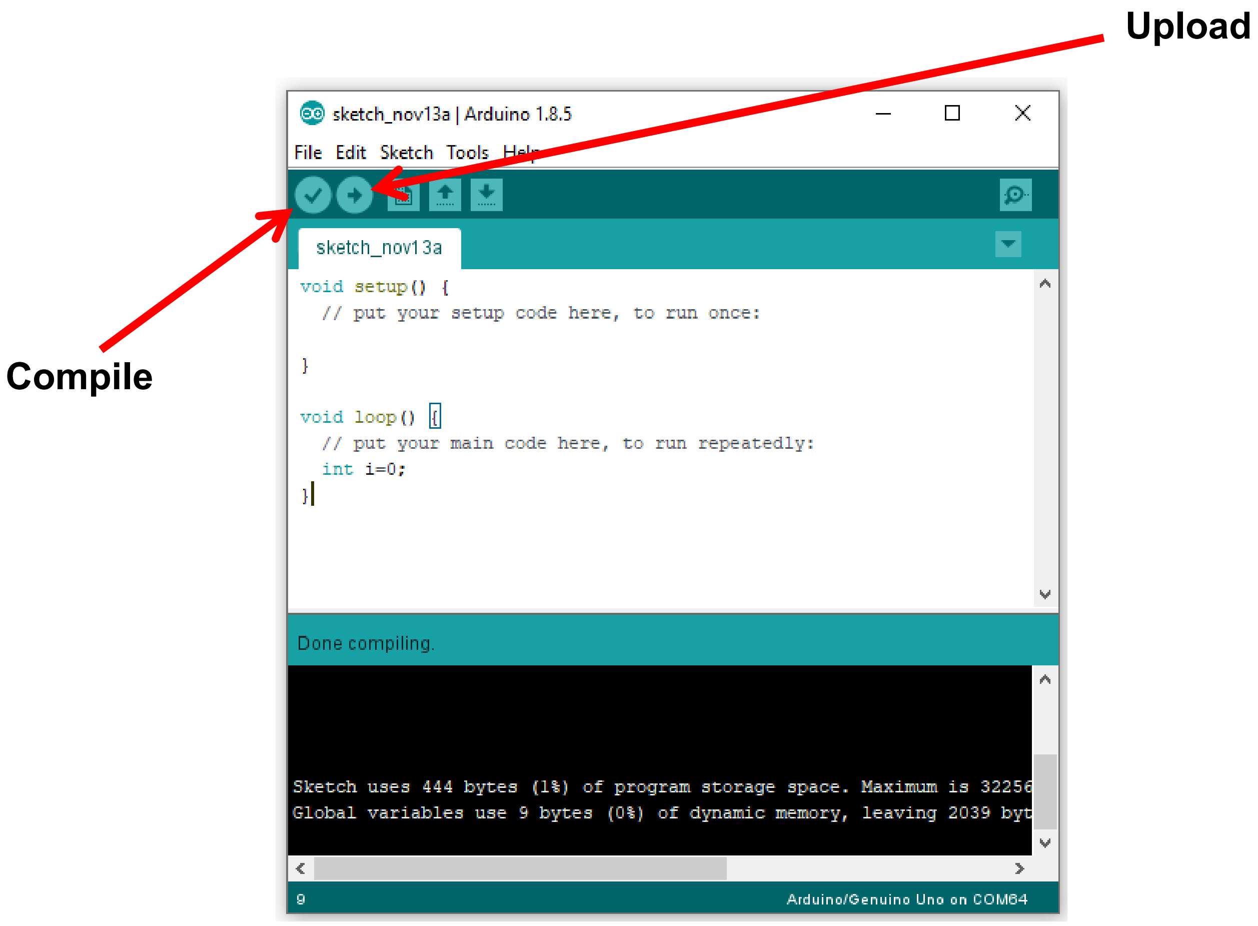
The Arduino IDE 2.0 is open source and its source code is hosted on **GitHub**.

DOWNLOAD OPTIONS

Windows Win 10 and newer, 64 bits
Windows MSI installer
Windows ZIP file
Linux AppImage 64 bits (X86-64)
Linux ZIP file 64 bits (X86-64)
macOS 10.14: "Mojave" or newer, 64 bits

If you double-click on an Arduino file (extension .ino) it will automatically open in Arduino IDE. The IDE might ask you to place your .ino file inside a folder with the same name.

You are ready to compile and load.



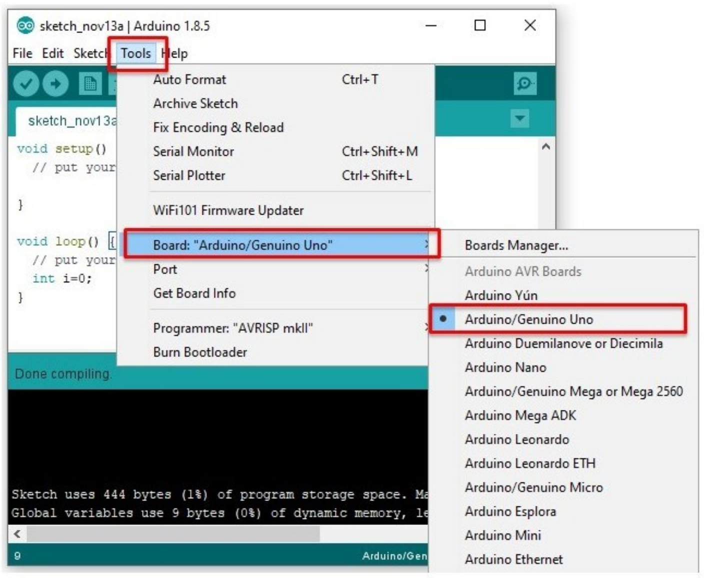
```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:
  int i=0;
}
```

Done compiling.

```
Sketch uses 444 bytes (1%) of program storage space. Maximum is 32256
Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 byt
```

9 Arduino/Genuino Uno on COM64

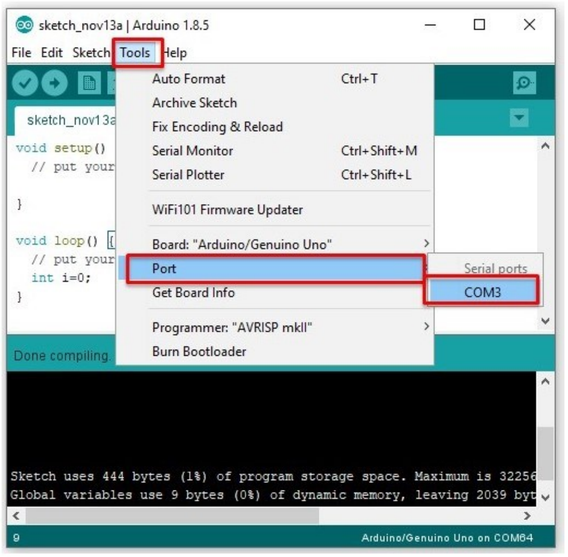
Before uploading, set the Board and the COM through the menu:



```
void setup()
  // put your

}

void loop() {
  // put your
  int i=0;
}
```

Done compiling.

```
Sketch uses 444 bytes (1%) of program storage space. Ma
Global variables use 9 bytes (0%) of dynamic memory, le
```


sketch_nov13a | Arduino 1.8.5
File
Edit
Sketch
Tools
Help
sketch_nov13a
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L
WiFi101 Firmware Updater
Board: "Arduino/Genuino Uno"
Port
Get Board Info
Programmer: "AVRISP mkII"
Burn Bootloader
Serial ports
COM3
9
Arduino/Genuino Uno on COM64

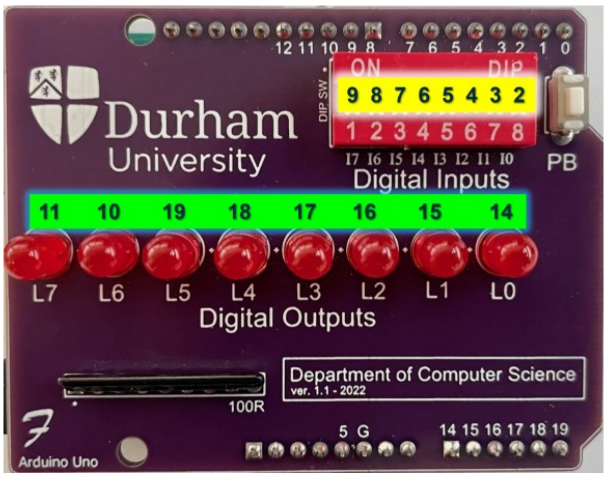
```
void setup()
  // put your

}

void loop() {
  // put your
  int i=0;
}
```

Done compiling.

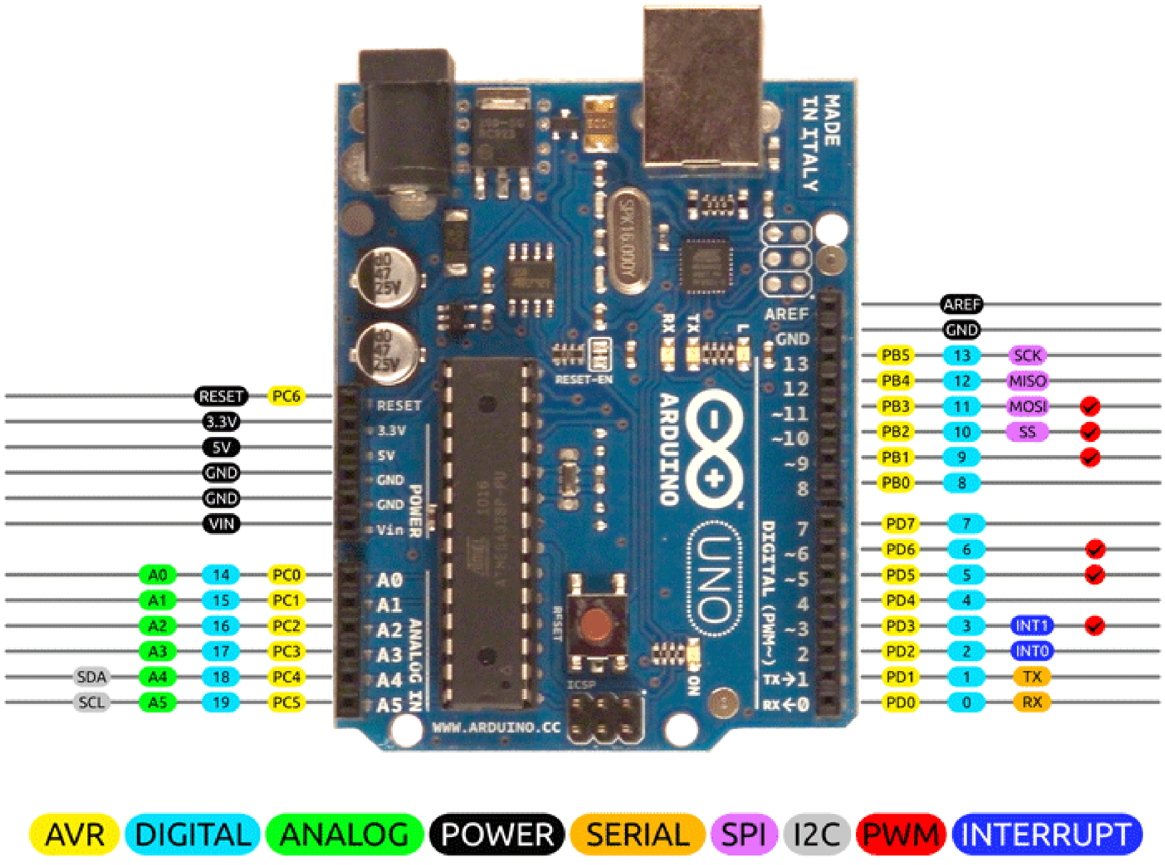
```
Sketch uses 444 bytes (1%) of program storage space. Maximum is 32256
Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 byt
```

Appendix B. Pin name mapping

The correspondence between Extension Board input and output pins and Arduino C++ pin names:



The correspondence between Arduino C++ pin names and AVR assembly ports/pins:


RESET PC6
3.3V
5V
GND
GND
VIN
A0 14 PC0
A1 15 PC1
A2 16 PC2
A3 17 PC3
SDA A4 18 PC4
SCL A5 19 PC5
AREF
GND
PB5 13 SCK
PB4 12 MISO
PB3 11 MOSI ✓
PB2 10 SS ✓
PB1 9 ✓
PB0 8
PD7 7
PD6 6 ✓
PD5 5 ✓
PD4 4
PD3 3 INT1 ✓
PD2 2 INT0
PD1 1 TX
PD0 0 RX
AVR
DIGITAL
ANALOG
POWER
SERIAL
SPI
I2C
PWM
INTERRUPT