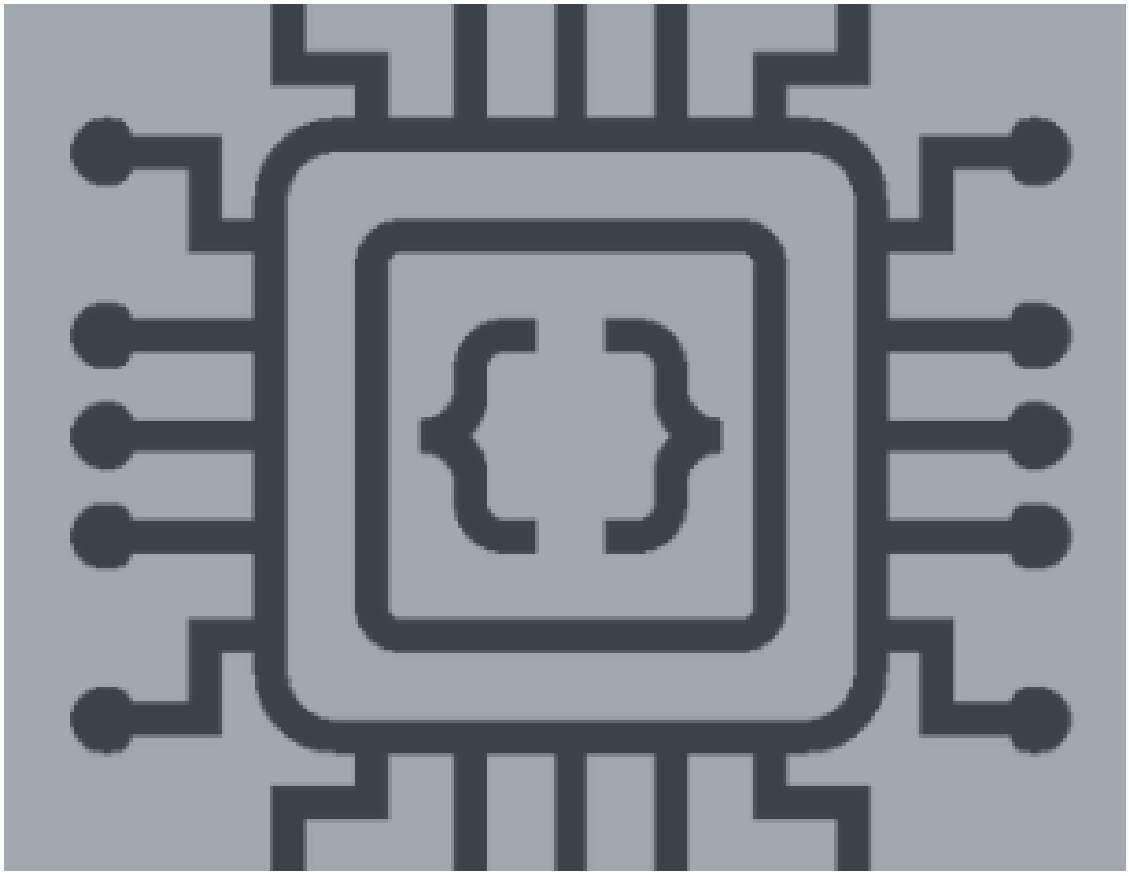
Embedded Systems



Assignment: 3
Date: 11/04/2023
Group: Johnson Domacasse, Vincent van Laarhoven
Version: V1

Contents

Abstract

We may run into problems with memory usage, compile and run time when we are writing bigger programs for a microcontroller. Sometimes we run out of bytes that can be used, the program isn't running and efficiently as it should or it can even be slow. We can solve this problem by using bit manipulation and registers. This way we eliminate unnecessary bytes that we don't make use of.

Introduction

Programming a microcontroller is almost essential if you want to dive deeper in to the world of embedded systems. In this case, we will be using an Arduino Uno board as our microcontroller. We give the Arduino tasks such as setting up the I/O, reading or writing a value to a pin. Naturally once we are done writing our software, we can build then upload this to our Arduino. If all goes well you will have a code than compiled correctly and within the memory available.

What if we had a bigger program than required us to use more pins. Or another program that required us to read and write frequently to these pins. You may notice then that you may even have a scarcity of memory. Or you may have a long compile and run time. These are but some of the issues that can be solved using *bit manipulation* and *registers*.

Instead of using for example the read or write functions in the program, We make use of the registers and bit manipulation. This results in us only using the bytes that we need for reading or writing to the Arduino. Which means that in the end, we would use less bytes than if we were to use the functions of the Arduino.

Research questions

Main question:

How can we make efficient use bit manipulations to make I/O efficient fast and accurate?

sub-question:

How can we do this without making use of the Arduino API functions(pinMode(), digitalRead(), digitalWrite())?

Procedures

Hardware setup:

We set up the LEDs to the Arduino By connecting each one of them to a resistor with a value of 470Ω to avoid damaging them. 4 of these LEDs, of the same colour, will be used to for sweeps and a singular LED will be used to blink at a set frequency.

We then worked on set up the 2 buttons. One of them would use the internal PULL-UP of the Arduino. Meaning a resistor would not be needed. The other would need the 10kΩ resistor because it uses a the PULL-DOWN of the button.

These are all connected to separate ports on the Arduino using jumper wires. See figure 1.

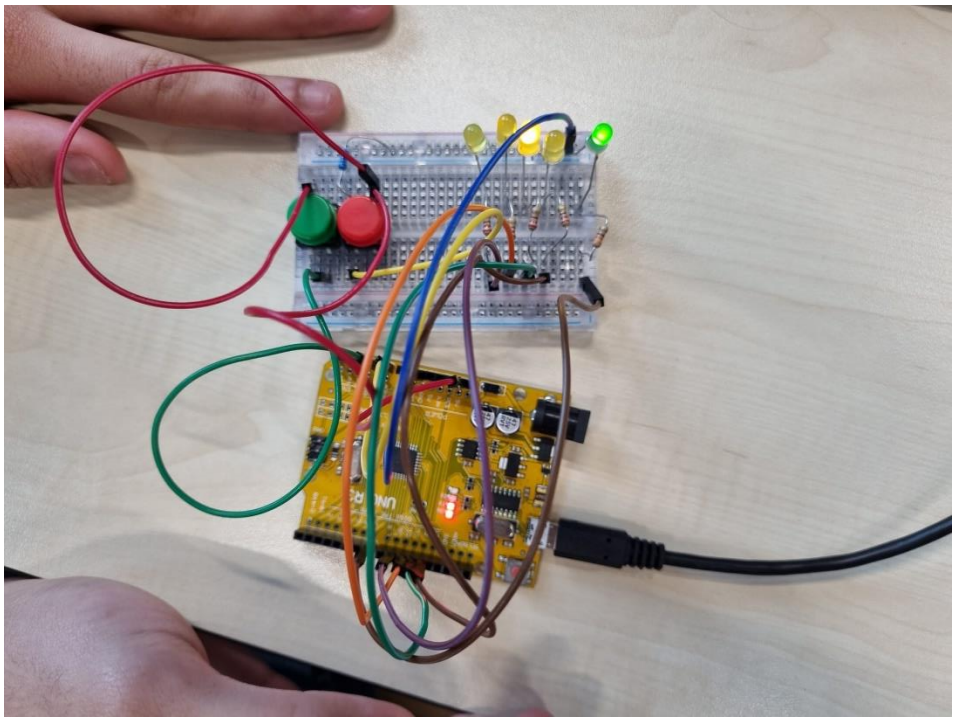


Figure 1, *complete hardware system.*

Software setup:

We first define the pins and masks we will be making use of for both the LEDs and the buttons. We then define a few timer intervals to make use of the "*Millis"* function in order to avoid use delays. Within the program we make use of different functions. **See *Table 1.***

Table 1, *complete list of functions.*

Function name	***Return type***	***description***
Setup()	Void	Set up I/O pins
ExternalLedBlink()	Void	Blinks external green LED on the breadboard.
BtnLedOnOrOff (bool btnState)	Void	Turns the builtin LED on the Arduino on or off, depending on button state.
ButtonDebouncePartA()	Bool	Simple debounce function for button used in part A.
SweepA (int statesSweepA)	Void	Function that goes through all the LEDs with the effect of Sweep A.
LedReset()	Void	Function that turns off all LEDs.
SweepB()	Void	
ButtonDebouncePartB()	Bool	Simple debounce function for button used in part B.
Loop()	Void	This function it holds all of the above functions.

Within two of those functions we make use of a sequence state patterns to increase the program's speed and efficiency. One for first sweep and one for the second sweep.

Blinking LED:

The blinking LED is implemented by using the void function to continuously turn the external LED on and off by changing its state using the Millis function. Next to this, we also implement a button that makes use of the PULL-UP feature in the Arduino. When this button is pressed, the built-in LED will stay on and when released it will stay off.

Sweep A:

The way sweep A works is that you have a sequence state pattern with 6 states. Each state would turn an LED off and another LED on. When the program reaches the final state, it will then go to the first state and start this process all over again. Here is an example line of code of one of the states that turns on the first LED and turns the second LED off:

```
PORTB = (PORTB | PIN1_MASK) & ~PIN2_MASK;
```

It turns on the first LED by applying LED1's mask with an OR-operator. Regardless of the value it will turn of PORTB it will turn on then. It also turns off LED2 by applying its mask with an AND-operator. However in this case it would always turn off since you are applying NOT-operator to the mask.

The rest of the states look almost exactly like the one above example and they are modified to their respective LEDs.

When you take a look at the void function, you will notice that that's where the states are changed by incrementing the state. When the state reaches the value of 6, it will start back from the start. This is to avoid any additional states.

Sweep B:

The way sweep B works is almost similar to sweep A where you have states and each state has a specific functionality. The difference between sweep A and B is that in sweep A, we turn the LEDs on and off directly using the registers assigned to them. For sweep B we make use of a 2 masks. One that is a constant and is used to control which LED gets turned on. The other is shifted around as we change states. These 2 masks are then compared to one another to create a new 16 BIT value. This value is what we use to turn the LEDs on and off.

Within sweep B we have 2 different types of shifts. One that shifts the state mask around and another that determines whether we should be sweeping left or sweeping right. The second shift is dependant on the first one. When the first shift reaches a certain threshold (in our case, after 6 shifts), it will change directions. Meaning that after 6 shifts to the left, the direction of the sweep will change and now go 6 shifts to the right. It repeats this process until the button has been pressed.

Notable anomaly:

When we finished developing sweep B, it was on the code the button that would change modes between sweep A and B. Reading and using the values of the button were quite simple. Our problem arose when we actually pushed the button and the change between the modes had an unusual behaviour. It functioned perfectly. It began to perform sweep A, when we pushed to button to go from sweep B to A. However, when we released the button to go back to sweep B, it would not start from the beginning like sweep A but at where sweep A left off. This would cause that the LED of which you stopped at to be permanently on. An LED next to the permanently on one would also not function in this circumstance.

What we did to solve this problem is by adding a new variable that checks if the state of sweep B. We would use this variable to always start from the beginning with sweep B. The second part of this solution was to add a new function that would turn off all of the LEDs before proceeding with the sweep. So all of the LEDs work flawlessly, we don't have the issue where the LED is always on and we always start from the beginning.

Research Questions answers

Main question:

In order to efficiently use the I/O pins on the Arduino We make use of the hardware registers of the Arduino in our program.

- Data direction register (DDRx). Used for setting direction of pin.
- Port output register (PORTx). Used to write to pin.

- Port input register (PINx). Used to read from pin.

Registers are storage of values used by computational tasks. They are also a set of switches that are part of an internal circuit.

We make use of sequence state patterns in order to boost the speed and efficiency. Next to these, we also make use the least amount of if-statements as possible in order to further the program's efficiency.

Sub-question:

We can do by applying bitmasks to these registers by bit manipulation. In other words, we can apply masks to these registers using bit operators.

- AND-operator (&).
- OR-operator (|).
- NOT-operator (~).

Bit manipulation is the practice of using operations on bits that make I/O programming efficient, fast, small and more accurate.

With this we eliminate the unnecessary of instructions or native code i.e. no waste.

We also make efficient use of the computers architecture.

See Sweep A section for an example code on how we use bit manipulation write to a pin to turn an LED on and another LED off.

Conclusion

In conclusion, you will notice a significant decrease in bytes used when the code has been run. You will also notice that the compile time is also less than what it would be if you would do it with Arduino's API functions. This is a rather difficult to grasp, but very efficient, method in terms of memory saved and compile time.

For now all we needed to develop was a small program that would blink and sweep LEDs. We can make use of this method later on when we are developing bigger programs that require more bytes to be used. This way we can make them more memory efficient.