PH435 Lab 1

Sankalp Gambhir
180260032

September 6, 2020

A. Hardware

A.1) 16 MHz. The frequency of the clock was found to be 16 MHz from the datasheet.

A.2) 16/20 MHz. Data differs between different datasheets I checked.

A.3) Power. Higher voltages are required to run at higher frequncies c.f. Fig 2a, presumably due to loads causing reduction in voltage and it becoming harder to distinguish between low and high. At the same time, the current required increases as well (Fig. 2b). Thus, the net increase in power will limit the frequency at which we can drive the chip due to thermal as well as electric constraints. Thermal should set in first. If you were able to provide external cooling say witha fan, electrical limitations would take over soon.

A.4) Setup done.

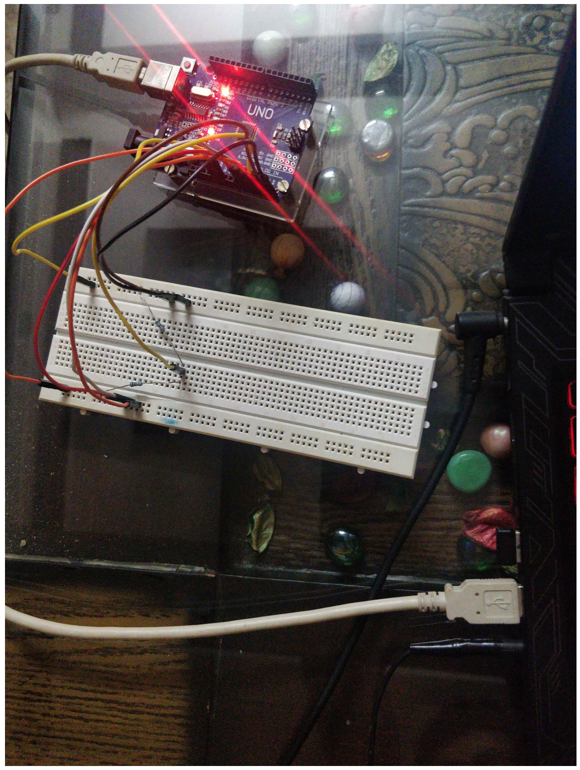


Figure 1: Arduino connected to laptop

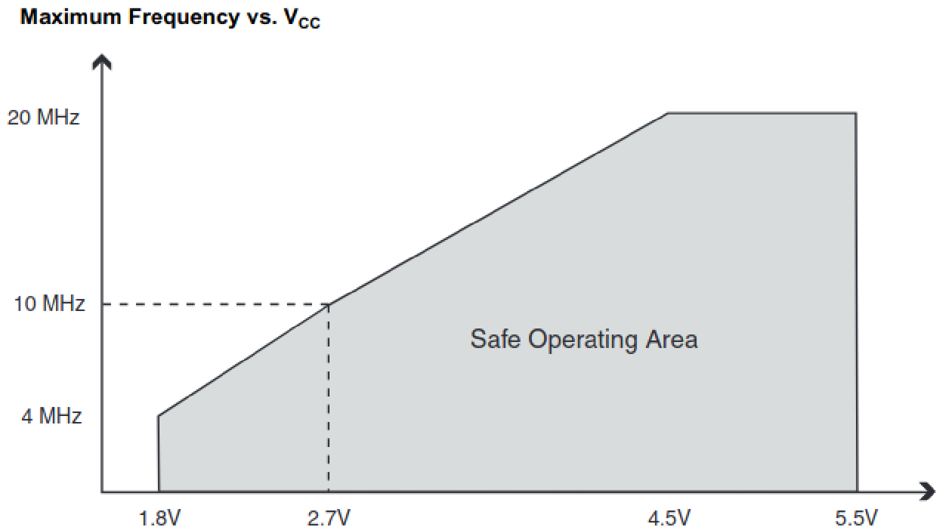
B. Digital I/O

B.1) With access to a DSO, the time taken to execute a digital write may be obtained using a program like this:

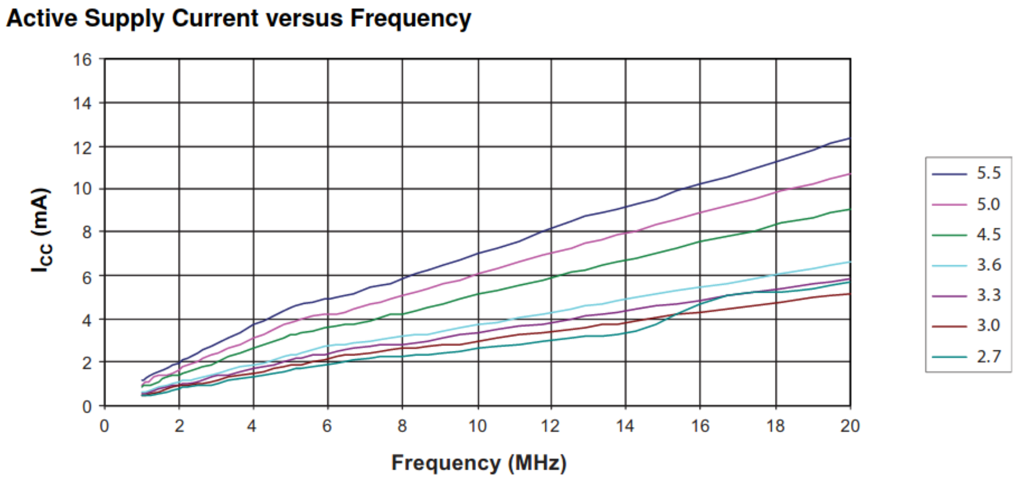
```
void setup(){
    pinMode(LED_BUILTIN, OUTPUT);
}

void loop(){
    digitalWrite(LED_BUILTIN, LOW);
    digitalWrite(LED_BUILTIN, HIGH);
}
```

The DSO probe would be connected to pin 13, corresponding to the LED, and ground. The time taken for the instruction will be half of the time period of the obtained (presumably square) wave.

B.2) We put the tested function in the `loop()` section (see Appendix) and then monitor the output using a C++ program with the serial output pipelined into it. The C++ program was designed to count



(a) $V - f$



(b) $f - I$

Figure 2: Graphs for electric constraints with frequency
Taken from https://www.microchip.com/wwwproducts/en/ATmega328p

instances of an arbitrary provided string and time it with a high resolution clock.

The following results were obtained by timing the given program:

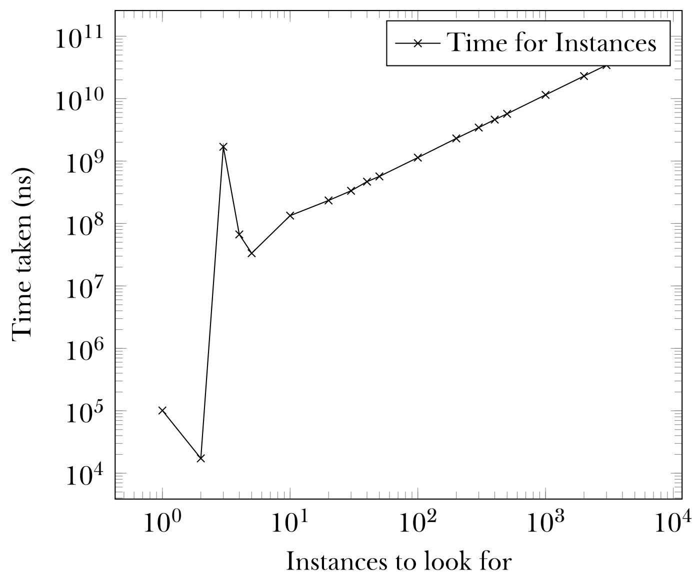


Figure 3: Time taken to receive instances of `"01Game Over"`

Looking at the figure, the first few points are quite erratic, possibly due many small erratic foactors interacting, such as transmission delay, serial poling delay, etc. The data very quickly settles into a linear plot, which was used to determine a best fit line of the form

$$y = mx + c$$

with obtained values,

$$m = 1.148 \times 10^{7}\text{ns}$$
$$c = 3.907 \times 10^{5}\text{ns}$$

Thus, the average time per function run was 1.148×10^{7} ns, i.e. **11.48 ms**.

It is also worth noting the obtained value for c. There seems to be a sizable constant delay for each instance run. This may be attributed to the overhead of the timing methodology, the buffer as well as the timing program itself. It being a couple orders of magnitude smaller than the actual slope ensures that our testing methodology was appropriate and did not skew the results too much.

Note — data for less than 10 instances was not included for the best fit.

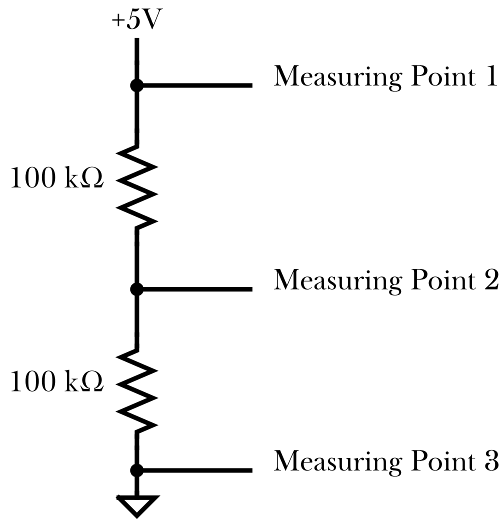
Actual data values and testing details may be found in the Appendix.

C. Analog Input

C.1) Quite mixed with the next part, so combined answer in **C.2)**

C.2) The analog ports A0-A5 were connected to a voltage divider between 5V and GND taken from the Uno itself as the circuit in Fig 4a and the program as in 4b. The actual setup was as already included in Fig ??.

The wires were moved between the three measuring points during the course of the execution to collect data.



(a) Voltage divider circuit

```
void setup(){
    Serial.begin(9600);
}

void loop(){
    Serial.print("\nValues : ");
    for(int port = A0; port <= A5; port++){
        Serial.print(analogRead(port));
        Serial.print(’ ’);
    }
    delay(3000);
}
```

(b) Code used for measuring analog inputs

Figure 4: Setup for the analog input measuring

The output received was as in the following figure, ordered as A0 to A5.

```
▲ sankalp ~ tail -f /dev/ttyUSB0

Values : 1023 1023 1023 511 0 0
Values : 1023 1023 1023 512 0 0
Values : 1023 1023 1023 769 0 0
Values : 1023 1023 1023 1023 0 72
Values : 1023 1023 1023 1023 512 512
Values : 1023 1023 1022 1023 512 512
Values : 1023 1023 1023 1023 512 512
```

It was interesting to see the values in the middle as I managed to read them while moving wires around. Not sure if this is dangerous (hopefully not!) but it was nice to see the input values changing, presumably due to a buffering circuit to smooth over the input in some way?

Appendix

Arduino

The code uploaded to the Arduino for testing. The function ‘test()’ is to make the testing more modular by changing the timed function once, and added multiple times as required without overhead, since the ‘inline’ keyword ensures resolution of calls at compile-time. The functionality was, however not utilized here in the end.

```
/*
Lab 1
Arduino Basics
*/

// variables

// functions
inline void test();

void setup(){
    Serial.begin(9600);
    pinMode(LED_BUILTIN, OUTPUT);
}

inline void test(){
    digitalWrite(LED_BUILTIN, LOW);
    Serial.print('0');
    digitalWrite(LED_BUILTIN, HIGH);
    Serial.print('1');
    Serial.print("Game Over");
}
```

C++ Timer

The code used to time the output at a lower level than via the IDE. The IDE seems to be made fairly efficiently as the results were close. The output is buffered using the UNIX `tail` utility, with the `-f` indicating `follow`, checking the stream at intervals provided by `-s`, or `sleep`, used as 1μs here for reasonable accuracy. The output buffer generated by `tail` is piped into the C++ program as `stdin` to be read the same as any user input. The tests were performed with (an unnecessary) nanosecond accuracy using a high resolution clock provided by `std::chrono`.

```
#include <iostream>
#include <chrono>
#include <vector>

std::vector<int> checks{1}; // add an empty check to account for initial setup delay
const char * test_string = "01Game Over"; // arbitrary string to look for

int main(){

    char c = (char) 0;
    int pos_test = 0; // position in test string during traversal
    int count = 0; // number of test strings seen

    // construct the checks vector as
    // 1, 2, 3, 4, 5,
    // 10, 20, 30, 40, 50, ....
```

```cpp
    // 10000, 20000, 30000, 40000, 50000
    int i = 1;
    for(int j = 1; j < 6; j++){
        checks.emplace_back(i*j);

        if(j == 5 && i < 10000){
            j = 0;
            i *= 10;
        }
    }


    auto curr_time = std::chrono::high_resolution_clock::now();
    auto last_time = curr_time;

    while(!checks.empty()){
        if(count == checks[0]){
            // log a hit
            curr_time = std::chrono::high_resolution_clock::now();
            std::chrono::duration<double, std::nano> time_taken = curr_time - last_time;
            last_time = curr_time;
            std::cout   << "Count " << count
                        << " : Time " << time_taken.count()
                        << " ns" << std::endl;

            // remove this check
            checks.erase(checks.begin());
            // reset
            count = 0;
        }


        // have we found the entire string?
        if(test_string[pos_test] == (char) 0){
            count++;
            pos_test = 0;
            continue;
        }

        // get the next character
        c = std::cin.get();

        if(c == test_string[pos_test]){
            pos_test++;
        }

    }

    return 0;
}
```

Results

The command used to obtain the output as well as the explicit output. The first "1 count" may be ignored, as its time is due to the `curr_time` being set outside of the loop. The library `std::chrono` has been previously measured to be fast enough to only cause delays of the order of 10ns, so it is not a rate limiter here. The tests were stopped at 5k for this case, since they would take a bit long for more, and don't add much information.

```
▲ sankalp .../lab1 tail -f /dev/ttyUSB0 -s 0.000001 | ./arduino-timer.out
    Count 1 : Time 1.05739e+08 ns
    Count 1 : Time 101053 ns
    Count 2 : Time 17291 ns
    Count 3 : Time 1.6937e+09 ns
    Count 4 : Time 6.67496e+07 ns
    Count 5 : Time 3.34086e+07 ns
    Count 10 : Time 1.33625e+08 ns
    Count 20 : Time 2.33841e+08 ns
    Count 30 : Time 3.34033e+08 ns
    Count 40 : Time 4.67719e+08 ns
    Count 50 : Time 5.67789e+08 ns
    Count 100 : Time 1.13591e+09 ns
    Count 200 : Time 2.30494e+09 ns
    Count 300 : Time 3.44066e+09 ns
    Count 400 : Time 4.6101e+09 ns
    Count 500 : Time 5.71234e+09 ns
    Count 1000 : Time 1.14914e+10 ns
    Count 2000 : Time 2.29831e+10 ns
    Count 3000 : Time 3.44411e+10 ns
    Count 4000 : Time 4.59325e+10 ns
    Count 5000 : Time 5.74241e+10 ns
```