# MAXIM DS1086 OSCILLATOR CONFIGURATION

This document describes the procedure to quickly program the Maxim DS1086 oscillator in order to act as an external clock for a MCU/MPU, without any prior knowledge on the part.

The procedure presented hereafter is based on a practical example: to get a deeper insight on how the DS1086 works, please refer to the part's datasheet available at the following link:

https://datasheets.maximintegrated.com/en/ds/DS1086-DS1086Z.pdf

The internal master oscillator of the DS1086 generates a square wave with a frequency between 66 and 133 MHz. In order to get lower-frequency output square waves, two values have to be tuned:

- Prescaler: a power of the form $2^x$ acting as divider for the master oscillator frequency.
- Offset: a value to fine-tune the output frequency.

Registers of the DS1086:

| REGISTER | ADDR | MSB | | | BINARY | | | | LSB | FACTORY DEFAULT | ACCESS |
|----------|------|-----|-----|-----|-----|-----|-----|-----|-----|-----------------|--------|
| PRESCALER | 02h | $X_1$ | $X_1$ | $X_X$ | J0 | P3 | P2 | P1 | P0 | 11100000b | R/W |
| DAC HIGH | 08h | b9 | b8 | b7 | b6 | b5 | b4 | b3 | b2 | 01111101b | R/W |
| DAC LOW | 09h | b1 | b0 | $X_0$ | $X_0$ | $X_0$ | $X_0$ | $X_0$ | $X_0$ | 00000000b | R/W |
| OFFSET | 0Eh | $X_1$ | $X_1$ | $X_1$ | b4 | b3 | b2 | b1 | b0 | 111-----b | R/W |
| ADDR | 0Dh | $X_1$ | $X_1$ | $X_1$ | $X_1$ | WC | A2 | A1 | A0 | 11110000b | R/W |
| RANGE | 37h | $X_X$ | $X_X$ | $X_X$ | b4 | b3 | b2 | b1 | b0 | xxx-----b | R |
| WRITE EE | 3Fh | | | | NO DATA | | | | | — | — |

The formula that governs the chip's output clock signal is the following:

$$Desired\ Output\ Frequency\ f_{out} = \frac{f_{MASTER\ OSCILLATOR}}{PRESCALER}$$

Consider the following example: the desired output frequency is 15 MHz. Therefore, using the above equation, we obtain:

$$15\ MHz = \frac{f_{MASTER\ OSCILLATOR}}{PRESCALER}$$

We know that the pre-scaler is a power of two, so by trial-and-error we search a power of two such that, solving for $f_{MASTER\ OSCILLATOR}$, we get a master oscillator frequency in the 66-133 MHz range, as required from the device's specification. For example, consider $x = 3$.

$$f_{MASTER\ OSCILLATOR} = 15\ MHz * 2^3 = 15 * 8 = 120$$

This value of $x$ produces a $f_{MASTER\ OSCILLATOR}$ in the acceptable range: let's fix the pre-scaler's power to 3. To do this, set the 4 least significant bits of the *PRESCALER* register to 0x03.

The next step is to determine in which *OFFSET* range our $f_{MASTER\ OSCILLATOR}$ is falling. Consider the following table from the datasheet:

| OFFSET | FREQUENCY RANGE (MHz) |
|---|---|
| OS - 6 | 61.44 to 71.67 |
| OS - 5 | 66.56 to 76.79 |
| OS - 4 | 71.68 to 81.91 |
| OS - 3 | 76.80 to 87.03 |
| OS - 2 | 81.92 to 92.15 |
| OS - 1 | 87.04 to 97.27 |
| OS* | 92.16 to 102.39 |
| OS + 1 | 97.28 to 107.51 |
| OS + 2 | 102.40 to 112.63 |
| OS + 3 | 107.52 to 117.75 |
| OS + 4 | 112.64 to 122.87 |
| OS + 5 | 117.76 to 127.99 |
| OS + 6 | 122.88 to 133.11 |

*Factory default setting. OS is the integer value of the 5 LSBs of the RANGE register.

The best fitting range – i.e. the one which has our 120 MHz master frequency about in the middle – is OS + 5. We could choose OS + 4, but the master frequency value would not be nicely centered in that range.

To store the OS + 5 value in the *OFFSET* register, we first need to read the *OFFSET DEFAULT* value from the five least significant bits of the *RANGE* register. This register contains factory defaults. Increment the *OFFSET DEFAULT* value by 5, and store it to the *OFFSET* register:

$$OS + 5 = OFFSET\ DEFAULT + 5$$

$$OFFSET \leftarrow OS + 5$$

Compute the *DAC* value by subtracting the range's lower bound from the master frequency and dividing the result by the 10 kHz step size. Since we are working in MHz, the 10 kHz step size is 0.01.

$$DAC = \frac{f_{MASTER\ OSCILLATOR} - 117.76}{0.01} = 224 = 0x00E0$$

Consider that the DAC value is composed of two bytes, stored left-aligned in the two registers DAC HIGH and DAC LOW, respectively. Therefore, trailing zeroes must be shifted out.

$$DAC = 0x00E0 = 0000\ 0000\ 1110\ 0000$$

There are 8 trailing zeroes, so we need to shift them out by performing a left bit-shift of 8 positions:

$$DAC \ll 8 = 1110\ 0000 = 0xE000$$

**Programming the DS1086 with Arduino**

The DS1086 registers can be read and written through a standard I2C interface. Using an Arduino as I2C master device this can be done very quickly and easily. First, consider the chip's pinout from the datasheet depicted below. The following figure also shows the required connections.
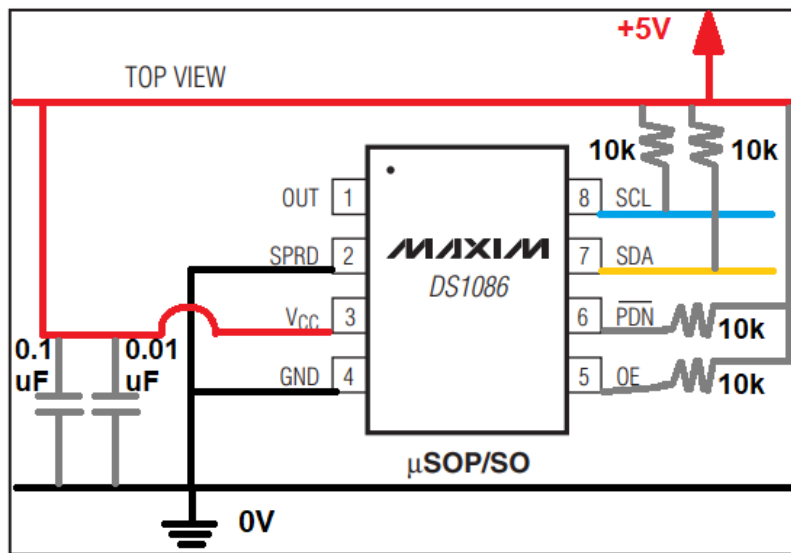


Since the DS1086 chip is a 8-USOP SMD component, if programming is going to be performed off-a-PCB, it should be soldered on a USOP-8 breakout. However, for testing purposes only, it is also possible to carefully solder pieces of wire directly to the pins to fit it into a breadboard. Be careful not to overheat the chip with a soldering iron: 280°C should be a good soldering temperature. In the latter case, you should end up with something like the picture on the right. Do not forget to connect the Arduino's ground with the breadboard's one: they must have common ground in order to work.
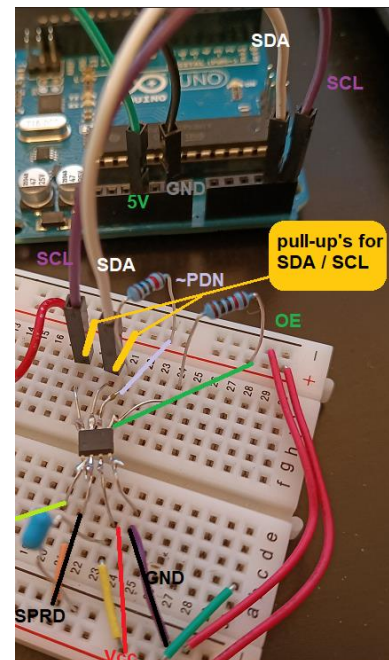


Before presenting the code to program the chip as shown in the example above, some considerations about the specific I2C implementation of the Arduino's library should be made. Technical details about the I2C protocol will be omitted in this document, but are

available in the Texas Instrument's documentation provided in the repository of this project: https://github.com/hvictor/DS1086/blob/main/TI_I2C_Protocol.pdf

Consider the following comparison between the I2C protocol definition and the data communication example provided in the DS1086's datasheet:
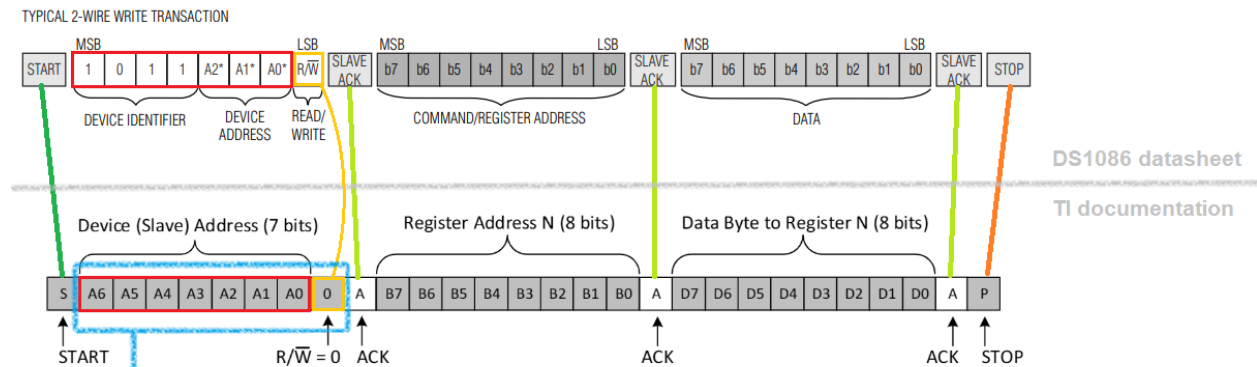


Figure 8. Example I²C Write to Slave Device's Register

WARNING: pay attention to the R/W bit: it is the 8th bit of the device's address. However, many I2C drivers work with 7-bit addresses.

Arduino's I2C implementation in Wire.h uses 7-bit addresses, and the R/W bit is handled by the driver: it is set automatically under the hood depending on the opreation that is going to be performed. So, this byte must be right-shifted of 1 position in order to get the right 7-bit address. Bitwise, it will become: [ 0 | A6 | A5 | A4 | A3 | A2 | A1 | A0]

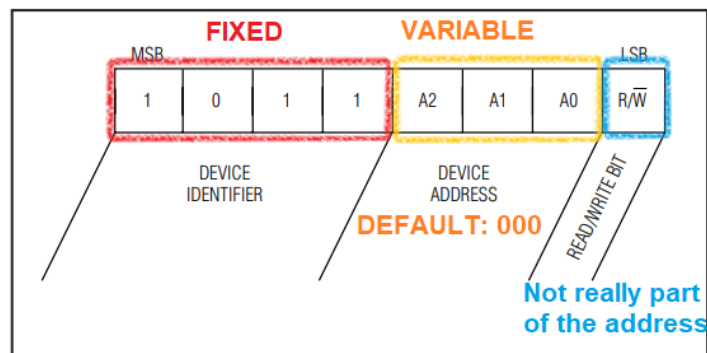For example, consider the following DS1086 device address from the part's datasheet:



Figure 5. Slave Address

Using an Arduino with the *Wire.h* I2C implementation, implies that the R/W bit is handled internally and therefore it must not be specified. Moreover, this specific implementation work with 7-bit addresses, and a right-shift of 1 position is required. The right address to define in the Arduino code would be the following:

$$byte\ addr = 1011A_2A_1A_0X \gg 1 = 01011A_2A_1A_0$$

If there is only one DS1086 device on the I2C bus, and its factory default address has not been modified, the correct address to the define in the Arduino code will be:

$$byte\ addr = 1011\ 000X \gg 1 = 0101\ 1000 = 0x58$$

Here is a write transaction extracted from the programming code for the Arduino UNO, which is available on the repository of this project. Note how the device's address was determined by right-shifting of 1 position the *Slave Address* defined in Figure 5 of the part's datasheet.

```
// Store 0xE000 in the DAC register (DAC HIGH <- MSB, DAC LOW <- LSB)
Wire.beginTransmission(0x58);
Wire.write(0x08); // DAC address = 0x08
Wire.write(0xE0);
Wire.write(0x00);
Wire.endTransmission();
```

**begin the transmission to device at 0x58**
**write the target register's address on the bus:**
**the device with address 0x58 has been triggered by the beginTransmission istruction**

Fire a STOP condition to terminate communication with the DS1086 device

**DAC value is made of 2 adjacent registers: DAC_HIGH at 0x08 and DAC_LOW at 0x09. Since they are adjacent, the two respective bytes can be written in sequence. The write pointer is incremented by the device.**
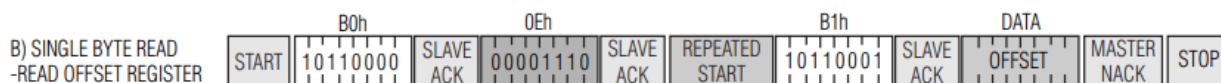
According to the I2C protocol, a read operation from a device's register can be performed as follows. Consider the following extract from the programming code.

```
byte read_PRESCALER()
{
    // PRESCALER register is at 0x02

    Wire.beginTransmission(0x58);
    Wire.write(0x02);                   // Point to PRESCALER register
    Wire.endTransmission(false);        // Do not release the line
    Wire.requestFrom(0x58, 1);          // Request to read 1 byte. The address has the last bit set (READ bit).
    byte retval = Wire.read();
    return retval;
}
```
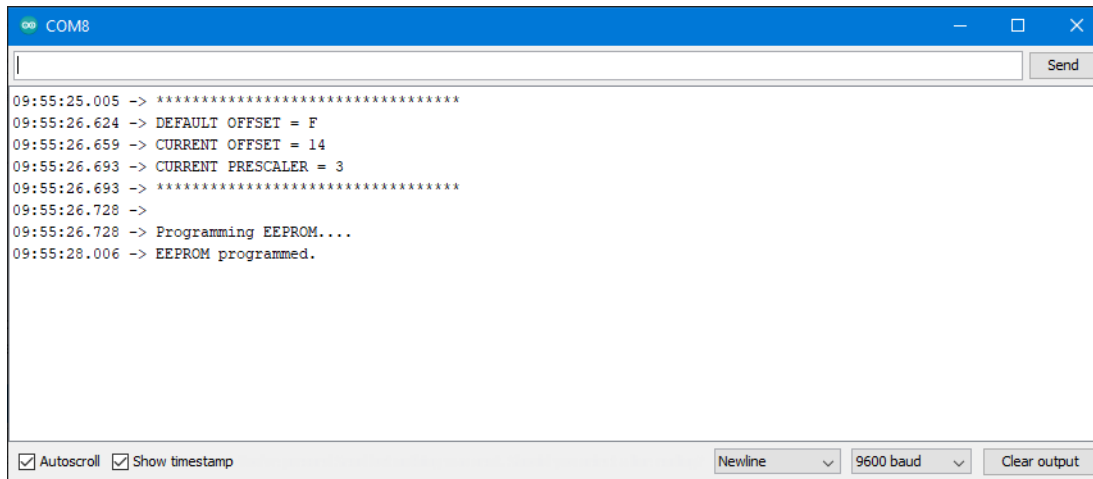
The previous function reads the content of the PRESCALER register of the DS1086 located at 0x58 on the I2C bus. Inside the device, the PRESCALER register is located at address 0x02. Note how the device is triggered at the same way as for a write operation. The first byte sent, again, consists of the target register's address. The third instruction indicates to fire a *repeated start* condition: this condition instructs the device to stay ready for an imminent data transfer, instead of closing the communication with the master. This situation – which is more commonly used in read transactions – is depicted in the part's datasheet:

| B) SINGLE BYTE READ -READ OFFSET REGISTER | START | B0h 10110000 | SLAVE ACK | 0Eh 00001110 | SLAVE ACK | REPEATED START | B1h 10110001 | SLAVE ACK | DATA OFFSET | MASTER NACK | STOP |
|---|---|---|---|---|---|---|---|---|---|---|---|

In order to read from the specified register (0x0E in the picture above, 0x02 in our example), after firing the *repeated start* condition, the device must be triggered again by transmitting its address on the bus, but the READ = 1 bit must follow immediately. In the previous picture, consider the difference between the two bytes following, respectively, the *start* and the *repeated start* condition: they differ from the last bit. The reason is that we have first **written** the address of the target register specifying ~W = 0, then we have requested a **read** specifying R = 1. In our example, this is bit is handled automatically by *requestFrom()*.

The Arduino program execution should first show the current values stored in the chip's EEPROM, then program it to output a square wave with 15 MHz frequency and 50% duty cycle.



```
09:55:25.005 -> *********************************
09:55:26.624 -> DEFAULT OFFSET = F
09:55:26.659 -> CURRENT OFFSET = 14
09:55:26.693 -> CURRENT PRESCALER = 3
09:55:26.693 -> *********************************
09:55:26.728 ->
09:55:26.728 -> Programming EEPROM....
09:55:28.006 -> EEPROM programmed.
```

Inspecting the chip's output with an oscilloscope should provide the result presented below. Be careful to the voltage of the output signal: when powered at 5V as suggested in the datasheet, the chip's output will reach peaks up to 5.8-6V. In the following picture, the yellow trace is a 15 MHz square wave with 50% duty cycle and 3.3V amplitude from a signal generator, while the blue trace is the DS1086's output.

It is possible to see that the DS1086's output can't be directly fed into the oscillator pin of a 3.3V microcontroller. Although this is not stated in the datasheet, **the DS1086 can be powered at 3.3V, reducing the output's voltage significantly.** The operation at 3.3V has not be extensively tested yet, but it can clock successfully a PIC32 microcontroller operating a FLIR thermal camera sensor and a 320x240 TFT on two separate SPI busses, which suggests that the clock signal's quality is good enough. To smooth the overshooting and make the signal's profile a little bit nicer, **place a 100 Ohm resistor in series on the output.** The final result should look like this and be compatible with non-5V-tolerant oscillator inputs on MCU/MPU units.
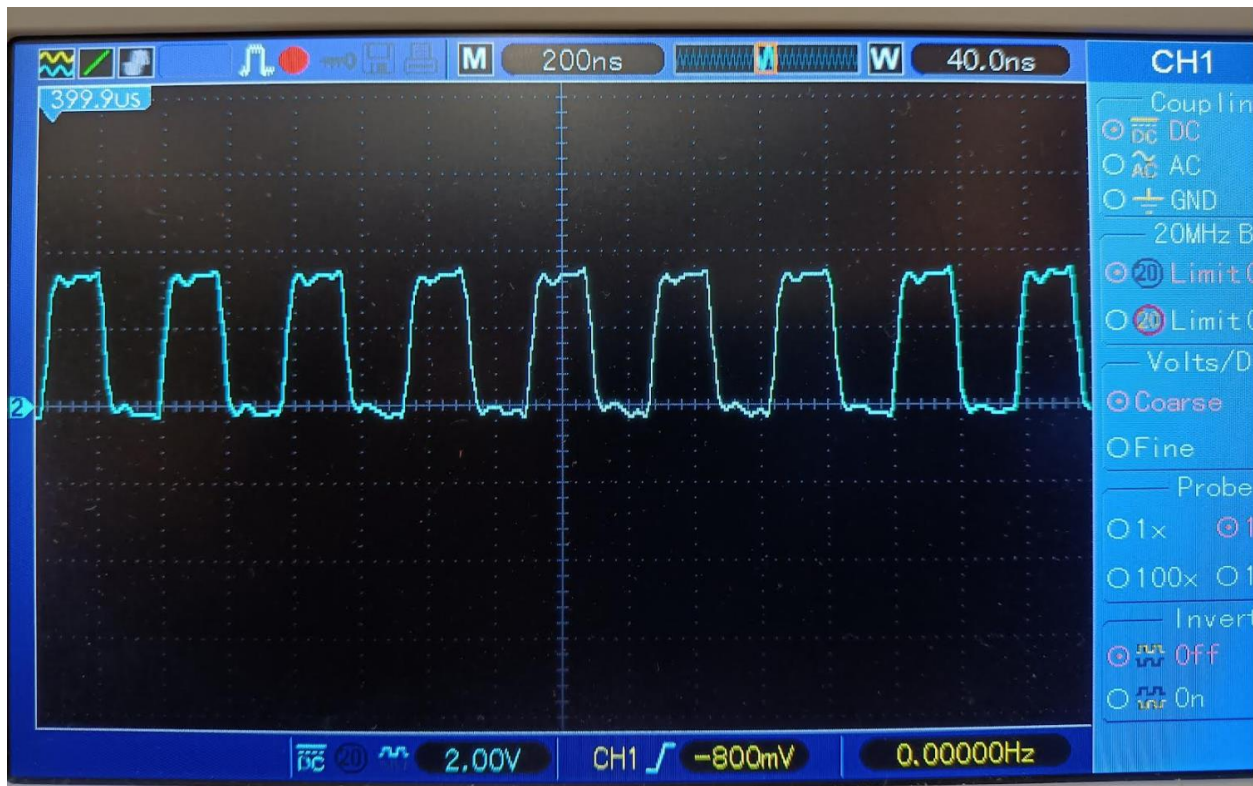


The following picture shows how the above signal has been used to clock the PIC32 microcontroller. Summarizing, the important points are the following:

- Place pull-up resistors on SDA / SCL when performing I2C communications.
- Place pull-up resistors on OE (output enable) and ~PDN (negated power-down).
- Connect the Arduino's ground with the breadboard's ground. The simplest thing to do is to power the breadboard from the Arduino's +5V / GND pins during the programming phase.
- To reduce jitter on the output, place the recommended storage / bypass capacitors (0.1 and 0.01 uF) on the chip's power line, as close as possible to the chip.
- Place a 100 Ohm resistor in series on the output.

Here is the final result:
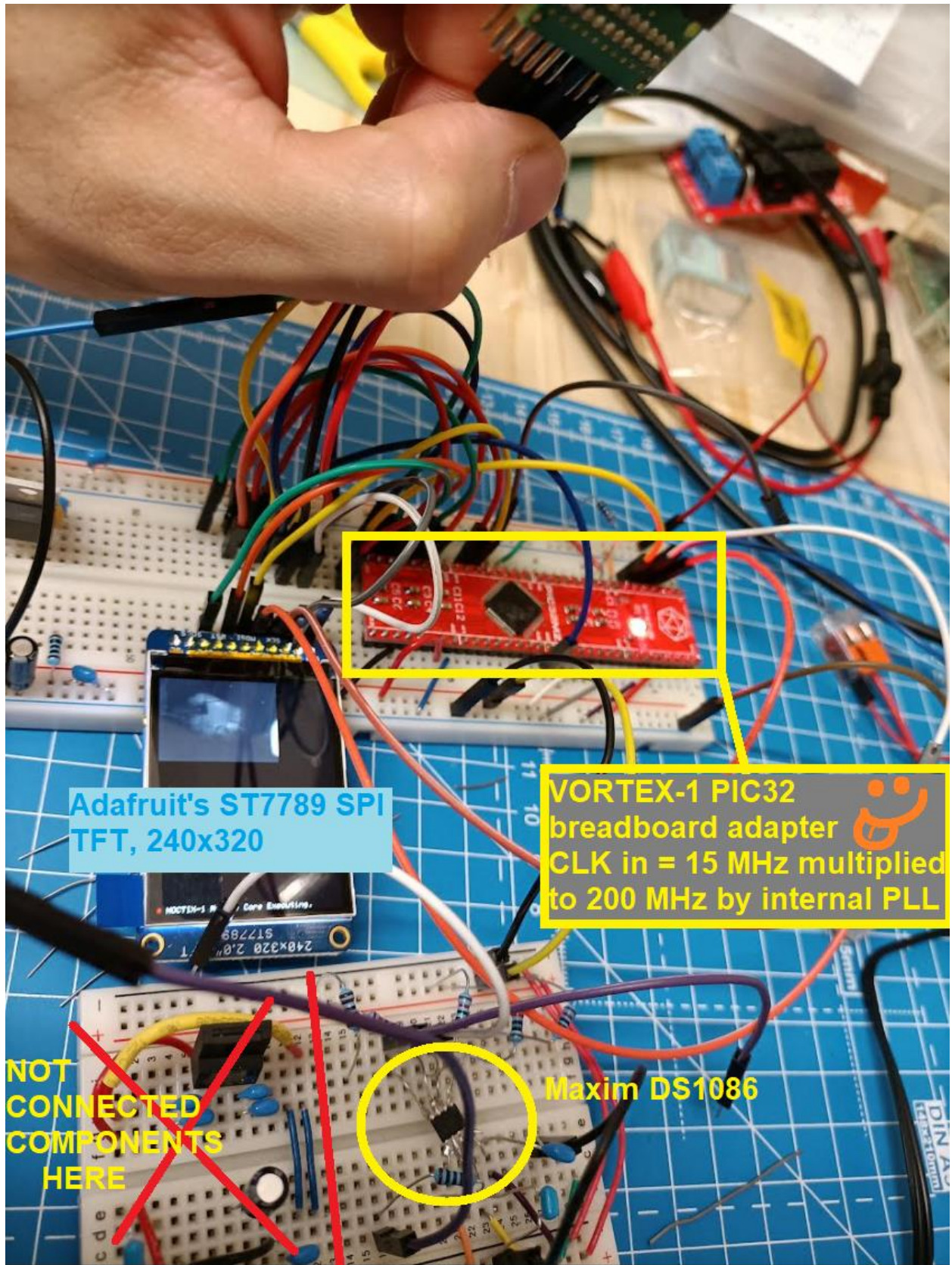
Adafruit's ST7789 SPI TFT, 240x320

VORTEX-1 PIC32 breadboard adapter CLK in = 15 MHz multiplied to 200 MHz by internal PLL

NOT CONNECTED COMPONENTS HERE

Maxim DS1086