Tampere University

Project work extra material

COMP.CE.100 Introduction to Embedded Systems

Interrupts

- Why to use interrupts?
  - **Briefly:** we do not have to wait for e.g. a button press in a loop or an if statement (i.e. no polling needed).
    Instead, the program can be left running and when the button is pressed, the execution moves to the interrupt handler,
    in the project work, it is the `ButtonHandler()`.

Button/switch interrupts

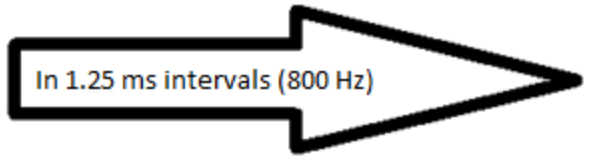
```
//Main loop
while(1){

}
```

The execution of the program is transferred to interrupt handler and after it has done what it needs to do, it will return to the same location in the main program where it was before the interrupt happened

```
//Interrupt handler for switches and buttons. Connected buttons and switches are at bank2. Reading Status will tell which button or switch was used
void ButtonHandler(void *CallBackRef, u32 Bank, u32 Status)
{
    //****Write code here ****

        //BTN0 is clicked
        if(Status==0x01) {


        }

        //BTN1 is clicked
        else if(Status==0x02){

        }
        //BTN2 is clicked
        else if(Status==0x04){

        }
        //BTN3 is clicked
        else if(Status==0x08){

        }

        //SW0 position is changed. 0xE000A068 address needs to be read if you want to know in which position slider is
        else if(Status==0x10){


        }

        //SW1 position is changed. 0xE000A068 address needs to be read if you want to know in which position slider is
        else if (Status==0x20){

        }


    //****END OF OWN CODE*****************
}
```

Timer interrupt (faster)

```
//Main loop
while(1){

}
```

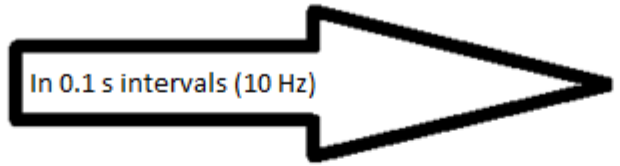


The execution of the program is transferred to interrupt handler and after it has done what it needs to do, it will return to the same location in the main program where it was before the interrupt happened

```
//Timer interrupt handler for led matrix update. Frequency is 800Hz
void TickHandler(void *CallBackRef){
    //Don't remove this
    uint32_t StatusEvent;

    //exceptions must be disabled when updating screen
    Xil_ExceptionDisable();

    //****Write code here ****



    //****END OF OWN CODE*****************

    //*********clear timer interrupt status. DO NOT REMOVE********
    StatusEvent = XTtcPs_GetInterruptStatus((XTtcPs *)CallBackRef);
    XTtcPs_ClearInterruptStatus((XTtcPs *)CallBackRef, StatusEvent);
    //*************************************************************
    //enable exeptions
    Xil_ExceptionEnable();
}
```

Timer interrupt (slower)

```
//Main loop
while(1){

}
```



The execution of the program is transferred to interrupt handler and after it has done what it needs to do, it will return to the same location in the main program where it was before the interrupt happened

```
//Timer interrupt for moving alien, shooting... Frequency is 10Hz by default
void TickHandler1(void *CallBackRef){

    //Don't remove this
    uint32_t StatusEvent;

    //****Write code here ****




    //****END OF OWN CODE*****************
    //clear timer interrupt status. DO NOT REMOVE
    StatusEvent = XTtcPs_GetInterruptStatus((XTtcPs *)CallBackRef);
    XTtcPs_ClearInterruptStatus((XTtcPs *)CallBackRef, StatusEvent);

}
```

Interrupts

```c
int main()
{
    //**INITS, DO NOT TOUCH****
    init_platform();
    *leds=0;
    *rgbleds=0;
    init_interrupts();

    //setup screen
    setup();

    //initial pixels to screen
    draw_init();


    Xil_ExceptionEnable();
    //****End of init**********

    //empty loop
    while(1)
    {

    }

    cleanup_platform();
    return 0;
}
```

- The `main()` function in the reference code will look like this when interrupts are used
  - setting `*leds` and `*rgbleds` to zero is unnecessary because their register values will be 0 when the program is loaded to PYNQ

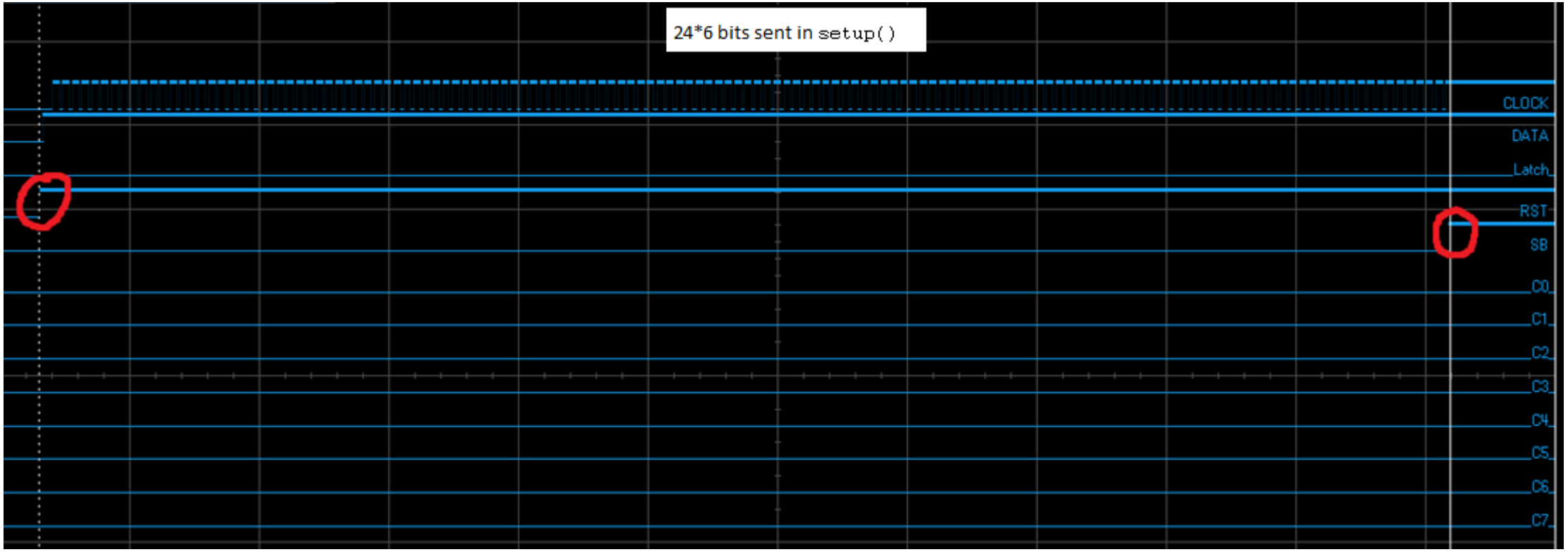
LED matrix

Basic terminology in the project work

- CLK(Clock) = SCK(Serial Clock) = DCK(Data Clock)
- SDA(Serial Data) = SIN(Serial Input) = DATA(dataline)
- SB(Serial Bank) = SELBK (Select Bank)
- $\overline{\text{RSTn}}$ = RST_B = $\overline{\text{RESET}}$ (active **LOW,** inverted)
- LAT = LAT_B = latch_bar = $\overline{\text{LATCH}}$ (triggers on **falling** edge)

NOTE: ColorsShield denotes it only as LAT!

setup()

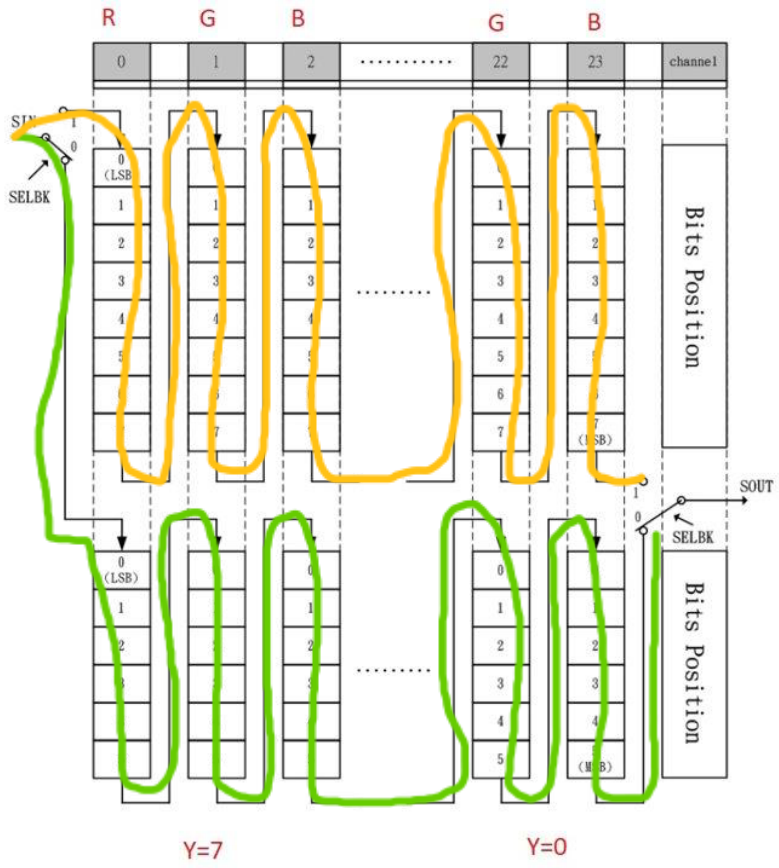


NOTE the lifted reset at the start and the lifted SB bit at the end!

setup()

- In the previous figure, you'll see how the reference `setup()` looks like when measured.
- You will send DM163 chip 144 pcs of bits, all set to '1'
- Can be done by setting the SDA line to '1' when there is a rising clock edge
- The clock can be generated in a `for` loop whose only purpose is to vary the clock line (SCK) up and down
  - In the code, you will set the SCK bit HIGH and then LOW on the next line

DM163 SB-bit



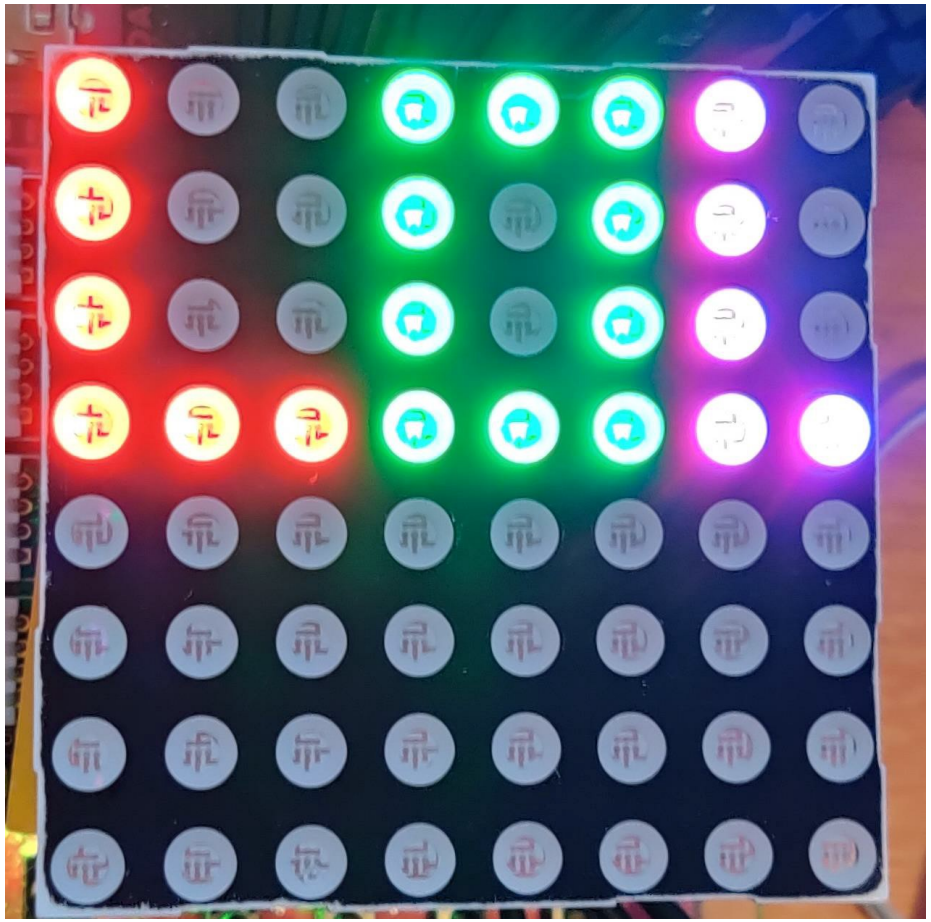
- SB=0
- SB=1
- Why do we want to set the 6-bit register to all ones?
  - **Because the LED brightness is briefly 6-bit reg*8-bit-reg**

- So, if we had set the 6-bit register to zero in bytes 0,1,2

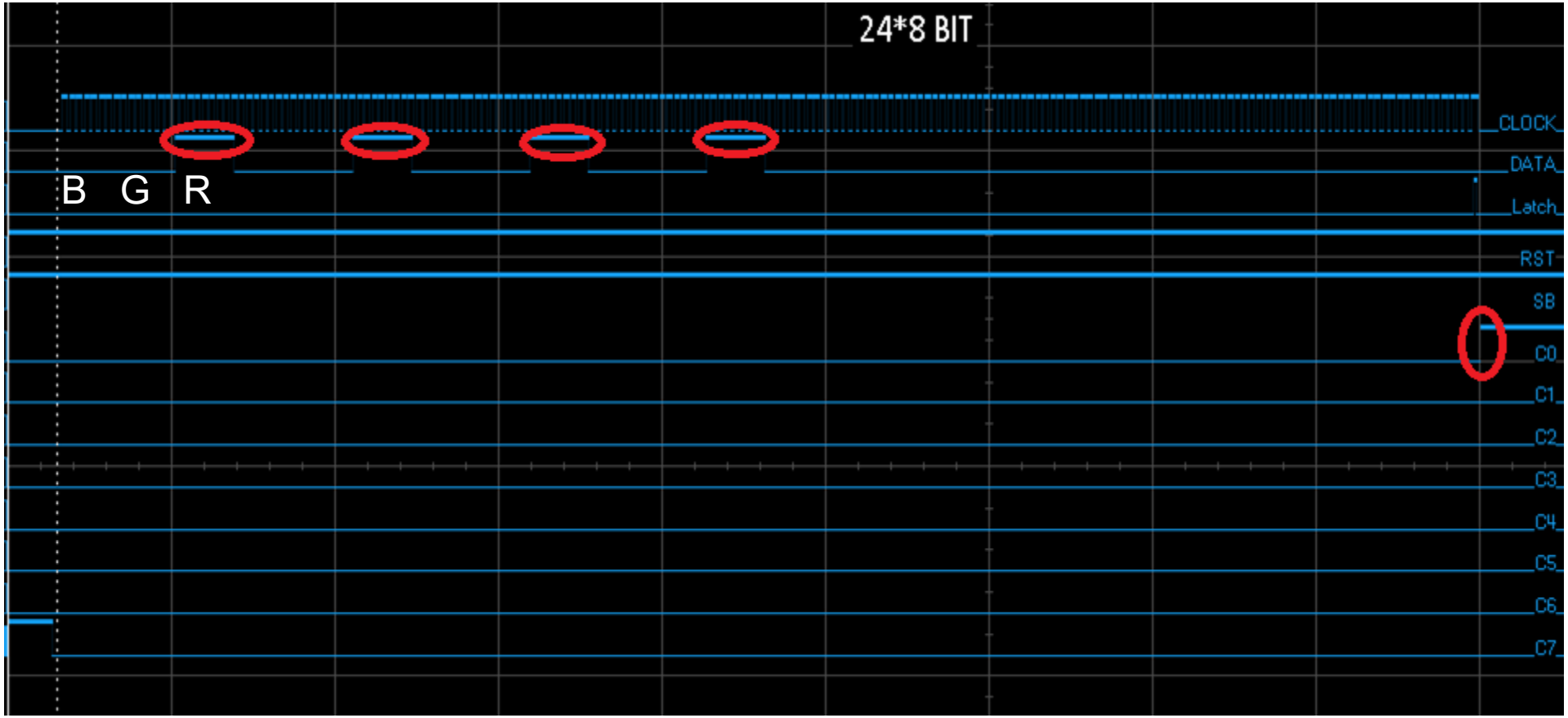
and the 8-bit register to all ‘1’s
=> Y=7 "pixel row" would not be lit in any column

Of course, nothing will light up if none of the channel bits (columns) are ‘1’

run()



run()



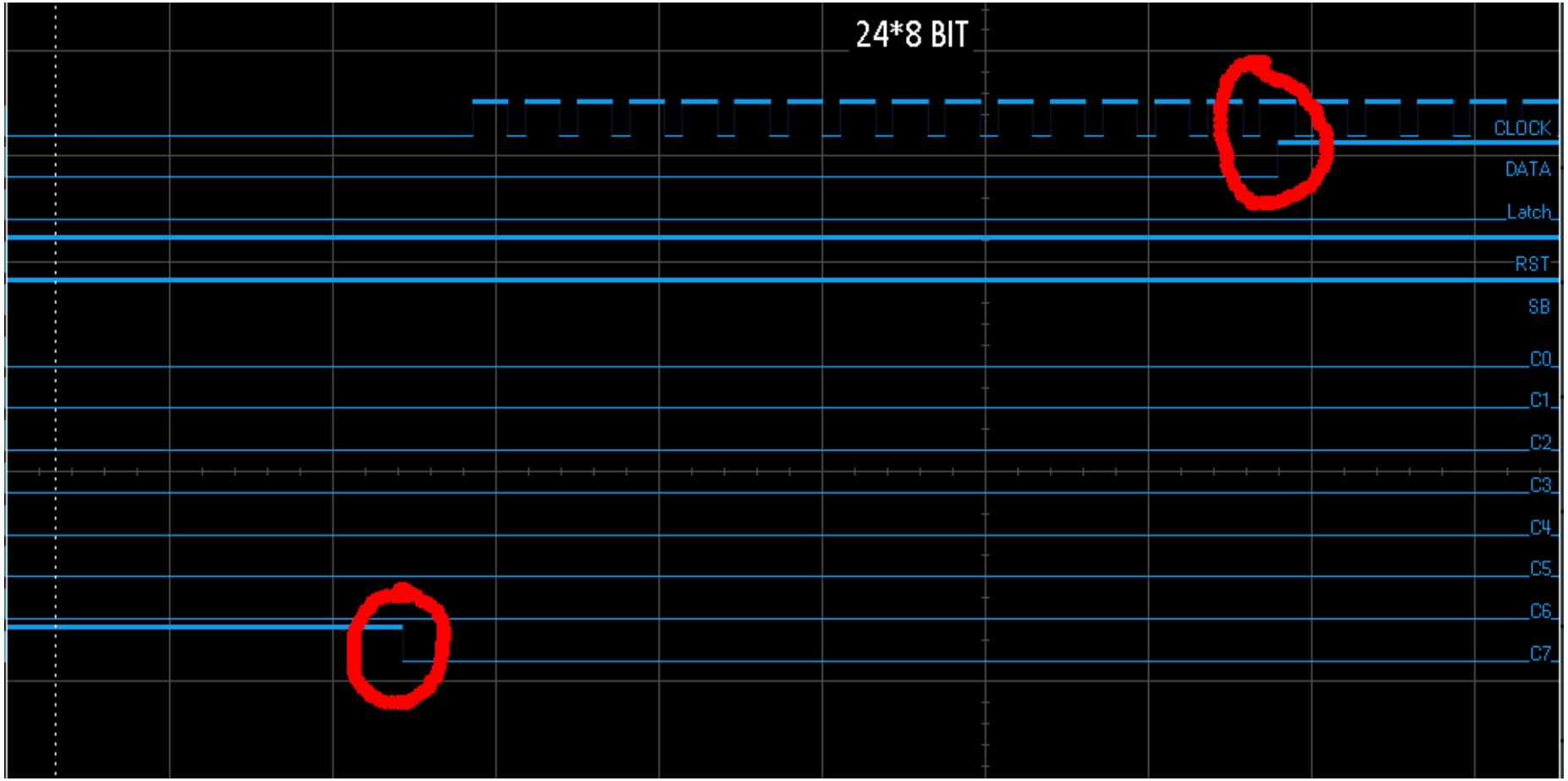
NOTE the 4 red pixels that are being sent (DATA) and the lifted Channel0 bit at the end!

run()

- In the previous figure, you can see what the `run()` looks like when measured with the LOL ending screen on the matrix. `run()` has been measured when

  the left corner (x=0) data is being sent.

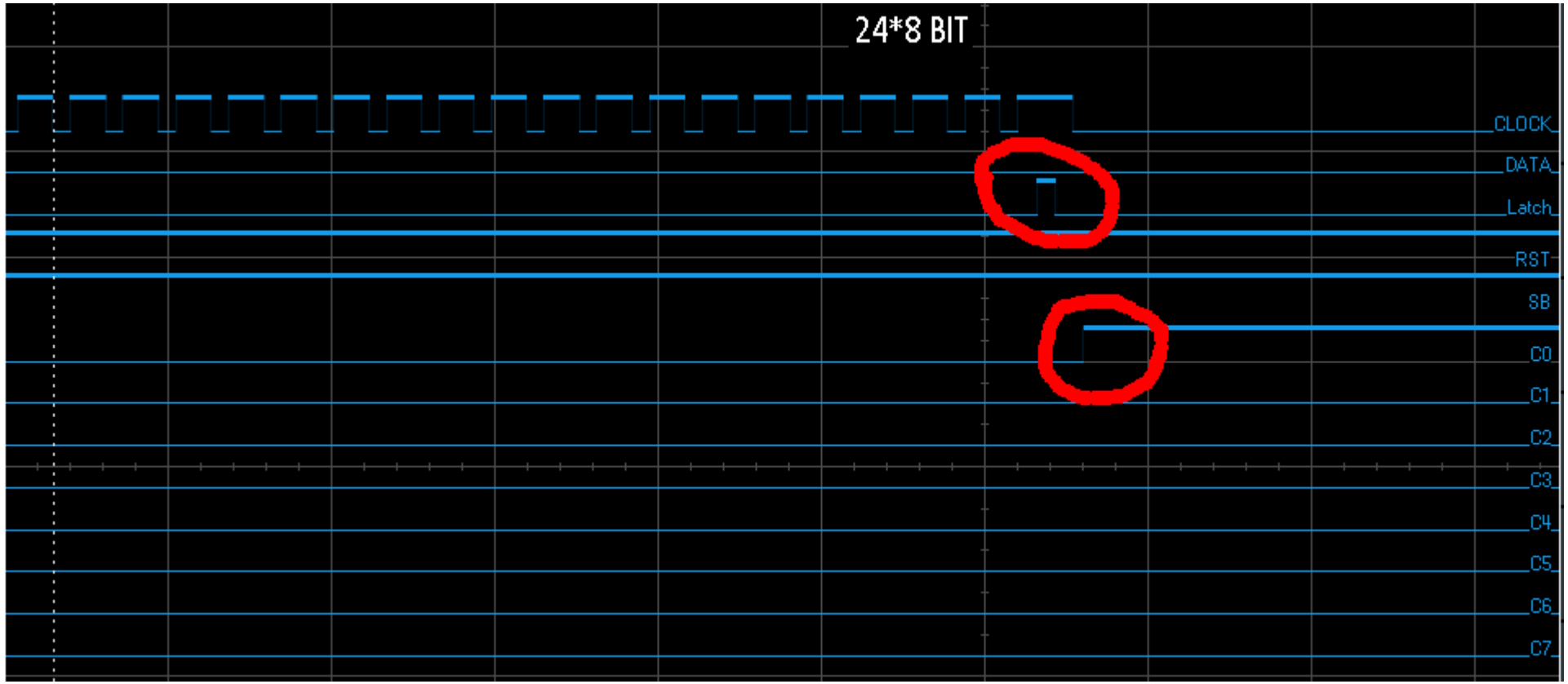
run()



run()

- C0-C7 are channels. Note at what points they are zeroed.

run()

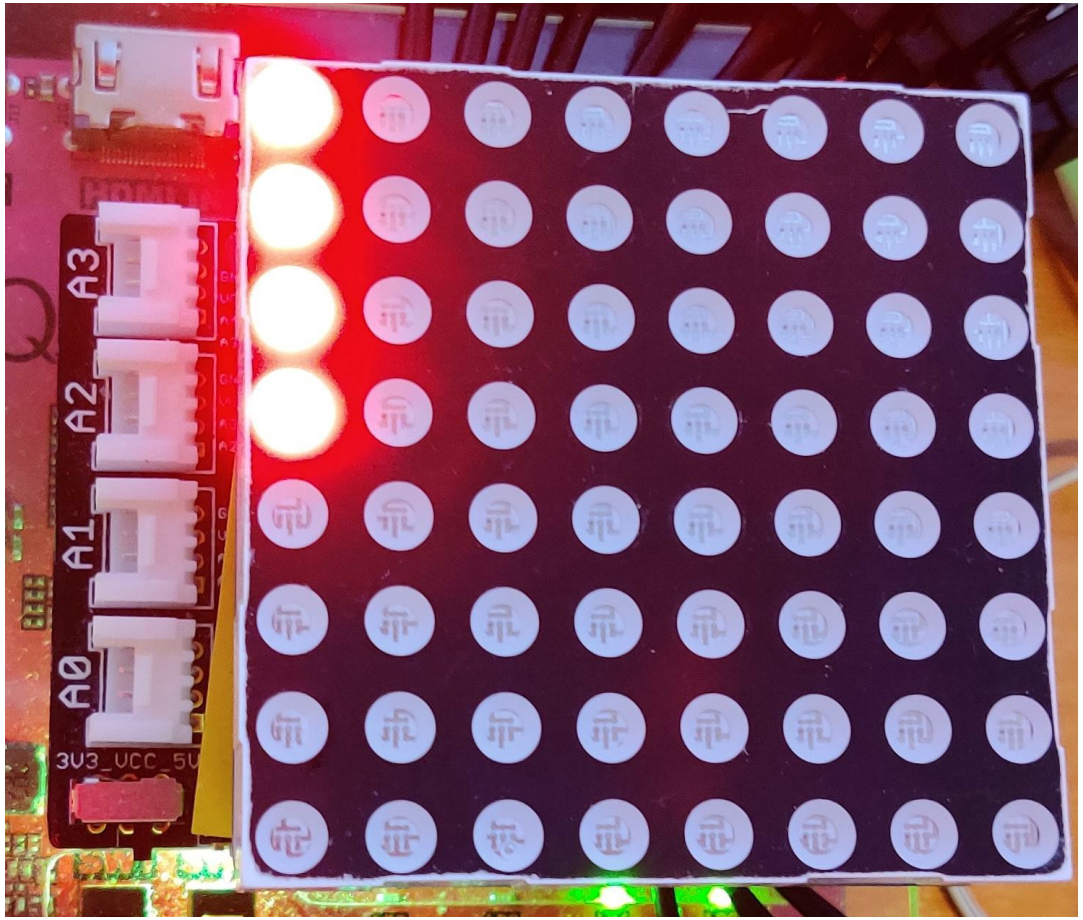


NOTE how we set the latch signal and lift the channel0 bit HIGH!

run()

- `latch()` is called at the end of `run()` and channel0 bit is set to ‘1’.
- Next will be an example picture of how the LED matrix looks like when the processor is paused with a **debug breakpoint** right after setting the channel0 bit.
- After that, a template for the `run()` loop is shown.

run()



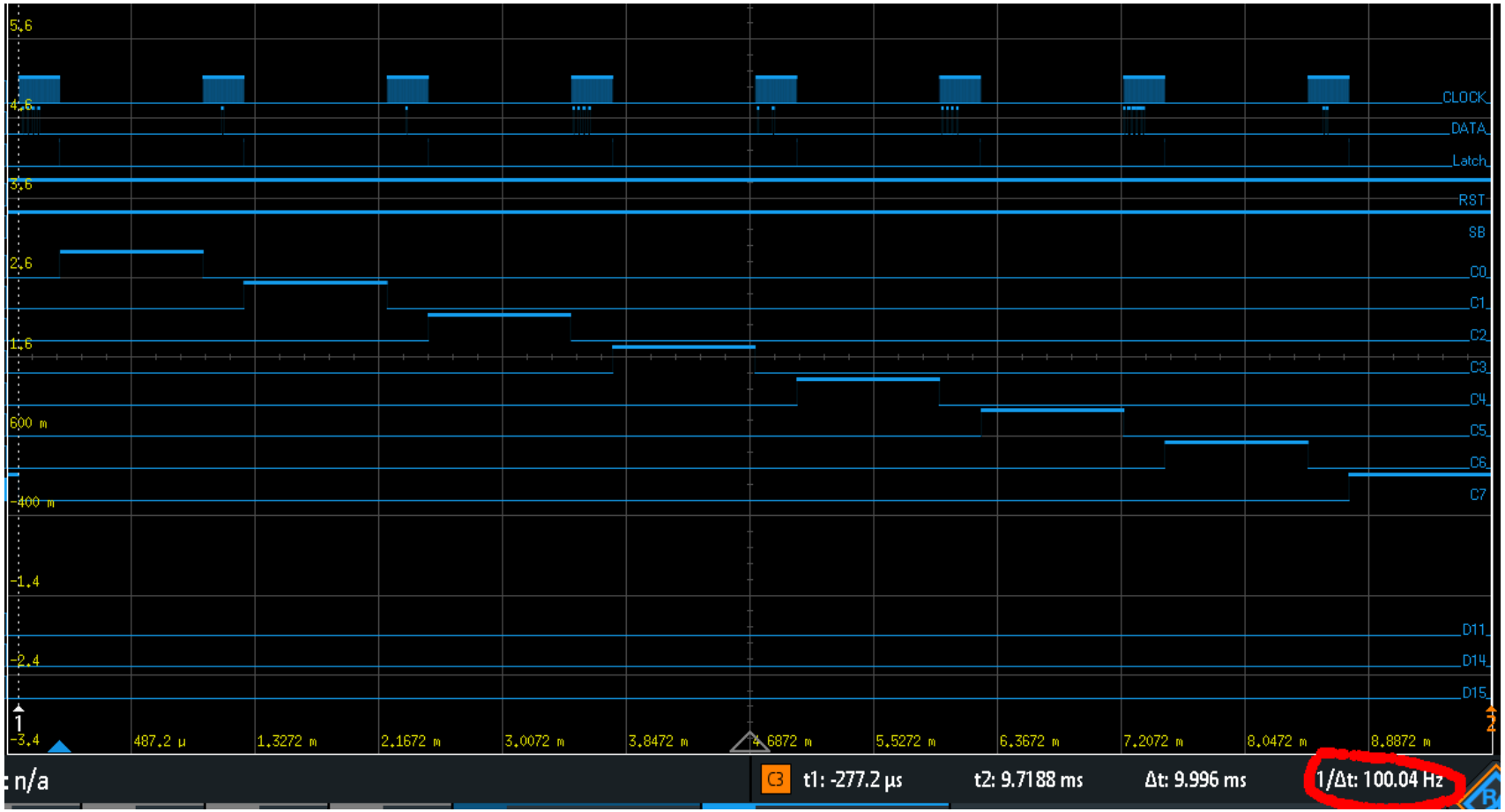
run() template

```
//Put new data to led matrix. Hint: This function is supposed to send 24-bytes and parameter x is for x-coordinate.
void run(uint8_t x){



    //Write code that writes data to led matrix driver (8-bit data). Use values from dots array

    for(uint8_t y=0; y<8;y++){
        for(uint8_t color=0;color<3;color++){
            //Read dots array to some temporally variable. this temporally variable is used in sending data
            for(uint8_t byte_count=0;byte_count<8;byte_count++){

            }
        }
    }


}
```

run()

- Next will be an example how the refreshing of the whole screen looks like when measured
  - LOL pattern is used still on the matrix screen

run()



NOTE how all the channels have been opened, one at a time.

SetPixel()

- Only three lines of code are needed and the purpose of this is to modify the `dots` array according to the function parameters.

latch()

- Only two lines of code are needed and the purpose is to store the color data of `run()` to registers.

open_line()

- Need to go through the channels one-by-one e.g. in a switch-case. Purpose is to show the stored data in the given *x* axis.