TAMPERE UNIVERSITY

Alien shooter game on an LED matrix

Exercise guide – COMP.CE.100 Introduction to Embedded Systems

Index

Exercise guide 2
Specs 2
Exercise Requirements 2
Development environment tutorial 4
Board 4
Xilinx SDK 4
Setting up project 5
Debugger 9
Define memory addresses to words 10
Where to start? 10
General guide 11
Interrupts 11
Timers 11
GPIO input interrupt 11
Important memory addresses and bit order 12
Led matrix 13
C-language 16
Variables 16
Global variables 16
Loops 17
Switch case 17
Arrays 18
Pointers 18
Bit manipulation 19
Static, Extern, Volatile statements 20
Usage of given functions in code 21
Game logic 21
Return 21

Exercise guide

Specs

In this exercise you will learn how the LED matrix works and you will code an Alien shooter game similar to the exercise in Introduction to Programming course. Difference here is that now it is programmed with C and Led matrix controlling functions are done by yourself. Used development board is Pynq-Z1 which uses Zynq-7000 SoC chip. Zynq includes ARM Cortex-A9 processor and Artix-7 family FPGA. In this course only processor is used for programming. However, since every used physical I/O is connected to FPGA, small bridge is programmed to connect processor and FPGA. This is just a nice to know fact in case you start to wonder why FPGA is programmed every time with your program.

Exercise Requirements

To pass this exercise with 10 p you need to code a game with the following functionality (5 p). **Before that, the LED matrix needs to be working (5 p).**

- Ship/gun can be moved using buttons.
- Alien is moving.
- Ship/gun can fire moving bullets.
- Hitting alien increases score or/and missing increases miss score.
- Game has an ending: Too many misses and/or enough hits to defeat the alien. Game ending needs to be shown to the user somehow.
- Score needs to be shown somewhere.
- Game can be restarted.

Look at the video at the game project folder to see what the game could look like (**game.mp4**).

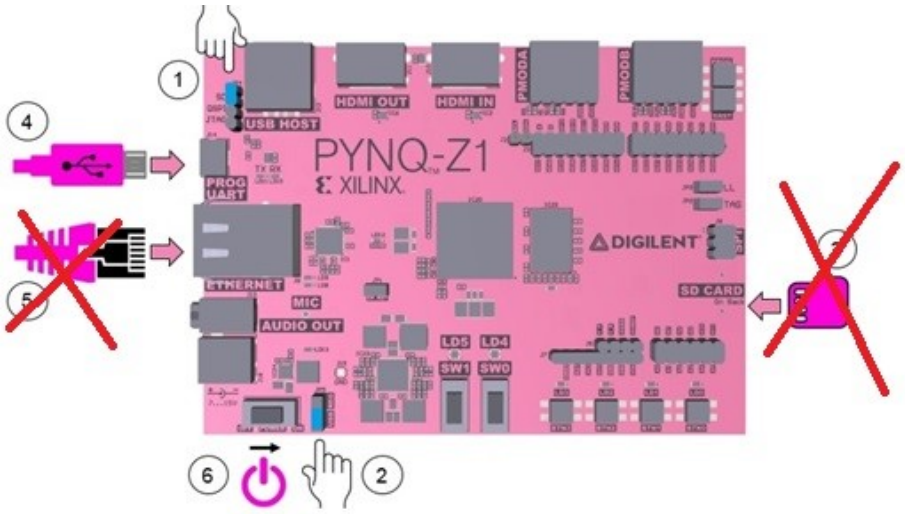
Two “extra” points is given if:

- You code with assembler code that makes button-LEDs (LD0-LD3) go left->right->left unlimited times
    - Functionality: First LD0 is bright -> LD0 goes dim and LD1 is bright ->… -> LD3 is bright, LD2 is dim -> LD2 is bright, LD3 is dim -> ….. LD1 is dim, LD0 is bright
- **Led transition must be visible to human eye!!**
    - See **asm_led_blinker.mp4** video in project folder if unsure what the blinker could look like.
- Look at blinker.S file and figure out what you need to write there and where to call this function: `blinker();`
- This site has a quite good tutorial for assembly: https://azeria-labs.com/writing-arm-assembly-part-1/

Development environment tutorial

Board

When you start to code the program, check that the board is set to use USB power (2), Micro-USB cable is connected (4) and power is ON (6). The LED near the power switch indicates that the board is ON. If an Ethernet cable (5) is connected, just leave it as it is.

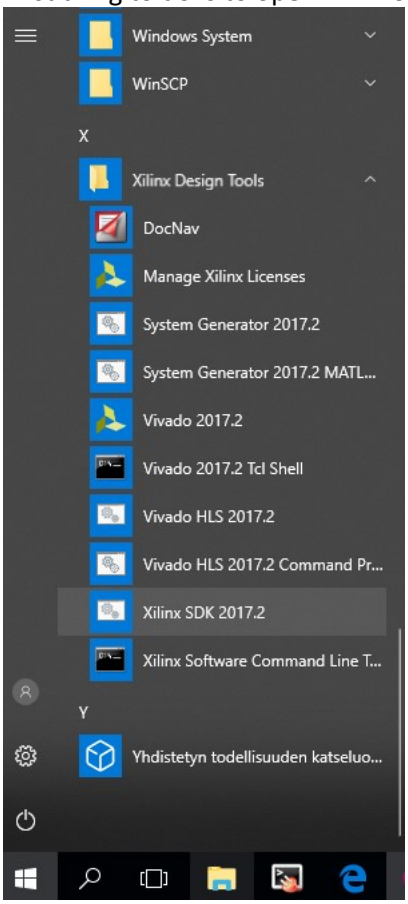


Xilinx SDK

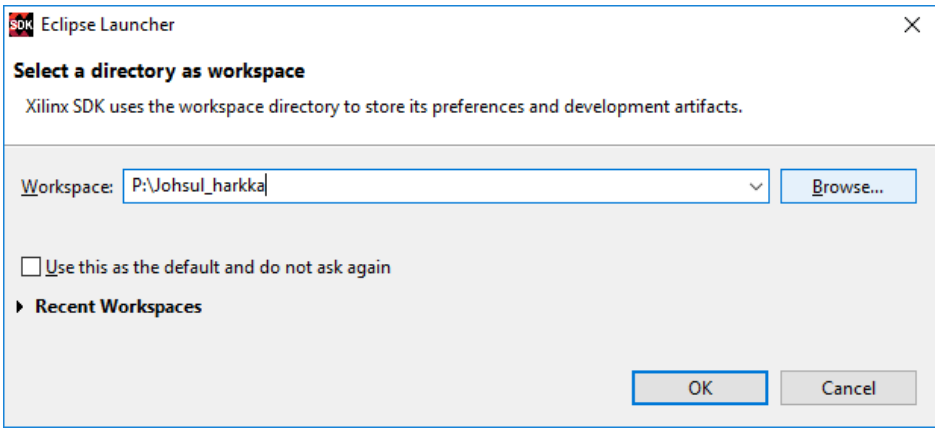
Xilinx SDK is an integrated development environment (IDE), which can be used to develop code to processor (bare metal, embedded Linux and other OS applications).

Setting up project

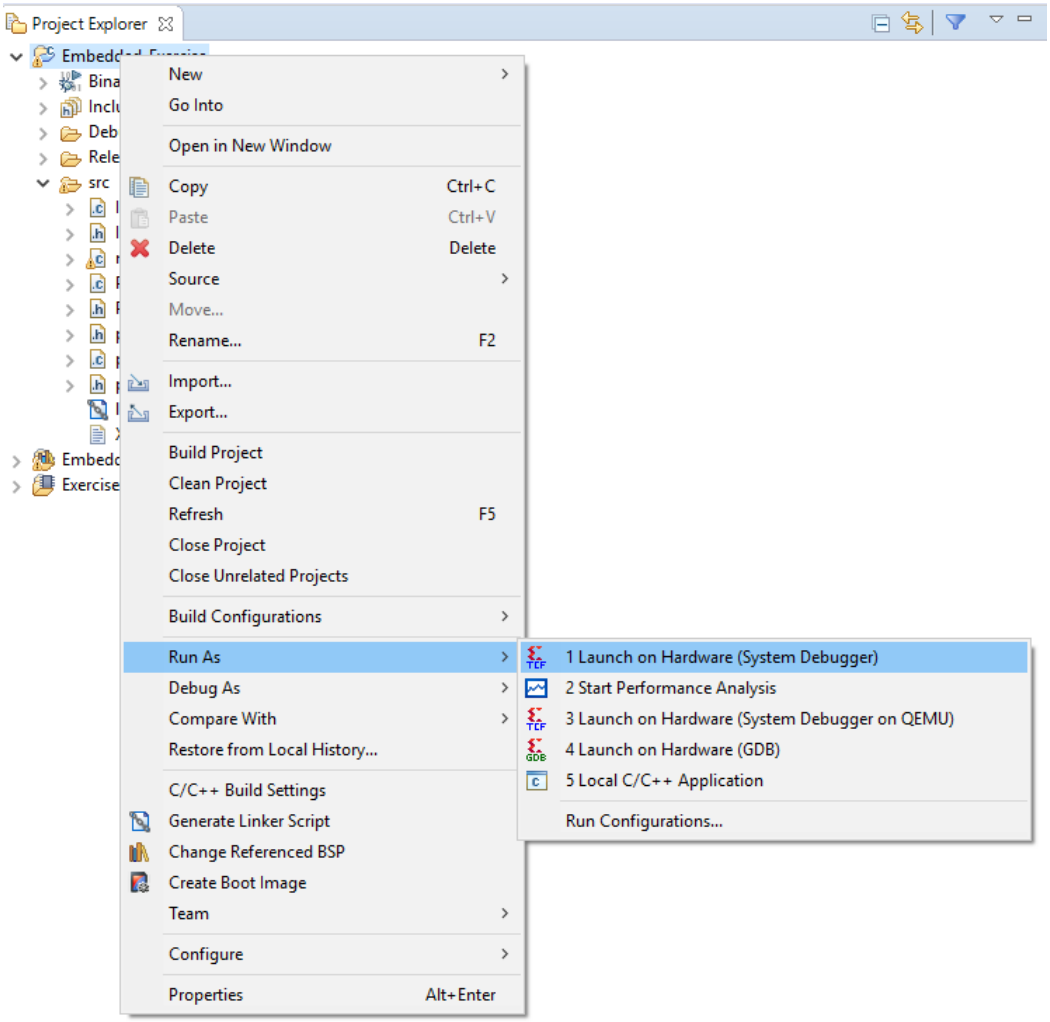
First thing to do is to open Xilinx SDK.



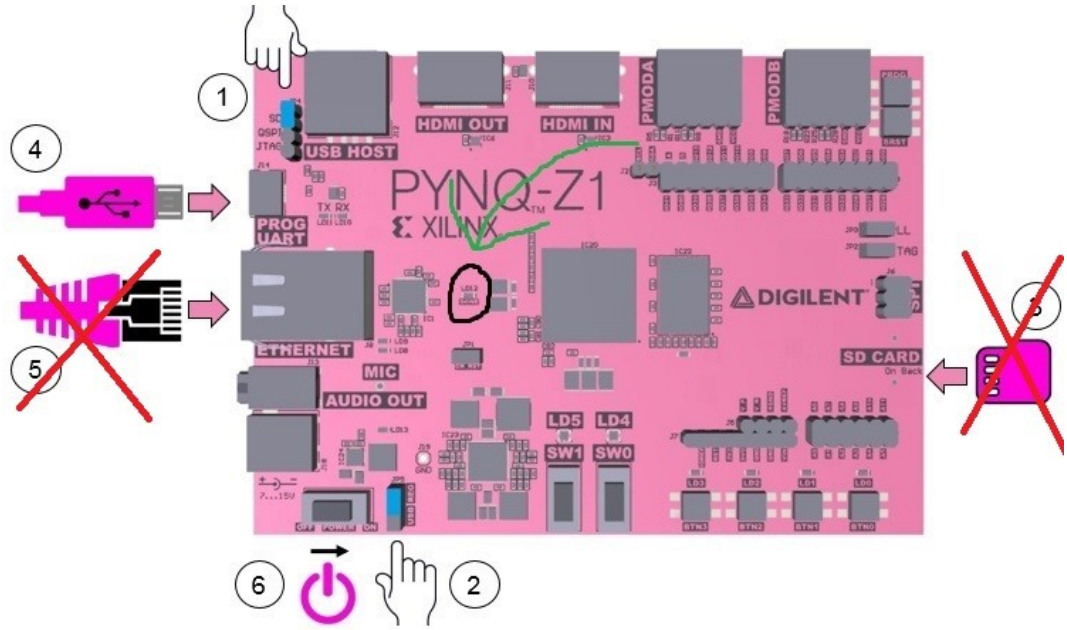
Second thing is to set up the workspace. **Unzip the given project to location as shown in picture**. In TC219, the location is in your personal network drive (**protip**: use “extract here” when unzipping). **On your personal** PC, unzip project somewhere where the file path doesn’t contain any spaces or any special letters like ö,ä.



To test if project is working, do a test run. Right click **Embedded_exercise_normal** on Project Explorer ->Run As->Launch on Hardware (System Debugger). PYNQ needs to be turned on, since software is loaded into the board, so turn it on if it's not already. If no errors come up, software loading to board is done correctly.

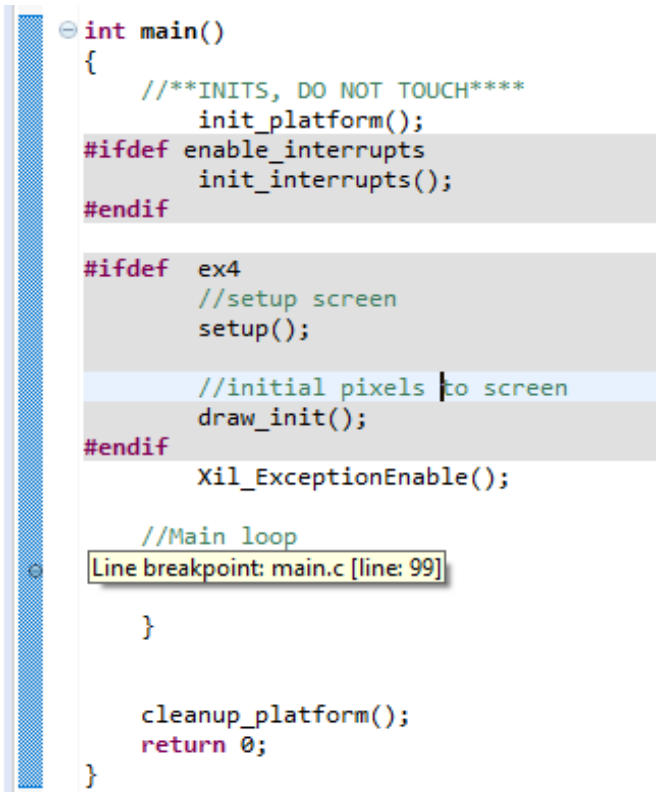


Also see if DONE light (LD12) is ON (*in the following picture with black circle*). This light tells if FPGA has loaded correctly (**NOTHING WILL WORK IF FPGA IS NOT LOADED**). If the light is dim, ask assistant for help.



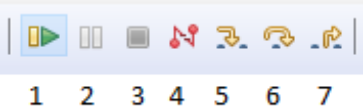
Debugger

Debugger comes handy when there is a situation that the code looks fine, but for some reason it does not work. In those moments using breakpoints and checking variables/register values at processor is a life saver. To use debugger right click **Embedded_exercise_normal** ->Debug As -> Launch on Hardware. It will now run code in debug mode, meaning that the program can be stopped in the middle of running. By default program will stop at beginning of `main()`. Your own breakpoints can be added simply by double clicking the far-left area of code. It will create blue circle to indicate that it will stop running at that place if program gets there:



To remove a breakpoint, just double click that blue circle again.

To use debugger, following toolbar tools are good to know:



1. Resume, when program is suspended.
2. Suspends program.
3. Terminate: after this program needs to be reloaded to the board.
4. Disconnect board.
5. When suspended, run one line of code (when in disassembler, runs one line of assembler code).
6. When suspended, run one line of code and jump over function calls.
7. When suspended, run out of function.

Define memory addresses to words

If you feel like that you don't want to play with pointers all the time when accessing certain memory addresses, you can access memory addresses directly with syntax ***(uint8_t *) 0x1234;** and define this into an easier form:

```
#define test *(uint8_t *) 0x1234
```

This line should be put where global variables are, meaning after `#include` "zzz" stuff. Also note that `uint8_t` is used if memory address contains only 8-bits or less, `uint16_t` with 9-16 bits and `uint32_t` with 17-32 bits.

Where to start?

If you are wondering where to start this project, you should look at ***pixel.c***. You need those functions to make led matrix to functional state. After led matrix is working and you can set pixels with SetPixel() function **(setting values to dots array through this function)**, code the game. Also remember to read comments on the code, they might give useful information.

General guide

Here are some general guidelines for the exercise.

Interrupts

Exercise has three interrupt handlers where two are timer interrupts and one is GPIO input interrupt.

Timers

Exercise's timers use ARM's Triple timer counters. **TickHandler(void *CallBackRef)** uses first timer of TTC0 and **TickHandler1(void *CallBackRef)** uses second timer of TTC0. They are configured in this exercise in a way that TickHandler() is called with 800 Hz frequency and TickHandler1() with 10 Hz frequency. Timers use **108.3336 MHz** clocks, just a good to know fact.

GPIO input interrupt

GPIO input interrupt mechanic in this exercise is pretty simple: when button is pushed and released, it will cause interrupt (falling edge interrupt) and **ButtonHandler(void *CallBackRef, u32 Bank, u32 Status)** is called. Switches on the other hand will cause interrupt when their state is changed, meaning that when switch is toggled up or down, it will cause an interrupt. Only information you need is **u32 Status** variable. This will tell which button or switch was the cause of interrupt. If its value is 0b1 **BTN0** was used, 0b10 means **BTN1**, 0b100 means **BTN2**, 0b1000 means **BTN3**, 0b10000 means **SW0**, 0b100000 means **SW1**. You can ignore the **Bank** variable since you don't need that information anywhere.

Important memory addresses and bit order

Channel

Bit7	**Bit6**	**Bit5**	**Bit4**	**Bit3**	**Bit2**	**Bit1**	**Bit0**
C7	C6	C5	C4	C3	C2	C1	C0

Memory
Address=0x41220000

Control signals

Bit4	**Bit3**	**Bit2**	**Bit1**	**Bit0**
SDA	SCK	SB	Lat	Rst

Memory
Address=0x41220008

Inputs

Bit5	**Bit4**	**Bit3**	**Bit2**	**Bit1**	**Bit0**
SW1	SW0	BTN3	BTN2	BTN1	BTN0

Memory
Address=0xE000A068

Leds

Bit3	**Bit2**	**Bit1**	**Bit0**
LD3	LD2	LD1	LD0

Memory
Address=0x41200000

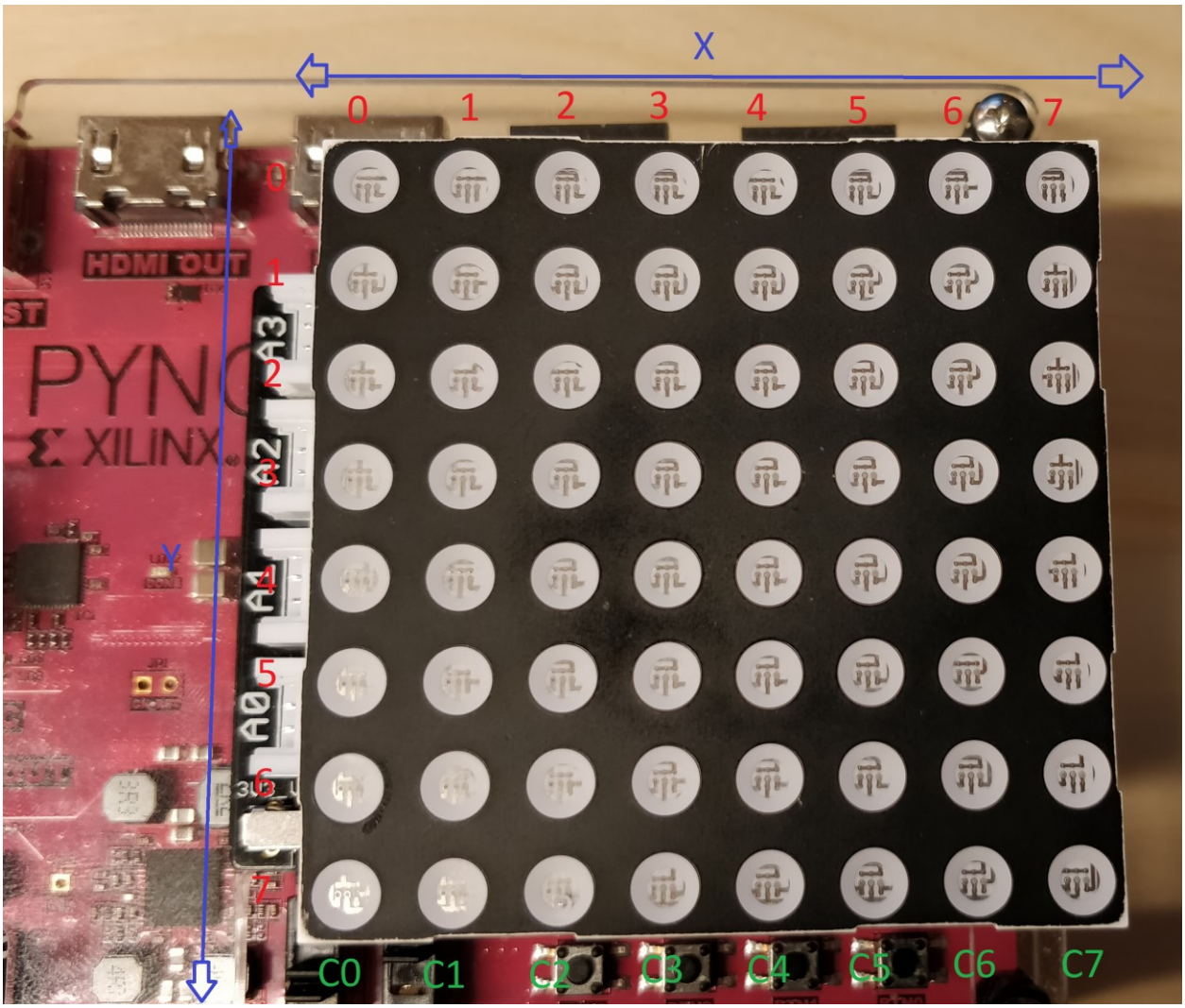
RGB leds

Bit5	**Bit4**	**Bit3**	**Bit2**	**Bit1**	**Bit0**
LD5.R	LD5.G	LD5.B	LD4.R	LD4.G	LD4.B

Memory
Address=0x41240000

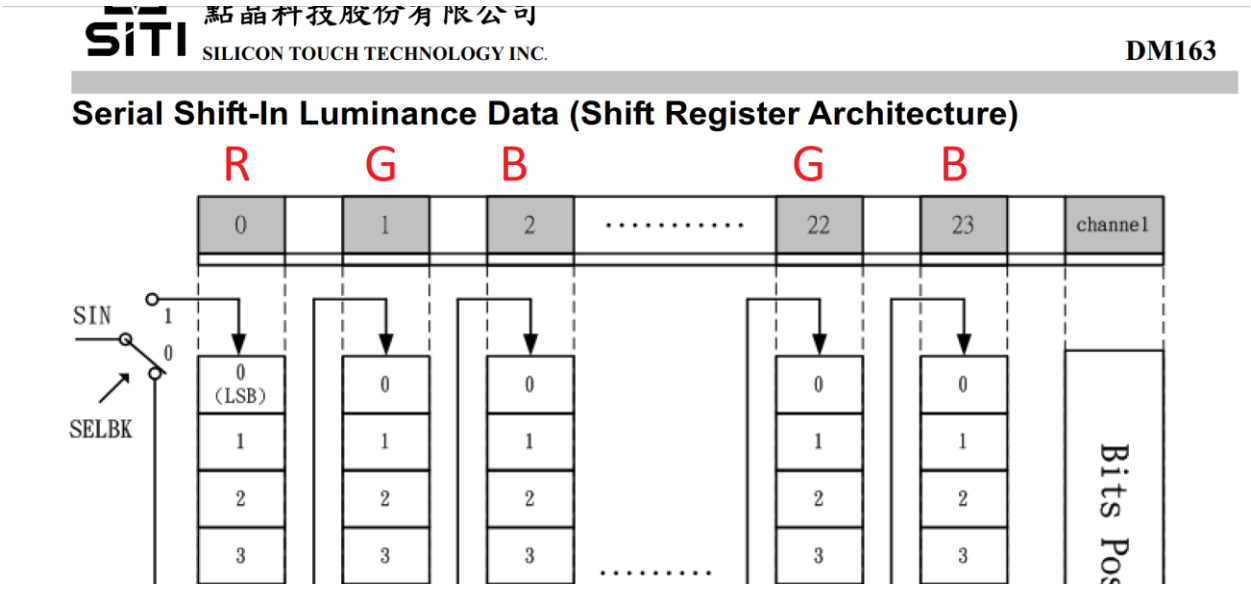
Led matrix

Led matrix screen coordinates is as in next picture (C:s are **channel** bits):

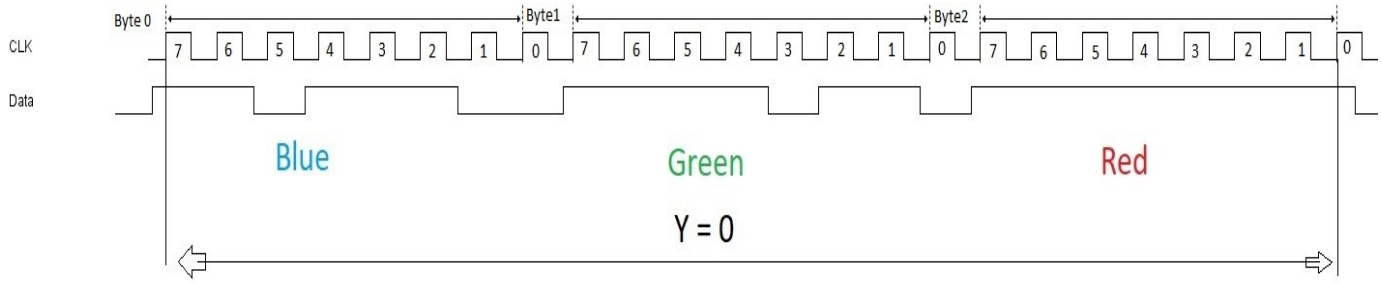


Screen is updated one column per time. But when updates happen at high frequency, screen looks like every led is on at the same time (if every led is set to be on). Remember, when you start to code the Led matrix, read their datasheets: **colorsshield.pdf and DM163.pdf**

The following picture shows color order: how to put data in **(don't look at channel numbers**).



What that picture should tell you (especially when looking at full picture at **DM163** datasheet) is that you need to figure a way to send serial data to Led driver. Hint: Use **control signals** shown before. **SDA** is for data and **SCK** for clock. New bit goes to the driver on **rising edge of the clock**. The picture below (hopefully) shows how the bit transfer works, and the order of bits. The order of bits is MSB first. This means that the bit that is sent first is at the end (MSB) when 8-bit transmission is completed. This is because the data is transmitted to the DM163 chip's shift register, meaning that every bit that comes in is **shifted to one bit higher every time the clock has a rising edge**. This means that the first byte that has come in is at last place at the end.

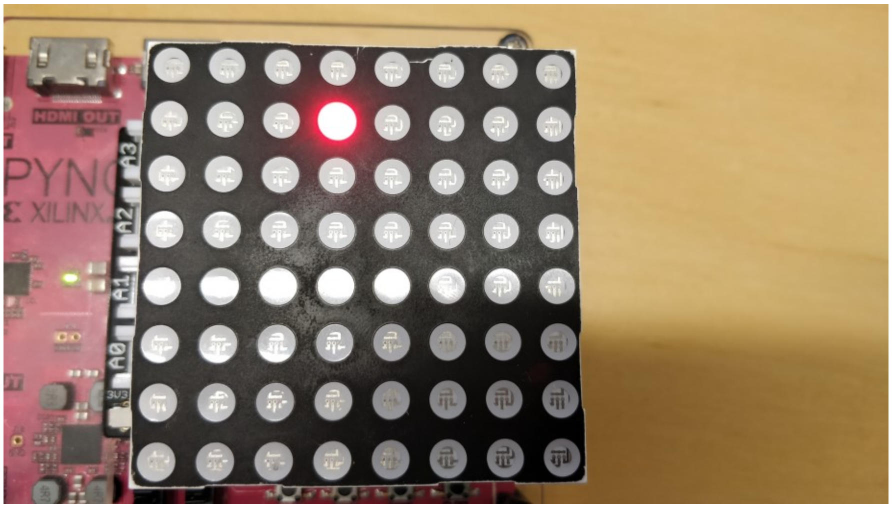


To put simply how things work: Send first bytes for Y0, Y1, Y2…..Y7 (in this order), then choose with Channel bit in which X-columns these are shown. X0 is C0-bit, X1 is C1-bit……. X7 is C7-bit. Y0 is bytes 0, 1, 2…. Y7 is bytes 21, 22, 23.

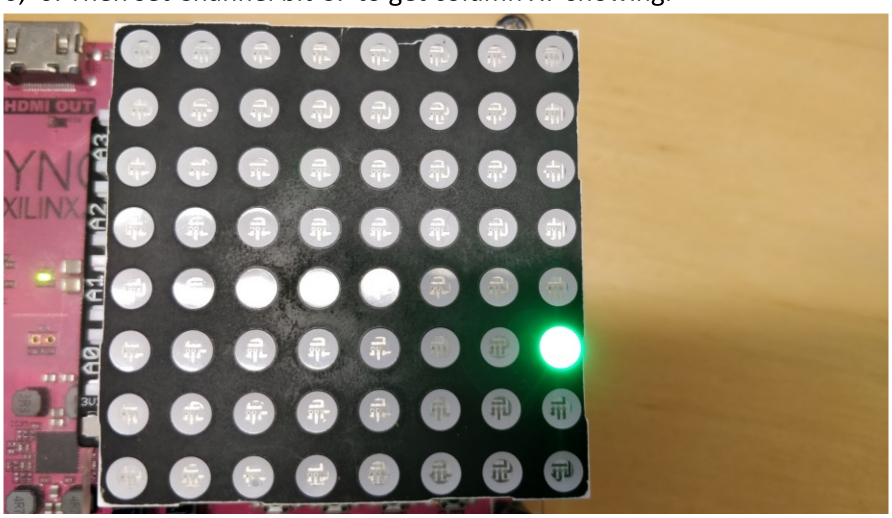
Value determines the brightness of the pixels: 0 is min, 255 is max (values shown in decimal).

So, send Y0 bytes in the order BGR, then Y1 bytes in the order BGR, up to Y7 bytes with blue value first, then Y7 green and finally Y7 red value. After this, set the correct channel bit active to choose the desired column.

Example1: Let's say you want to set red pixel active at coordinate X3, Y1 (at full brightness). You need to send following bytes first (values are in decimal): 0, 0, 0, 0, 0, 255, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0. Then set Channel bit C3 to get column X3 showing.



Example2: Now we are setting green pixel active at coordinate X7, Y5 (at full brightness). You need to send following bytes first (values are in decimal): 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 255, 0, 0, 0, 0, 0, 0, 0. Then set Channel bit C7 to get column X7 showing.



Note that if you want to see anything on the matrix you need also set **6-bit register bank to higher than 0.** This is because of how pixel brightness control works at driver**: brightness= 6-bit register* 8-bit register**.

Here is example how serial transmission works when transmitting **one 8-bit byte** (MSB first).

```
uint8_t data=0b11010110;     //Binary value, this is just example data
for(uint8_t data_counter=0; data_counter<8; data_counter++){
      if((data & 0x80) *ctrl|=0x10;      //SET only BIT4 to 1 in control signal (SDA bit)
      else *ctrl&=~0x10; //SET only BIT4 to 0 in control signal (SDA bit)
     *ctrl&=~0x08; //SET only BIT3 to 0 in control signal (CLK bit)
     data<<=1; //shift one to left
     *ctrl|=0x08; //SET only BIT3 to 1 in control signal (CLK bit)
}
```

C-language

As said before, this exercise is coded with C. Here are some of the most common things that you should know about C. Here is also a useful tutorial: https://www.tutorialspoint.com/cprogramming

Variables

In C, variables must always be defined to some type (unlike auto in C++11 or higher). Variable types are:

- 8-bit variables: `char, unsigned char, int8_t, uint8_t`
- 16-bit variables: `int, unsigned int, int16_t, uint16_t`
- 32-bit variables: `long int, unsigned long int, int32_t, uint32_t`

Unsigned word before variables means that variable is only positive. Example: max positive value of char is 127 and negative -128, but unsigned char max value is 255 and min value is 0.

Global variables

The difference between local variable and global variable is where you can use them. Global variables are usually introduced at beginning of the C-file (after `#include` "zzz" stuff). Global variables can be used anywhere of the code and local variables only in the function where it was introduced (with pointers you can of course pass the memory address of data to sub-functions and then use it there). Note that **if you have two variables with same names** in for example two different C files, you need to

either define the other one as **static if they are not the same variable** or **extern if they are same variable**.

Loops

Loops are pretty much same as in C++, but less intelligent. For loop example: loops 3 times and increases value of `b` by one.

```
for(uint8_t a=0; a<3; a++){
    b++;
}
```

While loop example: Loops while condition is true

```
a=0;
while(a<3){
    a++;
}
```

Switch case

Switch case could be better in some scenarios than if-else structure. For example, in cases where we want to know if a variable is 0, 1 or something else.

```
Switch(a){
    case 0: c=1; break;
    case 1: c=0x20; break;
    default: c=0;
}
```

This could be written with if else structure like this:

```
if(a==0) c=1;

else if (a==1) c=0x20;

else c=0;
```

Arrays

The only array type in C is tables. There are few ways how to declare tables:

`uint8_t a[]="tiptap";` //this will declare table to the size of string and it is initialized.

`uint8_t b[6];` //Array is declared to size 6, but is uninitialized.

`uint8_t c[2][2];` //Multidimensional array is declared to size 2x2, but is uninitialized.

Hint: When using tables, remember that the index starts at 0, so if your table is size 3, then the last index is at 2.

Pointers

The biggest difference in using pointers in C and C++ is that there are no smart pointers in C. So, to use pointers, use the "classic" way:

```
uint8_t *a;

uint8_t b=0;

a=&b; //a has b's address

*a=4; //now b has value 4

a=0xAABBFF00; //a has some random address
```

Remember that the pointer must be declared with the size of data you want to handle. Meaning that you can't have an 8-bit pointer to point to 16-bit data. So, **don't do something like this:**

```
uint8_t *a;

uint16_t b=0;

a=&b;
```

You might also see this kind of pointer syntax: ***(uint8_t *) 0x1234=3**; for reading address* or ***a=*(uint8_t *) 0x1234;** for writing to address.* This is one kind of way to write or read from memory address without

making pointer variable. Either way is fine, but some compilers don't like if memory address is given directly to pointer variable.

Bit manipulation

Bit manipulation is one of the most important things in low level programming since there will be situations when you need to set one I/O to high level without affecting another I/O at the same address. Most common way to manipulate bits is to use bitwise operations (AND, OR, XOR, One's complement).
Example:

`a|=1;` (same as `a=a|1`) //OR operation, and LSB (Least Significant Bit) of variable `a` is set to 1 (does nothing if it's already 1)

OR Truth table

A	B	new A
0	0	0
0	1	1
1	0	1
1	1	1

`a&=1;` (same as `a=a&1`) //AND operation. Variable `a` is set to 0 if it does not have LSB set.

AND Truth table

A	B	new A
0	0	0
0	1	0
1	0	0
1	1	1

`a^=2;` (same as `a=a^2`) //XOR operation. Toggles bit1. If a bit1 was 1, now it is 0. If bit1 was 0, now it is 1.

XOR Truth table

A	B	new A
0	0	0
0	1	1
1	0	1
1	1	0

`a=~a;` //One's complement to `a`. For example, if value of `a` was 0b1010, now it is 0b0101.

`a<<=1;` (same as `a=a<<1`) //shift bits to left. If `a` was 0b1 (1 in decimal), now it is 0b10 (two in decimal).

`a>>=2;` (same as `a=a>>2`)//shift bits to right. If `a` was 0b1100, it is now 0b0011, i.e. 0b11.

`a&=~0x02;` //clears bit1 from `a`

Static, Extern, Volatile statements

Other useful thing to know is how to use static, extern and volatile statements. Static and extern statements affects variable or function visibility, meaning that if variable or function is introduced as **static**, they can only be used within that C-file. **Extern** on the other hand makes function or variable visible to other C-files. **Volatile** statement is required when global variable is used in interrupt subroutine. This is to make sure that the compiler doesn't optimize the variable away.

Usage of given functions in code

setup() : This function is used for to initialize led matrix. This means you need to set reset to inactive state, transmit 6-bit register values to led matrix driver chip and change register bank where transmission goes after setup (***SB-bit in control signal!!***). Changing SB-bit to 1 will cause transmissions go to 8-bit register bank.

SetPixel(): This is used to set one pixel in given coordinates. You only need to write new values to **dots** array in given coordinates.

latch(): This should be used to generate latch signal. *Hint: requires only 2-lines of code. Read colorshield.pdf page 3*

run(): Function should be called every time that TickHandler() is called (800 times/s). Run() is used to transmit new values to screen in given X-column (variable `x` is used to indicate x-column). Pixel values are taken from **dots** array.

open_line(): Function will open one line to show data in the given x-axis.

All of the above functions are found in **Pixel.c**.

blinker(): this function is used to make led blinker if you are doing assembler task. This can be called anywhere in the code like normal function call. Note that this function is coded in assembler and is found in **blinker.S**

Game logic

There are some different ways how to code game logic, but here are some hints how logic could be programmed:

- Using variables to define locations is a good thing.
- Illusion of movement is done by clearing old pixels and drawing pixels to new place
- Function **SetPixel()** is used for setting and clearing pixels.
    - Only modifies dots array values given parameters, nothing else
- Create functions for different happenings, for example ship movement.
- **TickHandler1()** is good for creating movement for objects (or easy way to create blinking led).

Return

When your game is finished, demonstrate it to an assistant to receive your points. If doing remotely, return your project source codes using a project return guide and take a video of your project (show your student card in the video).