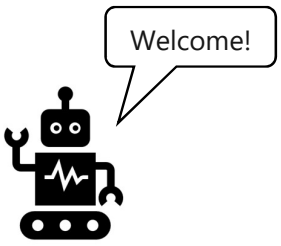


LED SIGN

Project Book

Parallel Systems Lab
המעבדה למערכות מקביליות

ABSTRACT

This document describes the design and attributes of the LED sign control software created as project 236504, in PSL lab at Electrical Engineering Department, Summer 2019.

Submitted by: Ofri Kahana, Samah Anabusy

Supervised by: Oz Shmueli, EE Department, Technion IIT

Contents

Introduction

Motivation

The new EE department building requires a modular sign to display information for visitors. For instance:

- A welcome message
- Parking spots availability
- Directions (arrows)
- Logos & Images

Goal

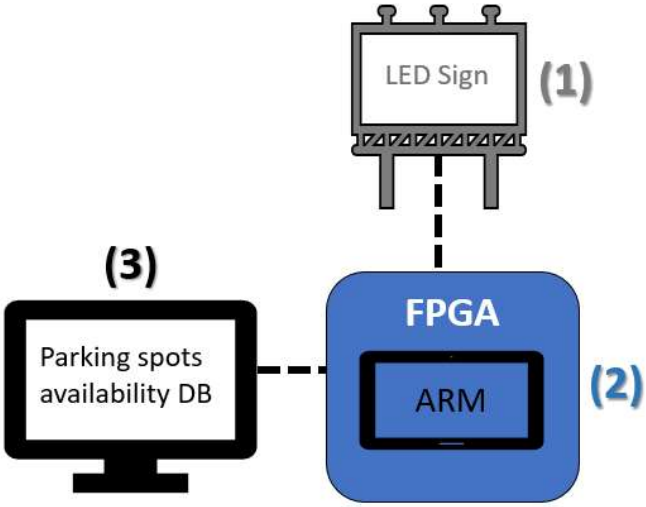
To enable simple operation of such sign, the following features are desired:

* Displaying text with scrolling ability and modifiable color
* Displaying pictures from a pre-defined database with modifiable color
* Editing pictures database
* Rotating/flipping the displayed content of the board (or part of it)
* Division of the board to sub-displays to be controlled separately

In this project we are to develop a complete and successfully functioning software control system for displaying such content on a LED screen, with implementation of all the desired features mentioned above.

System Overview

The system consists of 3 major parts:



We were given a sign with LED light bulbs which can display colors according to 0-255 RGB values, a Z-board with an ARM + FPGA sub-systems, and a database containing information about available parking spots which was collected from sensors located on the parking spots.

Our software was loaded to the ARM unit on the Z-board, which both handled the communication with the external database, and sent RGB information to the FPGA which loaded it to the actual LED sign.

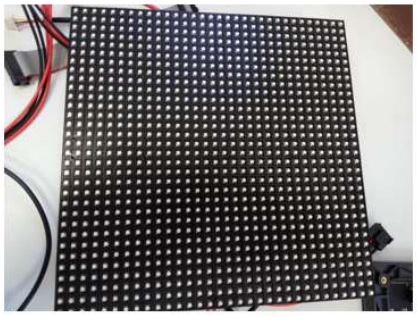
(1) LED Sign

A LED display is a flat panel display that uses an array of light-emitting diodes as pixels for a video display. Their brightness allows them to be used outdoors where they are visible in the sun for store signs and billboards.

In this project the sign consists of LED matrices, referred to as blocks. In each block (in default) there are 32*32 LED light bulbs. The following are pictures of a single matrix and its connectors:

Back

Front

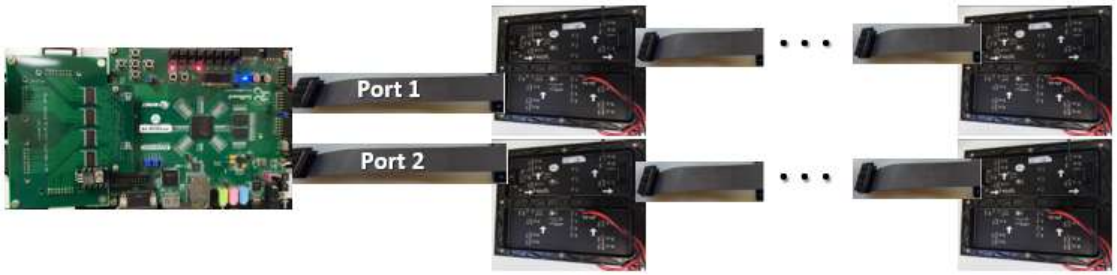


LED connector

The system writes information to the matrices via (in default) 4 ports. Typically, each port represents a row of matrices connected to each other.

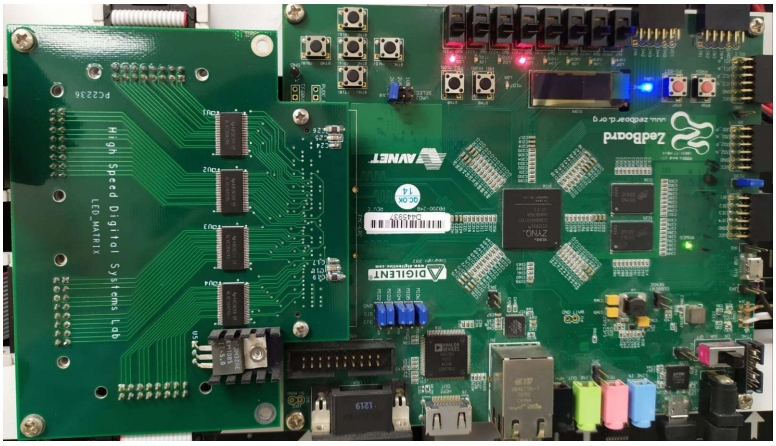
The information written is RGB values for each LED light bulb, basically a bitmap.

When information is pushed, the FPGA flickers the light bulbs on the matrices according to the RGB data, to create the different shades of colors (elaborated in (3) section). The following is an illustration of matrices connected in a port:



For debug purposes, we started working with a single port with 2 matrices connected to the main unit, and eventually expanded our control to a modifiable scale board.

(2) FPGA + ARM



The core of the system is a Xilinx SoC controller, which is a processor-centric platform that offers software, hardware and I/O programmability in a single chip.

It incorporates a dual core ARM based processing system and Xilinx Programmable Logic (PL) in a single device.

The Zynq-7000 architecture enables implementation of custom logic in the PL and custom software in the processing system. The integration of the processing system with the PL allows levels of performance that two-chip

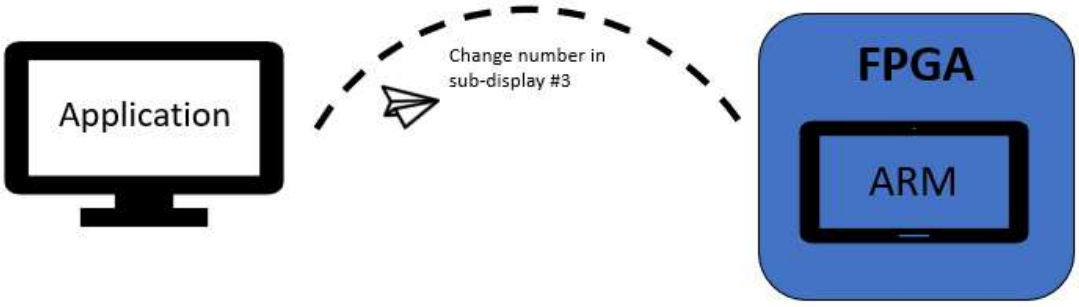
solutions (e.g., an ASSP with an FPGA) cannot match due to their limited I/O bandwidth, latency, and power budgets. [1]

Using Xilinx SDK we were able to load our software to the processing system, utilizing the TCP/IP communication capabilities of the controller (for interaction with the user) as well as FPGA control of flickering LED light bulbs to display multiple colors. [2]

(3) Communication Protocol with Parking Spots Availability DB

The system communicates via TCP/IP with an external application which handles the information about the availability of parking spots.

The application runs as a server to our client, sending requests e.g. increase the numerical value displayed in sub-display #3 which refers to the northern parkin lot.



We have developed a communication protocol to receive and parse commands from a user who has access to the Parking Spots Availability DB.

The available commands are as follows:

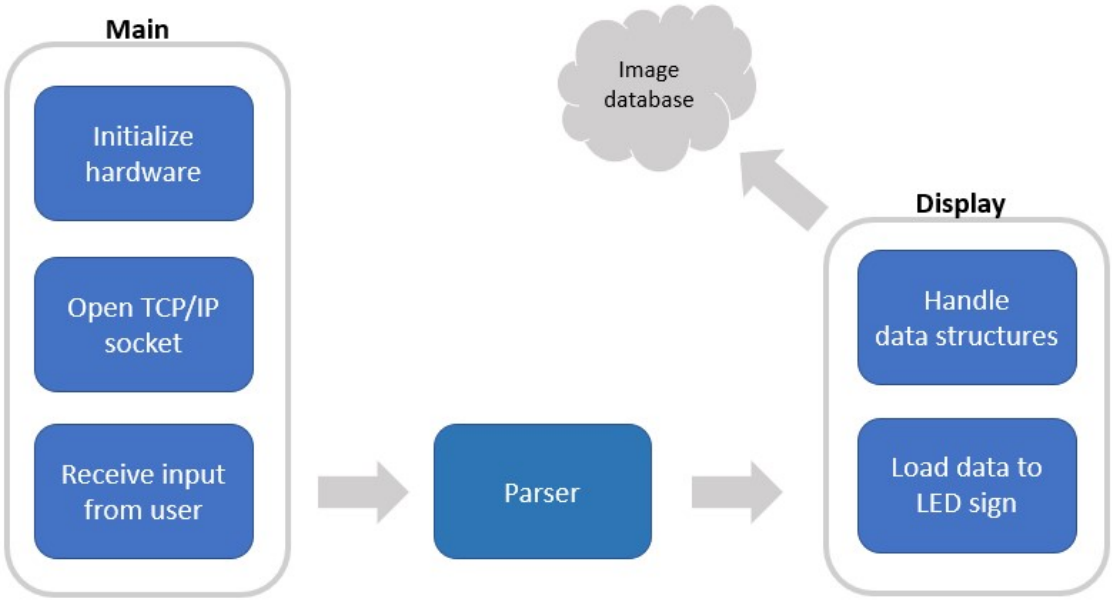
Initialize board
Insert/Clear/Delete sub-board
Insert/Delete text/picture object
Update text/picture
Change text/picture color
Flip/rotate board (or part of it)
Exit (delete board)

They will be described in detail in the Command chapter (in User Guide).

Developer Guide

Software Structure

The flow of information in the software can be divided into 4 major logical units:

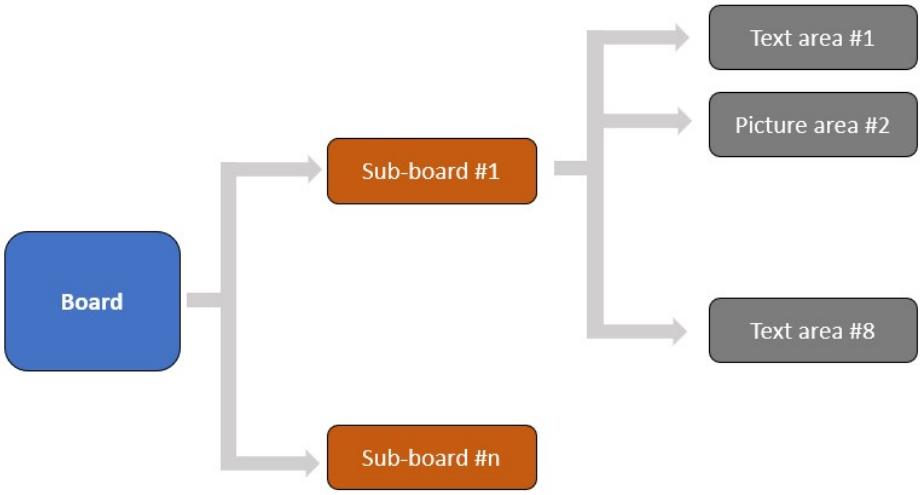


The main unit handles hardware initialization, communication with parking spot availability database via TCP/IP, and reading inputs from the user. The inputs are then parsed according to our original protocol and translated into data structure commands and loaded into the board. When needed, an update of the image database is performed.

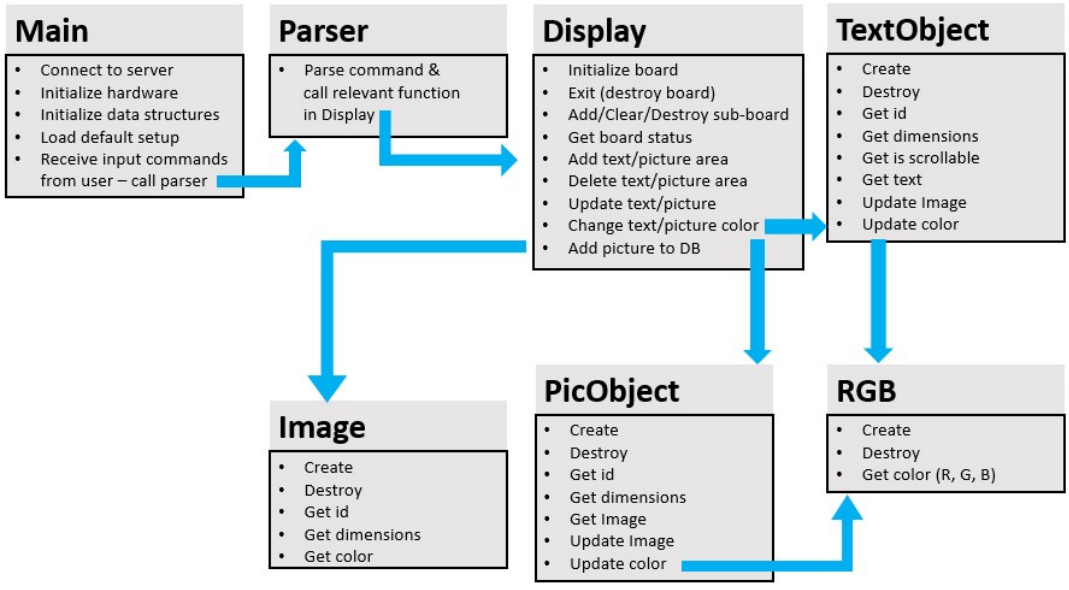
Our data structures include one main board which is allocated and initialized with the number or ports and matrices in the board, and how they are all connected. The board can contain as many sub-boards as the user chooses, typically it would be one for each parking lot and one for a general welcome

message. Each sub board is a square shaped part of the LED sign, with dimensions set according to the number of light bulbs in it.

Each sub-board can contain areas (an unlimited amount), either created to display text or a picture.



The code we have developed consists mainly of seven .c files, as well as matching header files, and a few other utility files. The relations between the source files and their general function can be seen in the illustration below:



A brief description about each file is as follows:

Main: Is responsible for initializing all the system hardware and connectivity, reads input commands from user, uses timer to perform scrolling.

Parser: Receives a string as an input command from user, parses it and calls the relevant function in Display

Display: Holds most of our data structures and logical mechanisms, performs board initialization/destruction, sub-board addition/deletion/clearing, area addition/deletion for either pictures or text, changing colors, etc. The commands are described in User Guide chapter.

TextObject/PicObject: Handles the object methods (create/destroy) as well as getting its attributes such as dimensions, color, etc. It refers to a text area or a picture area in user's terminology.

RGB: Since the FPGA prints colors according to RGB inputs, this basic data structure file includes create/destroy/get attributes (color) and is used by both TextObject file and PicObject.

Image: Printing pictures is only possible once the pictures are stored in the DB, this file handles the data structure of an object in this DB – create/destroy/get attributes (dimensions, color, etc).

Adding Features

The following are instructions to continue developing the system, and support changing and/or adding new features to it. In order to do so successfully and avoid bugs, we recommend first identifying in which source file the change should be performed:

* Main file: for changes in hardware parameters / communication / Initialization setup
* Display: for adding/changing user commands
* Object file: for changes in data structures behavior
* Image: for changes regarding image database

In case a new user command is desired to be added to Display file (and Display header file!), the developer should also add parsing for it in the Parser file (no need to change parser header) as follows:

1) Add a relevant case in "translateCommand" function according to the input name of the command

2) Add a matching case in "parseMessage" function, parsing the input parameters of the command and calling it from Display file.
Note: for your convenience, there is an internal function "get_numerical_input" created to extract integer inputs from a string command (returns the extracted value), and "get_array_input" created to extract array parameters in an unknown length (returns the length).
The main mechanism for both is the same: one of the inputs is an integer named "len", and one is its address. It refers to the length of the input extracted, meaning that we start looking after the previous extracted parameter and return where to start to look for the next one. The use of its pointer allows us to transfer information between function calling.
Note that array inputs are identified by "{" and "}" with commas between them.

User Guide

General Conventions

An operator who wishes to control the LED sign should use the following command set to display any desired text strings or picture. Firstly, we recommend going over the following conventions that apply to each command that is being sent:

- When the program starts, it initializes itself with default settings. That includes board initialization as well as creating some basic data structures (sub-boards) containing text/picture objects as described in appendix || on page 24. Those are the only commands that are called automatically. The rest should be manually invoked.
- Only after the operator receives the message "What would you like to do next?" – then it is safe to type in a new command.
- After each command the operator will receive a message stating whether the operation has been successful or failed. Note that if "Init" command has failed then the system should be rebooted. A notice will be displayed if so.
- The commands are called similarly to function calling: the name of the command is first (words separated by "_") and the input parameters are inside round brackets – "(" ")", separated by commas. If one of the parameters is a list, it should begin with "{", end with "}" and the list objects should be separated by ";".
- Whitespaces, tabs and line feeds between input parameters will make the command invalid. Please write the command string with only "," "{" "}" "(" ")" as delimiters. In this document we separated the parameters to new lines just in order to make the information clearer.
- Pay attention to lowercase/uppercase in command names. The Parser is case-sensitive.

Commands

Initialization

When the program starts, memory is allocated for a "Board" which is the main data structure. The allocated memory relies on the number of ports, number of matrices in each port (could be different for each port), and in case some matrices are connected in to the port rotated (rotated right 90° – "R", left 270° – "L", down 180°– "D", the default is up 0°– "U") it should be noted here.
The structure of rotation data is:
{index_row1,index_col1,direction1;index_row2,index_col2,direction2…}.
This command has 2 different types of syntax: Default mode (to load default settings – no need to insert any input parameters) & regular mode.

Syntax

Init(default)
Init(num_ports,array_of_matrices_in_each_port,list_of_rotated_matrices)

Examples

Init(default)
Init(4,{8,8,8,8},{})
Init(4,{8,8,8,8},{2,3,L;2,4,R;4,4,D})

Exit

When the operator wishes to close the program, clear the sign and free all allocated memory – she should type in "Exit" and it should destroy all data structures. This command receives no more inputs.

Syntax

Exit

Get Status

This command gets and displays to operator all the information about the current status of the board, including: Number of ports, number of matrices in each port, directions of matrices, positions & locations of sub-board and text/picture objects. It receives no input.

Syntax

Get_status

Flip Board

The user can use this command to flip the entire board and everything that is printed on it – rightwards or downwards. This command receives no inputs.

Syntax

Flip_right
Flip_down

Add Sub-Board

The main board is typically divided into sub-boards, each containing text and/or picture objects related to a certain database. After initialization the operator should define the location and size of each sub-board, before placing text/picture objects in them.

Syntax

Add_sub_board(ID,position_x,position_y,width,length)

Examples

Add_sub_board(1,10,32,50,50)
Add_sub_board(20,0,0,100,150)

Clear Sub-Board

This command deletes all the objects in the sub-board (but not the sub-board itself). The only input parameter it receives is the sub-board ID.

Note that when the operator wishes to exit the program she doesn't need to call this function manually for each sub-board, they are deleted automatically when "Exit" is called.

Syntax

Clear_sub_board(ID)

Delete Sub-Board

This command deletes the sub-board itself from the main board. After this command objects can no longer be added to this sub-board. The only input parameter it receives is the sub-board ID.

Syntax

Delete_sub_board(ID)

Add Text Area

Once the board is initialized and the relevant sub-board has been added, text area can be placed in it. Note that this command should be called once during initialization process, and to enter the actual text – Insert Text function should be called. Each sub-board can hold several text areas, therefore each text area has its own ID. If the size of the text area is too small for the planned text to appear in it, it can be scrollable (a boolean input parameter states so – 1 for scrollable, 0 for static). In addition, the color of the text (which can be changed via the command Change Color) should be received as input as well, divided into r,g,b data.

Syntax

Add_text_area(sub_board_ID,area_ID,position_x,position_y,width,length,
color_r,color_g,color_b,scrollable)

Examples

Add_text_area(1,10,32,50,50,100,67,43,52,1)
Add_text_area(2,0,32,50,50,0,0,200,0,0)

Insert Text

After a text area has been added to a sub-board, text can be filled in it. The operator should send the containing sub-board ID, the text area ID and the string itself. Since the platform does not support Hebrew, we created an ASCII-like table containing the values the operator should enter according to the desired string (appendix ||) in order to display it.

Syntax

Insert_text(sub_board_ID,area_ID,text_string,text_string_length)

Examples

Insert_text (10,1,{42,37,51,51,57,96,72,57,63,51,35}) – "Hello world"
Insert_text(10,2,{ 8,16,10,5}) – "חניה"

Change Text Color

This command allows changing the color of text in a text area (Remember RGB values are between 0-255).

Syntax

Change_text_color(sub_board_ID,area_ID,color_r,color_g,color_b)

Examples

Change_text_color (10,1,100,100,100)
Change_text_color (10,2,255,255,255)

Add Picture Area

Similarly to adding text areas, once the board is initialized and the relevant sub-board has been added, a picture area can be placed in it. The pictures that are available to print on the LED sign are stored in a small database. Some pictures (e.g. arrows) can be printed in different colors. Therefore, there is an input parameter stating whether the color of the picture is controllable or not, and there are input fields for RGB data in case it is (in other cases those fields can all be 0).

Syntax

Add_picture_area(sub_board_ID,area_ID,position_x,position_y,width,length,
Is_color_controllable,color_r,color_g,color_b)

Examples

Add_picture_area(1,10,32,50,50,100,1,255,255,255)
Add_picture_area(0,1,0,0,50,100,0,0,0,0)

Add Picture to Database

In order to print a picture, first it should be added to our database. Initially, the database is already storing the following items:
{Left arrow, Right arrow, Up arrow, Down arrow, smiley face, EE logo}
When left arrow index number is 0, and EE logo is 5.
When adding a picture to the database it should receive a new index (that is not in use by the others), so when the operator wishes to print it – she should make sure she is giving it a valid index (otherwise an error message will be displayed).
Inserting the picture itself is performed by writing the RGB data as a list: r values separated by commas, then g/b.

Syntax

Add_picture_to_db(picture_index, width,length,r_data,g_data,b_data)

Examples

Add_picture_to_db(6,2,3,{0,0,0,0,0,0},{0,0,0,0,0,0},{200,200,200,200,200,200})
Add_picture_to_db(7,2,2,{5,5,0,0},{0,0,8,0},{10,11,12,10})

Insert Picture

Similarly to inserting an actual text string to print after placing a text area inside a sub-board, once a picture area is placed and the relevant picture has been added to the database, it can be sent to print.
The operator should send the containing sub-board ID, the picture area ID, and the index of the picture in the database.

Syntax
Insert_picture(sub_board_ID,area_ID,index)
Examples
Insert_picture (10,1,11)
Insert_picture(2,3,4)

Change Picture Color

This command allows changing the color of a picture in a picture area (Remember RGB values are between 0-255), for a changeable color picture (e.g. arrows).
Syntax
Change_picture_color(sub_board_ID,picture_area_ID,color_r,color_g,color_b)
Example
Change_picture_color (10,1,100,100,100)
Change_picture_color (0,5,255,255,255)

Delete Area

As described above, each sub-board should contain text/picture objects which is basically an area designated to show text or picture, and it has its own ID. It can be deleted, and its allocated area in the sub-board will be free.
Syntax
Delete_area(sub_board_ID, object_ID)
Examples
Delete_area(1,1)
Delete_area(2,14)

Draw frame

Mainly for debug purposes, a frame around the board can be displayed, thus ensuring the initialization has been successful and corresponds to the borders of the LED sign. The frame is printed in a selected color given by the user.
Syntax
Draw_frame(color_r,color_g,color_b)
Example
Draw_frame(255,255,255)

Test Running Pixel

Mainly for debug purposes, a running pixel across the entire board can be displayed, thus ensuring the initialization has been successful and corresponds to the dimensions and connectivity of the LED sign. The running pixel is printed in a selected color given by the user.

Syntax

Test_running_pixel(color_r,color_g,color_b)

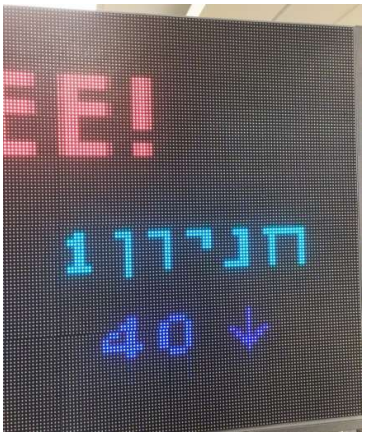
Example

Test_running_pixel (200,255,0)

Demo Example

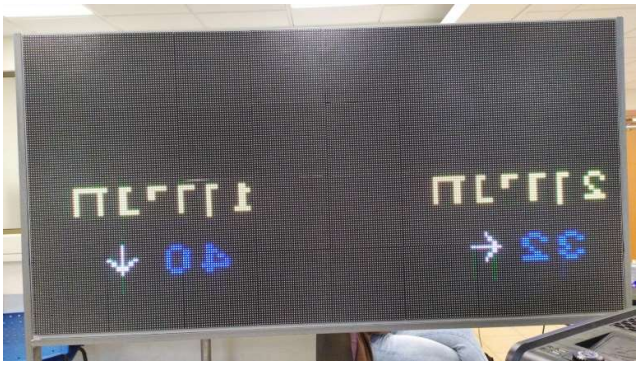
Our system supports Hebrew and English, displays numbers and special characters (see appendix || for details), and also displays images from a pre-defined database (e.g. arrows):

Supports color change of text/picture:



It also supports changing text, deleting sub-boards and creating different ones (in different locations across the board):

Flip right example:



Conclusions

In this project we had the opportunity to create a full functioning FPGA based controller application. We designed a modular software system from user requirements to a complete working and tested module.

We acquired comprehensive knowledge about integrating a software system with hardware components, structured programming and creating our own testing methods.

The application runs and displays information on the LED sign successfully. We have successfully added all the features requested, and the sign is ready to be in use. Creating sub-displays, adding text/picture objects, displaying scrollable text, rotating display according to physical connections of the boards and more – are all supported and been tested.

We have elaborated in this document how to add new features to the sign, so it can be further developed.

We would like to thank Mr. Oz Shmueli for the guidance and support, and VLSI, PSL, SSDL labs at EE department for the opportunity to take part in this project.

Appendix I: Initial Setup

For user convenience, there exists a setup configuration for the board which automatically gets uploaded to the board when the program starts running. The setup includes a text area at the top of the board with a scrolling welcome message, and two sub boards, each containing a small headline (parking lot name), a number (of available parking spaces in the parking lot) and an arrow directing to the relevant parking lot, for each of the parking lot floors (א & ב).

It has been carefully designed to display the information clearly and colorfully.

In addition, a colorful frame is drawn around the board to display control of the borders.

* The setup is currently hard-coded into the program, since there is no hard disk to read from on the ZYNQ.

The setup is created by calling using the following commands:

```
Init(default)

Add_sub_board(1,1,1,250,60)
Add_text_area(1,1,32,15,192,32,250,0,0,1)
Insert_text(1,1,{72,37,51,33,57,53,37,96,66,57,96,36,36,91})

// left side
Add_sub_board(2,5,64,90,63)
Add_text_area(2,1,5,64,80,16,255,255,0,0)
Insert_text(2,1,{81,17,6,10,16,8})
// floor A
Add_text_area(2,2,15,90,32,16,0,20,220,0)
Add_picture_area(2,3,52,90,16,16,1,255,255,255)
Add_text_area(2,4,68,90,16,16,255,255,255,0)
Insert_text(2,2,{82,81},2)
Insert_picture(2,3,0)
Insert_text(2,4,{1},1)
// floor B
Add_text_area(2,5,15,106,32,16,0,20,220,0)
Add_picture_area(2,6,52,106,16,16,1,255,255,255)
Add_text_area(2,7,68,106,16,16,255,255,255,0)
Insert_text(2,5,{85,86},2)
Insert_picture(2,6,2)
Insert_text(2,7,{2},1)

// right side
Add_sub_board(3,165,64,90,63)
Add_text_area(3,1,165,64,80,16,255,255,0,0)
Insert_text(3,1,{80,17,6,10,16,8})
```

```
// floor A
Add_text_area(3,2,170,90,32,16,0,20,220,0)
Add_picture_area(3,3,207,90,16,16,1,255,255,255)
Add_text_area(3,4,223,90,16,16,255,255,255,0)
Insert_text(3,2,{80,88},2)
Insert_picture(3,3,1)
Insert_text(3,4,{1},1)
// floor B
Add_text_area(3,5,170,106,32,16,0,20,220,0)
Add_picture_area(3,6,207,106,16,16,1,255,255,255)
Add_text_area(3,7,223,106,16,16,255,255,255,0)
Insert_text(3,5,{84,84},2)
Insert_picture(3,6,3)
Insert_text(3,7,{2},1)

// logo
Add_sub_board(4,110,80,30,28)
Add_picture_area(4,1,110,80,30,28,0,0,0,0)
Insert_picture(4,1,5)

Draw_frame(255,255,255)
```

And the printed result:

Appendix II: CII Table

The following table describes the matching integer values for supported symbols that can be displayed. This is relevant to the "Insert Text" command described on page 15.

Symbol	Integer
א	01
ב	02
ג	03
ד	04
ה	05
ו	06
ז	07
ח	08
ט	09
י	10
כ	11
ך	12
ל	13
מ	14
ם	15
נ	16
ן	17
ס	18
ע	19
פ	20
ף	21
צ	22
ץ	23
ק	24
ר	25
ש	26
ת	27

Symbol	Integer
A	28
A	29
B	30
B	31
C	32
C	33
D	34
d	35
E	36
e	37
F	38
f	39
G	40
g	41
H	42
h	43
I	44
i	45
J	46
j	47
K	48
k	49
L	50
l	51
M	52
m	53
N	54

Symbol	Integer
n	55
O	56
o	57
P	58
p	59
Q	60
q	61
R	62
r	63
S	64
s	65
T	66
t	67
U	68
u	69
V	70
v	71
W	72
w	73
X	74
x	75
Y	76
y	77
Z	78
z	79
1	80
2	81

Symbol	Integer
3	82
4	83
5	84
6	85
7	86
8	87
9	88
0	89
?	90
!	91
.	92
,	93
:	94
-	95
space	96

References

[1] *"Zynq-7000 SoC Data Sheet: Overview" Data Specification*, Xilinx, July 2018

[2] *"Parallella Virtual Memory" Project Book*, Mohamed Khateb and Banan Osman, December 2018