Flipdots Maestro

communication protocol

Introduction

FlipDots Maestro Driver is destined to communication with electromagnetic flip-dots display from AlfaZeta. It is responsible for a direct communication with a layout of displays that are in a given configuration. It introduces another layer of abstraction that facilitates creating FlipApps in various programming languages.
Programmer doesn't have to bother about panels arrangement and can treat FlipDots display as a hole.

Communication

The drawing application communicates with the driver via TCP sockets, thus FlipApps can be written in most programming languages, including Java, Python, C/C++ and more.
A Java library that implements the FlipDots Maestro Driver protocol is also available, and described in more detail in chapter *Flipdot-driver-lib Java library*.

Protocol fundamentals

Driver protocol is a client-server one.
Drawing application is a client.
Driver is a server listening on port 44000.
Commands are sent in a format:

[opcode] [parameters...]

Available commands

Every command starts with an op-code, that is decoded as a one-byte, signed two's complement binary number. **The data is send in *Network Byte Order* (Big-endian).**
Table 1 lists all valid opcodes, with a short description for each one. After the table, an in-depth description of each command is given.

Opcode	Description
0x0	Display a light pixel at the given coordinates
0x1	Display a dark pixel at the given coordinates
0x2	Display a transparent pixel at the given coordinates
0x3	Display a pixel of a specific color, at the given coordinates
0x4	Display a frame
0x5	Send FlipApp's UID
0x6	Get display resolution

Table 1. - index of available commands

Display a light pixel at the given coordinates

Displays a light pixel at the coordinates (x, y). Point (0, 0) is located in the top-left corner of the screen.

0x0	x	y
opcode	*x coordinate*	*y coordinate*

where:
opcode - 1 byte signed, two's complement binary number
x coordinate - 4 byte signed, two's complement binary number
y coordinate - 4 byte signed, two's complement binary number

Display a dark pixel at the given coordinates

Display a dark pixel at the coordinates (x, y). Point (0, 0) is located in the top-left corner of the screen.

0x1	x	y
opcode	*x coordinate*	*y coordinate*

where:
opcode - 1 byte signed, two's complement binary number
x coordinate - 4 byte signed, two's complement binary number
y coordinate - 4 byte signed, two's complement binary number

Display a transparent pixel at the given coordinates

Display a transparent pixel at the coordinates (x, y). Point (0, 0) is located in the top-left corner of the screen. A transparent pixel will be the same color as the Drivers background color. In the case of multiple apps sharing a display, thus simultaneously sending commands to the driver, a transparent pixel may covered by another application.

0x2	x	y
opcode	*x coordinate*	*y coordinate*

where:
opcode - 1 byte signed, two’s complement binary number
x coordinate - 4 byte signed, two’s complement binary number
y coordinate - 4 byte signed, two’s complement binary number

Display a pixel of a specific color, at the given coordinates

Display a pixel with color encoded by the color code, at the coordinates (x, y). Point (0, 0) is located in the top-left corner of the screen.

0x3	color	x	y
opcode	*color code*	*x coordinate*	*y coordinate*

where:
opcode - 1 byte signed, two’s complement binary number
color code - 1 byte signed, two’s complement binary number (for valid color codes, see *Table 2.*)
x coordinate - 4 byte signed, two’s complement binary number
y coordinate - 4 byte signed, two’s complement binary number

Color code	Color
0x0	Dark
0x1	Light
0x2	Transparent - by default, the background color, defined by the Maestro Driver. It may be overwritten by another application.

Table 2. - available color codes

Display a frame

This command allows to send and display a whole frame at once. It is an efficient way to draw larger images at a fast rate. After the opcode, follows a byte stream, build out of pixel rows, written from left to right. Every byte contains a color code, for the corresponding pixel on the display. The available color codes can be found in *Table 2.* ***NOTE***: it is important, that the number of bytes in the stream, matches the number of pixels on the display, otherwise the Driver’s behaviour is undefined. The desired number should be calculated based on the resolution, which can be retrieved with the get display resolution command.

0x4	byte stream
opcode	*bitmap*

where:
opcode - 1 byte signed, two’s complement binary number
bitmap - a byte stream, in which each byte denotes the color of the corresponding pixel on the display. The stream is built from pixel rows written from left to right, starting at the top row.

Send FlipApp’s UID

Allows to identify the FlipApp by the driver. This command should be the first command executed by the FlipApp. All commands send before this command will be ignored. When the UID is invalid, the driver closed the connection. The correct UID should be read from the configuration file, provided by the manger module, that starts the FlippApp and prepares the driver for the incoming connection.

0x5	number of bytes	byte sequence
opcode	*UID length*	*UID*

where:
opcode - 1 byte signed, two’s complement binary number
UID length - 2 byte signed, two’s complement binary number, representing byte-length of the identifier (***NOTE:*** that this is the amount of bytes, not letters)
UID - A unique identifier in ‘Modified UTF-8’ format

Get display resolution

Retrieve the display resolution.

0x6
opcode

where:
opcode - 1 byte signed, two’s complement binary number

Response:

width	height
horizontal pixel count	*vertical pixel count*

where:
horizontal pixel count - 4 byte signed, two’s complement binary number
vertical pixel count - 4 byte signed, two’s complement binary number

Protocol summary

Cmd code	Description	Example
0x0	`whitePixel(x: int, y: int)` Put white pixel at given coordinates	(hex) 0x00 0x00 0x00 0x00 0x0c 0x00 0x00 0x00 0x05 effect: color pixel in column 12 and row 5 (counting from zero) on white color.
0x1	`blackPixel(x: int, y: int)` Put black pixel at given coordinates	(hex) 0x01 0x00 0x00 0x00 0x0c 0x00 0x00 0x00 0x05 effect: color pixel in column 12 and row 5 (counting from zero) on black color.
0x2	`transparentPixel(x: int, y: int)` Put transparent pixel at given coordinates	(hex) 0x02 0x00 0x00 0x00 0x0c 0x00 0x00 0x00 0x05 effect: color pixel in column 12 and row 5 (counting from zero) on transparent color.
0x3	`putPixel(color: byte, x: int, y: int)` Put pixel of given color at given coordinate. Pixel colors: 0x0 - black 0x1 - white 0x2 - transparent	(hex) 0x03 0x01 0x00 0x00 0x00 0x0c 0x00 0x00 0x00 0x05 effect: color pixel in column 12 and row 5 (counting from zero) on white color.
0x4	`drawFrame(frame: byte[])` Draw frame on a display. Pixel colors: 0x0 - black 0x1 - white 0x2 - transparent	(hex) 0x04 0x01 0x01 0x01 0x00 0x00 0x01 0x01 0x00 effect: WHITE | WHITE | WHITE | BLACK BLACK | WHITE | WHITE | BLACK
0x5	`sendUid(length: short, uid: char[])` Send uid of an application.	(hex) 0x04 0x00 0x09 0x61 0x6c 0x66 0x61 0x2d 0x7a 0x65 0x74 0x61 effect: sending uid “alfa-zeta” to driver.
0x6	`getResolution(void): width: int, height: int` Read resolution of a display given to the application.	(hex) 0x06 response: (hex) 0x00 0x00 0x00 0x38 0x00 0x00 0x00 0x0e meaning: resolution is 56 width and 14 height.

Flipdots-driver-lib Java library

The Flipdots-driver-lib is a dedicated Java library, that implements the Driver Protocol, and wraps the commands in simple, one method calls. The library also contains various helper functions, like converting and scaling images to properly sized, binary pixel map. The library can be found here: https://bitbucket.org/alfazeta/flipdotsjavadriverlib/src/master/

FlipApp structure and requirements

To further improve the ease of implementing and running FlipApps, the Maestro utility comes with a dedicated Manager module, that manages a repository of FlipApps. It is responsible for launching and communicating with running FlipApps using configuration files and standard input. To make FlipApps compliant with this environment, the FlippApp should implement the communication interface for the manager.
That communication interface consists of 3 elements:

- definition file
- configuration file
- standard input.

The following sections describe each of those elements.

Definition file

A ‘YAML’ file, that defines various FlipApp properties, that may be set by the manager, thus it should be provided by the FlipApp author. The file defines the name, version and list of variable properties a FlipApp has.
Each property is described by 6 mandatory parameters:

- name - the name of the property
- type - a string describing the type of the property (f.e. text, number)
- label - the label of the property
- description - the description of the property, that the manager will pass to its users
- value - the default value for this property (look at this as a hint for the manager)
- dynamic - a boolean value, indicating whether the property i dynaminc, that is - whether the FlipApp wants to listen for new values, passed through standard input, during execution.

To give a better understanding of the definition file structure, *Listing 1* shows an example file for a simple FlipApp, which displays text provided by the manager, on the FlipDot display. Based on this file, the manager will know, that it should provide the text to be displayed and the font size in the *definition file.* Since the properties are defined as ‘dynamic’, the manager knows, that it can change the value of those properties, during FlipApp execution, through the *standard input* of the app.

```
name: "Text" # FlipApp name visible in maestro gui
id: "com.flipdots.flipapps.text" # identifier of a FlipApp. Try to make it unique
version: "1.0" # version of a flip app
properties:
 - name: "text"
   type: "text"
   label: "Text"
   description: "Text to be displayed"
   value: "Hello"
   dynamic: true
 - name: "fontSize"
   type: "number"
   label: "Font size"
   description: "Size of a displayed text"
   value: 12
   dynamic: true
protocolVersion: "1.0" # communication protocol ver. Now only 1.0 is available
```

Listing 1 - definition file example

Configuration file

Based on the definition file, the manager module prepares a '.properties' file, that provides values of the variable properties for the FlipApp. If a property was declared as non-dynamic (dynamic: false) in the *definition file*, the value of it will only be passed through this file. To better understand the structure of the *configuration file, Listing 2* shows a simple example, compliant with the the requirements declared by the definition file from *Listing 1.*

```
uid=abcd
text=Long text\nwith second line
fontSize=9
```

Listing 2 - configuration file example

The file above, sets the initial value of the text to be displayed to "Long text\nwith second line" (Note: '\n' denotes a line-break), by assigning this value to the property with name 'text', and the initial font size to 9, by setting the property named 'fontSize'.

Note: Every FlipApp should, expect an unique identifier passed in this file through the 'uid' property, which will be used to identify itself to the driver. This is a default property, and it shouldn't be defined in the *definition file*. For the above example, the unique identifier is 'abcd'.

Standard input (dynamic properties)

Properties defined as dynamic (dynamic: true), may change during FlipApp execution. New property values are send to the standard input of the flipApp in the '.property' format, one property at a time. Thus, if the FlipApp defined any property as 'dynamic', it should be listening on the standard input, for new values of those properties. The new values will be provided in the '.properties' format, line by line.

FlipApp Java example

To better understand the structure of a FlipApp that uses the Maestro driver and manager modules, this chapter shows a simple FlipApp example. The app draws a checker pattern onto the display. It has one dynamic parameter, that allows to change the field size (described in pixels).
The example uses the FlipDots and FlipDotsFrame classes from the *Flipdots-driver-lib java library* to communicate with the Driver. The program takes the path of the '.properties' file as a command line argument.

def.yml

```
name: "Checker drawer"
id: "org.flipfan.checker"
version: "1.0"
properties:
 - name: "fieldSize"
   type: "number"
   label: "Field size"
   description: "Size of one checker field in pixels"
   value: 5
   dynamic: true
protocolVersion: "1.0"
```

Main.java

```
import java.io.IOException;

public class Main {
   public static void main(String[] args) {
       if (args.length != 1) {
           System.out.println("Wrong argument count");
           return;
       }

       try {
           FlipApp flipApp = new FlipApp("localhost", 44000);
           flipApp.start(args[0]);
       } catch (IOException e) {
           e.printStackTrace();
       }
   }
}
```

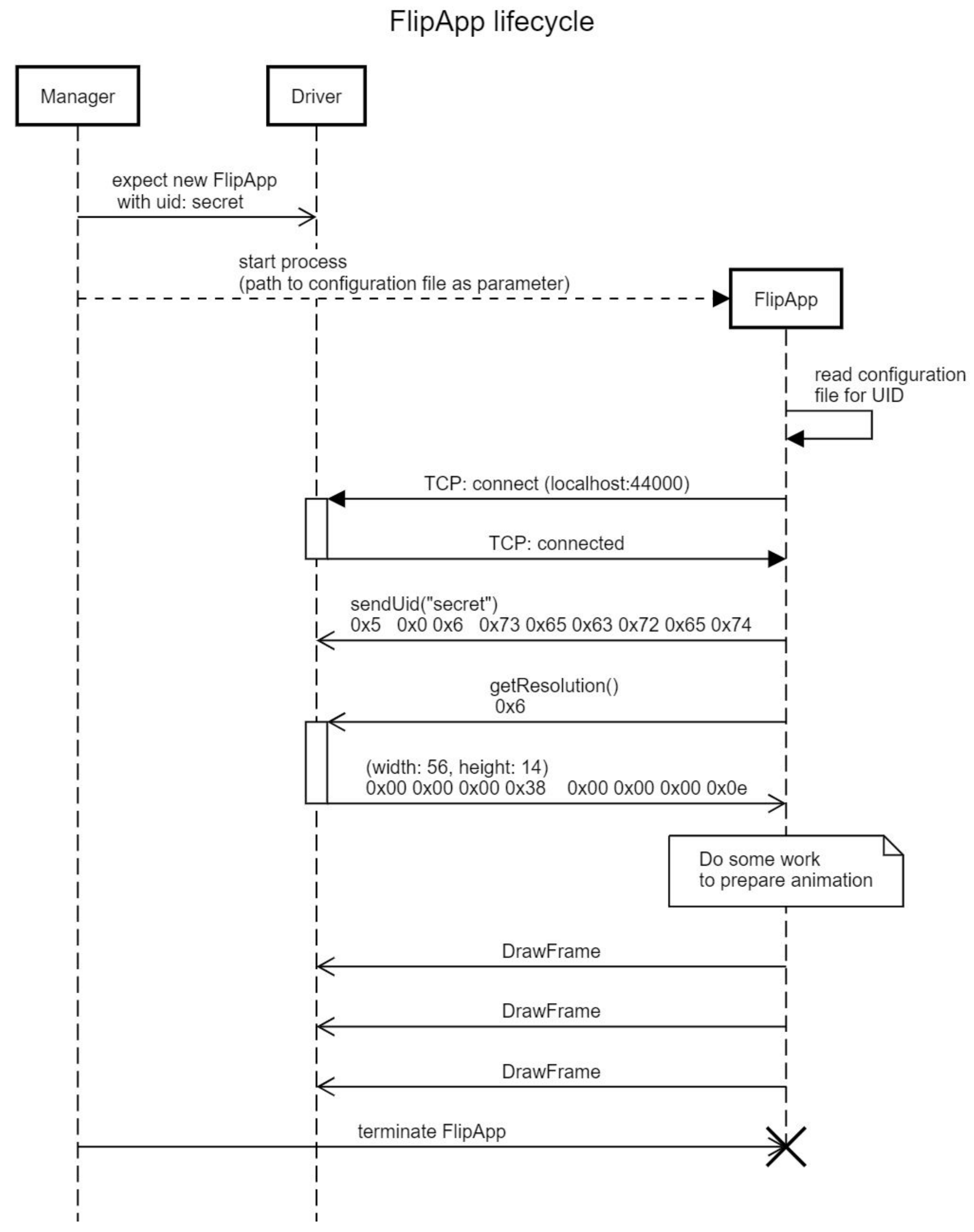
FlipApp.java

```
import com.flipDot.driver.lib.FlipDots;
import com.flipDot.driver.lib.util.FlipDotsFrame;

import java.io.*;
```

```java
import java.util.Properties;
import java.util.Scanner;
import java.util.Set;

public class FlipApp {
   private class FlipAppProperties {
       public int fieldSize;
       /* YOUR FLIPAPP PROPERTIES GO HERE */
   }
   private FlipAppProperties flipAppProperties = new FlipAppProperties();

   private FlipDots.Resolution resolution;
   private FlipDots flipDots;

   public FlipApp(String ipAddress, int port) throws IOException {
       flipDots = new FlipDots(ipAddress, port);
   }

   public void start(String propertyFilePath) throws IOException {
       InputStream inputStream = new FileInputStream(new File(propertyFilePath));
       Properties properties = new Properties();
       properties.load(inputStream);

       String flipAppUid = properties.getProperty("uid");
       flipDots.sendAppId(flipAppUid);
       resolution = flipDots.getResolution();

       /* <READ PROPERTIES>  */
       flipAppProperties.fieldSize =
           Integer.parseInt(properties.getProperty("fieldSize"));
       /* </READ PROPERTIES>  */

       /* <FLIPAPP STARTUP CODE> */
       drawChecker();
       /* </FLIPAPP STARTUP CODE> */

       mainLoop();
   }

   private void mainLoop() {
       Scanner scanner = new Scanner(System.in);
       while (scanner.hasNextLine()) {
           Properties dynamicProperties = new Properties();
           try {
               StringReader reader = new StringReader(scanner.nextLine());
               dynamicProperties.load(reader);

               Set<Object> setOfKeys = dynamicProperties.keySet();
               for (Object o : setOfKeys) {
                   updateProperty(o.toString(), dynamicProperties.get(o).toString());
               }

           } catch (IOException e) {
               e.printStackTrace();
           }
       }
   }
```

```java
     private void updateProperty(String key, String value) {
        switch (key) {

            case "fieldSize":
                flipAppProperties.fieldSize = Integer.parseInt(value);
                drawChecker();
                break;

                /* HANDLE YOUR PROPERTY CHANGES HERE */

            default:
                /* default behaviour */
                break;
        }
    }

    private void drawChecker() {
        try {
            FlipDotsFrame flipDotsFrame = new FlipDotsFrame(flipDots);
            int fieldSize = flipAppProperties.fieldSize;
            for (int i = 0; i < resolution.width; ++i) {
                for (int j = 0; j < resolution.height; ++j) {
                    if ((i / fieldSize % 2 == 0) ^ (j / fieldSize % 2 == 0))
                        flipDotsFrame.whitePixel(i, j);
                    else
                        flipDotsFrame.blackPixel(i, j);
                }
            }
            flipDots.drawFrame(flipDotsFrame);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

FlipApp life cycle diagram


FlipApp lifecycle
Manager
Driver
expect new FlipApp with uid: secret
start process (path to configuration file as parameter)
FlipApp
read configuration file for UID
TCP: connect (localhost:44000)
TCP: connected
sendUid("secret")
0x5 0x0 0x6 0x73 0x65 0x63 0x72 0x65 0x74
getResolution()
0x6
(width: 56, height: 14)
0x00 0x00 0x00 0x38 0x00 0x00 0x00 0x0e
Do some work to prepare animation
DrawFrame
DrawFrame
DrawFrame
terminate FlipApp