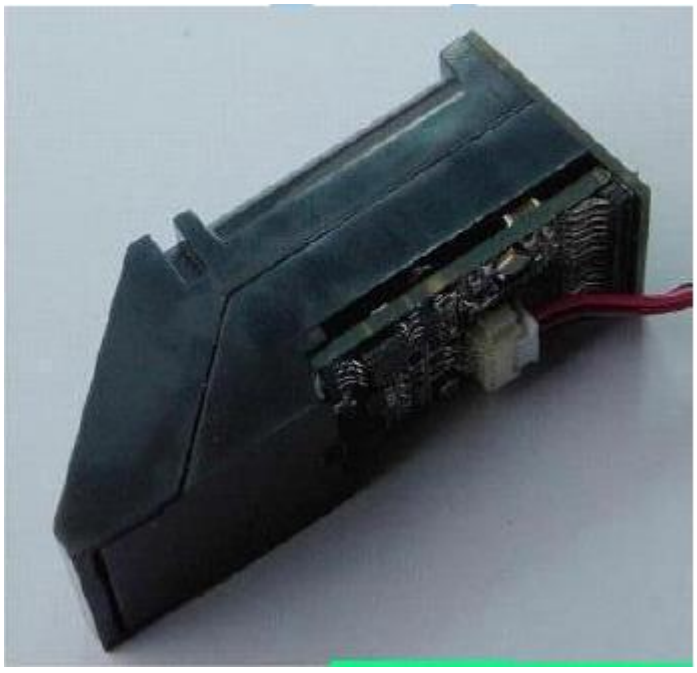
INTERFACING THE FPM10A WITH AN ARDUINO (by Devesh Kumar)

INTRODUCTION

The fingerprint sensor used here is the FPM10 fingerprint sensor module but most of the software and connections are compatible with the R305 and ZFM-20 modules. The module performs fingerprint enrolment, image processing, fingerprint matching and searching and template storage. It can perform 1:1 matching or 1:N matching. It employs the UART protocol to communicate with the host MCU. The default baud rate is usually 57600 bps though the module can support from 9600 to 115200 bps.

The module makes use of an image buffer and two 512-byte character file buffers, which are volatile, and non-volatile flash memory for storing fingerprint templates and permanent settings.



OPERATION

Communication is established at a specific baud rate. The default is 57600 bps.

Before any real commands can be sent to the module, there must first be a handshake between the host MCU (Arduino) and the module where the host

sends the pre-set 4-byte password to the module and the module acknowledges if the password is correct or not. If it is correct, the module goes into normal working condition. Else, it refuses to execute any further commands. The default password is 0. This value can be changed after the handshake is complete, if the user desires. The module also possesses a 4-byte address that must accompany each command. The default address is 0xFFFFFFFF and can also be changed. Changed settings (e.g. address and password) only take effect the next time the device is powered on.

To enroll a fingerprint, one must have the finger scanned twice, with an image generated each time and stored in the image buffer. Each generated image is converted to a *character file* (or feature file) and stored in one of the 2 character file buffers. Next, the image processor determines if the data from both buffers came from the same finger. If this is true, then a template is generated from the combined data in both buffers stored again in both buffers. Otherwise, the module returns an error packet and does not generate a template.

To finally store the template in non-volatile flash memory, one must specify a PageID in flash memory for storing the template. This ID will also enable the user access individual templates at any time. The module used here can store up to 1649 unique fingerprint templates (Template 0 – 1648). If this number is exceeded, an error occurs and the confirmation code corresponding to “Error writing to flash” is returned.

Note that each of the aforementioned steps must be initiated by the host i.e. the MCU must send commands to do everything from image generation to template storage. There are also commands to download images, character files or templates from the module to the MCU or to upload the same from the MCU to the module (e.g. for 1:1 matching). Individual templates can be deleted from the flash memory and the entire database can be deleted with one command.

1:1 matching stands for selecting 2 fingerprint templates from memory, placing them in the character file buffers and commanding the module to check if they match. 1:N matching refers to comparing one template against all the stored templates and looking for a match (N stands for number of stored templates).

HARDWARE

To communicate with the module, serial communication at 57600 baud (default) is required. When using an Arduino, messages to prompt the user for input or for debugging are sent through the hardware serial port and displayed on the serial monitor. If the Arduino Uno is being used, since it has only one hardware serial port, one must set up a SoftwareSerial port that will be used for module-Arduino communication.

For this project, a *SoftwareSerial* port was setup using pin 2 (RX) and pin 3 (TX) of the Uno. The module can be powered using 3.6 to 6 V. In this case, 5 V was used. The connections are as follows:

TX of module (Green wire) → RX of Arduino (Pin 2)

RX of module (Yellow or white wire) → TX of Arduino (Pin 3)

VCC of module (Red wire) → 5 V pin of Arduino

GND of module (Black wire) → GND of Arduino

You may solder leads to the module's wires to strengthen connections.

When you power up the Arduino, a red light should flash in the module briefly.

SOFTWARE

A library gotten from the Adafruit website and slightly modified is attached to this project. It contains functions to perform relevant tasks mentioned in the datasheet, which can be found here:

http://www.adafruit.com/datasheets/ZFM%20user%20manualV15.pdf.

Only one example will be explained here: the *enroll* example. It will describe how to setup the module, establish communications, send commands and receive responses from the module and also the different kinds of response that can be received and their meanings. For more detail, consult the datasheet or read the library source and header files.

Data package format

Header	Adder	Package identifier	Package length	Package content (instuction/data/Parameter)	Checksum

Every packet transferred between the Arduino and the module (data, commands, etc.) follows the above format. The library already implements this structure.

Header: 0xEF01 (a constant) **Address**: 0xFFFFFFFF (default)

Package identifier: 1, 2, 7, or 8 (Data, command, acknowledgement, End-of-data respectively). Indicates the kind of packet being received/sent.

Package length: Length of package content + length of checksum (in bytes)

Package content: Data packet, instruction code, or confirmation code or a combination of them

Checksum: 2-byte sum of package identifier, package length and package contents. Used for error checking.

For acknowledgement packets from the module, the “Package content” in the format above usually contains a 1-byte confirmation code that indicates success or otherwise of a command from the Arduino. Every function in the library called by your program returns a confirmation code which can then be tested against known confirmation codes to figure out if the command completed successfully or the exact error encountered, if one occurred.

The program below requests an ID from the user, scans a fingerprint, processes and then stores it with the ID. It does this repeatedly in the *loop()* function.

```
#include <Adafruit_Fingerprint.h>
#include <SoftwareSerial.h>

int getFingerprintEnroll(int id);


// pin #2 is IN from sensor (GREEN wire)
// pin #3 is OUT from arduino  (WHITE wire)
SoftwareSerial mySerial(2, 3);

Adafruit_Fingerprint finger = Adafruit_Fingerprint(&mySerial);

void setup()
{
  Serial.begin(9600);
  Serial.println("fingertest");

  // set the data rate for the sensor serial port
  finger.begin(57600);

  if (finger.verifyPassword()) {
    Serial.println("Found fingerprint sensor!");
  } else {
    Serial.println("Did not find fingerprint sensor :(");
    while (1);
  }
}
```

The header files for the fingerprint library and the *SoftwareSerial* are first included. The next line of code shows a function prototype and indicates that there is a user-defined function present in the program. A *SoftwareSerial* object is created with parameters as pins 2 & 3 (RX and TX). A fingerprint object (named *finger* here*)* is created and its constructor's sole parameter is the address of the *SoftwareSerial* port which will be used by the module.

The *setup()* function initializes the serial port which will be used to display messages and also for debugging, if necessary. The baud rate of the serial port need not match that of the *SoftwareSerial* object if the serial port is only being used for messages/debugging. However, if you transfer large packets of data to/from the MCU using hardware serial, it's recommended that they be at least the same. The *finger* object is initialized with a baud rate of 57600. The handshake protocol then commences: the *verifyPassword()* method of the object is called and if the function returns TRUE then a message is printed to the serial port indicating success and the module is ready to receive commands. Else, the other message is printed and the program remains in the *while* loop till Arduino power-off or reset and doesn't proceed to the *loop()* function.

The *loop()* function begins by obtaining an ID to be used to store the fingerprint. The ID is entered using the Serial monitor and the while loop below converts the ASCII characters received to actual digits to be stored in the integer ID. The loop() function then calls the user-defined function *getFingerprintEnroll()* whose sole argument is the ID.

```
void loop()                     // run over and over again
{
  Serial.println("Type in the ID # you want to save this finger as...");
  int id = 0;
  while (true) {
    while (! Serial.available());
    char c = Serial.read();
    if (! isdigit(c)) break;
    id *= 10;
    id += c - '0';
  }
  Serial.print("Enrolling ID #");
  Serial.println(id);

  getFingerprintEnroll(id);
}

int getFingerprintEnroll(int id) {
  int p = -1;
  Serial.println("Waiting for valid finger to enroll");
  while (p != FINGERPRINT_OK) {
    p = finger.getImage();
    switch (p) {
    case FINGERPRINT_OK:
      Serial.println("Image taken");
      break;
    case FINGERPRINT_NOFINGER:
      Serial.println(".");
      break;
    case FINGERPRINT_PACKETRECIEVEERR:
      Serial.println("Communication error");
      break;
    case FINGERPRINT_IMAGEFAIL:
      Serial.println("Imaging error");
      break;
    default:
      Serial.println("Unknown error");
      break;
    }
  }
```

The *getFingerprintEnroll()* function handles the actual fingerprint enrolment. The variable *p* is initialized to -1 and is used to store the return values of methods called. The *getImage()* method scans a fingerprint and generates an image that is stored in the module image buffer. It returns 0x00 if the command is successful, else it returns one of several other confirmation codes. The *switch* statement handles every possible confirmation code that can be returned by the method and prints the appropriate message.

**Throughout the program, the variables in upper case represent constants that have been defined in the library. These constants are the possible confirmation codes returned by the module in response to commands. **

The while loop in the function runs until the module successfully reads a fingerprint. If an error is encountered, a message is displayed according to the *case* statement where it falls, after which the switch statement exits and the loop restarts.

```
p = finger.image2Tz(1);
switch (p) {
  case FINGERPRINT_OK:
    Serial.println("Image converted");
    break;
  case FINGERPRINT_IMAGEMESS:
    Serial.println("Image too messy");
    return p;
  case FINGERPRINT_PACKETRECIEVEERR:
    Serial.println("Communication error");
    return p;
  case FINGERPRINT_FEATUREFAIL:
    Serial.println("Could not find fingerprint features");
    return p;
  case FINGERPRINT_INVALIDIMAGE:
    Serial.println("Could not find fingerprint features");
    return p;
  default:
    Serial.println("Unknown error");
    return p;
}
```

This part converts the image in the image buffer to a character file which is stored in Buffer 1 of the module. If the command is successful, the *switch* statement is exited and the program proceeds to the next part of the function. Else, the error code *p* is returned to the *loop()* function which then starts afresh.

```
Serial.println("Remove finger");
delay(2000);
p = 0;
while (p != FINGERPRINT_NOFINGER) {
  p = finger.getImage();
}

p = -1;
Serial.println("Place same finger again");
while (p != FINGERPRINT_OK) {
  p = finger.getImage();
  switch (p) {
  case FINGERPRINT_OK:
    Serial.println("Image taken");
    break;
  case FINGERPRINT_NOFINGER:
    Serial.print(".");
    break;
  case FINGERPRINT_PACKETRECIEVEERR:
    Serial.println("Communication error");
    break;
  case FINGERPRINT_IMAGEFAIL:
    Serial.println("Imaging error");
    break;
  default:
    Serial.println("Unknown error");
    break;
  }
}
```

This part waits for the user to remove the finger being scanned; it stalls in the *while* loop until it detects no finger. Then the finger is scanned again and the *while* loop is broken only when an image has been successfully generated.

The section below converts the image to a character file to be stored in Buffer 2 as was done previously.

**A more complete datasheet (but not as clear as the first):

https://sicherheitskritisch.de/files/specifications-2.0-en.pdf

```
p = finger.image2Tz(2);
switch (p) {
  case FINGERPRINT_OK:
    Serial.println("Image converted");
    break;
  case FINGERPRINT_IMAGEMESS:
    Serial.println("Image too messy");
    return p;
  case FINGERPRINT_PACKETRECIEVEERR:
    Serial.println("Communication error");
    return p;
  case FINGERPRINT_FEATUREFAIL:
    Serial.println("Could not find fingerprint features");
    return p;
  case FINGERPRINT_INVALIDIMAGE:
    Serial.println("Could not find fingerprint features");
    return p;
  default:
    Serial.println("Unknown error");
    return p;
}
  p = finger.createModel();
  if (p == FINGERPRINT_OK) {
    Serial.println("Prints matched!");
  } else if (p == FINGERPRINT_PACKETRECIEVEERR) {
    Serial.println("Communication error");
    return p;
  } else if (p == FINGERPRINT_ENROLLMISMATCH) {
    Serial.println("Fingerprints did not match");
    return p;
  } else {
    Serial.println("Unknown error");
    return p;
  }

  Serial.print("ID "); Serial.println(id);
  p = finger.storeModel(id);
  if (p == FINGERPRINT_OK) {
    Serial.println("Stored!");
  } else if (p == FINGERPRINT_PACKETRECIEVEERR) {
    Serial.println("Communication error");
    return p;
  } else if (p == FINGERPRINT_BADLOCATION) {
    Serial.println("Could not store in that location");
    return p;
  } else if (p == FINGERPRINT_FLASHERR) {
    Serial.println("Error writing to flash");
    return p;
  } else {
    Serial.println("Unknown error");
    return p;
  }
}
```

Next, a model/template is generated using the combined data in Buffers 1 & 2. The created template is stored in Buffers 1 & 2. Again, any error encountered instantly terminates the function.

Finally, the template is stored using the *storeModel()* method which takes the input ID as an argument. If successful, the module stores the template at the destination PageID and exits the function. The *loop()* function reaches the end and restarts as usual and the whole process is repeated indefinitely.

Sample results:

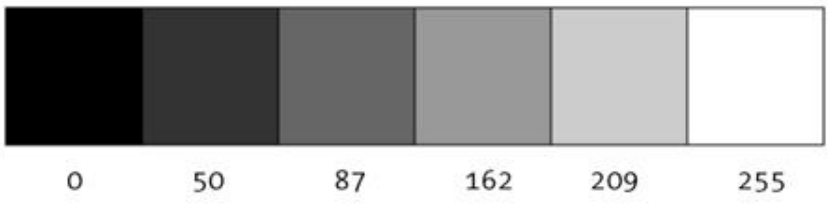
```
fingertest
Found fingerprint sensor!
Type in the ID # you want to save this finger as...
Enrolling ID #482
Waiting for valid finger to enroll
.
.
.
.
.
.
.
.
.
.
.
Image taken
Image converted
Remove finger
Place same finger again
......Image taken
Image converted
Prints matched!
ID 482
Stored!
Type in the ID # you want to save this finger as...
```

The remaining examples show how to implement fingerprint searching, deletion of templates, extract fingerprint images, empty the fingerprint database and set system parameters (like baud rate and packet size).

FINGERPRINT IMAGE EXTRACTION

The modified library contains a method for extracting fingerprint images from the module's image buffer. If the method is called, it issues the appropriate command to the module (check datasheet) and receives an acknowledgement packet if successful. After this, the module begins to transmit the data packets which constitute the fingerprint image.

The image is an 8-bit 256x288 grayscale bitmap image. This means that every pixel is represented using an 8-bit value (a byte) between 0 – 255 where 0 represents 'black' and 255 represents 'white'. In between these values are various shades of gray.



The image is 256 pixels in width and 288 pixels in height, giving a total of 256*288 = 73728 pixels or bytes. The module must send all 73728 bytes (each byte representing each pixel) in order for the host computer/MCU to re-assemble the image. In order to increase transfer speed, the module transfers the 2 pixels per byte instead. That is, normally each pixel = 1 byte, but instead the module takes the high nibble (4 bits) of one pixel and the low nibble of an adjacent pixel to create an 8-bit value that contains the data for both pixels. This way, the module transmits only 73728/2 = 36864 bytes to the MCU at the cost of losing some of the image data.

E.g. PIXEL 1: **1010**0001 PIXEL 2: 0100**0110**
Combined data: **10100110** (contains info about the 2 pixels)

The host MCU, upon receiving the data, must expand each byte to get the approximate 8-bit representation of each pixel. Finally, a bitmap image file (.bmp) can then be created using the resulting values.

The *sendImage()* method uses a '\t' character to indicate the start of image streaming. The method makes no attempt to save the image data to a buffer in MCU memory but expects that a separate program (running on a PC, for example) is listening to the serial port where it prints each byte. This is done to

save space since at least 74kB of data memory (which is not available on the Uno, anyway) would be required to save the whole image to the Arduino.

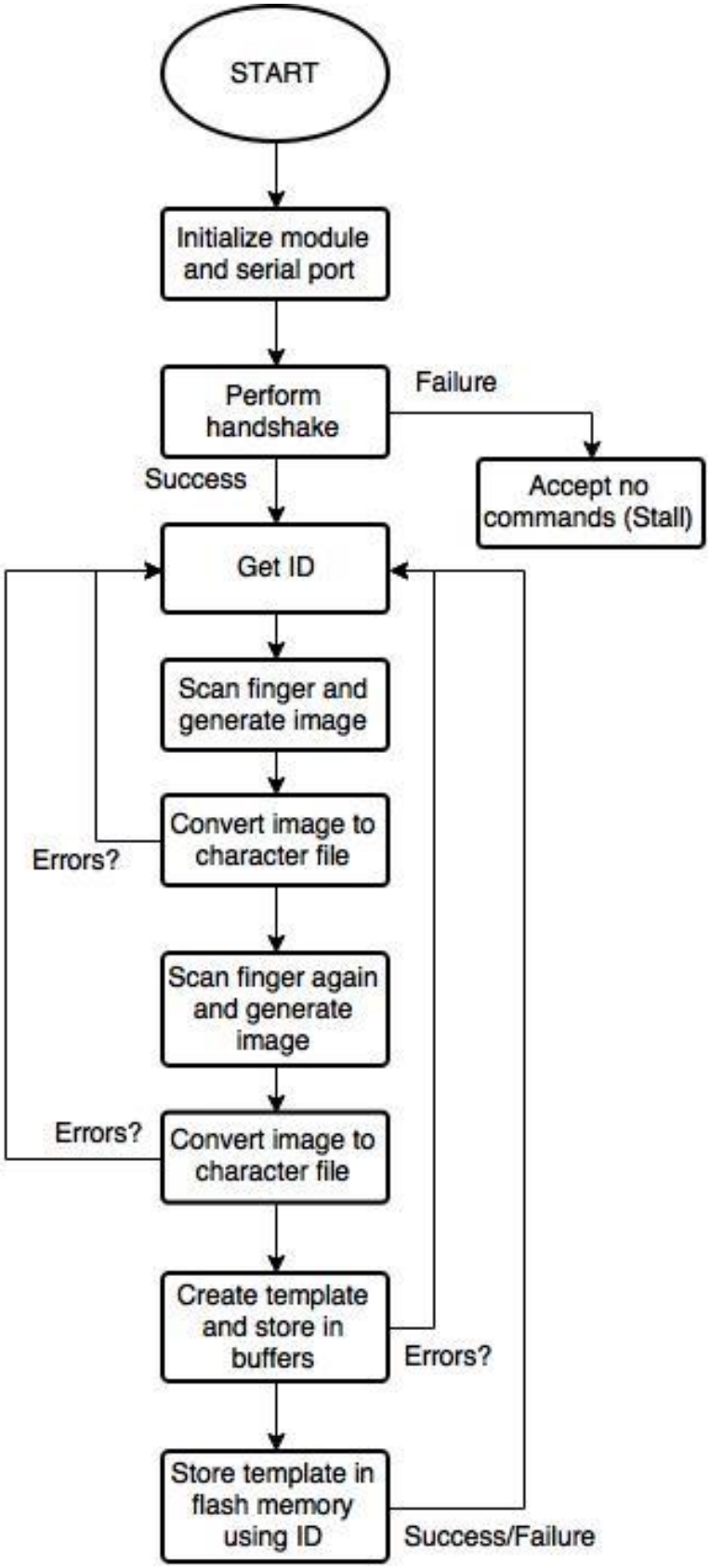
The external program that receives the data counts the bytes received and terminates after it receives the 36864th byte. I wrote some Python code (included in the library) that downloads the image, processes and saves it to a file (chosen by the user) on the host PC. The recommended baud rate for image streaming is 57600 baud. Using higher values (like 115200 baud) may cause errors due to the unreliability of *SoftwareSerial* at such speed.

SYSTEM PARAMETERS

Baud rate, security level and packet length can be changed from their default values using examples shown in the library. Each command should be run only once (i.e. in the *setup()* function) and the new parameter value is saved to the module's flash memory. The new settings take effect immediately, contrary to what the datasheet says about a power cycle. Read the example's comments and/or the datasheet to get the possible values for each parameter.

CHALLENGES

- I got unreliable results when using high baud rates (86400 and above). Use 57600 baud ideally.



FLOWCHART: ENROLL FUNCTION