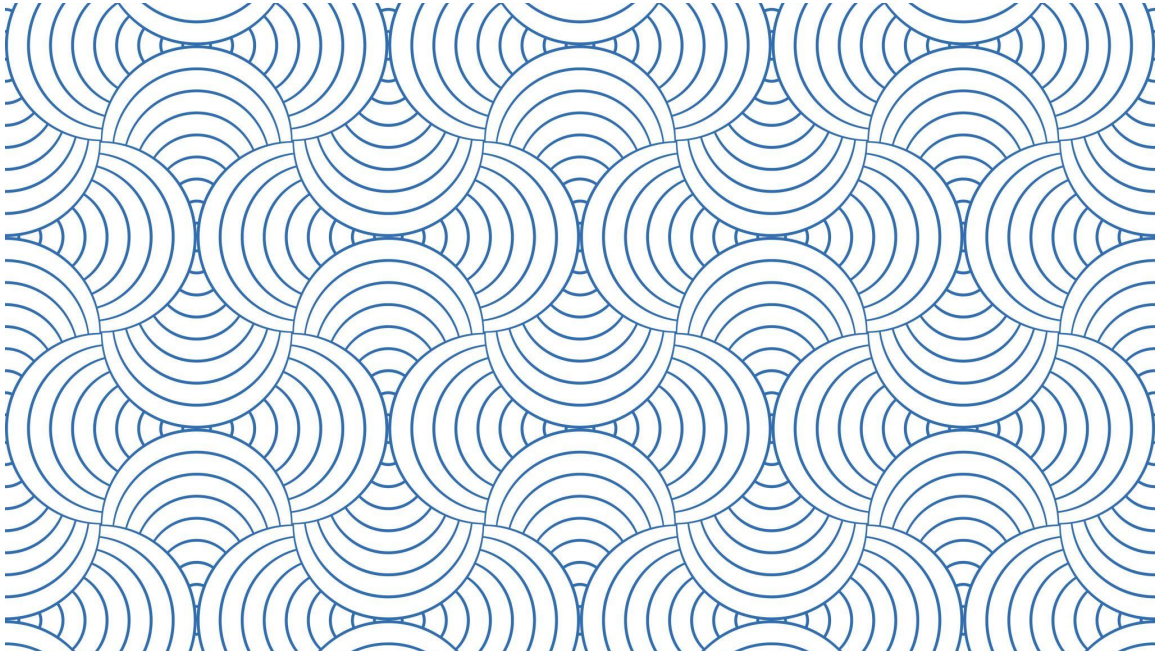
GM2 1st Interim Presentation: Lao Bytecode

Oisín Conlon

Diya Thomas

Oliver Lee



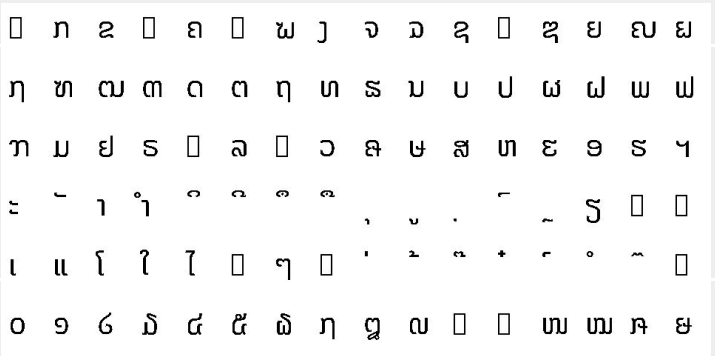
Reviewing our proposal…

01 **Continuing MakerBox's current solution**

Flash	SRAM	EEPROM
32kB	2kB	1kB

1,027 KB

02 **Rule-Based Rendering**



03 **Pre-processing of the text**

Our current solution!

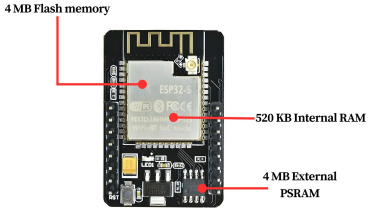
Our current solution

- Considered workarounds:
- SD card
- ESP32 board

Not as broadly applicable to diffe types of projects!

- Storage problems
- Troubleshoot glyph placement logic

Spoke to Ken, Bill and Oui
Difficult to prototype in a few days



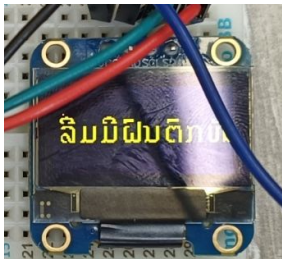
ESP32 board

Current use case - display predetermined messages to farmers:

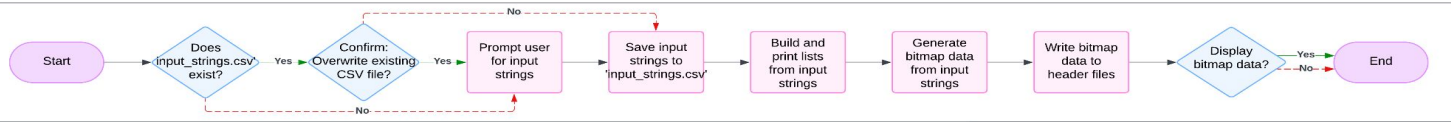
eg . water the field, big storm coming

CSV file of phrases → **Unique characters + their order** → **Generate bitmaps** → **Display!**

ສະພາບອາກາດໃນຊ່ວງນີ້ໃນເຂດທີ
ທ່ານຢູ່ແມ່ນຈະເລີ່ມມີຝົນຕົກໜັກ



Processing Inputs




```
An existing input_strings.csv was found. Overwrite it? (y/N): y
Overwriting input strings...
Enter strings one by one. Press Enter on an empty line to finish.
Enter string: héllo world
Enter string: Thís is a sìmple test!
Enter string:
Identifying unique characters...
Character List:
[h é l o   w r d T î s i a ì m p e t !]

Index List:
String 0: [0, 1, 2, 2, 3, 4, 5, 3, 6, 2, 7]
String 1: [8, 0, 9, 10, 4, 11, 10, 4, 12, 4, 10, 13, 14, 15, 2, 16, 4, 17, 16, 10, 17, 18]
```

```
#ifndef PHRASES_TO_DISPLAY_H
#define PHRASES_TO_DISPLAY_H

const uint8_t all_phrases[] = {
    0, 1, 2, 2, 3, 4, 5, 3, 6, 2, 7,    // phrase 1
    8, 0, 9, 10, 4, 11, 10, 4, 12, 4, 10, 13, 14, 15, 2, 16, 4, 17, 16, 10, 17, 18,    // phrase 2
};

const uint8_t phrase_starts[] = {0, 11};     // starting index of each phrase
const uint8_t phrase_lengths[] = {11, 22};    // length of each phrase
const uint8_t num_phrases = 2;
```

write_index_list_to_header(): → [phrases_to_display.h](#)
→ index_list concatenated
→ phrase_starts indexes string into index_list
→ phrase_lengths indicate length of each input string

build_char_and_index_lists():
Uses grapheme to cluster string

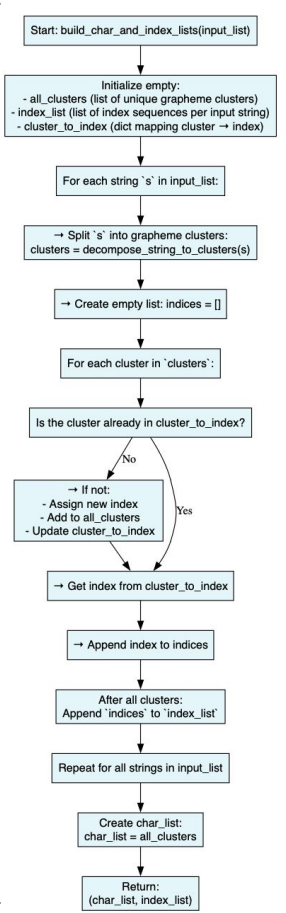
'héllo' → [h, é, l, l, o]

Each new cluster needs a bitmap
Duplicate clusters have 1 bitmap

[h, é, l, l, o] → [h, é, l, o]

String converted to index_list into char list

'héllo' → [0, 1, 2, 2, 3]



Creating the bitmaps

- Once the required bitmaps to be created are input in the form [‘consonant + vowel + tone marker’,…], the bitmaps for the required glyph combinations are created and input into a C++ file.
- We want this to be generalisable so font and size can be changed
- We also want this to be wrapped up in a function so it can be applied all at once and automatically generate the C++ file with the bitmaps so that main function can send the programme to the Arduino all with the press of one button

```
def generate_bitmaps_for_chars(char_list, font_path="./font_files/NotoSansLao-Regular.ttf", output_header="./arduino_code/glyph_bitmaps.h"):
```

Harfbuzz

- Harfbuzz is a shaping engine that calculates the positions of each component of the glyph so that it can be displayed correctly.
- In order to do that for our characters, Harfbuzz is first initialised with for the font of our choice (in this case Noto Sans Regular) like so:
- For each character (in a for loop), the Unicode string for each of the components of the combined glyph are fed in, harfbuzz calculates the shaping and returns where to put each component:

```
with open(font_path, "rb") as f:
    font_data = f.read()

# Initialize HarfBuzz
hb_blob = hb.Blob(font_data)
hb_face = hb.Face(hb_blob)
hb_font = hb.Font(hb_face)
```

```
buf = hb.Buffer()
buf.add_str(char)
buf.guess_segment_properties()
hb.shape(hb_font, buf)

infos = buf.glyph_infos
positions = buf.glyph_positions
```

Freetype

- If Harfbuzz is like the typesetter, Freetype is like the printer. It takes the glyph indices and corresponding position data and creates a bitmap of its own for each character.
- Freetype is initialised like so:

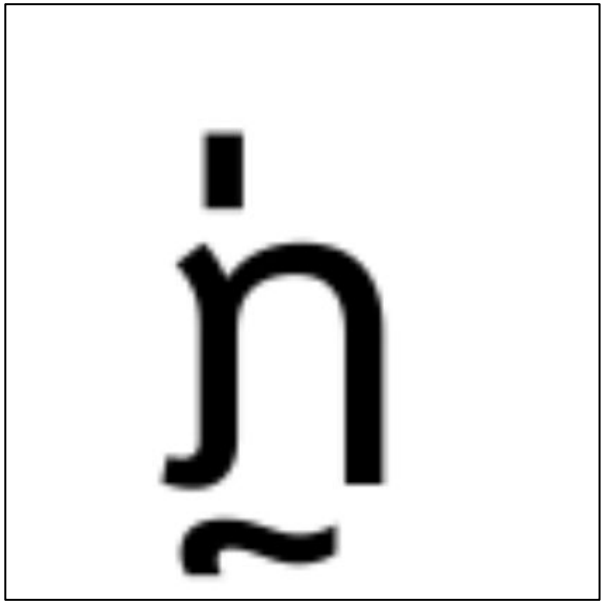
```
with tempfile.NamedTemporaryFile(delete=False, suffix=".ttf") as tmp_font_file:
    tmp_font_file.write(font_data)
    tmp_font_path = tmp_font_file.name

face = freetype.Face(tmp_font_path)
face.set_char_size(72 * 64)
```

- A temporary file is needed as freetype.Face needs a font file on disk. It cannot load a font directly from memory. The freetype.Face object contains a lot of key information for rendering each character.

Character to Image

An image of the character is then created like so:

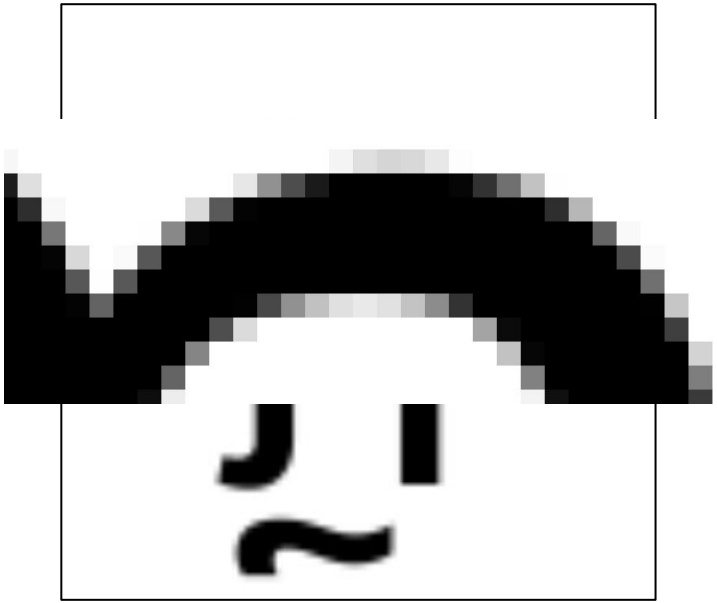
```
image_width, image_height = 100, 100
image = Image.new("L", (image_width, image_height), 255)
x, y = 25, 80  # Starting position

for info, pos in zip(infos, positions):
    glyph_index = info.codepoint
    face.load_glyph(glyph_index, freetype.FT_LOAD_RENDER | freetype.FT_LOAD_TARGET_NORMAL)
    bitmap = face.glyph.bitmap

    w, h = bitmap.width, bitmap.rows
    top = face.glyph.bitmap_top
    left = face.glyph.bitmap_left

    if w > 0 and h > 0:
        glyph_image = Image.frombytes('L', (w, h), bytes(bitmap.buffer))
        x_pos = x + (pos.x_offset // 64) + left
        y_pos = y - (pos.y_offset // 64) - top
        image.paste(0, (x_pos, y_pos), glyph_image)

    x += pos.x_advance // 64
    y -= pos.y_advance // 64
```



Character to Image

An image of the character is then created like so:

```
image_width, image_height = 100, 100
image = Image.new("L", (image_width, image_height), 255)
x, y = 25, 80  # Starting position

for info, pos in zip(infos, positions):
    glyph_index = info.codepoint
    face.load_glyph(glyph_index, freetype.FT_LOAD_RENDER | freetype.FT_LOAD_TARGET_NORMAL)
    bitmap = face.glyph.bitmap

    w, h = bitmap.width, bitmap.rows
    top = face.glyph.bitmap_top
    left = face.glyph.bitmap_left

    if w > 0 and h > 0:
        glyph_image = Image.frombytes('L', (w, h), bytes(bitmap.buffer))
        x_pos = x + (pos.x_offset // 64) + left
        y_pos = y - (pos.y_offset // 64) - top
        image.paste(0, (x_pos, y_pos), glyph_image)

    x += pos.x_advance // 64
    y -= pos.y_advance // 64
```



Too high a resolution?

- As you can see the image created is much too detailed compared to what is necessary.
- The pixels are a range of colours on a grey scale rather than just black or white.
- There are also far more pixels than are necessary

Solution: down-sample

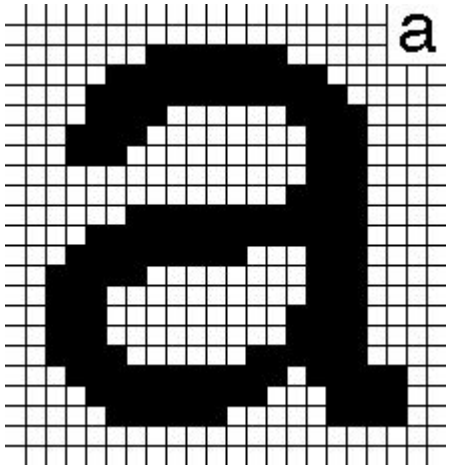
- The image is down-sized to 30x30 pixels (this can be changed) for different font sizes
- It is also converted to a 1-bit monochrome image to be more memory efficient and since that's the easiest way for the screen to operate.

```
img_resized = image.resize((30, 30), Image.Resampling.NEAREST)
img_bw = img_resized.point(lambda p: 0 if p < 128 else 255, mode='1')
```

Image to bits function

```
def image_to_bits(image):
    pixels = image.load()
    byte_array = []
    for y in range(GLYPH_HEIGHT):
        byte = 0
        bits_filled = 0
        for x in range(GLYPH_WIDTH):
            pix = pixels[x,y]
            bit = 0 if pix == 1 else 1
            byte = (byte << 1) | bit
            bits_filled += 1
            if bits_filled == 8:
                byte_array.append(byte)
                byte = 0
                bits_filled = 0
        if bits_filled > 0:
            byte <<= (8-bits_filled)
            byte_array.append(byte)
    print(len(byte_array))
    return byte_array
```

- Draws image of character
- Reads pixel value @ location in image
- Shifts bits to left and adds new bit
- Appends to an array once a byte is formed
- If we reach the end of a row and we don't have a complete byte, pad with zeros at end



WikimediaCommons, 2025, https://commons.wikimedia.org/wiki/File:Bitmapfont.png

Repeat for every row in the image

Convert image to bitmap

- Using an adaptation of the image to bits function that Diya has detailed, this image is converted to a bitmap

```
pixels = img_bw.load()
byte_array = []
for y_row in range(30):
    byte = 0
    bits_filled = 0
    for x_col in range(30):
        pix = pixels[x_col, y_row]
        bit = 1 if pix == 0 else 0
        byte = (byte << 1) | bit
        bits_filled += 1
        if bits_filled == 8:
            byte_array.append(byte
            byte = 0
            bits_filled = 0
    if bits_filled > 0:
        byte <<= (8 - bits_filled)
        byte_array.append(byte)
```

Output: C++ file

- Rather than having to paste the formed bitmaps into a C++ file, the code directly creates and writes a C++ file, in form that we had previously managed to produce phrases from.
- It writes this C++ file by doing the following:

```
with open(output_header, "w", encoding="utf-8") as f:
    guard = output_header.upper().replace('.', '_')
    f.write(f"#ifndef {guard}\n")
    f.write(f"#define {guard}\n\n")
    f.write("#include <avr/pgmspace.h>\n\n")
    f.write("static const uint8_t glyph_bitmaps[] PROGMEM = {\n")

    # Write bytes in rows of 16 for readability
    for i in range(0, len(all_bytes), 16):
        line_bytes = all_bytes[i:i+16]
        line = ", ".join(f"0x{b:02X}" for b in line_bytes)
        f.write(f"{line},\n")

    f.write("};\n\n")
    f.write(f"#endif // {guard}\n")
```

Sample output

```
#ifndef GLYPH_BITMAPS_H
#define GLYPH_BITMAPS_H

#include <avr/pgmspace.h>

static const uint8_t glyph_bitmaps[] PROGMEM = {
0x00, 0x24, 0x00, 0x00, 0x00, 0x6A, 0x00, 0x00, 0x00, 0x42, 0x00, 0x00, 0x00, 0x37, 0x80, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x0C, 0x00, 0x00, 0x00, 0x7F, 0x80, 0x00, 0x00, 0xC0, 0xC0, 0x00,
0x00, 0xC0, 0xC0, 0x00, 0x00, 0x63, 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x47, 0x80, 0x00, 0x00, 0x78, 0xC0, 0x00, 0x00, 0x30, 0x60, 0x00, 0x00, 0x30, 0x60, 0x00,
0x00, 0x30, 0x60, 0x00, 0x00, 0x30, 0x60, 0x00, 0x00, 0x30, 0x60, 0x00, 0x00, 0x30, 0x60, 0x00,
0x00, 0x30, 0x60, 0x00, 0x00, 0x30, 0x60, 0x00, 0x00, 0x70, 0x60, 0x00, 0x00, 0xE0, 0x60, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x18, 0x00, 0x00, 0x00, 0x3C, 0x00, 0x00,
0x00, 0x3C, 0x00, 0x00, 0x00, 0x18, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0C, 0x00, 0x00,
0x00, 0x7F, 0x80, 0x00, 0x00, 0xC0, 0xC0, 0x00, 0x00, 0xC0, 0xC0, 0x00, 0x00, 0x63, 0x80, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x47, 0x80, 0x00, 0x00, 0x78, 0xC0, 0x00,
0x00, 0x30, 0x60, 0x00, 0x00, 0x30, 0x60, 0x00, 0x00, 0x30, 0x60, 0x00, 0x00, 0x30, 0x60, 0x00,
0x00, 0x30, 0x60, 0x00, 0x00, 0x30, 0x60, 0x00, 0x00, 0x30, 0x60, 0x00, 0x00, 0x30, 0x60, 0x00,
0x00, 0x70, 0x60, 0x00, 0x00, 0xE0, 0x60, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
};

#endif // GLYPH_BITMAPS_H
```

Interfacing with the arduino

```
void loop() {
  //loops through every phrase and displays all
  for (int i=0; i<num_phrases; i++){
    const uint8_t* lao_phrase = &all_phrases[phrase_starts[i]];
    uint8_t len_phrase = phrase_lengths[i];
    scrollPhrase(lao_phrase, len_phrase);
}
}
```

Arduino code

phrases_to_display.h

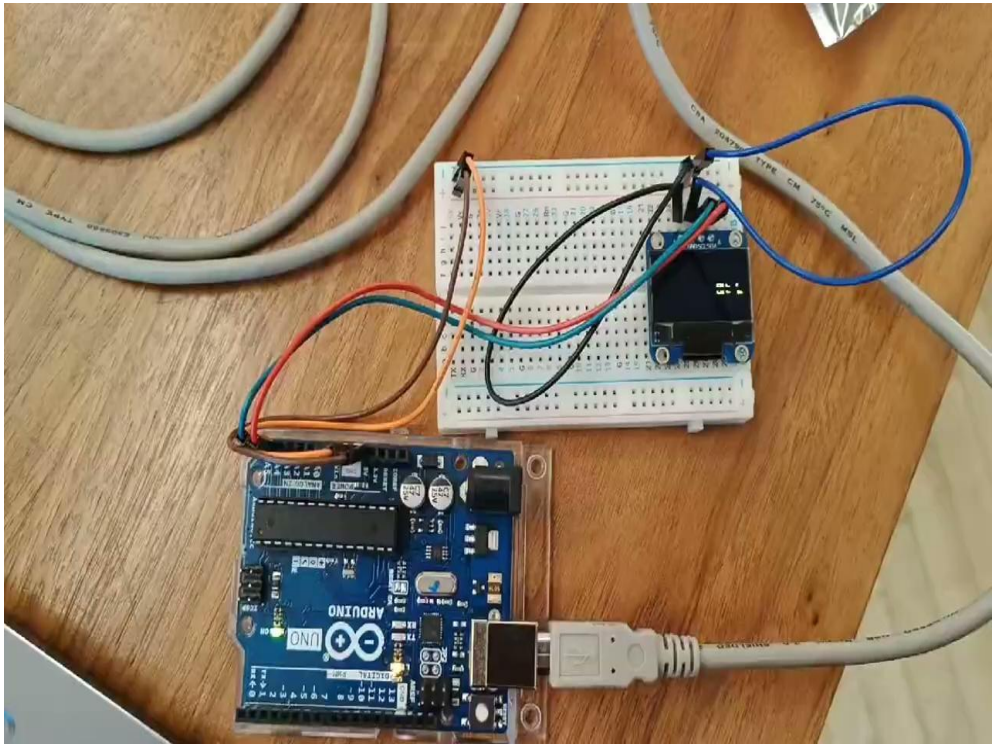
```
#ifndef PHRASES_TO_DISPLAY_H
#define PHRASES_TO_DISPLAY_H

const uint8_t all_phrases[] = {
  0, 1, 2, 3, 4, 5, 6, 7, 8,    // phrase 1
  0, 6, 6, 8,                   //phrase 2
  4, 4                         //phrase 3
};

const uint8_t phrase_starts[] = {0, 9, 13};     // starting index of each phrase
const uint8_t phrase_lengths[] = {9, 4, 2};    // length of each phrase
const uint8_t num_phrases = 3;

#endif
```

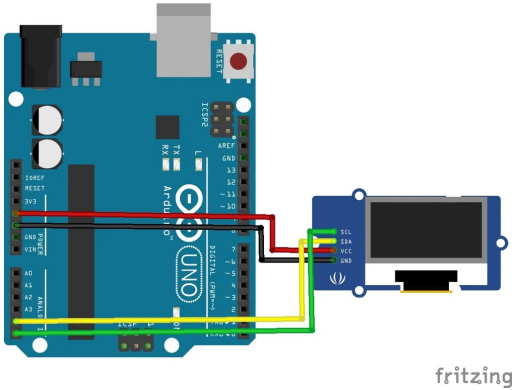
glyph_bitmaps.h

```
int x = -scroll_offset;
int y = (SCREEN_HEIGHT - BITMAP_HEIGHT)/2;  //y does not change
for (int i =0; i<len_phrase; i++) {
  int glyph_index = lao_phrase[i];
  memcpy_P(glyph_buffer, glyph_bitmaps + (glyph_index * BYTES_PER_BITMAP), BYTES_PER_BITMAP);
  display.drawBitmap(x, y, glyph_buffer, BITMAP_WIDTH, BITMAP_HEIGHT, SSD1306_WHITE); //draws
  x +=BITMAP_WIDTH/2; //moves along by BITMAP_WIDTH to the next grapheme position
}
display.display(); //display the word
```

```
static const uint8_t glyph_bitmaps[] PROGMEM = {
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
```

A video demo!





Autodesk Instructables,
https://www.instructables.com/OLED-I2C-DISPLAY-WITH-ARDUINO-Tutorial/

Maximising Ease of Deployment

- compact_lao_messages_app
  - arduino_code
  - compact_lao_messages_app
  - font_files
  - .gitignore
  - README.md
  - requirements.txt
  - run.py
  - debug.py
  - generate_bitmaps.py
  - main.py
  - preprocess_strings.py

Best practice file structure:
.gitignore for large files
requirements.txt for 1 line package installation
Isolated modules for easy re-use
run.py as easy entry point
Considered Docker & online hosting options

Key aims:
1. Ease of Makerbox deploying & reuse our code.
2. Accessibility for other individuals in laos

README.md contains:
Project overview
Setup & Run instructions
Explanation of main loop

Outputs in arduino_code

Modules contain:
Reusable functions
Comments
Docstrings

```
"""
Utilities for processing Unicode strings into grapheme clusters,
generating index mappings, and exporting them for embedded C++ use.
"""
```

Compact Lao Messages App for Arduino

This project provides a Python-based pipeline for:

Reading Unicode text (including complex scripts like Lao), Decomposing each string into human-readable grapheme clusters, Rasterizing each grapheme using a chosen font, Generating C++ header files to display text on a pixel-based embedded display (e.g., Arduino).

Supports any script thanks to HarfBuzz and FreeType.

Setup

1. Clone the Repository:
   `git clone https://github.com/ ...`
   `cd compact_lao_messages_app`
2. Create a Virtual Environment:
   `python3 -m venv .venv`
   `source .venv/bin/activate` # On Windows: `.venv\Scripts\activate`
3. Install Dependencies:
   `pip install -r requirements.txt`

   Required packages include:
   - Pillow
   - freetype-py
   - uharfbuzz
   - grapheme
   - tqdm

   You may also need system-level dependencies like freetype-dev.

Run

1. `python3 run.py` # On Windows: `py -m run.py` (from within the `compact_lao_messages_app` directory)

Main Loop Explanation

1. Input String Handling

   CSV Check:
   The app looks for `./input_files/input_strings.csv`, which stores Lao strings to convert. An example input_strings.csv file for editing can be found under the input_files folder.

   Overwrite Prompt:
   If the file exists, it asks the user whether to overwrite it. If the user chooses to overwrite (y), it: Prompts the user for new strings (`get_input_strings()`). Saves them to CSV using `save_strings_to_csv()`.

   Otherwise:
   It loads the existing strings from the CSV.

2. Character Analysis

   Build Unique Characters:
   It extracts all unique characters across the input strings and builds: `char_list`: the sorted list of unique characters. `index_list`: a list of indices representing which characters form each phrase.

   Debug Print: Displays both lists for developer inspection using `print_char_and_index_lists()`.

3. Bitmap Generation

   Create Bitmaps:
   It generates monochrome bitmap images (bit-packed) for each unique character using `generate_bitmaps_for_chars()`. These are written to a C header file: `./arduino_code/glyph_bitmaps.h`

Improvements to current solution

File size constraints
Currently indicates file sizes
Warning if **program files > flash memory**
Compression scheme
rules-based may achieve **< 32kB** size

Arduino Board	Family	MEMORY SRAM	FLASH	EEPROM
Duemilanove (328)	ATmega328	2K	32k	1kB
Uno	ATmega328	2k	32k	1kB
Arduino Mega 2560	ATmega2560	8k	256k	1kB
Arduino Mega ADK	ATmega2560	8k	256k	1kB
Arduino Ethernet	ATmega328	2k	32k	1kB
Arduino BT	ATmega328	2k	32k	1kB
Arduino Pro Mini 328 5V	ATmega328	2k	32k	1kB

More modularity
Easier selection of alternative fonts (Phetsarath..)
Simple change of font size option
Choice of fonts on arduino


```
Generating bitmaps for characters...
Processing characters: 100%|                                                  | 19/19
Bitmap header file size: 13.55 KB
Writing index list to header file...
Index header file size: 0.42 KB
```

Packed Bitmap	Run Length Encoding
Every **8 pixels = 1 byte** Each bit is a pixel Easy to index **bitmap[y * width + x]** Size: **width * height / 8** bytes Good for **random access**, poor compression	Each byte contains: • **1 bit**: color (MSB) • **7 bits**: run length (max 127) very **compact** **Requires tiled bitmaps OR pre-computed offsets**

Generalising to non-arduino microcontrollers
Abstract `PROGMEM` Access
Avoid Adafruit GFX, U8g2 libraries & use standard <stdint.h> var types
Use **PlatformIO** instead of **Arduino IDE**

Thank you for listening

We will now take questions.