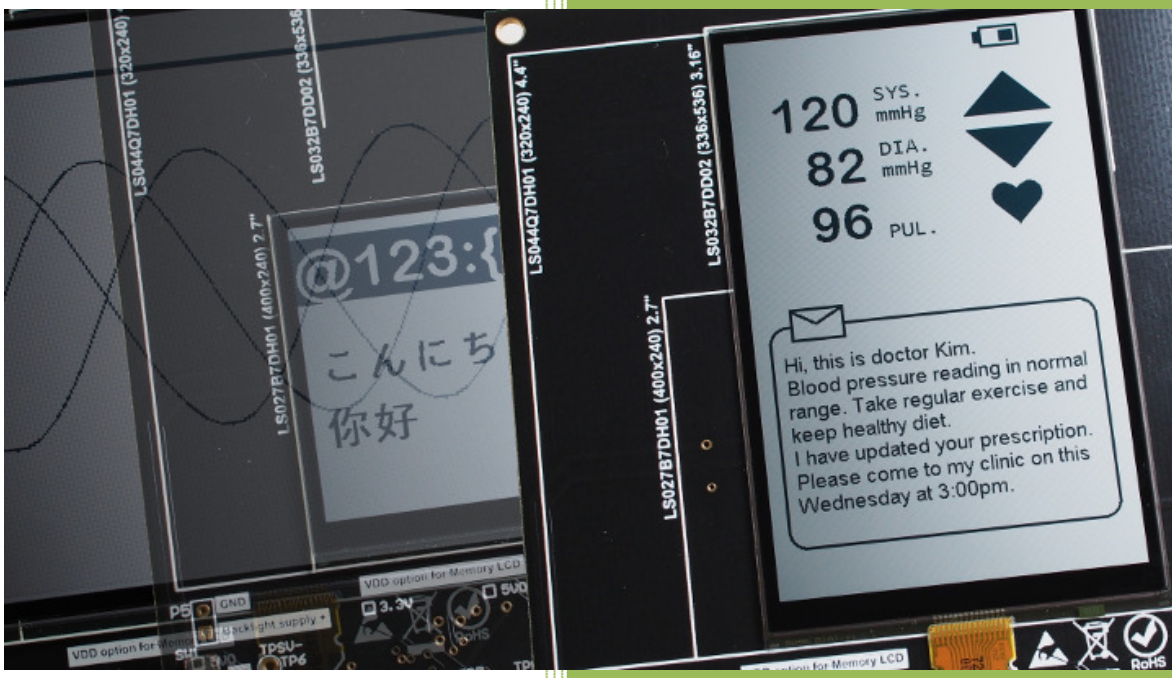
Sharp Memory LCD Shield User Guide



Written by John Leung

Document version 1.1

12/7/2018

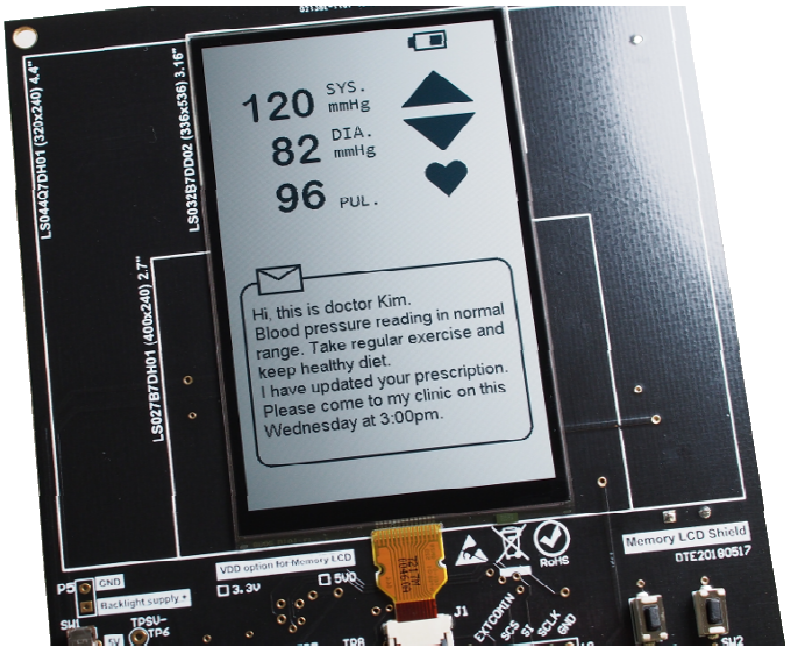
Update

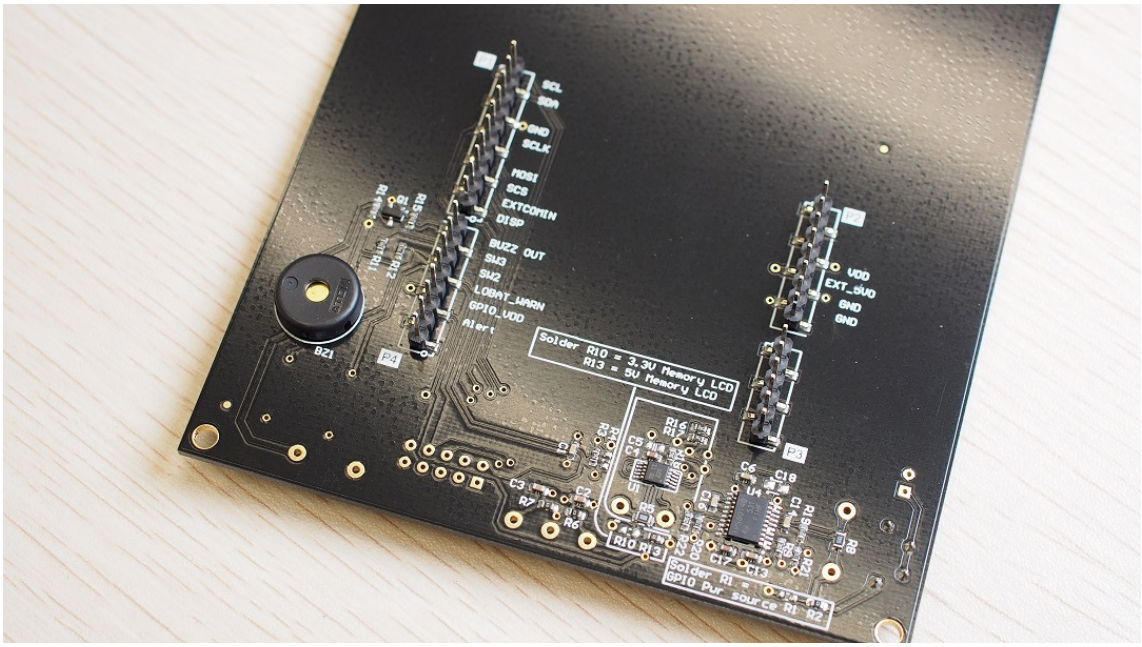
1) Add support for two new Memory LCD modules LS006B7DH03 (64*64) and LS011B7DH03(160*68)

Contents at a Glance

Chapter 1 Introduction

Sharp's Memory LCD (https://www.sharpsma.com/products?sharpCategory=Memory%20LCD) is a lightweight display with 1-bit memory in every pixel allowing high-contrast, ultra-thin and at the time same delivering a relatively high frame rate (20Hz max) at merely microWatt power consumption level. An evaluation kit compatible with 3.3V Arduino platforms has been built to facilitate testing and product development for the LCDs. At time of writing, five Memory LCD models LS027B7DH01 (2.7"), LS032B7DD02 (3.2"), LS044Q7DH01(4.4"), LS006B7DH03 (0.56"), and LS011B7DH03 (1.08") have been tested with firmware designed for two Arduino boards - Arduino M0 PRO and ESP32.

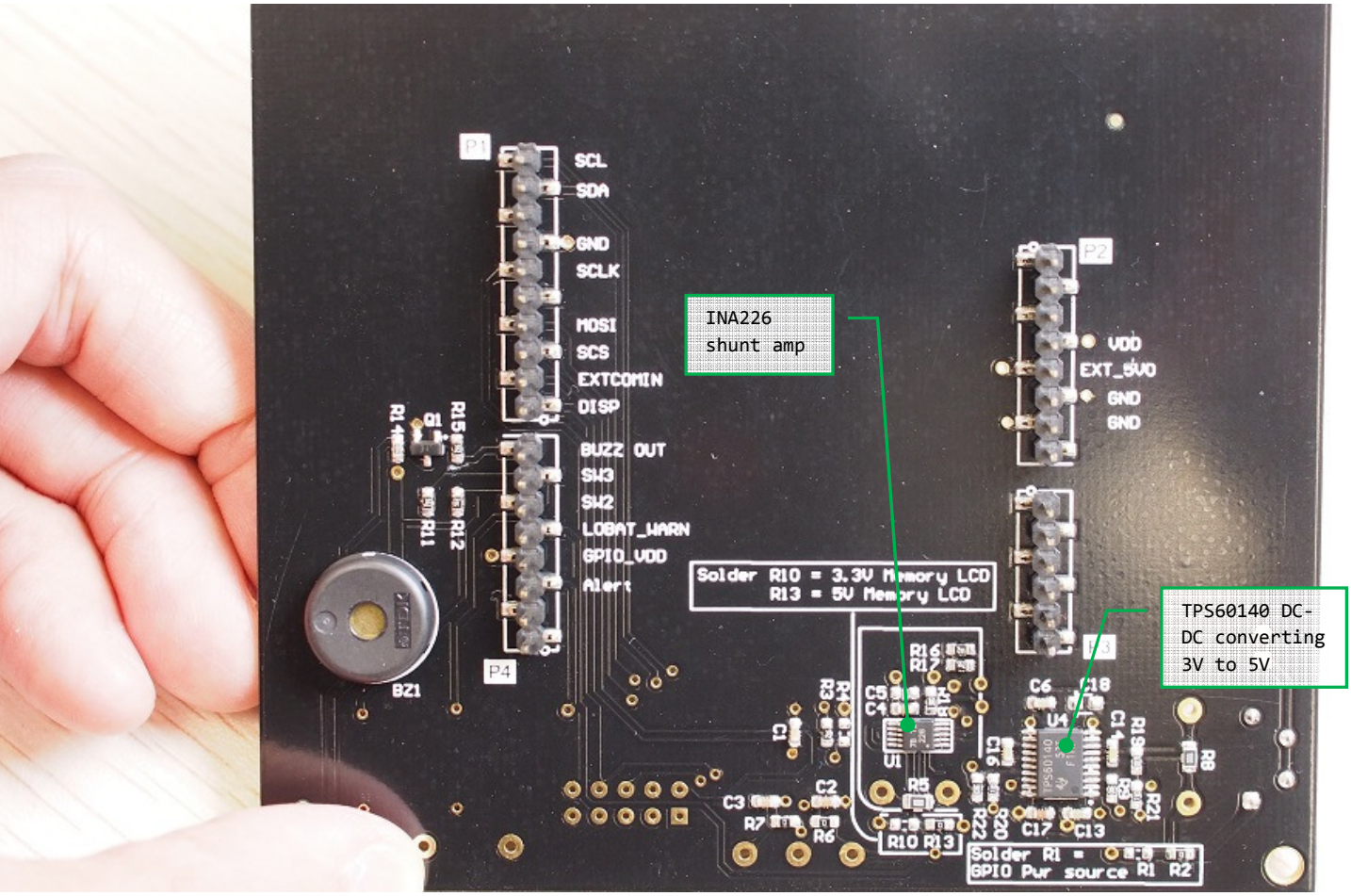




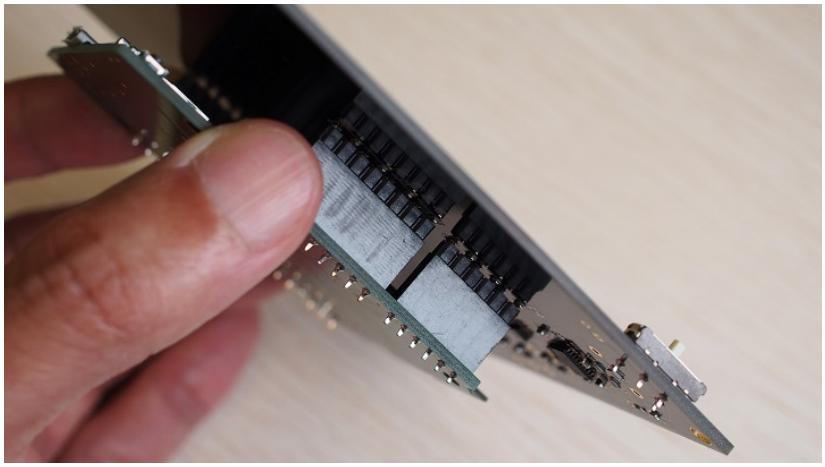
Chapter 2 Setting up the hardware

Memory LCD

Simply stack the shield on top of any 3.3V Arduino compatible board to start development. Bonus features include a high precision shunt amplifier INA226 for measuring power consumption on 100Ω ±0.1% shunt resistor. Illustration of the board is shown below. Schematic and PCB layout of the shield can be found in Appendix A of this document.



Stack Memory LCD Shield on top of any 3.3V version Arduino-compatible evaluation kit to start development.

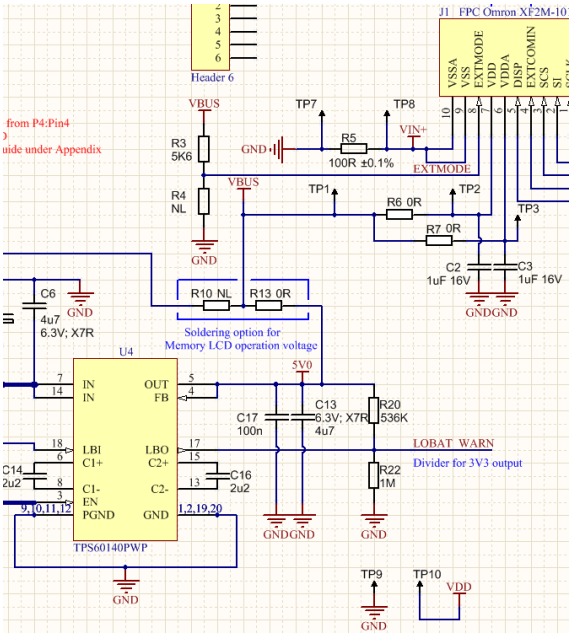


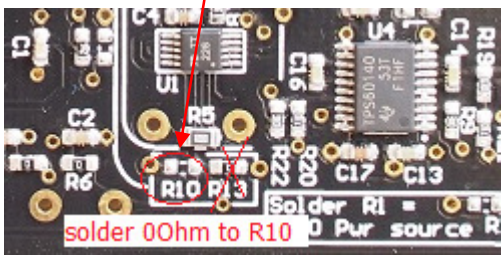
Select 5V *v.s* 3V LCD voltage

It is important to notice that there are two operation voltages for different Memory LCD models, 5V and 3V. Table below summaries the electrical characteristics of different models.

Model	VDDA MIN.	VDDA TYP.	VDDA MAX.
LS027B7DH01 (2.7")	4.8V	5.0V	5.5V
LS032B7DD02 (3.2")	4.8V	5.0V	5.5V
LS044Q7DH01(4.4")	4.8V	5.0V	5.5V
LS006B7DH03 (0.56")	2.7V	3.0V	3.3V
LS011B7DH03 (1.08")	2.7V	3.0V	3.3V

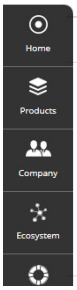
Compatibility is enabled by a jumper resistor to select between 5V and 3.3V as the VDDA (and VDD as well) supply. Extract below shows a part of the schematic. Resistor R10 is not soldered by default. Every evaluation kit ex-factory was soldered with a 0 Ohm resistor at R13 to source 5V output from TPS60140PWP DC-DC converter as the VDDA (and VDD). If you need to use 3V Memory LCD, please swap R13 and R10 which mean to unsolder R13 and use the same resistor at R10 to complete the bridge between 3.3V supply from Arduino host board to VDDA (and VDD) of Memory LCD.





Application Processor - ESP32 WiFi/BLE SoC

Application processor (AP in short) is the microcontroller or System-on-Chip (SoC) to draw every pixel on screen. We are doing the job with ESP32-WROOM-32 which is an off-the-shelf Wifi and Bluetooth Low Energy (BLE) SoC. By "*googling*" the keyword ESP32 there are myriad links and resources available. Official web site can be found at `https://www.espressif.com/en/products/hardware/modules`. The first module on list of the table is the model we have chosen.



Module	Description	Chip Embedded	Dimensions (mm)	Pins	Flash (MB)	PSRAM (MB)	Antenna	Development Board
 ESP32-WROOM-32	ESP32-WROOM-32 contains the ESP32 SoC, flash memory, high-precision discrete components, and a PCB antenna which provides outstanding RF performance in space-constrained applications.	ESP32-D0WDQ6	18x25.5x2.8	38	4	N/A	PCB antenna	ESP32-DevKitC

There are many reasons to opt for this device including its popularity, open-source, resourceful documentation, nicely supported SDK, low cost (~US$4), and most importantly there is an ample SRAM of 520KB together with 4MB Flash miniaturized on this 18x25mm PCB.

Gearing with 520KB SRAM is critical for our applications because a frame buffer is implemented to hold graphical contents. There are three Memory LCD models from resolution 320*240 (4.4") to 336*530 (3.2"). If we are storing full contents in SRAM for 1-bit pixel depth, the requirement is tabulated in table below. A divisor of 8 in arithmetic is required because we are using 1-byte to hold 8 pixels in 1-bit color depth.

Memory LCD	Resolution	SRAM (byte)
LS027B7DH01	400*240	400*200/8 = 12,000
LS044Q7DH01	320*240	320*240/8 = 9,600
LS032B7DD02	336*536	336*536/8 = 22,512
LS006B7DH03	64*64	64*64/8 = 512
LS011B7DH03	160*68	160*68/8 =1,360

As indicated a SRAM as high as 22KB is required and this memory is satisfied with 520KB SRAM of ESP32 SoC.

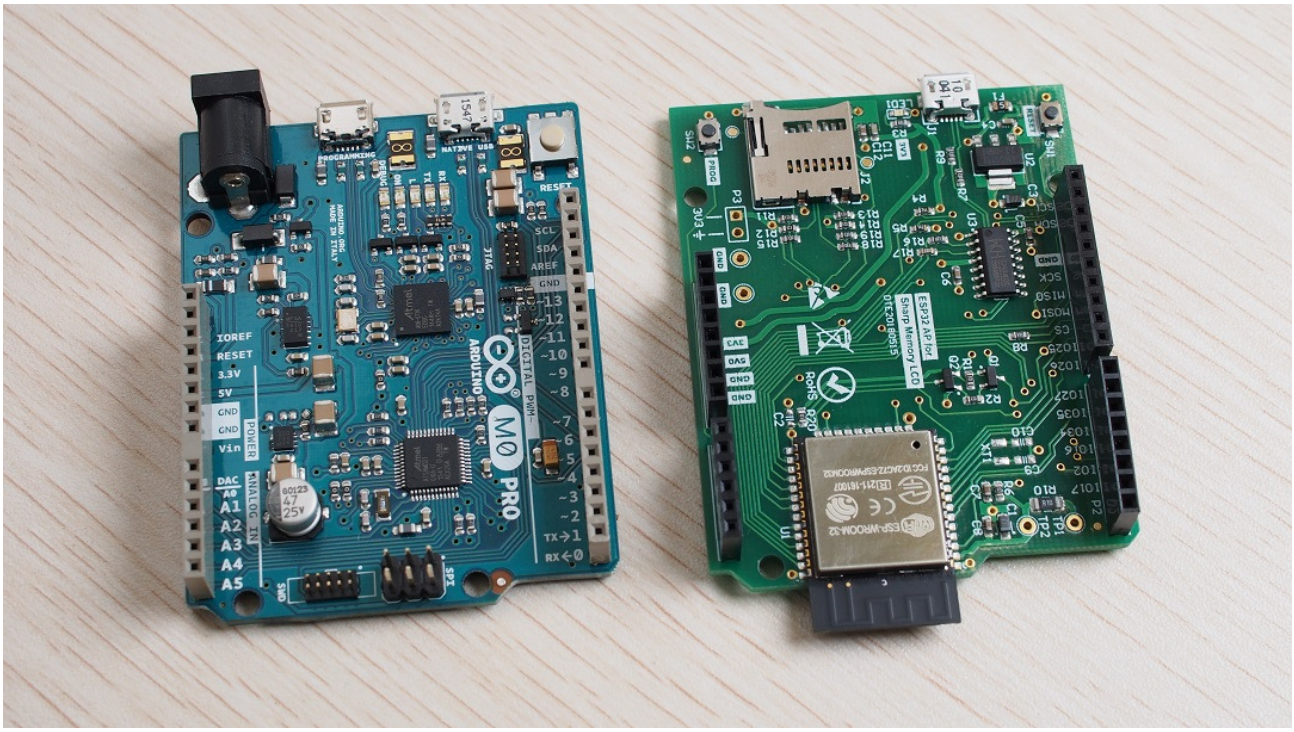
To fit an Arduino-UNO form factor, a PCB was designed with peripheral components including voltage regulator, USB-to-UART transceiver chip (CH340C), and an automatic binary file download circuitry for kick-starting development. Schematic and PCB layout of the ESP32 Application Processor is shown in Appendix B of this document.

Application Processor - Arduino M0 PRO

In addition to ESP32 we have also tested Arduino M0 PRO as the AP. This board has been chosen because it is readily available and there is a 32KB SRAM which is more than sufficient to support the frame buffer requirement explained in last section.

Getting started guide can be found under this hyperlink.

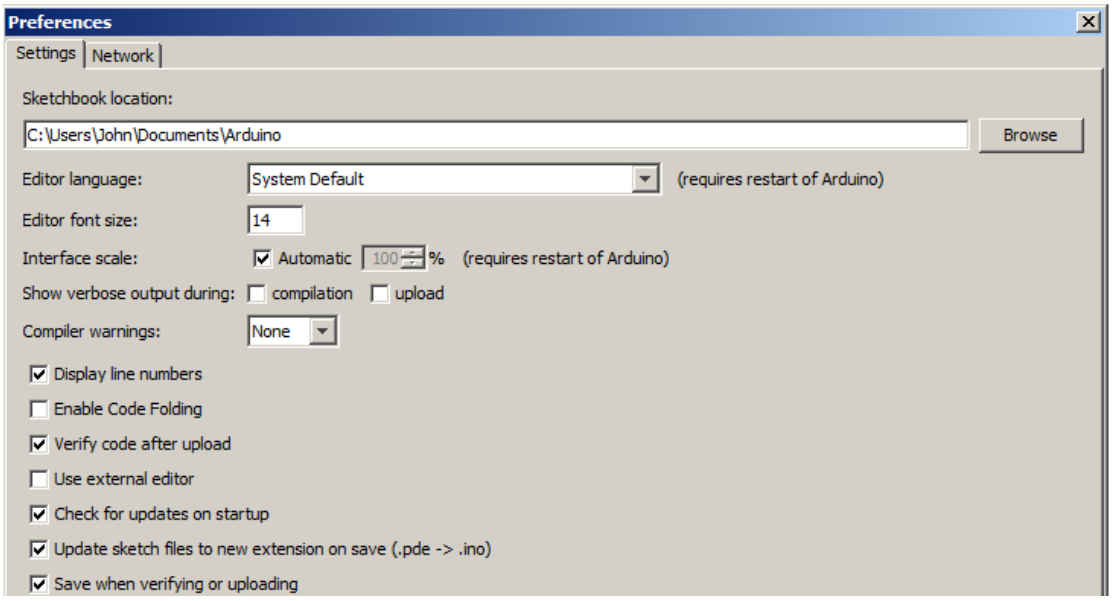
`https://www.arduino.cc/en/Guide/ArduinoM0Pro`



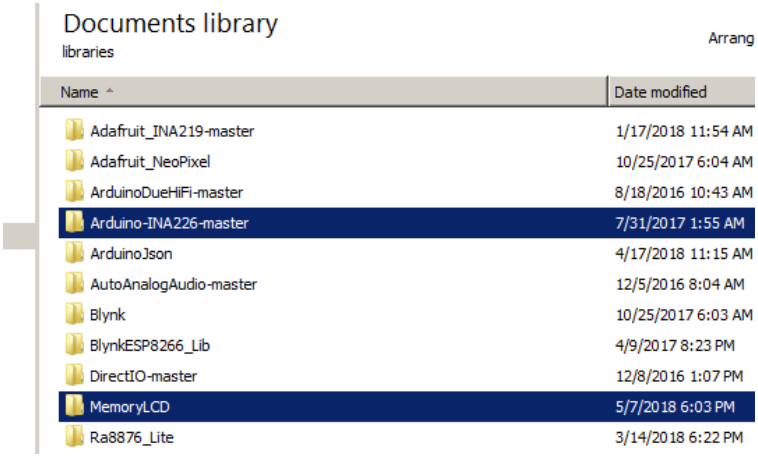
Chapter 3 Setting up the development environment

Download and install Arduino IDE version 1.8.1 or later.

Locate the library folder. This information is available from File→Preferences. In my case it is under `C:\Users\John\Documents\Arduino`.



By this time you should have obtained a copy of the Memory LCD driver from us. Extract the library and manually copy the folders to `C:\Users\John\Documents\Arduino`. *Your path may be different with your user name.* There are two libraries required, one for Memory LCD and the second for INA226 as shown below with highlights.

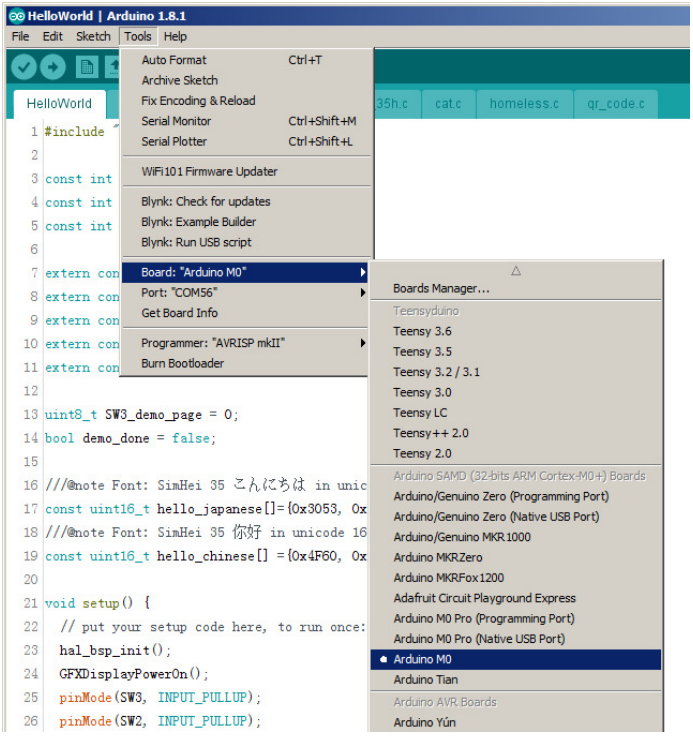


Restart Arduino IDE.

If you are using Arduino M0 PRO

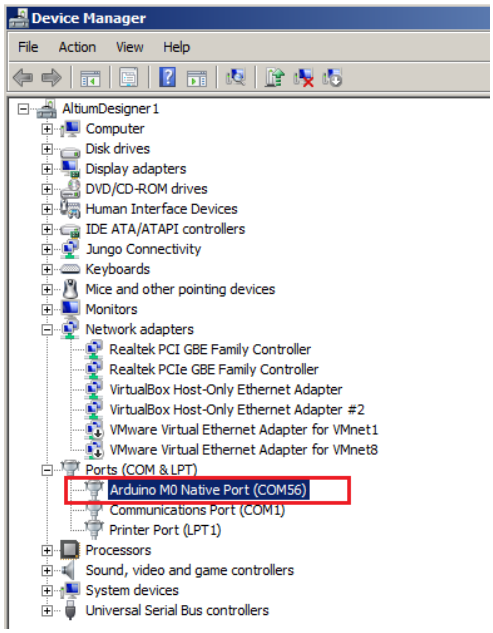
M0 PRO is natively supported in Arduino IDE so the only step is to Select Arduino M0 or Arduino M0 Pro (Programming Port) from `Tools→Board.`

If Arduino M0/M0 Pro is not available, you will have to install it from Boards Manager.



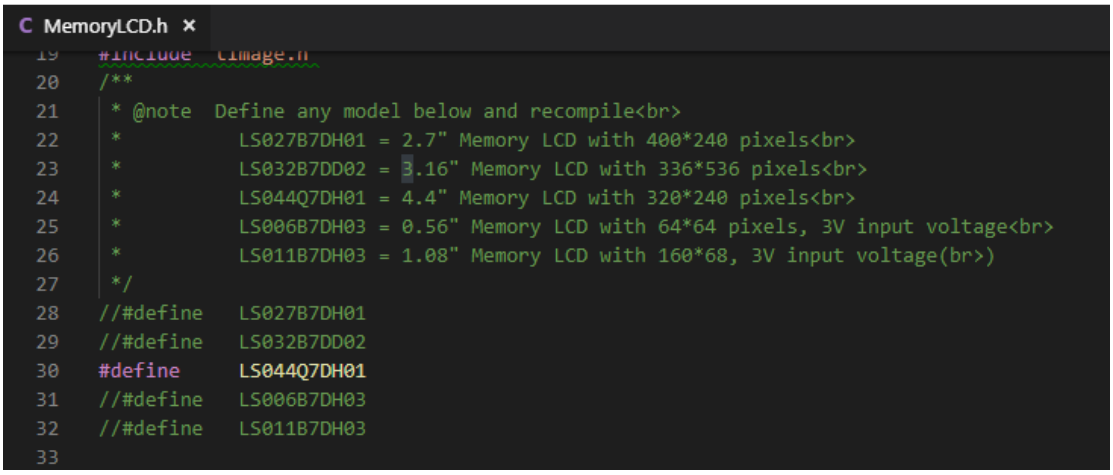
Connect USB port of Arduino M0 (or debug port of Arduino M0 PRO) to your PC.

From Device Manager you will see an enumerated port for M0/M0 PRO like screen shot below.

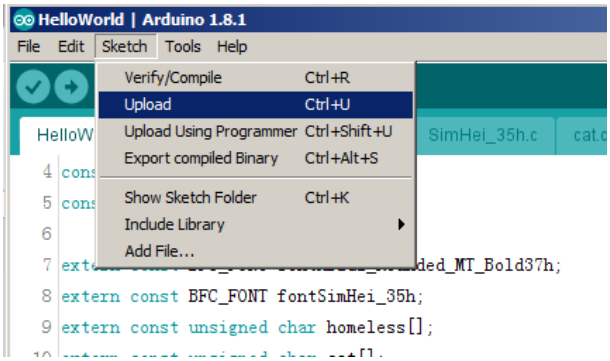


Make sure it is the same port number you have selected under `Tools→Port` for binary code download.

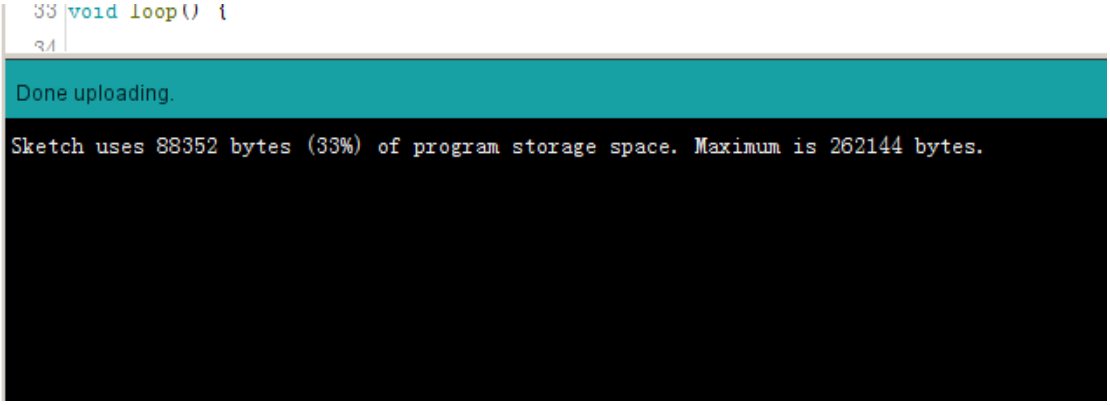
Memory LCD shield is compatible with five LCD models. The "switch" to control is located at the header file with `#define` as below. Select one of them for your Memory LCD model. You may open this .h file with your favorite text editing program. A Windows **Notepad** will be sufficient. My personal preference is Notepad++. **Don't forget to save the file**.



Finally click `Sketch→Upload`.



After download successful you will see a message from the console window



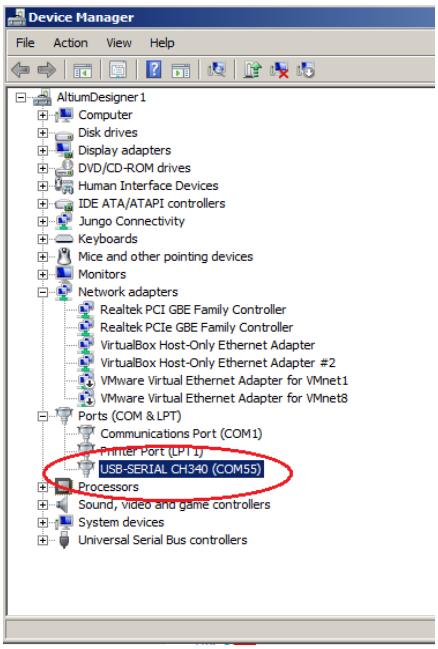
If you are using ESP32

Install ESP32 Arduino Core from `https://github.com/espressif/arduino-esp32`.

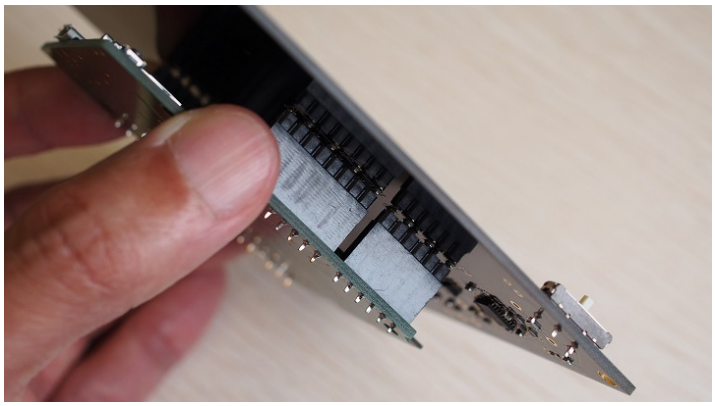
The installation procedure is available in full details from the download link above.

ESP32 AP board is using a USB-UART bridge CH340C to link up with a PC. Download its driver from http://www.wch.cn/download/CH341SER_ZIP.html and get it installed.

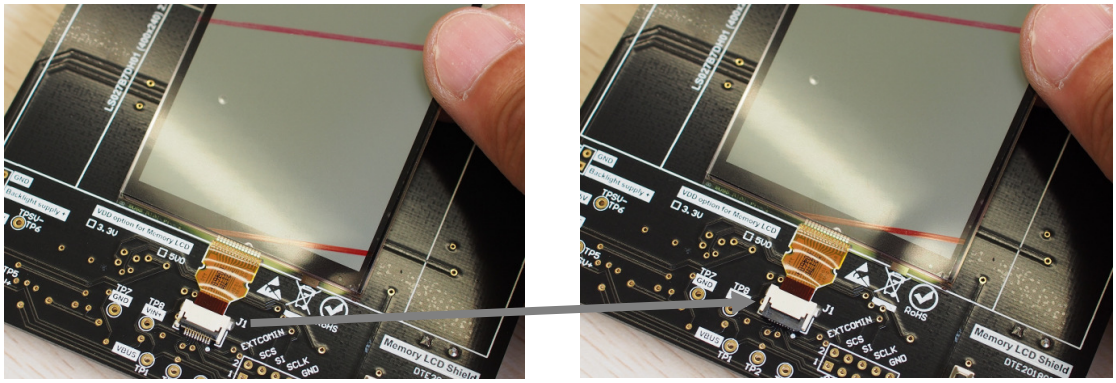
After download and installation, connect the USB port to PC and make sure there is a new COM PORT enumerated in Device Manager. In my case it is COM55. It could be different in your environment.



Unplug it from PC and stack Memory LCD Shield on it.

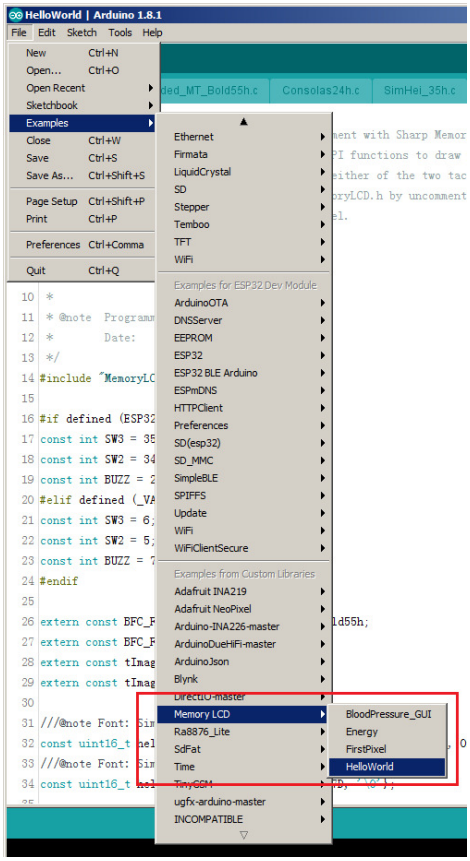


Install Memory LCD to the Omron dual contact connector onboard → close the zip lock. Reconnect USB cable.

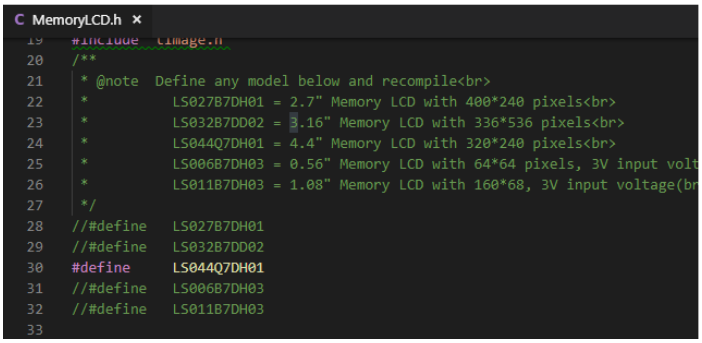


Restart Arduino IDE. Browse through Arduino Examples from `File→Examples→MemoryLCD→HelloWorld.`

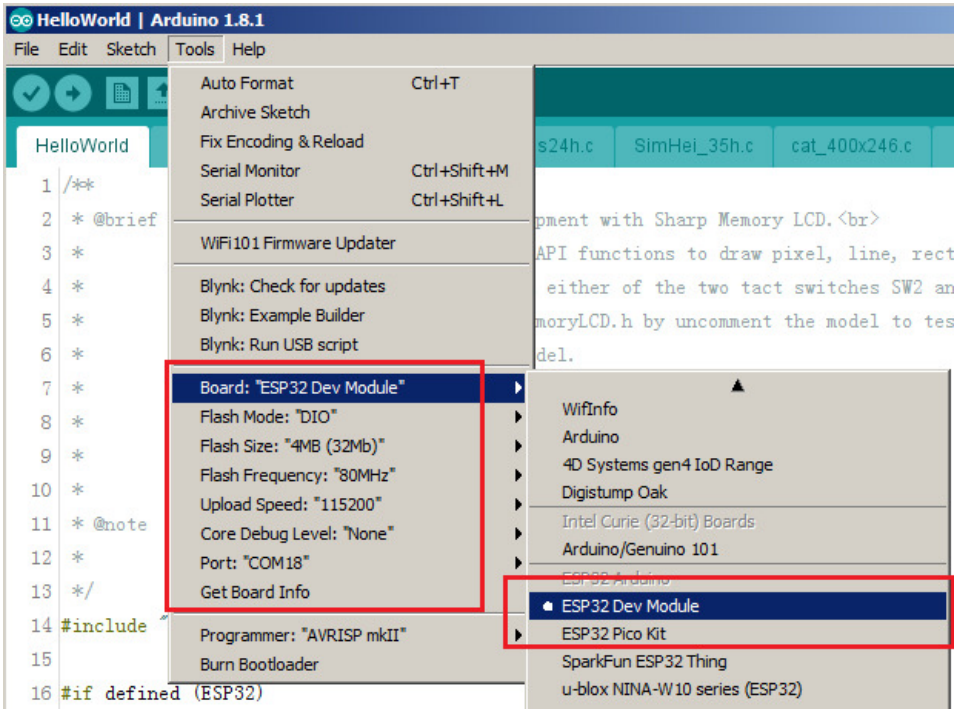
There are four examples at time of writing this document.



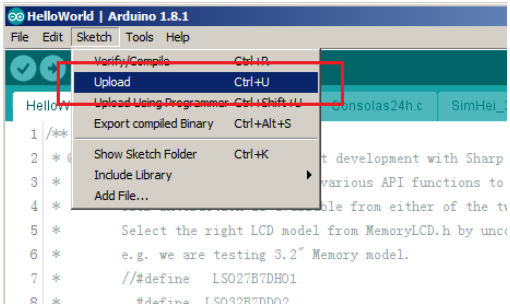
Make sure you have uncomment the LCD model from MemoryLCD.h header file for what you are using.



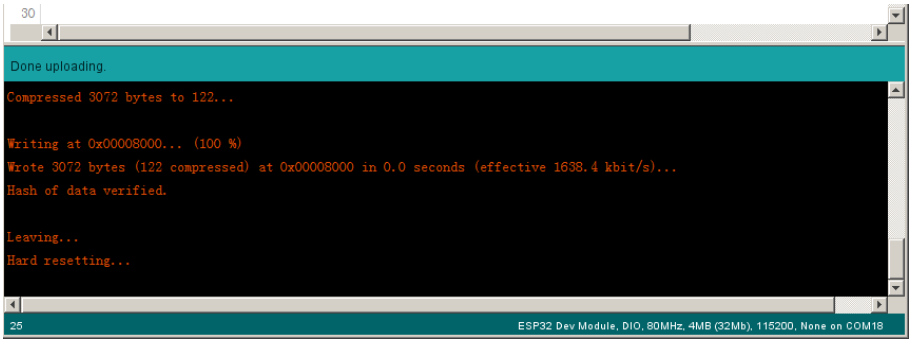
For ESP32 AP, from `Tools→Boards Manager`, select ESP32 Dev Module. Please also make sure parameters are selected according to screen shot below.



The last step is to click `Upload` from `Sketch→Upload`



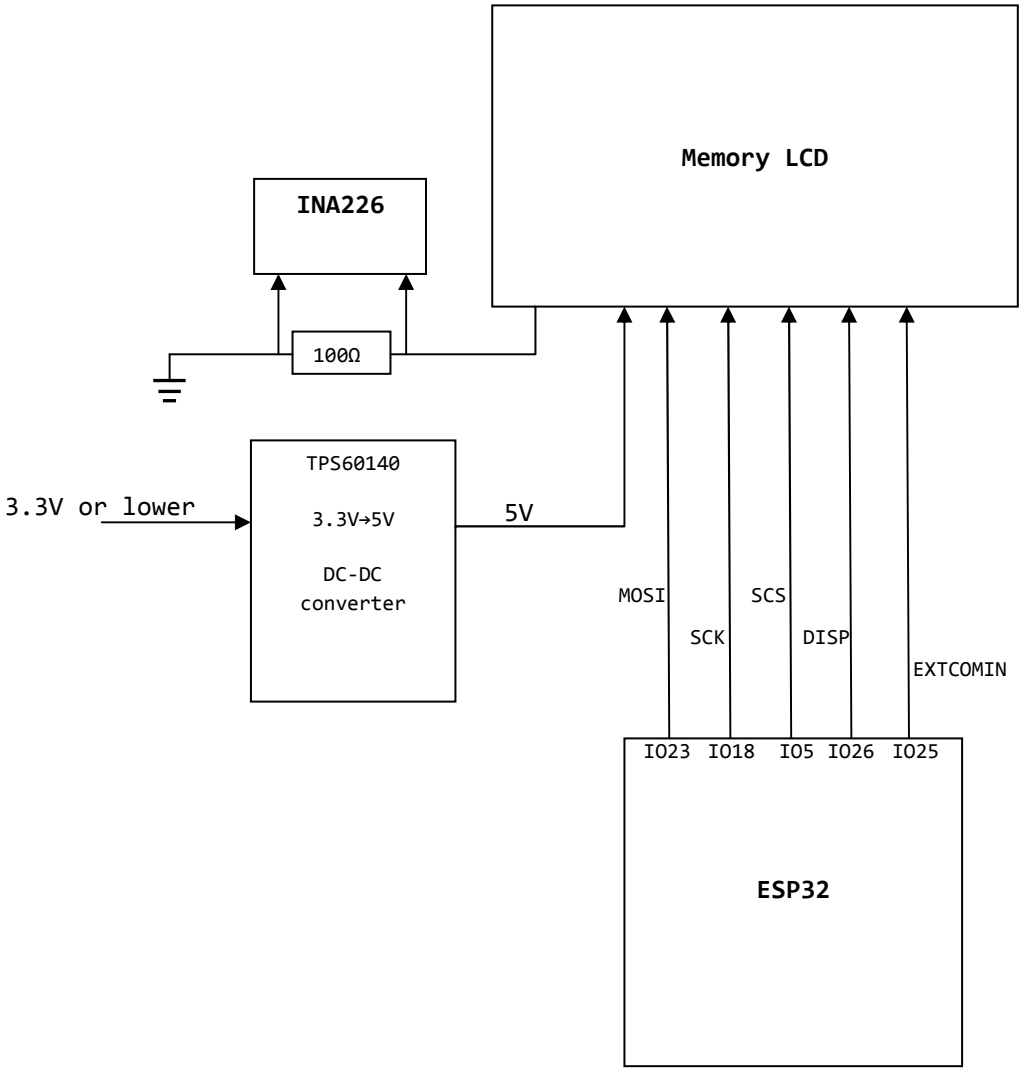
From Arduino's message window please make program compile OK with a `Done uploading` message.



Now if you are lucky, some texts will be visible. Free feel to click on any of the keys to browse through the demo.

Chapter 4 Describing how hardware works

Full schematics of Memory LCD Shield and ESP32 Application Processor are printed on the Appendix section of this document. A simplified block diagram showing different functional blocks is summarized below. ESP32 is shown in this block diagram for illustration purpose.



Connections for Arduino M0 PRO is similar with different pinout definitions declared in MemoryLCD.h controlled by directives in compile time.

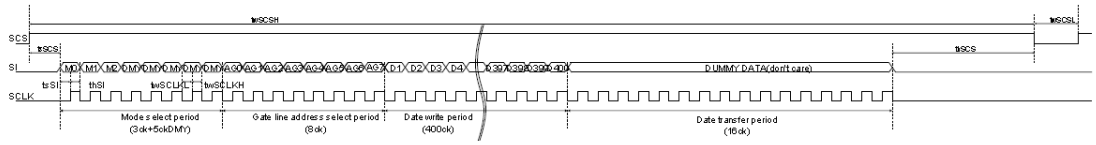
```
21
22  //@note pin number definition, hardware dependent
23  #if defined (ARDUINO)
24   #include "Arduino.h"
25  #if defined (_VARIANT_ARDUINO_ZERO_)
26      #define GFX_DISPLAY_SCS       10
27      #define GFX_DISPLAY_EXTCOMIN  9
28      #define GFX_DISPLAY_DISP      8
29      #define GFX_5V0_EN            3
30      #define USE_SERIAL            Serial
31   #else if defined (ESP32)
32      #define GFX_DISPLAY_SCS       5
33      #define GFX_DISPLAY_EXTCOMIN  25
34      #define GFX_DISPLAY_DISP      26
35      #define GFX_5V0_EN            2
36      #define USE_SERIAL            Serial
37  #endif
38  #endif
```

Describing how firmware works

Two data update modes are available, namely the 1-line mode and multiple-lines mode. Timing diagrams are available from LCD's datasheet with extracts shown below.

6-5-1 Data update mode (1 line)

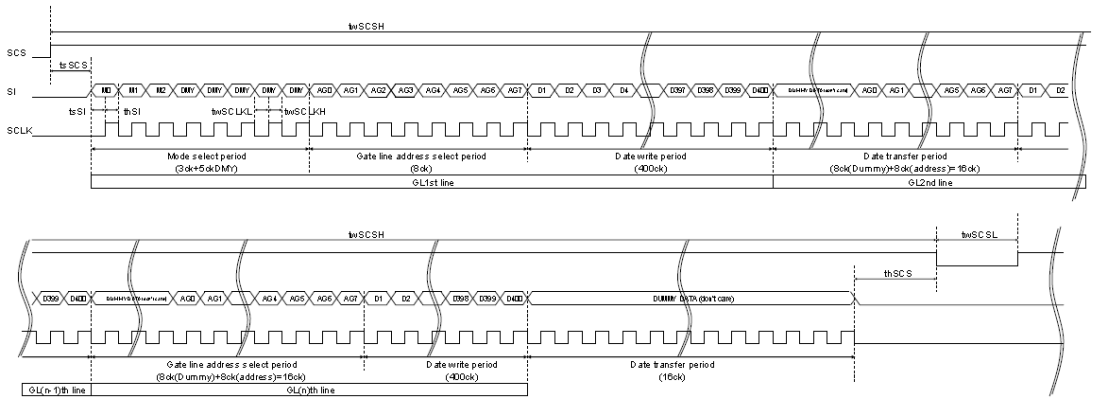
Updates data of only one specified line. (M0="H"、M2="L")



M0: Mode flag. Set for "H". Data update mode (Memory internal data update)
When "L", display mode (maintain memory internal data).
M1: Frame inversion flag.
When "H", outputs VCOM="H", and when "L", outputs VCOM="L".
When EXTMODE="H", it can be "H" or "L".
M2: All clear flag.
Refer to 6-5-4) All Clear Mode to execute clear.
DUMMY DATA: Dummy data. It can be "H" or "L" ("L" is recommended.)

6-5-2 Data Update Mode (Multiple Lines)

Updates arbitrary multiple lines data. (M0="H"、M2="L")



In the firmware these update modes are implemented by two local functions:

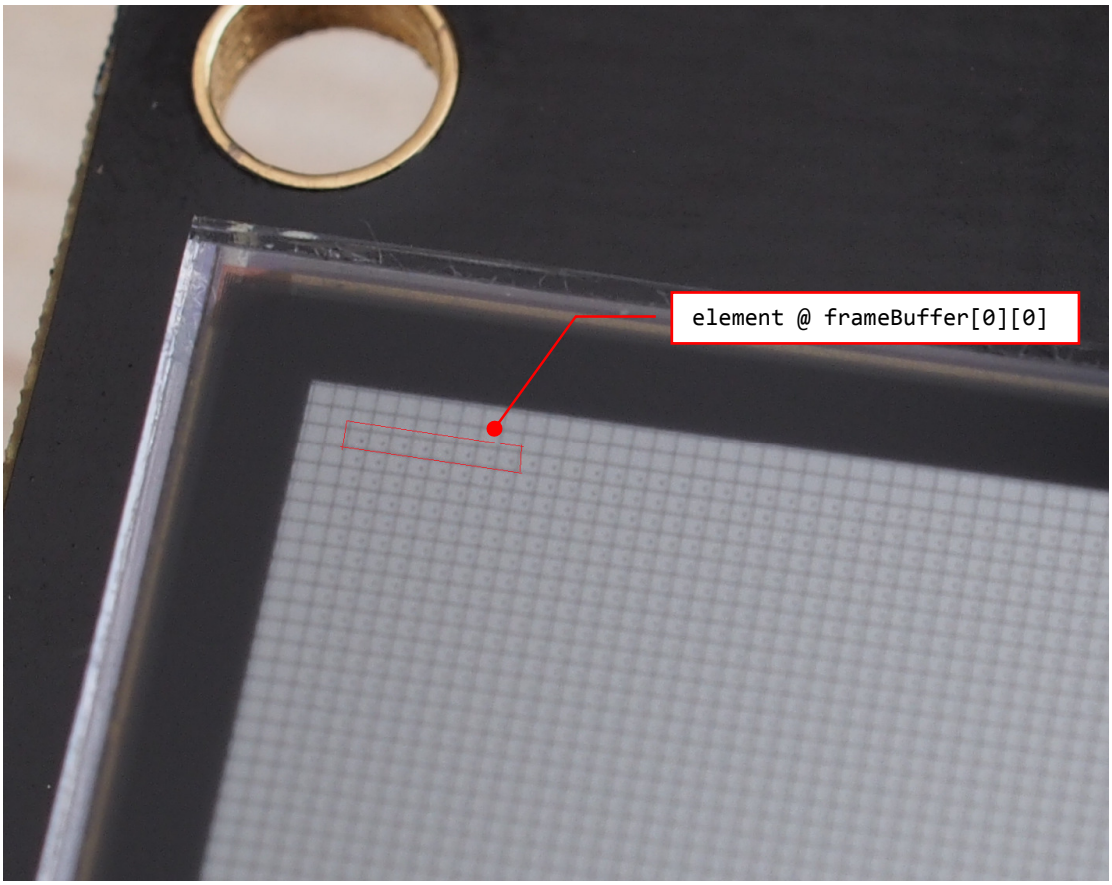
```
static void GFXDisplayUpdateLine(uint16_t line, uint8_t *buf);

static void GFXDisplayUpdateBlock(uint16_t start_line, uint16_t end_line, uint8_t *buf);
```

Frame buffer is declared as a double array to store all pixels of the LCD as below:

```
uint8_t frameBuffer[GFX_FB_CANVAS_H][GFX_FB_CANVAS_W];
//GFX_FB_CANVAS_H = Vertical resolution of the LCD
//GFX_FB_CANVAS_W = Horizontal resolution of the LCD /8
```

With a slightly better camera and a macro lens, we were able to visualize every pixel on a 4.4" Memory LCD as illustrated below . The outline in red color is a label for the first 8 pixels.



On system startup, all bytes in `frameBuffer[][]` array are initialized to 0xFF meaning all pixels set to white color. This is the first element `frameBuffer[0][0]` to store color content of the first 8 pixels in horizontal direction.

Now, suppose we need to set the odd pixels to black for the first 8 pixels (1,3,5,7), we may call the API function `GFXDisplayPutPixel(x,y,color)` with code snippet show below:

```
for (int x=0; x<8; x++)
{
	if(x%2)
		GFXDisplayPutPixel(x,0,WHITE);
	else
		GFXDisplayPutPixel(x,0,BLACK);
}
```

What's happening in the for-loop above is that, every time `GFXDisplayPutPixel()` is called, say `GFXDisplayPutPixel(0,0,BLACK);` it is the bit position at (0,0) set BLACK with the rest bit positions unchanged. To describe it in full details, memory content of the frame buffer array tabulated against each function call is shown on next page.

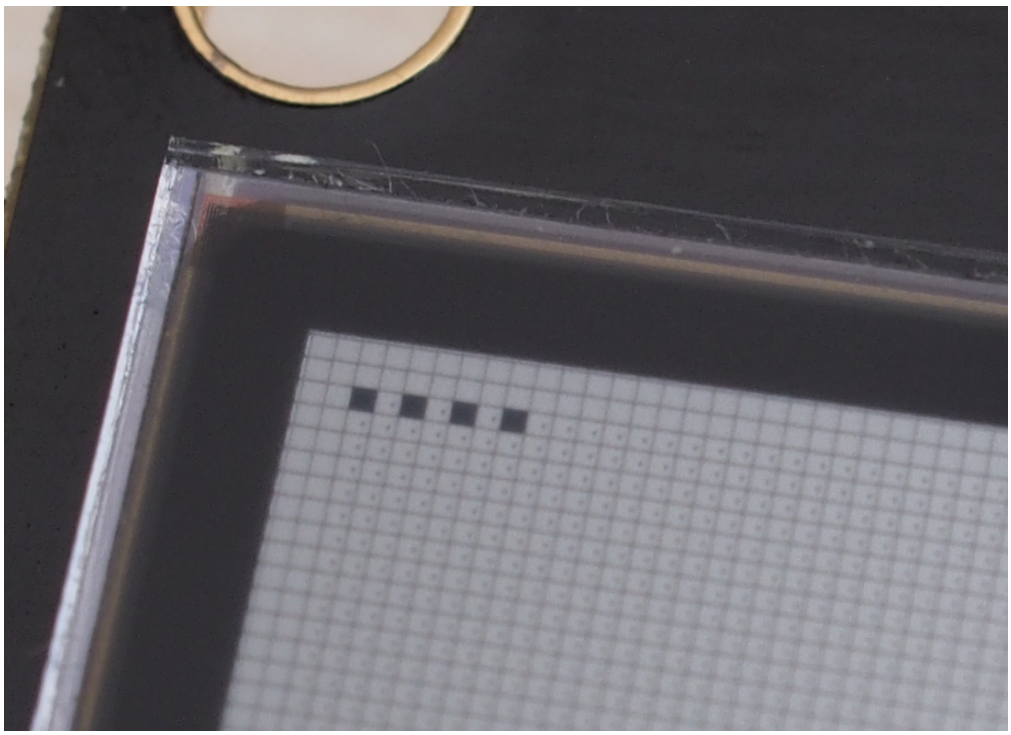
Function call	Frame buffer array for 4.4" LCD as an example. An underline at the bit-position shows the pixel to write.
GFXDisplayPutPixel(0,0,BLACK)	frameBuffer[0,0] = 0b0111 1111; frameBuffer[n,0] = 0b1111 1111; n=1-39
GFXDisplayPutPixel(1,0,WHITE)	frameBuffer[0,0] = 0b0111 1111; frameBuffer[n,0] = 0b1111 1111; n=1-39
GFXDisplayPutPixel(2,0,BLACK)	frameBuffer[0,0] = 0b0101 1111; frameBuffer[n,0] = 0b1111 1111; n=1-39
GFXDisplayPutPixel(3,0,WHITE)	frameBuffer[0,0] = 0b0101 1111; frameBuffer[n,0] = 0b1111 1111; n=1-39
GFXDisplayPutPixel(4,0,BLACK)	frameBuffer[0,0] = 0b0101 0111; frameBuffer[n,0] = 0b1111 1111; n=1-39
GFXDisplayPutPixel(5,0,WHITE)	frameBuffer[0,0] = 0b0101 0111; frameBuffer[n,0] = 0b1111 1111; n=1-39
GFXDisplayPutPixel(6,0,BLACK)	frameBuffer[0,0] = 0b0101 0101; frameBuffer[n,0] = 0b1111 1111; n=1-39
GFXDisplayPutPixel(7,0,WHITE)	frameBuffer[0,0] = 0b0101 0101; frameBuffer[n,0] = 0b1111 1111; n=1-39

In essence, what the function `GFXDisplayPutPixel(x,y,color)` doing is to set /clear the required bit position at (x,y) in frame buffer while keeping the rest bit positions unchanged. After bit set/clear is finished, the whole line will be updated by the local function `GFXDisplayUpdateLine()` for the complete horizontal line.

Interested readers may take a look at the source code for `GFXDisplayPutPixel()` listed below:

```
void GFXDisplayPutPixel(uint16_t x, uint16_t y, COLOR color)
{
	GFXDisplayPutPixel_FB(x, y, color);
	GFXDisplayUpdateLine(y+1, (uint8_t *)&frameBuffer[y]);
}
```

After running this code snippet the LCD is displaying something this:



To extend this concept to 2D, we may simply update a rectangular area with `GFXDisplayUpdateBlock()` to span it over the required top and bottom horizontal margin.

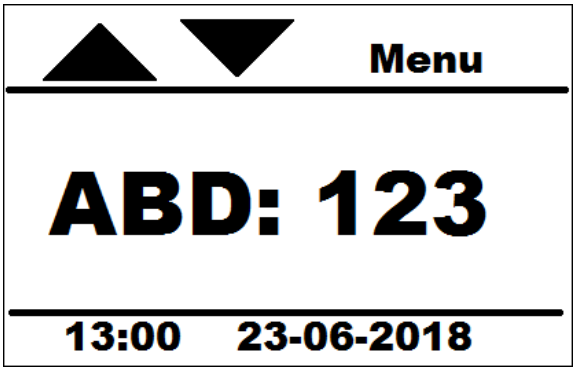
This implementation leads to a faster frame rate. Graphical interface illustrated in the `Arduino Sketch BloodPressure_GU.ino` shows a counting blood pressure reading at the left with a stood still icon of an up-arrow at the right. Feel free to open this sketch and change the delay constant in `loop()` from `delay(50)` to `delay(1),` or removing it to get an impression on how fast it can go.

Another unique features of Memory LCD is that partial update is allowed as long as it spans a complete horizontal region. **This means only the block occupying the font height of the SYS. pressure below is changed, whereas the top and bottom regions need not to be updated.**

A coin has two sides though. At the expense of a high frame rate with higher flexibility in GUI design, the down side of this approach is that we will need more expensive microcontroller with more SRAM memory to accommodate the frame buffer.

With better GUI planning there is a compromise. Consider an example below.

If we knew the top portion is not to be changed, we may copy graphical data from flash during system startup for the top region which will not be changed in the whole course of device usage. The lower part for numbers and date-time that need continuous update may be allocated a smaller frame buffer just enough for the area. This design approach leads to a wider choice of microcontrollers with fewer SRAM memory.



Chapter 6 Font creation

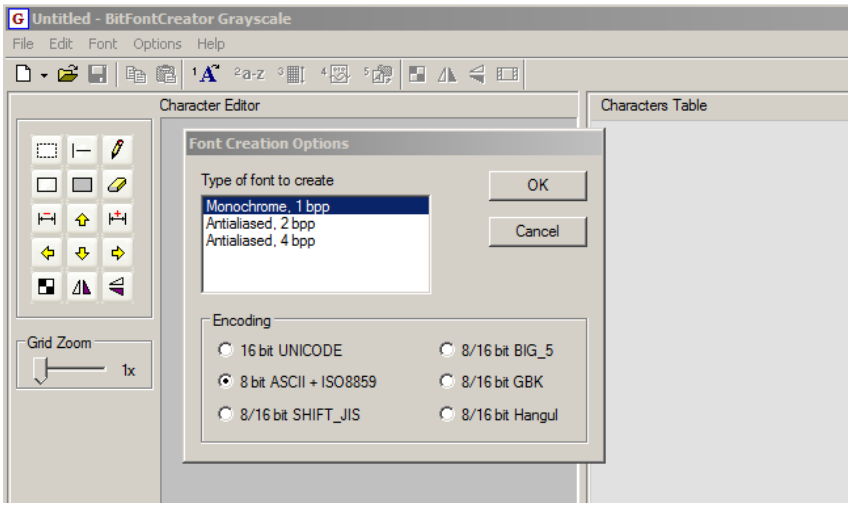
In this section we are going to introduce a commercial bitmap font creator tool (BitFontCreator) developed by a Beijing-based company called *Iseatech Software* (`http://www.iseasoft.com/company/about.html`). BitFontCreator converts any font from PC to C files for embedded projects.

There are three versions available namely BitFontCreator Latin, Pro, and Grayscale. The writer is using the Grayscale version but Latin version for 1-bpp font will be good enough for Memory LCD in Black/White color. Demo version is available from Iseasoft with a strikethrough on any character created. This limitation will be removed from formal version.

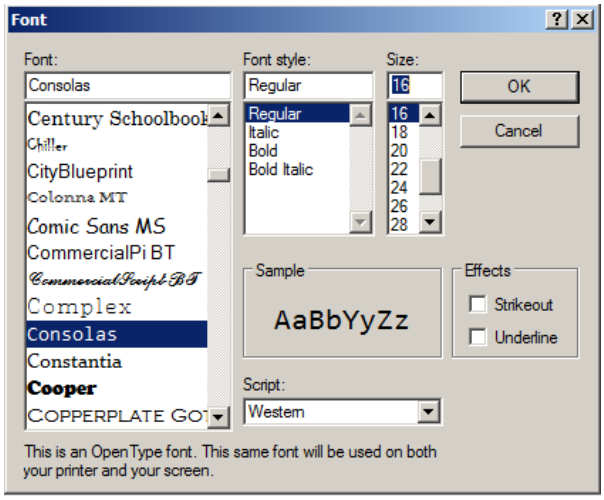
Procedures

Launch BFC, select `Monochome, 1-bpp` as the Font Creation Option. `Click OK.`

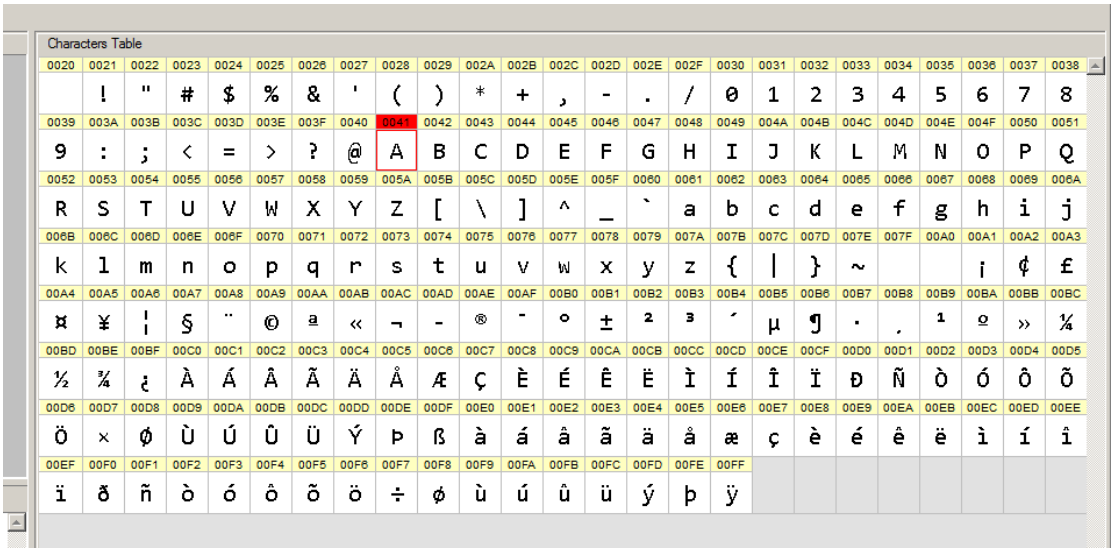
There are various encoding options available for Unicode, BIG5, and ASCII of course. If Unicode is selected, it will be possible to create C array from 16-bit characters for embedded projects. For now select 8 bit ASCII + ISO8859.



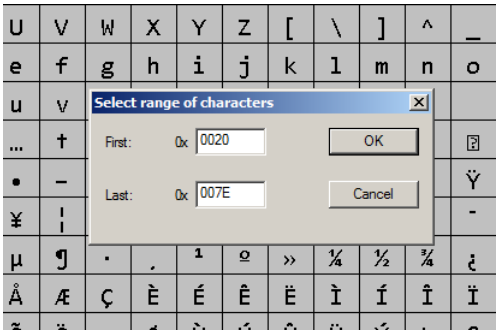
Select the font to use from your PC. Click OK.



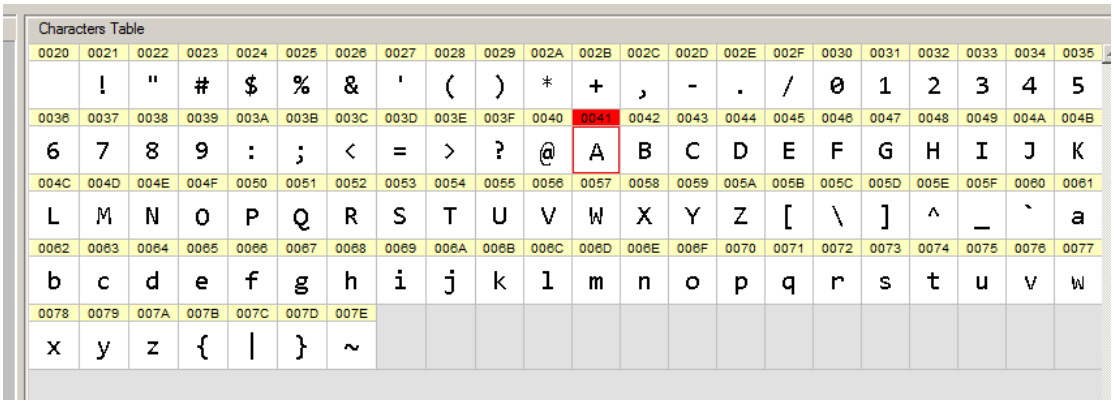
A Character Table is updated to show the default character range to use. To save flash memory it is possible to customize the range say, from ASCII code 0x0020 to 0x007E for ASCII characters from <space> to <~>.



Click `Font → Edit Characters Table (or Ctrl + T)`. Click `Disable All` and followed by `Enable Range`. Fill in 0x0020 for the First character and 0x007E for the Last character. `Click OK`.

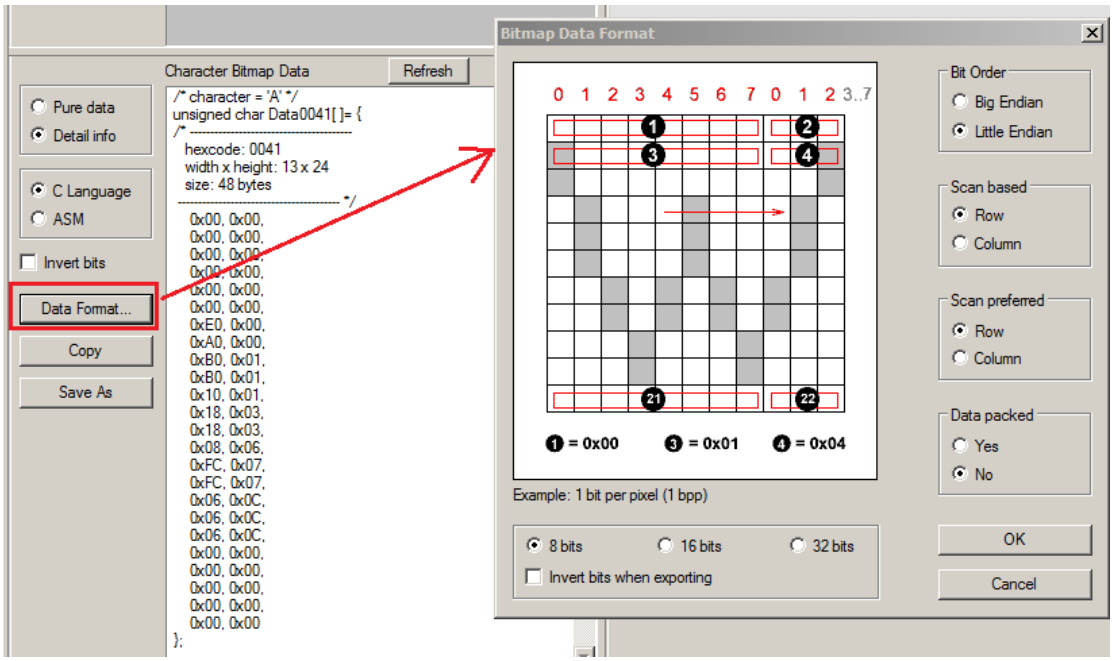


Now the Character Table will be updated with the range you have selected. A smaller character range just enough for application will create a smaller C array to save precious Flash space for microcontroller.



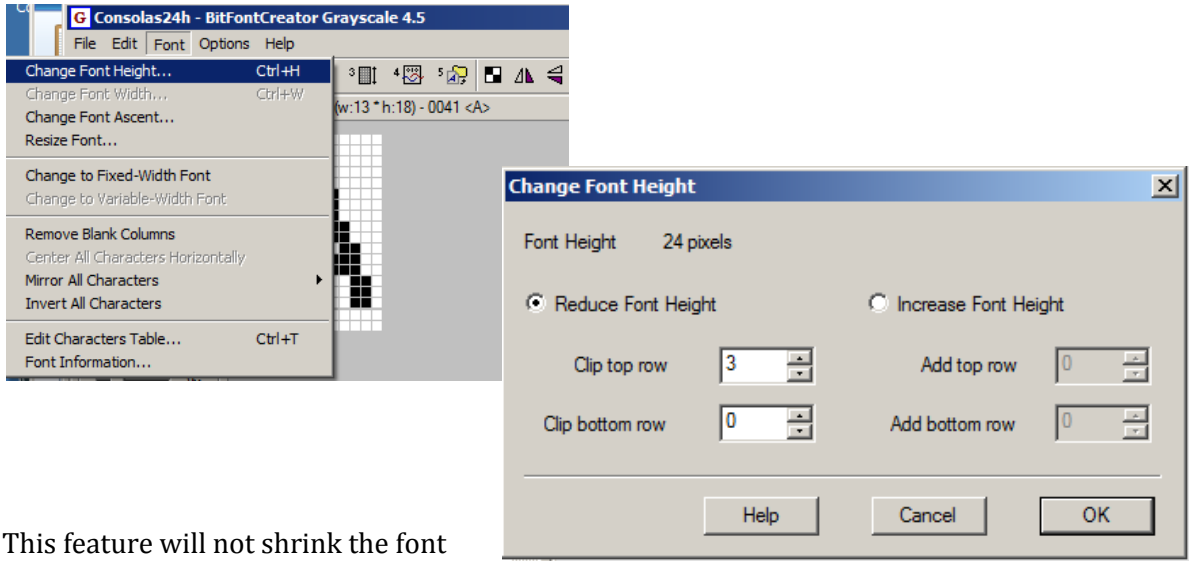
Click on `Data Format` button from the left panel, make sure the options are the same as screen shot below because the firmware has been designed for these options for data alignment.

Click `OK` to continue.



Now you may choose to save the project for future use. Using the default filename is OK here. Click `Save`. The format in *.foc is a proprietary format recognized only by BitFontCreator Suite.

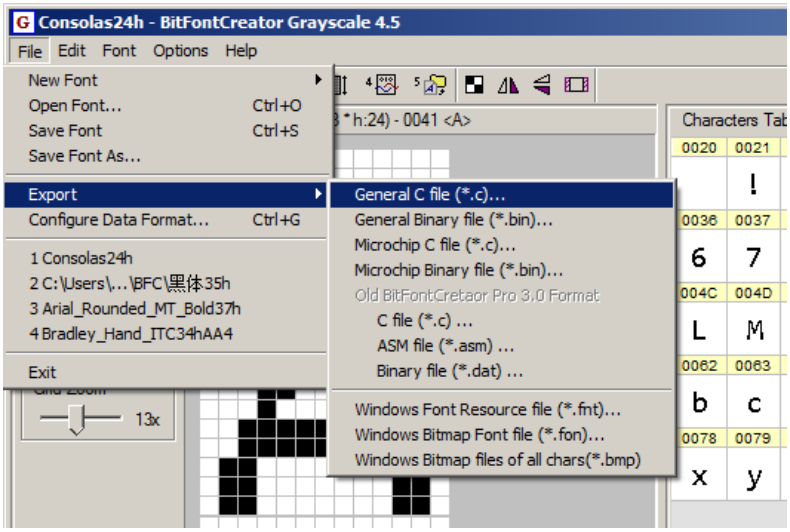
It is possible to application few modifications such as margin trimming, adding or removing some pixels, etc. To suit our application better the top margin has been trimmed by 3 pixels with feature from `Font → Change Font Height`.



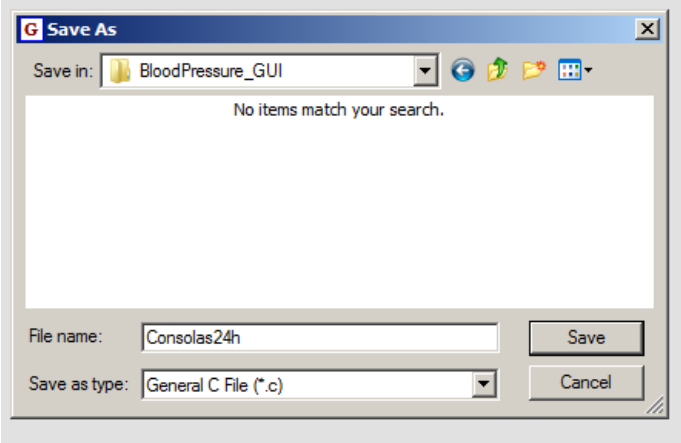
This feature will not shrink the font but only trim the top white space.

Finally, export a C file from drop down menu under

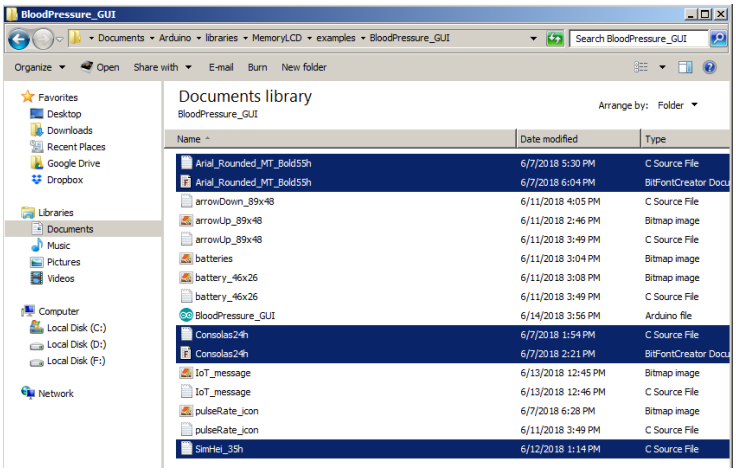
`File → Export → General C file (*.C).`



Save the C file to any location, preferably at the same location of the Arduino project folder. It is under `\BloodPressure_GUI` folder in our case.



Similar procedures can be performed with other fonts with project folder look like this:



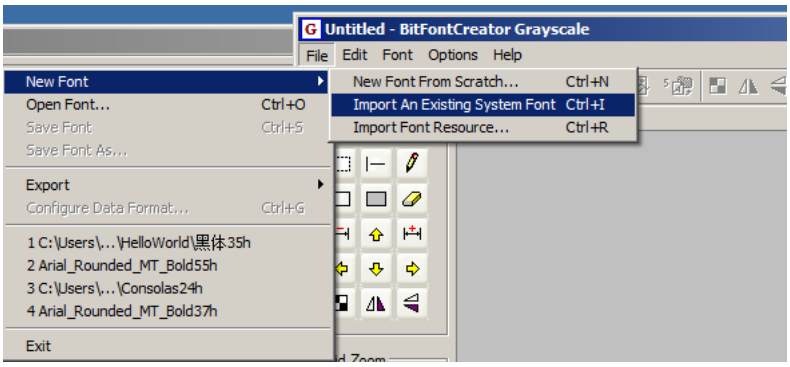
Only the C files and Arduino .ino file are useful for programming. Those *.foc files serve as references only.

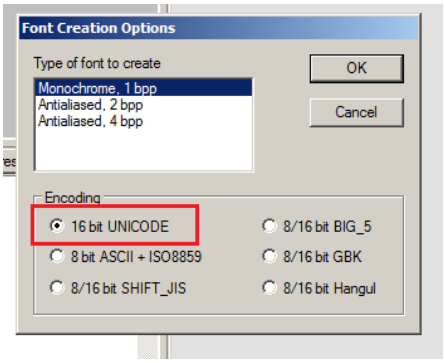
Supporting Unicode

A similar procedure can be used to support Unicode (`http://www.unicode.org/standard/standard.html`).

Procedures in this section describe how BitFontCreator is used to convert simple greeting messages in Chinese and Japanese to C files for the Arduino Sketches BloodPressure_GUI.ino and HelloWorld.ino.

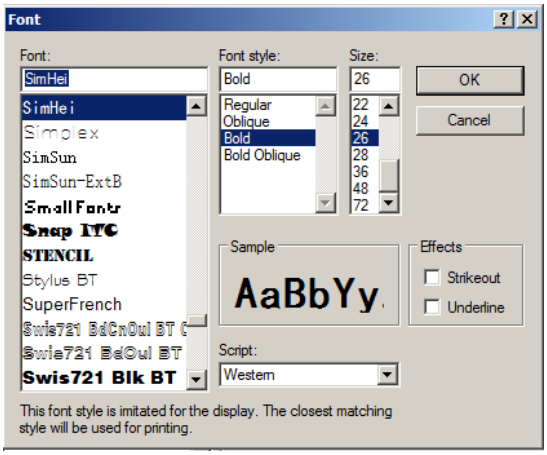
Under `File→New Font→Import An Existing System Font`, select `Monochrome, 1bpp` with Encoding set `16 bit UNICODE`. Click OK.



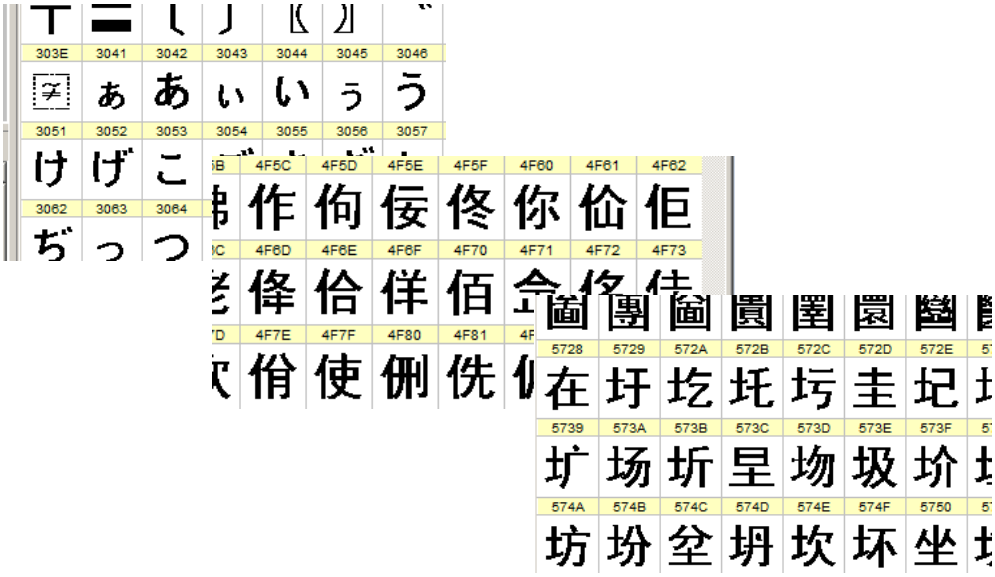


Pick any of the system Font with Unicode support, in our case the SimHei of Bold 26 selected.

`Click OK.`

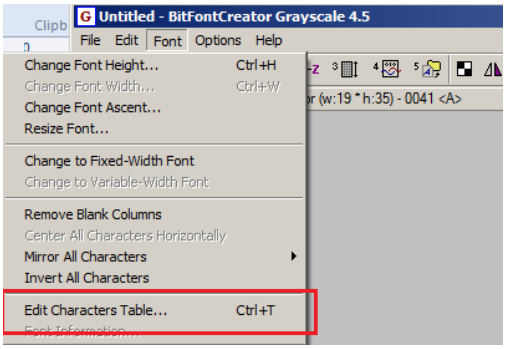


It may take a while for your PC to import the map into BitFontCreator. After successful import the whole character table will be available. Browse it through the end you will see some characters in Japanese and Chinese.



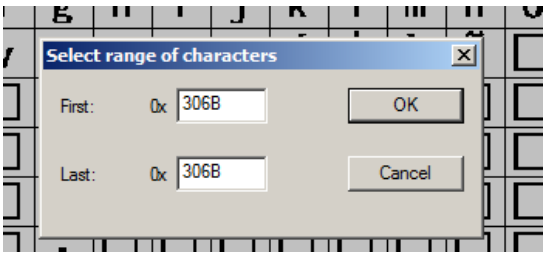
Although it is possible to include the whole character range from 0x0020 to 0xFFE5, another consideration is that flash memory of microcontroller will be eaten up by the static font data if we are doing so. It is more practical to include only the characters we need to save our precious microcontroller flash area.

Bring up the `Font→Edit` Characters Table.

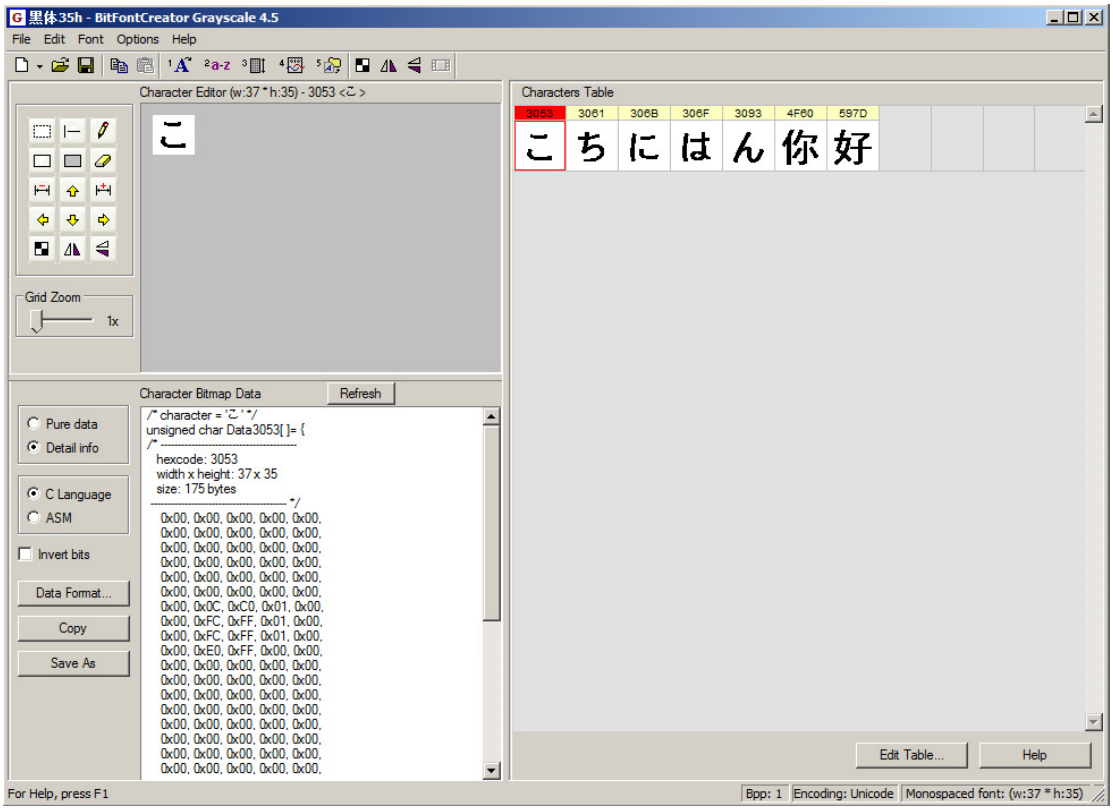


Click `Disable All` followed by `Enable Range`. Input the range to use, in our case only few characters are required for greeting in Japanese and Chinese. They are Unicode

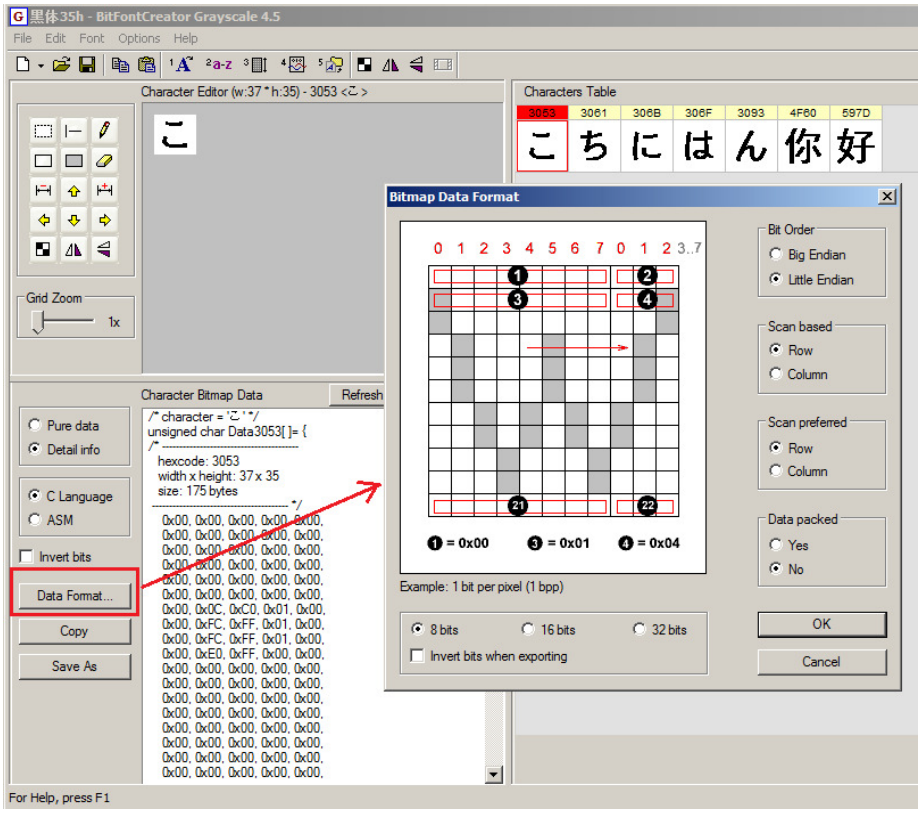
`0x3053, 0x3061, 0x306B, 0x306F, 0x3093, 0x4F60, & 0x5970.`



After repeating the selection procedure for individual characters the Characters Table will be filled up with just what we need.

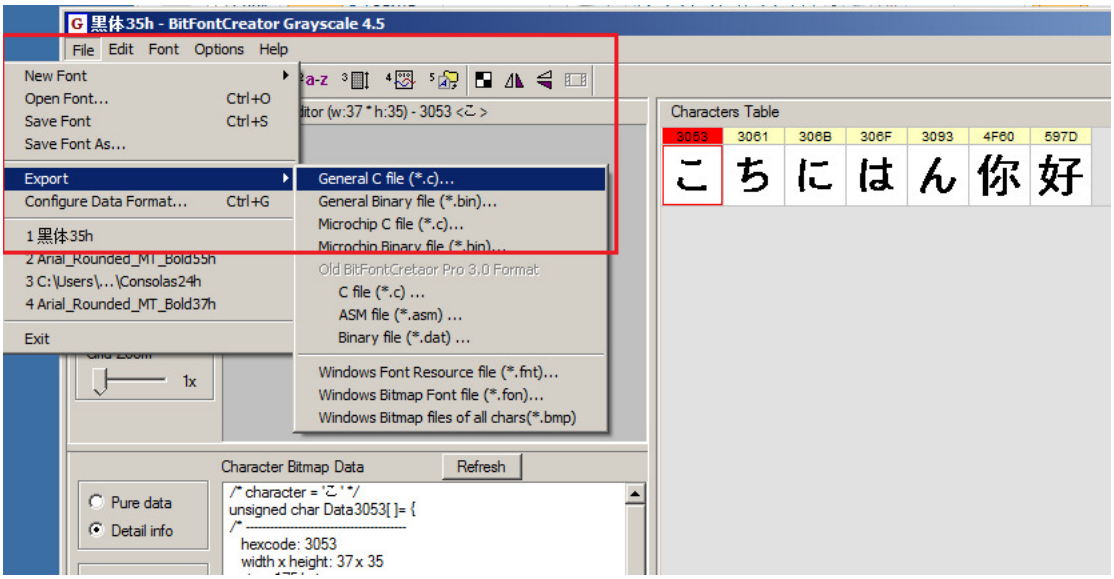


Bring up the Bitmap Data Format window to make sure settings as below. `Click OK.`



Now, click General C file (*.c) from

`File→Export→General C file (*.c)...`



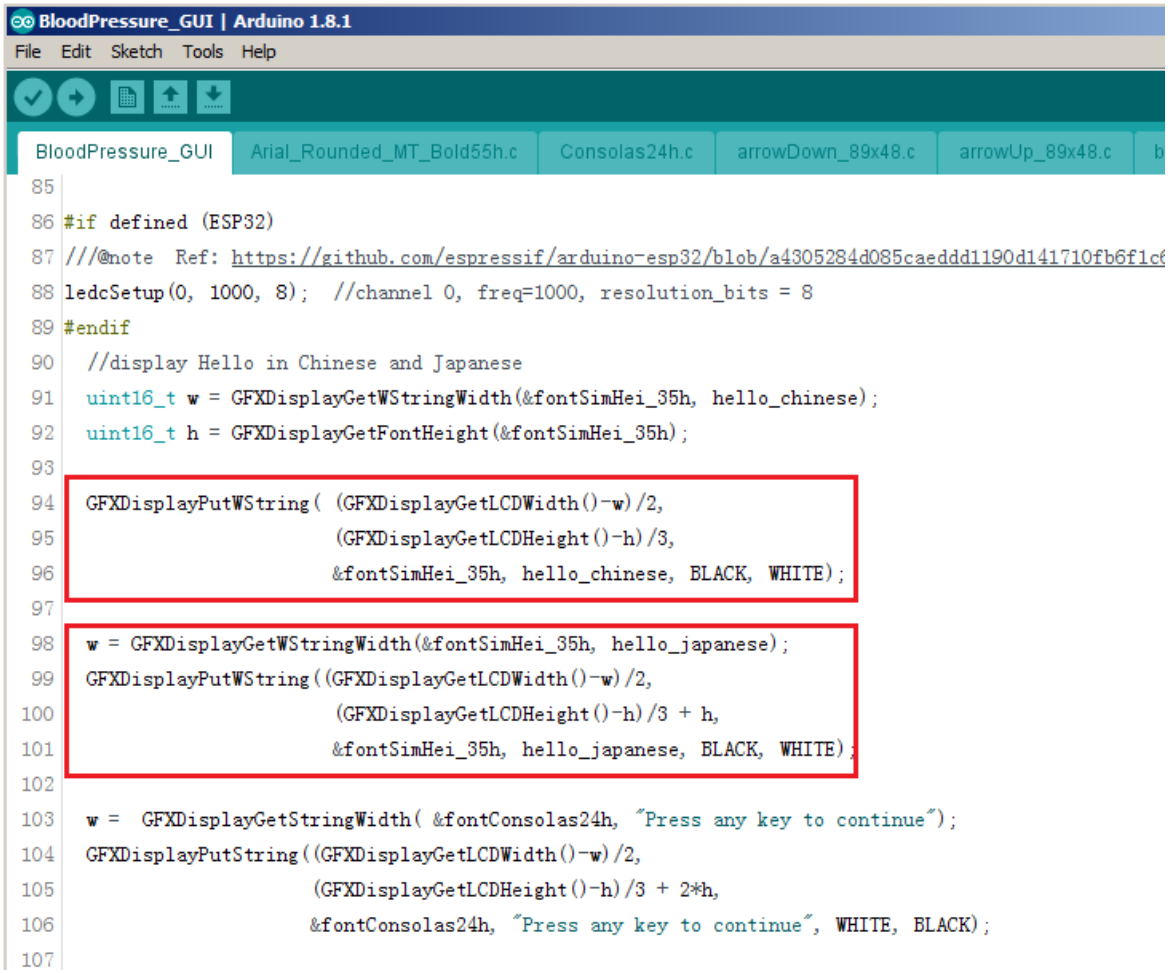
Save the file, given the name as `SimHei_35h.c` in our case.

To use the font created we need to "*let Arduino know*" that we have such data in our system. Now declare its existence in source code. Snippet shown below.

```
17 extern const BFC_FONT fontArial_Rounded_MT_Bold55h;
18 extern const BFC_FONT fontSimHei_35h;
19 extern const tImage cat_400x246;
20 extern const tImage qr_code_248x248;
21
22 uint8_t SW3_demo_page = 0;
23 bool demo_done = false;
24
25 ///@note Font: SimHei 35 こんにちは in unicode 16
26 const uint16_t hello_japanese[]={0x3053, 0x3093, 0x306B, 0x3061, 0x306F, '\0'};
27 ///@note Font: SimHei 35 你好 in unicode 16
28 const uint16_t hello_chinese[] ={0x4F60, 0x597D, '\0'};
29
```

Similarly for `fontArial_Rounded_MT_Bold55h` and `fontConsolas24h` the pattern for declaration is the same.

After announcing their existence the last step is to call the relevant functions to print the texts. Its usage can be found in the source code `BloodPressure_GUI.ino` and HelloWorld.ino from the example folder.



Chapter 7 Image creation

In this section we are going to show you a free utility to create image and font source files for embedded projects, namely LCD Image Converter. This tool's home page: http://www.riuson.com/lcd-image-converter.

From time to time we need to display some icons such as battery level, an arrow for navigation, etc.

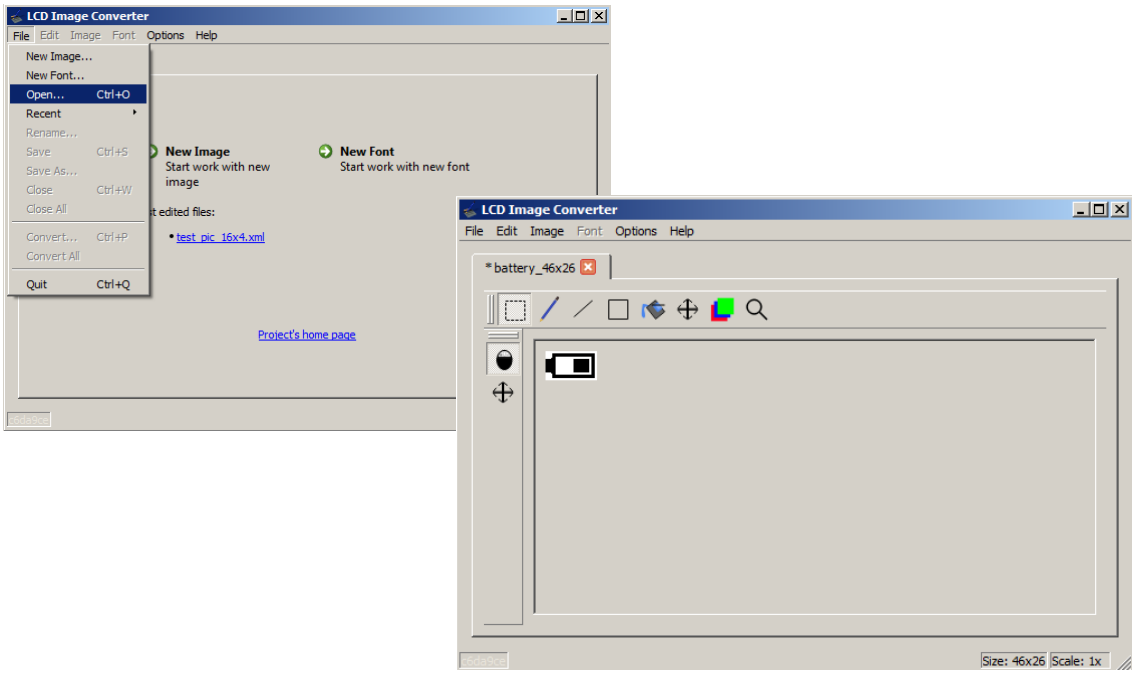
LCD Image Converter is an open source program that can be downloaded for free whereas BitFontCreator described in last chapter is a commercial program at an affordable cost. The purpose of using different programs to convert graphical assets (fonts and bitmaps) is to show flexibility.

There is no restriction on which tool to use as long as we are drawing pixels on LCD correctly from arrays, no matter they have been converted by a commercial program or shareware.

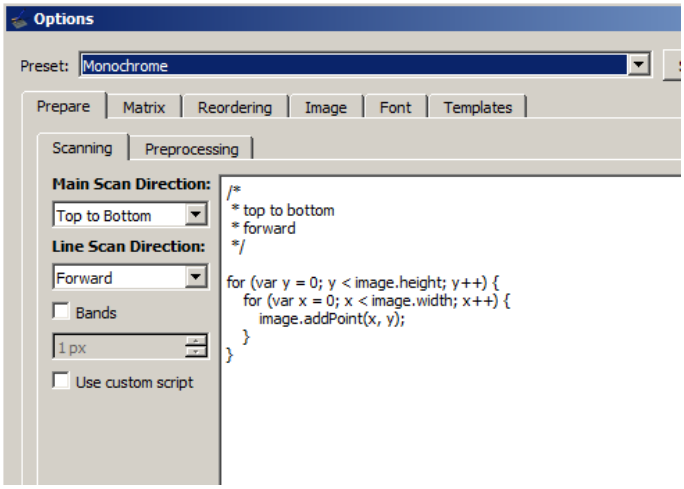
Here we need to convert 3 icons to C arrays for our embedded project. They can be found in the project folder of BloodPressure_GUI.ino

Launch `lcd-image-converter.exe`.

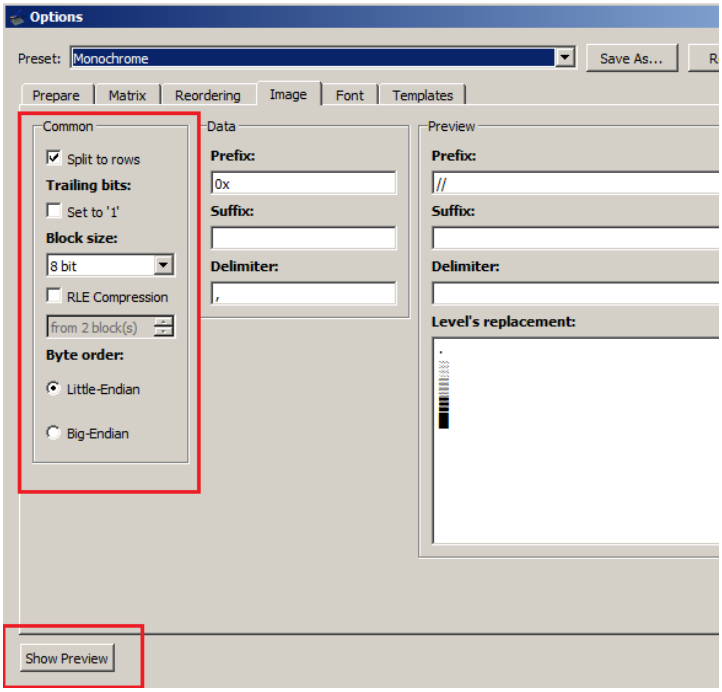
Start working with `File → Open`. Browse to the image to convert.



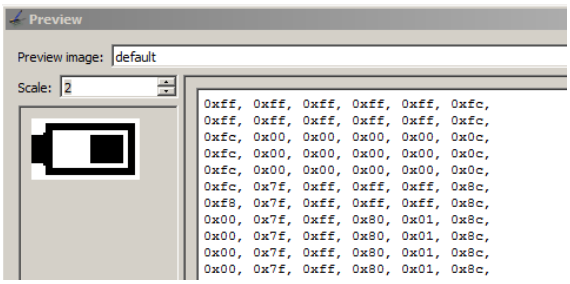
From `Options→Conversion`, make sure the **Main Scan Direction** and **Line Scan Direction** have been set to **Top to Bottom** and **Forward**, respectively.



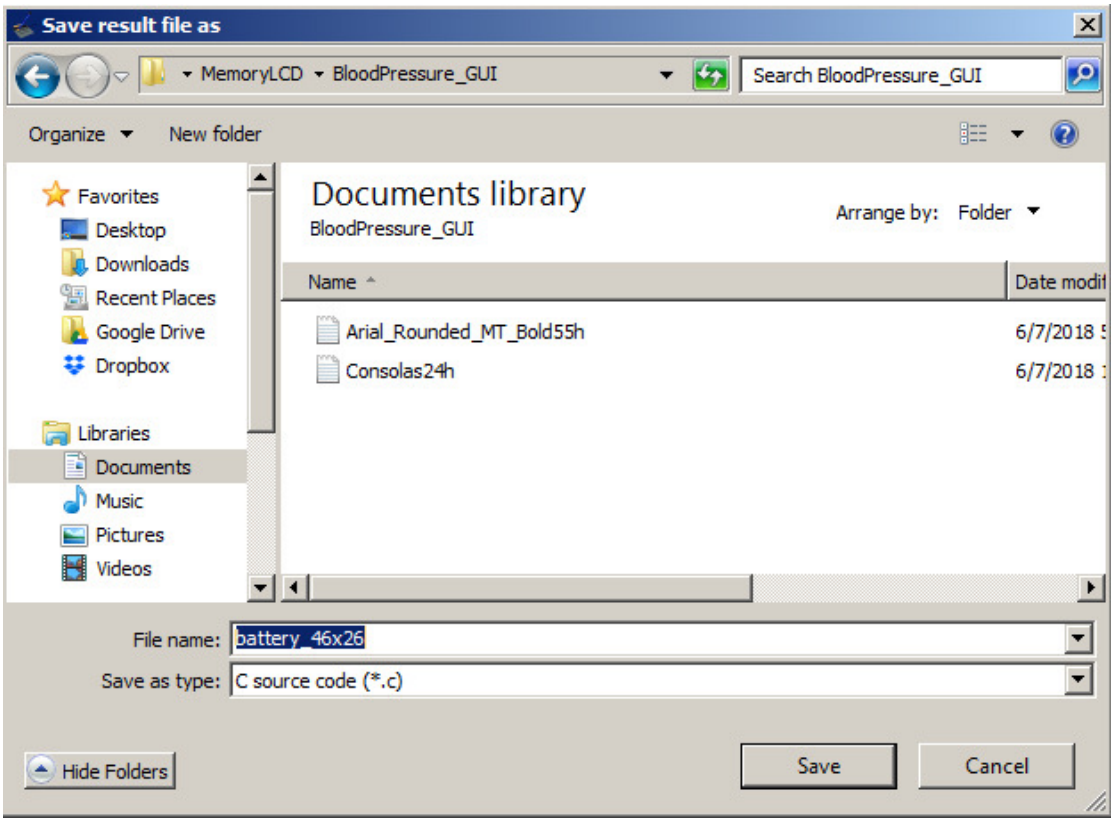
Under Image tab, select **Split to rows** checkbox with **Block Size** set to 8 bit, and Byte order set to **Little-Endian**.



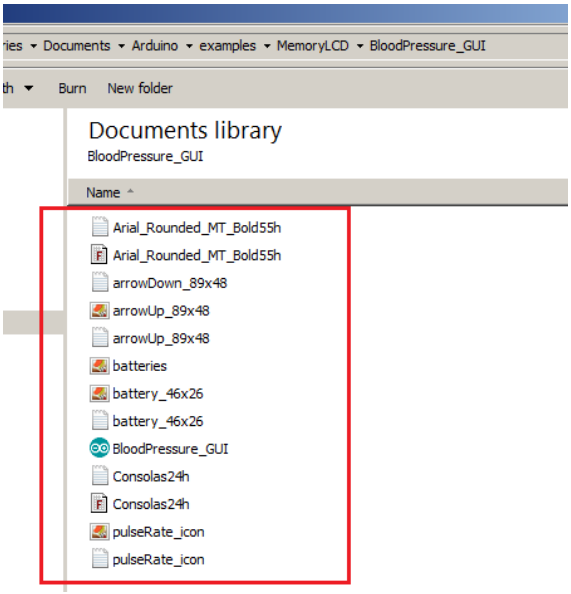
You may click the **Show Preview** button to preview the C array. Click **OK** and **Yes** to save change.



Now click Convert under `File→Convert… (Ctrl+P)` to export the *.c file to the root directory of your project.



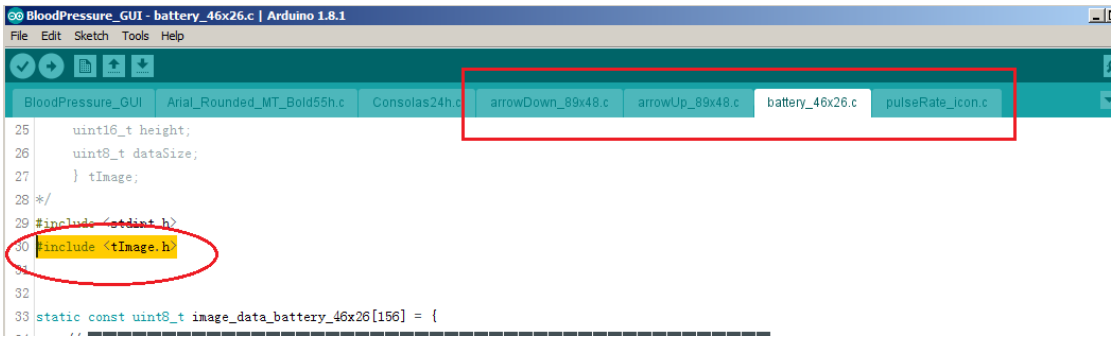
By following the same procedures two files `arrowUp_89x48.c` and `pulseRate_icon.c` were created. Screen shot of the sketch project folder is shown below for reference.



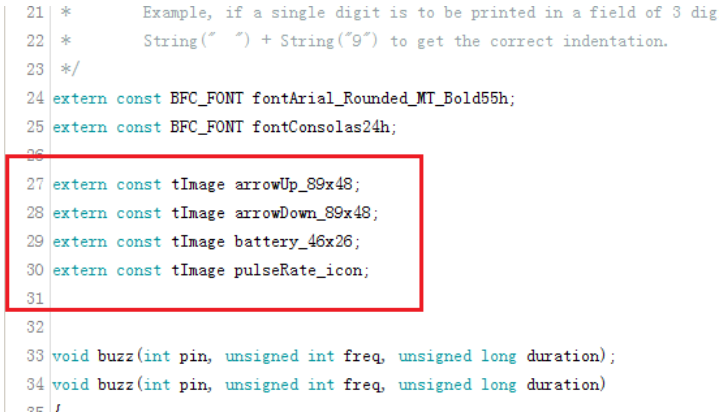
Please notice that the C file `arrowDown_89x48.c` was converted with the up arrow image flipped in LCD Image Converter with keyword *arrowUp_89x48* replaced by *arrowDown_89x48* manually in a text editor.

Launch Arduino IDE and open the sketch `BloodPressure_GUI.ino`, from the IDE all C files are show in different tabs.

There is one last step. Complete the C files created with a new header `#include <tImage.h>` for everyone of them.



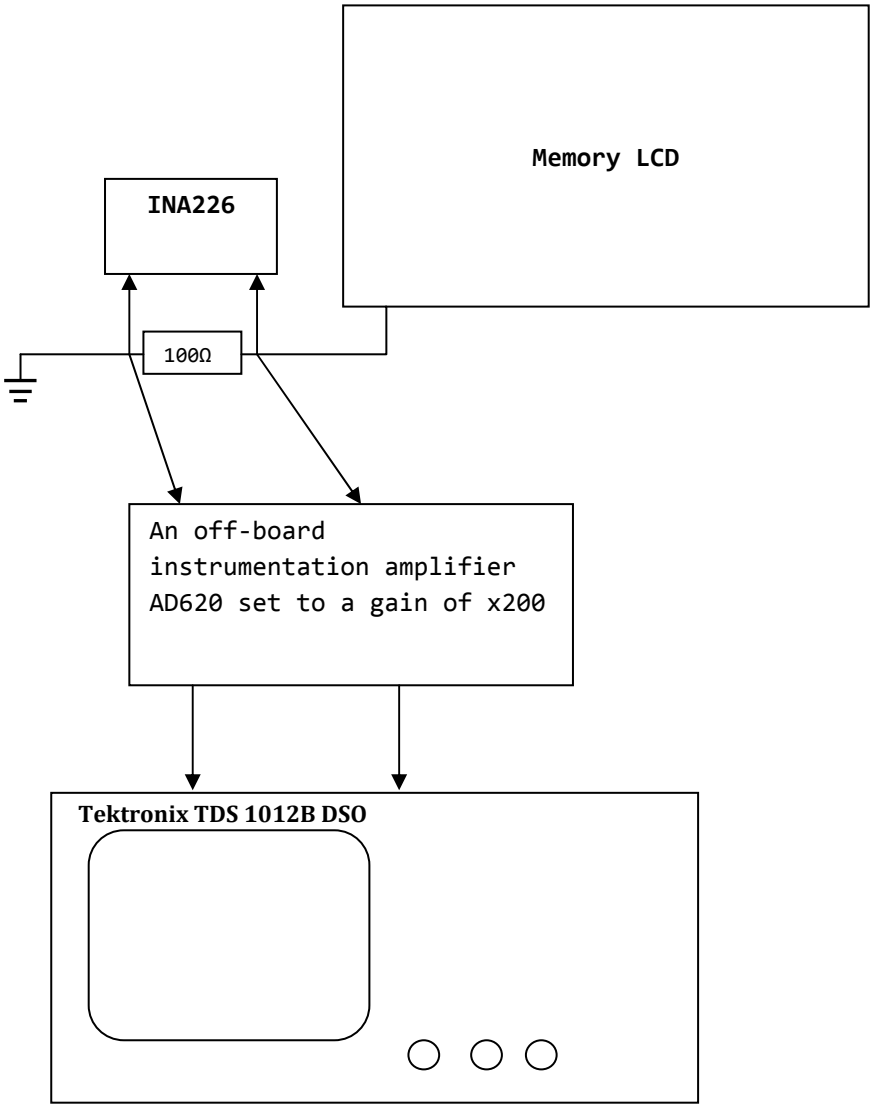
Now in the sketch file, *let the project know* we are using those external data by declaration with `extern` meaning that we are declaring the data somewhere outside.



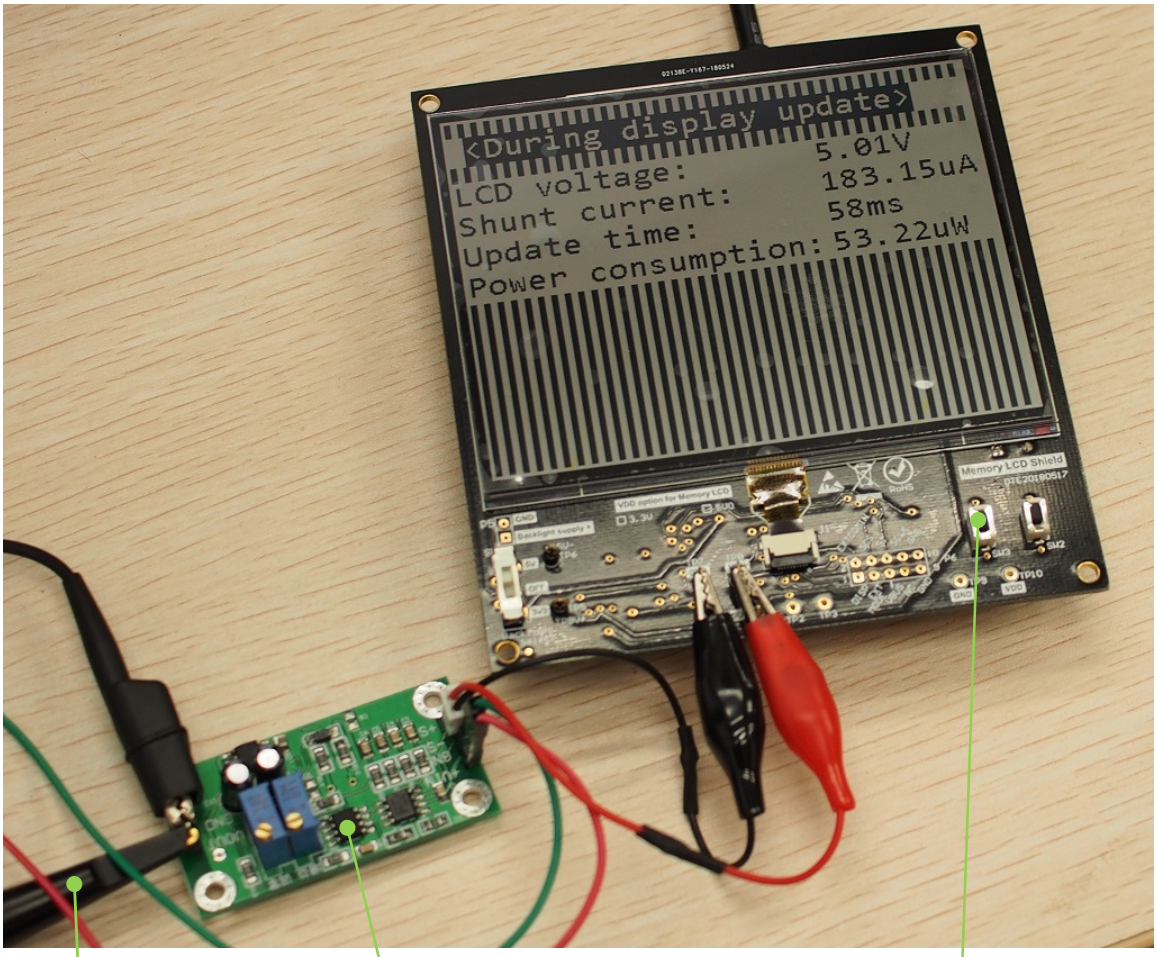
We may now display icons by calling the function `GFXDisplayPutImage()`.

Chapter 8 An estimation on power consumption

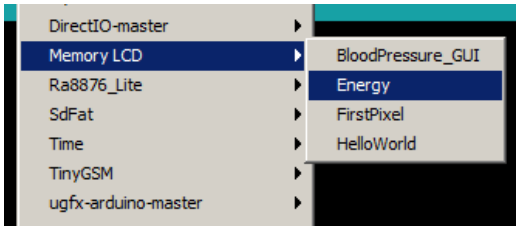
This section attempts to analyze the power consumption of a Memory LCD when it's content is maintained and updated. Although such data has been stated in each of the datasheets of various LCD models, a full understanding on when and how power is spent would make sense for future low power design. Block diagram below shows the functional blocks for measurement.



Photography on next page shows the set up in action.



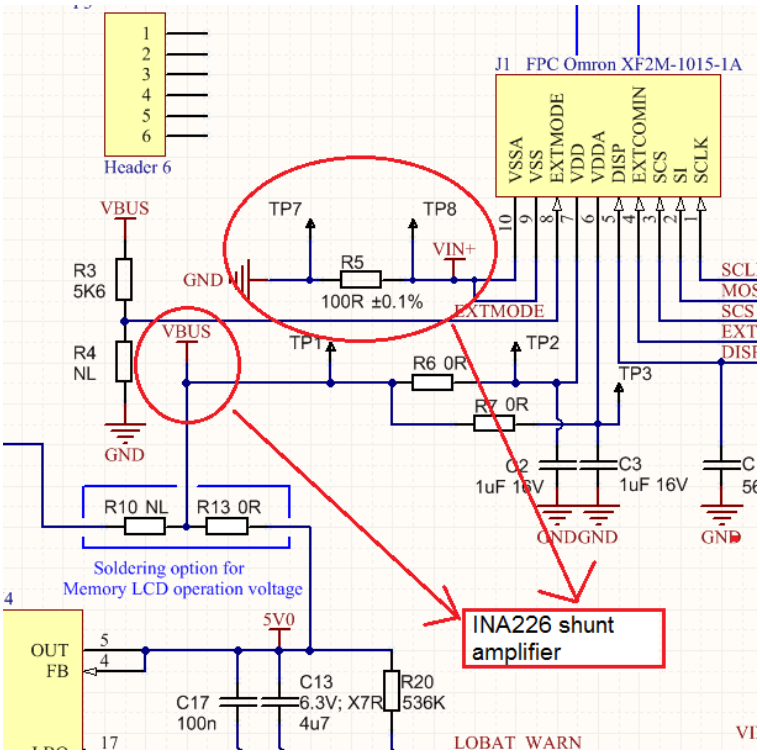
Run this sketch from `File→Examples→MemoryLCD→Energy`. Press SW3 to start measurement.



Energy measurement is performed by a function `void ina226_measure(void)` with listing shown below.

```
void ina226_measure(void)
{
  busVoltage_V = (float)ina.readBusVoltage(); //in V
  shuntVoltage_uV = (float)ina.readShuntVoltage()*1000000.0;     //in uV
}
```

The variable `busVoltage_V` returns the operation voltage of the Memory LCD. This value is measured from the output of TPS60140 DC-DC which got tethered to VDD & VDDA of Memory LCD via R13 (0 Ω). Current is measured from a shunt resistor of R5 (100Ω±0.1%) in the return path of the LCD to ground. An extract of the schematic is shown below:



It has been found that the time to make measurement is critical. The power consumption during display update(dynamic) to that of display maintain(static) is different.

To tackle this problem a function dedicated for power measurement during display update is created.

Listing on next page shows the function

`uint32_t GFXDisplayTestPattern(uint8_t pattern, void (*pfcn)(void)).`

```
uint32_t GFXDisplayTestPattern(uint8_t pattern, void (*pfcn)(void))
{
        uint32_t timing = 0;
#if defined (ARDUINO)
        uint32_t sMillis = millis();
#else
        #error Need to define the function to return millisec for other platforms
#endif
  hal_spi_start_transaction();
  hal_delayUs(3); //SCS setup time of tsSCS (refer to datasheet for timing details)

  for(uint16_t line=1; line<=DISP_VER_RESOLUTION; line++)
  {
        #ifdef LS032B7DD02

        hal_spi_write_byte(uint8_t((line<<6)|0x01));
        hal_spi_write_byte((uint8_t)(line>>2));
        #else
        hal_spi_write_byte(0x01);
        hal_spi_write_byte((uint8_t)line
        #endif

        uint32_t writePeriod = DISP_HOR_RESOLUTION>>3; //divide by 8 for 1-bit bpp

        while(writePeriod--){
        hal_spi_write_byte(pattern);
        }

        if(line==DISP_VER_RESOLUTION/2)
        {
        if(pfcn!=NULL) {
        pfcn();
        }
        }
  }
        hal_spi_write_byte(0x00); //dummy byte
        hal_spi_write_byte(0x00); //dummy byte
        hal_spi_end_transaction();

#if defined (ARDUINO)
        timing = millis()-sMillis;
#else
        #error Need to define the function to return millisec for other platforms
#endif

        return timing;
}
```

There are two arguments required to run this function:

1. `uint8_t pattern :` defines the horizontal bit pattern in 8-bit width so that the whole horizontal line spanning across the LCD will repeat in this pattern for the vertical direction.

2. `void (*pfcn)(void) :` is a pointer to function to execute when horizontal line is updated at half the height of the LCD. This is controlled by the snippet below.

```
...
        if(line==DISP_VER_RESOLUTION/2)
        {
        if(pfcn!=NULL) {
        pfcn();
        }
        }
  }
...
```

The time taken to update vertical strip pattern is returned in milliseconds.

When switch SW3 is pressed, the code will check for a simple software delay of 10ms to debounce the key and print a short message via Serial Monitor as "Energy measurement when display is updated ::". Then the function `GFXDisplayTestPattern(0xF0, &ina226_measure)` will be executed with the time taken returned with variable `updateTime`.

Current is calculated by simple Ohm's Law:

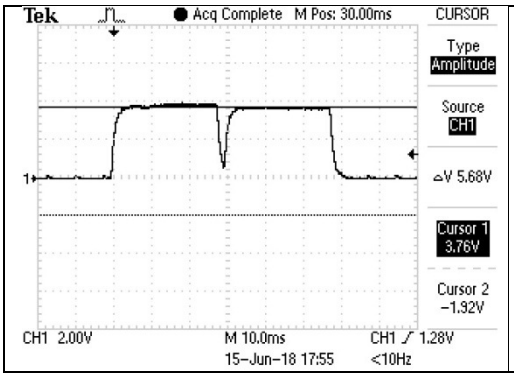
```
shuntCurrent_uA = shuntVoltage_uV/100;  //with a shunt resistor 100Ω
```

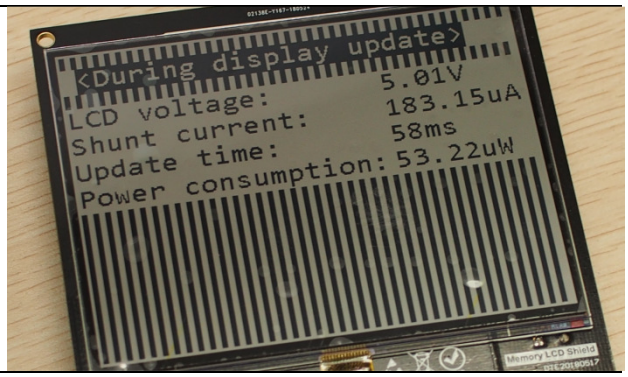
Power is calculated not from a direct equation P=V*I. A time factor has been included to take account of the fact that only a finite time interval is required to display the vertical strip pattern.

```
power_uW = shuntCurrent_uA*busVoltage_V*updateTime/1000; //update time in ms
```

For full listing readers may check the source code `Energy.ino` Arduino Sketch.

The accuracy was verified with a DSO (Tektronix TDS1012B). An instrumentation amplifier AD620 set to a gain of x200 was used to boost the shunt voltage so that it could be measured by TDS1012B. Data comparing measurements made by the external DSO to that of INA226 shunt amplifier onboard of the Memory LCD Shield are shown below.





Don't forget the signal fed to CH1 has been amplified by x200 gain factor.

Therefore, the true value measured by DSO = 3.76V/200 = 0.0188V = 18.8mV.

This agree quite well with INA226's result.

Time taken to update the vertical strip pattern was around 58ms as measured by the DSO. This result matches very well with the parameter returned by software. But hold! How come there was a dip in the middle? This was caused by INA226's ADC sampling on VIN- and VIN+. Occurring in the middle because we have programmed the measurement to be taken when the line number was equal to `DISP_VER_RESOLUTION/2` which is the half vertical height of the display.

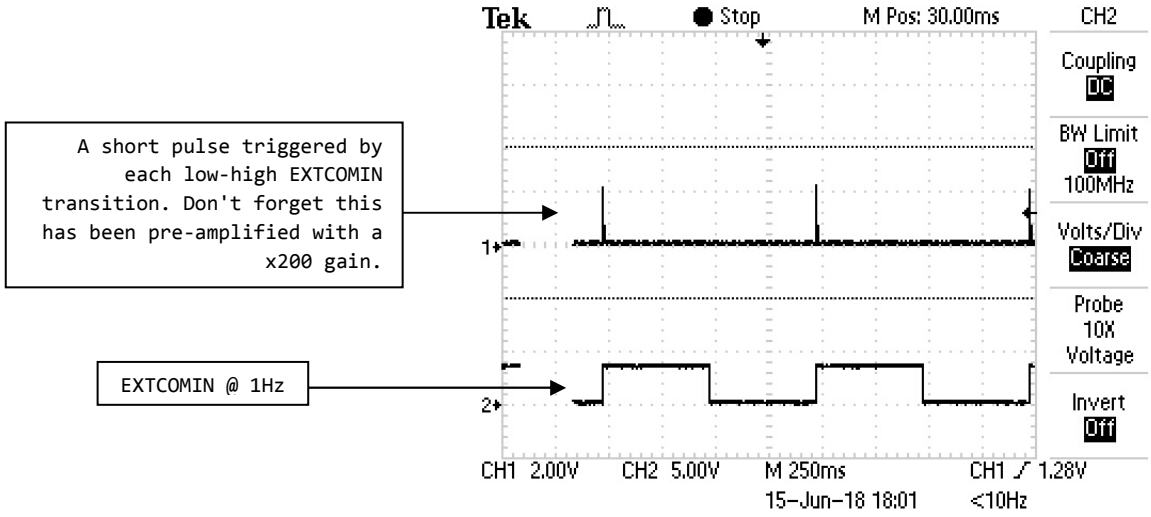
Putting together the LCD voltage, shunt current, and the time taken to update,

$Power_{update}$ = 5.01V * 183.15uA * 58/1000 = 53.22uW.

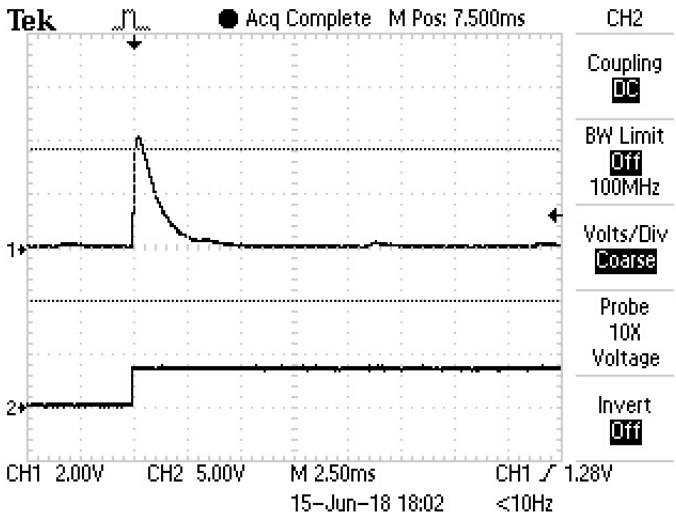
That's it when Memory LCD is being updated. But what about when it is not? When an image is maintained, which happens 90% of the time for modern GUI, a Memory LCD drains significant power only when the EXTCOMIN pin goes from low to high in its perpetual external COM inversion signal at 1Hz.

DSO trace below shows the same trace at CH1 with CH2 measuring EXTCOMIN.

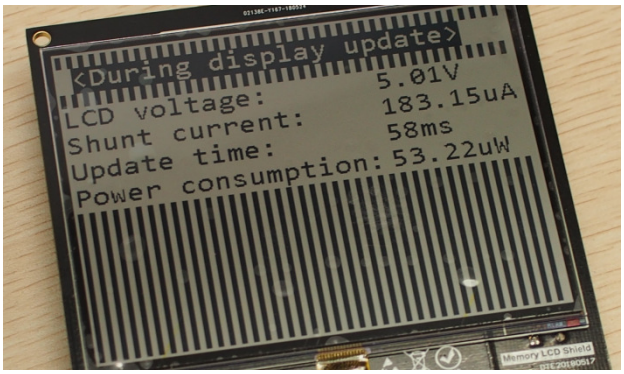
Again, don't forget CH1 was amplified at x200 gain.



A closer look at CH1 reveals a pulse of 2.50ms peaks at around 4/200 V (20mV) when EXTCOMIN goes from low-to-high. For the rest of 997.5ms in a 1 second interval the shunt voltage was not measurable even with the help of a x200 pre-amplifier!



This is equivalent to say that, power consumption when the Memory LCD is maintaining an image like this:



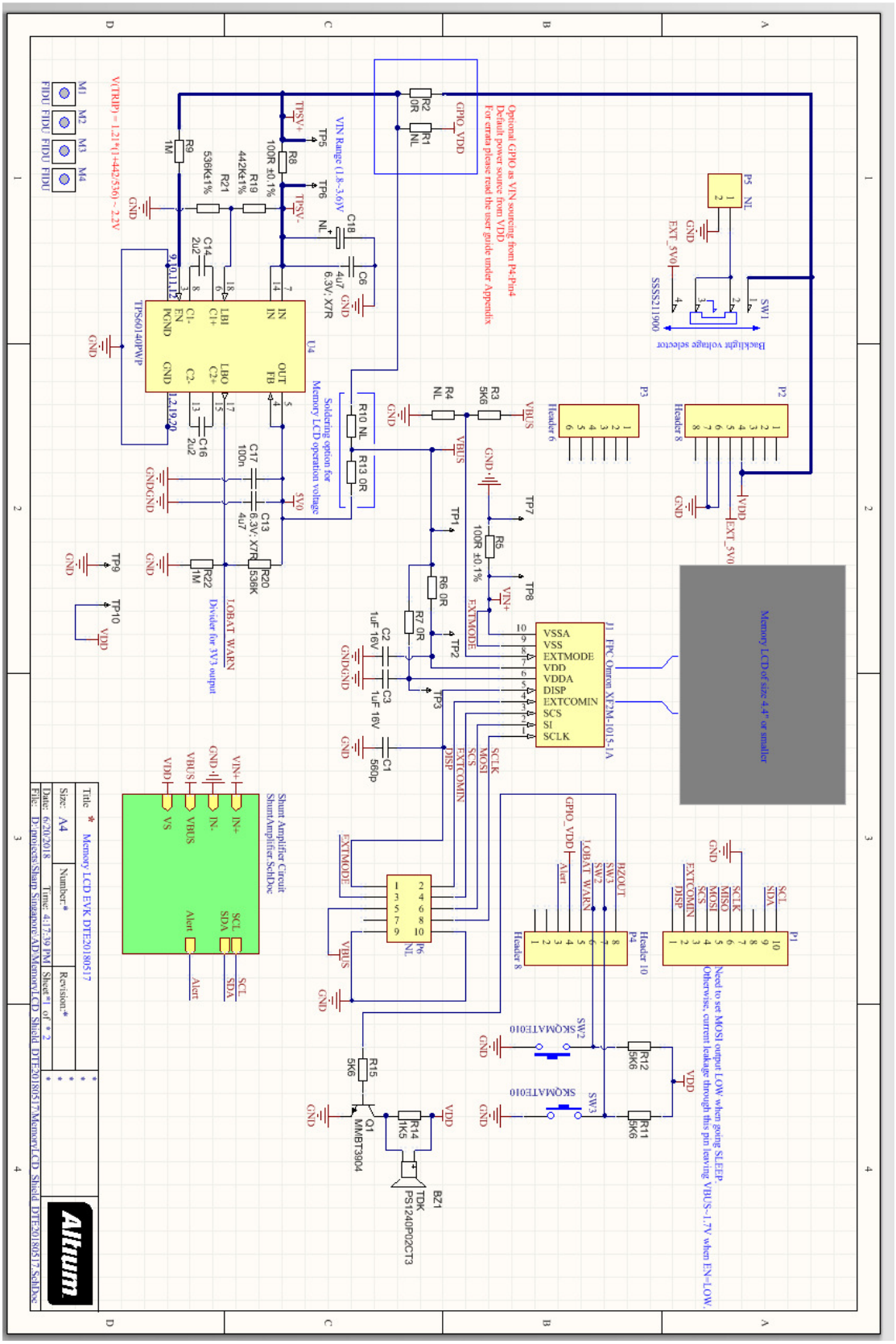
showing the vertical strip we have updated when SW3 was pressed, plus all readings covering part of the strip pattern is calculated as

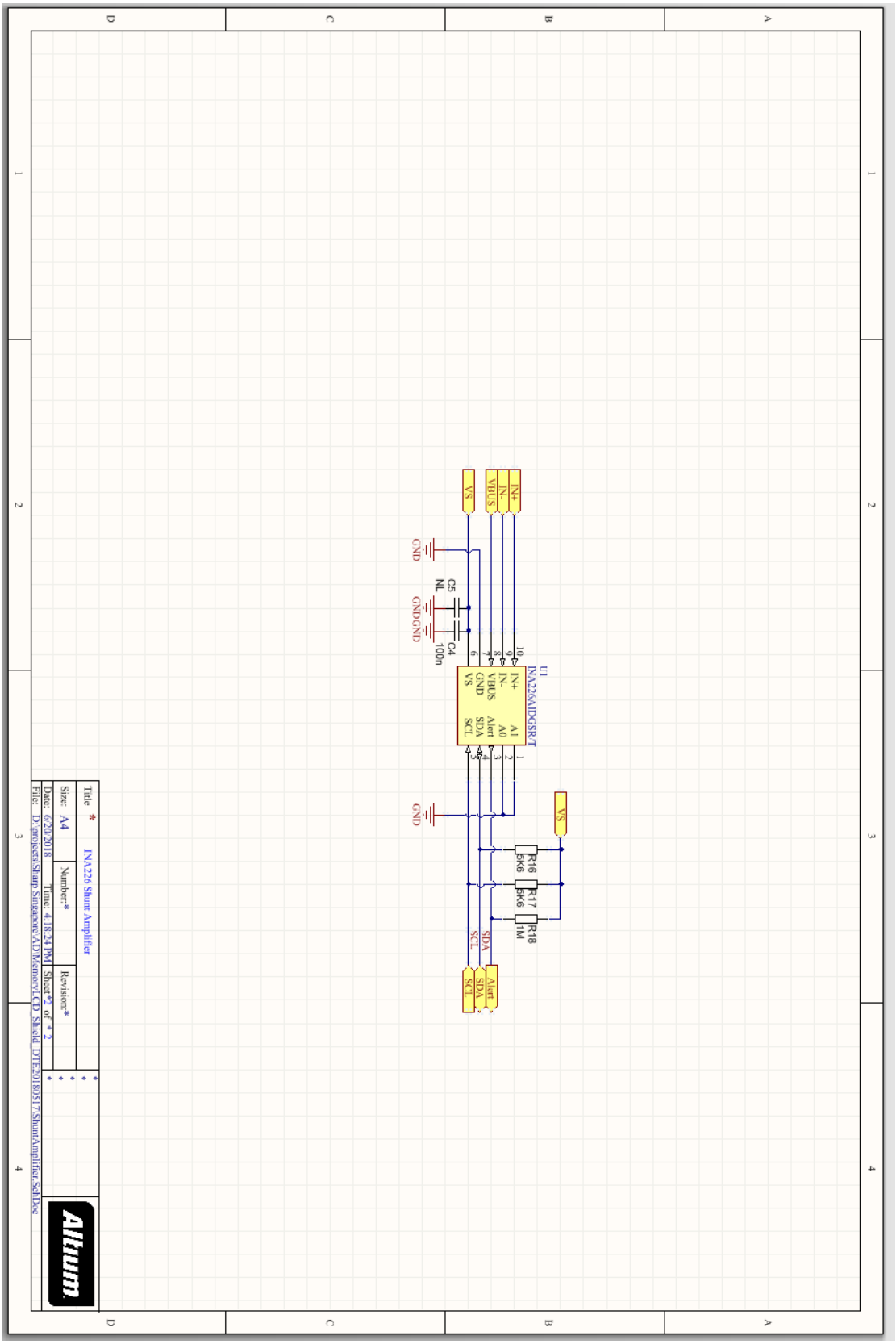
$Power_{maintain}$ = 5V * (20mV/100) * 2.5ms/2/1000 = 0.00125mW = 1.25uW.

Conclusion

Memory LCD is a high quality monochrome LCD delivering high frame rate, high contrast LCD, with partial update feature and the at same time consuming as small as few microWatt when display content is maintained.

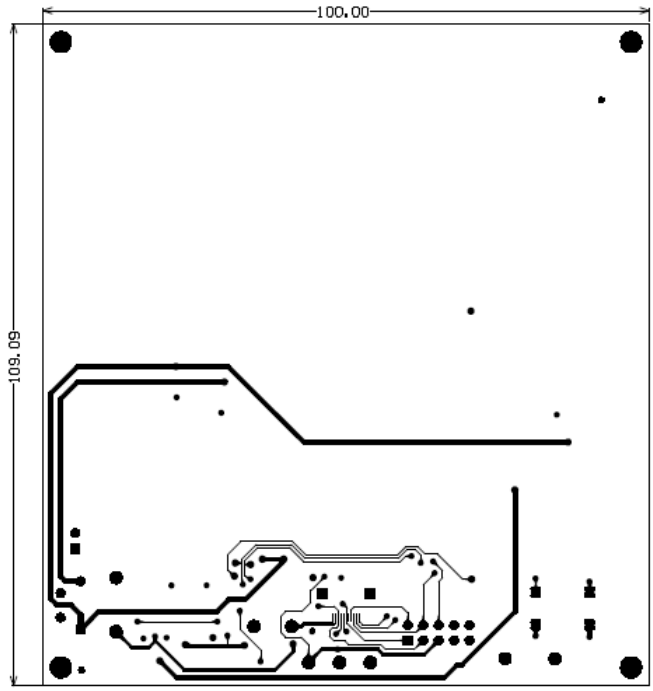
Appendix A Schematic diagram of Memory LCD Shield



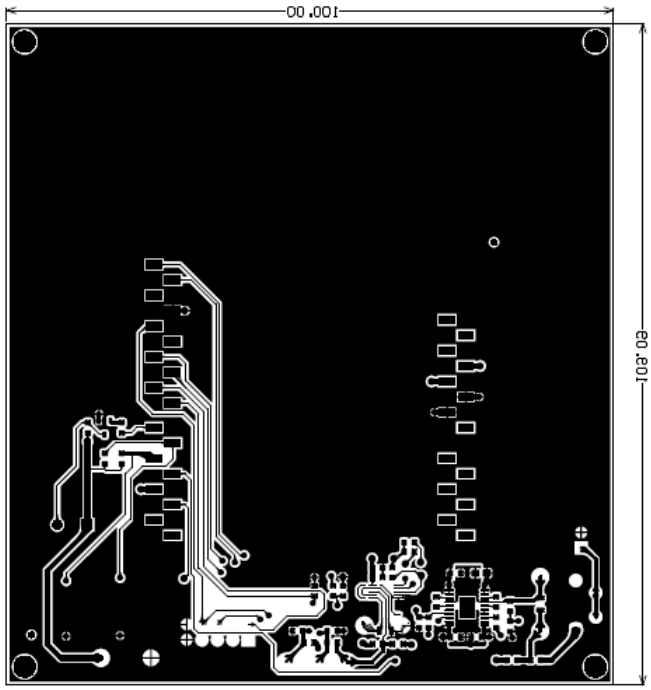

IN+
IN-
VBUS
VS
GND
C5
NL
C4
100n
U1
INA226AIDGSR/T
VBUS
IN+
IN-
GND
VS
A1
A0
Alert
SDA
SCL
R16
5K6
R17
5K6
R18
1M
SDA
SCL
Alert
Title * INA226 Shunt Amplifier
Size: A4
Number:*
Revision:*
Date: 6/20/2018
Time: 4:18:24 PM
Sheet *2 of * 2
File: D:\projects\Sharp Singapore\ADJ\Memory\LCD_Shield_DTF\20180517\ShuntAmplifier.SchDoc
Altium

PCB layout of Memory LCD Shield

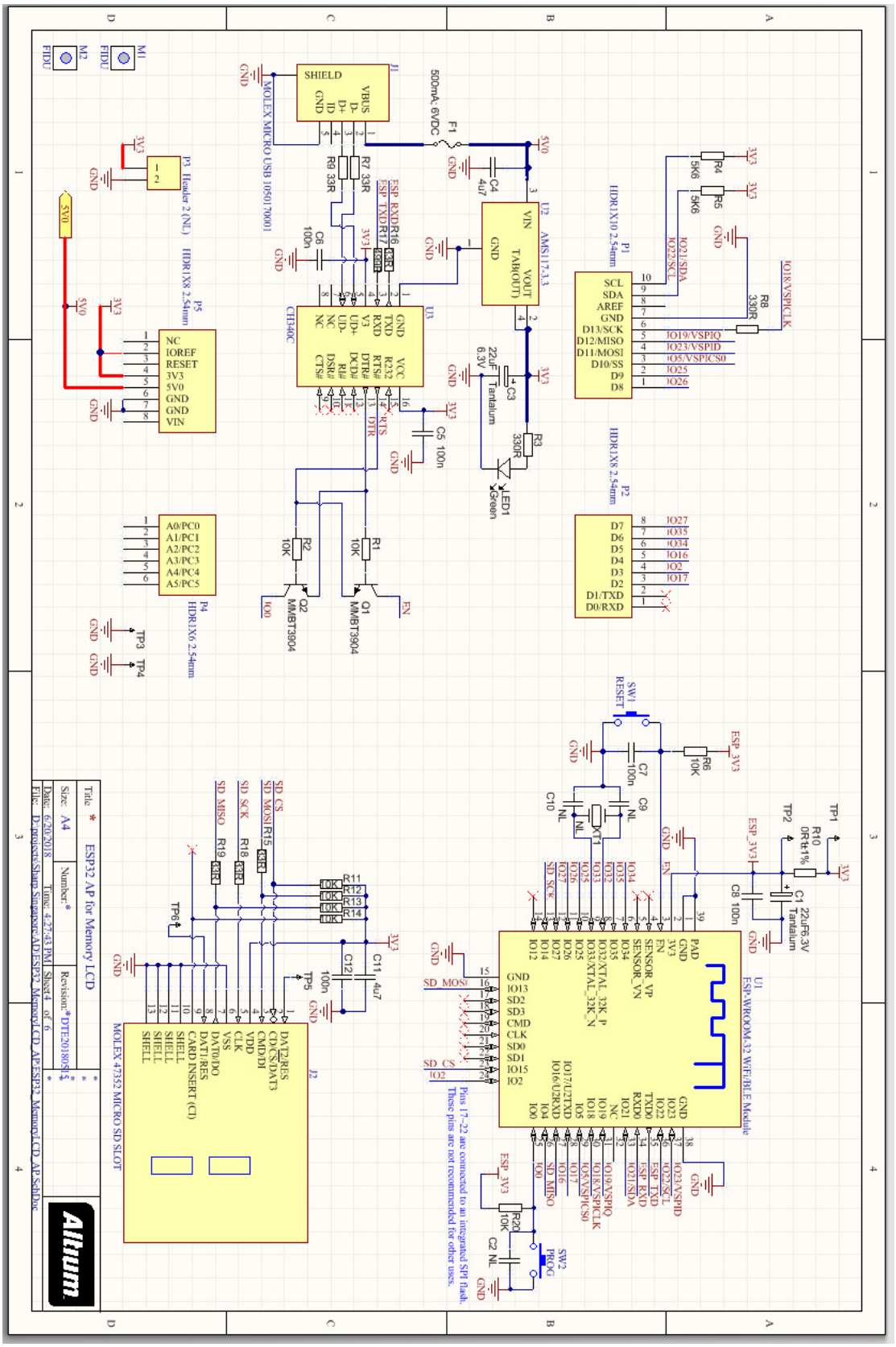
Top layer



Bottom layer

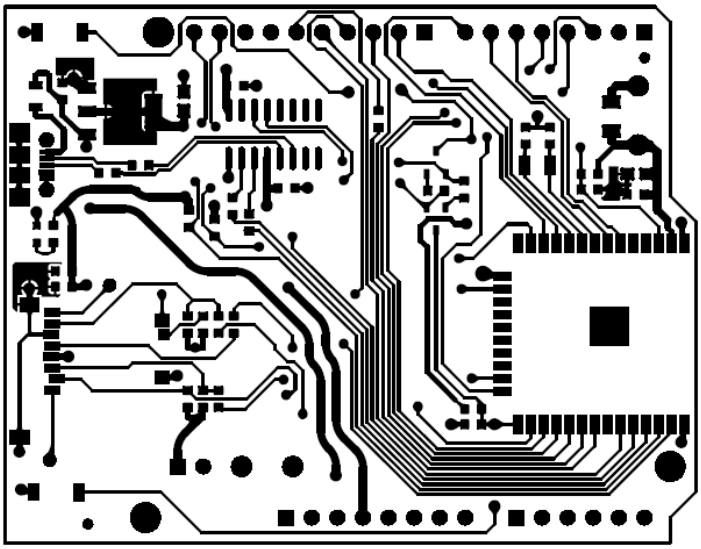


Appendix B Schematic diagram of ESP32 Application Processor



PCB layout of ESP32 Application Processor

Top layer



Bottom layer