UPPSALA
UNIVERSITET

UPTEC F 22002
Examensarbete 30 hp
Januari 2022

Embedded GUI Library Development

Sofia Dreborg

Civilingenjörsprogrammet i teknisk fysik

UPPSALA
UNIVERSITET

Embedded GUI Library Development

Sofia Dreborg

Abstract

This project aimed to create a simple open-source embedded graphical user interface library that could be used on more or less any microcontroller platform. The programming language was intended to be C++ for the GUI but as the project evolved C was chosen above C++. This was a decision based primarily on the fact that STM's development environment, STMCubeIDE, is less compatible with C++. The IDE offers great hardware support which in the end was more important than the advantages given by C++.

The hardware used in this project was an STM32F469 microcontroller. It has an ARM Cortex M4 processor core and 2 Mbyte of flash memory and 384 Kbytes of RAM. Wrapper functions for the Board Support Package, BSP, were written as a part of the library to allow easy access to the BSP needed for the hardware configuration.

The first part of the project goal was achieved, a simple GUI library was created. The resulting GUI library supports user interaction through buttons, it can display the current time and visualizes given data in graphs. The graph function can display the data live, as a scatter plot, a bar plot and a line plot. The library also supports an alarm function that allows the user to decide what will happen after the alarm time is up. However, even though the GUI library was written to be device-independent, the product has not been tested on other platforms.

For further development, this GUI library could be tested on another microcontroller. This would provide answers to how much software changes are needed to make the product as hardware independent as possible. To make the library lighter and faster, there is a possibility of optimizing the GUI core.

Teknisk-naturvetenskapliga fakulteten

Uppsala universitet, Utgivningsort Uppsala

Handledare: Johan Sundqvist Ämnesgranskare: Uwe Zimmermann

Examinator: Tomas Nyberg

Populärvetenskaplig sammanfattning

Grafiska användargränssnitt är idag integrerade i alla delar av samhället. De används i allt från persondatorer och mobiltelefoner till inbyggda system i industrin. Användargränssnitten vi ser är inte bara estetiskt tilltalande utan även homogena i utseendet och enkla att använda. I takt med att tekniken blir allt mer komplex ökar kraven och förväntningarna från allmänheten på enkla och tilltalande gränssnitt som gör det lätt och intuitivt att använda avancerad teknik.

Grafiska användargränssnitt har använts i industrin länge även om dessa gränssnitt ofta har haft ett enklare utseende än ett gränsnitt för kommersiellt bruk. Kraven på användarens upplevelse har varit lägre då tekniken i industrin främst är ett hjälpmedel för de industrianställda. En interaktiv display inom industrin presenterar ofta viss data på ett mycket tydligt och avskalat vis, till skillnad från till exempel en mobiltelefon där kunden ibland kräver att användargränssnittet inte bara är tydligt utan även estetiskt tilltalande och responsivt. [15]

De flesta som arbetar med mjukvaruutveckling använder externa bibliotek dagligen. Dessa bibliotek kan exempelvis vara samlingar av funktioner skapade av ett programmeringsspråks grundfunktioner och bidrar till att påbygga språket inom ett visst område. Speciellt inom utveckling av inbyggda system är det viktigt att lämna ett så litet avtryck som möjligt i både arbetsminne och programminne vilket leder till att bibliotek tjänar på att vara så oberoende som möjligt av andra bibliotek. Detta benämns ofta som att vara fri från *dependencies*. Det här projektet hade som avsikt att bygga ett enkelt GUI-bibliotek från grunden och publicera det som öppen källkod. Tanken med biblioteket var även att det skulle kunna användas på olika typer av hårdvara.

Biblioteket består av en fil innehållande alla kärnfunktioner som styr hur interaktionen med användaren sker, samt flera så kallade widgets. Widget betyder gränssnittskomponent och det är helt enkelt funktioner som utför en viss uppgift. Ett exempel är plotfunktionen som visar data på skärmen i grafer. Hårdvaran som används i projektet är en Discovery board från STMicroelectronics. Kortet har en 32-bitars mikrokontroller och en inbyggd 4.3" LCD display med pekskärm.

Det resulterande biblioteket uppfyller det första projektmålet och kan visa data grafiskt på LCD-displayen, visa vad klockan är och interagera med användaren genom knappar. Biblioteket är oberoende av externa bibliotek och bygger på det nödvändiga abstraktionslagret mellan mjuk och hårdvara. Mer finns dock att önska av slutprodukten där den mest uppenbara förbättringspunkten är att testa biblioteket på en annan plattform och modifiera koden för att göra den mer hård-

varuoberoende. Utöver att undersöka portabiliteten kan GUI-kärnan med fördel optimeras för att göra bilblioteket lättare och snabbare.

Acknowledgements

I would like to thank several people who in one way or another have been involved in this master's thesis. Firstly I would like to thank my brilliant supervisor Johan Sundqvist, for your support, encouragement and patience. Your great knowledge of embedded development together with your willingness to teach has been a great asset to me. I could not have done this without you.

Secondly I would like to thank Uwe Zimmermann, my subject reader, for great support along the way, always being available for questions. Thirdly, I would like to thank Mattias Abelsson, my group manager at Knightec, for showing interest and continuously making sure I had everything I needed.

I also want to thank my friends and family for supporting me both through this thesis and through the almost 6 years of study that predeceased it. Special thanks to my partner August Forsman, not only for excellent daily support but for a heroic rescue of my laptop and microcontroller when I accidentally dropped my bag in the ocean.

List of Abbreviations

ADC - Analog-to-Digital Converter
AHB - Advanced High-performance Bus
APB - Advanced Peripheral Bus
BSP - Board Support Package
DAC - Digital-to-Analog Converter
GPIO - General-Purpose Input/Output
GPL - GNU General Public License
GUI - Graphical User Interface
HAL - Hardware Abstraction Layer
I/O - Input/Output
IC - Integrated Circuit
I^2C - Inter-Integrated Circuit
LCD - Liquid-Crystal Display
LGPLv3 - GNU Lesser General Public License version 3
PWM - Pulse Width Modulation
RNG - Random Number Generator
RTC - Real-Time Clock
SPI - Serial Peripheral Interface
stderr - Standard Error Stream
TFT - Thin-Film-Transistor

Contents

1 Introduction

Today we assume that all interaction between humans and machines will happen effortlessly. The old way of manually controlling a machine through a complex Human Machine Interface, HMI, is replaced by touchscreens displaying advanced graphics. The pace of development is fast and people now demand these kinds of accessible interfaces for all applications. To meet this demand, knowledge and experience of Graphical User Interfaces, GUIs, is not only necessary for smartphone developers but for developers of embedded industrial applications in all areas where the human and the machine interact.

1.1 Problem formulation

The aim of this project is to create a GUI library that can be used on more or less any microcontroller platform. The hardware used in the development process will be a discovery board with an STM32F469 microcontroller from STMicroelectronics. The microcontroller has been chosen both because of the relatively generous amount of memory and because of its ARM Cortex-M processor architecture. The processor is commonly used in several industries so the platform is a good representation of the kind of hardware the GUI could operate on. [9]

The primary goal is to make a lightweight GUI library that is simple but efficient. The library should support simple functions such as displaying the current time and user data on the screen. Additionally, functions that allow the user to display data live are preferable since it is a required feature when using a GUI for monitoring purposes. An ideal portable library is completely platform-independent, which means that the library can be used on any platform. Instead of trying to make such a library, a delimitation of the project is made to suit its scope. The goal will instead be to create a library that can be ported from one microcontroller platform to another. The GUI library is intended for industrial applications rather than home devices and consumer electronics.

The objectives can be summarized in the following thesis goals:

1. To create a simple GUI library that is independent of external libraries.
2. To design this library so that it can be used on more or less any microcontroller platform.

2 Background

STMicroelectronics, STM, is a company that is enthusiastic about embedded GUIs and has introduced a concept called HMI of things. An HMI in this context can be both a GUI, voice control, a touch screen as well as gesture and VR. The development of these techniques will meet the demand from society for a better user experience in all applications. In addition to an improved user experience, there is a belief that this will contribute to safer data management. [7]

2.1 Design aspects

A product that will be used in the industry usually has fewer requirements regarding layout and more connected to reliability and readability. A product that is intended for a home needs to blend in with the environment and hence needs a more elegantly designed user interface. Since this GUI library is addressing the industry as the end customer, a user that handles the final embedded application including this GUI library, usually has a little more experience with technology than the average person. This might make them more forgiving, to some extent, if the user interface is not perfect. One of the fundamental differences between a GUI for a smartphone and a GUI for other embedded devices is that the smartphone is updated several times a year while the embedded device often is left untouched for several years. Hence, the design choices tend to differ when designing an embedded GUI. For other implementations, current design trends could be important to take into consideration, but for a GUI that is intended for an embedded device the durability of the design over time could be a much better focus. [12]

Another thing to take into consideration is whether or not the end product, in this context meant an embedded device that has the GUI implemented, will be stationary or portable. A portable device is less sensitive to font size since the user can move the screen closer to his or her eyes if the font size is too small. If the font size should be changeable for the end-user, i.e. the one interacting with a device that has the GUI implemented, it is important to make this task easy to make the customer experience enjoyable and avoid unnecessary irritation. It is nevertheless important to let the developer using the library have easy access to the font size tools to enable customized applications and broaden the library's field of application.

Several embedded GUIs exist on the market that already fulfills the requirement specified above and this project will not reinvent the wheel. However, understanding and writing GUIs for embedded applications is a useful competence since good-looking, easy-to-use GUIs are crucial for many technical areas.

2.2 Choice of programming language

The original strategy was to use C++ for the GUI software and C for all the hardware configuration files. C++ is fairly low level but still object-oriented which makes it suitable for an embedded GUI. However, the IDE used, STMCubeIDE, did not support C++ to the same extent as it supports C when auto-generating GNU Makefiles with the provided drivers. The IDE is a tool developed by ST for pure embedded applications and it works best with plain C for all code. The IDE provides useful hardware support that turned out to be more essential to the project than keeping C++ for its benefits in being object-oriented. As a consequence, the choice of programming language fell upon C for both the drivers and the GUI library. [13]

2.3 Embedded GUI libraries that already exist

Several embedded GUIs exist on the market but the demand for customized simple GUIs is still present. The three GUI libraries listed below are written in C ++ and all are well suited for different types of embedded graphic design.

- TouchGFX
- Qt
- GuiLite (open source)

TouchGFX is the STM32 family's own GUI toolkit that enables smartphone-like GUIs for embedded applications. The toolkit is written in C++ and developed specifically for STM32 by ST themselves which naturally makes the optimization for their own hardware hard to compete with. ST describes TouchGFX as a complete graphics software solution. It is memory-optimized and has high performance that makes it suitable for embedded graphics development. A lot of consideration has been put into making the software easy to use and it is also free of charge for all ST customers. ST recommends this tool and also simplifies the development process by offering support for costumer using TouchGFX together with ST hardware. [2] The RAM footprint of the library is improved and updated continually and at the time of writing the required internal MCU RAM to allow a simple GUI is only 16-20KB. [5]

Another famous library is Qt. It is an extensive and popular library written in C++ that can do advanced user interfaces including both 2D and 3D graphics. The drawback of this library is its size and that the license is quite expensive. Qt supports anything from desktop applications to embedded systems and is also compatible with a wide variety of operating systems like Mac OS, Windows, and

Linux as well as smartphone platforms as iOS, Android. It also supports smaller OS like Embedded Linux, QNX, and VxWorks. [4]

Qt has one commercial license that allows developers to create and distribute software without limitations set by open source agreement. The library is also available under GPL and LGPLv3 open source licenses. [17] GPL stands for GNU General Public License which is a series of free software licenses that guarantee end users the freedom to run, study, share, and modify the software [18]. LGPLv3 stands for GNU Lesser General Public License version 3 which is an open source license similar to GPL [19]. The central difference between the two is that it is allowed to include programs licensed under LGPL in a new program, without the new program being covered by LGPL [19]. The open source licensing is intended for students, hobby projects and similar applications where the developer has no intention to distribute the code [17].

The only GUI library that can be found completely open to the public on GitHub is the library called GuiLite. It is an open-source lightweight GUI library written in C++. It is available free on Github and also comes with some support. GuiLite is licensed under Apache License 2.0 which is described on GitHub as
"*A permissive license whose main conditions require preservation of copyright and license notices. Contributors provide an express grant of patent rights. Licensed works, modifications, and larger works may be distributed under different terms and without source code.*" [6]

3 Theory

A GUI differs from an HMI as it requires an interface, which typically is a screen with or without touch. An HMI can be practically anything that enables the communication between a human and a machine. In this context, the GUI is a way for a user to interact with the microcontroller through an LCD touchscreen. This means that the code must be written to handle interaction, not only execute some sequential predefined pattern.

The main goal is to create an operational GUI and not to optimize the code for performance. Some optimization may be done at the end of the project, but in the first stage of the project, the hardware is selected to make it as easy as possible to start the implementation. The choice fell on a discovery board with a built-in LCD. The board has an Arm Cortex-M processor architecture which is preferable since they are commonly used in several industries. [3]

3.1 Liquid Crystal Displays - LCD

LCD stands for liquid crystal display and it is a technique for displays with or without colour support. The technique dominates the display market and is used both for computer displays, cell phones as well as embedded devices. The display is very thin and uses liquid crystal and polarizing filters to render colours. [10]

LCD technology is based on unpolarized light being controlled to the desired intensity by letting it pass through several filters. The first filter is a polarizing filter and the second layer is a twisted nematic liquid crystal that twists the polarized beam with 90 degrees. After the twisted nematic liquid crystal, there is a polarizing filter that is 90 degrees from the first polarizing filter. Without the nematic liquid crystal, the beam would be completely blocked by the second polarizing filter since the light's vibrations are not aligned with the polarization axis. The nematic liquid crystal helps the light to change its vibrations by 90 degrees so that it can pass through the second filter. [10]

The nematic liquid crystal can be changed to let through different amounts of light by applying a voltage to the twisted nematic liquid crystal. When a voltage is applied, the liquid crystal untwists. This means that the beam of light is stopped by the second filter since its polarization is unchanged and hence can't pass through the second polarization filter. By applying different amounts of voltage to the liquid crystal the amount of light that is let through can be regulated. [10]

To make a colour LCD, every pixel that builds the LCD has three sub-pixels

containing a colour filter. The colour filter is placed between the liquid crystal and the last polarization filter. The technique is called additive RGB since the colours used are red, green, and blue. By regulating the light passing through each sub-pixel and combining them, any colour in the 32-bit space can be displayed on the screen. [10]

3.2 Touch displays

A touchscreen is an electronic subsystem that can detect a user's touch and translate the information in that touch so that a computer can understand it. All kinds of touchscreens consist of some variation of these three components: A touch sensor that recognizes a touch, a controller that translates between the sensor and PC or microcontroller, and a computer interface. [14]

There are two main types of touch screens on the market. Resistive and capacitive touchscreens. Resistive touchscreens are commonly used for counters, vending machines, and other machines commonly found in public spaces. This screen requires that you press on the screen hard enough to make it bend to trigger a touch event. This kind of screen is common but it can only handle one touch event at a time and hence it is not used for advanced applications such as smartphones. [14]

Multiple touch input, zoom and swipes can be done using a capacitive touchscreen. When a human finger touches a capacitive touchscreen it causes a change in capacitance. This change in capacitance triggers a touch event signal which is handled by the software drivers. More about the software drivers in section 3.5. Capacitive touch screens can be divided into projective and surface capacitive touch screens. Both use the same technique for recognizing a touch event but the hardware differs. [14]

3.3 Memory

Any embedded system requires three main hardware components: The Central Processing Unit, CPU, the system memory, and a set of input-output ports. The system memory can be divided into program memory and data memory. The program memory stores the software programs and the data memory stores data that is processed. As in a PC, the software programs are executed by the CPU. [16]

The memory used in an embedded system can be split into storage permanence and write-ability. Storage permanence means the ability of memory to keep its contents intact. Write-ability is a classification of how easily these memory contents can be modified. Storage permanence can be divided further into two sub-categories,

volatile and non-volatile. Non-volatile memory means a memory that can keep its content even when unpowered. Volatile memory means the opposite, the memory loses its contents when the power is lost. [16]

Three commonly used memory types in microcontrollers are flash memory, SRAM and EEPROM. The most important memory of all for an embedded system is the flash memory which is the memory used for the firmware. This memory is usually read-only and non-volatile to avoid corruption of code or text segments and to ensure that the executable program is intact even when the power is lost. [16]

In figure 1 the memory distribution of a STM32F469 microcontroller is shown. In

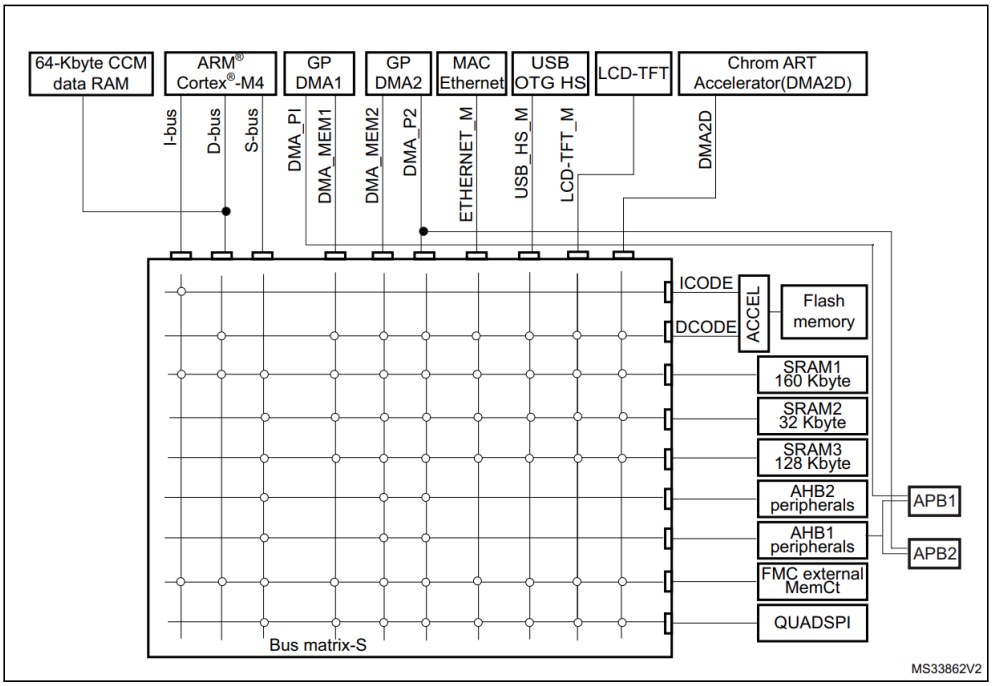


Figure 1: System architecture of STM32F469

the bottom left of the figure, the three SRAM components are grouped together. [1]

Flash

The flash memory is non-volatile, used as program memory and is a subset of EEPROM. The flash memory can only be erased a block at a time. NOR flash, as in the NOR logic gate, is the most common kind of flash memory used in microcontrollers and it can support either word-wise or byte-wise read and write operations. [16]

SRAM

SRAM stands for Static Random Access Memory. The SRAM is volatile memory where each memory cell consists of 6 transistors. Two of them are used for control purposes and four are used to store the data. SRAM is used because they are energy efficient and also easy to use. SRAM has a fast read-and-write speed, hence the SRAM is used as a cache memory both in computers and in microcontrollers. Since the memory is volatile all data inside the SRAM will get lost when the power is turned off. [16]

EEPROM

The EEPROM is a non-volatile memory and stands for Electrically Erasable Programmable Read-Only Memory. It is possible to erase the data stored in a single register at a time. Since the EEPROM memory is non-volatile, all data inside the EEPROM will be stored even when the power is off. [16]

3.4 STM32

The platform used is a discovery board from STMicroelectronics with an STM32F469 microcontroller. The microcontroller has an Arm Cortex-M4 32-bit RISC core which can operate at a frequency of up to 180 MHz. It has 2 Mbytes of Flash memory, 384 Kbytes of SRAM and up to 4 Kbytes of backup SRAM. [9]

The device has a large set of IOs and peripherals connected to two Advanced High-performance Buses, AHB, two Advanced Peripheral Buses, APB, and a 32-bit multi-AHB bus matrix. The device offers two digital-to-analog converters (DAC), three 12-bit analog-to-digital converters (ADC), twelve general-purpose 16-bit timers which include two pulse width modulation (PWM) timers for motor control. It also has a true random number generator (RNG), two general-purpose 32-bit timers, and a low-power real-time clock (RTC). [9]

The discovery board has a built-in 4" touch screen. The LCD has a thin-film-transistor (TFT) controller. [9]

When the microcontroller is started seven steps are executed regardless of what task the controller should perform.

1. HW configuration loading
2. Code execution area selection
3. HW stack pointer initialization

4. Reset vector fetching
5. SystemInit() function call
6. Memory environment setup
7. Jump to Main()

The first three steps are all purely hardware configuration and the latter involves software. [12]

3.5 Board Support Package - BSP

The aim is to create a GUI from scratch that is independent of any third-party code apart from the Hardware Abstraction Layer (HAL). This means that a lot of code must be written apart from the development of the actual GUI core. The hardware configurations are extended from the BSP drivers to the GUI files, using wrapper functions, to enable an easier software update if the platform is changed.

A Board Support Package is a customized operating system for an embedded device that enables a user to interact with the hardware through written code. The drivers are composed of several files containing supporting software for particular part of the hardware. In this project, the STM BSP drivers will be used. They contribute with a set of high-level APIs relative to the hardware components and features on the Discovery board.

The wrapper functions works as a link between STMs own BSP drivers and the GUI files where only the necessary configurations for the GUI library are included. BSP stands for Board Support Package and the drivers are included in one of the STM example projects named BSP. Reuse of STM BSP drivers is allowed as long as their work is acknowledged. Below is the copyright notice that follows with all software provided by STM.

"*Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:*

1. *Redistribution of source code must retain the above copyright notice, this list of conditions and the following disclaimer.*
2. *Redistribution in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.*

3. *Neither the name of STMicroelectronics nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.*"

The drivers are based on HAL and are delivered with the STMCubeIDE MCU environment, customized for the selected development board. The link between HAL and the external components is made independently within the BSP drivers. The organization of the HAL, BSP and user files can be seen in figure 2. The driver files can be used directly by a user by adding the files with the required services to the workspace. [11]

The STM BSP drivers are divided into four main parts.

1. Function drivers
2. Common driver
3. Component drivers
4. Bus I/O driver

Function drivers supply several high-level APIs for a specific class or functionality. An API is an interface that enables the developer to access an external service using a set of commands. In this project, especially the LCD and Touchscreen drivers are essential. Figure 2 is a block diagram of the STM BSP drivers where function drivers, common drivers, component drivers and bus input/output (I/O) drivers are specified. [8]

The common driver also provides a set of friendly APIs but for HMI. In this project the LEDs and the user button have been the most used components, for testing purposes, but the drivers also include joysticks and COM services. [8]

Component drivers are generic driver that can be portable on any board. They are used for an external device on the board that is independent of the HAL. They provide specific APIs to the external integrated circuit (IC) component. The component driver includes component core files, option configuration files and component register files. [8]

The fourth and final component is a bus I/O driver. This is a generic bus interface to supply the transport layer for components like Inter-Integrated Circuits (I^2C) or Serial Peripheral Interfaces (SPI). [8]

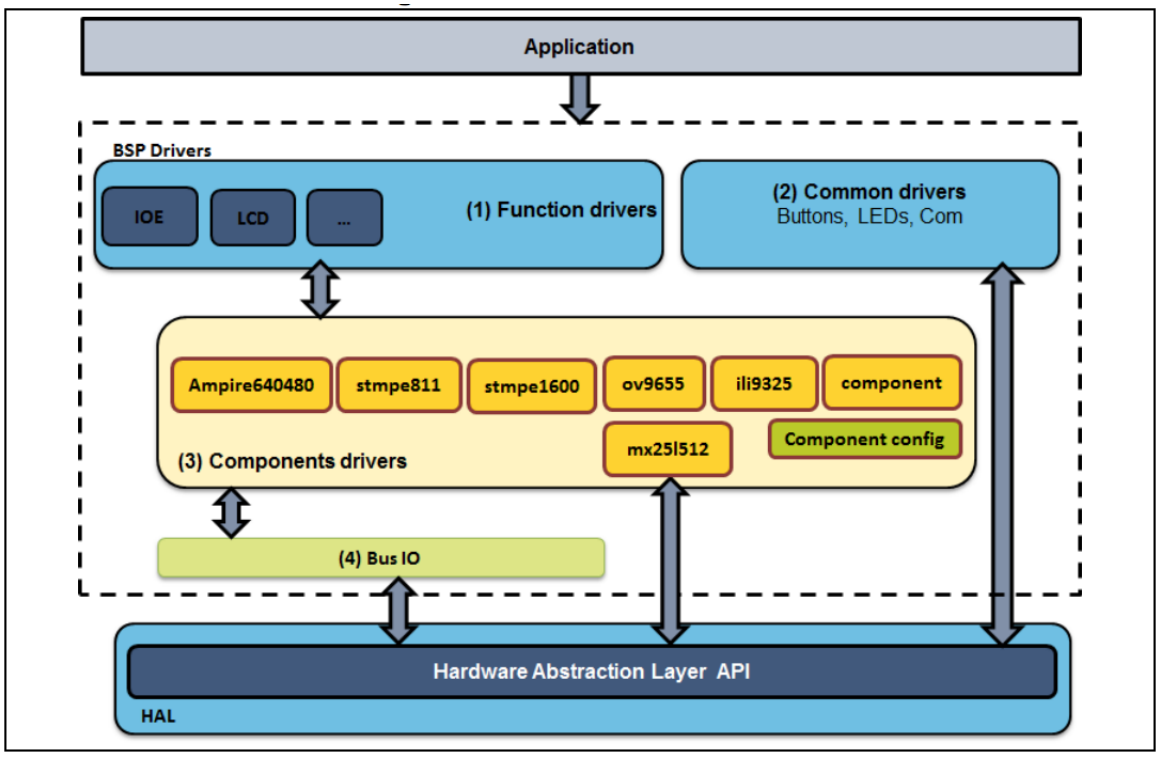


Figure 2: STM BSP driver from STM BSP driver user manual. [8]

The aim is to use the STM BSP drivers to create wrappers that makes porting the GUI code to another architecture easier. This will allow the GUI code to run on the microcontroller independent of any third part code apart from HAL libraries. The required files that will need to be modified will be function drivers like LCD and touchscreen as well as common drivers for LEDs and the user button.

3.6 Graphical user interface - GUI

A GUI differs from an HMI as it requires an interface, which typically is a screen with or without touch, whereas an HMI can be practically anything that enables communication between a human and a machine. A GUI is a way for a user to interact with a device using graphic icons such as buttons and figures instead of using traditional text based systems like the command line. In the context of this project, a GUI is a way for a user to interact with the microcontroller through the LCD. This means that the code must be written to handle interaction, not only execute some sequential predefined pattern.

This particular GUI has three main functions which build the GUI core. These

functions are named `GUI_Init()`, `GUI_Add()` and `GUI_Refresh()`, and they are described in detail below. The calling hierarchy of these functions can be seen in figure 3.

The `GUI_Init()` function initializes everything from HAL and the system clock,

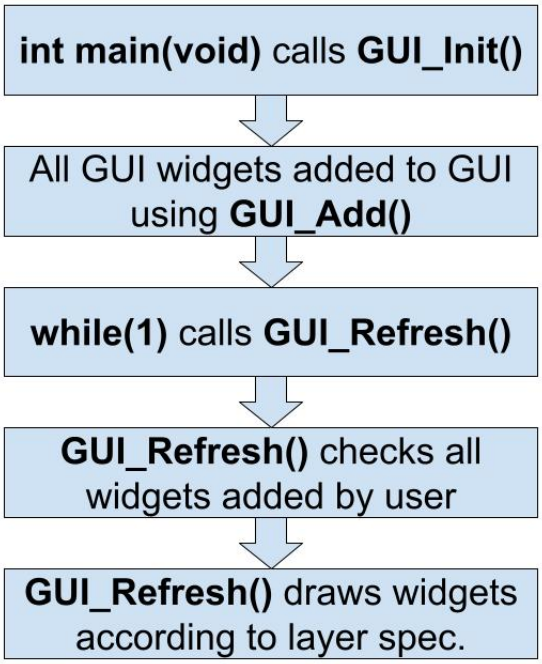


Figure 3: Calling hierarchy

to the background colour of the GUI. Right now the user is not able to set the background colour but this would be easy to change by adding a GUI struct in the main file and removing the colour settings from the init-function.

The add function is specific for every GUI object. To add a button to the GUI the `GUI_AddButton()` is used. This function is located in the button file where all other button-related functions are stored. The same follows for other objects like plots etc. The primary purpose of the add function is to register and track how many objects are added to the GUI. The function adds the pointer to the struct of a widget to an array where all widgets of that type are stored.

The refresh function is the engine of the GUI and it handles all actual interactions

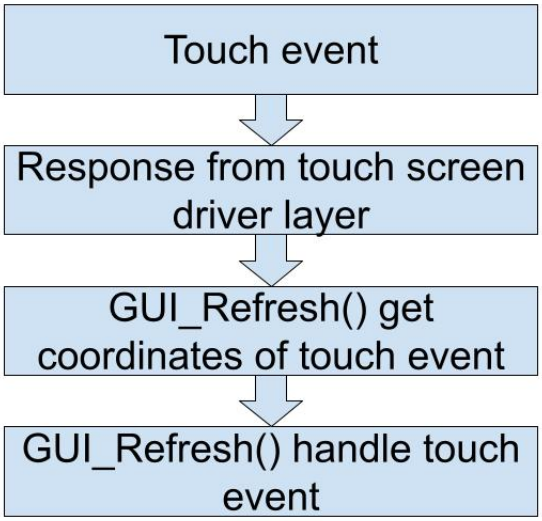


Figure 4: Event processing

between the user and the machine. The refresh function checks for touch screen interrupts triggered by a user touching the screen. As described in section 3.2, a sensor recognizes a touch on the screen. A controller translates between the sensor and the software drivers which in turn generates the touch as the coordinates on the screen on which the touch took place. The refresh function uses these touch drivers to get the coordinates. If the touch occurred where the user has placed a button the refresh function sends this information onward so that the user implemented callback function is executed. Figure 4 shows a flowchart of this process.

It is important that the code executed in the refresh function operates correctly because problems here will have great effect on the user experience. If there is a problem in the refresh functions that results in an unwanted delay, everything that is put on screen will be affected.

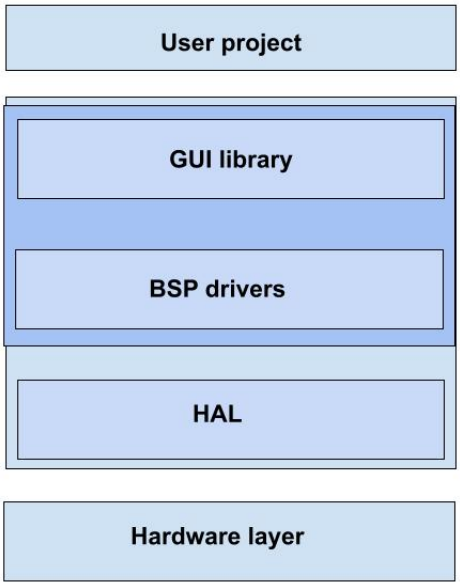


Figure 5: The hierarchy of the user code, the GUI library, the BSP drivers, HAL and the hardware.

The GUI is supposed to act as a middleware between the platform BSP drivers and the user code. To make it possible to use the GUI library on any microcontroller platform the aim is to make the entire GUI build upon a function that draws a pixel on the display. Figure 5 shows a block diagram of the hierarchy of the user code, the GUI library, the BSP drivers, HAL and the hardware.

3.7 Portability

A goal for this thesis is to make the GUI library that can operate on any microcontroller device. When the GUI is sufficiently developed the aim is to move the code to another platform to test the software on another architecture. A Raspberry Pi is more advanced than an STM 32 but they have some similarities since both operates on Arm cores. Another interesting board to use would be a board with less RAM than the STM32F469 with 324 kB of RAM. To change hardware to another 32-bit ST microcontroller would be relatively simple since the BSP drivers should follow the same naming convention and general pattern as the BSP drivers used for the STM32F469.

To evaluate how well the current library can operate in a different environment the BSP drivers needs to be rewritten and the HAL libraries need to be changed to match the new architecture. The GUI files will stay essentially intact apart from hardware initialization that will need to be updated.

4 Method

The GUI is built upon three parts, the GUI core which handles all user interaction, the wrapper functions which enables easy access to the hardware, and the GUI objects called widgets that are added to the GUI.

4.1 The GUI core

The GUI core is dependent on the `GUI_Init()` function together with the `GUI_Add()` and `GUI_Refresh()`. The `GUI_Init()` function, which can be seen in listing 2, is called in main once to initialize everything that the GUI needs for startup and `GUI_Add()`, see listing 3, adds the widgets to the GUI. The refresh function, `GUI_Refresh()`, see listing 5, then keeps track of the user interactions and draws the active widgets on to the LCD. The calling hierarchy of the GUI is shown in figure 3. Some of the properties defined in the GUI struct, shown in listing 1, is predefined in the initialization function and some set by the user.

```
typedef struct
{
  uint32_t  BackgroundColour;
  uint32_t  TextColour;
    sFONT    *pFontG;
  uint32_t  last_refresh;
  uint32_t  update_interval;
}gui_t;
```

Listing 1: GUI struct

```
void GUI_init()
{
        GUI1.mfx_toggle_led = 0;
        GUI1.lcd_status = GUI_LCD_OK;
        GUI1.lcd_status = GUI_HW_LCD_Init();
        GUI_TEST_APPLI_ASSERT(GUI1.lcd_status != GUI_LCD_OK);

        GUI_HW_Init();
        GUI1.BackgroundColour = GUI_LCD_COLOR_WHITE;
        GUI1.TextColour = GUI_LCD_COLOR_LIGHTGREEN;
        GUI_HW_LCD_SetFont(&GUI_LCD_LOG_TEXT_FONT);
        GUI_HW_LCD_SetBackgroundColour(GUI1.BackgroundColour);
        GUI_HW_LCD_ClearBackground(GUI1.BackgroundColour);
        GUI_HW_LCD_SetTextColour(GUI1.TextColour);
        GUI1.last_refresh = GUI_GetTick();
```

```
}
```

Listing 2: This function initializes the GUI core.

`GUI_Add()` will be called after `GUI_Init()` in main. An example of the add function is the `GUI_AddButton()`, see listing 3, which takes a pointer to a struct, here the button struct, as an input and adds this button to an array of buttons. All widgets have a similar add function which takes a pointer to a relevant struct as input and adds the pointer to an array. A variable called `nr_of_buttons` stores the number of buttons and whenever a new button is added to the GUI this counter gets incremented.

```
void GUI_Button_AddButton(gui_button_t *button)
{
  if(nr_of_buttons < GUI_MAX_NR_BUTTONS)
  {
    ADDED_BUTTONS[nr_of_buttons++] = button;
  }
}
```

Listing 3: The add function adds a pointer to a button struct to an array. The array holds pointers to all buttons that are added to the GUI.

The array called `ADDED_BUTTONS[]` will be accessible using a get function shown in listing 4.

```
gui_button_t *GUI_Button_GetButton(uint8_t i)
{
  return ADDED_BUTTONS[i];
}
```

Listing 4: This function returns a pointer to a button with index i.

The refresh function, `GUI_Refresh()`, is called in the while true loop in main. At the moment the buttons handle all direct user interaction since that is the only widget, apart from the alarm clock, having a callback function. The callback function of a button can be observed in listing 9 and the refresh function in listing 5

```
void GUI_refresh()
{
        for(int i = 0; i < GUI_Button_GetNrOfButtons(); i++)
        {
                gui_button_t *current_button =
                   GUI_Button_GetButton(i);
```

```
        if(current_button->layer == 1)
        {
                GUI_Button_DrawButton(current_button);
                if(TS_StateSofia.touchDetected)
                {
                        if(GUI_Button_PushButton(
                           current_button, x_click,
                           y_click, current_button->layer)
                            == 1)
                        {
                                GUI_HW_LCD_ClearBackground
                                   (GUI1.BackgroundColour)
                                   ;
                        }
                }
        }
}
for(int i = 0; i < GUI_Plot_GetNrOfPlots(); i++)
{
        gui_plot_t *current_plot = GUI_Plot_GetPlot(i);

        if(current_plot->time_until_next_draw <= 0)
        {
                if(current_plot->layer == 1) //>=1
                {
                        GUI_Plot_DrawPlot(current_plot);
                }
        }
        else
        {
                current_plot->time_until_next_draw =
                   current_plot->time_until_next_draw -
                   time_elapsed;
        }
}
for(int i = 0; i < GUI_Clock_GetNrOfClocks(); i++)
{
        gui_clock_t *current_clock = GUI_Clock_GetClock(i)
           ;
        if(current_clock->layer == 1)
        {
                if(timer >= current_clock->alarm_time)
                {
                        GUI_Clock_AlarmClock(current_clock
                           );
                }
        }
}
GUI1.last_refresh = GUI_GetTick();
```

```
}
```

Listing 5: The refresh function is the engine of the GUI.

The GUI library has its own layer function that controls what is displayed on the screen. The current implementation has two layers. One that means the object is not visible, represented by 0, and one that makes the object visible, represented by 1.

4.2 Graphics and BSP drivers

To obtain a GUI library without third party dependencies, a selected part of the STM BSP drivers must be integrated into the library. Instead of using drivers containing all possible hardware support wrappers are created only for the drivers that contain support for the tasks performed by the GUI.

An important part to access are the LCD files where all code connected to drawing on the screen is configured. Here are everything from colour definitions to functions that allows to write complete strings on the screen. Listing 6 shows an example of a wrapper function, here one that sets the text colour.

```
void GUI_LCD_SetTextColour(uint32_t Colour)
{
    void BSP_LCD_SetTextColour(uint32_t Colour);
}
```

Listing 6: An example of a wrapper function. Here is a function that sets the text colour, called `GUI_LCD_SetTextColour()`

Other important drivers are the touchscreen drivers which handles the touch screen interrupts. Here are functions that identifies if a touch has happen as well as where the touch occurred. These drivers are crucial for the refresh function shown in listing 5.

4.3 The GUI widgets

All the widgets are built upon the add function that keeps track on how many widgets are added and a draw function that draws the actual graphics. The plot function currently supports scatter plots, bar plots and a line plot. It is also possible to display the data live. Below the plot struct is shown which holds all plot properties.

```
typedef struct
{
  uint16_t  nr_of_data_points;
  uint16_t  *x_data;
  uint16_t  *y_data;
  uint16_t  origo[2];
  char    *x_label;
  char    *y_label;
  char    *title;
  uint16_t  x_axis_length;
  uint16_t  y_axis_length;
  uint8_t   layer;
  plot_type_t type;
  uint16_t  refresh_ms;
  int16_t   time_until_next_draw;
}gui_plot_t;
```

Listing 7: Plot struct

The plot is drawn using a draw function that adapts to the set plot type, origo, axis length and data length.

A button is defined as a struct with properties defined by the user. Listing 8 shows the button struct.

```
typedef struct
{
  uint16_t  x_position;
  uint16_t  y_position;
  uint16_t  width;
  uint16_t  height;
  uint32_t  text_colour;
  uint32_t  frame_colour;
  char    *text_string;
  sFONT     *button_font;
  uint8_t   layer;
  void    (*callback_function)(void);
}gui_button_t;
```

Listing 8: Button struct

The buttons are a crucial connection between the user and the platform. The developer defines what will happen when a button is pushed by defining callback functions in the user code. An example of a callback function for a button is shown in listing 9. In this example, there are four buttons and two plots added to the GUI. By setting the start button layer to 0 the start button will no longer

be visible on the screen. The same follows for the exit and back button and the second plot. The next button and the first plot will be visible to the user in the next refresh since their layers are set to 1.

```
static void Start_Button_Callback(void)
{
  start_button.layer = 0;
  next_button.layer = 1;
  exit_button.layer = 0;
  back_button.layer = 0;
  plot1.layer = 1;
  plot2.layer = 0;
}
```

Listing 9: Button callback example

The clock struct is defined as a struct with basic properties set by the user. Apart from position and other straightforward properties the clock struct contains a pointer to a callback function. This callback function is called after a certain time has passed and works as an alarm clock. The clock will display the time on the screen starting with the `start_time` set by the user. If no start time is set the clock starts ticking at zero. The implemented alarm callback function is shown in listing 12.

```
typedef struct
{
  uint16_t    x_position;
  uint16_t    y_position;
  uint8_t     hours;
  uint8_t     minutes;
  uint8_t     seconds;
  uint32_t    start_time;
  uint8_t     layer;
  uint32_t    alarm_time;
  void      (*clock_callback_function)(void);
}gui_clock_t;
```

Listing 10: Clock struct

```
int GUI_Clock_AlarmClock(gui_clock_t *clock)
{
    clock->callback_function();
    return 1;
}
```

Listing 11: Alarm function

As shown in listing 11 the alarm function simply calls the callback function of the current clock. The callback function shown below switches to the start page when the alarm is set of by setting the start button layer to 1 and all other widget layers to 0.

```
static void Alarm_Clock_Callback(void)
{
  start_button.layer = 1;
  next_button.layer = 0;
  exit_button.layer = 0;
  back_button.layer = 0;
  plot1.layer = 0;
  plot2.layer = 0;
}
```

Listing 12: Alarm callback function

4.4 The demo implementation

When developing a GUI library, it is essential to test the functions along the way in a demo application. The current demo is a main file that defines the desired GUI objects and adds them to the project. The `GUI_Init()` function is called in main. This function manages all system initiations including both necessary hardware and software configuration. All the widgets, which can be a plot or a button etc. are added to the GUI using a function called `GUI_Add()`. This function keeps track of the added widgets and their properties. In the `while(1)` loop the final core function, `GUI_Refresh()`, is called. How the user code is linked to the GUI functions can be seen in a flowchart in figure 3.

```
while(1)
  {

    if ( mfx_toggle_led == 1) {
      GUI_HW_LEDToggle(LED1);
      GUI_HW_LEDToggle(LED2);
      GUI_HW_LEDToggle(LED3);
      GUI_HW_LEDToggle(LED4);
      mfx_toggle_led = 0;
    }

    GUI_Clock_DrawClock(&clock1);

    //update y_data live
    for(int i = 0; i < plot1.data_length; i++)
    {
```

```
      y_data_1[i] = (uint16_t)(50*sin(j)+100);
      j = j + 30*pi/2;

      if(j%(30*7*pi/2) == 0)
      {
        j = 0;
      }
    }

    plot2.y_data[1] = (plot2.y_data[1] + 10)%200;
    GUI_refresh();
    HAL_Delay(50);
  }
```

Listing 13: While true loop in main() calling the refresh function and updating the data to demonstrate the plot function displaying live data.

To demonstrate how the plot function can display live data, the data for the two plots are updated in the `while(1)` loop before the refresh function is called. Apart from that, the platform's four LEDs are toggled.

5 Result

The resulting GUI can interact with a user through buttons, it is able to plot both live and static data, it can display the present time and it has an alarm function where the user can decide what should happen after a certain time has elapsed.

5.1 Graphics and BSP drivers

The BSP driver files support a lot of colour presets. The LCD drivers support a wide variety of colours and some example colour themes are shown in figure 6.

(a) Orange and green

(b) Blue and red

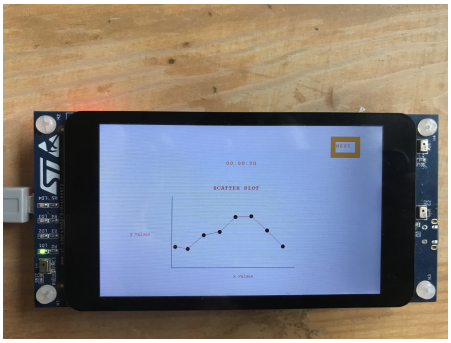
(c) Light and dark magenta

Figure 6: Colour themes displayed on the current demo homepage.

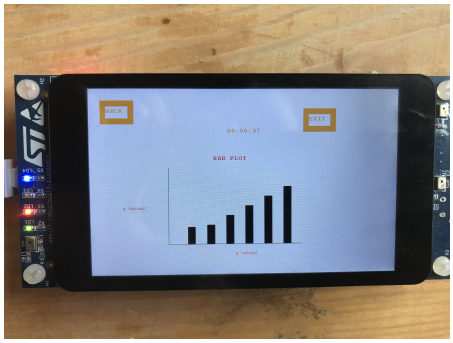
5.2 GUI widgets

The plot function supports scatter plots, bar plots and a line plot which all can be seen in figure 7. The plot is drawn using a draw function that adapts to the set plot type, origo, axis length and data length. It is possible to display live data like shown in the series of scatter plots in figure 8, but the user can also choose to

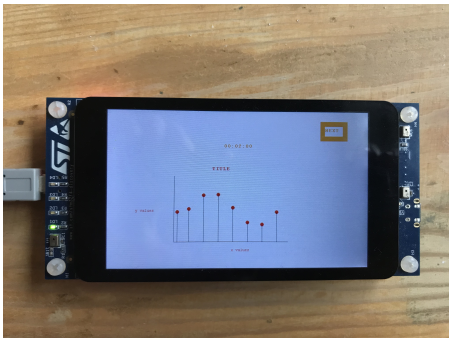
display still data only by setting struct property `live` to 0.



(a) Scatter plot

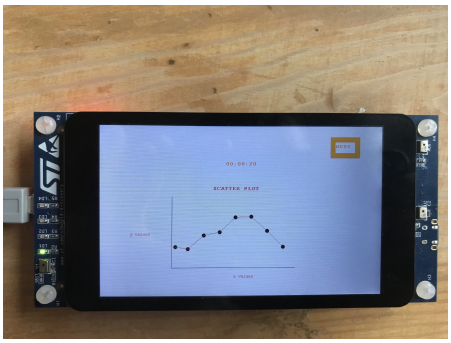


(b) Bar plot



(c) Line plot

Figure 7: Plot types.



(a)

(b)

(c)

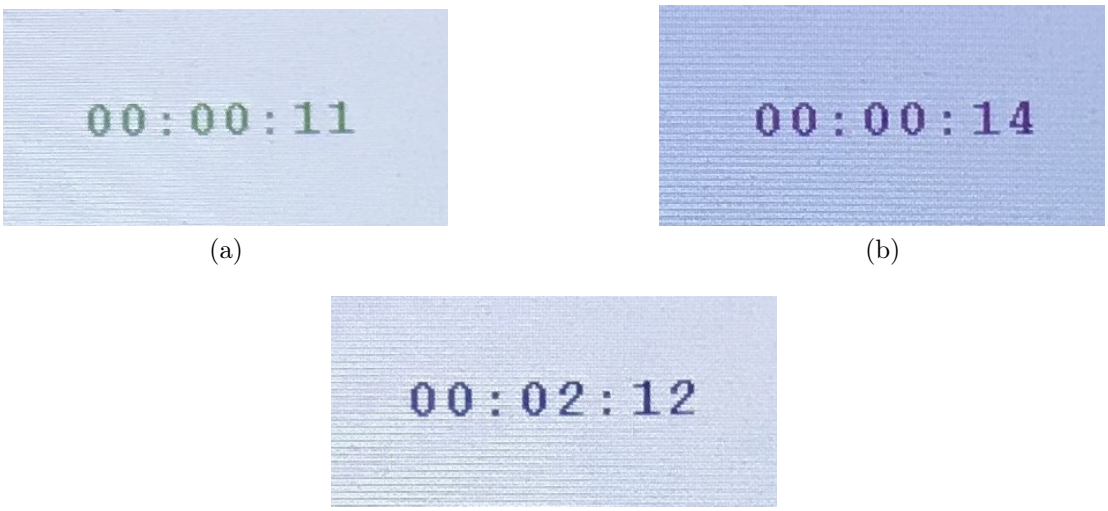
(d)

Figure 8: Subfigures 8a, 8b, 8c and 8d shows four freeze frames of live data displayed on screen in a scatter plot.

Figure 9 shows a start button in the red and blue colour theme.

Figure 9: Button in blue and red colour theme.

The clock widget is displayed as digital numbers shown in figure 10.



(a) (b)

(c)

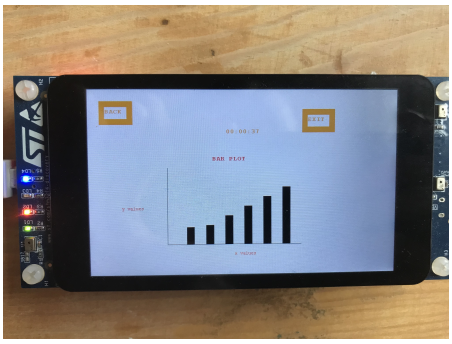
Figure 10: Sub-figure 10a, 10b and 10c shows the simple digital clock in different colour themes.

5.3 GUI demo application

The current demo of the GUI shows two categories of plots, four buttons and it displays a digital clock. Pictures of the demo are shown in figure 11. From the start page in sub-figure 11a the next page is reached by pushing the start button in the left corner. Once the push of the start button is recognized by the refresh function the start button layer is set to 0 by the corresponding callback function. Hence the start button will no longer be visible on the screen. The callback function also sets a plot and a button labeled "Next" to visible by changing both layers from 0 to 1. As a consequence a new page showing a scatter plot is shown, see sub-figure 11b. By pressing this "Next"-button the final page is displayed. The final page is shown in sub-figure 11c, which shows a bar plot and two buttons. The left button is labeled "Back" and leads back to the previous page. The right button is labeled "Exit" and leads to the front page shown in sub-figure 11a. All buttons use the same principle with callback functions as the start button described above.

(a) Start page

(b) Scatter plot



(c) Bar plot

Figure 11: From the start page in sub-figure 11a the next page is reached by pushing the start button in the left corner. A new page showing a scatter plot is shown like in sub-figure 11b. In the topmost right corner is a button labeled next. This button leads to the final page of the demo which shows a bar plot and two buttons, shown in sub-figure 11c. The left button, labeled "Back", leads back to the previous page and the right button, labeled "Exit", leads back to the front page shown in sub-figure 11a.

6 Discussion and further development

At the beginning of the project, it was time-consuming to find resources since a lot of ST documentation, as well as forum discussions, assumed previous knowledge in both embedded C/C++ combined with GUI library development. Quite a bit of time was spent on trying to make simple C++ files to compile in the IDE. It turned out to be too complicated to meet the possible benefit of using an object-oriented language for the GUI. There are obvious advantages to using an object-oriented language such as C ++ for a GUI, but since this particular GUI was quite simple, these advantages did not outweigh the difficulties with compatibility and a decision was made to change the language to C. This decision turned out to be good since it lowered the threshold for getting started with the actual library and increased the development pace a lot. It also became evident that the disadvantages that were expected to come with C did not appear to some crucial extent in this particular project.

The GUI library can be improved and developed in several ways. General optimization of the library should result in a GUI perceived as more responsive to the user, which contributes to a better user experience. Development of the layer function and the library's portability are two other parts that would enhance the quality of the library. The list below shows some suggested improvements to the library.

1. Increase the portability of the library
2. Update the GUI core to reduce number of draws
3. Develop the GUI layer properties
4. Develop the layout and introduce size scaling
5. Refine current widgets and add widgets to make the GUI more rich

6.1 Portability

The original idea was to write the GUI library on the STM32F469 microcontroller, create wrapper functions for using the BSP drivers, and then move the project to another platform. The process would involve writing wrappers for the BSP drivers for this other platform and integrating these drivers into the GUI library.

The software is built upon a function that writes pixels and all GUI code that is writing to the display is using this function. This is done so that adjustments

of the code can be kept to a minimum when moving the software to another platform. However, all the software drivers mentioned in section 3.5 would need to be modified to work on another architecture. Figure 5 shows how both the GUI library and the user code are connected to the BSP drivers but it also illustrates graphically how the entire lowest part of the system, including BSP, HAL and hardware, can be changed.

If another ST microcontroller were to be used as the next platform, it would be quite easy to update the code as the naming convention is the same for the BSP drivers. However, if you want to use a development board from another manufacturer, there is a risk that the HAL and BSP are significantly different and therefore makes the integration more complex and time-consuming.

6.2 Risk assessment

To further improve the library, another crucial point is to avoid errors by handling incorrect instructions from the developer. The current implementation does not handle these kinds of errors at all, hence it is guaranteed to cause problems for both the developer and the end-user. For example, there is nothing that prevents the developer from plotting a million data points in the plot function. Another typical error is that the x and y axes are initiated to different lengths by the developer by mistake. This error neither causes any warning to the developer.

There are occasional simple preventive measures in some places in the code, such as in the `Add_Button()` function in listing 3 where a new button is added only if the number of buttons in the GUI has not yet reached the `GUI_MAX_NR_OF_BUTTONS` value. However, if the button is not added, a printed-out error message would be preferable so that the developer will have any chance of figuring out what has gone wrong. The non-existent or deficient error handling is consistent throughout the library and that needs to change if the library is to be used in any real-life context.

One way to correct this issue is to use the `assert` function that allows a developer to control error messages. The `assert` function takes an expression as an argument. If this expression evaluates to true, nothing happens. If the expression evaluates to false the `assert` function exits the code running and writes a predefined error message to the standard error stream (stderr).

6.3 Optimization of the GUI core

Apart from making the GUI more portable and improving the error handling, a lot can be done for optimization of the GUI-core. The GUI core is implemented in such

a way that the entire screen is redrawn in each iteration. The contents of the screen are erased when the user presses a button to switch pages. Clearing the screen and redrawing is of course necessary when the user wants to switch from one page to another, but to call the draw functions in each iteration regardless of whether something has been updated on the screen is unnecessary. A better approach from an optimization perspective would have been to apply a function that checks if something is updated on the screen. If the check shows that nothing has changed, nothing needs to happen, ie. the screen is neither cleared nor redrawn but remains as the previous iteration. If it turns out that something has been updated, the screen must be cleared and redrawn with the new update.

6.4 Develop the GUI layer function

The GUI layers are a part of the GUI library that keeps track of what is displayed on the screen. All widgets have a layer property and the value is assigned by the user and changed in the callback functions. At present time only two layers are supported, 0 for not visible and 1 for visible. This is sufficient for the current implementation but, presumably, as the complexity of the GUI increases the maximum number of GUI layers have to be increased. To be able to implement some sort of hierarchy in the display of objects a third layer could be added so that 0 still represents not visible, 1 represents visible and 2 represents visible above the 1st layer.

To have the option of overlaying several objects is a necessary prerequisite for building a generic GUI library. An apparent way to implement this is to draw the objects in the order they should appear on the screen. Since the last object drawn ends up at the top of the other objects by itself, it should be relatively easy to implement such a solution.

With the current implementation, the refresh function goes through a certain type of object in the order in which they are entered in the array of pointers. The array is filled with pointers to the objects that end up in the order in which the user chooses to enter them. It would be possible, for example, to update the add function so that it instead places the object with the lowest layer value first in the array and the one with the highest value last. Each new item would need to be sorted and added according to the objects already added to the array, but once this is done the refresh function would need little or no modification. It would then be possible to add any number of layers without having to update the code significantly.

6.5 Layout and scaling

The focus of this project was not on layout nor scaling but it would be interesting to investigate these aspects further. To make the GUI adaptable for different environments it would be great to be able to change the font size as the end user. As soon as the font size can be changed using a BSP wrapper, it should be relatively easy to make it accessible to the end user using the existing button function. Another example where scaling could be improved is for the button widget. It would be preferable that the frame of the button always follows the size of the button label. For the current implementation however, the button size is defined by the user who needs to figure out a frame size that fits the button label length.

6.6 Widget development

Adding widgets would be relatively straightforward as it is possible to use previously added widgets as a template for implementation. Some widgets that could be added:

1. Display a picture
2. A live audio amplitude plot

To show pictures on the display should be quite uncomplicated. It would only need some frame definition so that the developer can decide where to place the picture.

It would be fun to be able to use audio as input data and display audio amplitude as a function of time. The current plot function can display live data but it would be necessary to add the audio files to the GUI integrated BSP drivers and create some additional wrappers since they are not yet implemented.

7 Conclusion

The usability of this library in the industry at this point is a bit limited and the user experience is somewhat unsatisfactory, but the GUI works. There is a lot left to develop before the library could be launched as a finished product to a potential customer. It is more of a prototype than a finished product, but this prototype still fulfills its most basic purpose which is to interact with a user. There are more elegant and optimized GUI options on the market against which this GUI library can not yet compete. However, since it is open-source and small it is easy to use it as a base for developing something more advanced and refined.

If you want to take on this project and continue to develop the library, the most important improvements are linked to the GUI core's optimization and function, as well as the library's portability. The GUI core can be improved by developing the existing layer function so that it supports that objects can be placed on top of each other in a hierarchy set by the developer. Furthermore, it would be good if the refresh function is optimized so that it redraws the screen less frequently and only cleans it if necessary. This would provide a better user experience as the GUI would be perceived as more responsive. More widgets could be easily added by following an already existing widget as a template.

References

[1] STMicroelectronics. "32F469IDISCOVERY - Discovery kit with STM32F469NI MCU" . https://www.st.com/en/evaluation-tools/32f469idiscovery.html.

[2] STMicroelectronics. "STM32 Graphical User Interface - STMicroelectronics" . https://www.st.com/content/st_com/en/ecosystems/stm32-graphic-user-interface.html.

[3] ARM. "Cortex-M – Arm Developer" . https://developer.arm.com/ip-products/processors/cortex-m.

[4] QT. "Qt - Supported Platforms & Languages" . https://www.qt.io/product/supported-platforms-languages.

[5] WebWire. "STMicroelectronics Eases Simple GUI Design for Ultra-Low-Cost Devices with TouchGFX Updates and New STM32 Nucleo Shield" . https://www.webwire.com/ViewPressRel.asp?aId=265078.

[6] idea4good. "GuiLite: The smallest header-only GUI library(4 KLOC) for all platforms" . https://github.com/idea4good/GuiLite.

[7] Manners, David. "The HMI of Things" October, 2020. . https://www.electronicsweekly.com/news/business/the-hmi-of-things-2020-10/.

[8] STMicroelectronics. "BSP drivers development guidelines" June, 2019. https://www.st.com/resource/en/user_manual/dm00440740-stm32cube-bsp-drivers-development-guidelines-stmicroelectronics.pdf.

[9] STMicroelectronics. "Discovery kit with STM32F469NI MCU" 2020. https://www.st.com/en/evaluation-tools/32f469idiscovery.html#overview.

[10] Yang, Deng-Ke and Wu, Shin-Tson. "Fundamentals of Liquid Crystal Devices" 2014.

[11] STMicroelectronics. "Description of STM32F4 HAL and low-layer drivers" June, 2021. https://www.st.com/resource/en/user_manual/dm00105879-description-of-stm32f4-hal-and-ll-drivers-stmicroelectronics.pdf.

[12] STMicroelectronics. "GUI Webinar: Fundamentals for designing an Embedded GUI" June, 2020. https://www.youtube.com/watch?v=LfWKOQ9kKiw&ab_channel=STMicroelectronics.

[13] STMicroelectronics. “STM32CubeIDE - Integrated Development Environment for STM32” June, 2021. https://www.st.com/en/development-tools/stm32cubeide.html.

[14] Bhowmik, Achintya K. “Interactive Displays: Natural Human-Interface Technologies” October, 2014. https://learning.oreilly.com/library/view/interactive-displays-natural/9781118706206/9781118706206c2.xhtml.

[15] Sarkar, Roy. “5 top embedded GUI trends for 2021” 2021. https://blog.cranksoftware.com/5-top-embedded-gui-trends-for-2021.

[16] Jiménez, Manuel and Palomera, Rogelio and Couvertier, Isidoro. "Introduction to Embedded Systems: Using Microcontrollers and the MSP430" 2014".

[17] The Qt company. "Qt - Licensing" 2021". https://www.qt.io/licensing/

[18] Free Software Foundation. “The GNU General Public License v3.0 - GNU Project - Free Software” 2021. https://www.gnu.org/licenses/gpl-3.0.html.

[19] Free Software Foundation. “GNU Lesser General Public License v3.0 - GNU Project - Free Software” 2021. https://www.gnu.org/licenses/lgpl-3.0.en.html.