Politecnico di Torino

Laurea Magistrale in Ingegneria Elettronica
indirizzo: Embedded Systems

Tesi di Laurea Magistrale

Implementation of a software interface layer between model-based-design tool and embedded graphic frameworks

Relatori:
Prof. Luciano Lavagno,
Ing. Massimiliano Curti

Candidate:
Lorenzo Rinaldi

April 16, 2021

Acknowledgements

The realization of this project was possible thanks to Teoresi Group.
First, I would like to thank my supervisor, prof. Luciano Lavagno, for his availability and clarity about information concerning this fundamental step of my life.
Then, I would like to express my sincere gratitude to the Technology Leader Massimiliano Curti, at Teoresi, for the great opportunity to do my master thesis, in a continuously growing company, for his willingness to help me, sharing his knowledge, experience and for his huge professionalism.
Moreover, I would like to thank Politecnico di Torino for these five years spent between moments of fun and sacrifice, in which a learnt not only technical but also human knowledge from many professors.
Vorrei ancora ringraziare tutta la mia famiglia, mamma, papà e Laura, senza la quale tutto questo percorso non sarebbe stato possibile, perché mi hanno permesso di portare a termine questo percorso e dato la fiducia per superare ogni ostacolo postomisi davanti. Per aver creduto in me e per essersi sacrificati permettendomi di arrivare fino a questo punto.
Infine, ovviamente non meno importante, vorrei ringraziare Federica, che mi ha supportato, ma soprattutto, ha sopportato gli sbalzi d'umore e le crisi durante le sessioni d'esame che a volte sembravano interminabili. Tutti i rifiuti ad uscire perché dovevo studiare, ma mi auguro di poter ripagare tutto il tempo perso, negli anni a venire.
Ringrazio tutti i miei amici, che per brevità non nominerò tutti, ma mi rivolgo agli amici "dell'oratorio", del liceo e dei "Lopez". Tutti hanno partecipato ai momenti di spensieratezza concedendomi, spesso senza saperlo, gioia e relax, durante tutte le feste, partite ed occasioni che ci hanno permesso di ridere insieme.
Grazie a tutti, per avermi accompagnato in questo percorso, GRAZIE!

Contents

List of Figures

CHAPTER 1

Introduction

In the last years electronics, but in particular embedded systems are increasingly part of our life. They are also present where we do not think about them.
Embedded products are used in a huge variety of applications: automotive, aerospace, medicine, home automation and so on. Products have to differ from each other for characteristics and dimensions, depending on application field.
Nowadays, almost every device has a screen and a microcontroller to perform actions described from developers. To satisfy users' requests, many products, but also many brands, are on the market, each with its own innovative technology, structure or algorithm. Therefore, there is a need to facilitate interaction with these products.
Briefly, the goal of the thesis project is to implement a ***Software Interface Layer*** able to simplify interaction between a Model-Based-Design (MBD) tool and an embedded graphic framework and also between their users, providing a limited number of instructions easy to understand. This Human Machine Interface (HMI) aims to allow a hardware developer to manage a Graphical User Interface (GUI) without knowing as it is realised and also vice versa, allowing to an embedded graphical developer to handle peripherals, of a target hardware, without finding out its structure.
The work, which thesis consists on, is the creation of an interface layer between Graphical User Interface and target hardware. Implementation

of Software Interface Layer begins by studying a diffused tool for creation of Graphical User Interface: TouchGFX, of STMicroelectronics. First of all, Designer has been studied, with all widgets available, understanding their functioning and the architecture that GUI has when code is automatically generated by the software. Then, with new information available, it has been implemented an interface, which will simplify interaction between graphic application and surrounded system. Thus, the structure of interface has been thought of. It has been chosen to use two different queues to accumulate events directed in both directions: from and to User Interface. After that, work proceeded with implementation of public and private functions, necessary to transfer events from a side to the other. Since, TouchGFX gives the possibility to simulate the entire behaviour of GUI on PC, it is chosen to establish a connection, through that interface, with a Model-Based-Design tool for a complete prototyping. In order to do this, it was studied a communication protocol to allow a connection between simulator process, available on graphic tool and a Model-Based-Design software, so, functions to establish connection are created. Then, to demonstrate interaction with MBD software, it is chosen to use Simulink, in particular Stateflow, of Mathworks. Stateflow is studied and used to generate some clients, for example projects. After that, a variety of tests are performed on Human Machine Interface, in order to enhance functions and structure, to make it more reliable. At the end, it is decided to prove that interface may be general purpose, therefore, HMI is used on another graphic tool: Embedded Wizard. In this case, it works properly, without any modification on target, while for prototyper (simulator of EW), interface needs to be implemented in a Dynamic-Link Library.

HMI is based on *STMicroelectronics* products, using the relatively new graphic software framework, TouchGFX, optimized for *STM32* microcontrollers, thus it is used a STM32F429I-Evaluation board.

To realise the User Interface is used TouchGFX Designer, which gives the opportunity to simulate the created GUI with a suitable simulator, implemented reproducing the entire behaviour with the support of SDL

library which is available on the web.
With the Hmi is given the possibility to simulate the full behaviour, supporting TouchGFX with a model based design tool as Stateflow, emulating interaction between target and GUI with a Transmission Control Protocol(TCP) communication. Thus, allowing a complete simulation without the need for target hardware, but producing a full project entirely loadable in a microcontroller. This part is very important for prototyping because it permits to try a new application on a Personal Computer without having to buy the target, that during testing of an application may prove to be unsuitable.
The Hmi manages interactions of UI with surrounding system. Communication is acting in two directions from UI to external hardware and viceversa. When an event, that has to interact with the hardware, occurs it is putted in a queue. Then, event is taken by the other side (*Backend*) and corresponding peripheral is updated. On the other hand, toward UI, communication is based again on a queue, to which elements are sent but not instantly processed. Events will only be processed when the queue flush request is sent.
In the following chapters, necessary softwares for implementation of the interface will be discovered and explained in their main functionalities, as well as each step that lead to the creation of final interface and the complete structure used to manage the communication will be described in detail. Obviously, with final interface is intended the final version and not the perfect version of it, because some improvements are always possible in terms of structure, algorithms and performance. Hereinafter, it is reported a list, with brief description, of chapters that will be covered in the text:

- In the second chapter, TouchGFX is explained in its structure and organisation. Describing how data can be managed on User Interface applications. A section of this chapter explains how to create a simple graphical interface and how to use it with the CubeMX package.

- Next step tells about general concepts of Software Interface Layer. Explaining how communication, between sides, is structured and managed. Also principal actors of HMI are presented.
- Fourth chapter describe Model-Based-Design tool used to interface graphic tool for a full prototyping. It explains how to implement a simple client and how to generate code loadable on target.
- In the fifth chapter, there is the documentation of interface. Each public function is explained in its behaviour, with necessary parameters and return value.
- Chapter number six reports a full example of Smart Home application used to demonstrate how interface works and how events are managed.
- In this part, generalisation of interface is proved. Testing it with another software and explaining the necessary changes to make.
- Last chapter lists some improvements, that might be done to the interface to increase performance and reliability.

CHAPTER 2

TouchGFX

TouchGFX is a graphical tool that enables developers to easily create a smartphone-like GUI to add on embedded devices. The software is available in a package composed by three main parts, two tools and a framework:

- Designer: an easy-to-use GUI builder in TouchGFX that lets you create the visual appearance of your TouchGFX application.
- Generator: is a package plugin used to configure and generate custom TouchGFX Abstraction Layer(AL) for STM32-based hardware.
- Engine: is a C++ framework capable of driving User Interface(UI) applications. It handles screen updates, user events and timing. Advanced TouchGFX technology is optimized for STM32 microcontrollers, giving maximum performance with minimum CPU load and memory usage.

2.1 TouchGFX Installation

Software CubeMX package can be downloaded by the official website of STMicroelectronics and easily installed following the corresponding guide section of TouchGFX documentation[1] or steps reported subsequently:

1. First of all, make sure to have an IDE on which Visual Studio C++ or GCC compilers can run. If it is not, it is possible to install the

STM32CubeIDE that helps with implementation of GUI but also with initialisation code to flash in the board.

2. Then, to install editor for GUI, download zipped package by clicking on *"Get Software"*, as in fig.2.1,
   at *https://www.st.com/en/embedded-software/x-cube-touchgfx.html* and extract it.

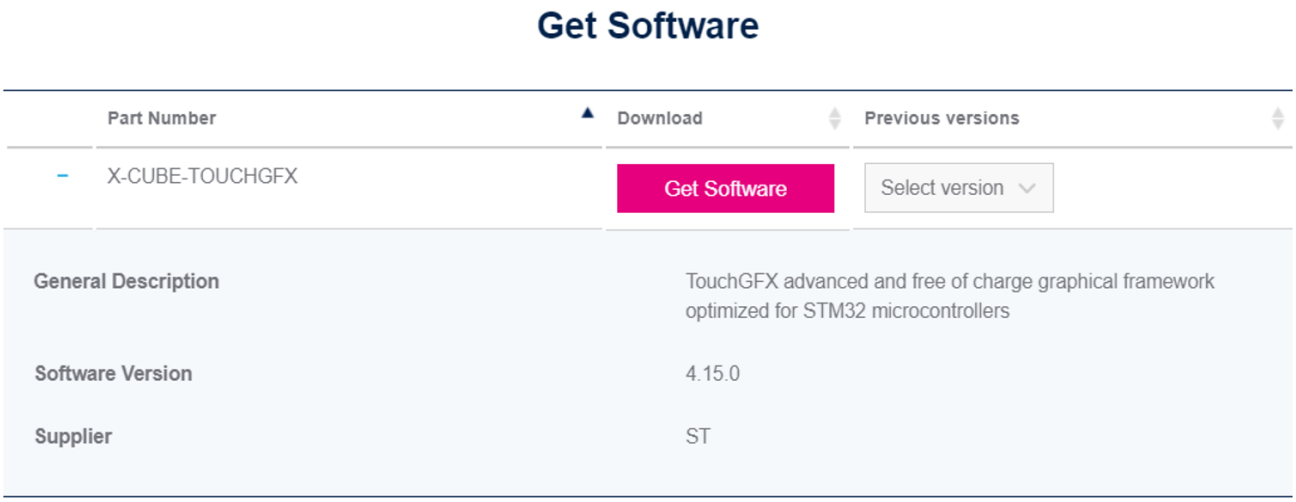

Figure 2.1: How to download the CubeMX package.

3. In the extracted folder under path *Utilities/PC_software/TouchGFXDesigner* there is an installer with <software_version_number>.msi, double click on it to start installation. Follow instructions to complete successfully.

4. Then, to be able to flash the board other two tools can be necessary: STM32CubeProgrammer and STM32 ST_LINK Utility.
   Download CubeProgrammer at link *https://www.st.com/en/development-tools/stm32cubeprog.html*, unzip the file and launch the executable.
   It has to be installed at default location to allow TouchGFX and Makefiles to use programmer for flashing a board.

5. Now, download ST_LINK Utility at *https://www.st.com/en/development-tools/stsw-link004.html*, uncompress and execute the unique *.exe* file

into folder. Again, install tool at default location for flashing target hardware correctly, with TouchGFX Designer and Makefiles.

6. Finally, to install TouchGFX Generator open STM32CubeIDE, installed in the first step, in the *"Help"* palette click on ***"Manage Embedded Software Packages"*** as shown in fig.2.2

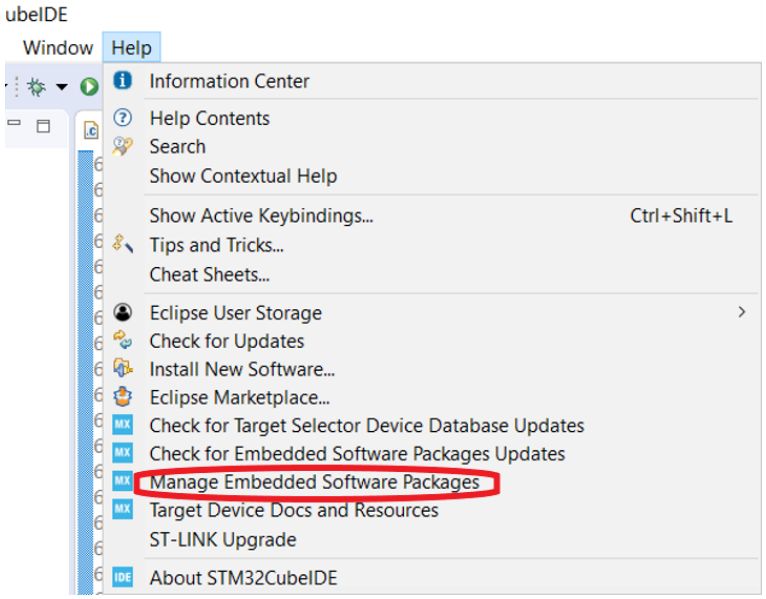

Figure 2.2: Help menu with evidenced option to select.

7. A pop-up window opens, the one reported in fig.2.3. Click on refresh to have an updated list of available packages, go to STMicroelectronics tab and scroll until X-CUBE-TOUCHGFX. Expand it, check the white box for *TouchGFX Generator* and click on *Install Now*. This will download the package and bring up the license agreement, read and accept it, then click on finish to install the tool in CubeMX.

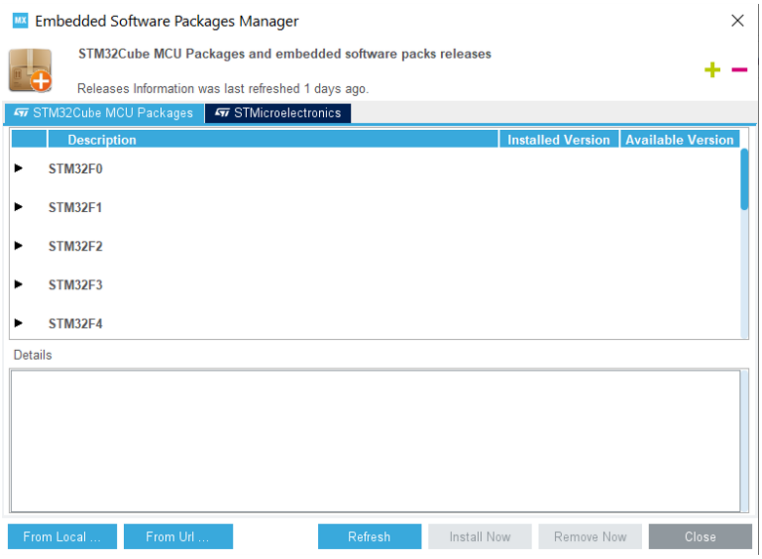

Figure 2.3: Pop-up window appearance.

2.2 TouchGFX Embedded Graphics

There are many interpretations pf *Embedded Graphics* terms, it spans from oldest to most recent technologies. TouchGFX, with *Embedded*, indicates any system based on an STM32 microcontroller, while *Graphics* means an interactive application with user interface running at 60 frames per second, this is the meaning that software developers give.[2] In an embedded systems, there are many components, but four are essential for graphic representations:

- Microcontroller Unit(MCU)
- Random Access Memory(RAM)
- Display
- Flash Memory

The MCU takes images from flash memory, calculates resulting colour merging images with a semi-transparent red text and save it into RAM. Then, MCU transfers images from RAM to the display. Some microcontrollers have specific capabilities to help realization of graphic like Chrom-ART, LTDC and so on. The RAM is read and written any times

a graphic update is necessary. Resulting images are stored, by default, in RAM internal to the MCU, but if it is not feasible, an external RAM can be added to the setup. Flash memory stores all static data: images, texts and fonts. In some cases it may be necessary to add an external flash to the setup. Next section will explain the complete procedure for adding external storage. Display receives calculated images from RAM and shows them to users, it is refreshed or updated at regular intervals.

How to add external flash memory For simple application with a limited number of widgets and screens, internal flash is enough, but if three or more screens are used and many images are stored in flash, it can be rapidly exhausted.
By default, images, fonts and texts are charged into internal flash memory, which is not really large, for instance, in F4x9I-EVAL board (used to test this tip), internal memory is of 2048 kB. For instance, creating a simple application with two screens: one with a clock and second for setting it, pretty half of memory is just used as showed in fig.2.4.

clock_settings_project.elf - /clock_settings_project/Debug - Nov 9, 2020 10:46:31 AM

Memory Regions | Memory Details

Region	Start address	End address	Size	Free	Used	Usage (%)
CCMRAM	0x10000000	0x10010000	64 KB	64 KB	0 B	0.00%
RAM	0x20000000	0x20030000	192 KB	151,7 KB	40,3 KB	20.99%
FLASH	0x08000000	0x08200000	2048 KB	1055,13 KB	992,87 KB	48.48%
NOR_FLA...	0x60000000	0x61000000	16384 KB	16384 KB	0 B	0.00%

Figure 2.4: Memory occupied by a simple application.

STM boards can have different types of external memory: NOR_flash, QSPI, NAND_flash and so on. Following steps describe how to enable external memory to store data and use them. In these steps it is adding a NOR_Flash, but the procedure is similar for other memories, like the one reported on a youtube video.[3]
First of all, open .ioc file in main folder of interested project, expand connectivity tab and click on FMC. In the right panel enable the checkbox for a suitable memory, like in image below.

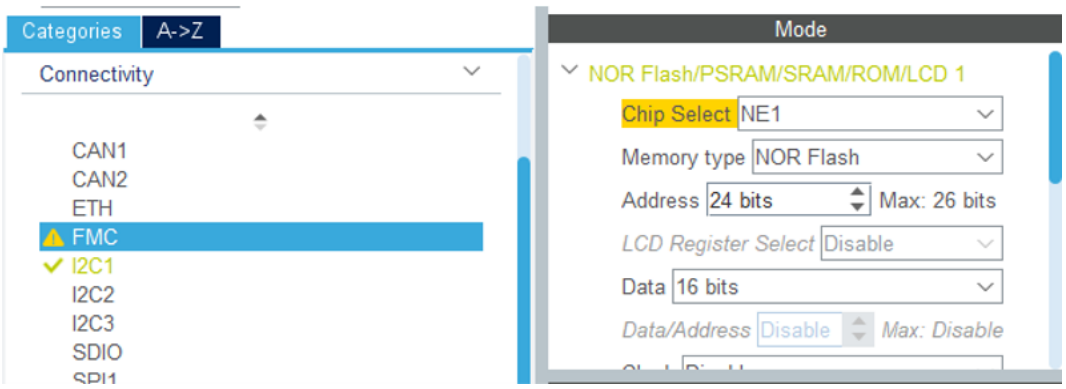

Figure 2.5: Enable memory available on the board.

After that memory is selected, it has to be initialised. This step is done automatically selecting *"Generate Code"* in CubeIDE but sometime errors can occur, so function generated for initialisation must be checked. In F4x9I-EVAL, board used to test this feature, CubeIDE generates code correctly.
At this point, loader has to be configured to save data into external memory. Open loader file <*board_name*>_FLASH.ld and check that added memory is present in 'memory definition' section.

```
62 /* Memories definition */
63 MEMORY
64 {
65   CCMRAM    (xrw)    : ORIGIN = 0x10000000,   LENGTH = 64K
66   RAM    (xrw)    : ORIGIN = 0x20000000,   LENGTH = 192K
67   FLASH    (rx)    : ORIGIN = 0x8000000,   LENGTH = 2048K
68   NOR_FLASH    (r)    : ORIGIN = 0x60000000,   LENGTH = 16M
69 }
```

Figure 2.6: Memory section with external flash.

Then, in the same file, go through, until added memory section loading is described. There are three portions of the new memory: ExtFlashSection, FontFlashSection and TextFlashSection. Now, it is necessary to uncomment the portion for which more space is needed. The first part is for images, second for font and third for texts as names suggest. For clock settings application, used to test this tip, only first section is

uncommented. Resulting portion of the file is shown in fig.2.7.

```
201
202   ExtFlashSection :
203   {
204     *(ExtFlashSection ExtFlashSection.*)
205     *(.gnu.linkonce.r.*)
206     . = ALIGN(0x4);
207   } >NOR_FLASH
208
209 /*
210   FontFlashSection :
211     {
212         *(FontFlashSection FontFlashSection.*)
213         *(.gnu.linkonce.r.*)
214     . = ALIGN(0x4);
215     } >NOR_FLASH
216
217   TextFlashSection :
218     {
219         *(TextFlashSection TextFlashSection.*)
220         *(.gnu.linkonce.r.*)
221     . = ALIGN(0x4);
222     } >NOR_FLASH
223 */
```

Figure 2.7: Three section for external memory.

Once memory is correctly configured, project can be rebuilt and the updated memory space observed, fig.2.8 reports the new percentage of memory use.

clock_settings_project.elf - /clock_settings_project/Debug - Nov 9, 2020 10:49:50 AM

Memory Regions | Memory Details

Region	Start address	End address	Size	Free	Used	Usage (%)
CCMRAM	0x10000000	0x10010000	64 KB	64 KB	0 B	0.00%
RAM	0x20000000	0x20030000	192 KB	151,7 KB	40,3 KB	20.99%
FLASH	0x08000000	0x08200000	2048 KB	1884,1 KB	163,9 KB	8.00%
NOR_FLA...	0x60000000	0x61000000	16384 KB	15555,03 KB	828,97 KB	5.06%

Figure 2.8: Memory occupied after that extrnal one is added.

Adding an external flash, as a NOR_FLASH available in F4x9I-EVAL, memory space is much more!

Sometimes can happen that with the new configuration, when project is run/debugged on a board, images are not shown. They have the right position but they are displayed completely wasted. This is, probably because CubeIDE does not program the flash with ST tools, so images are

loaded into memory addresses that are not the ones expected. To solve this problem, it is possible to use the CubeProgrammer or ST-Link that allow to program the external flash and attach the debugger. Otherwise, an external loader can be set directly in CubeIDE to have it each time a run/debug session needs. To set an external loader, go to run menu of CubeIDE and choose *Debug Configurations*, fig.2.9.

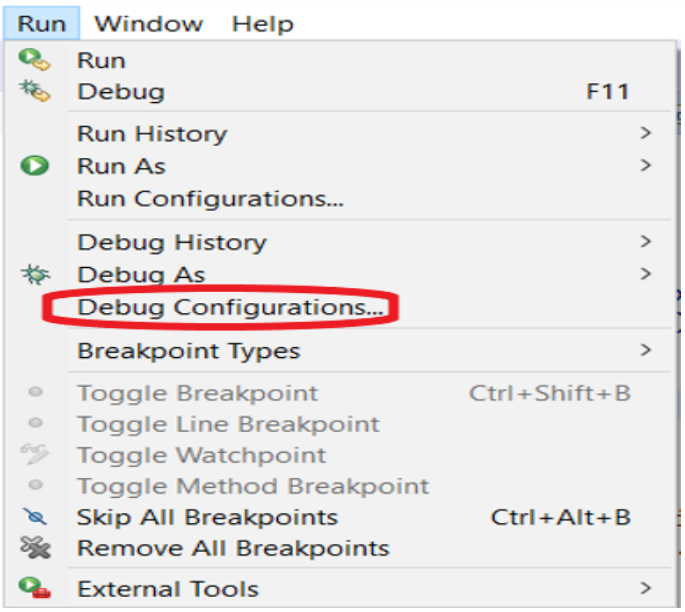

Figure 2.9: Debug configuration selection.

In the main tab, in C/C++ Application insert the project .elf file with its path. Into Project put the name of correspondent one and then go into Debugger tab. At the end of this tab there are some white boxes, select the one with external loader and press the 'Scan' button. It does not select a correct loader, but gives a list of possible ones. Select the right loader and then debug. Now, all images of the project will be shown correctly on the display of your board.

2.2.1 TouchGFX: Graphic Engine

TouchGFX has a *Retained Mode Graphic Engine*. This type of engine let the user to manipulate an abstract model that will be displayed with correct drawing operations performed at right time.

This approach has many benefits:

- It is **easy to use** because user operates, on component, invoking methods of model without thinking in terms of actual drawing operations.
- **Increase performance** allowing to draw only portions of visible components, managing framebuffers and much more.
- **State management**: keeps track of which part of scene model is active.

The main drawback, instead, is **memory consumption**. TouchGFX reaches performance levels at 60 frames per second, analysing scene model and optimizing the corresponding rendering done. Great efforts are involved to reduce the amount of memory used by the scene model, typically it is well below one kilobyte.

2.2.2 Main Loop

TouchGFX Graphic Engine can be thought as an infinite loop composed by three main activities executed in a tick:

Collect Events In this phase, inputs like touches on the screen, pressures of physical buttons, messages from backend and much more, are taken. Raw touch events are translated into more specific ones like Click, Drag or Gesture. Each event is forwarded to the User Interface elements which are currently active. Also a tick event is forwarded to perform time-based actions.

Update Scene Model This phase consists of reacting to input events, updating positions, colour, animation and so on, of components in the model. For instance, if a button on the screen is touched, it can modify its image to a pressed state, this change is notified to the engine. When a widget(UI element) needs a redraw, the area that it covers becomes invalidated.

Render Scene Model It takes the list of invalidated areas and redraw parts of the model that has been updated and make them appear on the display.

Wait This is not a real phase of the main loop, but it has a great importance in engine work. TouchGFX waits a signal before proceeding with updating and rendering of next frame. This allows to synchronize rendering operation with display, guaranteeing that framebuffer is updated only after rendering of previous modification. Graphic engine waits for a short time, after transmission is started, before rendering. Another benefit of waiting is to have frames rendered at a fixed rate.

2.2.3 Handling Framebuffers

There are two different possibilities to update and render a display depending on number of framebuffers available on the board. If there are **Two Framebuffers**, graphic engine alternates between the two. While one is drawing, the other is transferred and showed on the display. Sometimes can happen that a framebuffer is not modified so, there is nothing to render, in this case the same framebuffer is transmitted in next frame. If rendering time is higher than update frequency, the framebuffer is sent again in next frame and collect and update phases wait. With **One Framebuffer**, instead, process is forced to transmit the same buffer each frame. This is very risky because can happen that the framebuffer is a mix of previous and new frame. One possible solution is to hold back the drawing until the transfer is complete and only draw during timeslot before that transfer starts again. A drawback could be that incomplete frames(*tearing*) might still occur if drawing is not complete when transfer starts again. A more potential solution is to keep track of how much of the framebuffer is already transmitted and then limit the rendering to the appropriate part of the framebuffer. For more information about framebuffer strategy consult the documentation.[4]

2.2.4 Memory Usage

Typically, a TouchGFX application uses four types of memories: **Internal and External RAM** and **Internal and External Flash**. The entire memory space used by an application is preallocated, none is allocated at run time. This guarantees that if an application, initially, fits the memory, then, it cannot run out. So, TouchGFX uses a ***Static Memory Allocation***.
If an application has multiple screens, only one at a time is allocated in internal RAM, when a screen is deactivated its memory space is overwritten by new active screen and its widgets. TouchGFX libraries are stored into RAM, only if used, for instance if a button is inserted, corresponding class is allocated in memory. C++ compiler calculates the size of largest screen and reserves memory space for classes which compose that specific screen.

2.3 TouchGFX Project Development

A TouchGFX project is composed by five main software and hardware components:

- Display Board
- Board Initialisation Code
- TouchGFX Abstraction Layer
- TouchGFX Engine
- TouchGFX User Interface Application

Figure 2.10: TouchGFX project main components.

Each of these elements is given by a specific activity that make up the entire project development process.
Starting point is TouchGFX engine, downloaded and installed along with the whole package.

Hardware Selection Through this activity, the hardware, on which created project will be executed, is selected. If project is not really defined, it is possible to use an evaluation or a discovery kit offered by STM32, otherwise the TouchGFX simulator can be used, running the project directly on the PC. They have several peripherals available to use, so when all necessary hardware is known, a custom board can be used.

Board Bring Up This step enables the TouchGFX project to run on a board. The output of this step is Board Initialisation Code that initialise MCU and its peripherals. Using CubeMX it is possible to easily configure hardware needed for an application. CubeMX allows to enable peripherals, setting mode for each pin of the board and many other configurations just searching the component into a list and selecting parameters concerned. When the setup is completed, it needs to click on *Generate Code* to get initialisation code. TouchGFX also provides many Application Templates(AT) with their Board Initialisation Code, so if an available kit is used, it is possible to copy the code of a template without generating it.

TouchGFX Abstraction Layer Development This activity permits to the engine to run on hardware. This is performed by TouchGFX Generator which is a CubeMX plugin that gives the opportunity to configure and generate most of the Abstraction Layer(AL) code. TouchGFX AL will only work if MCU and all peripherals are correctly configured. This step can also be supported with an available Application Template given by the software. Important responsibilities of AL are:

- synchronize update of framebuffer and transfer to the display, ensuring correctness of main loop;
- report physical and touch events;
- synchronize framebuffer access, protecting this portion of memory and also, in two framebuffer setup, AL reports the area to update next.
- perform render operations, for instance, when MCUs use Chrom-ART graphics accelerator.

User Interface Application Primary tools for this activity are TouchGFX Designer and an IDE or text editor. At this step, the User Interface code will be created that will form the visible part of TouchGFX project. In

the Designer, main parts of UI application are set up, designed and created using C++ code. IDE or text editor is used for application logic, the non-UI part that interacts with code generated by TouchGFX Designer, for instance by creating callbacks that are used to transfer data through framework classes. This will be explained better in an appropriate section.

In each of the three software activities code will be generated by used tools. It can run independently, but usually, user code must be integrated to complete the application. For example, to take inputs from board, user has to write a sample function or to have a specific output on UART or other peripheral, the respective behaviour must be programmed. CubeMX toolchain helps in the configuration of device with automatic generation of initialisation code. There are also many downloadable examples from ST official website.[5]

2.3.1 Abstraction Layer

Abstraction Layer Generation

Particular attention is focused on the Abstraction Layer due to its important role in the development chain. AL is created by Generator, accordingly with the used board, previously set in CubeMX. Before being used, Generator has to be enabled in CubeMX. In CubeIDE open the .ioc file, in the top light blue bar, select *Software Packs* and click on *Manage Software Packs*, a pop-up window opens. Go into STMicroelectronics tab and scroll until *X-CUBE-TOUCHGFX*, expand it and check the whitebox near to **TouchGFX Generator**. After enabling, it will appear in the left menu of .ioc file under 'Additional Software' and when code will be generated a TouchGFX folder will appear in the current project.
Generator identifies three groups in the user interface of CubeIDE:

1. Dependencies contain warnings and errors in the configuration.

2. Display contains all characteristics about display of used board like colour depth, height and width. These features are reflected on the canvas inside TouchGFX Designer. This group configure also interface of display like SPI, FMC or Parallel RGB, buffer strategy and if necessary, also the buffer location.

3. Drivers related to tick source application(to drive application forward), graphic accelerator and RTOS(Real-Time Operating System).

Abstraction Layer Architecture Responsibilities of this layer could be implemented in the hardware part (Hardware AL) or with the part synchronized with TouchGFX Engine via an RTOS. HAL is part of TouchGFX framework and with generator, two generated classes are obtained *TouchGFXGeneratedHAL* and *TouchGFXHAL*, which is the only one that can be edited by user.

Synchronization between engine and display Main concept of this step is to block Engine Main Loop when rendering is done, avoiding production of further frames.
Rendering done is notified by an Engine hook, with same name, which calls *OSWrapper::waitForVSync()* after rendering is complete, now AL blocks the graphic until next frame is ready to be rendered, so when *OSWrapper::signalVSync()* is called. Generally, this block is implemented with a blocking read of a message queue. SignalVSync, usually, is called with an interrupt from a display controller, from the display itself or from a hardware timer.

Report touch and pyhsical button events Before rendering a new frame, engine collects inputs from ButtonController and TouchController interfaces. Touch events are translated into click-, drag- or gesture events and transferred to application.

Touch events are collected in two different ways:

- sending a request to TouchController, to know the status of touch, and polling for the result. This is done through I2C and takes about 1ms impacting on the overall render time;
- raising an interrupt regularly with a timer or when a touch hardware occurs.

2.3.2 User Interface Development

TouchGFX User Interface follows the architectural pattern known as Model-View-Presenter. Main benefits of this structure are: **separation of concerns** which means that code is divided into independent parts, making code simpler, reusable and easier to maintain; **unit testing**, since logic is separated from visual layer, it is easier to test isolated parts.
The structure of UI, reported in fig.2.11, is organized around three main classes:

- Model: is always-alive singleton class. It stores data for UI, particularly that which are preserved across screen transitions. The Model acts as an interface toward backend system, relying events from and to the active screen.
- Presenter: acts as interface between model and view, it takes data from model, formats and sends them to the view.
- View: is a passive interface to display data that arrive from the model and routes user commands to the presenter to act upon that data. It contains a setupScreen called when the screen is entered, typically used to configure widgets. It has also a tearDownScreen function executed when screen is exited. Framework automatically sets up a pointer to the respective presenter to notify UI events.

Another important feature of model is to handle communication with non-UI part, the so called backend system. Backend layer is a software

component which receives events from UI and feeds others into UI.
For more information about abstraction layer development read proper sections of documentation.[6]

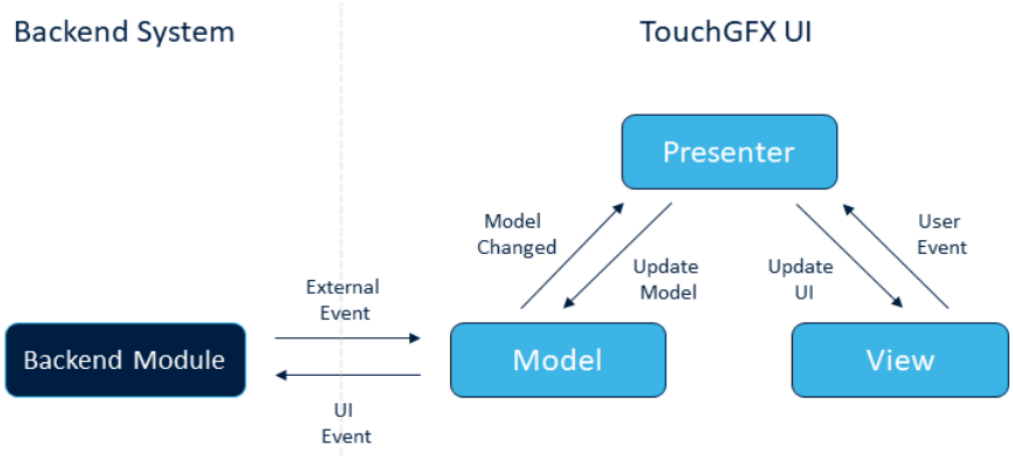

Figure 2.11: Model-View-Presenter structure.

Screens

An important concept in TouchGFX is the **Screen**. It is a group of UI elements and their associated business logic. Each screen has two classes: a **View** with all widgets appearing on it and a **Presenter** with corresponding business logic. There are no rules on how to divide screens, but separation it is generally adopted for visually and functionally unrelated widgets. For instance, in a clock application, it is possible to have two screens: main one with the clock and second with settings.
As mentioned in a previous section, a portion of memory equal to the largest screen is allocated in internal RAM, so no more than one screen is active at a time. Likewise, received events from backend or hardware peripherals are delegated to current visible screen. If a received event wants to act on a non-visible widget it is simply "discarded".

Code Structure

TouchGFX Designer like the other chain tools, allows a partial code generation. In the Designer, it is possible to create UI applications with a

drag and drop approach and then, by clicking on *Generate Code* all base classes are created in the project folder.
In this case, the code generated by Designer is completely separated from user code. The entire generated base classes are placed in *generated/gui_generated* folder and contain the initialisation code configured in the Designer, these files are not editable. On the contrary, user code will be placed in *gui* folder.
TouchGFX Designer application consists of three main layers of code:

- Engine: groups standard classes provided by TouchGFX. They are the base for generated classes.
- Generated: classes and corresponding files are regenerated whenever generate code, in the Designer, is clicked, therefore, this layer is no manually editable because files are overwritten at each run of code generator.
- User: these classes derive from generated ones and are intended for handwritten code. Designer does not alter in any way this portion of code.

2.4 TouchGFX Designer

2.4.1 Widgets and Containers

TouchGFX Designer is the tool which is in charge to help the developer generating user interface application. As said before, it uses a *drag and drop* approach to simplify creation of screens.
This section will describe in detail how to use TouchGFX Designer to implement an application. Any precise references will be to the version **4.15** of the tool .
Designer can create new projects, but also open existing ones or exploring demo, available for different types of boards, accessible widgets and for some typical hardware devices.
To create a new application needs to write a name and location of the root folder. Then, two very important steps: first, select which board the application will be for; second choose the starting point: initiating from a blank user interface or with one already implemented. Last fields are filled accordingly with the chosen application template. For instance, for a STM32F429I-EVAL board, colour depth is 16 bits, $width = 480$ and $height = 272$.
After this initial configuration, TouchGFX Designer is ready to use. A white rectangle appears in the window, which is the canvas, the application screen. Everything inserted in will be shown on display.
Designer supplies a wide variety of widgets. **Widgets** are simply an abstract definition of something that can be drawn on the screen and interacted with, in other words they are objects that may be inserted into screens and used to build a user interface, for the complete list it is demanded to visit TouchGFX official documentation on "Ui components" section[7]. There are *boxes, images, text areas* and many other passive objects utilized to show something, but also interactive widgets that user can modify to provide a value to the UI such as different type of *buttons, sliders, scroll wheel* and so on. Of course, if necessary, new widgets may be implemented from scratch or starting from a template.
Many of these input widgets can react to user event with a special ac-

tion triggered whenever a specific event occurs. They are called **interactions**.
Interactions are characterized by a **Trigger**, an event that initiate the interaction and an **Action** which is what happen after the stimulation. Many standard actions are available, they consent to move widgets, switch screens, execute external code or call functions and many others. Again, if UI needs of a special trigger or action, it can be created.
There is also another important widget category, the *container* which can group child nodes as widgets or other containers. There are many useful containers in Designer and also others can be designed by the user.

2.4.2 Images, Texts and Fonts

Designer gives the possibility to customize appearance of widgets and screen backgrounds, by adding corresponding figures in the *Images View*. For instance, a button can have different styles, choosing through the available set of standard images, but it can be more personalised loading a picture into Designer and selecting it for that button. Images are part of the application *Assets*, they must be inserted as *.PNG files and will then be converted by an *ImageConverter* to C++ language. Images can be stored in *RGB565, ARGB8888 and L8* formats. This allows to send unchanged pictures to framebuffer. Direct Memory Access can be used to partially save microcontroller load. It is not used when images have an alpha channel, because it is necessary to merge pixels of image with those of framebuffer (*pixel blending*), in this case DMA is used only if board has a graphic accelerator like Chrom-ART/DMA2D.
Also texts used into UI are customizable, typographyc parameters can be changed in *Typographies* menu. TouchGFX supports three different types of alignment and writing like LTR(Left-to-Right), which is the default one and RTL(Right-to-Left). When UI is built or simulator launched, configuration of entire text database is reported in a spreadsheet. After that, database is provided to *textConverter* which generates a *.CPP file with texts converted into a format usable by TouchGFX.

It supports all languages of Unicode Basic Multilingual Plane, but only with RTL and LTR orientation, it is not feasible for top-down writings. TouchGFX uses a 16-bit Unicode coding. Languages that require special reordering or positioning are supported in a limited way, moreover, TouchGFX has a set of rules for combining characters to represent contextual shaping words as reported from STM.[8]
Fonts follow the same procedure as texts.
These three conversions are carried out when building the user interface.

2.5 Compiling and Flashing

Executing program is the goal of developer, so it needs to know, how to compile and flash TouchGFX application in a specific setup.
Code can be compiled for PC simulator, to have a program running on the PC or for target hardware to obtain code usable on a board.

2.5.1 Compile for PC Simulator

In this case, code is compiled with GCC or Visual Studio. Executing the Makefile, automatically generated by Designer in the project folder, with TouchGFX Environment, included when graphic tool is installed, a *simulator.exe* file is created and then, it is possible to launch simulator. These steps can be executed directly inside TouchGFX Designer clicking on *Run Simulator* in the top right of window.
Instead, to compile code with Visual Studio, it is enough to open the solution *Application.sln*, placed in *<touchgfx_project_folder>/simulator/msvs/*, and run or debug the code.

2.5.2 Compiling for Target Hardware

Each application template generates projects for GCC, CubeIDE, IAR and Keil. First of all, code has to be compiled in the same manner that for PC simulator, by executing Makefile for target hardware in TouchGFX Environment or by clicking on *Generate Code* into Designer. CubeIDE, as many other IDEs, also compiles code for target.
Each built project generates a binary file that must be flashed into target with CubeProgrammer or ST-LINK. In previous sections is explained the procedure to install them.
It is possible to write the internal flash, if large enough, significantly reducing flash time, otherwise external flash must be enabled with the procedure explained into section **2.2**. Target can be flashed easily by clicking on *Run Target* in Designer. Using CubeProgrammer it is possible to load memory also within the CubeIDE, simply putting the *.ELF file in debug configurations.

2.6 TouchGFX PC Simulator

According with documentation[9], creating a TouchGFX UI can lead to a lot of difficulties to match UI specifications and also flashing a board can require quite some time, so doing this, any time that a small change is done on the application, it is very boring. The TouchGFX PC Simulator is a great add-on tool to avoid these problems.
Simulator, simply compiles the application for the PC and run it there. Project can be tested entirely for UI part such as placements of widgets, interactions, animations and so on, with the same accuracy that it might have on the target hardware. Backend part, instead, cannot be verified because Board Bring Up code and Abstraction Layer are made for the PC and not for target.
It is even possible to easily debug the application with Visual Studio or GCC compiler, which opens up to many other IDEs, but again performance analysis and interactions, with real backend system, must be done only with a board connected.
This drawback will be fixed with the Software Interface Layer which allows to support simulator with a Model Based Design tool, but details and full explanation will be given in next pages.

2.7 TouchGFX Backend Communication

The structure of TouchGFX framework permits to dedicate the Model class to interaction with the rest of embedded system. This is a very useful aspect and it is used in many applications because this connection allows to exchange data between UI and backend, interfacing hardware and software or software modules, when communicating with Operating System.
Model class is suitable to interface with surrounding hardware because:

- It has a ***tick()*** function, automatically called every frame by an interrupt request of LCD, so it could be used to look for or react to external events.
- Model has a pointer, of ***ModelListener class***, to the currently active presenter, that could be used to notify UI of incoming events.

Furthermore, model is utilized to store data, for instance to exchange data between screens. Due to this is fundamental to understand the correct way to transfer data from model to a view of application and vice versa.

2.7.1 Data from View to Model

For View to Model direction, communication is guaranteed by the Presenter class, in fact, a pointer to the respective presenter is instantiated by default, into view. Using this pointer, the View can call a method of Presenter, passing, as argument, the value to transfer. Now, value is in the middle, but as well as to view, also Presenter has a pointer(named *model*) to the Model class to call its methods, passing relevant data and saving them into Model. Therefore, data arrived in the singleton class can then be accessed by other screens.

2.7.2 Data from Model to View

In this case too, communication works with pointers. Model class has a special class, *ModelListener* for interacting with Presenter, which inherits it. ModelListener has to implement all methods used in this type of communication, but they are usually left empty to not transfer any data when the method is called while screen, that has to receive data, is not active. Instead, when screen is visible, the method definition into ModelListener is bypassed, because of inheritance, the method is overridden into Presenter. After that, Presenter class has a view pointer to call methods for updating visual objects.

2.7.3 External Event Sampling

In many applications, data to be transferred from Model to View, comes directly from backend. On target hardware there might be physical button presses, a switch state change or something else. All of these events can be used to update UI, from a state to another.
To react to external events, Model will sample them. For this purpose, two distinct ways may be followed.
The first method is a *"quick and dirty"* approach, usually intended for prototyping. This first solution is very simple and consists in sampling directly into the Model::tick() function. This method may be time consuming and also time crucial, sampling operation must be fast, typically 1ms or less, otherwise frame rate starts to suffer, since sampling is done in the GUI task and it will delay the drawing frame.
Second approach is architecturally better, accordingly with STMicroelectronics webinar[11], it allows to properly link the user interface with the remaining components in a real-world application. This method is based on a new Operating System task responsible for sampling events. In this case, if sampling is time crucial, it is possible to set an higher priority with respect to the GUI task. With higher priority, task will run exactly with specified timing, but in this case if it is a CPU consuming process, it affects the frame rate.

For a large variety of applications, a lower priority is sufficient, because GUI task spends much time for rendering and during this time, lower priority tasks can be executed. To manage communication between tasks may be implemented an inter-task messaging system, using the available Real-Time Operating System, such as mailboxes or message queues.
This part is fundamental to understand implementation of the Software Interface Layer, because it is based on the simplification of this type of interaction from the point of view of graphical and hardware developers.

2.8 Create a new project

At this point, it is very important to clarify a concept that has characterized the IDE. In this case, referenced IDE will be the CubeIDE of STMicroelectronics. This IDE has many tools plugged in, like CubeProgrammer which is briefly described in previous section, but also CubeMX, this software allows board configuration, selecting the used board every time that a new project begins. This part might be simple, but there are at least two main ways that can be followed.
Probably, the clearest, but surely not the fastest is to open the project in CubeIDE, configure all peripherals from scratch and generate code. At this point board initialisation code is ready to be flashed, but GUI has to be realised yet. Therefore, add the TouchGFX Generator to the project and regenerate the code in order to have also a TouchGFX project inside the root folder. Then, opening the TouchGFX file, into the folder with the same name, realisation of GUI can start. With this method, the first part is quite crucial because by selecting the board in CubeMX all pins can be set to clear or default mode and this does not match at all the configuration needed to run a GUI project. So, it is necessary to set up all required hardware and it is not very easy, especially for memories.
Another approach is based on existing templates. This is the easiest and probably the fastest way to follow when creating a new project. Also in this case generate the new project within CudeIDE, selecting the board. Then, create a new application into TouchGFX Designer, it is necessary to select the same board chosen for CubeIDE, as application template while Ui one can be empty. Then, click on *"Generate Code"* and proceed with following steps that are not so difficult, but must be done precisely. Rename the configuration file of TouchGFX template (*.ioc) with the same name given to CubeIDE project, open it with a text editor. Going through this file, some lines starting with *ProjectManager* need to be modified:

- on *ProjectFileName* change the name of .ioc file to yours (e.g. <myProject.ioc>);

- again the same name, without extension, has to be set in the *ProjectName* line;
- finally, *UnderRoot* line has to be set to true.

After these changes, save the file and replace the one already present in the main folder of CubeIDE project. Open the configuration file within IDE and, ensuring that full setup is correct and TouchGFX selected, generate the code. Now, a TouchGFX folder is created in the root directory. Inside TouchGFX there is "ApplicationTemplate.touchgfx.part" file, double click to launch it and then, UI may be implemented or left blank for the moment, code must be generated to create all necessary folders for the GUI application. At this point, it is necessary to copy and replace, the flash loader, of previously created TouchGFX application, with the homonymous one within CubeIDE. Then, from driver folder copy the Board Support Package (BSP) directory from TouchGFX application to CubeIDE project, again in the "Drivers" folder. After that, compiler has to know that library, added before, needs to be linked. Therefore, go into properties of project and in "Path and Symbols" under C/C++ category, add for all languages the paths for Drivers, BSP, Components and its four sub-folders. Now, touch has to be enabled, in order to do this copy from "target" folder of TouchGFX template, files dedicated to TouchController, both .cpp and .hpp. Files with same name within CubeIDE project have to be replaced, they are located in <root_project_folder>/TouchGFX/Target. Finally, the last file that has to be copied is the main.c, inside Core folder of template to the Core of project, then compile the project and all is configured. It's time to develop GUI and Firmware code. In order to better understand steps to follow it is possible to see a tested video guide, which reports all these steps.[10]

CHAPTER 3

Software Interface Layer: Main Concepts

With the graphic environment described in previous chapter, developing a Graphical User Interface could reveal expensive, especially if using the CubeMX toolchain, at some point in the design phases, board chosen at the beginning becomes useless and there is a need to change it. Furthermore, development is only partially completed because, using the simulator with Visual Studio or other IDE, GUI part is fully checked, but it is not possible to verify the hardware side, nor its interactions with the GUI.
The implemented Software Interface Layer is required to solve these problems. It aims to simplify the management of communication between two sides of an application as much as possible, at the eyes of developers, by giving them a limited number of instructions easy to understand.
In addition, the Human Machine Interface(HMI) permits to support the graphic environment with a Model-Based-Design tool such as Simulink, allowing generation of a model to simulate what happens on the board and producing, with another tool (Embedded Coder for Simulink), C code, which can be flashed into board.

Each GUI application, as explained before, can be run directly on target or on the PC simulator. In this two circumstances, the interface works differently: on target, all code is loaded in the same chip, so code is fully available and communication can be managed locally, instead, with PC

Simulator transmission is handled through Transmission Control Protocol(TCP), where the server is the TouchGFX Simulator running on Visual Studio, while client is the Simulink model, created specifically by developer, to support the Simulator.
The idea behind the implementation is that, when an event occurs, it takes place on a particular object (of backend or UI), this event is put in a queue with its modification value. Then, the other side receives the event and looks for it on a table indexed by event identifier. In same position there is a value, linked to that identifier, which is modified with the one received and then, through a function pointer(always stored in table) modification is applied on target object.
In following pages, all steps of interface process are analysed for a better comprehension of implementation.
The structure thought for the interface is reported below in fig.3.1.

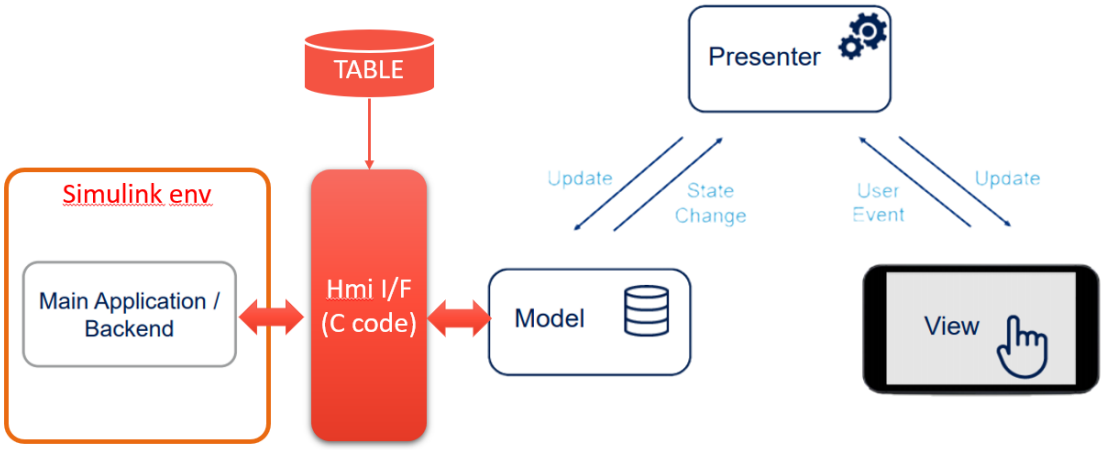

Figure 3.1: Model of Software Interface Layer.

In the picture are highlighted the main blocks of interface, fully programmed in C language, to have suitable code for many types of target hardware. An important block, represented in red in the image, is the *Table*. It needs to store information relative to "interactive objects" of both sides. There are two tables, one for UI and one for backend, each with its particular structure:

- **User Table**, fig. 3.2, is more complex, it stores information about events directed toward UI. Rows are indexed by event identifiers of

hardware peripherals with four fields each. Two possible types of event, integer and string, a section to store the function pointer and at the end event type.

	A	B	C	D	E
1	User_eventID	evValueInt	evValueTxt	Callback	evType
2	switch_playStereo	0	NULL	clbk_c_toggleStereo	evINT
3	switch_lightLiving	0	NULL	clbk_c_switchLightLiving	evINT
4	switch_livingShutterDown	0	NULL	clbk_c_downShutterLiving	evINT
5	switch_livingShutterUp	0	NULL	clbk_c_raiseShutterLiving	evINT
6	sensor_livingTemp	20	NULL	clbk_c_livingTempMeasure	evINT

Figure 3.2: Filled user table example.

- **UI Table**, fig. 3.3, is simpler, it has only three fields, one for modification value of double data type and a function pointer. Last field is used to take into account a consecutive reception of events: it is a counter checked when each event is received, in order to avoid processing the same event more than once.

	A	B	C
1	Ui_eventID	evValueDb	Callback
2	tglBtn_play	0	switchStereo
3	tglBtn_lightLiving	0	handleLightLiving
4	rptBtn_shutterLiving	0	setLvlShutter
5	scrollWheel_tempLiving	22	setTempLiving
6	slider_lightIntensityLiving	25	NULL

Figure 3.3: Example of table for GUI side.

The interface works differently depending on whether it is running with PC simulator or on target. Communication is handled through two queues, one for each direction. On sender side, events are pushed into queue, the other side receives those events and with function pointer, related to a specific event identifier, an action is performed on an object of receiver. Tables are important for storing "actual" state of a widget or peripheral. Each table with its fields, listed below, is filled in by developers, copied into spreadsheet and then, provided to a Python script which results in different files, that are the compiled version of tables. Each spreadsheet must follow the same rule:

1. The first row is occupied by name of fields of table structure;
2. other lines are for objects, one per line;
3. The first column is occupied by the event identifier used to create the list of possible IDs;
4. Next columns are, in order: integer event value, text event, callback and event type for table of events directed to UI side, while toward user there are double event value and callback.

3.1 Interface Work on Target

As explained in the previous chapter, a TouchGFX project can interact following two direction: from UI to backend and on the contrary from backend system to UI.

3.1.1 Events Toward UI

Events, directed to UI, are generated on surrounded system through hardware peripherals like physical buttons, joystick, Serial Interface, Controller Area Network(CAN) and so on. When event is triggered, it has to be sent to UI through the *UI Queue* of Software Layer, therefore, a public function of HMI must be used: ***Hmi_SendUiStateXY***, where XY can be substituted with Int or Txt, based on type of value to be transmitted, also inserting two parameters: identifier and value/text. Using named instruction, a data structure is queued.
Queue is a simple array whose elements are of a data structure type (*HmiEv2UiType*) similar to that used for table, but with fewer fields. Each element in the queue has an identifier, to distinguish one event from another, and an integer value or text string.
In this direction of communication, events can be pushed continuously by filling the queue, only when the flush of queue is requested, with the ***Hmi_Flush*** command, the User Interface takes events, one at a time, and processes them. When queue is full, it is no longer possible to store events and excess ones are simply ignored.
From UI side an instruction of Hmi is called, to check a flag in polling, this is done inside *tick()* method of Model class to execute the control regularly. When that flag assumes *FLUSH_STATE* value, UI calls ***HmiUi_ProcessEvents***, so it starts taking elements from queue, one by one, with a private instruction. Each taken event is processed, again with a local function, supported with information in the table. First, the value received from queue, is used to update the corresponding one in table, accordingly with event type, last field of table row. For text events

it has been chosen to copy received string, into table, directly when event is entered in the queue, to avoid heavy weight on queue. Then, respective function pointer is called to execute corresponding instruction. In this case, pointed functions, which are implemented in C language, but inside a C++ file, must be linked to method of Model class to activate the transmission of data toward View, to modify a widget of the application. Later in the text, how function calls method Model will be clarified.

3.1.2 Events Toward Backend

Opposite direction is in charge of transferring events, arising on UI, toward backend, calling a final function to act on a specific target peripheral.
In this case, when a touch event occurs, it is propagated from View to the Model via member methods and in the corresponding Model method there is ***HmiUi_AddDblEvtToQueue*** function to insert event into *User Queue*. In the meanwhile, in the main loop of running target, queue is polled to take events as soon as they are inserted. Similarly to other direction, received events are extracted with ***Hmi_GetUiNextEvtState*** and then processed with ***Hmi_ProcessEvent*** function. Event evaluation is done by replacing the double value received, into table, at corresponding column of row indexed by event identifier and then, by calling the instruction pointed, which can be implemented inside a C file generated with python script or directly in the main file of target.

3.1.3 How To Integrate the Interface in the IDE

In order to integrate the interface in a project, it is necessary to follow some steps:

1. First, in the project folder, create a directory in which all library files are copied, to avoid confusion with other files;

2. Then, the folder must be added as additional include directory, in the properties menu of current project. So, open the properties and

in *"C/C++ General"* category, select *"Path and Symbols"*;

3. on the right side of panel click on 'Add' and for **all languages** select the folder path of current workspace;

4. at this point, the interface is available for the project, it misses only to include the header file where it needs.

3.2 Interface work on PC Simulator

TouchGFX, as mentioned in previous chapter, has an important feature like PC Simulator which allows to evaluate behaviour, of implemented application, running it on your own PC. This functionality is improved with the Software Interface Layer, because through this, the simulator can interact with a Model-Based-Design tool to obtain and simulate the full behaviour of developed project. To establish a communication between two different processes, queues used for target operation are not sufficient. It needs an *inter-process* communication like UDP or TCP.

Inter-Process Communication Before proceeding, it is necessary to provide some knowledge on *end-to-end* connectivity, for more details it is suggested to read the presentation by Professor Panagiota Fatourou[12]. In this case, it is a private network between two different processes running on the same pc. To allow a correct communication data must be formatted, addressed, transmitted, routed and received at destination. The address format used is the IPv4, which means that each one is divided into four octets, first up to three can be used to identify the network, while the other octets are used to recognise the node on the network.
Most used protocols are Transmission Control Protocol(TCP) and User Datagram Protocol(UDP). For this interface, it has been chosen to utilize the TCP, because it has a reliable byte-stream channel which ensure that all packets arrive in order without duplicates, it has a flow control, is bidirectional and also connection-oriented. Hence, similar to UDP, TCP uses port numbers as communication endpoints and 16-bit unsigned integers to provide transport. TCP is preferred for its higher load capacity and it is more suitable for persistent connection.
Connection works between *Berkley Sockets* that is an abstraction through which application may send and receive data. Sockets are uniquely identified by address, end-to-end protocol and port number.

The simplest communication is with only two ends involved, usually, one(**Server**) passively waits for and responds to other side(**Client**) which, knowing address and port number of server socket, initiates communication.
For , Special functions are required to generate sockets and manage their interaction, therefore, for this purpose Winsock library must be linked to the project.

Communication Steps First of all, it is needed to create the server endpoint with *Socket* function which returns the generated socket identifier of family, type and protocol passed as parameters. Then, with *bind* instruction, port, which will be used by socket, is associated and reserved. After that, TCP server has to wait for a new socket connection, this is performed by *listen* function which is a **non-blocking** one, that means that returns immediately. Successively, server executes a blocking function necessary to accept a client connection. Then, on the other hand, with exactly the same instruction used for server, client also creates its socket and successively with *connect* function, a connection between the two sockets is established. With this, server procedure is unlocked and communication can start. Each socket can send or receive messages through *send* and *recv* instructions. These two functions are generally *blocking*, but they can be set as non-blocking by providing specific parameters to a particular function: *ioctlsocket*, this is particularly useful when transmission is inserted into an application which has to run continuously, exactly like a GUI. When connection is no longer needed, it is closed and socket deleted with *close* method.

3.2.1 Inter-Process Communication for Interface

Functions are implemented, in the interface to initialise sockets, for both sides of communication. The User Interface is intended as a server, while the client is the model created with a Model-Base-Design tool. The use

of TCP communication is not demanded to specific functions and it is totally transparent to developers. In this way, development approach can be the same whether it is programming on target or on PC simulator. This is possible with a directive already present inside TouchGFX simulator. The **SIMULATOR** directive gives the possibility to write code which will be used only in simulator mode, therefore, functions that must have a different behaviour depending on running type, have a double definition: that for simulator, specified by respective directive and the other for target. An example is reported into Appendix A, in order to clarify this concept. It shows implementation of target initialisation function which sets the queue in target mode only, while for simulator, it performs preliminary steps for TCP connection. Moreover, during implementation of Software Layer it is chosen to use another directive: **CLIENT** to allow a distinction among the two endpoints. This is particularly useful for table definition, because function pointers used on server are not present on client and vice versa.
TCP communication is used to transfer events from server to client and vice versa. For instance, when a widget is touched, into Model within corresponding method, as for target, the *HmiUi_AddDblEvtToQueue* function is called, but in this case event is pushed inside a TCP queue, used to accumulate events, avoiding losing them when event generation is faster than sending them to backend. Then, in the main loop of simulator, this queue is continuously checked and when at least one element is present, it is picked up, encoded and sent to the other socket. After that, on the other side, a message is received, decoded and put into respective queue. In the same way, on the opposite direction with *Hmi_SendUiStateXY* a message with occurred event is transmitted, then, received and processed by UI simulator.
In order to receive messages correctly, it is necessary to modify main loop, executed inside Simulator. To emulate target behaviour, simulator is implemented using *Simple DirectMedia Layer* library[13]. It is a Cross-Platform library designed to provide low level access to audio, keyboard, mouse, joystick and graphics hardware. This freely usable

library, written in C, also works with a variety of other programming languages and it is supported by many Operating Systems.
The SDL library supports the implementation of Hardware Abstraction Layer into simulator. In the **taskEntry** function, main loop is implemented. Called from GUI task, it will wait for VSYNC signal and then process next frame.
To support communication, in particular to create socket and exchange messages, previously named function must be modified adding some line to generate socket, establish a connection and to guarantee sending/receiving of messages. A final definition of *taskEntry* for TCP communication is given in appendix B.
At the beginning, there is **HmiUiTcp_Init** instruction, to initialise the Winsock library and to create Server endpoint and **HmiUi_QueueInit** to reset Ui queue. At the end, instead, there is **HmiUiTcp_SrvProcess** to send and receive events to/from backend system. Then, outside main loop, there is the **HmiTcp_CloseSocket** function to delete socket.
These three important public functions are composed of various internal private instructions, not accessible by developers but very important to allow the correct functioning of HMI.
On the other hand, client creation is quite similar, using the same functions. For instance, by creating a Simulink model, it is possible to use public functions available in the interface. Similar to what is done for server, *Hmi_Init* function is used to initialise the library and create client socket. Then, in an endless state, there is a loop for sending events and another for receiving them. As already mentioned, from the point of view of developers used code must be the same, in fact, as reported for interface running on target, the function *Hmi_GetUiEvtStatePending* receives event from network and push it into user queue returning the number of elements inserted into that queue. Then, with *Hmi_GetUiNextEvtState*, event is extracted from user queue and processed with *Hmi_ProcessEvent*. The flow of instructions is identical to that used for target.

3.2.2 How To Integrate the Interface for Simulation

To have interface functions available, for a Visual Studio solution, it is necessary to perform some preliminary steps:

1. open solution menu right-clicking on project name, select 'Add' and *'Existing element'*. A window opens, there, all files to be added must be selected;

2. enter their folder path as *'additional include directory'* in the project properties menu, under the 'General' tab of 'C/C++' category. Now, hmi functions will be usable, but a few more steps are required;

3. to ensure correct behaviour of functions it is very important to set some preprocessor directives, so, in the *'Preprocessor'* tab of 'C/C++' category insert the directive **SIMULATOR**, if it is implementing a client for TCP communication it is necessary to also add the **CLIENT** directive;

4. Finally, it is necessary to link the library needed for TCP communication, so in the *"Input"* tab of 'Linker' category, add the Winsock library (ws2_32.lib) as additional dependency.

3.3 Software Interface Layer Test

Since functions, needed to manage communication between the two system sides, are implemented for both modes of execution, it is time to test interface and analyse its performance.
First of all, tests are performed on target. Initially, normal behaviour is tested with a variety of projects, which have many widgets and a different number of screens. Queues are proven to work, as expected, when fully filled or not. Events sent, when queue is full, are discarded. When an event arrives on the UI and a widget is not visible, it is maintained original behaviour, that is to call, through Model, a ModelListener method not overloaded by actual presenter, so no modification is directly done. But anyway, it is possible to act on variables of model, for instance, it is possible to use a variable to store state of widget and when it returns visible it is read and widget will be shown in the new state.
If events with same identifier are sent, they are processed one by one, for both target and UI side, calling the appropriate callback each time, but not all modifications appear on GUI because screen is refreshed at regular intervals and events are usually evaluated rapidly.
Additionally, different queue sizes are checked and queues work as expected. The case in which UI and backend events are interleaved is also checked, resulting in all occurrences being managed: on target events are processed immediately when received, one at a time, while on the other side, processing starts after that flush is requested.
Same tests are performed also for Simulator proving that behaviour is equal to one described for target. In this case, is verified also, that TCP connection does not affect operation of the interface. Therefore, it is verified the case in which a side sends some events, while connection is ready and then it is disconnected, all events resulted correctly received and processed. This is because TCP line acts as a queue, that accumulates sent events until receiver extracts them with recv operation. It is also tested that events are not accumulated, even when connection between two sockets is not ready, so when only one socket is generated.

This with TCP is possible, but for this interface, it is chosen to manage all events occurred when connection is ready for both parts, this is achieved using a function to initialize tcp queue after that connection is open.

CHAPTER 4

Simulink Model

Before proceeding with a complete documentation of functions available in the interface, it is necessary to briefly describe some functionality available in a Model-Based-Design tool such as Simulink: *Stateflow* and the more general *System Function*.

4.1 Stateflow

Stateflow is a **MathWorks** software, which can support Matlab and Simulink to describe how algorithms and models react to input signals, events and time-based conditions.[14] With this tool it is possible to model decisional, combinatorial and sequential logic by simulating them as blocks in Simulink model or executed as Matlab objects. Graphic animation allows to analyse and debug the logic while it is running. Modification time and runtime controls ensure consistency and completeness of design prior to implementation.
Using the Hmi functions, it is possible to support a graphic simulator with this powerful software, thanks to TCP connection. Permitting to evaluate performance of sending/receiving events among hardware and software. In a Stateflow block, it is possible to implement Finite State Machine, Truth Table and other types of logic statement. For instance, it is possible to create a simple Simulink model, with a Stateflow block that, for some specific conditions, sends events to GUI and also can receive them from. When model is complete, with **Embedded Coder** is

possible to generate C code which emulates model behaviour. This code can be joined with GUI code, built and loaded on target. In this way, the whole project, composed by GUI, implemented in C++ and model written in C, can be easily compiled, loaded into a target hardware and run. This permits a complete prototyping without the need to have a board on which to test developed project.

Stateflow Client Stateflow permits to generate a simple client to send events, through the interface, to UI application. It does not need many states, essential ones are an initialisation state and another with Client processing. The second state is the core of program and can have two sub-states one for receiving events, usually in an endless loop and another to send events.
In order to integrate the Software Layer in the Simulink model it has to be properly configured. As first step, hmi files must be in the same folder of model and then, it is necessary to configure the *'Simulation Target'* category of *'Model Settings'*. Inserting, in the panel shown in figure 4.1, for custom C code:

1. source files, with declaration of global variables;
2. header files and external definition of global variables;
3. an initialisation function;
4. a terminate function.

While, as build information:

1. folder paths which contain useful files for the model;
2. source files of library;
3. necessary libraries, like Winsock for TCP communication and User32;
4. pre-processors directives, SIMULATOR and CLIENT to correctly configure tables.

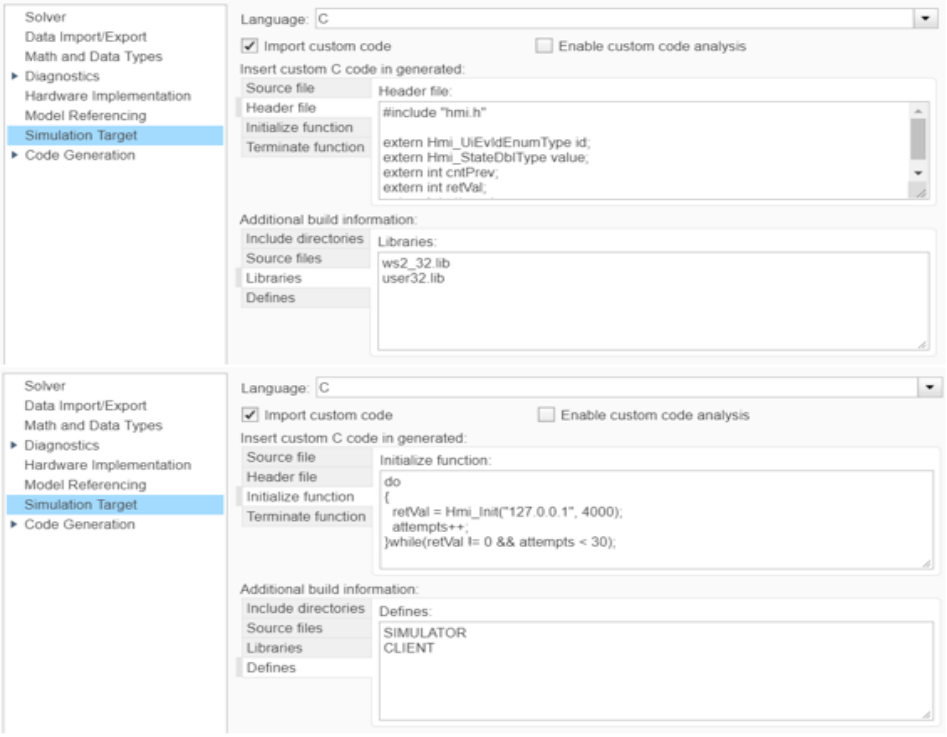

Figure 4.1: Model settings configuration.

After that, the language used from Stateflow block must be set in the *'Chart Properties'* selecting C code to use interface functions.4.2
When client is designed and its correctness verified, model can be built with Embedded Coder, which generates C code. Before building, it is necessary to set Simulink configuration for code generation in *Model Settings* panel. First of all, select the system target file and output language. Then, under code generation category, in "Custom Code" copy the information about header and source files, initialize and terminate functions, additional libraries and directories, or check the whitebox that imports this, automatically, from configuration of simulation target.
After that, code is ready to be built. Once code has been generated, behaviour of implemented system block is translated entirely into C language. At this point, it is possible to create a client with visual studio or another IDE and to use the three main functions made available in block header file. The *initialize* function is to set up the model if it needs, then, usually in an endless loop, variables, which correspond to

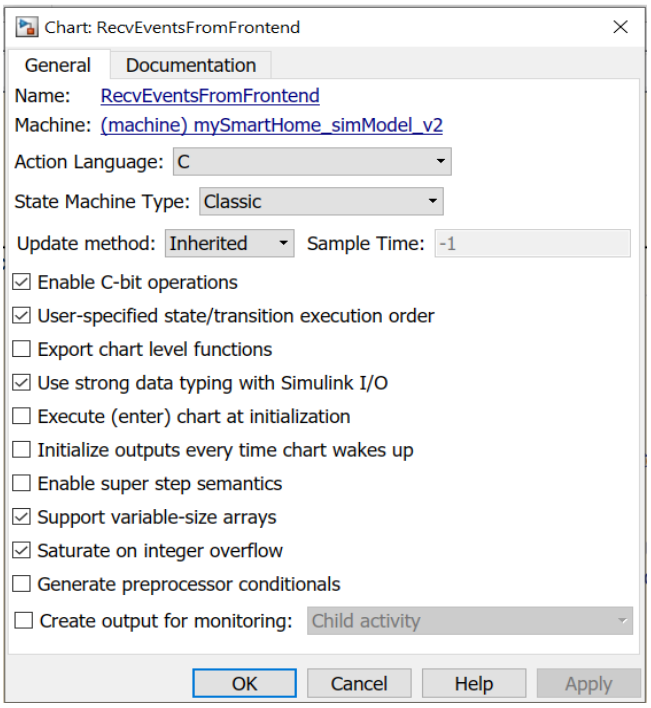

Figure 4.2: Chart properties configuration.

system inputs, are updated with new values and *step* function is called. This instruction simulates business logic computing the output results, which could be used to achieve a final goal. At the end, when model has to stop, thus *terminate* instruction is called, to deallocated resources or free memory space.

4.2 S-Function

After that, it has been chosen to test the HMI with a more generic Simulink block, the System Function(S-Function).
It extends the capabilities of Simulink Environment by describing the block with C, C++, Fortran or Matlab language. S-Functions are compiled as MEX files and dynamically linked subroutines that Matlab engine can automatically load and execute.
S-Function defines how the block work within each simulation phase. At each step corresponds a different function invoked by Simulink engine to fulfill a specific task. As previously seen for Stateflow, s-function can also be used to generate code through the Embedded Coder. Obtained C code has three main functions:

- *initialize* executed in the first phase of model execution, to perform primary operations needed by the block, like setting up user data or initialising vectors;
- *output* which calculates values to assign at each block output port;
- *terminate* used to free memory space when simulation is finished or when block is deleted from model;

Then, sometimes other two functions, *update and derivative* could be implemented if the block has continuous or discrete states.
S-Function has been integrated in a client and tested with an example GUI, then code is generated and tested on target hardware. Also in this case, full code is loadable on board and Software Layer works as expected.
There are a few steps to follow in order to use external code in an S-function builder block:

- first, on top, assign a name to the s-function and select the language to use;
- then, it is possible to insert, in the "Editor view", the inclusion of interface header file;

- after that, it is necessary to list ports and parameters, which block needs to function properly, in the homonymous tabs below;
- in the "Libraries" tab enter the names of source and library files, one per line, then also *SIMULATOR* and *CLIENT* directives;
- if files, inserted with the previous step, are in a different folder with respect to the Simulink model it is necessary to specify the path in the same tab.

PORTS AND PARAMETERS | LIBRARIES

Tag	Value
ENTRY	User32.Lib
ENTRY	WS2_32.Lib
ENTRY	hmi.c
ENTRY	hmi_table.c
ENTRY	mainClbk_list.c
ENTRY	SIMULATOR
ENTRY	CLIENT

Figure 4.3: Libraries configuration of s-function builder.

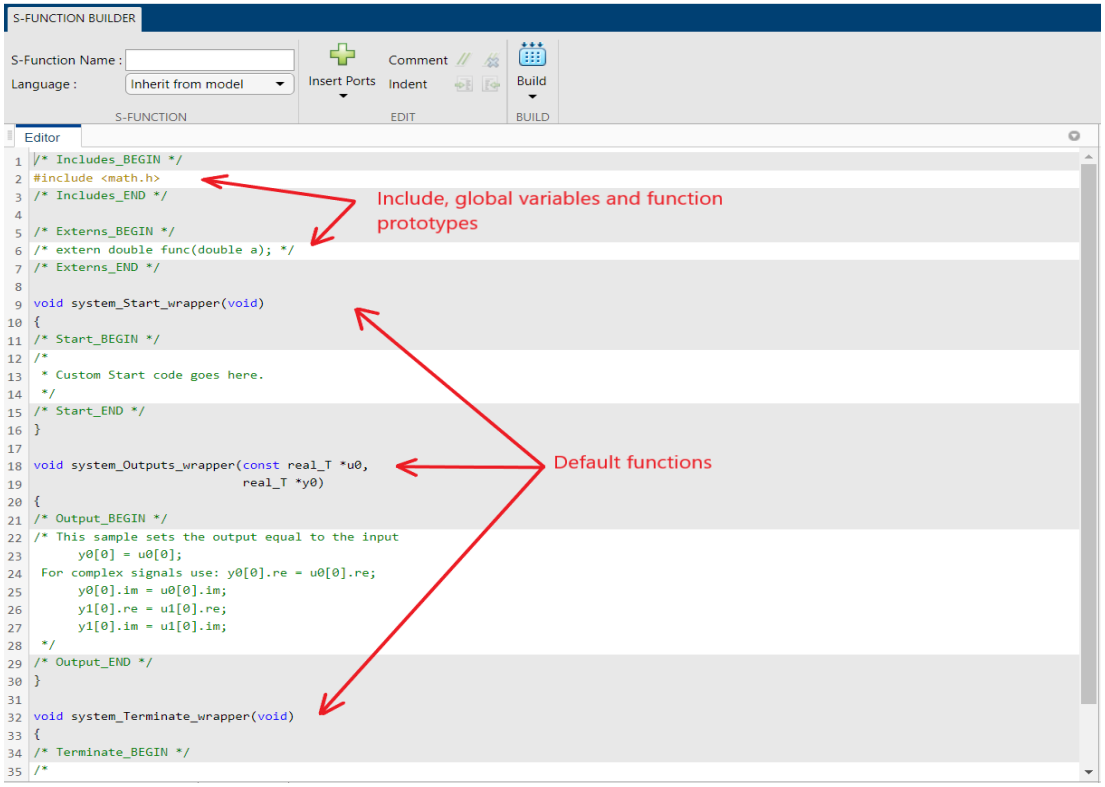

Figure 4.4: Editor view of s-function builder.

CHAPTER 5

Functions Documentation

In this chapter, it is explained the behaviour of functions implemented into Software Interface Layer, to better understand the idea behind and how to use them.
Many functions have a double implementation, this is to allow a different behaviour, depending on running mode. In this way developers can use the same instructions either on target and in simulation, without being aware of different instruction behaviour. In the interface it is possible to distinguish between functions behaviour with a preprocessor directive. In fact, developer, to ensure the correct approach, must define, on IDE, the preprocessor directive: ***SIMULATOR*** when using the interface in simulation or none when loading the project on target.
Also, another directive was used, for simulation, to distinguish table initialisation when the interface is working on client or server. The default behaviour is for the server, thus Ui table is initialised with all function pointers while the user's one is only declared. On the other hand, specifying the ***CLIENT*** directive, for instance on Simulink, User table is completely initialised, while the Ui table is only declared because the callbacks are not known.

5.1 Public General Functions

This section lists all functions available in the interface, to initialise queues or configure TCP connection, to close sockets or handle events on server. Each function suggests which side it should be called from. Instructions for User Interface side have HmiUi as prefix while other side and general functions have Hmi. Then, in some cases also Tcp is added, referring to connection functions.

5.1.1 HmiUi_QueueInit()

This function is used to initialise queue for events arriving from back-end. In fact, as prefix suggests, the function must be called on the User Interface side.

5.1.2 HmiUiTcp_Init(const unsigned long addr, const unsigned short port)

Following instruction, always for UI side, has a double implementation, as the example reported in appendix A. For *embedded* target it simply returns a zero while in simulator mode it aims to initialise Winsock library and to create Server socket. The function needs of two parameters: address and port of socket to generate. In *simulation*, it returns a "-1" when error occurs or "0" otherwise.

5.1.3 Hmi_Init(const char* addr, const unsigned short port)

In this case, it is used a single function to initialise queue, in both running mode and to create the endpoint for connection on backend side, suggested by Hmi prefix. It is possible to implement only one instruction because the two internal function are always used in the same file, while this does not happen for the Ui side. This function also accepts two parameters for address and port of client socket. It returns "-1" in case of error and "0" when nothing happens.

5.1.4 HmiUiTcp_Connect()

The Connect function is used by server to accept an incoming client connection. It returns a “0” when connection is correctly established or “-1” on error.

5.1.5 HmiTcp_CloseSocket()

As name suggests, this function is used to close connection socket. This is the only function where the prefix is not precisely referred to unique side. Closure of socket is identical for both parts, therefore, prefix is used in a more generic way. A “0” is returned when socket is closed, “-1” otherwise.

5.1.6 HmiUiTcp_SrvProcess()

This function is the core for sending and receiving messages, via TCP, to and from backend system. It should be placed in the main loop of server, to be frequently executed. It uses private functions to receive TCP messages, decode and push them into queue processed by UI. On the other direction, function takes events occurred on GUI, encode and send them through TCP socket. Towards backend there is a queue to store events if generation is faster than message sending.

5.2 Public Backend To Ui Functions

In following sections will be listed and described functions that allow backend system to send events and to user interface to take and process them.

5.2.1 Hmi_SendUiStateInt(const Hmi_UserEvIdEnumType evId, const Hmi_StateIntType value)

This instruction, as its name indicates, sends integer events to user interface. It is called by backend when a widget needs to be modified consequently to a specific event on target. The instruction has a different behaviour dependently on running mode. In embedded mode(on target) events are immediately pushed into queue directed to UI, while in simulator mode events are sent to UI via TCP messages. Function needs of two parameters: the event identifier, which is unique for each "object" that generates a trigger for a widget modification and, the value to modify corresponding UI object. If sending fails "-1" is returned, "0" otherwise.

5.2.2 Hmi_SendUiStateTxt(const Hmi_UserEvIdEnumType evId, const char* text)

Function Behaviour is the same of previous instruction for both modes of execution, but in this case modification value is a text string, to modify widgets that accept characters, like text areas. Again, "-1" is returned in case of error and "0" otherwise.

5.2.3 HmiUi_GetUserEvtStatePending()

It indicates whether there is at least one event inside UI queue. It returns the number of elements stored or "0" when queue is empty.

5.2.4 HmiUi_ProcessEvents()

This functions checks if there is at least one element into user interface queue, so it extracts and processes it, updating referenced value of table and calling the corresponding callback to taken event.

5.2.5 HmiUi_IsFlushRequested()

This statement controls if state variable, used to manage User Interface queue, has *FLUSH_STATE* value, in that case returns a "1", otherwise a "0".

5.2.6 Hmi_Flush()

Function behaviour depends, again, on running mode. If it is on target, it sets the state variable of UI queue to *FLUSH_STATE* to block the queue until flush is finished. Against in simulator mode, a special message with opcode "-1" is encoded and then sent to server to enable flush operation.

5.3 Public Ui To Backend Functions

In this section, there are listed all functions needed to transfer events occurred on User Interface toward backend system.

5.3.1 HmiUi_AddDblEvtToQueue(const Hmi_UiEvIdEnumType evId, const Hmi_StateDblType value)

As some other functions, it has a double behaviour: on target it pushes UI events into queue directed to hardware, if it is not full; on simulator, instead, pushes events to a TCP queue from which server will take elements, one by one, to successively send to client. This function needs of event identifier and modification value as parameters, returning "-1" on error and "0" otherwise.

5.3.2 Hmi_GetUiEvtStatePending()

Equally to similar function used for reverse direction, this instruction on target returns the number of items into queue directed toward backend, "0" if nothing is stored. On simulator, instead, receives events from TCP connection, if present, decodes and puts them into backend queue, returning the number of elements in that queue.

5.3.3 Hmi_GetUiNextEvtState(Hmi_UiEvIdEnumType *evId, Hmi_StateDblType *evValue)

This instruction, as suggested by name, takes the first occurred event from user's queue, saving identifier and value into variables passed by reference from user. It returns "0" if all is done correctly, "-1" when error occurs.

5.3.4 Hmi_GetEvtCnt(Hmi_UiEvIdEnumType evId)

Following instruction is used to take the event counter assigned to a determined event, whose identifier is given as parameter, to check whether it proceeds or is always the same.

5.3.5 Hmi_ProcessEvent(const Hmi_UiEvIdEnumType evId, const Hmi_StateDblType evValue)

This is the last function called by user to manage an occurred event, it processes the event, updating corresponding value on table and calling respective callback to make something from hardware point of view. It receives event identifier and modification value as parameters.

CHAPTER 6

Home Automation Example

Before proceeding with the explanation of the example project, it is essential to focus on the entire process of integrating and building an application using the Software Interface Layer. The figure 6.1 helps to understand the main steps of procedure.

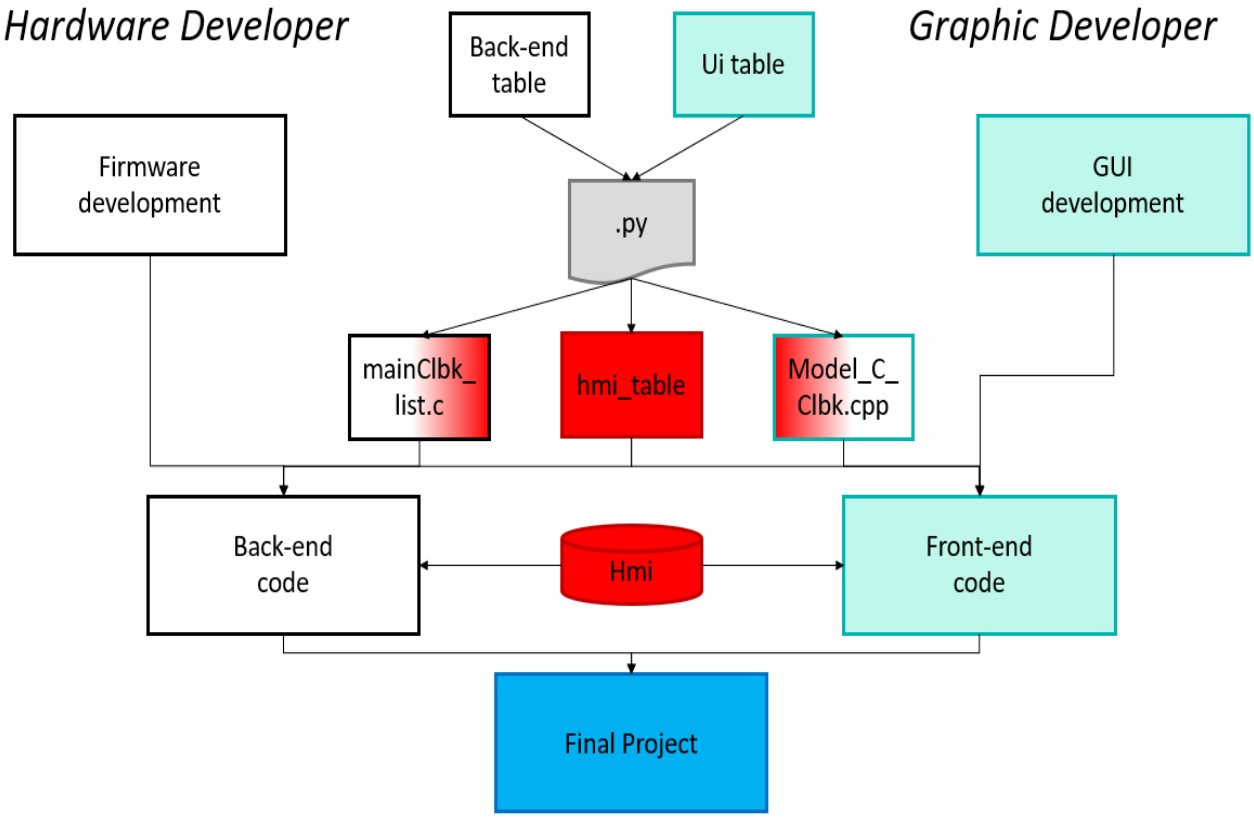

Figure 6.1: Main steps to implement a project with hmi.

First, it is necessary to create a new project in STM32CubeIDE by selecting a board which can be used to implement the final application. Then, generate the Board Initialisation Code to configure all necessary components. Successively, it is possible to integrate the interface library into the project and use it when programming the hardware and GUI parts. Of course, the two sides can be programmed at different times. After that events of a side have been outlined, the corresponding developer can fill in the related table. When both tables are ready, the Python script was run twice, one per table, automatically producing all the files needed to make the HMI working in the project. Only *"mainClbk_list.c"* and *"Model_C_Clbk.cpp"* should be completed with the callback implementation. At this point, the interface functions can be called correctly within the project code. Therefore, application can be completed. Then, when implementation is ended, the project can be tested by simulating its behaviour with a Simulink client, suitably created with the integration of the interface. When the application is consistent, the C code can be generated from the Simulink model and incorporated into the project. Finally, the project can be fully loaded on target board.

6.1 Home Automation Project

In this chapter, an example project is explained in detail to further clarify how data are transmitted from View to Model, in TouchGFX and how the interface works to manage interaction with hardware.
The example is a Home Automation GUI with a variety of screens to manage different rooms. In this application, there are four main areas that could be part of a house. *Living Room, Kitchen, Bedroom* and *Garden*, each occupying a screen of the application. Then, for living room, kitchen and bedroom, which have objects to set more precisely, there is a related screen for settings. When application is launched, home screen immediately appears, showing a summary of current status of main objects, which can be set for each room. It has four pages, one for each room and its widgets. From the Home Screen, with a simple tap, on the room box, it is possible to access the chambers. Areas can be scanned with a horizontal slide, while settings can be accessed, for rooms that have it, with a vertical slide or by touching gear icon on top right corner. When an object of a space is configured, it is possible to come back to the Home Screen simply by waiting for a few seconds or touching the Home icon. An image of living room implemented into application is reported below, in fig.6.2. Then, for each widget, that can trigger an event, an action is defined. For example, the shown room has six different types of widget that can trigger an action. The two icons, in top corners, are configured to change screen when touched, up and down arrows of shutter, implemented with repeat buttons, make an animation on corresponding image, lowering or raising the shutter; to turn on stereo and light are used toggle buttons. TouchGFX Designer gives the possibility to initially configure the action that a widget has to execute when it is touched, into "*interactions*" panel. If chosen action can be customized, such as "call new virtual function", the empty function is created into base class, when application code is generated. Hence, the function has to be inherited into user class and then defined by GUI developer. When button has to transfer a data to another screen or to backend system,

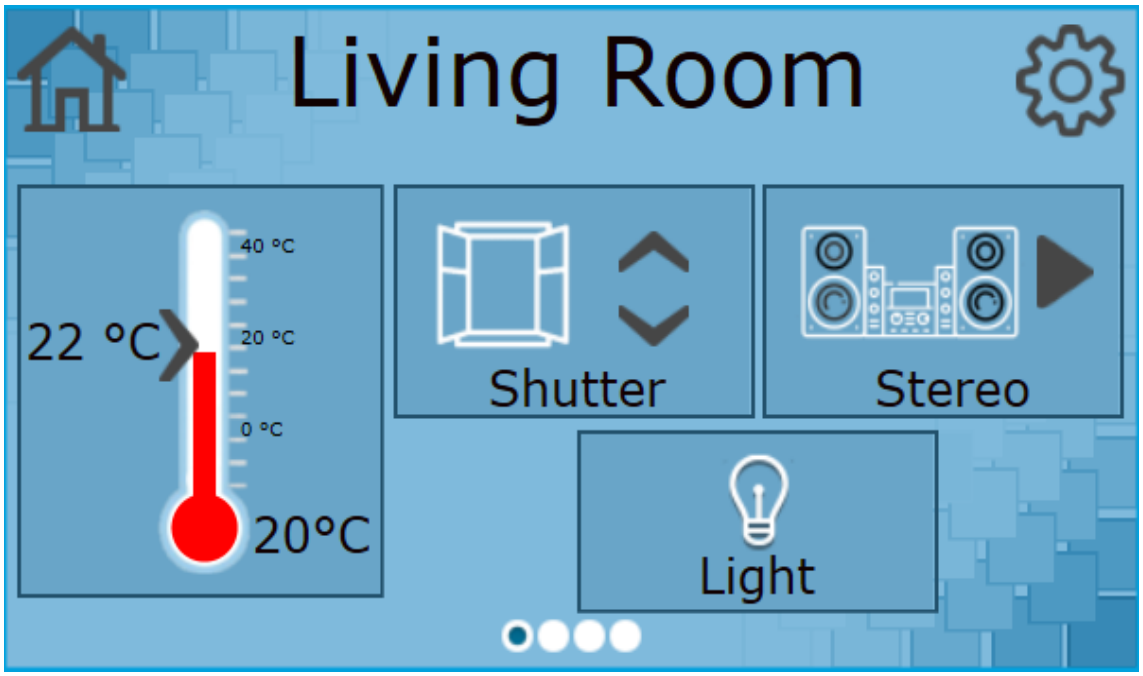

Figure 6.2: Screen implemented for Living Room.

the action triggered has to call a presenter function which user must implement. After that, the Presenter itself, calls a Model function, in which data can be stored. At this point, value may return to another screen View, calling methods of classes through pointers explained in chapter three, or it could be sent to hardware side. The Home Automation GUI makes many interactions between the two side of project, to best demonstrate how the Software Interface Layer works. Therefore, inside the Model method, called by Presenter, there is the HMI function to send an event to backend. Then, on target, events are taken and with respective C callback something happens. For instance, each light of different rooms turn on a specified LED on target hardware. Other functionalities that are not really implementable, like raising shutter, send a message, with corresponding received value, on the serial interface. To program properly peripherals behaviour it is necessary to use the board user manual, in this case that for STM32F429I-EVAL.[15]
For opposite direction, having only two physical buttons available, it has been chosen to use a button to send the flush and another to decide, with number of pressures, to which room events are sent. Therefore, occur-

rences are taken into Model, when flush is requested, and corresponding callback flow executed, until modification is set on the View.
In order to perform the correct call of Model method, inside the file *Model_C_Clbk.cpp*, automatically generated with a python script, functions, pointed by pointers stored into table, must be implemented. To make an example, in the Home Automation application, Model class implements a function which returns the instance created when application is started. In this way, it is possible to access from callback file, to the correct model instance and execute the right method. Function to get the model instance and an example of callback are reported into appendix B.2 and B.3.
After that, UI has been correctly implemented, it needs to fill UI table with all events identifier, accordingly to ones which will interact with surrounded system. For this example, each widget that has the possibility to set a real physical object sends an event to the target. When table is complete, it is supplied to the python script which generates files for the compiled version of table.

Python Script The script is organised to generate code for User and Ui tables in different moments, this is because the two sides of project might be developed by different people at different times. It can be executed from command line by entering as parameters file name of table and a "-c" or "-a" if compiled table has to be created from scratch or it is only needed to append information about the second part. After that, script must be run and files are generated, developers should only implement callbacks to link Model methods for the UI part or main functions for target side.
As explained in previous sections, the Interface Layer makes inter-process communication possible. Therefore, to demonstrate the use of interface through the implemented TCP communication, it is created a Simulink model to simulate management of received events on the hardware. Client is created with a subsystem, that internally has different Finite State Machines. Top level view of block is reported in figure 6.3.

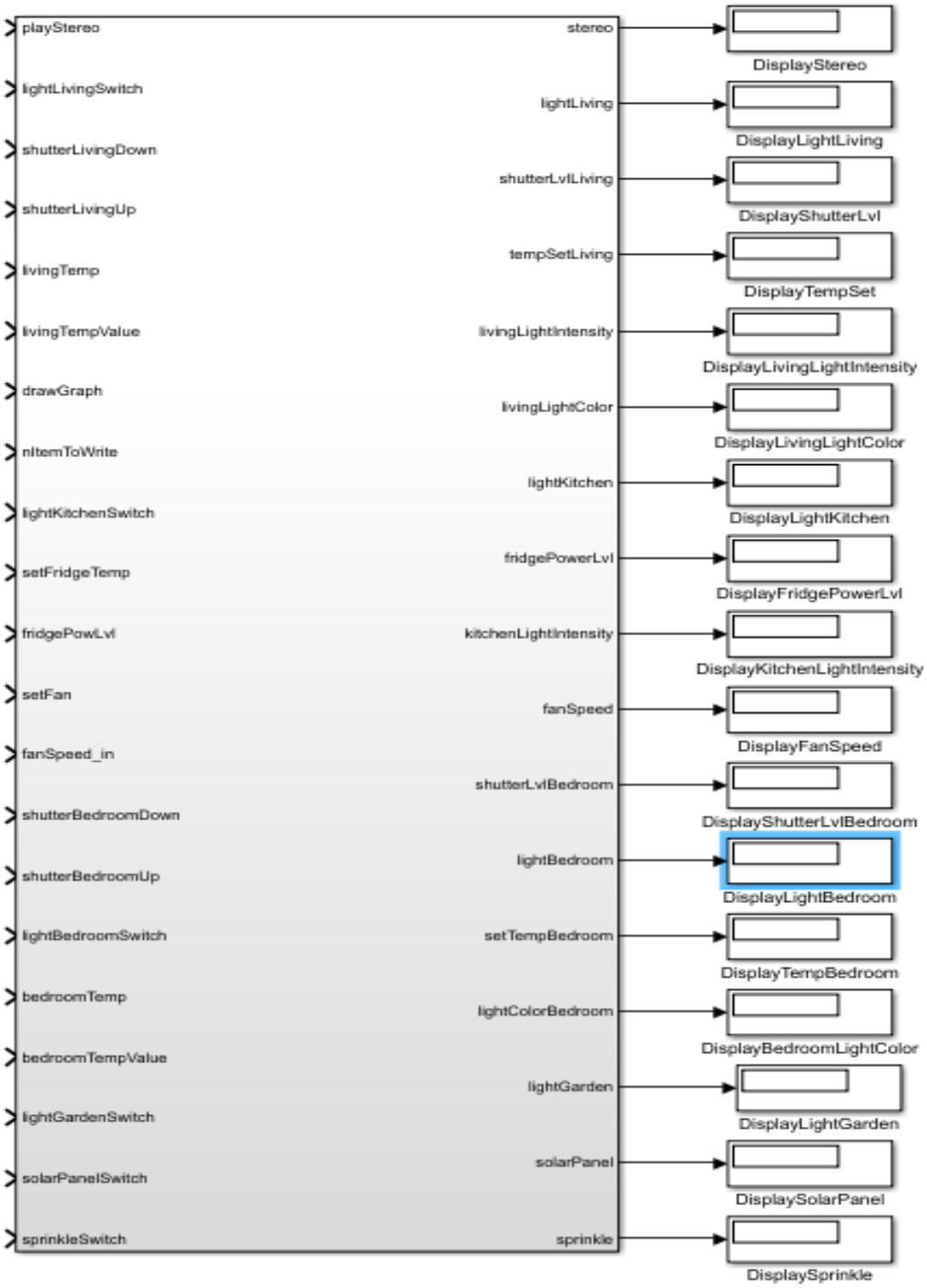

Figure 6.3: Top level view of Simulink subsystem.

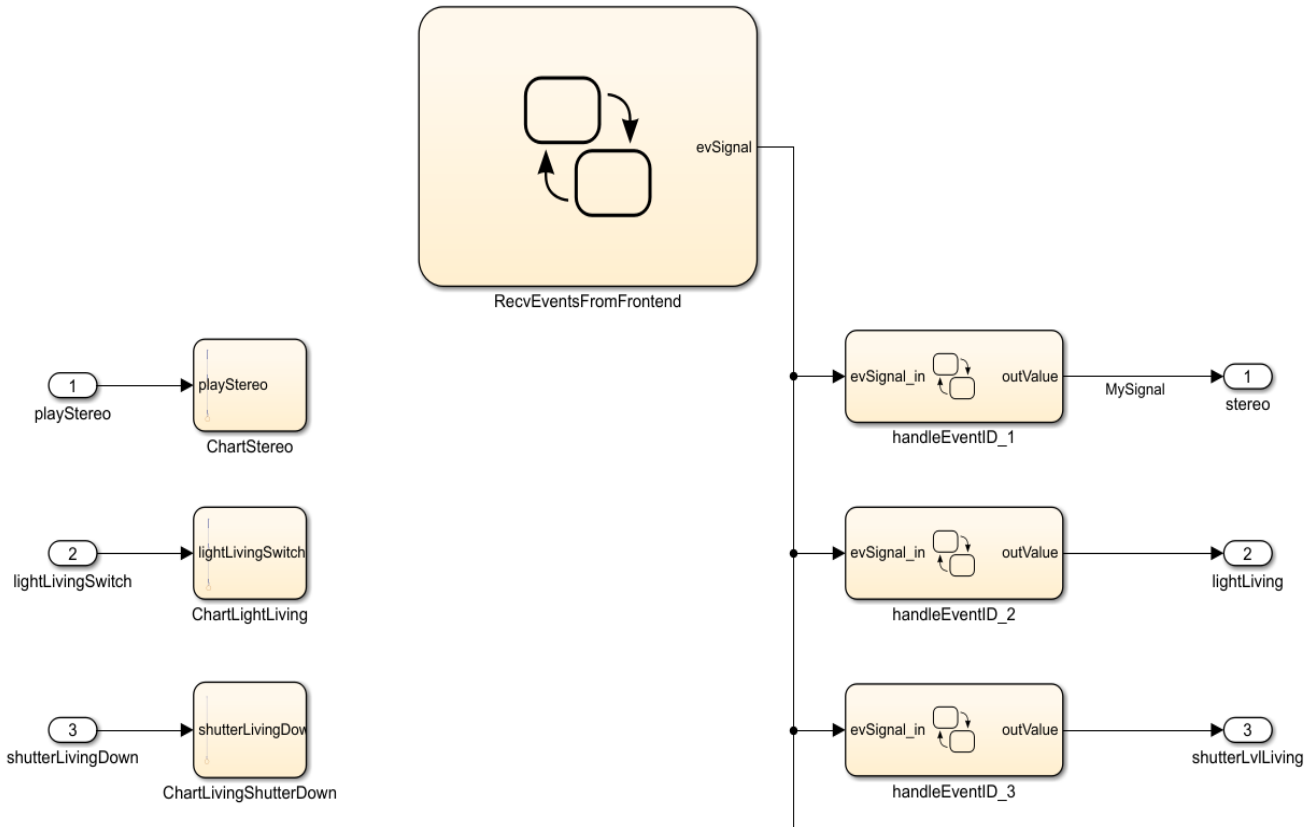

Figure 6.4: Internal structure of Simulink subsystem.

Internal structure is showed in the image above(fig.6.4), at the top there is a FSM to receive all states from the User Interface. This chart extracts all events occurred on UI from user queue, its structure is reported in fig.6.5.

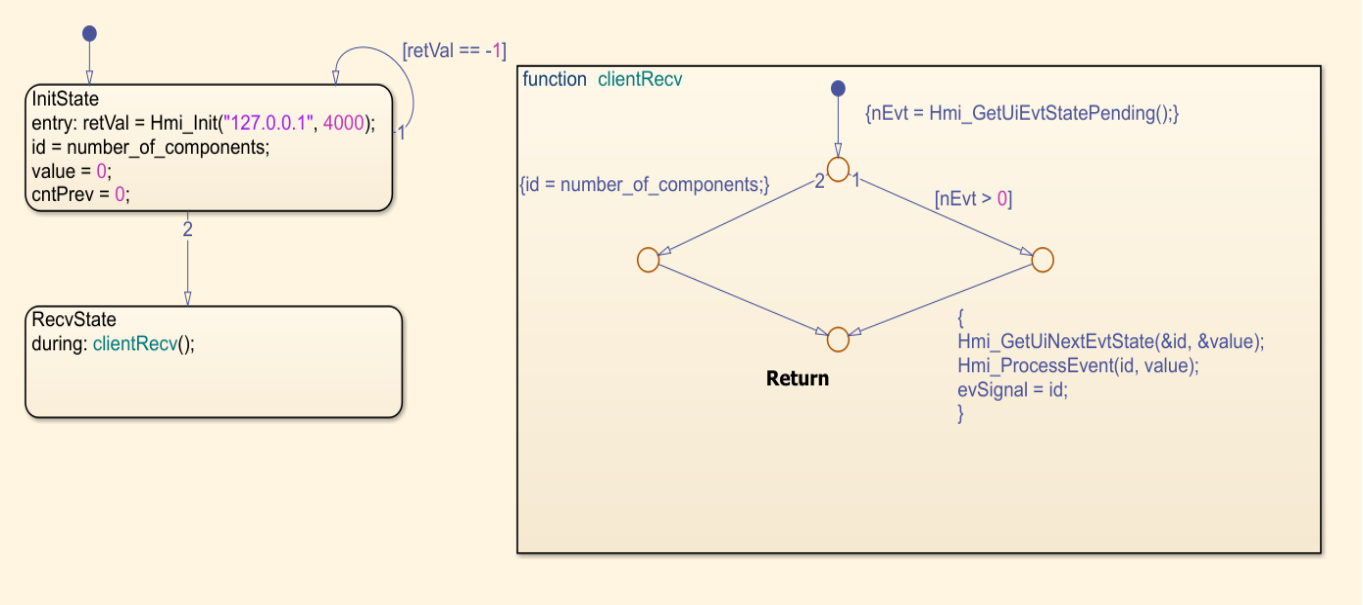

Figure 6.5: FSM that receives all events occurred on UI.

Then, under this chart, there are many other FSMs, one for each input and output, with different functions. Charts, on the left, manage sending of events simply by calling up respective HMI function, as reported in fig.6.6.

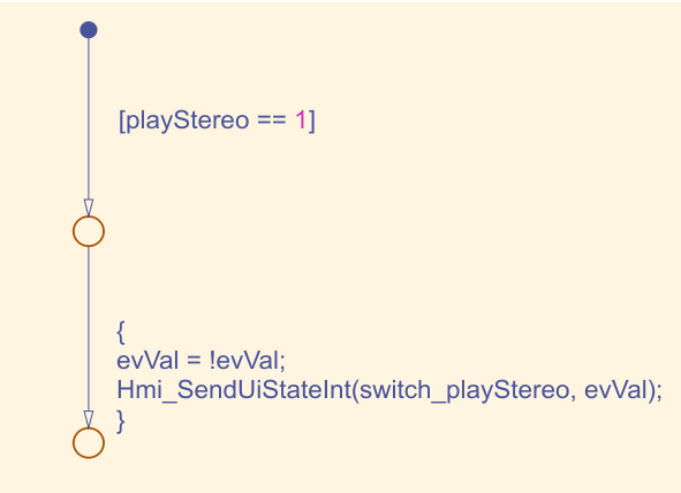

Figure 6.6: Send event state, for each input of subsystem.

Then, on the right, each output of system is set to received value provided by an FSM, when respective event is taken from queue. The whole logic is implemented as figure 6.7 shows.

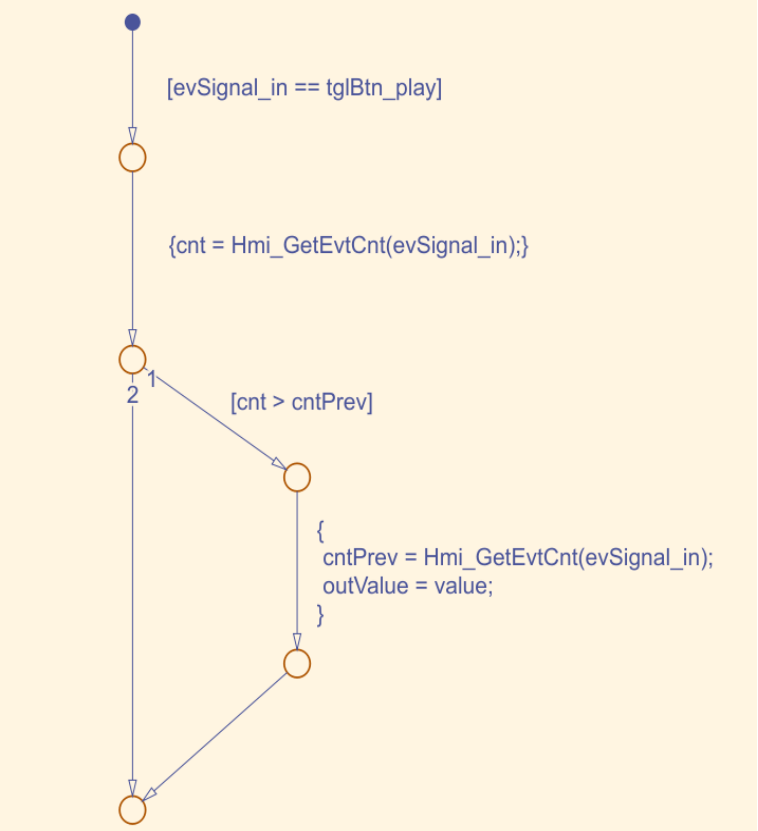

Figure 6.7: Flow of states to manage taken event.

After that, Client is ready to interact with Server to simulate project behaviour. It is verified that model runs correctly, without any error or crash, then, it can be built with Embedded Coder. In this way, corresponding C code is obtained and the whole application can be loaded on target board. When model code is generated, Simulink provides three main functions: *initialize* to configure the model, *step* which takes inputs of model and checks their values and, if necessary, processes them. Generally, this function is placed in a main loop and finally, *terminate* function is called to stop the execution of model.
These functions are called within main function, opened in an IDE, like for instance STM32CubeIDE, to reproduce model behaviour on target, after that memory board has been flashed. First of all, model must be initialised, then in an endless loop, inputs, exported from Simulink model with a particular data structure, are set. After that, calling step

function, model inputs are collected and business logic, described on Stateflow, is executed to obtain model outputs. In the example, when an input signal is asserted corresponding event is sent to UI and, after flushing, proper widget is modified. Events towards target are also managed.

CHAPTER 7

Interface Generalisation

When the interface is reliable enough, it is chosen to make it **general purpose**. Therefore, to prove that, interface is used with another graphical tool: Embedded Wizard. This software also has a *drag and drop* approach to implement the interface. It has, like TouchGFX, the possibility to simulate the behaviour of a screen or entire User Interface with the Prototyper. Therefore, it is perfect for demonstrating that the interface does not depend on the structure of TouchGFX.

7.1 Embedded Wizard

Embedded Wizard is a technology that enables to create platform-independent and high-performance Graphical User Interfaces (GUIs) for resource constrained embedded systems.
To develop the whole appearance of user interface and behaviour of a GUI, the user-friendly tool: **Embedded Wizard Studio** can be used. It allows to easily create GUI components from scratch or starting from some templates.
It has an important feature, the **Prototyper** that simulates GUI appearance and its behaviour at every step of design, without the need to configure target hardware. If written code does not behave as expected, the built-in *Debugger* helps to find the cause of problem. Once the development process is finished, Embedded Wizard Studio generates source

code, with ANSI-C language, optimized for chosen hardware platform. Depending on build environment, generated code is compiled into a binary file and then, executed on target system.
Embedded Wizard supports a wide range of target systems. Also, platforms with hardware graphic accelerator may be used. Generally, prerequisites to support certain targets are very limited. For the most common targets, build environments with a ready-to-use display adaptation are provided.
Embedded Wizard combines two approaches: composition the appearance of GUI application with a GUI builder tool and separately the code to implement the behaviour.
Embedded Wizard Studio allows to develop the GUI application with *drag and drop* approach, putting all components into *Composer* window. In that window, there are elements that are needed by user to implement the application, there is also a rectangle highlighted with a blue border, the *Canvas*, which represents the display of target. Outside of Canvas, *bricks* are inserted, representing non-graphical members of application. Each member, once selected, can be configured in a panel on the right: the *Inspector*.
This tool uses a special programming language **Chora**. It is universal, target system independent, permitting to implement complete functionality of GUI applications without being reliant on any further development tools.
Chora is largely based on C, C++ and Java programming languages. It is object oriented, like its counterparts, supports classes, simple inheritance and polymorphism. Open arrays and pointer arithmetic is completely absent in Chora. It supports the concept of signals, observers and notifications to better manage communication between GUI components.
The provided environment allows to develop a highly platform independent GUI application. This abstraction is possible with Chora language and *Platform Package* concept. The Platform Package consists of three main blocks: *Code Generator, Resource Converters, Graphics Engine* and *Runtime Environment* which guarantee that GUI can run on differ-

ent hardware. Code Generator takes care of the right translation from independent Chora code to valid code for target system. Actually, generated code can be in ANSI-C or JavaScript. Resource converters, instead, translate bitmaps and fonts used in the application to format and code valid for target system. Graphics Engine is a library to perform graphical operations, it supports alpha-blending and, additionally, specified colour or opacity gradient. With this library each graphic operation is delegated to hardware or graphics API available on target. Finally, Runtime Environment is a library that provides functions necessary to run applications implemented in the Chora programming language, it acts as an interface between generated code and surrounded system.
For more details relative to this software, visit the documentation.[16]

7.2 How to use Hmi for Target on EW

First of all, it is tried to use the interface layer for communication between GUI and target hardware. A simple application is implemented with Embedded Wizard Studio: it is composed of a screen, with a gauge, which receives input values to update its indicator. Then, when necessary, values shown by gauge can be saved pressing a *Save* button on the screen. When it is pressed, value is sent to the system and also four LEDs are turned on, accordingly to number of saved values, represented in binary format. Since code for GUI objects must be Chora, but the interface is implemented in C language, it is used the **native** statement. This keyword encapsulates a piece of code, which is reported into generated code, exactly as it is written in the code editor. In this way, Chora compiler does not check the correct syntax, but simply copies it in the proper location. Native statement is used only for target, in fact, it is not visible from prototyper and it is evidenced by a warning. To avoid this warning, native statement has to be preceded by *$if !$prototyper* directive. Therefore, to implement event management, a timer is instantiated, in the composer of main screen, to periodically call a function which checks if flush of UI queue is requested. During flushing, events are taken from queue and processed, calling, via function pointer a method implemented for the screen. In order to do this, *ewmain* file, used to implement a generic framework for running Embedded Wizard generated GUI applications on a dedicated target with or without the usage of an operating system, is modified. Specifically, it is added a function to return the instance of implemented screen, permitting to pointed function to call the correct method through that instance. After that, called method will modify the proper widget.
On the other direction, communication is simpler, it needs to associate a slot method to the save button, when it is pressed, instruction is called and in its definition, the HMI function is inserted to send event, always in native statement. Then, when code is generated, the whole project might be opened in STM32CubeIDE and in GUI task, of main file, func-

tion to check if at least one element is into user queue, is used. If an event arrives, it is picked up and processed, exactly as it was for TouchGFX.

7.3 How to use Hmi for Prototyper on EW

Embedded Wizard and in particular its Chora code, are platform independent, but this also impacts on prototyping environment. While this can be easily overcame on target, for prototyper it is not. Main problems to achieve to use Hmi on prototyper is to use the Software Layer, implemented in C language, into code editor of Embedded Wizard, which accepts only Chora language. Native statement has also been discarded, because it is not visible from prototyper, only the **intrinsic module**[17] remains to extend EW by that specific functionality. From technical point of view, intrinsic module is an ordinary Dynamic Loadable Library containing C functions of the Software Layer. EW recognizes and loads this module automatically and then, functions can be used like any other built-in function of Chora. These functions are called *intrinsics*.
DLL is written using Microsoft Visual Studio C++ 2019 and following the example shown in Embedded Wizard documentation. Main steps are reported for completeness.
Initially, a project based on available DLL template is created. Then, access to properties menu of project to adjust few parameters. Into page *General* change target extension to ***.ewi**, so that resulting DLL with that extension is obtained, to be recognised as a valid intrinsic module by Embedded Wizard. Then, add, to include directories, all paths of necessary folders, where files to include are stored. At this point, configuration is complete and header and source files of Hmi can be added to the DLL project. To implement the library, interface files must be modified to adapt them to the new software. First of all, it is included the file with functionality common to all intrinsics: *ewrte.h/.c* located into Chora/Sdk just below of EW installation directory. Then, data types of function prototypes declared into Hmi must be translated to data types available in Chora. Body function can remain the same as it has to run on any other system. In other particular cases, like memory management and use of strings, Runtime Environment function, provided into headers previously included, must be used.

After that, in order to be recognised as a valid intrinsic module it has to implement *EW_MODULE(...)* section reported into appendix C.1. Then, Hmi functions have to be exported, so they can be seen from the Embedded Wizard prototyping environment. For this purpose, it is necessary to enhance implementation of the intrinsics module by an *EW_DEFINE_INTRINSICS...EW_END_OF_INTRINSICS* table. This table references all affected C functions and provides information about their parameters and return values. Knowing this, Embedded Wizard can find and invoke functions. For each exported function an *EW_INTRINSIC* section is needed and each one expects six parameters:

1. literal string with name of intrinsic function, which will be called inside Chora editor;
2. literal string with Chora data types of value returned by function;
3. an integer number indicating parameters expected from intrinsic;
4. literal string containing a comma separated list of Chora data types matching parameters of respective C function;
5. literal string with comma separated list of parameter names according to declaration of C function;
6. C function associated to intrinsic.

An example of exported function with that statement is reported into appendix C.2.
Finally, project can be built and the DLL with *.ewi extension can be copied into EW project folder. When project is opened a message asks to load the intrinsic module.
For the example mentioned before, Hmi is used for prototyper only to manage communication from Ui to backend, testing it through a simple Simulink model. Designed model receives event occurred whenever button, to save speed value, is pressed. For the opposite direction, Hmi is

intended to send speed values to UI gauge, but in order to do this, callback pointed by table must call a method within the instance of screen. Accessing that instance is not that easy for prototyper. Initially, it is tried to get instance as it was for target, but in this case calling the wrapper method, available into generated code, to modify proper widget. At the end, it has been chosen to use "brute force" by implementing a new function *HmiUi_GetEvents*, which returns event identifier and integer value from queue items, one by one, while method call is demanded to user implementation. Therefore, graphic developer has to recognise which event is extracted and then call the proper method.

7.4 MinGW

Another generalisation is based on compiler. In fact, the entire Software Interface Layer can be built with the free MinGW compiler. This feature is tested, for example, with **CLion** of *JetBrains*. It is tried creating a new project either in C99 and C11 languages. It generates a warning, where *strcpy_s* or *sprintf_s*, security enhanced version of homonymous functions without suffix "_s", are used, but it works correctly. This important feature opens to a large variety of IDEs because GCC compiler is free on the web.

CHAPTER 8

Further Improvements

After that, idea and structure of Software Interface Layer have been described and understood, some space is left for possible improvements, which can upgrade performance of implemented interface.
A possible enhancement to speed up the flush, for instance, is to use two different queues. While one is flushed the other is being pushed by new events. Another approach could be to save index position, when flush is requested and perform event extraction up to that position, in the meanwhile, new occurred events are inserted into next array locations.
Another improvement is oriented to use different encodings. Through a *typedef* it is possible to set a custom type for char data, in order to manage a Unicode coding, for instance.
Hence, the interface can be further generalised by testing it on a wide variety of graphics software, but also on different hardware board families.
Another improvement is, of course, working on Embedded Wizard, to have a more general Prototyper behaviour and similar to the approach used for TouchGFX. Understanding how to modify the DLL allowing to call User Interface methods to update visible widgets.

Bibliography

[1] **TouchGFX Documentation**, rev. Dec. 2020, *Installation*
https://support.touchgfx.com/docs/introduction/installation

[2] **TouchGFX Documentation**, *Embedded Graphics*
https://support.touchgfx.com/docs/basic-concepts/embedded-graphics

[3] **embryonic.dk**, Sept. 2019, *Configuring QSPI for TouchGFX and CubeIDE on the STM32F746G-DISCO board*
https://www.youtube.com/watch?v=237lPdMsDZs&feature=youtu.be

[4] **TouchGFX Documentation**, rev. Dec. 2020, *Framebuffer*
https://support.touchgfx.com/docs/basic-concepts/framebuffer

[5] **ST Official Website**, *STM32F4 example package*
https://www.st.com/en/embedded-software/stm32cubef4.html

[6] **TouchGFX Documentation**, *TouchGFX AL Development*
https://support.touchgfx.com/docs/development/touchgfx-hal-development/touchgfx-al-development-introduction

[7] **TouchGFX Documentation**, *UI Development, UI Components*
https://support.touchgfx.com/docs/development/ui-development/ui-development-introduction

[8] **TouchGFX Documentation**, *TouchGFX Engine Features, Languages and Characters*
https://support.touchgfx.com/docs/development/ui-development/touchgfx-engine-features/languages-and-characters

[9] **TouchGFX Documentation**, *Working with TouchGFX, Simulator*
https://support.touchgfx.com/docs/development/ui-development/working-with-touchgfx/simulator

[10] **EE by Karl**, Jan. 2020, *STM32CubeIDE 1.2.1 and TouchGFX 4.13.0 with STM32F746G-DISC0 kit*
https://www.youtube.com/watch?v=12KXreXaLp0&t=613s

[11] **TouchGFX, STMicroelectronics**, July 2018, *Hardware integration on STM32F769 with TouchGFX - Webinar*
https://www.youtube.com/watch?v=jQO7zhX0e0Q&t=557s

[12] **Prof. Fatourou Panagiota**, May 2012, *Introduction to Sockets Programming in C using TCP/IP*
https://www.csd.uoc.gr/ hy556/material/tutorials/cs556-3rd-tutorial.pdf

[13] **SDL Library Website**
https://www.libsdl.org/

[14] **Mathworks Official Website**, *Stateflow*
https://it.mathworks.com/products/stateflow.html

[15] **STM32F429I-EVAL User Manual**, March 2015
https://www.st.com/resource/en/user_manual/dm00093451-stm32429ieval-evaluation-board-for-the-stm32f429-line-stmicroelectronics.pdf

[16] **Embedded Wizard Documentation**, rev. 10.0, *Embedded Wizard*
https://doc.embedded-wizard.de/

[17] **Embedded Wizard Documentation**, rev. 10.0, *Platform Integration Aspects, Implementing Prototyper intrinsics* https://doc.embedded-wizard.de/implement-intrinsics?v=10.00

APPENDIX A

HMI Function with double definition

```
/*
 * Initialises backend.
 */
int Hmi_Init(const char* addr, const unsigned short port)
{
    #ifndef SIMULATOR
        _lock_Hmi_QueueInit();

        return 0;
    #else
        int retVal = 0;
        retVal = _lock_Tcp_CommInit();
        if (port != 0)
        {
            retVal = _lock_Tcp_ClientCreateSocket(addr, port);
        }
        else
        {
            retVal = -1;
        }
        _lock_Hmi_QueueInit();

        return retVal;
    #endif
}
```

APPENDIX B

TouchGFX Code Modifications

B.1 HALSDL2: Main Loop modification

```
void HALSDL2::taskEntry()
{
    uint32_t lastTick = SDL_GetTicks();
    SDL_AddTimer(1, myTimerCallback2, 0); // Start timer

    SDL_Event event;
    bool client_connected = 0;

    // Initialisation winsock library and creation of socket server
    if(HmiUiTcp_Init(SERVER_ADDR, SERVER_PORT) == -1)
        exit(EXIT_FAILURE);

    // Initialisation of Ui -> User queue
    HmiUi_QueueInit();

    while (SDL_WaitEvent(&event) && isAlive)
    {
        switch (event.type)
        {
        case SDL_USEREVENT:
            {
                uint32_t thisTick = SDL_GetTicks();
                  [...]
                  // Check if a client is available for connection
                if (client_connected == 0 && HmiUiTcp_Connect() == 0)
                {
                    // To accept only one client
```

```
                    client_connected = 1;
                }
                break;
            }
        [...]
        }

        if (client_connected == 1)
        {
            client_connected = HmiUiTcp_SrvProcess();
        }
    }

    HmiTcp_CloseSocket();
}
```

B.2 Model getInstance() function

```
// Function to return the instance of Model Class that calls it
static Model* getInstance()
{
        return instance;
}
```

B.3 Model Callback example

```
// Function to turn on/off the stereo
extern "C" void clbk_c_toggleStereo(void *value)
{
        return Model::getInstance()->toggleStereo(*((int*)value));
}
```

APPENDIX C

Embedded Wizard DLL

C.1 EW Intrinsic Module Validation

```
EW_MODULE
(
        INTRINSICS_IFC_VERSION,
        L"IntrinsicsModule",
        L"This is an intrinsic module to include Software Interface Layer into EW
)
```

C.2 Code to Export C Function

```
EW_DEFINE_INTRINSICS
    EW_INTRINSIC
    (
        L"IntrinsicHmiUiTcp_Init",
        L"int32",
        2,
        L"handle,uint16",
        L"addr,port",
        HmiUiTcp_Init
    )

    [...]

EW_END_OF_INTRINSICS
```